Paper 40-25

OOP Needs OOA and OOD

Andrew Ratcliffe, Ratcliffe Technical Services Limited

Abstract

So SAS/AF® supports object-oriented programming (OOP), but does that mean that all applications developed using SAS/AF® are object-oriented? Not so. This paper provides an outline of an object-oriented approach to producing true object-oriented applications. The paper emphasises a focus on objects, not processes, combined with proper object-oriented analysis (OOA) and design (OOD).

Techniques and practices covered in more detail include the use of CRC (Class, Responsibility, and Collaboration) cards, the UML (Unified Modelling Language), software modelling, and an iterative approach to OO projects.

Introduction

Since reading Grady Booch[1] in the early 90's, I have been convinced of the merits of the OO approach. I have also been convinced that it takes a lot of work for those like me, brought-up with a procedural and modular training, to learn and truly understand the new approach.

The introduction of Frame entries to SAS/AF® at around the same time was accompanied by a large amount of talk about SAS software applications being object-oriented. The truth was that a) the programmer's interface with the SAS system had gained a large amount of object orientation, and b) programmers now had the **opportunity** to write object-oriented applications. However, not all programmers took that opportunity.

To this day, the majority of my new clients are not producing object-oriented applications. Many of them believe that they are, but a five-minute chat about the classes that they have in their applications soon convinces them otherwise.

The objects in object-oriented applications are represented by classes. Objects are items in the business-world that have attributes, states, and behaviour. Examples of objects include accounts, patients, users, and production lines. Actions like run, load, and measure are not objects, they are behaviours of one or more objects. I often get questioned "but running has attributes such as speed, so surely it's an object." Think carefully: the attribute of speed more accurately pertains to the object that is running. Running is behaviour, and speed is an attribute of the thing that is running.

With this brief paper I aim to guide the reader towards a clearer understanding of the object-oriented approach. I will do this with brief descriptions of some key areas of interest and with copious pointers to further reading.

Objects and Processes

In some ways, the OO approach is an evolution of the modular school. Before I get flamed by groups of OO enthusiasts, let me explain. One of the key concepts of modularity is that of breaking the problem down into pieces, defining the interfaces between the pieces, and then solving the individual problems without a great deal of cross-reference between the individual pieces and their solutions. It's called loose-coupling.

Loose-coupling is a key concept of OO too, but the loose-coupling is combined with other concepts such as abstraction, polymorphism, and hierarchies.

Using the traditional modular approach, data would be analysed almost independently of the tasks (processes) to be performed upon it by the application. OO bundles data and processes together into objects. Thus, individual items of data and related processes are tightly-coupled, whilst different data items are loosely-coupled.

One of the key tasks in building an OO application is finding the right classes in your analysis phase. This is distinct from the traditional programming methods where one would be looking for algorithms. A class should provide a representation of something in the vocabulary of the problem domain (or solution domain). It should be simple (small) yet flexible (and extendable). It should provide a good abstraction whilst hiding its implementation. Each class should have a clearly identifiable responsibility to fulfil, just like each person in a team.

In your search for candidate classes, look for nouns not verbs. Interview users and sweep any documentation you can find (requirements documents and sample reports, for example).

Things such as 'run,' 'load,' and 'validate' are typically not good classes. The fact that these words are verbs is a major clue. In these cases the more appropriate classes are likely to be things such as a database or a transaction.

When implementing your classes, keep the principal of abstraction in your mind. The consumer of your class (the programmer who calls your methods) should be ignorant of how your class is implemented. A clear example is that your consumer should not need to know whether you're storing your data in a data set, an SLIST, or an external file. Your consumer should focus on what your class delivers, not how it is delivered, and your consumer should **never** access your class's data directly.

For further reading on the subject of good classes and responsibility-driven design, see Booch[1] or Rebecca Wirfs-Brock[2].

CRC Cards

As I said, the majority of SAS software application developers have adopted a procedural approach to their work. This has come about from a number of reasons, not least being that the DATA step and PROC approach is very sequential and procedural in nature. To change one's entire mind-set on application design can be very challenging. Class, Responsibility, and Collaborations (CRC) cards provide an excellent, natural means of adjusting that mind-set.

First introduced by Kent Beck and Ward Cunningham in 1989, CRC cards are easy to use and have additional benefits arising from their interactive nature. In simple terms, the idea is that you create a 4" x 6" index-card for each proposed class in your application. On each card, you identify the name of the class, its responsibilities, and the classes with which it collaborates to fulfil its responsibilities. Then you run through scenarios of activities that the application must perform ('use cases' in UML terminology).

In running-through the scenarios, participants in the exercise take hold of the cards and act-out the responsibilities of their given classes. The cards act as a central part of an iterative exercise: running a scenario, analysing the outcome, and adjusting the classes and responsibilities.

Responsibilities are high-level descriptions of the purpose of the class. The size of the card prevents the creation of over-loaded classes, i.e. those that have too many responsibilities.

The simplicity of the CRC card concept (and its practical implementation in index cards) results in a concentration on the analysis and design rather than the implementation. It is ideal! Participants focus upon the objects in the system and are divorced from platform-dependence and language-dependence. By physically holding the cards and acting-out their classes' responsibilities, participants are forced to identify with their classes, to see the design from the perspectives of their classes.

The cards form a communication medium that is acceptable to both business-focused and technically focused participants. I find them to be an excellent facilitator when trying to get users, designers, and developers to sit around one table.

The physical interaction between participants is unavoidable and desirable. It contributes a) to the understanding of the design, and b) to the effectiveness of the team both during and after the exercise. CRC card activities are good team-building exercises.

In addition to using CRC cards as part of analysis and design phases, I have found them to be tremendously useful as training tools for those who are new to OO concepts. Students are taken away from all that they have learnt about traditional techniques and take to the class-focused approach very quickly.

For further reading on the subject of CRC cards, see Beck[3] and David Bellin[4]. Wirfs-Brock's[2] approach is responsibility-driven and supports the use of CRC cards.

QuickCRC is a software product produced by Excel Software. It permits CRC cards to be created and stored on MS-Windows and Macintosh. Further, it permits scenarios to be created, stored, and acted-out. Information is available at http://www.excelsoftware.com. Note, however, one of the big benefits of the CRC card approach is its interactivity. This benefit is lost with the use of computer software. As a means of storing the outcome of a CRC exercise, QuickCRC does have benefits.

Methods of Control

The interactions that occur between objects should also be the subject of your design thoughts. There are several different patterns of collaboration and control that might be used.

All objects are not created equal. Some are tasked with cajoling others into doing detailed work, in a pattern similar to the manager-staff relationship most of us work with in our own lives. Objects might be classified as controllers, information holders, or interfaces. Those objects that are more active (such as controllers) warrant more of our attention.

Methods of control range from 'uncontrolled' through to 'centralised' and 'de-centralised.' Applications that use an uncontrolled style usually result from lack of design. Thus, control and co-ordination is performed in many different parts of the application without any structure to it.

I have found that novice OO designers tend toward the centralised approach. In this approach, one (or a small number) of the objects takes a large amount of responsibility for interfacing with other objects. The central object acts like a traditional procedural solution, "running" one object after another in order to run through a sequence of tasks.

The centralised approach tends to result in a complex controller object that is difficult to maintain because of its complex code. In addition, it can be difficult to assign work to teams of developers when the bulk of the application's logic is contained in one single class.

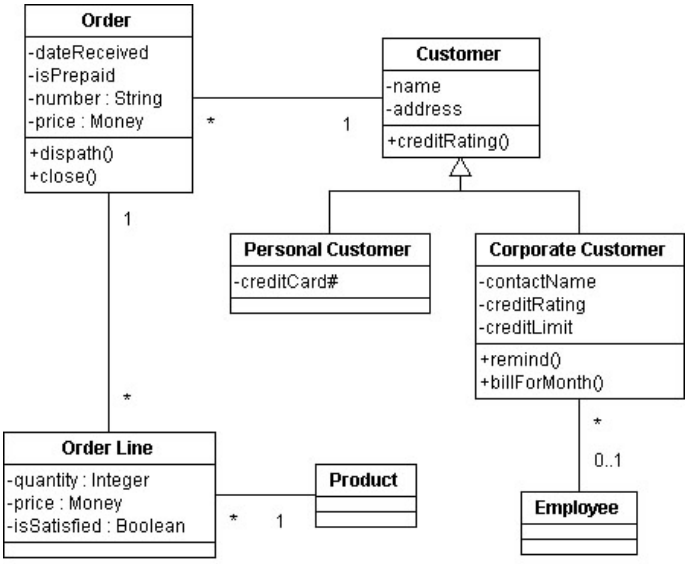
I much prefer the de-centralised method of control. Applications designed using this style tend to have clusters of objects. Each cluster has a clearly defined purpose and role within the application. Each object in a cluster co-operates and interacts with its partners rather than being centrally controlled. Similarly, a sub-set of classes in each cluster will co-operate and interact with classes in other clusters.

On the down-side, the de-centralised approach can make it harder for a maintenance programmer to follow the flow of events within the application. But, on the plus-side, responsibilities are spread more evenly (and to a shallower depth) amongst the application's classes. Each and every class is easier to maintain and enhance because of this.

For further reading on the subject of control styles, see Wirfs-Brock[5].

The UML

Drawing diagrams to describe your OO application needs something more than simple flow-charts. The



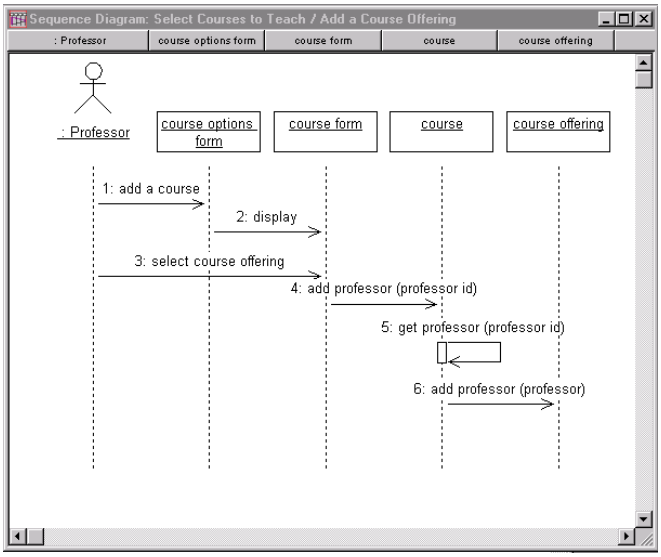
UML (Unified Modelling Language) is the definitive notation for describing OO applications.

The UML is just a notation - a series of symbols and diagrams. The set of symbols (and associated semantics) that comprises the UML is designed to optimise the information conveyed by any diagram that you should draw. The UML defines seven types of diagrams: use case, class, package, sequence, collaboration, state, activity, and deployment. The scope of the diagrams runs all of the way from high-level analysis down to individual class's methods and

attributes (instance variables) and the ways in which they interact.

The OMG (Object Management Group) are the independent standards group for all things object-oriented. They have accepted the UML as the standard modelling language. Thus, diagrams drawn using the UML notation will be more easily understood by your peers than those drawn using any other notation or style.

The class diagram is the one that most people instantly associate with. It represents the static



relationships between the classes: association ("has a") and subtypes ("is a"). It also documents the attributes and operations of the classes. The figure above shows an example class diagram.

Sequence diagrams have similarities with flow-charts. They demonstrate the sequence of communication between objects at run-time. All of the objects involved in the activity are listed across the top of the diagram, and the sequence runs down the page. Horizontal lines represent communication between objects. The figure alongside shows an example sequence diagram.

They say a picture paints a thousand words. Drawing diagrams as part of your design is almost a necessity. If you use UML diagrams then the "language" of your diagrams is universal, so your diagram will be understood by a greater number of people.

There are many sources for further reading on the subject of the UML. For those new to it I recommend the OMG Primer[6] and Martin Fowler[7].

Software Modelling

Drawing UML diagrams can be done with pencil and paper, but in many cases it is preferable to use some computer-based drawing package to ease storage and maintenance of the diagrams. A number of specialised packages are available for drawing and modelling.

Modelling packages are superior to plain drawing packages because they build a model behind the scenes that collates all the information from all of your diagrams. If you've already drawn and defined a class on one diagram and then choose to show it on another, the information you entered on the first diagram will automatically be echoed to the second. If you change the information in one diagram, all others are automatically updated. Thus, your diagrams remain consistent with each other.

A number of packages are available commercially. A selection of these were given a review[8] in the newsletter of the UK independent SAS User Group (VIEWS). The two figures shown in "The UML" were drawn with MagicDraw (http://www.nomagic.com) and Rational Rose (http://www.rational.com) respectively.

Iterative Development

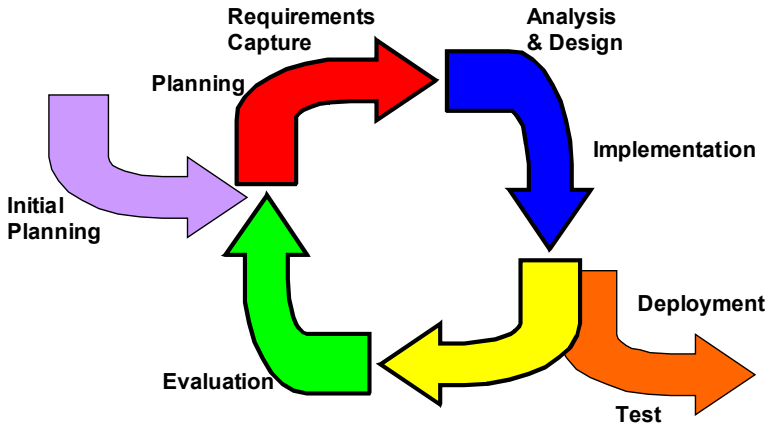
There is no hard-and-fast reason why they should be but iterative development seems to be associated with OO and waterfall with traditional approaches. Certainly, the vast majority of OO practitioners use the iterative development method.

In my experience, iterative development pleases sponsors, users, and developers alike. Sponsors like it because it decreases the inevitable risk involved in any sizeable project. Users like it because a) they do not feel bound-in to a large complicated specification from the outset, and b) they see results quickly.

Traditional waterfall development consists of three key phases: Define, Build, and Test. These are run sequentially. Iterative uses the same phases but they are used for small parts of the project, and those parts of the project run in parallel.

As the project progresses, the delivered article at the end of each phase grows in size. The risk in the

project is attacked at earlier stages because the deliverables are functional, integrated articles.



One of the deliverables of the initial planning phase is often an architecture that will provide a solid, resilient baseline for the subsequent detailed design and development. Achieving a solid architecture is crucial. That architecture will have been built using many of the techniques that I have discussed earlier, such as CRC cards for analysis and UML modelling software for documenting and communicating the architectural design.

Conclusion

Object-oriented techniques are an established means of developing applications. SAS software supports the development of object-oriented applications, but it does not do the object-oriented analysis and design. Without an object-based analysis and design, the resulting application will not be taking advantage of the merits of object-orientation.

Whilst CASE tools continue to grow in their abilities, they show no signs yet of replacing the analyst and designer! The tasks of analysing and designing continue as a requirement for successful application development. If

- CRC cards are used with the analysis;
- an appropriate method of control is chosen at the design stage; and
- the architecture and design are documented and communicated with the UML,

then the developers will be better able to create a truly object-oriented application (and achieve some or all of the associated benefits).

References

[1] Booch, Grady 1994. *Object-Oriented Analysis and Design With Applications.* The Benjamin/Cummings Publishing Company, Inc.

[2] Wirfs-Brock, Rebecca 1990. *Designing Object-Oriented Software.* PTR Prentice-Hall, Inc.

[3] Beck, Kent 1989. *A Laboratory for Teaching Object-Oriented Thinking.* http://c2.com/doc/oopsla89/paper.html

[4] Bellin, David 1997. *The CRC Card Book.* Addison, Wesley, Longman, Inc.

[5] Wirfs-Brock, Rebecca. *Characterising Your Application's Control Style.* http://www.wirfs-brock.com/characterizing_object_control_style.htm

[6] OMG 1998. *What Is OMG-UML and Why Is It Important?* http://www.omg.org/news/pr97/umlprimer.html

[7] Fowler, Martin 1997. *UML Distilled.* Addison, Wesley, Longman, Inc.

[8] VIEWS 1999. *VIEWS News, issue 5.* http://www.views-uk.demon.co.uk/issue5.pdf

Biography

Andrew Ratcliffe is a freelance SAS software consultant with over 15 years experience of SAS software. He specialises in object-oriented application development. Through his company (Ratcliffe Technical Services Limited), he is able to offer services including analysis and design consultancy, mentoring, training, and programming resources. Andrew can be contacted as detailed below:

Telephone: +44-1322-525672

Fax: +44-870-050-9662

Email: andrew@ratcliffe.demon.co.uk

Web: http://www.ratcliffe.demon.co.uk