

Advantages of using the object-oriented paradigm for designing and developing software

E. Post

poste@lincoln.ac.nz

Applied Computing, Mathematics and Statistics group,
Applied Management and Computing Division, Box 84, Lincoln University,
Lincoln, Canterbury, New Zealand

Abstract

There are ever increasing demands to develop good software rapidly. The cost of developing such software is a major factor. Thus the methods used to develop software should be those that will produce robust software as rapidly as possible, and preferably re-using existing software wherever possible. This paper first looks at some older approaches to developing software and then describes what is meant by using object-oriented techniques to design and develop software and how using an object-oriented approach when designing and developing new software is beneficial and helps developers to produce better software more quickly.

Keywords

Object-oriented analysis and design, function-oriented, reports-oriented, data-oriented, encapsulation, inheritance, polymorphism, design patterns

Introduction

Nowadays there is tremendous pressure to produce new software rapidly. And not only must the software be developed quickly but it must be of high quality, useable, easily maintained, and also easily adapted to changing requirements or extended for new requirements. In addition the scale of such software has changed dramatically from small programs for single users on a computer with relatively low capacity to being extremely large applications involving perhaps thousands of simultaneous users and running on anything from one powerful mainframe to a distributed network of perhaps thousand of computers, maybe even worldwide. In fact, Bill Gates is reputed to have said in 1993 “The mainframe industry will cease to exist on the first day of the new millennium” (Brown, 1997). Maybe that hasn’t quite happened, but there have been major changes in the ways computers are used and things are not as simple as they once were. Also, as technology is changing so rapidly the lifetime of many applications may be short, so that an application may be out of date within a year or so, or require significant changes to keep up with changing demands. The question is with these demands how is it possible to design, build and deliver applications which are robust, flexible and extendable, and which are developed and delivered in a very short time?

Over the past fifty years of commercial computing there have been great advances in developing techniques for analysis and design of software. In the early days when programs were single-user applications and at most a few thousand lines of code it was relatively easy to design as one coded. However, as applications became larger and more complex it was necessary for better analysis and design techniques to be developed.

This paper briefly describes some of the older techniques of analysis and design and compares them to object-oriented analysis and design, showing how using object-oriented analysis and design techniques properly makes it easier to design, build and deliver great applications rapidly.

One of the reasons for writing this paper is that despite the move to the object-oriented paradigm in recent years there are still many people who are not quite sure how this differs from other analysis approaches. Also many people have progressed into using object-oriented languages without having changed their analysis techniques, or even thought about doing so. Some people regard the object-oriented approach as just another step further from structured analysis and entity-relationship modelling, and that it is not radically different, just further along.

For instance, occasionally people may think that because they have been programming using an object-oriented language for years this implies they are using object-oriented techniques. However, it is perfectly possible to design

and develop programs using an object-oriented language where the program does not use object-oriented techniques at all, but rather a procedural or functional approach, and none of the benefits of using an OO-approach are realised. Programmers can write the most appalling, long, cumbersome and buggy code, even in an object-oriented language such as C++, Java, or even JADE. This is also not helped by the fact that some manufacturers produce a language that they claim is object-oriented but which does not include such features as inheritance or polymorphism that are fundamental to the object-oriented concept. No wonder people are confused by the hype and are unsure exactly what the object-oriented paradigm is.

Similarly, because class diagrams are in some ways not unlike entity-relationship diagrams it sometimes happens that analysts use data modelling techniques to produce class diagrams as a product of analysis and call this object-oriented analysis. However, if the process has not included the analysis of functionality as well as the analysis of data then it is not object-oriented analysis, even if a class diagram is produced.

The object-oriented paradigm is a revolution rather than an evolution and requires a radically different way of thinking from previous approaches. It is hoped that this paper will clarify these issues so that people are able to determine whether or not they are genuinely using the object-oriented approach in analysis, design and implementation, and if they are not currently using a genuine object-oriented approach that they learn to appreciate the benefits of using the object-oriented paradigm for analysis, design and implementation and join the object-oriented revolution!

Analysis and Design Paradigms

There have been many approaches to analysis and design over the past fifty years or so. Ignoring the earlier “design-as-you-program” methods, these approaches could be summarised as follows:

- ***Function-oriented***, where the emphasis is on what you *do* in the application, and the consideration of data is secondary. This approach is sometimes also called ***action-oriented*** or ***process-oriented***.
- ***Reports-oriented***, where the consideration of what data must appear in the *reports* and what data must be *input* to the program drive the design of the application, and what must be done to the data in the application to achieve this is of less importance.
- ***Data-oriented***, where the most important consideration is designing the *database* and deciding what data must be *stored* together and how it *relates* to other data, and the functionality of how the data must be processed in the application is only considered after the database has been designed.
- ***Object-oriented***, where *both the data and the functions* to be performed on the data are considered *together* from the start of the analysis process, right through design and implementation.

This section looks briefly at each of these methods, describing each more fully and showing some of the disadvantages of the first three approaches as compared to an object-oriented approach. This material is only covered briefly here and more extensive material on this subject can be read in books such as Brown (1997) and Schach (1999).

Function-oriented analysis

This type of analysis is also sometimes known as action-oriented or process-oriented and includes techniques such as functional decomposition, dataflow diagrams and structured analysis.

In this type of approach the emphasis is on what the application *does* with the data, and the design of data structures and any database is secondary. This approach will often be used by those who come from a procedural or functional programming background, and have done a lot of programming in languages such as Fortran, Pascal, C, Visual Basic etc. It is perfectly possible to write very good and well-designed programs in languages such as these, using techniques such as structured programming, modular programming or top-down programming. Nevertheless, the emphasis remains on what is *done* to the data in a subroutine, procedure or function, and different items of data are associated only by being *used* together, not because they have anything else in common.

Even in the best designed programs of this type where sub-program units are modular, having one entry point, one exit point, and no global variables with all necessary data passed into the sub-program unit as arguments or parameters, difficulties may arise because the design of the database is largely ignored. For instance one sub-program unit may use say three different items of data. Then each of these data items may be used elsewhere in the

application in association with different data items. This can lead to inefficiencies in data access as data that are not stored together may be used together, thus leading to reduced performance in the application. Also if the application is extended and new data items are needed, since these were not in the original design these are now likely to be stored in different places from other data with which they may be functionally associated and thus used together. A large number of such changes can seriously adversely affect the performance of an application and is one of the reasons why applications may deteriorate in many ways as they are extended.

Even more serious problems arise if for instance a data item is no longer needed, or the structure or format of a data item is changed, such as from integer to floating-point, or string to integer, and this data item is used in several places in a large application. Such changes require that everywhere in the application where that data item is used the necessary changes must be made, and often these changes may have further far-reaching consequence affecting other data items, or functional changes to the program, such as in argument lists, or perhaps reports where the data item is used. In very large applications it can be very hard to ensure that all necessary changes have in fact been made, and that they also have not caused new bugs in a program that used to work. It is this sort of change that causes very high maintenance or extension costs.

Another problem with function-oriented applications is that there tend to be large sections of code together, even in the best-designed applications, and it is always harder to debug larger pieces of code than shorter sections, as in object-oriented methods. Also, it often happens that these larger code sections may perform more than one task in the same sub-program unit. Changes in requirements may then affect one part of the code of a sub-program unit, but not another, all increasing the likelihood of errors.

These drawbacks may be most apparent in non-object-oriented applications. However, if someone uses these techniques of concentrating primarily on functionality while trying to design an object-oriented application it can lead to inappropriate grouping of otherwise unrelated data items into classes for no other reason than these items are used together, or to inappropriate programming of functionality. For instance I have seen programmers write long methods for form interface classes where these methods do a lot of work even on other objects such as other form or entity objects. This is inappropriate in an object-oriented application and such functionality should usually rather be on methods belonging to other classes.

Thus an analysis approach that considers primarily the functionality of an application and largely ignores the data needed can lead to many difficulties in implementation, maintenance, extension and modification.

Given these arguments some people still say that if the structured paradigm is so inferior to the object-oriented approach why has it produced such good results? Schach (1999) explains that when the structured approach was first adopted in the 1970's and 1980's software engineering had not been widely practiced, and software was frequently just written without any analysis or design at all. Thus the structured paradigm was the first time that many developers were exposed to methodical techniques, so these new approaches led to major improvements in the software industry. However, as applications grew in size it became apparent that structured techniques were no longer adequate for these very large applications, and new techniques were needed.

Reports-oriented analysis

In this section it is important to understand the difference between *reports-oriented* where the reports needed drive the analysis and determine how data is stored which is quite different from taking reports into account when doing overall analysis, which is very important and will be discussed further below.

In the past the *reports-oriented* approach has been a common approach to analysis, especially in trying to establish understanding between customers and analysts where neither talks the other's "language". As a first step in analysis it is common for the analyst to ask the customer what reports are needed. From there it is then possible to determine what data must be input to produce these reports. This approach tends to produce designs for databases where data that is used together to produce the same report is stored together, even if it is not otherwise related. In this type of analysis little consideration is given to the functionality needed to produce the reports until later stages of design and implementation.

This approach may appear to work adequately initially. However, problems arise when additional reports are required, or existing reports are changed. This means that decisions have to be made regarding how the additional

data needed is to be input, and most importantly where it is to be stored. In a design driven by reports needed it is likely to happen that data used in the same report is stored in the same table of a relational database. However, if this data is needed together with additional data in a new report the decision has to be made about where to store the new data. If it is to be stored in the same table as the original data then the structure of the table must be changed, and the old data converted to the new format. Also this sort of change could lead to data duplication and redundancy. Frequently it will be seen that such changes necessitate a complete redesign of the database, perhaps involving new tables and regrouping different data items into different tables, so data items that were in the same table are now in different tables. This sort of change often involves considerable changes to code as everywhere these data items were accessed now has to be changed (unless the program has been very well written when the changes should be minimal). All these contribute to problems that increase maintenance and extension costs and increase the likelihood of errors.

Thus the longer the life of an application the more likely it is to change with more reports being added or changed. The resulting changes to the application tend to produce an application that becomes progressively more inefficient and cumbersome, and more difficult and costly to maintain. Often it becomes cheaper to scrap the old application and redesign a new one from scratch.

Again, although this approach is most commonly used with non-object-oriented systems it is still very common that the first thing discussed in preliminary analysis is “what reports are required?”, but this can also be good. As we have seen it is important that the reports do not drive the design of the system and the way the data is stored. However, during the analysis phase it is very useful and important to study the major reports required by the system, including such things as what data needs to be in them, how it is to be derived, how much data will be involved, how often the reports will be needed. The answers to these questions affects the design of such things as keys needed, locking strategies, collections, and whether data should be global or local, and whether the system will scale. However, you will notice that these questions are looking at not only the data produced in reports, but also at functionality associated with how reports are requested and used by users of the system, how often the reports will be created, how they will be created, and how much data will be involved, which affects the designs of the behaviour of the objects containing the data or relating to the data, such as collection objects. Therefore studying reports needed by looking at both the data needed for the reports and the functionality of how the reports are to be produced is object-oriented rather than data-oriented and analysis of reports forms an important part of object-oriented analysis and design.

Thus a reports-oriented analysis approach focussing only on the data needed in the reports could adversely affect the design of an object-oriented application. However, it is essential to give due consideration to reports during object-oriented analysis and design, considering not only the data reported but all the associated functionality.

Data-oriented analysis

This method of analysis focuses primarily on analysing the data needed and how it relates to other data, and designing the way it will be stored in a database. This approach is most commonly found among database programmers who have learned the importance of designing databases properly. In data-oriented approaches the functionality is usually considered only after the structure of the database has been designed.

There are several different methods falling under this heading but perhaps the currently most widely-used is the Entity-Relationship approach, first introduced by Chen in 1976. In this approach the emphasis is on entities and the relationships between them, and the data describing each entity is stored together.

Entity-relationship modelling has had considerable success because it addresses the problems arising in the function-oriented or reports-oriented approaches where often the rationale for how data is stored is driven by what data is *used* together or *reported* or *input* together, which often led to poor database design, and where it was difficult to extend the database to meet changing needs of the application. In entity-relationship modelling the focus is more on identifying the *entities* to be modelled, rather than on what is *done* with the data. Entities may be real or conceptual. However, entities tend to map onto real-world entities such as people, addresses, accounts and so on. Thus data is grouped together according to how it describes a real-world entity rather than how it is used or reported together. This also makes it easier to add new data describing an existing entity, as it is logical that this new data should be stored with the existing data for that entity. Another important concept in entity-relationship modelling is the

relationships between entities. Again this approximates real world relationships, such as Customer *has a* BankAccount, Mother *has* Children, BusDriver *drives* Buses.

This emphasis on grouping together data that describes a real-world entity means that applications designed by using this approach have tended to produce databases that are more easily changed or extended as the application changes. Also databases designed in this way avoid many of the problems that can arise in databases with duplication and redundancy of data. Overall the entity-relationship modelling approach has led to databases that are much better designed than formerly.

The entity-relationship approach does not allow the modelling of class hierarchies or inheritance. However, this is overcome to some extent by the extended entity-relationship model that does allow this.

The major drawback of data-modelling approaches such as entity-relationship modelling and extended entity-relationship modelling is that they still do not consider functionality during the analysis stage. The emphasis during analysis is on *what* must be done rather than *how* it must be done, which is left to the design phase. Thus a designer or programmer still has considerable flexibility on *how* to implement the design, and this implementation may be done entirely without reference to the analyst. Frequently in fact there is no designer and the programmer has complete freedom on design and implementation. This may work well for an experienced programmer but can be disastrous with an inexperienced programmer. For instance the implementation could end up locked into a particular approach to the functionality that is not easily modified or extended at a later stage, and may end up being very expensive in both time and money to change.

This major division between the analysis and design phases in both data-oriented and function-oriented approaches has led to many major breakdowns in application development. In fact programmers often say that they take no notice of the documents produced by analysts because they have no relation to the implementation issues that arise during design, and these implementation issues often force major changes to the initial analysis so that frequently the end product does not have much similarity to the product intended by the analysts. So it is obvious that analysis focussed on only one of functionality and data is less likely to produce a good design without modification, possibly considerable, during the design phase. Therefore a better approach is needed.

Object-oriented analysis

Now that we have had a brief look at some other methods of analysis it becomes important to understand what is meant an object-oriented approach and how is it different from other approaches. Essentially the main difference between object-oriented analysis and earlier methods of analysis is that in object-oriented analysis **functionality and data are both considered together** and given **equal importance** throughout the analysis and design phase, where earlier methods only considered **either** data **or** functionality. Unlike entities in entity-relationship modelling that only have data, objects have **both data** (descriptive and relationships) and **functionality**, hence the term **object-oriented**. However there is more to the object-oriented paradigm than this. The object-oriented paradigm also includes making effective use of **classes** and **objects**, **encapsulation**, **inheritance** and **polymorphism**. The object-oriented approach is more than just a paradigm shift, it is a whole new way of thinking, including several new concepts as well as new methodologies.

But perhaps not as new as all that. Roman Emperor Marcus Aurelius said “*When an object presents itself to your perception, make a mental definition or at least an outline of it, so as to discern its essential character, to pierce beyond its separate attributes to a distinct view of the naked whole, and to identify for yourself both the object itself and the elements of which it is composed, and into which it will again be resolved.*” This is not a bad description of an object (and aggregation) for more than 1800 years ago!

In contrast to the sharp transition between analysis and design in function-oriented or data-oriented approaches, when object-oriented analysis is used objects (with both data and functionality) enter the process from the very beginning. The objects are extracted in the analysis phase, designed in the design phase and coded in the implementation phase. Also, modern approaches to object-oriented analysis and design frequently involve an iterative process, so there may be several iterations, in particular of the analysis and design phase, and it is unlikely that all analysis will be done before any design is done. Thus the object-oriented paradigm is an integrated approach; the transition from phase to phase is far smoother than with function or data-oriented analysis, thus reducing the number of faults during development. (Schach, 1999) Also because the design includes the complete detailed design

of both data and functionality it is more likely at an inexperienced programmer can still produce good code as he or she does not have the flexibility to make poor design choices as with other analysis approaches.

Let us now see what is meant by each of these characteristics of classes and object, encapsulation, inheritance and polymorphism, and how they provide a better paradigm.

Classes and Objects

An *object* has:

- Descriptive attributes
- State attributes
- Relationships to other objects
- Behaviour (or functionality)

A *class* consists of a group of objects that has the *same* set of attributes, relationships and behaviour although the actual values for those properties will usually be different for each object in the class.

The major difference between objects and entities as in data modelling is that objects consist of both data and functionality, and the functionality is directly associated with the object unlike in non-object-oriented implementations where the functionality is totally separate from the data. The data of an object is stored in properties of the objects, and the functionality that uses and manipulates that data is coded in methods belong to the class to which the object belongs and not in totally unrelated code. Thus data in an object should only be changed by methods belonging to the class of the object. Any methods belonging to other classes must use the class methods of the object concerned to change the data and not change the data directly. A class corresponds approximately to an entity in an entity-relationship diagram, although entities do not have functionality. Objects are instances of the class.

Like entities, object map well onto objects in the real world, although with objects functionality is included as well as data. This close correspondence between objects in a product and their counterparts in the real world promotes better software development. An advantage of the object-oriented paradigm is that well-designed objects are independent units. Since the actions performed on the independent units are included in the object then the object can be regarded as a conceptually independent entity. Everything in the product that relates to the portion of the real world that is modelled by that object can be found in the object itself. Since these objects are independent entities they can be re-used in other products, and re-use reduces the time and cost of both development and maintenance, whether corrective or adaptive. (Schach, 1999)

Further advantages of using classes and objects become more apparent as we look at the advantages of using encapsulation, inheritance and polymorphism.

Encapsulation

One of the most important features of the object-oriented paradigm is encapsulation. This means that the data is “hidden” inside the object and can only be accessed via its public interface as provided by the class methods associated with that object.

In a well-designed object-oriented implementation any changes to the way data is stored within objects should have no impact on the rest of the application providing the signatures of the methods accessing the data within the objects do not change, even if the actual implementation of these methods changes. Thus the changes are localised entirely to the attribute changes within the object and any changes to the implementation of the methods accessing this data. Since the signatures do not change these changes should not affect any other code in the application.

This is also particularly important when it comes to re-use. Ideally changes to classes, whether to the structure of the data or the implementation of the methods, should also not have adverse effects on other applications that re-use these classes, but sometimes this is more difficult to achieve. It usually happens that classes that are widely re-used may actually go through several iterations as their design is improved until the design is stable, and properly re-usable.

Encapsulation is not only useful in design but can also be beneficial with regard to the implementation of the database, depending upon how the database is actually implemented. In a pure object-oriented database such as JADE objects map directly from the code through to the database and any changes to the structure of objects is encapsulated in those objects and usually have minimal impact. However, if an object-oriented application is implemented using a non-object-oriented database then there is the problem of impedance mismatch as changes in the structure of the objects may still cause considerable implementation difficulties, again depending on the implementation. However, these difficulties will be lessened if there is a well-defined “layer” between the application and the database, as changes to the structure of objects should only affect this layer and not the underlying database.

On the other hand it would be more difficult to implement such changes to data in a non-object-oriented implementation where data and code manipulating that data are quite separate. In this situation the changes would mean both changes to the database and also changes to code throughout the applications, wherever these data items were accessed. There could be a large number of these changes and it would necessitate a complete new revision of the application, whereas in an object-oriented application changes are localised only to the definition of the class and the implementation of the methods of that class.

Thus objects are essentially independent units with a well-defined interface. Consequently they are easily and safely maintainable and the chance of a regression fault is reduced. It is safer to construct a large-scale project by combining these fundamental building blocks rather than using for instance a functional-oriented approach such as the structured paradigm. (Schach, 1999)

Inheritance and Polymorphism

These are two of the most powerful features of the object-oriented paradigm and will be discussed together as they are closely related.

Inheritance provides the opportunity for creating new classes without affecting or changing existing classes, where these new sub-classes inherit data and/or functionality from the original classes. One particular advantage is that the existing methods of the superclasses have already been coded and debugged thus reducing costs. Thus inheritance is particularly useful for adapting or extending an existing application and is also one of the major ways of writing reusable code.

Polymorphism is closely linked with inheritance as inheritance is essential for polymorphism to work. In polymorphism different classes may have methods of the same name which all perform the same conceptual task, even though the implementation may be different for each class. With polymorphism each object “knows” to which class it belongs and uses the method of that name belonging to that class, and any calling method does not need to first determine to what class the object belongs to determine which method of that name to execute. A further advantage of polymorphism is that existing code using polymorphism will automatically work with objects of new classes when these classes did not exist when the code was originally written. Thus polymorphism is an extremely powerful feature of the object-oriented paradigm and is especially useful for writing generic, re-useable code.

To try and implement inheritance and polymorphism to extend existing data and code in a non-object-oriented application is not so simple. In particular, even if some approximation to these features is possible, it is likely that this would not easily be extendable or adaptable. On the other hand, in an object-oriented application such problems are easily solved with inheritance and polymorphism.

Although inheritance may be modelled using extended entity-relationship diagrams, polymorphism cannot as it is essentially about behaviour. It is important to consider behaviour at an early stage in the analysis and design process as this can lead to tremendous benefits, such as by making use of polymorphism. In fact it is essential that possibilities of polymorphism should even drive analysis as otherwise trying to add polymorphic behaviour at a later stage may necessitate a complete redesign. Using data-modelling techniques to produce a class diagram will not discover any relevant polymorphism. The need for polymorphic methods must be determined during the object-oriented analysis process, even though their final name and signature may only be determined during the design phase. This discovery of necessary functionality while at the same time discovering necessary data is part of object-oriented analysis and differs from earlier forms of analysis.

Object-oriented analysis and design

The previous sections have described briefly the features involved in object-oriented analysis and design. However there also need to be techniques to actually do the analysis and design. A large number of methodologies to do object-oriented analysis and design have been developed. One that seems to be growing in acceptance is the Unified Modelling Process that has arisen through collaboration between Grady Booch, Ivar Jacobson and Jim Rumbaugh. (Jacobson, 1999), (Quatrani, 1998), (Kruchten, 1999) These three leading practitioners decided to make their contribution to peace in the methodology wars by combining their approaches into one. They have also made significant advances in getting the Unified Modelling Language to be accepted as the standard notation for object-oriented analysis and design. However, the methodology wars are not yet over and there continue to be other methods and notations, such as those developed by Coad.

Since object-oriented analysis and design are considering both data and functionality in the process, there are a number of techniques involved, for instance to discover functionality and the way the application will be used, and by analysing functionality identify what data is needed. These techniques are another way in which object-oriented analysis and design processes have advanced over earlier methods, such as function-oriented or data-oriented analysis that tended to rely more on brainstorming and argument rather than organised processes of discovery. This is partly because function-oriented or data-oriented analysis approaches have mainly been associated with relatively smaller applications where it was possible for an analyst to keep the whole picture of the application to be designed in his or her head. Nowadays applications have become so huge and complex that it is virtually impossible to consider the whole application at once during analysis, and the analysis process has to be broken down into smaller units. Thus it has been necessary to develop techniques and methodologies as used in object-oriented analysis and design.

These techniques are used to produce various types of diagram such as use-case diagrams, sequence diagrams, collaboration diagrams, state diagrams, activity diagrams, class diagrams, component diagrams and so on. It is not necessary to produce all these diagrams for every analysis, but usually some combination is used in an iterative process that culminates in the production of usually many class diagrams, with each class diagram describing a different part of the final system. The iterative process usually means that the class diagram goes through several stages, usually incorporating both analysis and design phases, and consideration of both data and functionality, before the process is complete. Then, with modern CASE tools, such as Rational Rose¹, it is often possible to generate the basic code skeleton for the application from this design, often in a choice of different languages. This can save a tremendous amount of time in avoiding tedious coding, and also avoids the bugs that might arise during hand-coding.

Design Patterns

So far we have seen how the object-oriented paradigm enhances the possibility of reuse as objects are well-suited to the concepts of creating reusable components and application frameworks. However, not only code but designs can be reused as well. In 1977 an architect, Christopher Adams, described the concept of patterns: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice", quoted in (Gamma, 1995). This concept has been adapted by the software community which has realised the significant advantages of reusing designs, rather than continually re-inventing the wheel.

There are many advantages in re-using designs. Usually a good design is only achieved after several iterations, so reusing someone else's good design saves the time required to develop the design. And since the analysis and design phases may take a significant part of overall development time, perhaps as much as 40% to 50%, reusing already proven designs can save significant time and cost. Also experienced practitioners are more likely to produce good designs than inexperienced practitioners are, and experience can take years to acquire, so reusing designs takes makes it possible to take advantage of someone else's experience and perhaps specialised expertise.

As object-oriented approaches matured and greatly improved the chances of good designs leading practitioners realised the advantages of reusing designs, and in 1995, the "Gang of Four" produced the book "Design Patterns"

¹ We acknowledge the Rational SEED program which generously makes their software available to us at Lincoln University for teaching purposes.

(Gamma, 1995) which has had significant impact on the software community. This book documents and describes a number of commonly occurring patterns such as the Abstract Factory, Singleton, Composite, Observer and many more patterns, grouped in categories of creational patterns, structural patterns and behavioural patterns. These patterns all represent good designs for commonly occurring needs, where the designs have usually resulted from several iterations of designs by experienced practitioners. Another good book is by Larman (1998).

Practitioners can then take these design patterns and put them together to make applications. Usually several patterns will be used in combination in each application. In particular design patterns are well-suited to being used to build reusable application frameworks, which also reduce development time and costs once they have been developed.

One advantage of design patterns is that they are language-independent and can generally be implemented in any truly object-oriented language, although occasionally language-specific requirements may require slight modifications. In addition code generators are becoming increasingly available, so the same design pattern can be used to generate code in a variety of languages.

As the need grows for being able to develop good software quickly the use of design patterns is becoming increasingly important, and more and more effort is being spent in designing good patterns. Also as computing changes from being focussed on mainframe computers to distributed systems this has led to a need for a large number of new patterns to handle these issues, and issues such as exception handling, concurrency and security. It is likely that in the future design patterns will play an increasingly important role in designing and producing good software.

Advantages of Object-oriented Analysis and Design

Object-oriented analysis and design has now become established as the new direction for the information systems industry. (Brown, 1997) This move has been accelerated by two factors:

- The success and rapid spread of object-oriented programming
- Recognition that object-oriented methods produce more effective, efficient, flexible and stable information systems

In answering the question of whether using object-oriented methods produces better software, Brown (1997) shows how in the past some of the major problems to do with producing software were

- Maintenance, where fixing errors and adapting or extending a program are incredibly costly in time and money,
- Backlogs, where so many systems were not delivered on time
- Reliability, because errors in design led to systems that were inflexible, unreliable and of poor quality.

Brown (1997) then summarises some of the benefits of using the object-oriented paradigm as follows:

- ***System stability*** - object-oriented techniques tend to produce systems that are resilient to change, where changes can be made without major disruption, with minimal time and effort, and with little fear of disturbing something else in the system. This stability and resilience happen with objects because the system is designed to manage and support the user's business and is based on the fundamental data needs of the system, rather than the ad hoc needs of a few reports.
- ***Maintainability*** - object-oriented methods produce systems that can be maintained and enhanced more readily. These systems have fewer errors than systems produced by other methods and are more easily adapted or extended to meet changing requirements.
- ***Reusable components*** – object-oriented features such as inheritance and polymorphism lead to smoother, more efficient reuse in both code reuse and design reuse.
- ***Reality-based systems*** – the techniques used with object-oriented methods give a far more accurate picture of the users' business operation and its information needs, leading to a final system that is closer to what the customer actually wants.
- ***Data accessibility*** – object-oriented methods tend to lead to better design of databases, thus data is more accessible and usable.
- ***User involvement and ownership*** – with object-oriented techniques the users are more fully involved in the systems project. The end result is that users develop a sense of ownership in the system, and this repeatedly engenders a level of co-operation that was rarely seen in the old days.

In addition systems produced by using object-oriented methods are more suited to incremental development, increasing the likelihood that a useable system will be produced on time, even though it may not have all the features originally planned.

Schach (1999) answers the question of whether the object-oriented paradigm is in fact superior to all other present day techniques by quoting research by IBM (Capper, 1994) that reports on three projects developed using object-oriented technology. This report showed that in almost every respect the object-oriented paradigm greatly outperformed the structured paradigm. In particular there were major decreases in the number of faults reported and far fewer change requests during development and maintenance that were not the result of unforeseeable business changes. Also both adaptive and perfective maintainability increased significantly. There was also an improvement in usability, but no meaningful difference in performance.

Brown (1997) also quotes the Koontz (1994) review of the August 1994 Conference on Object-Oriented Software Development. He summarises reports from a variety of companies describing the varying levels of success of their object-oriented conversions. From these he gave several observations, including these two:

- The first or second projects you can expect will be about 30% more expensive using object-oriented techniques (probably mainly as a result of learning new skills)
- The third project and onward will be money savers.

Is the object-oriented paradigm the “Silver Bullet”?

Fred Brooks in his famous paper *No Silver Bullet* in 1987 (Brooks, 1987) considered the difficulties in producing good software and was fairly pessimistic about whether there was any practical way of achieving major improvements in producing software. Now that we have seen what is really meant by using object-oriented techniques can we say that using these techniques are the “silver bullet” that will ensure that great software is produced?

My answer is no. For one thing, object-oriented techniques are only a stage in history. Already newer techniques to produce good software quickly are being developed, and we will surely have even better techniques in the future. Secondly, and I believe of overwhelming importance, while using object-oriented techniques will produce a major improvement in producing good software, these techniques are not enough by themselves. For these techniques to be effective it is essential that top-level management “buys-in” and provides the support and training and tools needed. It is my experience in talking to people in industry, whether programmers or management, both in New Zealand and South Africa, that there are still many companies that do not realise the importance of such things as good requirements analysis, systems analysis and design, testing, risk analysis and management and project management.

Many managers still do not arrange for requirements analysis to be done and are so keen to see the programmers writing code that they do not provide time for systems analysis or design before code is written. Thus even if the programmer wants to do analysis and design before coding he or she is not given the opportunity. This attitude is very short-sighted and dangerous and in the end tends to cost more as the code written without first being designed is often poor-quality, and not easily extended or modified, thus often leading to costly re-implementation as requirements change – and requirements always do change!

It has been shown that of all errors made in constructing information systems, 56% occur in the initial determination of the user’s requirements, and that in terms of the effort and expense required to correct such errors, 81% is spent on these early requirements definition errors! (Brown, 1997) Schach (1999) also shows that correcting an error in the implementation phase may be up to about 50 times more expensive than correcting one in the requirements phase, and that correcting an error after the product has been delivered to the customer may cost several hundred times more than correcting an error in the requirements phase. Schach also shows that on average about two-thirds of a projects costs are spent on the maintenance phase, which may include both correction of existing faults and also extension of the application. Thus it is surely worth it to ensure that the requirements are correctly determined at the beginning of the project when it is still cheap to make corrections, rather than much later at great expense?

And it is essential that management realise that if a significant amount of the time for a project is spent on systems analysis and design then it is much more likely that a good product that is flexible and extendable and easily maintainable will be delivered on time. If time is not devoted up-front to systems analysis and design then it greatly

increases the chance that at some stage of development either large sections or the entire application may need to be rewritten from scratch, significantly increasing costs and the likelihood of late delivery. If management does not provide time, training and tools for good systems analysis and design then it is unfair to expect programmers to produce good software on time.

Also, using good CASE tools and devoting time to systems analysis and design increases the chances that re-usable designs and code will be produced, thus further reducing costs in future projects. And many CASE tools now provide opportunities for “round-trip-engineering” where the tool will actually generate a code skeleton in a particular language, and also allow for code to be re-imported into the tool so the design can be extended or modified, or even so that the same design can be used to generate code in a different language. Thus using a CASE tool can produce significant cost reduction in coding. An added benefit of using CASE tools is that apart from the time saved in constantly redrawing diagrams, many of the diagrams produced are reasonably easily understood by customers and produce a useful way of checking that the customer’s requirements are being met correctly.

Ad-hoc development seldom produces re-usable code, thus meaning that often similar code or even applications are produced over and over again. Initial investment in good CASE tools and training may initially seem expensive but these costs will ultimately be recovered as good extendable and flexible software is produced quickly, usually re-using at least some components from previous projects.

Also, in my experience it seems that few companies really implement proper project management, although it appears that more companies, usually larger ones, are beginning to take this issue seriously. I believe that this is exacerbated in New Zealand by the independent attitude taken by many programmers who want to be left alone to get on and program, and by the New Zealand attitude of recognising other people’s right to independence. Unfortunately these attitudes can lead to programmers pursuing their own independent paths, which although they may actually be good ideas, have probably not been budgeted for. Nowadays when it is so important to meet deadlines, and where applications are usually developed by many programmers, it is essential to implement proper project management to ensure that projects are completed on time and within budget. Unfortunately in New Zealand there are still far too many software companies who fail to complete projects on time and within budget.

A part of project management is risk analysis and management. Again, sadly, I see that few companies do any risk analysis and management, more likely taking the attitude of “She’ll be right, mate!” This attitude can again lead to serious problems leading to exceeding budget and failing to meet deadlines as mistakes are discovered too late in the development process.

Thus I strongly recommend that to produce great software senior management must implement all of good requirements analysis, systems analysis and design, testing, risk analysis and management and project management. For those who have not done these things before and need some ideas of where to start I recommend the following books and articles (full details under “References” section). This list is not exclusive and there are many more sources of information and the list below is only a few of my favourites. In particular I am mainly suggesting some more pragmatic ways of doing things that are more likely to be adopted by small companies starting to do these things for the first time, rather than more difficult and complicated ways which will immediately be rejected as too cumbersome and expensive. However, as companies progress in these areas or get larger they may well want to move onto more comprehensive and formal methods.

Requirements analysis:	Jacobson (1999), Anderson (2001)
Systems analysis and design:	Quatrani (1998), Beck (1989), Kruchten (1999), Jacobson (1999)
Project management:	Woodward (1999), Sommerville (2001)
Risk analysis and management:	Williams (1997), Sommerville (2001)

In particular of the above list I find Quatrani (1998) an excellent introduction to the Unified Modelling Process. CRC cards (Beck, 1989) are also a good way to start with object-oriented analysis. Woodward (1999) describes the very practical evolutionary (or incremental) project management which means that each stage of the development produces a useable product for the customer, even if it only has a subset of the final requirements, unlike the “waterfall” approach where the product is only ready at the end of development. Each subsequent stage of the evolutionary project adds more features, but every stage is a useable, tested product. Object-oriented analysis, design and implementation techniques are very well suited for evolutionary project management as it is easier to produce a useable product at each stage than with functional or data-oriented approaches. This approach obviously has many

advantages, especially in meeting deadlines and budgets, as even if money and time run out there is a useable product even if it does not have all the features originally planned. Sommerville (2001) also gives useful suggestions on project management. Williams (1997) gives a fairly simple and pragmatic method of risk analysis and management that is particularly suitable for those who have not done it before. Sommerville (2001) describes some other methods of risk analysis. Schach (1999) also provides much useful material on these topics.

Conclusions

This paper has shown that object-oriented designs are much more easily implemented, extended or modified than non-object-oriented systems because of the significant advantages in using the object-oriented techniques of classes and objects, encapsulation, inheritance and polymorphism. Also, in object-oriented applications code is written in (usually) short methods that are more easily tested and debugged without having far-reaching consequences on the rest of the application. Thus object-oriented applications are more likely to be good and reliable code and are more easily maintained or extended. Good object-oriented design and proper use of inheritance and polymorphism also increase the possibility of re-use of both designs and code, thus saving on future development costs and increasing the rapidity by which new applications can be developed.

In addition this paper has suggested that to produce great software it is not only necessary to use good analysis, design and development techniques but it is also essential to use good management. This involves both top-level management who must see the importance of these techniques, and also good project management. If management doesn't "buy in" then it doesn't matter how heroic the efforts of the analysts, designers and programmers, the projects are usually doomed to failure.

However, if management makes the decision to implement good project management, including proper requirements analysis, and systems analysis and design as well as implementation and testing, then using object-oriented techniques properly will produce better software more quickly and cheaply than other methods.

References

- ANDERSON, J., FLEEK, F., GARRITY, K. AND DRAKE, F. (2001) *Integrating Usability Techniques into Software Development*, Software, Vol. 18, No 1, January/February 2001
- AURELIUS, M., *Meditations* (Book 3, Entry 11), quoted in Schach (1999)
- BECK, K. AND CUNNINGHAM, W., (1989) *A Laboratory For Teaching Object-Oriented Thinking*, OOPSLA '89 Conference Proceedings and SIGPLAN Notices, Special Issue, 24:10, October 1989, <http://c2.com/doc/oopsla89/paper.htm>
- BROOKS, F.P. (1987) *No Silver Bullet*, Computer, Vol.20, No 4, April 1987
- BROWN, D., (1997) *An Introduction to Object-Oriented Analysis : Objects in Plain English*, John Wiley & Sons, Inc.
- CAPPER, N.P., COLGATE, R.J., HUNTER, J.C. AND JAMES, M.F. (1994) *The Impact of Object-Oriented Technology on Software Quality: Three Case Histories*, IBM Systems Journal 33, No 1, 1994
- GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J. (1995) *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, Inc
- JACOBSON, I., BOOCH, G. AND RUMBAUGH, J., (1999) *The Unified Software Development Process*, Addison-Wesley Longman, Inc
- KOONTZ, R.W. (1994) *Lessons from the Experts*, A review of the August 1994 Conference on Object-Oriented Software Development at St. Thomas University, St. Paul, MN, Object Magazine, Vol. 4, No. 7, November-December 1994
- KRUCHTEN, P., (1999) *The Rational Unified Process – An Introduction*, Addison-Wesley Longman, Inc
- LARMAN, C., (1998) *Applying UML and Patterns*, Prentice-Hall PTR
- QUATRANI, T., (1998) *Visual Modeling with Rational Rose and UML*, Addison-Wesley Longman, Inc
- SCHACH, S. R., (1999) *Classical and Object-oriented Software Engineering with UML and Java*, 4th edition, WCB McGraw Hill.
- SOMMERVILLE, I., (2001) *Software Engineering*, 6th Edition, Pearson Education Limited.
- WILLIAMS, R. C., WALKER, J. A., DOROFEE, A. J., (1997) *Putting Risk Management into Practice*, Software, Vol. 14, No. 3, May/June 1997
- WOODWARD, S., (1999) *Evolutionary Project Management*, Computer, October 1999