UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 4: Case Study One

This case study is concerned with the systematic derivation of an *implementation* of a software system. The system has an initial description in the form of a *software design* written in a "simple" object oriented design notion. In this simple notation, a *class* is described in terms of its *properties* and its *operations* each of which may be further described in terms of other user defined types and also in terms of simple and structured types provided by conventional programming languages.

Recall, first, the partial design from lecture 3:-

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

option_= (swimming_pool, sauna, tennis, golf, ...)

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

CLASS person
  PROPERTIES
    name, address: string
    date_of_birth: date
    ...
  OPERATIONS
    ...
END
```

An implementation of the `hotel` class is developed below. This implementation is written in the "model" object oriented programming language outlined in Lecture 2[1]:-

STEP 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;
```

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

This is again an implementation decision[2] 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;
```

STEP 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 through a "type-supertype" relationship.[3]

```
TYPE company;
END;

TYPE person;
END;

TYPE option_set;
END;
```

STEP 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;
```

Now, enumerated types are not directly supported in the model object oriented language - enumerations are constructed out of user defined types. In general, the approach used to implement enumerations is:-

4.1. Define an "abstract" type to represent the enumeration. Abstract types are not intended to be instantiated, but act as a means of aggregating the types representing elements of enumerations.

```
TYPE option;
END;
```

4.2. For each element of the enumeration, a subtype of the abstract type is defined.

```
TYPE swimming_pool;
  SUPERTYPES option;
END;
TYPE sauna;
  SUPERTYPES option;
END;
TYPE tennis;
  SUPERTYPES option;
END;
TYPE golf;
  SUPERTYPES option;
END;
```

STEP 5. The chosen representation type for a set of options realises the type as a list structure.

An abstract type "option_set" is used to aggregate types which represent *empty sets* and *non-empty sets*.

```
TYPE option_set;
END;
TYPE no_options;
  SUPERTYPES option_set;
END;
TYPE some_options(first: option; rest: option_set);
  SUPERTYPES option_set;
END;
```

STEP 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.

With the implementation strategy used here, objects are *guaranteed* initially to have a well-defined internal state as each attribute must be supplied with an actual argument in the expression which generates that object.

STEP 7. Types are elaborated with a "method interface" which allows the internal state of the corresponding objects to be manipulated. Methods can be partitioned into several kinds:-

7.1 Inspectors: allow an object to obtain the value of an attribute in another object, e.g.

```
TYPE person ...
  FUNCTION name_of: string;
  BEGIN
    result:=first
  END;
```

7.2 Adaptors: allow an object to change the value of an attribute in another object, e.g.

```
TYPE person ...
  PROCEDURE set_name(new_name: string);
  BEGIN
    name:=new_name
  END;
```

7.3 Auxilliary: compute a value based on the internal state of an object, e.g.

```
TYPE some_options ...
  FUNCTION cardinality_of: integer;
  BEGIN
    result:=1 + rest.cardinality_of
  END;
```

7.4 Generators: compute a unique object of the given type, e.g.

```
TYPE browser_window ...
  h:=hotel(...);
```

7.5 Interactors: generate representations of objects, for example concrete representations, storage representations, e.g.

```
TYPE hotel ...
  PROCEDURE display(w: text_window); ...;
```

Implementation Of The Hotel Example

```
TYPE browser_window;

  SUPERTYPES text_window;

  VAR h: hotel;

  PROCEDURE initial;
  BEGIN
    h:=hotel('Kings Head',
             '1 West St, Glossop',
             company('ACME Holdings', 'London', 01811002000),
             person('John Smith', '23 Main Street Hadfield', date(28, 11, 60)),
             no_options.add(swimming_pool).add(golf)
            )
  END;

  PROCEDURE click(x, y: integer); self.clear; h.display(self);

END;
```

The type `browser_window` is defined as a subtype of a predefined type `text_window`. It "manages" a single object `h` of type `hotel` via its `initial` method which is "guaranteed" to be executed "first". The `click` method manages user interaction.

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

  PROCEDURE display(w: text_window);
  BEGIN
    w.write('hotel(');
      w.write(name   );      w.write(', ');
      w.write(address);      w.write(', ');
      owner     .display(w); w.write(', ');
      manager   .display(w); w.write(', ');
      facilities.display(w);
    w.write(')')
  END;

END;
```

The type `hotel` provides a single method `display` which generates a representation of objects of type `hotel` via calls to the predefined operation write (which takes a string as a parameter) and via display methods defined in the types `owner`, `manager` and `facilities`.

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

  PROCEDURE display(w: text_window);
  BEGIN
    w.write('company(');
      w.write    (name        ); w.write(', ');
      w.write    (headquarters); w.write(', ');
      w.write_int(telephone   );
```

```
    w.write(')')
  END;
END;
```

Likewise, the type `company` provides a single method `display`.

```
TYPE person(name, address: string; date_of_birth: date);
  FUNCTION name_of: string;
  BEGIN
    result:=name
  END;
  PROCEDURE set_name(new_name: string);
  BEGIN
    name:=new_name
  END;
  PROCEDURE display(w: text_window);
  BEGIN
    w.write('person(');
      w.write(name   ); w.write(', ');
      w.write(address); w.write(', ');
      date_of_birth.display(w);
    w.write(')')
  END;
END;
```

The type `person` provides a method `display` and two additional methods `name_of` and `set_name`. The method `name_of` returns a value of type string representing the name of the person object. The method `set_name` assigns a string value as the name of the person object.

```
TYPE date(day, month, year: integer);
  PROCEDURE display(w: text_window);
  BEGIN
    w.write('date(');
      w.write_int(day  ); w.write('/');
      w.write_int(month); w.write('/');
      w.write_int(year );
    w.write(')')
  END;
END;
TYPE option;
  PROCEDURE display(w: text_window);
  BEGIN
    w.write(self.name_of)
  END;
  FUNCTION name_of: string;
  BEGIN
    result:=''
  END;
END;
```

```
TYPE swimming_pool;
  SUPERTYPES option;
  FUNCTION name_of: string;
  BEGIN
    result:='swimming_pool'
  END;
END;
TYPE sauna;
  SUPERTYPES option;
  FUNCTION name_of: string;
  BEGIN
    result:='sauna'
  END;
END;
TYPE tennis;
  SUPERTYPES option;
  FUNCTION name_of: string;
  BEGIN
    result:='tennis'
  END;
END;
TYPE golf;
  SUPERTYPES option;
  FUNCTION name_of: string ='golf';
END;
```

The type `option_set` supports three methods `add`, `display` and `cardinality_of` and is further refined by inheritance into two "variant" subtypes, i.e. `no_options` and `some_options`. Note how the "variant" subtype `no_options` redefines the meaning of the `add` method such that an object of type `some_options` is generated and how the method `cardinality_of` in `some_options` is given a "new" definition.

```
TYPE option_set;
  FUNCTION add(o: option): option_set;
  BEGIN
    result:=self
  END;
  PROCEDURE display(w: text_window);
  BEGIN
  END;
  FUNCTION cardinality_of: integer;
  BEGIN
    result:=0
  END;
END;
```

```
TYPE no_options;
  SUPERTYPES option_set;
  FUNCTION add(o: option): option_set;
  BEGIN
    result:=some_options(o, self)
  END;
  PROCEDURE display(w: text_window);
  BEGIN
    w.write('{}')
  END;
END;
TYPE some_options(first: option; rest: option_set);
  SUPERTYPES option_set;
  FUNCTION add(o: option): option_set;
  BEGIN
    rest  :=rest.add(o);
    result:=self
  END;
  PROCEDURE display(w: text_window);
  BEGIN
    rest.display(w);
    w.write(' + ');
    w.write('{');
      first.display(w);
    w.write('}');
  END;
  FUNCTION cardinality_of: integer;
  BEGIN
    result:=1 + rest.cardinality_of
  END;
END.
```

<u>Summary</u>

The example above serves to demonstrate how a number of "decisions" must be made by an implementor based on implicit information in the original design and also upon the "target" language used for the implementation. However, the technique of refinement used in this case study is equally applicable to any general purpose notation for describing classes and objects, indeed, if the notation <u>is</u> general purpose then it <u>must</u> provide explicit support for refinement.

It is left as an exercise for students to develop alternative implementations of the design in any other object oriented programming language of their choice, and to compare and contrast the chosen language's support for refining a design into an implementation with the support provided by the "model" language used in this lecture.

Chris Harrison, January 1997 (Updated Feb. 2002)

Appendix 1. Information Systems

The very nature of so-called "information systems" has always been rather (intentionally?) vague - after all, any system that stores, manipulates and/or generates information (whether computer-based or not) is arguably an information system.

Over time, and especially by the early 1980's, the term "information system" had become synonymous with software systems whose (structured) development "process" involved distinct but interrelated stages, each of which involved the successive transformation of a description of what was "required", into a "software design" and ultimately into an implementation. The resulting descriptions captured the notions of the "data" to be processed and the "operations" on that data in such a form that it was relatively straightforward to hold the persistent representation of the data in a relational database and to support user-interaction by "queries" on the (stored) data.

The relatively tranquil world of computer-based "information systems" as they were perceived in the early 1980's has been transformed from by the advent of the internet and especially the notion of "electronic commerce" because the nature of the software systems needed to support such applications required radical changes in the way that "information systems" are designed and implemented.

Fortunately(?), some of the conventional tools available to information systems developers remained largely unchanged, for example, the notion of a database and especially a relational database remains the same regardless of whether it is the heart of some information system a-la the early 1980's, or the "back-end" of an internet-based site. What has changed fundamentally is the organisation of the <u>infrastructure</u> that software systems of the latter kind must interact with, i.e. such systems must adhere to the client-server model that underpins modern "information sytems".

Other fundamental changes that have required "information system" developers to rethink their approach to system development include changes in the kinds of programming language that have become widely used, the advent of associated (OO) design notations, and the need to support more sophisticated forms of user-interaction (e.g. browsers). There has also been a radical change of emphasis in the sense that modern information systems are seen as "mission critical" (as the jargon goes), i.e. in the past it was possible to muddle through if one's information system "fell over" - one simply reverted to the paper version of the original system (if one was available) until the database was "rolled-back" and the system was restarted, or managed the situation in some other ad-hoc manner[4]. Now, the very software system that enables "e-commerce" is, in effect, "the business" in the very real sense that if it fails then so too does income generation[5].

Thus, it is now necessary to consider a vast range of possible alternative technologies (e.g. JSP, PHP, applets, servelets, browsers, SQL server, etc, etc, etc), many with propriety constraints, as the basis for a modern "information system" and where such systems interact with more conventional legacy "information systems" so-called "middle-ware" may be required to enable the different systems to interact safely with one another.

1.1.1.1.1.1 A.1 UML

One relatively useful development in the world of "information systems" has been the recognition that data-oriented notations, especially those based on the notions of entities and relationships, fail to capture sufficient meaning especially when such descriptions must be translated into implementations written in modern object-oriented programming languages[6]. As if to the rescue came UML (Unified Modelling Language) - a notation technique that attempts to embody the "best" of traditional "structured" approaches to information systems development (essentially E-R modelling) with those notions that cannot be ignored from the object oriented (programming) language paradigm, i.e. class, class instance(or object), and inheritance.

In a UML model of an information system, there are object classes and individual objects (or object instances) of that class. In a resulting implementation, the object class may itself be stored as an object such that individual instances of the object class are "similar" to individual "entities" that were instances of an "entity set". Each (of the potentially very large number) of object instances is considered to be stored as a separate object with object methods stored once in the object class (rather than being replicated in each instance of that class) such that instances inherit their methods from their object class.

An object instance is identified by a value of one of its attributes which, using the UML notation, is underlined to denote its (unique) identifying attribute.

CLASS NAME	product	<u>p99: Product</u>
PROPERTIES	product no: string;	product no: p99;
	supplier: string;	supplier: s42;
	cost: sterling;	cost: £101;
METHODS	add_product(product no)	add_product(product no);
	Delete_product(product no)	delete_product(product no);

An object class diagram shows object classes and their interrelationships in much the same notational style as a conventional E-R diagram, i.e. it uses association, containment and inheritance rather than relationships, subsets and dependent entities but has a similar notation.

A continuous line between classes represents an *association relationship* which has a name and is arrowed to indicate the "direction" of the association, and has an explicit cardinality. A dotted line represents an *association class*. A *composition relationship* (or containment, or aggregation) denotes an object composed of objects from other objects and is represented by a continuous line from a property to an object class with a diamond symbol at the object class end of the line. In general, it is assumed that association relationships are "temporary" and that composition relationships are "permanent".

In the mind of a UML modeller, inheritance and subsets are similar notions (Eeekkk!). In particular, it is accepted that an object can inherit the *features* of another object (rather than the state and behaviour as is realised in OO programming languages), add additional features, and replace inherited features.

Perhaps the most muddled thinking to be found in UML books is that surrounding "polymorphism", i.e. in such books polymorphism is variously described as being "similar to inheritance", "closely related to inheritance" etc. Inheritance in OO programming languages **<u>IS</u>** inclusion polymorphism, and a given message may cause different effects depending upon the instance of the object it is sent to. Fortunately, in the mindset of a UML user, the same notions, i.e. of a *message* and an *object* and an *effect* that may (for the same message) change depending upon the particular object remain the same. Perhaps because of the desire to make the notation as expressive as possible, the notion of multiple inheritance is supported by UML, and it introduces the same fundamental problem that it causes in an OO programming language that supports the same notion, i.e. the potential for ambiguity. For example, in UML if one or more "parents" (i.e. parent classes) have a "feature" with the same "name" then it must be possible to determine which "inheritance links" are defined, i.e. under what circumstances a given feature is inherited - this is simply denoted in UML by the sequence in which the inheritance links are defined.

In UML, "behaviour" is modelled in terms of "use cases", i.e. the starting point for describing processing is in terms of the (so called) "real-world" interactions between the system and its users. A UML use-case model comprises "activities" and "actors" who take part in such activities - a level of abstraction far above the corresponding implementation of a software system that can support such notions. The resulting model of a use case is given in terms of what appear to be similar "statements" but which are in fact and often very dissimilar statements in nature[7], e.g. for the classic(al) example in a UML text book of an ATM:-

Use Case - Using the ATM

A greeting message is waiting on the ATM

The customer inserts their card into the machine

The ATM asks the customer for their PIN

The customer enters their PIN

Etc

Etc

It is evident that the first of these statements involves rather different behaviour from the underlying software that must implement it than the second.

When modelling larger systems[8], it quickly becomes evident that many use cases have common parts, and hence the notion of a use case relationship and even an extends relationship can be supported.

Because the level of abstraction of a use case, however detailed it may be, is far above that of the corresponding implementation in terms of classes and class instances (objects), it is simply not possible to go directly from a use case to an object class diagram - a big problem! For this very reason, UML provides a large number of techniques and a "rich" (for "rich" read large) notation for translating from a use case to an object class diagram, the most straightforward of which is a sequence diagram which denotes messages exchanged between objects in a use case. Only general rules can be given for sequence diagrams (presumably hence the "creative" nature of the way in which such techniques can be used?) for example:-

? Each actor becomes an object (but not necessarily so.)

? Time moves "downwards" implicitly

? No decisions are shown

? Lines between objects are messages so they should be labelled with nouns

Alternatively (or in conjunction with.) sequence diagrams, UML supports transition diagrams for modelling the "dynamics" of a (use) case. Briefly, a transition diagram shows all of the "exceptions" in a single diagram.

To "glue" all that might have been described thus far in a UML model of an "information system", there are also "event flow diagrams" with their own "rules", e.g.

? All objects in each sequence diagram appear only once in the corresponding event flow diagram

? All messages are shown in the event flow diagram

and collaboration diagrams which are greater in number than the corresponding event flow diagrams and which show the (wait for it.) sequence of message flows.

1.1.1.1.1.2 A.2 Other Techniques

The are as many different techniques for OO modelling as you care to identify or even invent. From the recognition that "structured methods" were inadequate, through early attempts, e.g. Coad and Yordan in the early 1990's, to Grady Booch in the mid 1990's, and other applications-oriented techniques, e.g. Object Oriented Software Engineering (OOSE[9]) also in the early 1990's, and the so-called "rapid application development" techniques that have recently appeared that claim to support "software evolution" and "synthesis" (whatever that means.).

1.1.1.1.1.3 A.3 On scale and Complexity

See footnote (3).

One of the most common "cheats" in a text book is to use carefully chosen simplified examples of the application of a technique and suggest or imply that it scales-up naturally to cope with much larger problems. In addition, and in some cases explicitly, it is also suggested that when problems of scale arise, the solution is to automate the design process using software tools that support the technique. It has long been recognised in the "software engineering community", although many would perhaps disagree, that software tools can indeed automate a technique, and in doing so ensure syntactic and some semantic correctness for the descriptions developed using such tools. But, and it is a BIG but, there is little evidence that tools per-se increase productivity. Where tools and toolsets or environments are available, the inherent cost associated with their use often outweighs any savings even if tool users do not like to admit it.

A better approach to design recognises the need for a *specification*, and more properly, a *formal specification* of the desired properties of a software system which embodies a design. Of course, developing such specifications has its inherent problems, including scale and complexity. But for some systems, an informal design is simply not adequate and may not be legally acceptable, and as the reliance upon software systems increases to the extent that "mission critical" in the business sense equates with "safety critical" in the most general sense, all software must be robust and reliable (whatever that means!).

Thus, we remain at the mercy of the same problem that object orientation was seen as a solution, however partial, to, i.e. **as the size of a piece of software increases the number of potential interactions between its components increases exponentially.** The resulting complexity is difficult to document, difficult to reason about and hence impossible to resolve using any known technique, and it will be far in the distant future before we even foresee likely technologies for solving this problem. What remains are partially effective techniques, often application specific, which can ameliorate some of the difficulties, especially when they are used properly, but which provide no general purpose solution. At the same time, diversity of applications and the complexity of computer-based systems continues to increase by orders of magnitude within the lifetime of an individual.

It is not all "gloom" though, i.e. it is up to YOU to invent and develop new solutions that are better than the solutions we currently employ - that is why you read for a degree - so that you know what the fundamental problems are, what the limited techniques we have available are and can do, and hence understand what needs to be done to make things better. Here's hoping.

---

[1] A more detailed consideration of this language is provided in a later lecture.

[2] The approach used here supports a "functional" style of *object generation*. An alternative approach would use *instance variables* to represent 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 initially undefined. Initialisation is the responsibility of a method, e.g., "initial", defined in the type in which the instance variables are declared.

[3] Recall how, in our "model" language, types without any explicit supertype are implicitly subtypes of an all-encompassing "any" type. The type any is a subtype of itself.

[4] I know of one organisation that employs programming staff on a 24-hour basis solely to prevent known database "errors" causing the system to fall-over, and when it inevitably does, the same staff know how to quickly roll back the errant transaction so that it can be dealt with "manually" and the system restarted so that all of the remaining transactions can be processed before the next "batch" of updates appears and needs to be processed.

[5] Perhaps the best piece of jargon for the classic(al) dot.com is "burn-rate", i.e. the rate at which (like a Saturn 5

booster burns through its fuel once ignited) such organisations use up the venture capital that enabled their start-up without generating any profit or return on that investment.

[6] Is an entity a "class" or an "instance", and how does a relationship hold in terms of a subclass?

[7] There are documented examples of fundamental problems with early ATM's whose software was presumably designed in this manner, i.e. a customer withdraws cash up to their limit, then reinserts their card and continues to withdraw amounts up to their limit several times. In the one example I remember reading, the customer was intelligent enough to realise that a record would be kept of each transaction and returned to the bank in question as soon as it opened on Monday morning. On hearing the customer's story, the bank manager at first claimed that it was "impossible" to repeatedly withdraw funds once a customer's limit had been reached for that day - only to have the cash dumped on his desk by the customer as proof that the ATM system was in fact permitting such transactions.

[8] Information system developers seem always to be concerned with difficulties that have long been recognised by software engineers as being central to their discipline, i.e. scale and complexity. The inability of widely-used techniques that apply in the small to scale up naturally to support larger-scale systems suggest that such techniques are fundamentally flawed, not least, because the only useful large scale systems we have are composed of simpler (sub)systems and hence what applies on the lager scale MUST apply on the smaller scale in exactly the same manner. Any other form of large scale system system is simply incomprehensible and hence useless.

[9] Why "software engineering" should be fundamentally different if one is using object oriented techniques is particularly curious.