# ObjectiveView

**The Journal for Managers and Technical Staff in OO & Component Based Software**

## Welcome to ObjectiveView

*Welcome to the first edition of ObjectiveView, a quarterly journal focusing on the commercial and technical issues surrounding object oriented and component based software development. Major features in this issue take a look at the commercial motivations for object oriented software development, and the source code structuring of large object-oriented software development projects.*

*Future issues will focus on a wide variety of topics relevant to industry based managers and developers, including product overviews (OO databases, ORBs, etc.), industry news, in-depth reviews of OOA/D techniques (use cases, object modelling, sequence diagramming, etc.), case studies, and so on.*

*We hope you'll enjoy ObjectiveView and look forward to hearing from you.*

## Structuring Large OO Software Projects

***For:* Project Managers, Technical Managers and Development Staff**

*As software applications grow in size and complexity the structure of the source code becomes an issue in its own right. Robert C. Martin lays down the rules for effective OO packaging and source structure.*

See Page 17

## Design Focus

***For:* Object Designers & Senior Technical Staff**

*Tips and discussion of object oriented design issues.*

See Page 15

## The Commercial Case for OO

***For:* Business, Project and Quality Managers**

*What are the commercial motivations behind OO? What benefits should you hope to gain from the move to an Object Oriented approach to software development?*

See Page 5



## Subscription Details

FOR YOUR **FREE SUBSCRIPTION** TO OBJECTIVE VIEW
Email: objective.view@ratio.co.uk Tel: Kate Evans on 0181 579 7900 or Write to: The Editor, Objective View, c/o The Ratio Group, 2nd Flr, 17/19 The Broadway, Ealing, London W5 2NH, or simply complete the box below and post or FAXBACK (0181 579 9200) to The Editor at the above address)

IN ALL CASES, PLEASE SUPPLY THE FOLLOWING INFORMATION:

Name ..... Daytime Tel.No.....
Job Title..... Email Address.....
Co..... Preferred delivery: (Please tick)
Address..... Hardcopy by post
..... Zipped Word document by email
.....
Postcode.....

*We would appreciate your input – tell us about major areas of interest to you/your company, i.e. what topics would you like included in our future additions, and keep us posted on any comments you might have on articles in this and future editions.*

## Object News for First Quarter, 1998

Events and Trends in Object Technology Industry
By Dion Hinchcliffe, Editor, Object News (www.objectnews.com)

This first quarter of 1998 was an exciting one for the object technology community with events demonstrating this is an industry in the midst of huge change with the rapid shift to Java and UML and an industry with increasing maturity with tools in high version numbers and stable standards (Java excepted). Notable news included events such as the continued maturity of the middleware business, a sharp increase in the adoption rate of UML along with important new releases of UML tools, the increasingly intertwined relationship between CORBA and Java, and the hugely successful JavaOne conference which reinforced the onslaught of Java into all aspects of the object industry.

**Here are some of the significant events in the object technology industry in Q1 1998**:

## Common Object Request Broker Architecture (CORBA) News

CORBA continued to enjoy an improvement in both tools and standards in Q1. The first real implementations of the all-important (at least to the enterprise user) CORBA Object Transaction Services began trickling into the market. Both Visigenic (www.visigenic.com) and IONA (www.iona.com) at least shipped beta versions of OTS in Q1 and are now shipping fully compliant OTS services for their ORBs. In February, the Object Management Group (www.omg.org) released the specification for CORBA 2.2 which includes some important new additions to the standard including Portable Object Adapters, IDL/Java mapping and a few other minor improvements. In February, ORB vendor Visigenic (www.visigenic.com) was acquired by software maker Borland International (www.borland.com) in a surprise move generally applauded by analysts. Visigenic along with IONA (www.iona.com) remain the top two CORBA ORBs with both becoming very active in providing products to meet the biggest trend in CORBA during Q1 1998: ORBs written in 100% pure Java along with tools to support CORBA object implementation in Java.

## Component Object Model (COM) News

COM, Microsoft's (www.micrsoft.com) alternative to OMG's distributed object model, had no revisions during Q1 1998 although information continued to come out about COM+, the successor to COM which now looks to provide plug-and-play development for all implementation languages, advanced run-time services, and integration of Microsoft Transaction Server into COM. In fact, very little was heard from Microsoft outside of a few noisy announcements about CORBA/COM bridges, which were announced by IONA and Expersoft (www.expersoft.com) early in the quarter. Also announced in Q1 was that Microsoft would maintain a reference version of *COM for Solaris*, the most popular flavour of UNIX in the IT community, which would receive the benefit of Microsoft's full distribution and enterprise technical support. COM remains quite popular as a component technology for departmental applications but it remains unclear how much COM for Solaris, availability of CORBA/COM bridges, and the forthcoming COM+ will help alleviate the Windows-only stigma that is holding it back from enterprise-level acceptance. It is anticipated that few new COM announcements will be made until a COM+ beta release, which is estimated to coincide with the Windows NT 5 beta at the end of Q2 1998 spurring speculation that NT 5 and COM+ may be tightly integrated. The lull-before-the-storm pattern is typical when Microsoft is busy readying a major new release of software.

## Unified Modelling Language (UML) News

UML made significant inroads in software development shops since adoption by the Object Management Group in November 1997. This standardisation as well as a growing familiarity with the UML and improved tools has led to estimates that upwards of 49% of object modelling notation users expect to adopt UML this year as their primary notation (Object Magazine Online questionnaire.) This informal poll is somewhat in contrast with the Cutter Corp.'s (www.cutter.com) Q1 1998 report titled The Corporate Use of Object Technology (www.cutter.com/itgroup/reports/corpuse.Htm) which cites that 15% of all object modelling users were utilising UML in December, 1997

but noted that "*almost all indicate that they will be transitioning to UML in the course of 1998*

In February, 1998 Rational (**www.rational.com**) the leading UML modelling vendor and home of the so-called Three Amigos who created UML, released a new iteration of their flagship UML tool which they dubbed Rose 98. Rational Rose 98 is a significant upgrade to the Rose product line and includes full support for all nine UML diagrams for the first time. Rose 98 was an important release for Rational since it's feature list (if not its market share) was beginning to come under fire from rival vendors boasting products that could seriously compete with Rational, a relatively new phenomenon for Rational.

## Java News

Java was apparently on Internet time in Q1 1998 with many new announcements, initiatives, products, and JDK releases. The biggest event by far was the now famous JavaOne conference, which brought together the best and brightest in the industry for one of the most exciting conferences of the year. Scores of announcements for dozens of new Java products including Borland's JBuilder 2.0 and Aonix's STP round-trip development for Java, new APIs such as Embedded Java and Personal Java, and the amazing Java VM on a ring from Dallas Semiconductor. Many other notable events in Java community in Q1 1998 that will have significant impact in the years to come on Java application designers and developers:

See http://java.sun.com/javaone.

- JDK 1.1 became the de facto standard for Java applications.
- Enterprise JavaBeans specification 1.0 and development kit was released to great industry fanfare. Enterprise JavaBeans are server-side JavaBeans that are essentially transactionally-aware business objects. This allows Java to be a major player in the middle-tier.
- Microsoft was forced to stop using JavaSoft's Java-compatible logo as a result of a court injunction requested by Sun.
- Microsoft released Visual J++ 6.0 in early beta with the Windows Foundation Classes, a non-portable Windows-only application framework. The Java community has been unhappy with this development as it completely breaks Java's write-once run-anywhere model.
- Java Foundation Classes 1.1 were released in late February and provides the new look-and-feel for Java
- Sun made available the pre-release specification of the Java Transaction Service for public comment. The JTS will "[ensure] interoperability with sophisticated transaction resources such as transactional application programs, resource managers, transaction processing monitors and transaction managers." This will be vital for Java to be a major contender in middle-tier application services.

Sun made available a detailed technology roadmap (http://java.sun.com/pr/1998/03/pr980324-06.html) that will allow organisations to better determine future directions for Java technologies and products, this much needed document will help companies better plan their technology strategies and enterprise object architectures as they pertain to Java. Java continues to be one of the more exciting emergent object technologies. Much more than just an object-oriented language, Java is becoming a major technology platform for the entire enterprise including front-end applications, application servers, component software, distributed systems, and much more. Although Java still has hurdles to overcome, it promises to be one of the most exciting places for object technology. Look for the language to begin to mature by Q3 1998 with the first minor point releases of JDK 1.2 and expect Enterprise JavaBeans to become a major technology as big as Java itself by Q4 1998.

## Method and Process News

Ever since Ivar Jacobson, Grady Booch, and Jim Rumbaugh joined forces to created the Unified Modelling Language, there has been much anticipation about when they would come out with a Unified Process. The UML is merely an object notation and does not describe the actual step-by-step process of design and developing object-oriented software itself. The Three Amigos decided it was better to create a process-neutral modelling language first that was not tied to any existing process and could replace most of the notations in use (such as OMT, Booch, etc) without affecting the existing development process. By not tying the notation to a process, UML could become the Switzerland of notations, which it has in fact become. But all the recent activity in the UML world obscured the fact that no

Unified Process has emerged from the Three Amigos, not even a proposal.

This silence was apparently broken in a back of the magazine article in April, 1998 Object Magazine www.objectmagazine.com) by Rational's Ivar Jacobson that boldly pronounced that Objectory, Rational's commercial process would be the Unified process Though many debate the wisdom of a one-size-fits-all process, Objectory is as close as it gets and we hope to hear more in the near future from the Three Amigos to see how this will turn out. Look for a forthcoming book on Objectory from Jacobson in Q3 of 1998.

## Q1 Object News Summary

C++ in Q1 1998 remained the most popular language for developing object-oriented applications. The Cutter Corp. estimates that this will change by Q4 1998 and Java will become the most popular language. CORBA remained the most popular distributed object model in Q1 with DCOM only representing half of CORBA's market share. Interest in object technology (OT) increased in Q1 1998 and will continue to increase throughout the year (Cutter Corp). The root cause of increased OT acceptance is most likely due to the fact that Internet development tools are primarily object-based, Java is becoming the most popular development language in the world, and CORBA, UML, and COM are all maturing and gaining mind share with designers, developers, and architects. Look for Q2 1998 to bring us JDK 1.2, COM+ beta, the first implementations of Enterprise JavaBeans, UML 1.2, and an increasingly popularity in CORBA/Java solution

*Don Hinchcliffe is a senior consultant with Object Systems Group (www.osgcorp.com) and Editor of Object News (www.objectnews.com), a daily object-oriented newsletter on UML, COM, CORBA, and methodology issues. Mr. Hinchcliffe can be reached at dhinchcliffe@objectnews.com*

# The Commercial Case For Object Oriented Software Development

***Mark Collins-Cope discusses the commercial motivations for adopting an object oriented approach to software development***

## Introduction

This article looks at the factors influencing the cost of software development and maintenance, and explains how by adopting the OO approach to software development software costs can be reduced. When companies first venture into developing bespoke software systems, they are often surprised that the cost of the software development is two or three orders of magnitude greater than the cost of the hardware on which the software will run. And that's even before they've considered the cost of maintaining their systems. To explain the reasons for these costs it is necessary for us to take a step back and look at the issues underlying the economics of software engineering, looking at the two main areas of costs:

- those that are incurred during the development of systems,
- those that are incurred during the maintenance of systems.

Finally, this article takes a look at the more recent trends towards distributed systems, and the costs associated with these

## Development Costs

### Low Level Building Blocks Equals High Cost Of Development

There is no getting away from the fact that the development of large software systems is one of the most complex engineering feats achieved by man. Programming languages offer infinite flexibility in the way software

may be built. If you ask two software developers to write the same system, it is quite likely that they will come up with *completely* different implementations to meet the same set of requirements. One of the reasons for this is that the building blocks provided by traditional programming languages are at such a low level. Writing a piece of software with a procedural programming language is rather akin to building a suspension bridge out of matchsticks and a tube of superglue: the design possibilities are infinite but, assuming you manage to build your bridge, you're likely to end up with a maintenance nightmare!

The addition of facilities to assist in the structuring of software has been one of the major focuses of programming language improvement over the years, however procedural programming languages still leave much to be desired in this area. As the complexity of the software systems we develop grows (a trend which is showing no sign of abatement - note the more recent moves towards GUI, client/server and three-tier systems) the ability to manage and organise the complexity of software is becoming even more of an imperative than it is at present. The Object Oriented approach to software development offers a route out of this quandary by providing a mechanism with which to build the higher level building blocks necessary to manage this complexity: Objects.

An object is a well defined software building block that provides a set of services (its *methods or operations)* to its users (i.e. *other* software developers) in such a way that they can use these services without having to worry about exactly how they work. Furthermore, OO programming languages enable the building block provider to define the structure of the object (its *class*) leaving it to the user of the *class* to create as many objects of any particular type as they need. To see the benefit of this, consider a high street banking system. The system will manage thousands of accounts, but the behaviour of each account will be identical, providing facilities to: record debits; record credits; print statements; and calculate monthly charges

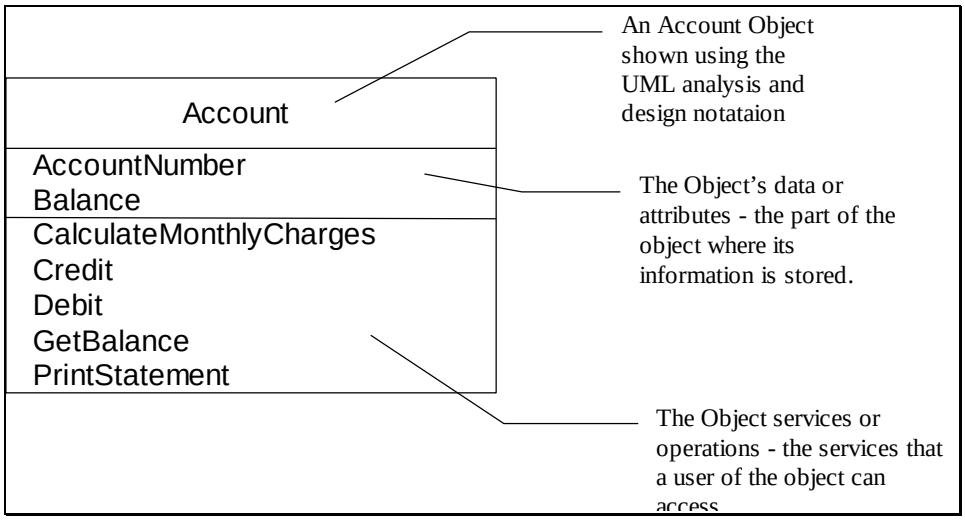



**Figure 1 - An account object using the UML notation**

Having the account as a fundamental system building block will obviously be of great benefit in such a system, and OO programming languages enable us to build such objects by defining an Account class enabling the application developer to create instances of these Accounts (a.k.a. objects) as is necessary for the particular application they are developing. Note that this approach has significant benefits over the more traditional Cut'n'Paste approach to software development, where an area of code is copied when similar functionality is required in another part of an application, not the least of which is that it doesn't involve copying a multiplicity of bugs around the application as well. The OO approach, when adopted using appropriate development methods, encourages re-use of code within the application. Since the number of bugs within a system is generally proportional to the number of lines of code written, any approach which reduces the number of lines of code written (and designed, and debugged, and tested, etc.) offers *cost benefits*, as well as guaranteeing more consistent behaviour within the application.

On the related subject of code re-use across applications (such as sharing the account

object described above across many applications), Jeffrey S. Poulin reports in his book *Measuring Software Re-Use (Addison Wesley)* a number of impressive statistics, some of which are detailed below:

- Nippon Electric achieved 6.7 times higher productivity 2.8 times better quality through a re-use programme
- GTE Corporation saved $14M in costs of software development with re-use levels of 14%
- Toshiba saw a 20-30% reduction in defects per line of code with re-use levels of 60%
- A study of nine companies showed re-use led to 84% lower project costs (!), cycle time reduction of 70%, and reduced defects.

## *Experience is an Expensive Commodity*

Traditional software does not encourage code re-use, for the reasons already discussed in this article (lack of appropriate building blocks, etc.). Obviously once an account object has been defined, it can be re-used repeatedly, with the consequent cost savings. However, the concept of re-use (*equals reduced costs*) can be applied at the analysis and design levels as well as at the coding level. One relatively recent innovation in the world of OO is the emergence of design and analysis ***patterns*** as an area of interest in its own right. Patterns enable the experiences of expert developers to be captured in a form that is easily communicated to less experienced OO developers, enabling them to shortcut the 'learn from your mistakes' approach that their more experienced colleagues were forced to adopt.

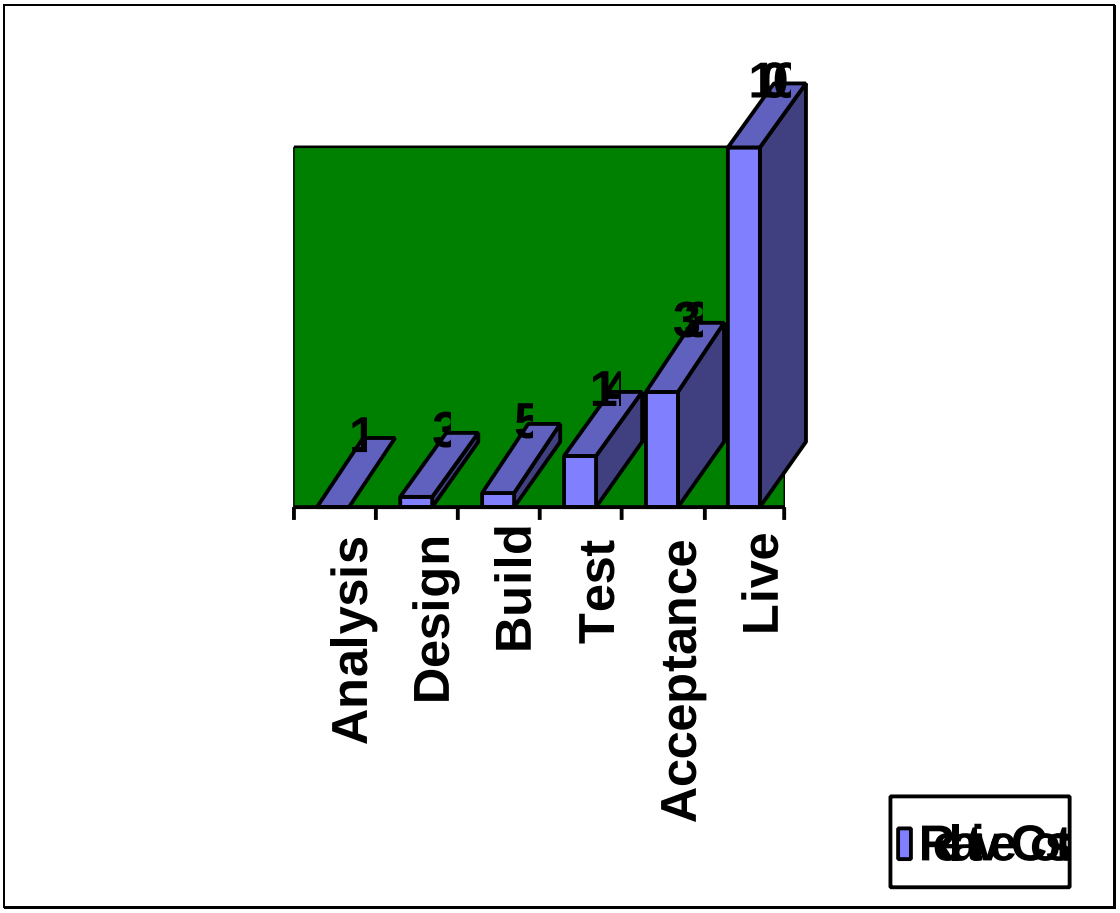



**Figure 2 - Use of a standard design pattern ("strategy") to provide database independence**

If the banking system we discussed earlier is supplied by a product company to a variety of banks, one problem they will face is that each bank will have its own preference for relational databases. To deal with this problem, the product company must provide a framework which removes their system's dependency on the database. This somewhat common software engineering problem maybe addressed by the use of a standard design pattern culled from the experiences of many developers (and documented along with 20 or so other patterns in an excellent book called 'Design Patterns' by Gamma et. al.)

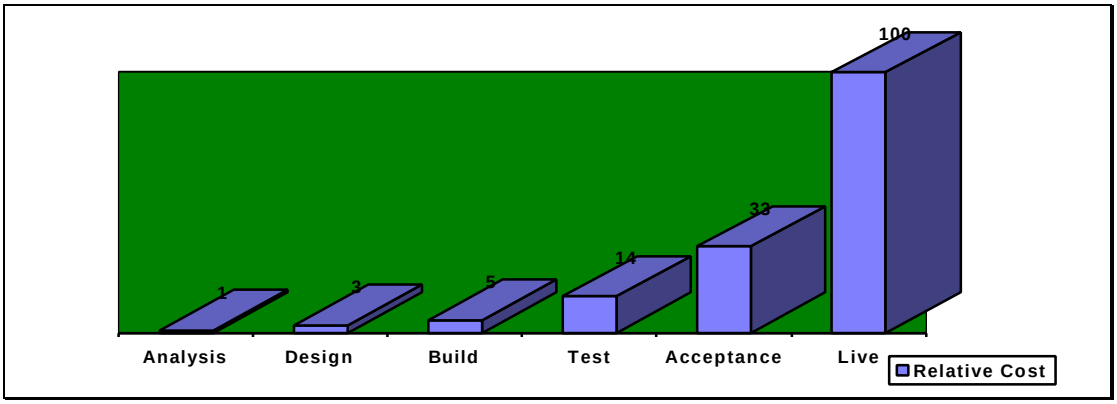



**Figure 3 - Relative cost of fixing the same problem across a project lifecycle**

The statistics shown in Figure 3 - showing the relative cost of fixing the same problem during different phases of a project lifecycle - have been available for many years, however the costs associated with late fixes are still common - especially on larger projects. Such projects often exhibit what is euphemistically known as 'hockey stick' behaviour, whereby all seems to be well and the project is apparently coming in on schedule, until late in the day when fundamental design errors or integration problems are revealed and the schedule goes out of the window. Whilst not removing the possibility of such problems completely, the systematic use of OOA/D techniques such as those provided by UML can substantially reduce the chances of such errors occurring. Three factors contribute towards this:

### *1. OOA/D notation makes design decisions explicit*

It is extremely difficult to represent the design of a procedural program without actually writing the code - by which time any problems are embedded into application. The UML OOA/D notation offers a means by which the detailed design of an (OO) application (i.e. its classes and their inter-relationships) can be made explicit - and hence can be **reviewed** by more experienced staff during the design process, thereby avoiding costly mistakes. Figure two gives an example of UML in action - showing an object model. Further notations are available, however, detailed discussion is beyond the scope of this article.

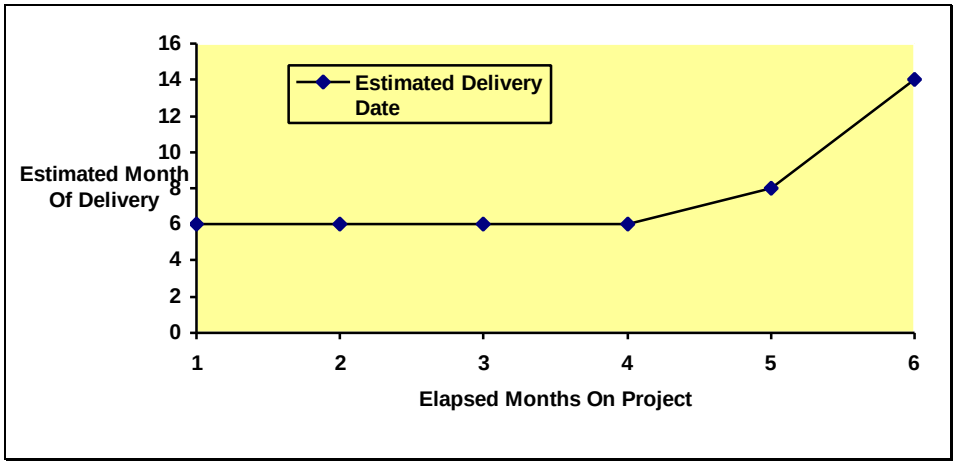



**Figure 4 - The hockey stick project –major costs appear late in the day**

## 2. A good OOA/D process will use the facilities of UML to apply consistency and completeness cross-checks

Used with an appropriate process (i.e. work process) UML can provide cross-checks to enable designers to be very confident that the design they are proposing will actually implement the required system functionality. The reason this is possible is that there are logical 'hooks' between the design notations provided within UML that complement and consolidate each other, in much the same way that an architectural plan would be complemented and consolidated by its corresponding electrical and plumbing plan if one were designing a building.

## 3. A good OOA/D process will use the facilities of UML to encourage code re-use

Note also that the use of an appropriate process with UML will also encourage intra- and inter-project re-use of code because it will make the *opportunities* for code re-use visible, and therefore reduce the number of potential bugs in the application. This is especially true on multi-person projects where such opportunities are often lost through lack of team communication.

## Maintenance Costs

Figures for the relative proportion of project costs that are consumed during system maintenance vary, however it is not uncommon to see estimates in the region of **70%.** Reducing the cost of maintenance is therefore a vital factor in the economics of software development. In this section, I take a look at some of the reasons that underlie maintenance costs.

## Traditional Software Is Inflexible and Expensive To Change

A further problem with traditional software is its lack of flexibility once it has been written. One of the major causes of this inflexibility is the ever increasing number of inter-dependencies that grow between different parts of the system. In this respect, procedural programming languages have two main components: the data and the lines of code. The code tells the application what to do, and data provides the raw information on which to do it. The lines of code therefore become inherently dependent on the data which they use. As systems grow in size, these dependencies become increasingly unmanageable: a change in one part of the system may well have an impact on many hundreds of other parts of the system. Introducing a small change often leads to a large number of bugs in disparate parts of a system, with all the associated costs.

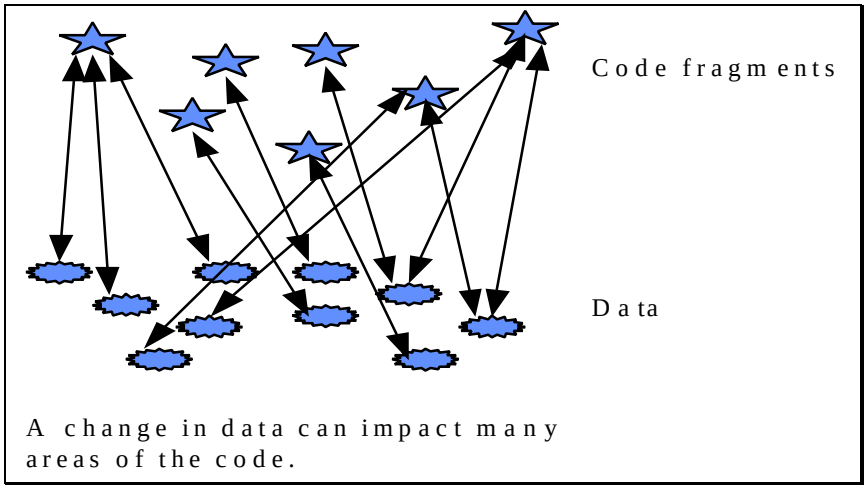



**Figure 5 - Code and data inter-dependencies become unmanageable in large systems**

In effect replacing one length of metal on our suspension bridge causes problems hundreds of feet away. As each bug is fixed, yet further bugs may be introduced, and so the process continues. Eventually such systems become unmaintainable, and must be rebuilt from

scratch. Object Oriented systems reduce the problems associated with inter-dependencies by packaging the code and the data that it uses together into objects. This localises the inter-dependencies between code and data to single objects, and hence offers substantial cost savings during software maintenance. But this is not the whole story: Object Oriented systems go much further than this in reducing the cost of software enhancements. Object Oriented languages have built-in facilities (known as *inheritance* and *polymorphism*) to enable new objects to be added to the system perhaps years after it had been originally written, without having to change the original system at all.

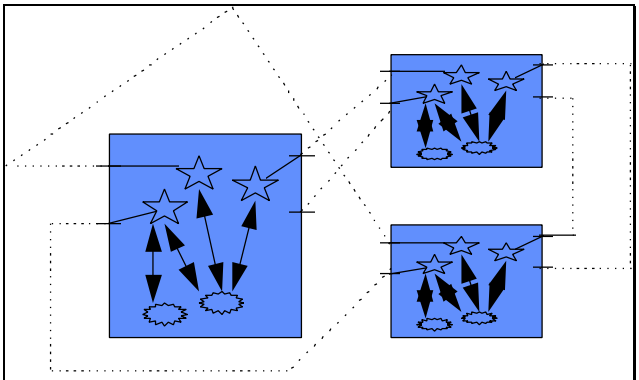



**Figure 6 - Object oriented systems restrict the impact of change**

To see how this might work, let's go back to our banking system example. Our account object provided facilities for: recording debits; recording credits; printing statements and calculating monthly charges. Many banks have different types of bank accounts, and whilst the crediting, debiting and printing of statements may be common to all types of account, the calculation of charges is very likely to vary.

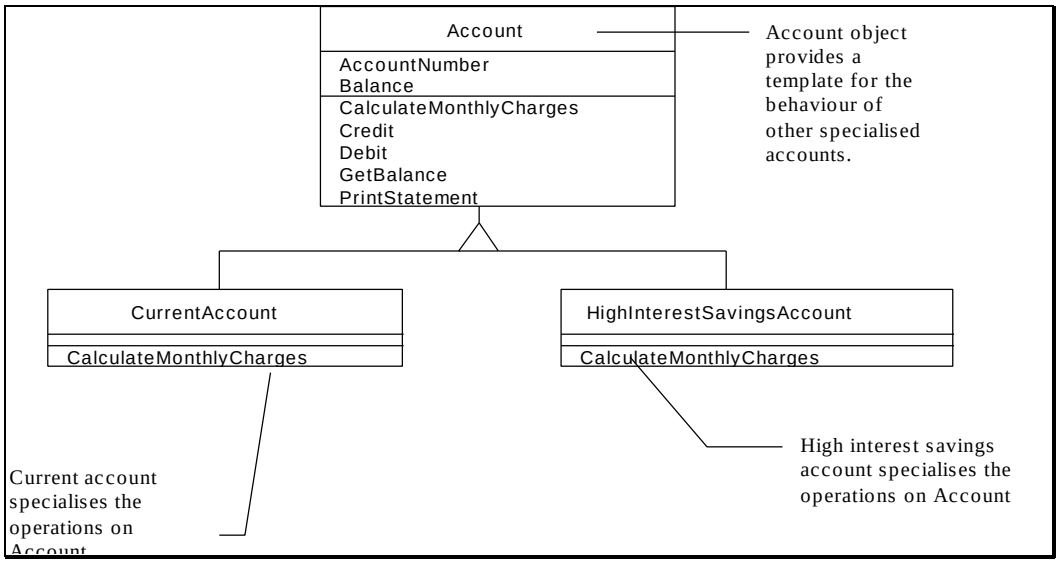



**Figure 7 - Account object template from which more specialised objects can be derived**

To deal with this, Object Oriented systems enable us to define a 'template' objects, which Outline the services provided by all accounts, and then to refine special accounts which may deviate in their implementation of specific services. The banking system will define a template Account Object which will enable special refined accounts (for example: 'high

interest savings' accounts; 'current' accounts) to vary the implementation of some or all of the services. Thus a 'high interest savings' account would not apply charges at all whilst a 'current' account would undoubtedly apply a set of exorbitant charges for the privilege of letting us get at our own money. Let us suppose our banking system has been built and usefully in service for a couple of years. One day, someone in the marketing department dreams up the new concept of the "save and borrow flex-account". Needless to say, this account has a whole different approach to charges, and even allows customers to become overdrawn without first asking permission (at a price, of course). The banking system obviously needs to be updated to deal with the new type of account. To do this, the Object Oriented developer refines a new type of account from the basic template. The 'save and borrow flex-account' shares the original implementation of the crediting and printing statement facilities, but redefines the services for debiting and calculating charges to take account of its individual peculiarities. The new account object thus defined is 'plugged into' the existing system, and hey presto, everything works as if the 'save and borrow flex-account'had been part of the original system in the first place.

## *Traditional Software Structure Doesn't Reflect The Real World*

One final problem with traditional software. Its structure (or lack of it) means that it doesn't reflect the problem domain (real world) to which it applies. This lack of continuity between the problem domain and the software structure means that often, what seems to the user of the software to be a small change in the systems functionality frequently costs a disproportionate amount to implement. Whilst most users would realise that turning a banking system into an aircraft maintenance system would be a costly exercise, they may be somewhat dismayed (to say the least), that adding a new type of account is going to cost half as much again as the original system. As we have already seen (see figure 7), a good Object Oriented system, produced using the appropriate Object Oriented Analysis and Design (OOA/D) techniques, are structured such that the program code written will have a strong relationship to the real world situations it models. Rather than rewriting half of the system, adding a new account becomes a simple operation.

## *Costs Into The Future*

So far, this article has argued the commercial case for Object Oriented software development by showing how it improves on the engineering deficiencies that underpin the economics of traditional software development. As the new millennium approaches, the trends of systems development will become more and more focused on client/server and three-tier systems: enabling users to access their systems from their desktop PCs, whilst also providing the benefits offered by monolithic systems (shared data, shared facilities, greater computational power, etc).

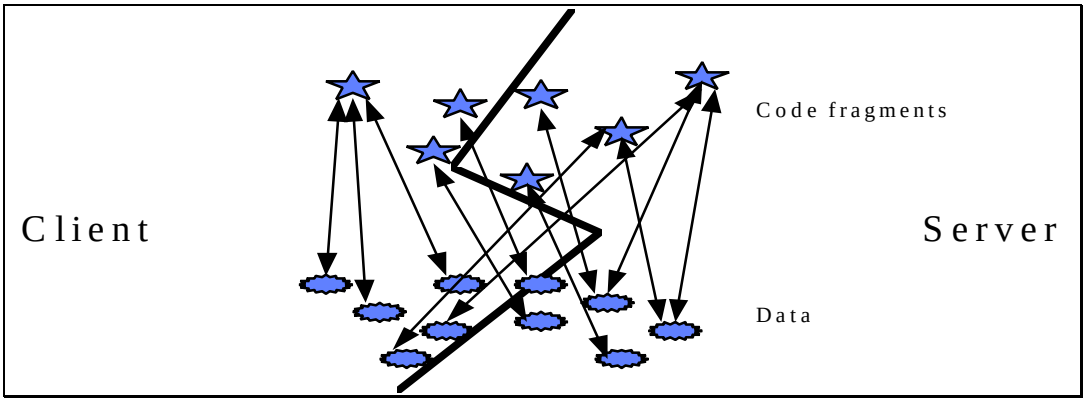



**Figure 8 - Trying to distribute a traditional application can be nigh on impossible**

Whilst developing current systems is merely uneconomic using traditional software development approaches, developing multi-tier systems can become nigh on impossible.

The building blocks provided by Object Oriented systems provide the basis by which systems may be distributed: the facilities provided by an object on one machine may be

accessed via the object's interface (its services or operations) from another machine. The object comes to the rescue by providing the building block upon which inter-machine communication can be based. In fact, although extra design work is necessary, it is interesting to note that the problem domain structure of a well analysed system (called the problem domain object model) will remain largely unchanged regardless of whether the system is implemented as a monolithic, client/server or three-tier system.

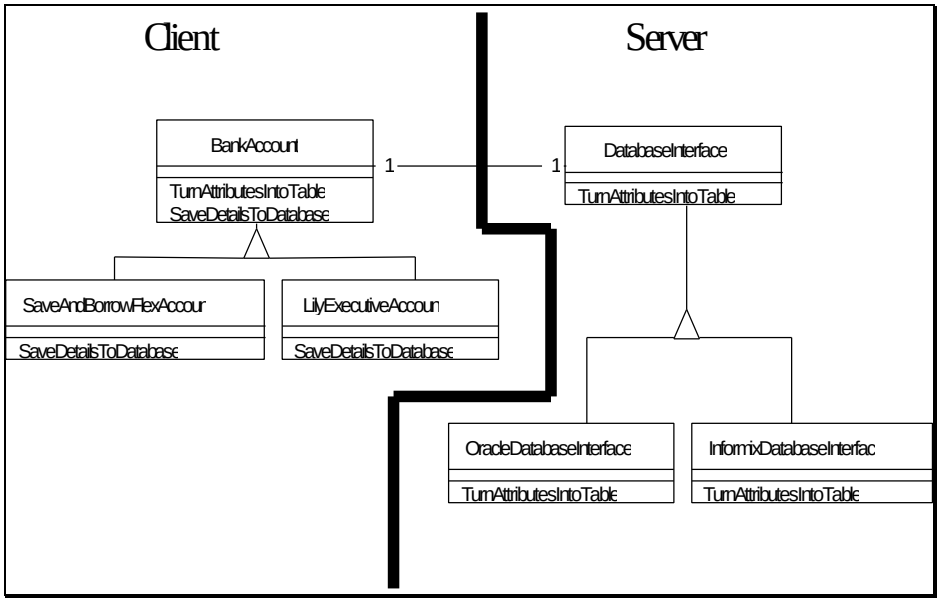



**Figure 9 - Objects provide a natural mechanism on which to base distributed systems**

## *The Millennium Bug*

No article on software costs would be complete without at least a brief (although in this case slightly tongue-in-cheek) look at the millennium bug. As I'm sure most of the world now knows, the millennium bug is caused by systems treating dates as two-digit numbers - and consequently failing when the year turns from '99' to '00'. Had such systems been developed using an Object Oriented approach, the date would have been a fundamental system object, providing services to set the date value; compare dates; print the date in a certain format, etc. Even if the designer of this object had had the lack of foresight to store dates as two digits, this would have been fixed by simply modifying the internals of the date object and relinking the system (which of course used the same date object whenever a date was needed) - thus solving the whole millennium bug problem in less than half a days work

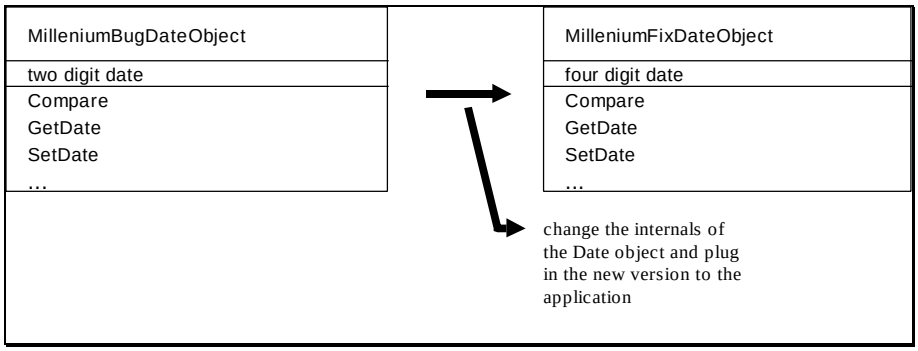



**Figure 10 - The Millennium Bug is fixed in half a day!!!**

## Conclusions

This article has looked at the costs associated with software development and how an Object Oriented approach to software development can help to reduce these costs. To summarise, object oriented software development can:

- provide higher level building blocks that *reduce costs* by encouraging re-use and reducing the inter-dependencies with a piece of software that often underlie high maintenance costs
- help in the transfer of knowledge between more and less experienced software developers through the use of 'patterns' *reducing the cost* of learning
- provide a framework to ease the extension of system, *reducing the cost* of enhancements
- assist in bringing the *cost of changing* systems more in line with users' expectations.

For larger projects, the use of industry standard analysis and design notation (UML) *in conjunction with an appropriate development process* can also *reduce the cost* of communication errors between analysts, designers and developers - the same notation is used by all aspects of the development team.

**Some words of warning**. Although Object Orientation would not be possible without programming languages that support it (C++, Java, etc.), there is much more to the commercially effective use of OO than simply changing programming languages:

- staff must undergo a fundamental change in mindset to use OO successfully, in particular 'not invented here' syndrome may often lead to some of the commercial benefits of OO being lost (it is often much more cost effective to buy standard objects than to build them yourself)
- OO must be used at all stages of the project lifecycle - there is little point using an OO development language at the end of a traditional analysis and/or design cycle
- management must 'buy-in' to the change to the OO philosophy, and in particular must set **the commercial objectives of the change** - the reduction of software costs - and ensure during analysis/design reviews that the appropriate decisions are being made with respect to these objectives
- migration to OO must be structured and planned to ensure its effectiveness.

There are also some tricky dilemmas to overcome: the software needs of one project may be different to the needs of the organisation as a whole - extra costs will be involved if the objects developed in one project are to be used on a cross-organisational basis; the short and long term objectives of software development are often at odds with each other - getting the project completed (on time and to budget) in the short term vs. designing in the hooks necessary for to cut down longer term enhancement costs.

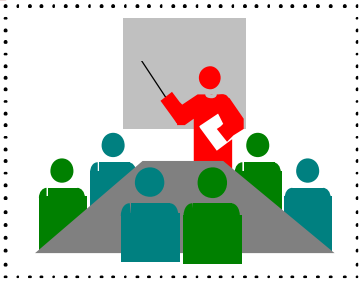
Many commercial products are now jumping on the 'object' bandwagon. Be sure to **check them all out thoroughly**. If they don't provide the facilities discussed in this paper (many don't) then **you won't get the commercial benefits**.

Further information on all of the topics discussed in this article (including other whitepapers)can be found on the Ratio Group website at www.ratio.co.uk.

.

*Mark Collins-Cope is Technical Director of Ratio Group Ltd., a consultancy and training company specialising in helping companies migrate effectively to an OO approach to software development. For further information contact Kate Evans of Ratio Group on 0181 579 7900 (fax 0181 579 9200, email katee@ratio.co.uk)*

# Ratio Group 1998 Public Course Schedule



| June | | | July | | |
|---|---|---|---|---|---|
| **Dates** | **Course** | **Cost** | **Dates** | **Course** | **Cost** |
| 1-5 | OO C++ Programming Workshop | £1,195 | 6 - 10 | OO C++ Programming Workshop | £1,195 |
| 8-12 | OOA/D Using UML | £1,250 | 13-17 | OO Java Development Workshop | £1,250 |
| 15-19 | Design Patterns | £1,250 | 20-24 | OOA/D Using UML | £1,250 |
| 22-26 | Implementing Patterns in C++ | £1,250 | | | |

| September | | | October | | |
|---|---|---|---|---|---|
| **Dates** | **Course** | **Cost** | **Dates** | **Course** | **Cost** |
| 7 | OO Concepts Overview | £300 | 5-9 | OO C++ Programming Workshop | £1,195 |
| 8 | OOA/D Using UML Overview | £300 | 12-16 | OOA/D Using UML | £1,250 |
| 9 | OO Project Management | £300 | 19-23 | Implementing Patterns in C++ | £1,250 |
| 14-18 | OO C++ Programming Workshop | £1,250 | | | |
| 21-25 | Design Patterns | £1,250 | | | |

| November | | | December | | |
|---|---|---|---|---|---|
| **Dates** | **Course** | **Cost** | **Dates** | **Course** | **Cost** |
| 2 | OO Concepts Overview | £300 | 7-11 | OOA/D Using UML | £1,250 |
| 3 | OOA/D Using UML Overview | £300 | 14-18 | OO C++ Programming Workshop | £1,195 |
| 4 | OO Project Management | £300 | | | |
| 9-13 | OO C++ Programming Workshop | £1,250 | | | |
| 16-20 | OO Java Development Workshop | £1,250 | | | |

**Please contact Ratio Group for details on:**

- **free seminars (object oriented analysis and design);**
- **special discounts (e.g. for the headstart programme);**
- **in-house courses;**
- **or further details on our object oriented computer based training programme;**

*For further information contact*
*Kate Evans on 0181-579-7900 or email Katee@Ratio.co.uk*

# OMG Analysis

***by Eric Leach, OMG's UK Representative***

1998 has been a momentous year so far for the CORBA community. Over 600 organisations throughout the world have now publicly proclaimed their adoption or use of CORBA (ourworld.compuserve.com/homepages/eric_leach). Over 70 of them reveal details of how they are using CORBA on OMG's Web site (www.omg.org). OMG membership is now some 830 strong, making it the largest software consortium the world has ever seen. Over 500 delegates from over 25 countries attended OMG's most recent Technical Meeting, held in Manchester, UK, March 30 to April 3, 1998.

Specifications which reached the final voting stage at Manchester included Currency, Business Object Component Architecture (BOCA), Notification Service (for telecoms) and Lexicon Query Service Interface (for healthcare). The Currency specification is one of the first Common Business Objects defined as part of OMG's efforts to standardise domain-specific and cross-domain software components. The BOCA specification leverages the existing CORBA and UML infrastructure to provide a framework for distributed business applications. The BOCA describes the constructs and types used to build a business object system, while an optional Component Definition Language (CDL) built on OMG IDL enhances the expression of business concepts in object definitions.

Common Business Objects such as the Currency object fit into the BOCA, where they assemble naturally into smoothly running integrated applications. BOCA concepts build upon a forthcoming OMG technical specification for distributed components, to be issued later this year. In April, OMG announced new specifications for Java-to-OMG IDL and OMG IDL-to-Java mappings that make it easier for developers to write Java applications that can run across heterogeneous environments.

## CORBA in Telecoms and Manufacturing

CORBA in telecoms is enjoying spectacular growth and OMG recently announced a collaborative agreement with the Telecommunications Information Network Architecture Consortium (TINA-C). TINA-C is an association of more than 40 of the world's largest telecom network providers.

Global One (a joint venture by Deutsche Telekom, France Telecom and Sprint) is currently trialling a CORBA-based implementation of the TINA-C architecture. The trial will run to the end of 1998. And what is likely to be the largest CORBA in Telecoms conference in the world will take place in London in June (see www.iir.co.uk/CORBA).

In April, OMG announced new specifications for Java-to-OMG IDL and OMG IDL-to-Java mappings that make it easier for developers to write Java applications that can run across heterogeneous environments. At the OMG Salt Lake City TM in February, OMG membership approved a Product Data Management (PDM) Specification. The PDM specification provides a standard interface and object framework on which to develop interoperability solutions. An imminent conference which showcases this specification is "Lifecycle '98" which takes place later this month in Birmingham, UK (www.ourworld.compuserve.com/homepages/eric_leach). OMG also announced new partners in Southern Europe and Washington, DC, USA.

Genesis Development Corporation (www.gendev.com) was appointed in February as the OMG liaison office for Southern Europe, based in Milan, Italy. It will focus its efforts in France, Italy, Spain, Portugal and Greece. In January, The OBJECTive Technology Group - The OTG (www.theotg.com) – was appointed as OMG's Government Liaison Office in the Washington DC area.

## CORBA Products Blitz

In the marketplace IONA Technologies (www.iona.com) decided to licence Microsoft's COM software. Over the years, IONA has consistently publicly denigrated Microsoft's object/component products, but has seemingly finally given in to customer pressure to try to make COM/CORBA interworking as seamless as possible - alternatively, one could take the view that Microsoft has decided to collaborate closely with the progenitors of the largest CORBA

implementation community. ICL released its elegant COM2CORBA software bridge in March, but has not seen fit to actually licence COM. In March ICL also announced the availability of a free Java ORB - J2. Also announced was a soon to be available J2 upgrade, called J3. Buying J3 will give users unique CORBA compliant secure transactional capabilities. Borland completed its acquisition of Visigenic, renamed itself Inprise (www.inprise.com/(BORL)) and gave very strong public support for CORBA.

Inprise's commitment to CORBA is important as it offers CORBA-integrated, industry leading development tools (JBuilder, Delphi and C++Builder) to desktop developers. Cisco's massive business-to-business Internetowroking Product Center (IPC) is undergoing a rewrite from C/Perl/CGI technology to CORBA with the help of Alta Software. Cisco's IPC drives the Web's premier electronic commerce site

At the UK's Butler Component Based Development (CBD) Forum meeting in April, IBM revealed more details on reference sites for its Component Broker software, including Swiss Bank and Volvo. IBM also announced it was in the final stages of implementing 9 CORBAservices.

The trend for platform providers to incorporate third-party ORBs into their products became more pronounced in 1998. Following companies such as Oracle, BEA Systems and Inprise (formerly Borland), Hewlett-Packard (www.hp.com) decided to drop its ORB Plus products in favour of IONA's Orbix. Orbix will be integrated with HP-UX this Autumn. NEC also revealed that it too would adopt Orbix in preference to its home grown ORB.

In the embedded operating software arena, Integrated Systems Inc. (www.isi.com) - the clear market leader, with over 25 million installations - announced in early May that it would CORBA-enable its RTOS software.

Highlander Communications is porting Inprise's VisiBroker for C++ to Integrated Systems' psoSystem

***Eric Leach, OMG's UK Representative***

.

# Design Focus

## Design Tips: *The Ellipse-Circle Dilemma*

**Yonat Sharon summarises discussion Object Oriented Design issues.**

### *Is a Circle a Kind-of an Ellipse?*

*Source Comp.Object.*

Maybe an ellipse is an extension of circle?
This is an instance of two more general problems:

- Should the OO model (Ellipse class and Circle class) of an application match the Real World model (mathematical concepts of ellipse and circle)?
- Does similarity between classes always implies inheritance?

**Russ McClelland wrote:**
A circle is an ellipse, ask any mathematician, the real question is can a circle be modelled in software exactly as it's defined in the real world. Yes, just because an ellipse has two foci, and a circle apparently has one, doesn't mean that we need to create an independent circle class, they both have the same attributes. It is an optimisation to remove the second focus for circles.

**Terry Richards then noted that:** Nobody says a line is an ellipse, nobody says a point is an ellipse, so why say that a circle is an ellipse?

**Roger T. was quick to reply:**
Because a circle, in every way, meets the definition of an ellipse.
That definition is:
x^2/a^2 + y^2/b^2 = 1

**Robert C. Martin** then pointed out the difference between mathematics and OOD:
No! That is the geometric definition. The OO definition of an ellipse is somewhat different.

## Class Ellipse

```
Class Ellipse
{
public:
   Ellipse(const Point& f1, const
            Point& f2);

   double Area();
   double Circumference();
   Line MajorAxis();
   Line MinorAxis();
   // and maybe some accessors and
   //mutators.
private:
   Point f1;
   Point f2;
};
```

This is just one possibility for the definition of an Ellipse in an OOPL. There are an infinite variety of others depending upon the application in which it is to be used. Now, the definition I used above cannot be collapsed into a special case as you have done below. Because the Circle ought not to have two foci. The circle ought not to have functions that retrieve the major and minor axes. We want the circle class to use algorithms that are tuned to operate on circles. We don't want to use algorithms that are tuned for ellipses and just happen to produce the right answer for circles after a lot of extra work.

## A circle is a state of an ellipse

If we were to use the notion of special cases then the following logic would apply: A circle is a state of an Ellipse. An Ellipse is really a state of a Conic Section. A conic section is really a state of a polynomial. A polynomial is really a state of a generic nth order equation. So perhaps we should have a GenericNthOrderEquation class and then circles, ellipses, paraboli ,hyperboli, lemnicati, etc, etc, would all be states of this class. But I don't think anybody would want to use such a class.

**Lex Spoon** questioned the importance of the ellipse-circle dilemma to every-day practice: Reading through some of the complicated descriptions in this thread, I can't help but think about some poor programmer who "just" wants to write a program that draws a Circle around a smiley face. How much grief should we put him through in the name of purity?

**Robert C. Martin explained:**
Don't underestimate this issue. It is not an issue of purity, it is an issue of software engineering. The whole circle/ellipse discussion is really an instance of a much broader concept. That is Programming by Contract. Every method of every class has a set of preconditions and postconditions. The preconditions must be true before the function is called, otherwise the results of the function will be undefined. The function guarantees that the postconditions will be true once it returns. Now, given function u that invokes function f in some class. Function u ensures the preconditions for f are true and then expects the post-conditions of f to be true once f returns

.

## Object Oriented Software Construction

However, if u invokes f on a base class b, what can we say about the pre and post conditions of f in d a derivative of b? Bertrand Meyer worked all this out in the mid 80s and wrote it down in a wonderful book called Object Oriented Software Construction. Meyer said that the preconditions of d::f can be no stronger than the preconditions of b::f. That is, derived classes cannot their users do to more than users of b::f are already doing. Moreover, d::f can accept fewer preconditions. This does no harm since users of b::f will still be able to call d::f. Also, d::f must conform to all of the postconditions of b::f. This ensures that functions like u can call d::f polymorphically and still get everything they need. d::f has the ability to add more postconditions if necessary, but that is just gravy as far as u is concerned. Now, consider the Circle/Ellipse problem. One of the functions of Ellipse is StretchX(x). This function stretches the ellipse by x units in the X direction. The post conditions for this function might be:

-new horizontal length is x units longer.
-new vertical height is unchanged.

These are reasonable postconditions for this function, and u can depend upon them. However, if we derive Circle from Ellipse and then modify StretchX such that it also increases the vertical height of the circle in order to keep it circular, then we have violated the postconditions of Ellipse::StretchX. This will confuse any users (u) that call StretchX on what they think are Ellipses. Of course these issues are important for all classes. Every method of every class has a set of preconditions and postconditions. And it is important to make sure that the methods of derived classes expect no more than the bases, and deliver no less than the bases. This is just another way of stating the Liskov Substitution Principle (LSP).

Compiled by Yonat Sharon from comp.object. See. http://www.kinetica.com/ootips for more ootips.

# Design Tips: Observer Pattern

## Problem: How to keep multiple views of a single object in sync with the object?

**Frank Prindle** asked: In MS Windows 95 or Windows NT, one can easily launch many independently developed applications which each display a slider/scale widget showing the soundcard volume, for example. If the volume is changed by moving any of these sliders/scales, all the other sliders/scales IMMEDIATELY reflect the change. Is there a commonly used programming method whereby, if an object has visual representations in one or more windows [...], that when the value of that object changes [...], all its visual representations are informed of, and display the new value? **From the book Design Patterns by Gamma, Helm, Johnson & Vlissides** The Observer pattern describes how to establish these relationships. The key objects in this patterns are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronise its state with the subject's state. This kind of interaction is also known as publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who observers are. Any number of observers can subscribe to receive notifications.

Compiled by Yonat Sharon from discussions on the comp.object Usenet group.
See http://www.kinetica.com/ootips for further details and more object oriented tips

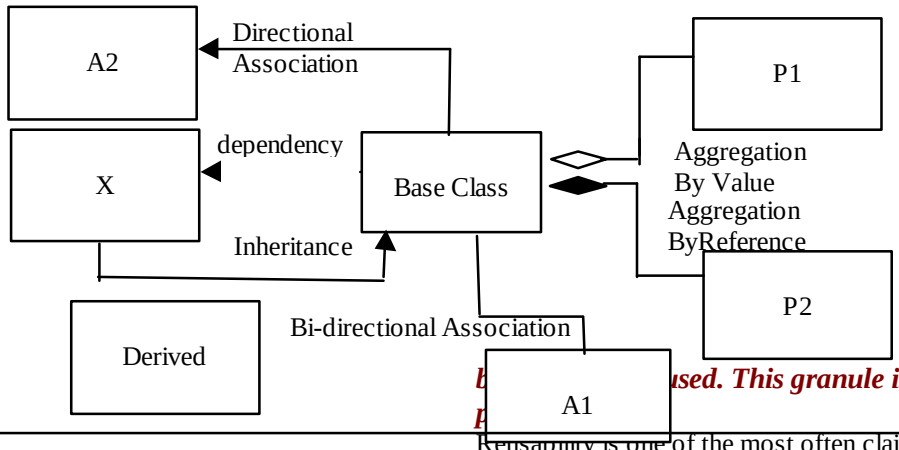
# Granularity

*As software applications grow in size and complexity the structure of the source code becomes an issue in its own right. Robert C. Martin lays down the rules for effective OO source structure and packaging*

## Granularity

As software applications grow in size and complexity they require some kind of high level organisation. The class, while a very convenient unit for organising small applications, is too finely grained to be used as an organisational unit for large applications. Something "larger" than a class is needed to help organise large applications. Several major methodologists have identified the need for a larger granule of organisation. Booch, uses the term "class category" to describe such a granule, Bertrand Meyer refers to "clusters", Peter Coad talks about "subject areas", and Sally Shlaer and Steve Mellor talk about "Domains". In this article we will use the UML 0.9 terminology, and refer to these higher order granules as "packages". The term "package" is common in Ada and Java circles. In those languages a package is used to represent a logical grouping of declarations that can be imported into other programs.

In Java, for example, one can write several classes and incorporate them into the same package. Then other Java programs can 'import' that package to gain access to those classes.




***b ...used. This granule is the p...***

## Designing with Packages

In the UML, packages can be used as containers for a group of classes. By grouping classes into packages we can reason about the design at a higher level of abstraction. The goal is to partition the classes in your application according to some criteria, and then allocate those partitions to packages. The relationships between those packages expresses the high level organisation of the application. But this begs a large number of questions.

1. What are the best partitioning criteria?
2. What are the relationships that exist between packages, and what design principles govern their use?
3. Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
4. How are packages physically represented? In C++? In the development environment?
5. Once created, to what purpose will we put these packages?

To answer these questions, I have put together several design principles which govern the creation, interrelationship, and use of packages.

.

## The Reuse/Release Equivalence Principle (REP)

***The Granule of reuse is the granule of release. Only components that are released through a tracking system can***

Reusability is one of the most often claimed goals of OOD. But what is reuse? Is it reuse if I snatch a bunch of code from one program and textually insert it into another? It is reuse if I steal a module from someone else and link it into my own libraries? I don't think so.

The above are examples of code copying; and it comes with a serious disadvantage: you own the code you copy! If it doesn't work in your environment, *you* have to change it. If there are bugs in the code, *you* have to fix them. If the original author finds some bugs in the code and fixes them, *you* have to find this out, and *you* have to figure out how to make the changes in your own copy. Eventually the code you copied diverges so much from the original that it can hardly be recognised. The code is *yours*. While code copying can make it easier to do some initial development; it does not help very much with the most expensive phase of the software lifecycle, maintenance.

I prefer to define reuse as follows. I reuse code if, and only if, I never need to look at the source code (other than the public portions of header files). I need only link with static libraries or include dynamic libraries. Whenever these libraries are fixed or enhanced, I receive a new version which I can then integrate into my system when opportunity allows. That is, I expect the code I am reusing to be treated like a product. It is not maintained by me. It is not distributed by me. I am the customer, and the author, or some other entity, is responsible for maintaining it. When the libraries that I am reusing are changed by the author, I need to be notified. Moreover, I may decide to use the old version of the library for a time. Such a decision will be based upon whether the changes made are important to me, and when I can fit the integration into my schedule. Therefore, I will need the author to

make regular releases of the library. I will also need the author to be able to identify these releases with release numbers or names of some sort. Thus, I can reuse nothing that is not also released. Moreover, when I reuse something in a released library, I am in effect a client of the entire library. Whether the changes affect me or not, I will have to integrate with each new version of the library when it comes out, so that I can take advantage of later enhancements and fixes. And so, the REP states that the granule of reuse can be no smaller than the granule of release. Anything that we reuse must also be released. Clearly, packages are a candidate for a releasable entity. It might be possible to release and track classes, but there are so many classes in a typical application that this would almost certainly overwhelm the release tracking system. We need some larger scale entity to act as the granule of release; and the package seems to fit this need rather well.

## The Common Reuse Principle (CRP)

***The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.***

This principle helps us to decide which classes should be placed into a package. It states that classes that tend to be reused together belong in the same package. Classes are seldom reused in isolation. Generally reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same package. A simple example might be a container class and its associated iterators. These classes are reused together because they are tightly coupled to each other. Thus they ought to be in the same package. The reason that they belong together is that when an engineer decides to use a package a dependency is created upon the whole package. From then on, whether the engineer is using all the classes in the package or not, every time that package is released, the applications that use it must be revalidated and re-released. If a package is being released because of changes to a class that I don't care about, then I will not be very happy about having to revalidate my application.

Moreover, it is common for packages to have physical representations as shared libraries or DLLs. If a DLL is released because of a change to a class that I don't care about, I still have to redistribute that new DLL and revalidate that the application works with it.

Thus, I want to make sure that when I depend upon a package, I depend upon every class in that package. Otherwise I will be revalidating and redistributing more than is necessary, and wasting lots of effort.

## The Common Closure Principle (CCP)

***The classes in a package should be closed together against the same kind of changes. A change that affects a package affects all the classes in that package.***

More important than reusability, is maintainability. If the code in an application must change, where would you like those changes to occur: all in one package, or distributed through many packages? It seems clear that we would rather see the changes focused into a single package rather than have to dig through a whole bunch of packages and change them all. That way we need only release the one changed package. Other packages that don't depend upon the changed package do not need to be revalidated or re-released. The CCP is an attempt to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package. This minimises the workload related to releasing, revalidating, and redistributing the software. This principles is closely associated with the Open Closed Principle (OCP). For it is "closure" in the OCP sense of the word that this principle is dealing with. The OCP states that classes should be closed for modification but open for extension. As I have disussed in a previous article that described the OCP, 100% closure is not attainable. Closure must be strategic. We design our systems such that the are closed to the most likely kinds of changes that we foresee. The CCP amplifies this by grouping together classes which cannot be closed against certain types of changes into the same packages. Thus, when a change in requirements comes along; that change has a good chance of being restricted to a minimal number of package

## The Acyclic Dependencies Principle (ADP)

***The dependency structure between packages must be a directed Acyclic***

***graph (DAG). That is, there must be no cycles in the dependency structure.***

Have you ever worked all day, gotten some stuff working and then gone home; only to arrive the next morning at to find that your stuff no longer works? Why doesn't it work? Because somebody stayed later than you! I call this: "the morning after syndrome". The "morning after syndrome" occurs in development environments where main developers are modifying the same source files. In relatively small projects with just a few developers, it isn't too big a problem. But as the size of the project and the development team grows, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code trying to make it work with the last changes that someone else made. The solution to this problem is to partition the development environment into releasable packages. The packages become units of work which are the responsibility of an engineer, or a team of engineers. When the responsible engineers get a package working, they release it for use by the other teams. They give it a release number and move it into a directory for other teams to use. They then continue to modify their package in their own private areas. Everyone else uses the released version. As new releases of a package are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release. Thus, none of the teams are at the mercy of the others. Changes made to one package do not need to have an immediate affect on other teams. Each team can decide for itself when to adapt its packages to new releases of of the packages they use. This is a very simple and rational process. And it is widely used. However, to make it work you must *manage* the dependency structure of the packages. There can be no cycles. If there are cycles in the dependency structure then the "morning after syndrome" cannot be avoided. I'll explain this further, but first I need to present the graphical tools that the UML 0.9 uses to depict the dependency structures of packages. Packages depend upon one another. Specifically, a class in one package may `#include` the header file of a class in a different package. This can be depicted on a class diagram as a dependency relationship between packages (See Figure 1).


**Figure 1 - Dependencies between Packages.**

Packages, in UML 9.0 are depicted as "tabbed folders". Dependency relationships are dashed arrows. The arrows point in the direction of the dependency. That is, the arrow head is placed next to the package that is being depended upon. In C++ terms, there is a `#include` statement in a class within the dependent package that references the header file of a class in the package being depended upon.

Consider the package diagram in Figure 2. Here we see a rather typical structure of packages assembled into an application. The function of this application is unimportant for the purpose of this example. What *is* important is the dependency structure of the packages. Notice how this structure is a *graph*. The packages are the *nodes*, and the dependency relationships are the *edges*. The dependency relationships have direction. So this structure is a *directed graph*.

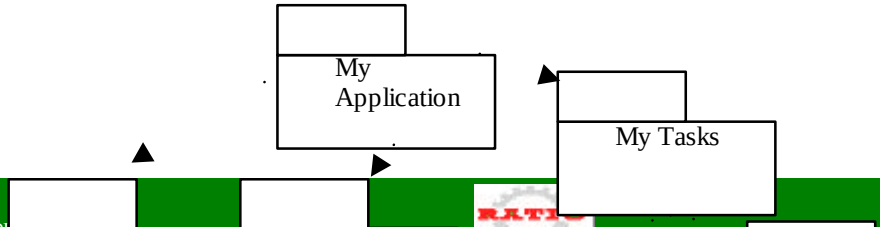


**Figure 2 Package Diagram without Cycles**

Now notice one more thing. Regardless of which package you begin at, it is impossible to follow the dependency relationships and wind up back at that package. This structures has no cycles. It is a *directed acyclic graph*. (DAG). Now, notice what happens when the team responsible for MyDialogs makes a new release. It is easy to find out who is affected by this release; you just follow the dependency arrows backwards. Thus, MyTasks and MyApplication are both going to be affected. The teams responsible for those packages will have to decide when they should integrate with the new release of MyDialogs. Notice also that when MyDialogs is released it has utterly no affect upon many of the other packages in the system. They don't know about MyDialogs; and they don't care when it changes. This is nice. It means that the impact of releasing MyDialogs is relatively small. When the engineers responsible for the MyDialogs package would like to run a unit test of their package, all they need do is compile and link their version of MyDialogs with the version of the Windows package that they are currently using. None of the other packages in the system need be involved. This is nice, it means that the engineers responsible for MyDialogs have relatively little work to do to set up a unit test; and that there are relatively few variables for them to consider. When it is time to release the whole system; it is done from the bottom up. First the Windows package is compiled, tested, and released. Then MessageWindow and Mydialogs.These are followed by Task, and then TaskWindow and Database. MyTasks is next; and finally MyApplication. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

## *The Effect of a Cycle in the Package Dependency Graph*

Let us say that the a new requirement forces us to change one of the classes in MyDialogs such that it #includes one of the class headers in MyApplication. This creates a dependency cycle as shown in Figure 3.

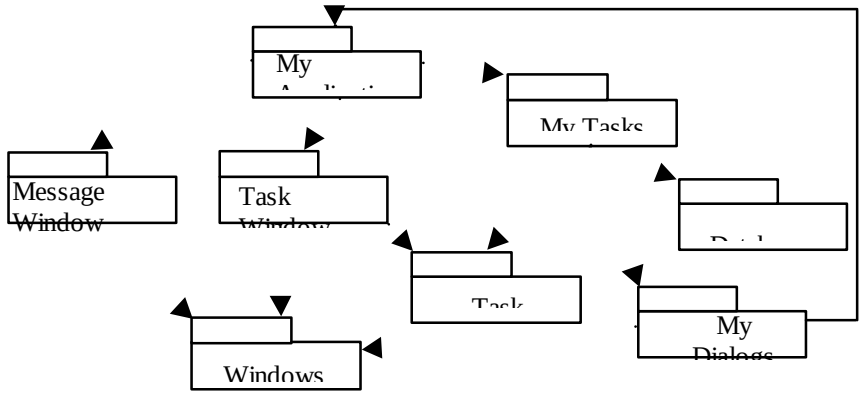


**Figure 3 Package Diagram *with* Cycles**

This cycle creates some immediate problems. For example, the engineers responsible for the MyTasks package know that in order to release, they must be compatible with Task, MyDialogs, Database, and Windows. However, with the cycle in place, they must now also be compatible with MyApplication, TaskWindow and MessageWindow. That is, MyTasks now depends upon *every other package in the system*. This makes MyTasks very difficult to release. MyDialogs suffers the same fate. In fact, the cycle has had the effect that MyApplication, MyTasks, and MyDialogs must always be released at the same time. They have, in effect, become one large package. And all the engineers who are will be stepping all over one another since they must all be using exactly the same release of each other. But this is just the tip of the trouble. Consider what happens when we want to unit test the Mydialogs package. We find that we must link in every other package in the system; including the Database package. This means that we have to do a *complete build* just to unit test MyDialogs. This is intolerable. If you have ever wondered why you have to link in so many different libraries, and so much of everybody else's stuff, just to run a simple unit test of one of your classes, it is probably because there are cycles in the dependency graph. Such cycles make it very difficult to isolate modules. Unit testing and releasing become very difficult and error prone. And compile times grow geometrically with the number of modules.

## *Breaking the Cycle*

It is always possible to break a cycle of packages and reinstate the dependency graph as a DAG. There are two primary mechanisms.

1. Apply the Dependency Inversion Principle (DIP). In the case of Figure 3, we could create an abstract base class that has the interface that MyDialogs needs. We could then put that abstract base into MyDialogs and inherit it into MyApplication. This inverts the dependency between MyDialogs and MyApplication thus breaking the cycle. See Figure 4.
2. Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.

## *The "Jitters"*

The second solution implies that the package structure is not stable in the presence of changing requirements. Indeed, as the application grows, the package dependency structure jitters and grows. Thus the dependency structure must always be monitored for cycles. When cycles occur they must be broken somehow. Sometimes this will mean creating new packages, making the dependency structure grow.

## *Top Down Design*

The issues we have discussed so far lead to an inescapable conclusion. The package structure cannot be designed from the top down. This means that is it not one of the first things about the system that is designed. Indeed, it seems that it gets designed after many of the classes in the system have been designed, and thereafter remains in a constant state of flux. Many should find this to be counterintuitive.

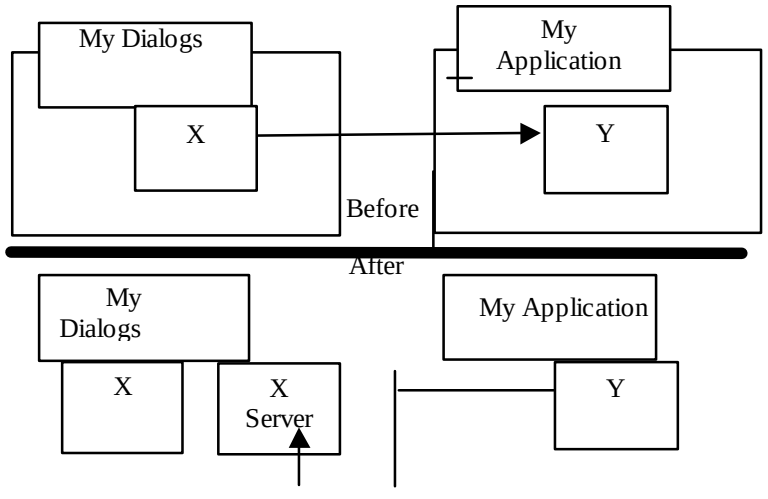


**Figure 4 Breaking the Cycle with Dependency Inversion**

We have come to expect that large grained decompositions are also high level functional decompositions. When we see a large grained grouping like a package dependency structure, we feel that it ought to some-how represent the function of the system. Yet this does not seem to be an attribute of package dependency diagrams. In fact, package dependency diagrams have very little do to with describing the function of the application. Instead, they are a map of how to *build* the application. This is why they aren't designed at the start of the project. There is no software to build, and so there is no need for a build map. But as more and more classes accumulate in the early stages of implementation and design, there is a growing need to map out the dependencies so that the project can be developed without the "morning after syndrome". Moreover, we want to keep changes as localised as possible, so we start paying attention to the common closure principle and collocate classes that are likely to change together. As the application continues to grow, we start becoming concerned about creating reusable elements. Thus the Common Reuse Principle begins to dictate the composition of the packages. Finally, as cycles appear the package dependency graph jitters and grows. If we were to try to design the package dependency structure before we had designed any classes, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create packages that produced dependency cycles. Thus the package dependency structure grows and evolves with the logical design of the system.

## *Conclusions*

Managing a complex project using packages and their interdependencies is one of the most powerful tools of OOD. By creating packages that conform to the three principles described in this paper, we set the stage for robust, maintainable, and reusable software.

Packages are the units that focus change, that enable reuse, and that provide the unit of release that prevents developers from interfering with each other.

This article is an extremely condensed version of a chapter from Robert Martin' s new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall.

 Object Mentor Inc, 14619 N. Somerset Circle, Green Oaks, IL, 60048, USA phone: 847.918.1004 fax:847.918.1023
email:oma@oma.com web:http://www.oma.com