UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 3: Object Orientation and Software Design

" 'An Object Orientation' defines a new visual and movement aesthetic used to show the man and machine relationship through performance... The actors playing objects in the play continually recast themselves in new roles - becoming in a way, "reusable" characters, modifying their behaviour according to changes dictated by the various scenes and interactions."

From "Jones on Object Orientation and Object Oriented Programming", via the WWW.

Introduction

This lecture examines, first, how "traditional" software design methods and object oriented design methods seek to abstract away from implementation specific detail. Secondly, the notions of a *class*, a *class instance* (*object*), *aggregation*, *generalisation* and *constraints* and *triggers* are introduced in the context of their use in an "informal" style in object oriented approaches to design. Finally, a method for "translating" object oriented designs into implementations is proposed. The method embodies the fundamental and general design technique of "stepwise refinement" in order to abstract away from particular design and implementation languages.

"Traditional" 'vs.' Object Oriented Design

The development of an "information system" is usually partitioned into distinct "stages" each of which culminates in a *description* of some kind[1]. Typically, the designers of such systems place an emphasis on descriptions of two related notions, i.e. *data* and *processing*. This dual perspective reflects an attempt to separate these two concerns[2] and results in at least two different kinds of description. Ultimately, since the operations will be applied to the data, it is necessary to combine descriptions of data and of operations so that any inconsistencies, ambiguities and omissions can be resolved, e.g. when developing an *implementation* of a software system.

Traditional approaches to describing data include *E/R modelling*, *relational analysis* (normalisation) and *database schema design*. Processing is usually described in terms of *transformations* of input data into output data and a description of this processing is refined by successively decomposing each *process* into simpler subprocesses until each simpler process can be implemented directly. Each process is described in terms of *functions* which transform input data into output data. Typically, functions are grouped together if they are constituents of the same "higher level" function. However, functions grouped together in this way may operate on different data objects, thus, the principle of *data abstraction*, where functions operating on a data object are *encapsulated* by that object, is set aside in a conventional approach to systems analysis. This results in descriptions which are over complex, especially for large-scale information systems.

A Simple Example

When developing an information system we will often wish to represent *sets* of "things", e.g. a set of "hobbies" in the *attribute* of an *entity type* "person". Such sets usually contain *members* whose definition comprises only their (distinct) *name*, i.e. the elements of the set have no "structure" and hence are "atomic". A relational database would require that "hobbies" are treated as a *repeating group* and that *values* are extracted from this group into a separate relation, e.g.

```
PERSON (ID#, NAME, .......)
HOBBIES (ID#, HOBBY_NAME)
```

Such definitions result each set instance being held separately by each person despite the fact that it is a simple attribute of a person ! The significance of this example lies in the fact that the members of the set are NOT entity types, i.e. the proper rôle for the set is as a domain type[3].

When an object-oriented approach to design is adopted, the first stage is to determine which *classes* of object will be included in the design[4] not which objects (class instances). A class of objects has common *structure* and *behaviour*, thus, some "thing" from the "real-world" may be abstracted into the design as a class if, and only if, it represents a *set* of similar physical objects or abstract concepts which have meaningful properties and operations that must be maintained[5].

```
CLASS person;
  PROPERTIES
    id_number: integer;
    name     : string;
    hobbies  : set (hobby);
  OPERATIONS
    create                   -> person;
    id_number_of(p: person)  -> integer;
    name_of(p: person)       -> string;
    hobbies_of(p: person)    -> set(hobby);
END.
```

Conventional design techniques, e.g. data modelling[6], seek, at their most abstract level, to be implementation independent, i.e. they seek to identify "entities" and "relationships" without regard to notions associated with particular or specific "data models", e.g. elimination of repeating groups, the decomposition of many-to-many and non-binary relationships, the introduction of explicit identifiers for entities, etc. At this highest level of abstraction, traditional approaches to modelling are similar to object-oriented approaches, however, the fundamental difference is that object-oriented approaches are inherently unconstrained by implementation considerations.

Class and Class Instance

Consider, next, how classes (once identified !) may be organised as an *inheritance hierarchy* in which properties or operations are present because they apply to certain instances of an object class. In such a hierarchy, the *behaviour* of each class is characterised by the operations which may be performed on objects of that class or by objects of that class "in" another class. Operations must be separated into those which are "internal" to the class and those which are available "externally", and the semantics of each operation must be determined, i.e. the meaning of each operation (function) must be given in terms of it either:-

? Constructing a new instance of a class

? Yielding the values of attributes over the class

? Transforming a class instance

The notions identified above are used in an "informal" style[7] when object-oriented designs are developed. In the example below, a `hotel` is defined as a *class definition* which excludes considerations of *instances*, i.e. actual hotels. One or more applications may use the `hotel` class definition with the instances described as part of an application's implementation.

```
CLASS hotel
  PROPERTIES
    name      : string
    address   : string
    owner     : company
    manager   : person
    facilities: option_type;
    ...
  OPERATIONS
    create(...)
    reserve_room(room#:  integer;  guest:  person;  arrive_date,  depart_date:
date)
    ...
END
```

Note how, in this example, a `hotel` has five `PROPERTIES` some of which are defined in terms of other "assumed" types typically provided by programming languages and database systems, e.g. `string`, `integer`, some of which are assumed to be defined elsewhere as explicit `CLASS` definitions, e.g. `company`, `person`, and some of which are "built" using type constructors, e.g. a `set` type constructor. The type `option_type` in the above example could be defined by enumerating the values or more usefully via a set type constructor, e.g.

```
option      = (swimming_pool, sauna, tennis, golf, ...);
option_type = SET OF option;
```

The classes `company` and `person` might take the general form:-

```
CLASS company
  PROPERTIES
    name, headquarters, telephone: string
    ...
  OPERATIONS
    ...
END

CLASS person
  PROPERTIES
```

```
    name, address: string
    date_of_birth: date
    ...
  OPERATIONS
    ...
END
```

Class instances (objects) developed from the class definitions shown above might take the general form shown below:-

```
HOTEL INSTANCE
name        Kings Head
address     1 West St, Glossop
facilities  (swimming pool, Golf)
owner       -------------------------> COMPANY INSTANCE
manager                                name  ACME Holdings
  |                                    Headquarters Londom
  |
  |
  '-----------------------------------------------> PERSON INSTANCE
                                                     name: John Smith
                                                     address: 23 Main  Street,
Hadfield
                                                     date_of_birth: 28/11/60
```

In addition to classes and class instances (objects) a designer may organise a description in terms of the *aggregation* of two or more objects into a composite description, or by *generalising* a set of objects with similar properties as a *generic* object, and also by introducing *constraints* and *triggers*, e.g.

```
CLASS car
  INHERIT vehicle
  PROPERTIES
    fuel: (leaded, unleaded)
    size: (compact, medium, large)
    extras: set (option)
  CONSTRAINTS
    IF year > 1985 THEN fuel = unleaded
    IF make = 'saab' THEN extras include sunroof
  TRIGGERS
    IF year = 1985 AND make = 'vauxhall' THEN ....
END
```

This is all rather informal and results in descriptions which must be *translated* into implementations, i.e. a design developed using an object-oriented approach is usually written in an informal language or notation technique.

Developing an implementation from such a design may involve a large number of **decisions** on the part of the implementor, for example, how to represent the constructed types, how to embody the constraints and triggers, etc.

Implementing Object Oriented Designs

The method used here is to *step-wise refine* a collection of types which realise the classes from the design above. The method can be summarised as follows:-

1. Define an initial type.

Given the above design, the type "hotel" is implicitly the initial type in the implementation, however, the very nature of a design means that it is open to more than one interpretation, and, as a consequence, our initial choice may not be appropriate. For example, we could have chosen "person" as our initial type and developed an implementation which reflects this decision.

```
TYPE hotel;

END.
```

2. Define the attributes (or instance variables) of the initial type

This is again an implementation decision[8] based upon an interpretation of the design and a knowledge of how the resulting types will be manipulated in a program.

```
TYPE hotel(name       : string;
           address    : string;
           owner      : company;
           manager    : person;
           facilities: option_set
          );
END;
```

3. The attributes identified in 2. may lead to the need for further "user defined" types.

Each new type must be added to the collection of types.

```
TYPE company;        TYPE person;        TYPE option_set;
END.                 END.                END.
```

4. The process of identifying attributes and new user defined types continues until no further refinement is possible.

```
TYPE company(name, headquarters: string; telephone: integer);
END;

TYPE person(name, address: string; date_of_birth: date);
END;

TYPE date(day, month, year: integer);
END;
```

5. Aggregate types can now be defined in terms of compositions of user defined types.

```
TYPE option_set;
END;

TYPE no_options;

  SUPERTYPES option_set;

END;

TYPE some_options(first: option; rest: option_set);

  SUPERTYPES option_set;

END;
```

6. If instance variables have been defined, methods are required to construct objects such that, initially, they are "well-defined", i.e. each instance variable of the object is a value of the type of that variable.

The most "reliable" implementation strategy is to ensure that objects are initially in a well-defined internal state.

7. Types are elaborated with a "method interface" which allows the internal state of the corresponding objects to be manipulated.

The methods in the method interface can be partitioned into several kinds. We can identify the following kinds of method:-

? **Inspectors**: allow an object to obtain the value of an attribute in another object

? **Adaptors**: allow an object to change the value of an attribute in another object

? **Auxiliary**: compute a value based on the internal state of an object

? **Generators**: compute a unique object of the given type

? **Interactors**: generate representations of objects

Summary and Conclusions

Traditional approaches to "systems analysis" have been largely superceded by more recent approaches which seek to exploit the object-oriented paradigm. The justification for adopting an object-oriented approach lies in the form of the resulting descriptions, i.e. they are models which embody abstractions of "real-world" objects at differing levels of refinement and (arguably) reflect their real-world counterparts more accurately than models developed using traditional approaches. Most significantly, such models have the inherent advantage of combining the notions of data and of operations via the concept of a class definition.

Ultimately, however, such designs will be implemented in an object-oriented programming or database language, and this may involve a large overhead since the implementor must *translate* informal notions embodied in the design into a description written in a programming language.

Chris Harrison, January 1997.

----

[1] Approaches to designing of other kinds of system, e.g. "real-time" systems, "event-driven" systems etc., also typically partition the development process into distinct stages.

[2] We must be careful, when we separate concerns, to ensure that we separate only those concerns which can usefully be separated !

[3] As you should be aware, a *relational database* consists of a set of *domains* and a set of *relations*.

```
<database> = <domain set> X <relation set>
```

An individual *relation* is a *variable* in that its *value* is assumed to change over time. However, a *domain* is NOT a variable and hence does not change its value over time. A *domain* consists of a domain name, a fixed non-empty set of domain values and an ordering.

```
<domain set> = SET OF <domain>
<domain>     = <domain name> X <domain values> X <ordering>
.
.
<domain values> = SET OF <value>
```

Each value in a domain is "atomic" and cannot be decomposed, and all values in a domain are of the same "data type" (we may assume a domain may contain "null" values). An ordering indicates whether a domain supports a "greater than" operator between pairs of domain values.

[4] Therein lies the same fundamental difficulty associated with any approach to developing abstractions of the "real-world", i.e. there is no single universal rule by which we can determine which "things" should included in our abstraction and which should not !

[5] It is in this context, i.e. as an aid to determining which "things" can usefully be grouped together because of identifiable characteristics, that object oriented approaches provide an arguably more "natural" means of developing abstractions.

[6] *Data modelling* involves the development of some form of "semantic" data model which captures the properties of, and interrelationships between, "things" from the "real" world. Such models must usually be translated into an alternative implementable form when a software system is developed from them.

[7] The term "informal" is NOT intended to denote that such styles are of less practical use. Rather, to denote that they seek to abstract away from considerations of how such designs will ultimately be implemented.

[8] The approach used here supports a "functional" style of object generation. An alternative approach would use instance variables to represent specification attributes. In this case, the type "hotel" would look like thus:

```
TYPE hotel;
    VAR name      : string;
        address   : string;
        owner     : company;
        manager   : person;
        facilities: option_set;
END;
```

In common with other object-oriented languages, instance variables are assumed initially to have an undefined value. Initialisation is the responsibility of a method, e.g., "initial", defined in the type in which the instance variables are declared.