UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 5: Object Oriented Analysis and Design Methods

PROBABILITY SQUARE - Method of analysing problems, used by Avon. It involved representing each element of a problem by a cube, and moving the cubes in relation to each other so that new and previously unthought-of interactions could be perceived.

Blakes 7. The Program Guide. Tony Attwood.

Introduction

This lecture examines in more detail the interrelated notions of *object oriented analysis* and *design*. This lecture examines, first, what is meant by the terms *analysis* and *design*. Secondly, an example method of analysis and design is introduced. Thirdly, several examples of other methods together with criteria for their choice are identified. You are recommended to read one of the texts associated with a method identified in this lecture as each method provides enough material for a 20 hour lecture course !

Object-oriented Analysis and Design

The notion of *object oriented analysis* as a means of developing a software design has its basis in models of "reality" which reflect more naturally the structure and behaviour of "things" in our everyday world. Here is an example of a definition of the terms "object-oriented analysis" and "object-oriented design" which is freely available on the WWW[1]:-

"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain. This emphasises the building of real-world models, using an object-oriented view of the world. Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

The object model is the collection of principles that form the foundation of object-oriented design. It is the conceptual framework of all things object-oriented. There are four major elements of this model. Without any one of these elements is not object-oriented.

- ? *Abstraction*
- ? *Encapsulation*
- ? *Modularity*
- ? *Hierarchy*

There are three minor elements of the object model. Each of these elements is a useful, but not essential, part of the object model.

- ? *Typing*
- ? *Concurrency*
- ? *Persistence*

Without this conceptual framework, you may be programming in a language such as Smalltalk, Object Pascal, C++, CLOS, Eiffel, or Ada, but your design is going to smell like a FORTRAN, Pascal, or C application.

You will have missed out on or otherwise abused the expressive power of the object-oriented language you are using for

There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence. From the most to the least useful, these kinds of abstractions include the following:

? *Entity abstraction*

An object that represents a useful model of a problem-domain entity.

? *Action abstraction*

An object that provides a generalised set of operations, all of which perform the same kind of function.

? *Virtual machine abstraction*

An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations.

? *Coincidental abstraction*

An object that packages a set of operations that have no relation to each other.

We strive to build entity abstractions, because they directly parallel the vocabulary of given problem domain."

As the above definition implies, methods for analysis and design seek to be "all encompassing" in the sense that they are intended to provide a means of describing, i.e. at least *documenting*, many different aspects of the development of software systems.

<u>An Example Method: Booch</u>

The Booch method covers the analysis and design stages of the development of a software system[2]. The method has been criticized for its relatively large set of symbols (recall the "all encompassing" aim of such methods) each of which is included in the hope that (almost) every aspect of an analysis and its associated design can be documented.

The basis of the method, in common with other methods described below, is a means of representing the notions of a class and a class instance (object). Thus, this particular method uses *class* and *object diagrams* as a means of representing by an analysis/design.

? <u>Class diagrams</u>[3]

These define which classes "exist" and also the relationship(s) between classes. The static properties[4] are described in terms of a (hopefully) unique *class name* and a set of associated *attributes*. The dynamic properties are described by a state-transition diagram. Various kinds of class are distinguished by different icons, i.e. Class, Class utility, Parameterized class, Class category, etc.

[pic]

? Class (dynamic behaviour is defined in a state transition diagram)

[pic]

? Class Utility (can also denote a class with only static members)

[pic][pic]

? Parameterised Class

[pic]

? Class Category

Syntax For Attributes

? `Attributes`

```
A Attribute name only
:C Attribute class only
A : C Attribute name and class
A : C = E Attribute name, class and default expression
```

? `Methods`

```
N() Operation name only
R N(Arguments) Operation return class, name and formal arguments
```

? `Class adornment`

```
{ } Semantic constraint (e. g. {0 .. 7})
```

? `Export control (for attributes/operations and relationships):`

[pic]

? `Properties (for relationships):`

[pic]

? <u>Object diagrams</u>

Define which objects "exist" and also links between objects together with "traces" of the execution of a "scenario". Again, properties are described in terms of a unique object name and a set of associated attributes and different icons distinguish various kinds of object, i.e. Object(s), Object Relation(s), Object Syncronisation (for "active" objects) and Object visibility.

[pic]

? Object

[pic]

? Object relation

[pic]

? Synchronisation (for active objects)[5]

[pic]

? Visibility (value/reference)

In addition a variety of other kinds of diagram may be developed including:-

? Module diagrams

Module diagrams relate the "logical" view of a system (represented by class and object diagrams etc) to "physical" view of the system, i.e. they show how classes are allocated to modules written in a programming language. Typically, a modular programming language, e.g. C++, associates modules with files, thus, the default would be to allocate each class to a separate file. However, if more than one classes is to be represented by a single module then module diagrams enable the internal structure of the module to be represented.

? Process diagrams

Define the allocation of *processes* to *processors*.

? Interaction diagrams

Interaction diagrams "trace" the execution of a "scenario" in a "real-time" environment.

You are recommended to read the opening chapters of "*Grady Booch: Object-oriented Analysis and Design with Applications, 2nd edition. Benjamin Cummings, Redwood City. ISBN 0-8053-5340-2, 1993.*".

Criteria for Choosing a Method

A variety of different criteria can be adopted when choosing an object oriented design method. Three criteria used in a recent research project are:-

? It must be a "truly" object-oriented method.

? It must be extensively described in a textbook and thus be "widely available".

? It must be regarded a "popular" method.

These criteria were then used as a means of choosing, comparing and contrasting the following methods:-

? Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) from Coad and Yourdon. [Coad 1991a] and [Coad 1991b]

? Designing Object-Oriented Software (DOOS) from Wirfs-Brock et al. [Wirfs-Brock 1990]

? Object Modeling Technique (OMT) from Rumbaugh et al. [Rumbaugh 1991]

? Object-Oriented Analysis and Design with Applications (OOADA) from Booch. [Booch 1994]

? Object Lifecycles (OL) from Shlaer and Mellor. [Shlaer 1992]

? Object-Oriented Analysis and Design (OOAD) from Martin. [Martin 1993]

? The Fusion method from Coleman et al. [Coleman 1994]

? Object-Oriented Software Engineering (OOSE) from Jacobson et al. [Jacobson 1992]

? Object-Oriented System Development (OOSD) from de Champeaux et al. [de Champeaux 1993]

? The MOSES method from Henderson-Sellers and Edwards. [Henderson-Sellers 1994]

Coad and Yourdon's OOA/OOD

Coad & Yourdon identify seven "motivations" and "benefits" for OOA/OOD over more traditional methods of analysis and design:-

? Tackle more challenging problem domains

? Improve analyst and problem domain expert interaction

? Increase the internal consistency across analysis, design and programming

? Explicitly represent commonality between classes and objects

? Build specifications resilient to change

? Reuse OOA, OOD and OOP results

? Provide a consistent underlying representation for analysis, design and programming

Coad & Yourdon identify one fundamental benefit of OOA/OOD as a reduction in the complexity of a problem and of a (software) system's responsibilities within its solution. This reduction in complexity is reflected in the method's basis, i.e. it is based on a set of general principles for managing complexity which exploit a uniform underlying representation. This uniformitu of representation (it is argued) has major implications including:-

? No major difference between analysis and design notations

? No 'transition' from analysis to design

? No waterfall model has to be followed, spiral and incremental are also possible

? There are still different skills and strategies needed for analysts and designers

? There is a uniform representation from OOA to OOD to OOP (OO programming)

In summary, Coad & Yourdon's approach consists of definitions of notions fundamental to the object oriented paradigm, i.e. *classes*, *objects*, *inheritance* and *communication* via *message passing*.

The Designing Object-Oriented Software (DOOS) method by Wirfs-Brock et. al.

The basis for DOOS as a means of managing complexity is the use of abstraction, i.e. an object oriented design in DOOS is an abstraction of the "real world" in which "knowledge" of the "real world" is encapsulated by objects[6].

Once identified, objects have their responsibilities defined to form an abstract "world" (or model) of collaborating objects. Perhaps the most striking claim for this method is that software is seen as a living thing, i.e. an anthropomorphic view is taken of software[7] !

The major benefit of this approach (over conventional approaches) is claimed to be the reuse of objects and also the ease with which DOOS designs can be translated into implementations written in object oriented programming languages.

The Object Modelling Technique (OMT) by Rumbaugh et. al.

Rumbaugh et al. identify a fundamental difference between conventional approaches to analysis and design and object oriented approaches, i.e. the latter is not based on functional decomposition (stepwise refinement) but on describing the real objects that play a role in the real world[8].

Using OMT software is organised as a collection of discrete objects that incorporate both data-structure and behaviour. The essential characteristics of "objects" in OMT are identity, classification (the grouping of Objects in Classes), polymorphism and inheritance. Rather confusingly, the basis for the approach is described as the "*identification and organization of application-domain concepts instead of implementation domain concepts*". Thus, Rumbaugh et al. argue that the benefits of an object oriented approach arise from a greater emphasis on essential properties of an object (not simply its structure) with a corresponding emphasis (from the software developer) on "*thinking more carefully and more deeply about what an object is and does*", i.e. not how it does it.

Rumbaugh et al. identify several "advantages" for their approach (over conventional approaches) including:-

? Shifting of development effort into analysis

? Emphasis on data structure before functions. This provides stability

? A seamless development process

? An iterative rather than sequential approach, i.e. features are added and clarified in iterations.

Object Lifecycles (OL) by Shlaer and Mellor

The "original" OOSA (Object Oriented Systems Analysis) method was developed as a means of resolving a problem with conventional approaches to analysis and design, i.e. such approaches provide no means of defining possible "conceptual entities" and subsequently determining the "implications" of those definitions. The basis technique which underpins OOSA is termed "Information Modelling" since such models (it is argued) provide an aid to formalising knowledge.

OL developed from OOSA by permitting actions to be partitioned into processes with desirable properties and by introducing higher-level diagrams as a means of developing large scale modular descriptions.

Principles of Object-Oriented Analysis and Design (OOAD) by Martin & Odell

The OOAD method, in common with other methods, has its basis in the management of "real world" complexity by *abstraction*,[9] i.e. removing distinctions between objects in order to concentrate upon common properties, in generalization (distinguishing object types that are more general than others) and in composition (forming objects from component parts).

Unlike many such methods OOAD is grounded in a theoretical foundation (logic and set theory) and each concept is formally defined.

The Fusion method by Coleman et. al.

The Fusion method is a ’second-generation’ method in the sense that it integrates and extends object oriented approaches which seek to provide a method encompassing requirements definition, analysis, design and implementation. In particular, the "object model" in Fusion is almost identical to OMT, the "operation model" in Fusion is similar to the functional model in OMT, however, the dataflow diagrams of OMT are considered inappropriate and have been formalized with pre- and postconditions, and finally Class Responsibility Collaborator (CRC) extended with communication information has influenced Fusion’s object interaction graphs whilst Booch’s OODA visibility information has influenced Fusion’s visibility graphs.

Fusion adopts a small set of diagramming techniques and comprises three sequential "phases":-

? analysis

? design

? implementation

Within each phase, individual "steps" order the decision making process and the output ("deliverables") from one step are the input to the next step. Associated with each separate rules for checking the consistency and completeness of the model being developed. Interestingly, the developers of Fusion acknowledge the need for such "all encompassing" methods to be adaptable in the sense that they may need to be simplified when all techniques are not required and guidelines for adapting the method, e.g. combining Fusion with other methods are proposed.

Object-Oriented Software Engineering (OOSE) by Jacobson et. al.

OOSE combines three different techniques which have been in widespread use for some time:-

? Object-oriented programming, more specifically, the concepts of *encapsulation*, *inheritance* and *relationships* between *classes* and *class instances* (*objects*).

? Conceptual modelling.

In OOSE conventional approaches to conceptual modelling are extended such that they incorporate object-oriented concepts and permit dynamic behaviour to be modelled.

? Block design

The notion of block design has its origins in hardware design where hardware "modules" have their own functionality and are connected together via together with well-defined interfaces.

Object-Oriented System Development (OOSD) by de Champeaux et. al.

In the same style as OOSE, OOSD integrates existing object oriented approaches. OOSD exploits an analysis notation called "our analysis notation" (OAN) and a design notation called "our design notation" (ODN) as a means of representing a minimal collection (set) of "objects" from which preferred objects can be distracted.

The MOSES method by Henderson-Sellers and Edwards

MOSES seeks to address all of the stages in a conventional lifecycle model of the software development process and provides activities, guidelines and deliverables for all aspects of an OO project. Exploiting the method involves the use of eight textual and five graphical models, however, its main advantage is claimed to be the use of a single object model throughout the product lifecycle, in particular, that such a model enables smoother transitions between stages than the correspondingly less smooth transitions associated with conventional approaches.

Summary and Conclusions

This lecture examined two interrelated notions, i.e. object oriented analysis and design. A wide variety of different methods have been identified each of which seeks to embody notions drawn from the object oriented paradigm, i.e. the notions of class, class instance (object), inclusion polymorphism (inheritance) etc, primarily as a means of developing descriptions which translate more naturally into software systems implemented in object oriented programming languages. In addition, the notion of complexity, more specifically, the need to manage complexity, is used as a primary justification for object oriented approaches to analysis and design.

It is instructive to reflect on the "state of the art" in software engineering methods and supporting tool(sets) in the mid 1980’s (see for example the STARTS guide) with the "state of the art" in object oriented approaches to software development in the mid 1990’s. In the early to mid 1980’s there was a variety of different methods of analysis and design often termed "structured" some of which were informal and some of which were formal, some of which used informal diagrammatic notations whilst other used formal languages, some notations embodied or supported an underlying method for their use, some methods had associated tool(sets). Much the same can be said currently for object oriented approaches to analysis and design. It will be interesting to see how this field of endeavour develops !

Chris Harrison, January 1997.

----

[1] http://arkhp1.kek.jp/managers/computing/activities/OO_CollectInfor/Methodologies/Booch/BoochBook/BoochBookContents.html

[2] The notion of a "model" of the software development process comprising distinct sequentially ordered stages (each of which results in a well defined output which is usually the input to the "next" stage) is described in any book on "Software Engineering". Such "lifecycle" models seek to draw on approaches to the development and production of more conventional engineering artefacts, i.e. the notion of *process partitioning* is used to order and differentiate between various stages in the development process.

[3] The "cloud" shaped symbols are a distinctive feature of the Booch notation !

[4] The notions of *static* and *dynamic* properies arise from the very nature of objects, i.e. their definitions comprise a static component in which signatures (syntactic properties) are given, together with a dynamic (semantic) component which describes the behavioural properties associated with function types (methods) and object types. The behavioural (dynamic) properties (which are generally not captured by the signature only) may be important for the correct execution of a piece of software. For example, when a stack is substituted with a queue (in which both have the same signature if *push* and *insert* are renamed into *put*, and

pop and *retrieve* are renamed into *get*) then we will get incorrect results when a program depends on the FIFO behaviour of the stack. Another way of characterising the same notions is in terms of the kinds of "checking" that occurs when we develop software, i.e. a description of a computation may be checked statically for syntactic and (some) semantic properties and also dynamically (at run time) for some semantic properties.

[5]

Synchronous - wait forever until supplier accepts message

Balking - abandon message if supplier cannot service message immediately

Time out - same as balking but wair a specified amount of time

Asynchronous - supplier queues the message and client then proceeds without waiting

[6] Such abstractions specifically do not include algorithmic notions.

[7] A very unusual viewpoint which contrasts sharply with the notion of software as essentially an automaton. At least one world famous computer scientist has expressed serious doubts about the use of anthropomorphic notions albeit in the context of teaching computer science, see "On the Cruelty of Really Teacing Computer Science" by Dijkstra.

[8] Note again the use of the term "real world" which is, of course, open to various forms of philosophical discussion ! The so called "real world" manifestly contains objects whose complexity suggests that they may be unamenable to description by functional decomposition alone, e.g. the brain that you are using to percieve and reason about what is written here ! However, if we start to view the "real world" as containing objects whose structure and behaviour are inherently complex then we are in danger of entering a domain of dicourse in which everything is "magic". See Arthur C Clarke for a definition of a :"technology" which, if suitably advanced, is akin to "magic" !!

[9] The term "abstraction" requires some form of definition. Consider, first, the notion of an algorithm, as a concept due to a particular form of abstraction, i.e. executional abstraction[Dijk79]. The purpose of executional abstraction is to map different computations upon each other. To do this, it must be possible, first, to distinguish between different computations, but the ability to distinguish computations from one another is not in itself enough, rather, it is a prerequisite of a more powerful form of reasoning, namely, collecting together similar computations into a class, abstracting away from the mutual differences between members of a class based upon properties of the class as a whole, and, as a result, making assertions about each member of a class including the specific computation under consideration. Thus, a computation is what an algorithm abstracts over, i.e. an algorithm embodies a class of computations which may take place 'under its control'. In the context of object oriented analysis and design methods we must be able first, to distinguish between different objects, collect together similar objects into classes by abstracting away from the mutual differences between members of a class based upon properties of the class as a whole, and, as a result, making assertions about each member of a class including the specific object under consideration !