MODERN OBJECT-ORIENTED SOFTWARE DEVELOPMENT

A.N. Dunlop
University of Southampton, SO17 1BJ, England

Abstract
Object-oriented (OO) programming has been around for a few years and there are many users of OO programming languages. There are, however, few true practitioners of this method. Just as it is possible to use a screwdriver as a hammer, programming with an OO language does not imply that the language is being used correctly. The same was true with the onset of structured programming languages. The development of these languages did not instantly result in users producing beautifully structured code without use of the infamous "goto". It was only later that good programming practice or an appropriate methodology was defined for this programming style. OO programming has now reached a similar situation where "good programming practice" can be defined. In this paper we describe the change in emphasis in moving from procedural to OO programming and describe some of the main techniques that are now promoted to ensure the development of good OO programs.

1 FROM PROCEDURAL TO OBJECT-ORIENTED PROGRAMMING
Object-oriented programming is the dominant programming paradigm, replacing the “structured”, procedural-based methods of the early 1970’s. Procedural programming is based on developing algorithms and subsequently a suitable data structure [18] for the problem. Object-oriented programming reverses this order putting the emphasis on designing the data structure first, and only then determining the algorithms that operate on the data. This is the more logical order as requirements often change during program development and maintenance, necessitating the development of additional functions. In contrast, the data to be manipulated in a problem rarely changes. By concentrating on developing the data structure first, it is a relatively simple task to add additional functions.

Superficially, Rambaugh [15] describes the term “object-oriented” to mean that software is organised as a collection of discrete objects that incorporate both data structure and behaviour. Users of objects may only access their functionality in predefined ways. This is in contrast to structured, or procedural programming in which the data structure and behaviour are only loosely coupled. In this paradigm, users are able to access and manipulate data structures directly, and use procedures to manipulate data for which the system may not have been intended.

Traditionally systems were constructed using the waterfall method [14]. This was based on the idea that clients would formally agree a requirements document. A design would then be put together, which would be further agreed. The system would be implemented, and there would then follow an endless process of maintenance. Modern ideas move away from this waterfall approach. Iterative methods are considered more appropriate for many system development approaches. This still follows the notion of analysis, design and implementation, but on a cyclical basis, where subsequent cycles build on earlier cycles. Object-oriented systems are significantly easier to fit to this new way of thinking about software than procedural systems.

2 ELEMENTS OF A MODELLING TECHNIQUE
An analysis or design method is a coherent approach to describing a system. This consists of a number of techniques that are used to describe different aspects of the system. Each technique emphasizes some
aspect of the system and neglects others. The method also has some process to say how the techniques are combined.

Each modelling technique has two key elements; syntax and semantics. The syntax is usually a diagrammatic notation, while the semantics may or may not be formally defined depending on how rigorous the technique is. A rigorous technique is one that does not permit any ambiguity in its interpretation. This is stressed to its utmost in formal methods, such as Z[12] and VDM[13]. Less rigorous techniques lead to ambiguity which has to be dealt with by programmers at a later date.

Modelling techniques can be evaluated according to the criteria of comprehensibility and expressiveness. Comprehensibility is a fundamental requirement of a modelling technique, as its value lies in its ability to highlight certain important details in a system. A method that is no more comprehensible than program code is not useful. Similarly, a method that is unable to express all the required concepts is not useful. Expressiveness is not entirely a virtue, however. More expressive techniques have more concepts and are thus more difficult to learn. Thus a less expressive technique loses in some respects but the lack of concepts may be worth suffering for easier learning. A good technique strikes a balance between comprehensibility and expressiveness.

3 MODELLING METHODS

3.1 Structured Analysis/Structured Design (SA/SD)

One of the first software analysis and design methodologies was the Structured Analysis/Structured Design (SA/SD) method [11]. SA/SD includes a variety of techniques for specifying software. During the analysis phase of software development data flow diagrams, process specifications, a state transition diagram and entity-relationship diagrams are used to logically describe a system. In the design phase, details are added to the analysis models and the data flow diagrams are converted into structure chart descriptions of the programming language.

Data flow diagrams model the transformations of data as it flows through the system. These are the focus of SA/SD. A data flow diagram consists of processes, data flows, actors and data stores. The SA/SD process is based on recursively dividing complex processes into subdiagrams until only small processes are left that are easy to implement. At this point each process is specified with decision tables, pseudocode or other techniques. The data dictionary defines the data flows, data stores and the meaning of names within the data flow diagram.

The state transition diagrams model the time dependent behaviour of the system. These diagrams describe the control process or timing of function execution and data access triggered by events.

Entity-relationship (ER) diagrams highlight relationships between data stores, with each ER data element corresponding to one data flow diagram data store.

Although the SA/SD methodology has served many users well over the years, it was developed at the time when procedural programming was in vogue. Accordingly this method is best suited to this style of programming. In the SA/SD approach, the functional model dominates, in the same way that procedural programming styles encourage users to think of the functional requirements of the system first. The dynamic model is the next most important, and the object model least important. In contrast, object-oriented programming places the greatest importance on the object model, then the dynamic model and finally the functional model. This change in emphasis brought about by object-oriented programming has accordingly resulted in the development of a number of true object-oriented development methods.

3.2 The development of Object-Oriented Analysis and Design Methods

Prior to 1990’s SA/SD was the lingering methodology although industry was moving towards true object-oriented software development. There were, however, few options available to those who sought an alternative to this methodology. By the early 1990’s though, a number of object-oriented software method-
ologies were available. We outline a few significant methods below:

One of the most popular object-oriented methodologies, due largely to its early presence, was developed by Booch [1]. This early work concentrated more on design rather than the full lifecycle but was extended later [3] [2] to address analysis and design.

Another well established object-oriented methodology is the Object Modelling Technique (OMT) developed by Rambaugh et al [15]. This method is traditionally applied to commercial/information applications, and is not particularly well known for application to complex systems due to its weakness in modelling sub-systems and different levels of abstraction. Lockheed has, however, updated the OMT to make it more suitable to realtime application [5]. OMT consists of a well defined development process featuring five distinctive phases: conceptualisation, analysis, system design, object design and implementation. The OMT notation uses three interrelated models to describe the system: the object model, the dynamic model and the functional model.

Jacobsen [9] developed the object-oriented software engineering method (OOSE), that defined “use-cases”. These are a sequence of transactions that are performed by a user of the system. Each “use-case” outlines a likely thread of control through the many objects of the system, potentially unifying the static and dynamic system views. Jacobson’s Objectory method partitions systems along use-case boundaries, then identifies user interface, control and entity objects for each use-case. Interaction diagrams describe how a group of objects work together to implement a given use-case and state transition graphs specify an object’s behaviour over time and include symbols for sending and receiving messages.

Another major object-oriented analysis method was developed by Shlaer and Mellor [16] using three analysis models. An information model identifies the conceptual entities and objects; the dynamic model formalises the interaction of these objects, while the process models formalise the processing required.

A number of other methodologies have also been developed, to support more specialised uses such as the development of real-time systems. HOOD [4], ROOM [17] and MOOSE [7] are examples of this.

3.3 The development of the Unified Modelling Language (UML)

Although the development of object-oriented modelling methods satisfied the need to obtain a method more suited to object-oriented programming, the proliferation of methods caused confusion with users. However, in 1995, Jacobsen, Rambaugh and Booch joined forces to create a unified method that blended ideas from all three approaches. The result was the Unified Modelling Language (UML) released in 1996 with version 1.0 [19] being submitted to the Object Management Group (OMG) analysis and design task force for standardisation in January 1997.

Given that UML is the creation of three of the foremost figures in object-oriented analysis and design methodologies, and that it has been endorsed by major corporations, such as Microsoft and Oracle, success seems inevitable. Indeed, Fowler [6] and others believe that UML will by accepted by the OMG and will become the dominant object modelling method. Other methods will become niche market players.

UML has resulted from fusing the concepts of Booch, OMT, and OOSE. UML is accordingly a superset of the techniques of these methods and is therefore more widely applicable than existing methods. In particular, UML targets the modelling of concurrent, distributed systems, meaning that UML contains elements that address these domains. The result is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.

Of interest is the fact that UML defines a standard modelling language, not a standard process. This distinction is significant as different problems require different processes and modelling techniques. Moreover, the choice of modelling techniques has a profound influence upon how a problem is attacked and how a solution is shaped. For this reason, UML defines a notation for the majority of modelling techniques. These are as follows:
Use-case diagram

Class diagram

Behaviour diagrams

- Sequence or Interaction diagram
- Collaboration diagram
- State Transition diagram
- Activity diagram

Implementation diagrams

- Component diagram
- Deployment diagram

In the following three sections we provide an overview of use-case diagrams, class diagrams and behavioural diagrams, and describe the role each of these play in OO program development.

4 USE-CASE DIAGRAMS

One of the most fundamental problems in software engineering is determining the requirements of a system. Use-cases were introduced by Jacobson [9] to solve this problem.

The use-case approach requires the analyst to determine all the potential actors involved in a system. Actors are external to the system and make use of it. An actor is typically a person, but may be a third-party organisation or another computer system. One person may in fact be multiple actors, say a shop assistant may be a customer of the same shop at another time. We model actors, not individuals.

An actor makes use of a system in different ways. Each of these ways is known as a use-case. Jacobson defines a use-case as “a behaviourally related sequence of transactions (performed by a user/actor) in a dialogue with the system”. A use-case may involve a number of actors, just as an individual actor may make use of several use-cases. We represent use-case diagrammatically by ovals and actors by stick men.

Consider the example given by Lunn [10] of a banking system where a customer can withdraw money. A use-case for this could be drawn as in Figure 1.

![Figure 1: Use-case of a customer withdrawing money](image)

Of course, to withdraw money, a customer must also be able to deposit money. So there is at least one more use-case (Figure 2).

Now it might be that the system which is being implemented in the bank needs to involve a cashier for depositing, but that to withdraw money the customer has to use the cash machine. The cashier is then an actor. Taking this into account and allowing customers to check their account balance requires further additions to the model (Figure 3).
Use-cases may also use other use-cases. The withdraw cash use-case would make use of the account balance use-case before issuing the money. We represent this by connecting use-cases in our graphical model (Figure 4).

A use-case is a very abstract view of what the system provides to the various actors who use it. It is not intended to give a detailed, blow-by-blow account of how the services are provided. One of the big dangers of use-cases is that of structuring the software to mimic the use-cases. Use-cases provide an external view of the system. The software is often structured in a completely different way. The biggest danger is that of turning each use-case into a procedural controller which acts upon simple data holders.

Use-cases a vital part of object-oriented development and should be used to understand the requirements of a system. They are also a suitable interface between the clients, users and developers of the system and should be used to drive system testing.

5 CLASS DIAGRAMS
5.1 Objects, Attributes and Operations
The class diagram is a central modelling technique that is common to nearly all object-oriented analysis and design methods. This diagram describes the type of objects in the system and the relationships between them. First let us consider what objects are. Rambaugh [15] defines an object as “a concept,
abstraction or thing in the world with crisp boundaries meaningful to the problem at hand”. In general, objects tend to be physical entities that are relevant to the problem being solved.

Objects can be described by their attributes and operations. Attributes are the changeable characteristics of an object - cats have colour, size, weight and sex. Operations are the things an object does or can have done to it - cats can catch mice, eat, miaow, and be stroked. In UML notation we draw the general class of cat objects as shown in Figure 5. The name is shown at the top, the attributes are listed underneath and the operations are listed below that.

<table>
<thead>
<tr>
<th>Cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>weight</td>
</tr>
<tr>
<td>sex</td>
</tr>
<tr>
<td>Catch mice</td>
</tr>
<tr>
<td>Eat</td>
</tr>
<tr>
<td>Miaow</td>
</tr>
<tr>
<td>Be stroked</td>
</tr>
</tbody>
</table>

Figure 5: Example object class specification for a cat

In an object model, all data is stored as attributes of some object. The attributes of an object are manipulated by the operations. The only way of getting at the attributes is through an operation. Attributes may sometimes be objects in their own right.

In an object model, all functionality is defined by operations. Objects may use each others operations, but the only legal way one object can manipulate another object is through an operation.

Object modelling is about finding objects, their attributes and their operations, and tying them together in an object model.

5.2 Three Perspectives

Obtaining candidate objects from a problem statement is relatively simple. The difficulty lies knowing what objects are appropriate, and what their appropriate attributes and operations are. This is a question
of focus. Fowler [6] identifies three perspectives from which class diagrams can be drawn:

- **Conceptual Perspective.** In this case the class diagram represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them, but it is often not a direct mapping. Indeed the model is drawn with little or no regard for the software that might implement it, and is generally language independent.

- **Specification Perspective.** The specification perspective considers the software interface, not the implementation. Object-oriented development puts a great emphasis on the difference between interface and implementation. An interface may have many implementations due to implementation environment, performance characteristics, or vendor.

- **Implementation Perspective.** In this view the class diagram lays the implementation bare. This is probably the most often used perspective, but in many ways the specification perspective is often a better one to take.

Understanding the perspective is crucial to both drawing and reading class diagrams. Unfortunately the lines between the perspectives are not sharp, and most modellers do not take sufficient care to get their perspective sorted out when they are drawing. In general, UML takes an implementation perspective, although its use of associations (discussed below) is often more conceptual than implementation.

### 5.3 Associations

Associations represent relationships between object classes. For example, if “person” and “company” are objects in a model, the statement, “a person works for a company” relates the two objects. The interpretation of associations varies with the perspective. From the specification perspective these are responsibilities for “knowing”, which must be made explicit by access and update operations. This may mean that a pointer exists between classes, but that is hidden by encapsulation. A more implementation interpretation implies the presence of a pointer. Thus it is essential to know what perspective is used to build a model in order to interpret it correctly. Associations are drawn as lines between objects. For example, consider the association shown in Figure 6. If this is a specification model, the company should have methods to determine its employees, while a person object must be able to determine his/her employer. From an implementation perspective, this association indicates a pointer link between the objects.

![Figure 6: Example of an association between classes](image)

All associations can be thought of as bi-directional, but uni-directional associations are important for specification and implementation perspectives. For specification models bi-directional associations give more flexibility in navigation but incur greater coupling. In implementation models a bi-directional association implies coupled sets of pointers. UML tend to use bi-directional associations for analysis but uni-directional in design.

Associations have cardinality. This specifies whether the association is mandatory, optional, one to many, or many to many. For example, the association “person works for company” in Figure 6, should indicate that one or more people work for a company. This is made explicit in the UML notation by indicating the cardinality on the links. The syntax for this is shown in Figure 7.
Data modellers often have difficulty distinguishing the difference between attributes and associations. The definition furnished by most methods, including UML, is that attributes are seen as internal to a class, while associations link classes. Although this distinction is important in the modelling sense, it makes little difference at the implementation level. In particular, Booch considers attributes equivalent to uni-directional associations which are implemented by containment in C++.

5.4 Aggregation

We can construct objects of other objects. This is known as aggregation. The behaviour of the larger object is defined by the behaviour of its component parts, separately and in conjunction with each other. These special forms of association are indicated graphically by a diamond shape at the end of the line at the aggregating object. For example, Figure 8 shows that an engine consists of a starter motor and an ignition system.

It is notoriously difficult to define the difference between an aggregation and an association, or to indicate whether the distinction is useful. Implementation perspectives often use aggregation to imply C++ containment.

5.5 Generalisation

Often we find that there are objects which have something in common. It is then useful to create an abstract object which groups together the common features, and to use inheritance to define the original objects. Inheritance means that all the attributes and operations of an abstract object are available in the
specialised object. This is indicated diagramatically by placing a triangle on the link between objects. The triangle in the diagram indicates inheritance. The point of the triangle indicates where operations and attributes are inherited from. Consider the example of modelling a system that contains many types of engines. All engines contain some basic properties, but petrol and steam engines are also fundamentally different. This is neatly represented using inheritance as shown in figure 9.

![Class diagram example](image)

Figure 9: Steam, Petrol and Diesel engines inherit properties from the abstract engine class

Inheritance is considered good for software re-use and for clarity of description. When new objects are created which are similar to other objects, they can have many of their attributes and operations already defined. This saves on the effort required to introduce new objects into a system. Inheritance is also a used naturally to describe things and thus can serve the purpose of clarifying an object and its role.

Class diagrams are the backbone of the UML as well as other object-oriented analysis and design methods. They can, however, easily become over-complex largely as a result of getting bogged down in implementation details. To avoid this, the conceptual or specification perspective are most useful initially.

6 DYNAMIC MODELLING

6.1 Event Traces, Interaction Diagrams and Collaboration Diagrams

Dynamic modelling tries to capture how objects behave and how they interact. In this way, we can find new operations, attributes and relationships for the object model. Dynamic models are perhaps the most effective way of uncovering the behaviour of systems.

We start dynamic modelling by obtaining a number of scenarios for the use of the system. These scenarios, or stories, list the chain of events that occur for the use-cases. Hence, use-cases are a high level view of the interactions actors have with the system, while event scenarios are a list of events that would be required for each use-case. For example, in Figure 1, we have a use-case for a withdrawing money from an automatic teller machine. The event scenario for this might be something like that given below:

- ATM requests Card
- User inserts Card
- ATM asks user to enter personal identification (PIN) number
- User enters PIN number
- ATM displays operation choice
- User requests withdrawal
- ATM asks user for the amount
- User enters amount
- ATM sends card information, PIN, and amount requested to consortium
- Consortium forwards request to users bank
- Users bank checks users account for funds
- Users bank sends acknowledgement to consortium
- Users bank updates users account
- Consortium sends acknowledgement to ATM
- ATM prepares cash
- ATM returns users card
- ATM dispenses cash

In practice this is likely to be significantly more complex, but this is sufficient for our purposes. Of course, there are many different scenarios for users withdrawing money from an ATM, including situations where the wrong PIN number is entered, insufficient funds exist in the account, to name but a few. These event scenarios are then usually expanded into interaction diagrams (see Figure 10). These diagrams indicate the source and destination of messages from the event trace. Down the left hand side we can list the actions or events in a scenario. The vertical lines indicate objects. The arrows represent the interactions between the objects. The labels on the arrows are the operations.

The particular purpose of interaction diagrams is to identify objects, attributes and operations that were initially overlooked in the object model. This follows the philosophy that everything in object modelling is to do with expanding the object model.

An alternative form of the interaction diagram is the collaboration diagram. This conveys the identical information to an interaction diagram, but with objects shown as icons rather than vertical lines. Again arrows indicate the messages sent in the use-case. See Fowler [6] for more on this subject.

6.2 State Transition Diagrams
State transition diagrams have been used right from the beginning in object-oriented modelling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to move from one state to another. These diagrams therefore identify what events can occur in a system, and what effect they can have on the object. The most commonly used state transition diagram is the Harel statechart [8] and is the form adopted by the UML. An example statechart is shown in Figure 11 for a packing box object. A box starts in an empty state. It then receives “add” events until the box is full, upon which it can receive a “seal” event to close the box. This diagram indicates that the box cannot get a “seal” event until it is full, and that a full box cannot get any further “add” events. It also shows that the box can
Figure 10: Example Interaction Diagram
receive a “split” event in either the state of “being filled” or the “full” state. Diagrams of this type can provide a good sense of the events that can occur in a system and the effect they have on the object.

In general it is not possible to define all the possible states of a system. Whilst this is all right for small systems, it soon breaks down in larger systems as there is an exponential growth in the number of states. This state explosion problem leads to state transition diagrams becoming far too complex for much practical use. To avoid this problem, object-oriented methods define separate state-transition diagrams for each object class. As with constructing interaction diagrams, the reason for developing statecharts is to identify further operations and attributes to the object model, to check the logical consistency of the object, and to more clearly specify its behaviour. It is therefore only necessary to construct statecharts for objects that would benefit from a formal description of their behaviour.

Statecharts are often constructed from interaction diagrams. A separate statechart is constructed for each object in the interaction diagrams. Each event arriving at an object results in the object changing state. By identifying the object states from each interaction diagram created from a scenario, a statechart can be constructed.

7 Summary
This paper describes the transition from procedural programming to object-oriented programming and presents an overview of the main modelling techniques associated with object-oriented analysis and design using the Uniform Modelling Language syntax. In using modelling techniques it is important to bear in mind that these techniques are only there to assist in developing and maintaining program code. For this reason religious adherence to modelling all aspects of the system using all the techniques available is not appropriate. It is far better to have a few, well considered and up to date models, than a host of out of date models for trivial aspects of the system.

Many books have been published on the subject of object-oriented analysis and design, and much information is available on the web. The interested reader is referred to the reference list for further information.
References