DIVISION OF COMPUTER SCIENCE

A Survey of the Current State of Reuse in a Software Environment

Technical Report No.146

A. Mayes

September 1992

A survey of the current state of reuse in a software environment.

Audrey Mayes

Sept 92

1 Introduction

The term 'reuse' when used in a software environment covers a very large area. This document contains a survey of some aspects of reuse. The main aims are to define the different aspects of reuse, the problems associated with reuse and current attempts at promoting various forms of reuse. The final aim is to identify possible areas in which to focus my research into software and traditional engineering.

The main reasons for increasing reuse are to improve the reliability of products and to reduce the cost of production.

Various aspects of reuse are presented in this document. The first section gives an indication of the range of ways in which the word reuse can be interpreted. This is followed by a section examining the reuse potential of the various stages of the development cycle. The subsequent sections cover requirements for code and design reuse, the problems limiting reuse and some suggested approaches to reuse. The final section identifies a direction for future research.

2 Meaning of reuse.

There is some debate about what exactly constitutes reuse. In some cases it is taken to mean the extension of an existing system to add new functionality, that is the code is reused in successive releases of a product. This type of reuse occurs in many systems. An alternative meaning is using some part of one system in a different system. Wirfs-Brock et al. use the following definitions [1].

"Software is reused when it is used as part of software other than that for which it was initially designed. Software is refined when it is used as the basis for the definition of other software."

If the meaning of reuse is restricted to mean reuse without any change, the scope for reuse will be restricted but reliability should be enhanced. If, on the

other hand, the scope is extended to include refinement in the above definition, then the increase in reliability is likely to be compromised but reusability will be extended.

The above definitions are concerned with code reuse. Rubin [2] extends the potential for reuse to include analyses, designs, implementation, test cases, and documentation. A similar broad defininition of reuse is used by the editors of Software Reusability [3]. The definition can be further extended to include the reuse of personnel and processes [4] , [5].

3 Reuse potential of the different stages of the development cycle

Meyer [5] and Sommerville [6] identify source code reuse as a possible area for reuse, pointing out that all software is finally and fully defined by its source code. However, there are several reasons why source code is unlikely to be the most beneficial level of reuse. A major reason is that the source code can be easily modified by the developer of the new system. Any changes mean that all the verification, validation and documenting of the software must be repeated even though most of the code is unchanged. This negates much of the potential benefit. This type of reuse is sometimes known as white-box reuse or cloning. A result of cloning is that the code becomes very difficult to maintain and understand. This is because lack control over the software can easily lead to a loss in structure and an exponential growth in the lines of code. The large volume of code and lack of structure makes the task of anticipating the effects of changes very difficult.

A second factor which reduces the potential benefit is that the source code contains implementation details and restrictions which may not be applicable to the new system [7].

A more promising area for increasing reuse is at the design stage of system development. As Meyer [5] points out, this requires that the design documentation of a system accurately reflects the actual system. Otherwise, uncorrected design errors may be propagated to subsequent systems. There are already reusable designs for operating systems and compilers [8]. These are two areas which have well defined functions and common patterns of behaviour which can be captured in the designs. Another area of well defined functionality is numerical computation [3]. This is also a fairly static and well understood discipline. A library of this type of reusable components is likely to remain static which enhances the possibility of reuse. A well understood mature environment may be necessary for the production of highly reusable designs or components. It is stated by Lenz [9] that common abstractions and concepts become apparent when an application domain matures. The very high apparent productivity rates of some software houses is suggested to be due to the reuse of entire designs

and requirements [10].

Lamb [8] states that it is also possible to use the same modular decomposition for new versions of a system, even if the code is completely changed. Specifications of the modules can also be reused for different implementations or languages.

A major factor in favour of reusing designs is that they are free of implementation details and the restrictions imposed by the language and hardware used. Harandi and Young [7] have developed a method using task-oriented abstract algorithms. These are similar to abstract functions which can then be applied to any data structure. The use of parametric programming is said to provide some of the required ideas but to allow only forseen modifications. The designs must be adaptable in unanticipated ways.

The method uses both a library of abstract algorithms and a collection of generic abstract data structures. The design and implementation process involves choosing an appropriate abstract algorithm which is then specialised for the required generic data structure. Each part of the abstract algorithm is repeatedly refined to give an implementation. The resulting programs are claimed to be easier to understand because they have standardised plans and data structures. The algorithm has a layered structure.

One example of an abstract algorithm is a search routine for finding a specific item from a data structure.The abstract algorithm definition contains details of what procedures must be provided by the data structure chosen, a list of possible implementation decisions, formal descriptions of each property of the operations and a list of impossible operations.

Domain analysis is identified by Horowitz [3] as a possible area for reuse. Domain analysis involves a thorough study of the application domain. This will lead to the identification of all the major abstractions, such as data types and processes, required by systems in that domain. Any major constraints on the interactions between data types and processes will also be identified. The analysis results could be reused for any system within that domain. Compiler writing is said to be an area in which domain analysis has already been used.

The most common forms of reuse at the moment are concerned with personnel [5] and processes [4] such as design inspections. Personnel reuse involves using the knowledge and experience gained during the development of one system to help in the development of the next system. The reuse of personnel is most effective if the knowledge gained can be passed on in some way, such as the development of the methods used for the production of models for estimating costs and time scales. Basili [4] also identifies the possible reuse of products, such as Ada generic packages. Customisation of an existing software package is another form of reuse, (Horowitz in [3]). It is likely that the most cost effective reuse will be made by reusing products from early in the life cycle.

4 Requirements for code and design reuse.

There are several generally accepted requirements for the reuse of software components [8], [5], [11], [6]. These are:-

1. some form of information hiding. This means that the user of a piece of code does not need to know how the code works in order to use it. The users of the components can treat them as black boxes. The separation of specification and implementaion allows many implementations to be produced for one specification.
2. these components should have a well defined purpose with a well documented interface.
3. the library of these components must allow the easy identification and retrieval of required components, that is some form of classification system must be developed. It is suggested by Rubin [2] that the contents of the library should be specially developed for inclusion in the library rather than be produced as a by-product of a specific project. Meyer suggests that all software should be with reuse in mind.

Other researchers have identified further requirements for the reuse of software components [12], [13], [7], [4], [14]. Some suggestions are:-

1. good support must be provided to allow library components to be used outside the originating division.
2. a framework or 'megamodule' to define standards or policies to allow individual modules to work together.
3. a standard set of abstractions for (re)use in software designs, such as the algorithms used for searches. This would make programs easier to understand and reuse.
4. a systematic approach for reusing existing experience .
5. tools and processes must be developed to support finding, understanding, modifying, and composing components.

5 Existing Approaches to reuse.

The majority of reuse at present appears to involve code reuse and method or process reuse.

There are two main current approaches to code reuse.

1. Code libraries. These are libraries of routines, such as mathematical, scientific and statistical subroutines, which are generally very small in nature and cannot apply to complex data structures [5],[8]. Biggerstaff envisages a set of routines in a run-time library which are designed to be used in more than one system [3].

2. The adaptation of one system to another with a very similar set of functions [3]. This can also be extended to include the maintenance of programs which is largely reusing one system to develop another with minor changes.

It is stated [3] that the reuse of components stored in code libraries has not been very productive. Their experience was that a system developed using more than a small proportion of reusable components was unlikely to meet its specification.

Methods or processes, such as design methods eg. SSADM and JSD and inspections, are widely reused.

A reusable processor was developed (ref'd in Horowitz [3]) for a single sophisticated specification language. The language used by the processor is called MODEL, from module description language, and requires two types of input. These are the description of the data and the relationships between variables. The input is checked for incompleteness, ambiguity and inconsistency. A fully functional program is produced in the language PL/1. This appears to be a program generator working from the formalisation of object oriented analysis results or entity relationships.

Methods for reusing transformation systems have been investigated (ref'd in Horowitz [3]). Transformation systems take high level specifications and convert them into efficient programs. The specification of the systems was said to be more complex than the final code.

6 Problems of reuse

The main problems limiting reuse of code and design can be divided into two main areas. These are technical and non-technical [6],[15],[2], [3].

The technical problems concern all aspects of the components. It is difficult to assess the reusability of a component because it is not known which of the component attributes are critical in aiding reuse. This difficulty in identifying critical attributes leads on to problems in defining a formal representation of components suitable for use in designing a storage and retrieval scheme for a library. The components could be in a programming language or in a program design language. The classification scheme must allow for both these forms. The specification of a retrieved component must be easy to understand.

In systems where speed of execution is critical, code reuse may not be seen as applicable. This is because the more general a product, the less likely it is to be the most efficient solution to a specific problem [2]. The analysis and design may still be reusable.

The reuse potential of any product will decrease as it becomes domain specific and forms a larger unit. The cost savings gained by the reuse of larger units will be greater than the reuse of smaller, more general products, such as lists and windowing systems [3].

Non technical issues include the costs involved. Generalised components cost more to develop in the first instance. This is because the component must supply all the functions which may be required in the future and the component must be designed in a general way. Reuse is a possible long term benefit but incurs an increase in cost in the short term. A lack of trust in the work of others is another obstacle to reuse. Productivity of staff is sometimes measured by the lines of code produced. Reuse will reduce productivity measured in this way and it is seen as necessary for a company to give some reward for reuse.

The above problems all relate to reuse without any change in the component. The concept of inheritance [5] allows one piece of software to be used as the basis for another piece of software. The original class is not affected by this change and the code is not edited. Additional fields and functions can be added. Inappropriate methods can be over-ridden. This is sometimes called black box reuse and allows changes to be made without encountering the problems associated with white box reuse.(see section 3). This enables the extension of the definition of reuse to include refinement,(see section 2). Jaime Nino [11] shows that inheritance, the ability to define new software modules based on one or more existing classes, without affecting the original class(es), allows small incremental changes to be made in the software and results in the formation of class hierarchies. This incremental development helps in the production of reusable software by allowing unanticipated changes to be made in classes and systems. This ability is said to lead to greater flexibility and reusability than is possible by the use of abstract data types in object based languages. It is conceded that inheritance can break the encapsulation of the parent class by allowing the descendant class direct access to its data structure. It is this access which compromises the reliability of the system. Nino suggests that this problem could be overcome by allowing only the functions to be inherited, forcing the descendant classes to use these functions to access the data structure of the parent class and therefore retain the reliability.

Problems have been encountered when using class hierarchies [16]. These problems resulted from the fact that code relating to decisions concerning error handling and validation of parameters was distributed throughout the hierarchy. A decision to validate all the required parameters together rather than one at a time, was shown to mean that apparently useful classes required major reworking and recoding before they could be used. This involved the breaking of encapsulation. These problems show that the design choice made concerning

the way in which errors are handled can lead to incompatibility of components. In some systems, the provision of a default value may be the best way to deal with an error produced by a missing or invalid parameter. In another system the user may not wish to use the chosen default value and so is unable to use the class without modifications.

Biggerstaff [3] suggests that component reuse involves solving four problems. They are

1. finding components from a library or catalogue
2. understanding components, said to be a fundamental problem.
3. modifying components to fit exact requirements.
4. composing components to make new computational structures.

The reuse of design information is said by Biggerstaff [3] to be hampered by the lack of a clear implementation independent system for representing the information. It is further suggested [17], that it is impossible to record all the information required to allow a newcomer to a system to understand it completely.

There are several problems with domain analysis (Horowitz in [3]). These include the method employed for recording the results and how to make use of them. Neighbors (ref'd in Horowitz [3]) has done a lot of work in this area.

Large scale reuse is hampered by the fact that there is no agreed "best" technical approach to reuse and no method to analyse a real situation where the domain is not narrow and well defined [3].

The author has found problems with reusing the Eiffel class libraries due to poor documentation [18]. Esp [19] has reported similar problems with the Smalltalk-80 library.

7 Suggested Approaches to reuse

The first and most obvious suggestion is to look in the library or records to see if a similar class has been developed before, or if you have written one before.[15]

Wirfs-Brock et al. [1] have the following ideas for the production of applications using object oriented principles. They suggest three ways to produce software in order to increase reuse.

1. Components such as lists, arrays etc can be produced. Other examples are buttons and check boxes on interfaces. These components are discovered when programmers write very similar pieces of code many times. The ultimate aim is to abstract out common functionality during the design of

a piece of software. Components are standalone pieces of software used to provide a given piece of functionality. They do not usually require services from the systems into which they are inserted. They are the servers in a client-server architecture.

2. Frameworks or skeletal structures of programs can be developed. These require to be fleshed out to build complete applications eg. windowing or simulation. The goal is to make them refinable. The interfaces must be as clear and precise as possible. They contain both reusable design and reusable code. They use unmodified components plus framework specific extensions and use code unique to the framework. They can act as clients or servers in the system.

3. Applications or complete programs. The primary goal is maintainability. Ideally applications are built by fleshing out frameworks. Domain specific details are required and the application should present an interface consistent with existing components and frameworks. It is suggested that sections of programs which could be useful in different applications should be designed as components.

The first two products are the results of abstracting reusability from an application whilst it is being built.

Rubin [2] identifies four general purpose techniques for developing reusable components. The first two techniques are said not to lead to the production of new reusable classes because they produce specialised, complex classes. The remaining two techniques increase reusability by producing simpler classes.

1. Refinement. New classes are developed by refining existing ones, making them more specialised. This is referred to as programming by difference and uses the inheritance mechanism.

2. Composition. New classes are created by combining existing ones. This process can be called delegation. The new component delegates responsibility for supplying the state and behaviour to its component parts by using the functions supplied by them.

3. Abstraction. One class, called a base class, is designed to contain the state and behaviour common to a group of objects. The required classes can then be developed, by refinement, from the base class. The base class is reused in the system and is available for future reuse.

4. Factorisation. This is the converse of composition. A complex class is split into smaller, simpler ones which can then be combined, by composition, to form the required complex object. The simple classes will then be available for reuse.

The second two techniques of abstraction and factorisation mirror the first two of refinement and composition. The general idea is, that if there are no simple classes available from which to build your system, design the simple objects yourself and then specialise them. These techniques are based on the assumption that smaller components are more reusable than complex specialised ones.

The Eureka Software Factory project [14] is working on the factory style use of CASE products. They identify the need for two types of producers. These are the producers of the components and factory producers which use the components to produce systems. This reflects other views [20] that the reuse of designs falls into two categories, namely design for reuse and design with reuse. The Eureka Software Factory project research concentrates on the design with reuse aspect. A client-server architecture is used. A system consists of small user interaction components (clients) and larger less specific service components (servers). A software bus is used to link the components. The interfaces of the components must conform to a set of protocols defined by the bus. These protocols could be used to prevent the problems of incompatibility caused by variations in error handling etc. identified by Berlin.

A reusable software library has been developed [20]. The library contains data structure components and consists of a database and four subsystems to provide library management, query facilities, a software component retrieval and evaluation facilities, and a software computer aided design package. They suggest that a software components library should be separated into multiple databases containing project specific components. They also found that the addition of the software computer aided design tool made it possible to use the library for both design and implementation.

Parnas, in [3], suggests that a system should be divided into three modules. Each module is used to hide one aspect of the system. The modules are:-

1. a hardware hiding module to contain any hardware dependent programs.
2. a behaviour hiding module to contain any parts that would need changing to cope with changing system requirements.
3. a software decision module to hide all the software decisions about theorems used and efficiency.

Parnas admits that this division into modules is fuzzy. The reason is that some hardware systems support functions which must be carried out by software in other systems. The software is designed to be easily adaptable, not general because generality is considered to be too inefficient. All the modules are specified to provide a substitute for the information hidden by telling how to use the module. The module decomposition forms the basis of the cataloguing system for the components.

Lamb [8] also advocates the use of information hiding to increase reuse. He points out that modules which do not (or appear not) to use any other modules are the easiest to reuse. He also states that parameterised modules are more readily reusable.

Goguen, in [3], advocates the use of parameterised programming. The generic packages available in Ada provide part of the requirements for parameterised programming but are said to lack the semantic information needed. Parameterised programming requires the possibility of using modules as well as simple types as parameters to functions or methods. Parameterised programming is said to imply bottom up development but several suggestions are made for producing top down designs.

Three forms of abstraction are used by Lenz [9] in the development of building blocks for use in systems software. These are data abstraction eg. abstract data types, functional abstraction eg. conversion and sorting functions, and procedural abstraction eg. message handling and input checking. The code implementing these abstractions is encapsulated. In order to allow more flexibility and greater reuse, building blocks can be used as parameters to other building blocks.

Gossain and Anderson [21] suggest an iterative approach to object-oriented design in order to increase the reusability of components. They concentrate on reuse within a single domain stating that 'one cannot expect to design components within a domain for reuse outside that domain'. They also state that: "The designer must also design for reusability i.e.must cater for anticipated changes." This suggests that they are using reuse to mean adding functionality to an existing system or reusing the classes in the development of a similar system.

Basili and Rombach [4] have developed a comprehensive cataloguing scheme for products to help analyse the differences between existing products and the requirements of the new system. The results of the analysis are used as the basis for deciding if reuse is appropriate. The scheme is intended for use with all the products of software development such as software modules, processes and estimation techniques.

Several guidelines for enhancing software reusability in both object-oriented and traditional systems are given by Smith [22]. Some of the guidelines seem to try to address the problems of incompatible classes identified by Berlin, and others are considered requirements for reuse by other authors. The following is a list of the guidelines.

1. Use encapsulation to hide details. This is considered a requirement by other authors.

2. Use access methods for inter-module communication. This appears to mean reduce the public interface of modules to control access to methods.

3. Use message passing for inter-object communication. In a fully object-oriented system this is the only way for communication to take place. The system developed by Smith was implemented in C++ and used the ability to mix object-oriented and procedural styles to improve the performance, hence this guideline.

4. Use built-in support for array indexing. This is again due to the language used. C++ is an extension of C and has the ability to use pointer manipulation for array indexing but reuse between compilers and hardware may not be feasible if pointers are used.

5. Provide multiple constructors for classes. This allows more flexibility for the programmer and increases the reusability of the class.

6. Identify abstract and concrete classes. Smith recommendeds that a deep hierarchy of classes is developed. Each layer would add only a few extra features to the previous layer. This would allow another developer to identify a good starting point when new classes are needed. A potential problem with the use of a deep hierarchy is the large number of classes to be examined for potential reusability.

7. Refine hierarchies fully before reducing to binary form. This is again a language dependent guideline. The Eiffel equivalent is closing a class [5]. The reduction to binary form means that the class should not be altered and additions should only be made by inheritance.

8. Hide low-level classes within high-level classes. This refers to the nesting of objects. A high level object is formed as a grouping of smaller lower level objects. This agrees with the advice given by Rubin.

9. Put the flow of control in the objects. These need standard protocols to increase the reusability by defining standard operations for each class and permitting polymorphism.

10. Use virtual methods where appropriate. These methods specify what to do at high level in a class hierarchy but not how. It is usually better to have a few high-level virtual functions rather than more low level ones.

8 Conclusions

Reuse can be considered at all stages of the development cycle. Reuse of products from early in the development cycle give the most benefit.

There are many areas in which further research could be carried out in order to overcome some of the technical problems identified as limiting reuse in software development. These include:-

1. defining a method for identifying abstract components in a given application domain (Gossain),

2. development of parameterised programming to enable more flexibility of components (Goguen),

3. defining a standard design for components which would give maximum flexibility to users of the component while maintaining the integrity of the component. The components must also not be either too large to be comprehensible or too small to be useful,

4. further development of a scheme for classifying components in a software library,

5. development of a system for representing design information and ensuring that it clearly represents all the information known to the designer.[3], [17]. The information includes the structure of the component parts and the relationships between them.

Areas 1-3 are concerned with designing for reuse and 4-5 with designing with reuse.

The last topic was selected for further work. A good system for representing design information and ensuring its completeness would allow a newcomer to understand the whole system and it's components easily. This would make maintaining or extending a current system easier as well as enabling the reuse of the components in a new system.

References

[1] R. Wirfs-Brock, B. Wilkerson, and L. Weiner. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[2] K. S. Rubin. Reuse in Software engineering: An Object-Oriented Perspective. In *COMPCON Spring '90 Thirty-Fifth IEEE Computer Society International Conference*, 1990.

[3] T. Biggerstaff and A. Perlis, editors. *Software Reusability, Vol 1.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[4] V.R. Basili and H.D. Rombach. Support for comprehensive reuse. *Software Engineering Journal'*, 9 1991.

[5] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[6] I. Sommerville. *Software Engineering.* Addison Wesley, third edition, 1989.

[7] M. T. Harandi and F. H. Young. Software design using reusable algorithm abstractions. *Communications of the ACM*, 29(4), 4 1990.

[8] D. A. Lamb. *Software Engineering: planning for change.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[9] M. Lenz, H.Schmid, and P. Wolf. Software Reuse through Building Blocks. *IEEE Software*, 29(4), 4 1987.

[10] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Redwood City,California, 1991.

[11] J. Nino. Object Oriented Models for Software Reusability. *IEEE Proceedings- 1990 Southeastcon*, 1990.

[12] D. Coleman and F. Hayes. Getting the best from Objects: the experience of HP. *Communications of the ACM*, 29(4), 4 1990.

[13] D. R. McGregor. Reusability - The Major Promise and challenge of the Object Oriented Approach. In *IEE Colloquium on 'Application and Experience of Object-Oriented Design*, 1991.

[14] C. Fernstrom, K. Narfelt, and L. Ohlsson. Software Factory Principles, Architecture, and Experiments. *IEEE Software*, 3 1992.

[15] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1991.

[16] L. Berlin. When Objects Collide: Experiences with Reusing Multiple Class Heirarchies. In *ECOOP/OOPSLA '90 Conference on Object-Oriented Programming: Systems,Languages, and Applications*, 1990.

[17] P. Naur. Programming as Theory Building. *Microprocessing and Microprogramming*, 15, 1985.

[18] J. A. Mayes. Eiffel, the universe and everything. TR 138, Hatfield Polytechnic, 1992.

[19] D.G.Esp. A beginners experience of smalltalk-80 for the evolutionary prototyping of an expert system. *IEE Colloquium on 'Applications and Experience of Object-Oriented Design'*, 18, 1991.

[20] B. Burton et al. The Reusable Software Library. *IEEE Software*, 7 1987.

[21] S. Gossain and B. Anderson. An Iterative Design Model for Reusable Object-Oriented Software. In *ECOOP/OOPSLA '90 Conference on Object-Oriented Programming: Systems,Languages, and applications*, 1990.

[22] J. D. Smith. *Reusability and software construction: C and C++.* Wiley, 1990.