Object-Oriented Software

A "Pocket" Guide To Object Orientation

Part 1. Objects and Messages

The notions of a "message" and "message passing" seem. initially, to be almost self explanatory. However, it is important to be able to relate such notions to other ideas which underpin the object oriented paradigm, in particular, the notion of a *method*.

1.1 A Single Initial Notion

We assume a single initial *notion*, i.e. that of an *object*. An object receives and sends *messages*. To define an object we describe its *class*. Each individual object of a class is termed an *instance*. If we create an object "O" of the class "C" then the class "C" defines what it is to be an object "O" in terms of all of the *messages* M0, M1, ...Mn an "O" object can "act" upon. To act upon a message an object invokes a *method*. We can, of course, create several objects O1, O2, ..., On from the class "C".

1.2 A Pure Language

In a "pure" object oriented language, e.g. the expression-oriented language SmallTalk, we might write:-

```
a <- 1;
b <- 2;
c <- a + b;
```

In SmallTalk, the expression `c <- a + b;` is an *assignment expression* which is evaluated thus:-

1 The expression `a + b` requests the receiver **object** (the result of evaluating the subexpression *a*) to perform the operation `+` with the **object** yielded by*b* as an argument.

2. The expression `a + b` when evaluated yields a result which is always a reference to an **object**.

3. The assignment of the result of the evaluation of the expression `a + b` has the side effect of making the variable *c* refer to the same **object** as that which results from the evaluation of the expression `a + b`

[pic]

2.0 A Model Language

In a "model" object oriented language[1], an *integer expression* of the form `a + b` can be an *argument* to a *method application*, or part of an *assignment statement*, or part of the *body* of a *function*.

In the example below, the expression `a + b` is an argument in an application of the `show_integer` method. This method takes three arguments, an integer (expression), and an "x" and "y" coordinate at which the integer is to be "displayed", e.g.

```
show_integer(a + b, 90, 40)
```

A method must be applied to some *object* that supports that method. In a "model" language, the predefined type `window` supports a method `show_integer` together with another predefined operation called `display`. Thus, a subtype of the type `window` called `int_in_window` can redefine the meaning of the `display` operation inherited from the supertype, e.g.

```
TYPE int_in_window;
  SUPERTYPES window;
  PROCEDURE display;
  BEGIN
    self.show_integer(a + b, 90, 40)
  END;
END.
```

To "drive" the application of the `display` method, an `initial` method is defined for <u>objects</u> of type `int_in_window`. The `initial` method (which is guaranteed to be executed <u>first</u> when objects of type `int_in_window` are <u>created</u>) assigns suitable "initial" values to the *instance variables* `a` and `b` and then *applies* the `display` method to `self`, i.e. an <u>object</u> of type `int_in_window`.

```
TYPE int_in_window;

  SUPERTYPES window;

  VAR a, b: integer;

  PROCEDURE initial;
  BEGIN
    a:=1;
    b:=2;
    self.display
  END;

  PROCEDURE display;
  BEGIN
    self.show_integer(a + b, 90, 40)
  END;

END.
```

Chris Harrison, January 1997.

----

[1] The term "model" is used to denote a language which is intended to exemplify good design. Typically, such a language is suitable for both teaching and also the systematic construction of engineering of large scale software systems by skilled software developers. We will consider the syntax and semantics of such a language in more detail later lectures. (It is based upon the Eiffel language for those who wish to develop their understanding further).

Briefly, in such a language, a class is defined by a type definition, e.g.

```
TYPE example_type;
END.
```

A type may be a subtype of some other type, e.g.

```
TYPE example_type;
  SUPERTYPES some_supertype;
END.
```

A type may have instance variable, e.g. *i: integer*, and methods (realised as functions and procedures), e.g. *double*:-

```
TYPE example_type;
  SUPERTYPES some_supertype;
  VAR i: integer;
  PROCEDURE double; i:= i * 2;
END.
```