**Behaviour**

The two principal mechanisms by which a user can invoke a business behaviour are by selecting an action from a pop-up menu on a business object and by dragging one object onto another. (The latter is not the same as dragging an object into an empty field inside an object.) It is also possible to invoke business behaviours at the class level – through the pop-up menu on that class – or by dragging an instance icon onto the class icon.

1. At the bottom of the pop-up menu for an object are the business methods that can be applied to that object.

2. An ellipsis \( \ldots \) following the menu action indicates that this will return another object as a new window.

3. Dragging Home, Boston directly onto Logan Airport, Boston will trigger the creation of a new Booking that uses those two locations as Pick Up and Drop Off respectively. This shortcut is most useful when both locations show up in a list of frequently-used locations inside a Customer.
4. The method name `actionReturnBooking` is stripped of its action prefix and reformatted to generate the menu option Return Booking... automatically. For language localization it is possible to over-ride this automatic correspondence. The ellipsis in the menu option reflects the fact that an object will be returned by this method - in this case a Booking.

5. `createInstance` creates a new instance of the Booking class, called `returnBooking`. Just using the simple Java `new` keyword instead of `createInstance` would not perform all the necessary initialisation.

6. This code swaps the pick-up and drop-off locations for the return booking.

7. If an action method requires an input parameter (in this case another Location) then this behaviour will not show up on the pop-up menu. Instead, it will be automatically invoked when the user drops an object of the required type onto this object.

8. This method creates a return booking from an existing booking.

9. If another object has invoked this method then the newly created Booking will be passed back to it. If the method was invoked by the user from the menu, then the returned object will automatically show up as a new window.
3.1 The anatomy of a naked object

methods. You will see also that both the getDropOff and the setDropOff methods have had specific calls added within them:

```java
public class Booking extends AbstractNakedObject {
    private Location dropOff;

    public Location getDropOff() {
        resolve(dropOff);
        return dropOff;
    }

    public void setDropOff(Location newDropOff) {
        dropOff = newDropOff;
        objectChanged();
    }
}
```

In the getDropOff method, we can see a resolve call, which is a static method provided in the superclass hierarchy. The purpose of the resolve call is to ensure that the referenced object (in this case the Location held in the DropOff field) actually exists, fully formed, in memory. The fact that the Booking object has all its data loaded does not imply that all the other objects it references are also complete: if the framework operated that way, loading a single object could take a long time! Instead the framework endeavours to load objects from storage only when they are needed.

This is best explained with an example. In the image below the left hand side (with the blue background) shows the user’s view. The right hand side (grey background) is a portrayal of what is happening in working memory:
In this first screen, the user has retrieved the **#2 Confirmed** booking from storage and displayed it as an icon. On the right-hand side we can see that a booking object has been instantiated in memory, and that the value fields, which are an integral part of the object, have been retrieved from storage. In this case, those value fields provide sufficient information for the booking object’s title to be generated (specifically from the **Reference** and **Status** fields). For each of the objects associated with that booking, a new instance of the appropriate type has been created within the association field, but none of those objects have yet been resolved. We have portrayed this graphically by using icons without titles.

The user now opens up the booking object to show the form view. All value fields and all associated objects are now displayed:

In order to display the titles of each of the associated objects (such as the customer for that booking) the framework has automatically resolved those objects. (The booking object still exists in memory, but this is not shown for reasons of space). Each of these objects has now had each of its value fields retrieved from storage, but where the objects contain references (associations) to other objects, these have been instantiated, but not themselves resolved. If the associated object has already been retrieved then that reference is used. In our example this has happened with the **Home** telephone, the first two
A useful technique is to look at each of your know-what responsibilities (i.e. the associations and attributes) and ask yourself if it could be better stated as a know-how-to responsibility. Some examples are shown in the panel.

It can sometimes be valuable to take this principle to extremes. For example, on one occasion we challenged a public sector organization about why their Customer object definition included the date of birth as a know-what. The audience was incredulous. Date of birth was needed for lots of purposes, they said: for identification (have we got the right Fred Smith?), for authentication (is the person on the end of the phone who he says he is?), for determining if a client is an adult or not. We argued that each of those uses should have been modelled as explicit know-how-to responsibilities. It may be that the object uses a stored date of birth for these purposes, or it may not (the object might delegate the responsibility to an external agency). In fact, it is bad practice to base identification and authentication on the same data. Moreover, if they design the Customer object to be able to answer the question ‘Is [the customer] over 18?’ rather than ‘What is [the customer’s] date of birth?’ then they are not leaving themselves vulnerable to charges of misuse of age information. (Anyone who thinks that is a spurious line of argument should read the European electronic privacy legislation.)
Avoid the temptation to fill up the page with responsibilities that are automatically provided by the framework. There is no need, for example, to specify for each attribute and association a corresponding know-how-to responsibility for reading and/or writing it – that should be assumed. The only reason you should specify that the Booking object ‘knows how to choose a Seat [on the Flight]’ is if it is the Booking object itself that will do the selection rather than, say, the user.

Similarly there is no need to specify the responsibility for the object to make itself persistent, or to manage authorization and security, distribution or version control. These are generic capabilities provided by the infrastructure and are presumed to apply to all classes of business object.

You can also omit the generic class responsibilities. All naked object classes automatically provide the generic class responsibilities (unless the programmer suppresses them):

- Create a new instance of that class.
- Retrieve a particular instance by its unique reference.
- List all instances that match a set of criteria.
- List sub-classes.

You can add class responsibilities specific to a business class – for example to create a particular kind of report on instances of that class – but the need for these is more likely to emerge later in the exploration phase.

**Dealing with the concept of process**

We have already made clear that we do not agree with the practice of translating use-cases into objects in their own right, nor of the split in roles between Entity or Model objects, and Controller objects. Using Naked Objects, only the former category of objects is permitted. To put it another way, all classes of business object should be thought of as persistent classes.

That said, it is quite legitimate to think about different broad categories, or stereotypes** of business object. In particular, we often draw the distinction between purposeful and non-purposeful objects. Non-purposeful objects are things like Product, Customer, Employee, and Location. The state of those objects will change over time, and that state will be made persistent, but the changes don’t go in any particular direction. They can be thought of as random.
Network Position

This is position in the sense of trading (e.g. long, short, balanced) not geography. The Network Position is made up from the individual Location Positions and Link Positions for the same period. Knows how to calculate the profitability of the Position.

Location Position

Knows how to find and summarize Contracts active at this Location in the specified period.

Link Position

Knows how to calculate how much of the available capacity is being used. Knows how to create a ‘movement’ (an operational instruction to transmit a specified amount of power).

A typical energy trading scenario

The day’s Network Position shows that we are 60 Megawatts (MW) ‘long’ in the Tennet location and 60 MW short in the RWE location:
Right-clicking on the Link Position for the Link between the two, we select ‘Buy More Capacity’, which creates a new ‘short-term’ Contract with the owner of that Link, for which we specify a flat Power Profile of 60MW:

We now have transmission capacity to move power from where we are long to where we are short (this might take several trades at different prices). Right clicking on that link position again we can now create a Movement instructing the line operator to ‘transport’ 60 MW: