Are there any parallels between object-oriented system development and other branches of engineering?

Technical Report No.139

A. Mayes and C. Britton

May 1992

Are there any parallels between object-oriented system development and other branches of engineering?

Audrey Mayes and Carol Britton

19 May 1992

Abstract

The task of a software developer is to produce systems which can adapt to changing requirements. Object-oriented methods attempt to manage the complexity of the system by using encapsulation, inheritance and polymorphism to build a modular system. In this paper, civil engineering and production engineering have been studied to identify any existing parallels between their designs and those of object-oriented systems with specific attention paid to the reuse of components. Production engineering was found to provide more parallels than civil engineering.

1 Introduction

This paper presents the findings of preliminary investigations designed to find correlations between object-oriented software engineering and other, well established branches of engineering, such as civil and production engineering. The original premise was that these well established branches of engineering may have some methods or techniques which could be adapted to an object-oriented software environment in order to speed up progress in the development, and use, of reusable components. The prelimary stage of the investigation was to determine similarities between the way systems are built in the three different branches of engineering. The next stage of the investigation will concentrate on the branch of traditional engineering which shows the closest correlation with object-oriented software engineering.

A major difficulty for software developers is the complexity of the required systems. The object-oriented approach attempts to simplify complex situations by viewing the required system as a group of interacting objects which represent objects in the physical world [1]. A system built by following object-oriented principles consists of components which communicate, via interfaces, to perform the required functions, in much the same way that a car consists of many parts

all joined together to form a mode of transport. In object oriented software construction, the components are called objects. Each object is defined in terms of its attributes (the data it contains) and its methods (the way it interacts with other objects).

It is possible to have many instances of the same type of object in a software system, for example, in a payroll system there would be many employees. All these instances share the same definition, that is they contain the same items of data (but have different values) and the same functions. The written definition is called the class and the instances of the class found in the system are called objects.

The grouping of object attributes with its methods into one unit is called encapsulation. The interactions between these software objects can be identified without considering how the object will be implemented. This hides the implementation of the class from the user and is known as information hiding. Other features which are used in object-oriented designs are inheritance, the ability to produce a new class by modifying an existing class, and polymorphism, the use of one name to mean different things depending on the context. Inheritance, as a software concept is unique to object-oriented systems. The other concepts are necessary for, but not confined to, object-oriented systems.

Another difficulty for software developers is that the system requirements are not static. New and modified requirements are often added both during the development of the system and after the system is in use. An ability to provide upgrades should be built into the original design.

The following two sections are the results of an analysis of two different branches of engineering, production and civil engineering, in order to identify any analogies.

2 Production engineering

In view of the supposed direct mapping of software objects on to physical objects [2], analysis of some common hardware systems was carried out to determine any correlation between the way they were assembled and the three concepts of encapsulation, inheritance and polymorphism. The method used for dealing with changing requirements was also compared.

The first system selected was a car because it is something with which most people are familiar. The following is a list of the features required by software systems and how similar features are exhibited in a car, plus other points of comparison between object-oriented software and production engineering.

2.1 Encapsulation

Encapsulation is the grouping of data with the operations which can be applied to it. Encapsulation is often accompanied by information hiding. The two

concepts together mean that the data associated with an object can only be accessed via an interface. A class can be implemented using encapsulation and information hiding. The user of such a class cannot access the code to change the implementation and so cannot introduce errors. Another advantage is the ability to change the implementation of one class without effecting the rest of the system. For instance, the representation chosen for the persistent data required by a system can be encapsulated. A different representation can then be substituted, should the need arise, without any effect on the rest of the system. The disadvantage is that it is impossible to correct any defects that may be present in the class.

A similar situation is present in some car headlamps. All the functions of light production, direction of beam, ability to dip the light and the fixing mechanism are located within one unit. It is not possible for the user to change the internal parts of the assembly. The annoying result is that if a headlamp bulb breaks, the whole headlamp assembly must be replaced. A beneficial result is that if a better version of the light is developed, this can easily be used to replace the old version, assuming that the same interface is used.

2.2 Inheritance

Inheritance is the mechanism by which a new class can be produced from a pre-existing or base class. The new class is defined in terms of the base class and inherits all the attributes and methods from it. In one form of inheritance, new methods and attributes can be defined for this new class. The addition of these new features allows the developer to make small or large scale increases in the capabilitites of the class. This is inheritance with enrichment. There are two different ways of using this form of inheritance. These are:-

1. the base class and the new class are concrete classes. [2] This means that it is possible to declare instances of the class. For example, a class, person, can be defined containing the name and address with all the required operations. Another class, worker, can then be derived using inheritance to add an extra attribute, occupation, and all necessary additional operations;

2. the base class is an abstract class. [2] This means that an instance of that class will never be created because it cannot be used on its own. The reason for including such a class is to allow common features to be grouped together. A software example is a printer class. This abstract class contains the behaviour required by all printers such as their status (busy or idle), speed and capabilities (fonts available). Each type of printer, dot matrix, laser or line printer would be concrete classes derived from printer and adding behaviour specific to themselves.

   Abstract classes can be implemented in the Eiffel language [3] in two ways. The printer class could be declared:-

(a) as a deferred class. This means that the additional features required are identified in the abstract class but not implemented as they are printer specific. The concrete classes derived from the abstract class are responsible for the implementation;

(b) as a class requiring a user defined creation procedure which is not supplied. All objects, in Eiffel, have to be created before they can be used. A default creation mechanism is supplied and used if no user defined creation is declared to be needed. Declaring a creation feature as necessary and not supplying it, makes it impossible to create an object from the class definition. The concrete class derived from this class would provide the required creation feature.

An alternative use of inheritance is when a method supplied with a class is not suitable for the required class. The new class can still be formed by inheritance. The only change required is the redefinition of the unsuitable method. A software example of this type of inheritance is a polygon class which defines a method for calculating the perimeter. In the case of a square this could be redefined with a simpler algorithm. This is inheritance with redefinition.

The investigation into production engineering revealed that cars from different manufacturers can use some of the same parts, for instance, filters, tyres, wiper blades. This seems to imply that all the cars must have the same fixing mechanisms. However, this is not the case. Wiper blades, for instance, can be adapted to allow fixing to different models or makes of car. This adaptability can be used as an analogy for both of the types of inheritance mentioned above.

1. Inheritance with enrichment.

   If the basic part always required an additional fixing mechanism to be attached, the basic class would be analogous to an abstract class. In the wiper blade example the following features are supplied by the wiper without its attachment:-

   - a spring to hold the blade against the window,
   - the ability for the blade to flip to wipe in either direction,
   - the correct size to fit the intended models.

   These functions correspond to those supplied by an abstract class. When the capability to be fixed to a car is added, the whole wiper becomes a usable object. This is then the equivalent of a concrete class.

2. Inheritance with redefinition.

   If the wiper only needs additional fixings for some car models, the basic class would be a concrete class, that is, it is usable in its current form. A new class would be derived from this by redefining the fixing mechanism.

Another analogy, possibly a better one, from the wiper example could be portability. In software terms this means that the system would be able to run on a variety of computers. One way to provide this is to channel all the interactions with the machine architecture and operating system through an interface, that is to encapsulate all the interactions with the hardware. This interface is known as a portability interface and is the only part which needs to be changed to allow the system to run on a different computer [4]. In the wiper example above, the basic blade is supplied with a fixing mechanism specific to the car model.

2.3 Polymorphism

There are two types of polymorphism. One is overloading where one name is used for a different function applied to different types. A software example is the use of '+' to mean perform addition for both integers and real numbers. The same signal is used but the result is either integer or floating point arithmetic.

In a car, all the electrical equipment relies upon an electrical impulse, to activate the terminal equipment but the result can be entirely different. For example a switch could send a signal which results in either blowing the horn or putting the wipers on. This could be interpreted as polymorphism, sending a message "switch on" which is interpreted by the relevant object.

The other type of polymorphism is genericity. The same function is applied to different types. A software example is the ability to manipulate the elements of a list. It does not matter what type they are, the same function can be used to add or delete them. No example of this type of polymorphism was identified in the car example.

2.4 Upgrading of the system

There are several software examples of this upgradable facility. Some are

- the availability of many versions of operating systems. It is possible to upgrade to a higher version offering more functionality.
- golf games can be bought with one or two courses and more added later.
- the standard and professional versions of a knitwear design package, DesignaKnit. The basic version can be upgraded to the higher one.

Similar facilities can be found in a car. All cars of the same model share some common parts but others which are different, such as the colour and cover fabric of the seats, engine size, presence or absence of a sunroof, the standard of the electrical equipment. The dashboard layout is the same for different versions, but the material may be different. The lower priced models have dummy slots which could be used to house a new piece of equipment or the switch for it. Often these features can be added at a later date.

Computer hardware is upgradeable in the same way. It is possible to buy a computer with a small amount of memory and add more at a later date.

2.5 Object interface design

The ability to define classes of objects without considering the implementation means that is is possible to have more than one implementation associated with a single definition. There are many examples of this in production engineering. There is a choice of supplier for most consumer durables such as telephones, washing machines, cars, chairs. Some of these require access to outside service providers, the telephone network, the electrical supply or a petrol pump. This means that all types of the same product must conform to a standard interface design, for example a car petrol tank must have a filler tube suitable to take the available petrol pump nozzle or vice versa. In an electrical system there must be compatible connections such as a plug and socket.

Leech's book on engineering design [5] suggests that if it is necessary to design a new part, such as a motor, then it should be designed in such a way that it fits into a standard framework (meaning that it conforms to a standard size) to minimise the changes required to use the new component. The same idea can be applied to software. If software components are to be used in different systems then it is necessary that the components will communicate in a defined way via the interface. In a software system it is possible to embed code of one language inside another language and thus provide a compatible interface.

2.6 Component variation

There are many forms of motors or other components. There are many different types of screw, for instance, for wood, metal and chipboard which all perform the same function and have the same basic design but are used in different circumstances. A similar example would be a class with various forms available, such as a class person which could come in the form of student, bank customer, club member, client, patient. Each variation would be used in a different application. These classes could be used as the basis for a new class if extra features were required.

2.7 Generic applications

Some application packages can be tailored to suit an individual's requirements. These include spreadsheets and database packages and payroll systems. A motoring analogy is the transit van which has a basic design which can be adapted to suit various needs such as minibus, delivery van, dormobile.

2.8 Catalogues

Catalogues of engineering components are produced to allow designers to choose the most appropriate component for the system. Similar catalogues would be needed if programmers were to be able to choose classes for reuse. A classification system would have to be developed to allow identification of the required class.

2.9 The mass production process

When a computer system has been developed, it is an easy matter to produce many copies. The same applies to the reproduction of software classes. The effort required to produce software is concentrated in the development and maintenance not in the mass production. This is not the same as in production engineering where each new system has to be assembled. Pirating of designs and ideas occurs in most industries resulting in loss of revenue for the original producer. In an industry, such as the software industry, where it is so easy to make multiple copies of products, pirating is almost impossible to control. This may limit the willingness of companies to market software components 'off the shelf'.

2.10 Reuse

In production engineering, the reuse of components is considered at all stages of development. The ability to mix the components from different manufacturers depends on the parts conforming to specified standards. This indicates that reusable software components would need to be carefully specified. The easy reuse of components implies a need for open systems, where components produced by different suppliers are compatible.

3 Civil engineering

The field of civil engineering, bridge design for example, has a different approach to development. The following list compares the development of a bridge with the requirements for a software system.

3.1 Encapsulation and inheritance

The use of off the shelf components is seldom considered except for small span bridges [6]. As a result it was not possible to identify examples of encapsulation and inheritance.

3.2 Polymorphism

Instead of using standard parts, the designers use standard formulae to calculate the sizes and form of the required parts. The reason for this is that no two bridges are exactly the same. They may both be road bridges of the same size but the places they link will not be identical. The differences between the locations and the size of the structures involved makes it cost effective to design special components for each bridge. The process of designing a bridge seems to concentrate on the major function of the bridge and uses templates for this in the form of standard styles such as suspension bridge, and standard methods for calculating the sizes and types of the required parts, one such method is stress analysis. The actual components from which the structure is eventually built are designed specifically for that bridge. The template is used with the added information about the size and material required. This is analogous to the form of polymorphism known as genericity mentioned earlier.

3.3 Upgrading the system

The development of a software system must take into account the changing needs of the user. This is relevant to civil engineering (at least for roads and bridges) when the changes in the size and nature of vehicles are taken into account. The bridge designers use safety factors which add in extra strength at the start of the project in an attempt to cope with the changes. There is no design effort made to allow for extra strength to be added after completion of a structure. The use of safety factors is not always successful as can be seen from the need in March 92 to remove a bridge from over the M4.

During the design process little or no effort is made to consider the costs of maintaining the structure to comply with the original specification. Only the original building cost is carefully estimated. Spector [6] sees this as a defect in the process.

3.4 Reuse

There is very little component reuse in civil engineering, but specifications are well defined and new designs must adhere to them. A software engineering interpretation of this would be a system built according to well defined standards of reliability using a new design and new code for each new application. This implies a reliance on thorough proof and testing for each system. The most important areas of reuse in this system are the personal expertise of the workers and the specifications.

There may be a correlation between the design methods used in civil engineering and the methods used in a functional approach to the structured development of software systems. A system designed by following the functional approach concentrates initially on the functions required and the necessary data

structures are determined later. This possible correlation could be verified by further work but is outside the scope of project concerned with object-oriented software.

4 Conclusion

Two branches of engineering, production and civil, were investigated in order to identify parallels between designs in those areas and object-oriented software design. The major features used in object-oriented designs are encapsulation, inheritance and polymorphism. A major system requirement is the ability to adapt to changing needs.

There is a major difference in the approach adopted by the two chosen branches of engineering. Civil engineering is concerned with 'one off' products whereas production engineering is concerned with the mass production of many products of the same design.

Production engineering was found to have examples of encapsulation, inheritance, one type of polymorphism and the ability to upgrade the system as the requirements changed. The use of ready made components is universally accepted as the most efficient method of production. These components conform to defined specifications.

Civil engineering was found to have examples only of polymorphism. There is no consideration given to extending the requirements of the system. Little use is made of ready made components. A comprehensive system for specifying the required parts of the structure has been built up over many years.

Civil engineering does not appear to be a fruitful area in which to look for analogies and techniques to improve reuse of components in the software environment. The reason for this is that little use is made of standard parts and the maintenance requirements of the structure are not considered. It may however provide useful information about methods used for the production of generic systems.

From the above analysis, production engineering seems to relate to object-oriented software development and reusable components much more readily than civil engineering. The decision has been made to continue the research by concentrating on the methods and techniques used in manufacturing engineering.

References

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

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

[3] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.

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

[5] D. J. Leech and B.T. Turner. *Engineering Design for Profit.* Ellis Horwood, 1985.

[6] A. Spector and D. Gifford. A computer perspective of bridge design. *Communications of the ACM*, 29(4), 4 1986.