

COMP1120-2003-Assign3

Assignment 3: A UBoat Driver

1 Objectives

These include:

- to implement a *discrete time simulation* program, an increasingly important area of computer application, particularly in science and engineering.
- to use a systematic strategy for testing software.
- to implement a generic library operating on non-trivial objects.
- to implement, in a high-level way, a non-trivial algorithm using non-trivial classes.
- to illustrate the use of classes in providing *information hiding* and *data security*.
- to gain an appreciation of the design issues in the modelling of such a system.

2 Background

See the Assignment 2 document for background of the UBoat/EBoat simulation system.

Similar to the *EBOAT*, the *UBOAT* class is a sub-class of the *BOAT* class. Your main task is to write a ‘driver’ for such a boat, this boat (rather than the EBoat) now has a *OCEAN* attribute.

The *UDRIVER* (UBoat Driver) class will be a root class. It creates an *OCEAN* object, links that to a *UBOAT* object that it then creates, and proceeds to drive that boat through the channel defined by *OCEAN*.

The *OCEAN* class is now highly extended, and performs the following functions:

- parses the *command line* for parameters determining the initial position of the boats, the channel size, whether an animation window will be created and whether an EBoat is present (and if so, which strategy it will use). See Section 2.2.
- applies all interactions between the boats and the channel, setting their sonars according to their relative positions, detecting the release of charges from the EBoat, and ‘destroying’ a boat if it collides with the channel walls or it is too close to an exploding charge.
- via the *BOAT_LOG* class, provides a facility for the logging of events of interest of the two boat objects, including turns, speed changes, charge release, gain/loss of sonar contact, and destruction.
- via the *SIMDISPLAY* class, creates and updates an optional animation window.
- outputs a message when the result of the simulation (see Section 2.1) is determined.

2.1 The Simulation as a Game

The *OCEAN* class defines the channel as being the area defined by the rectangle with opposite corners given by (X_{min}, Y_{min}) and (X_{max}, Y_{max})

In ‘simulation mode’, the UBoat begins at position (x_u, y_u) and orientation $\theta_u = 0$, where $X_{min} \leq x_u \leq (X_{min} + X_{max})/2$, and $Y_{min} \leq y_u \leq Y_{max}$. The EBoat begins in a position to the right of x_u . The channel size is set at $X_{min} = Y_{min} = 0$, $X_{max} = Sim_x_max$ and $Y_{max} = Sim_y_max$.

Its objective is to cross the line $x = Sim_u_x_dest < X_{max}$ within Sim_time_max seconds.

The following circumstances signify the end of the game.

1. the UBoat is destroyed by collision with the channel’s sides.
2. the UBoat is destroyed by a depth charge.
3. more than Sim_time_max time steps elapse with neither the UBoat or EBoat destroyed, nor the UBoat reaching its destination.
4. the EBoat is destroyed (under the same circumstances as can the UBoat).
5. the UBoat reaches its objective within the time limit.

The *OCEAN* class acts as a moderator to the game.

The *OCEAN*, *UBOAT* and *EBOAT* classes contain constants which are the parameters of the game. They are intended to provide a ‘balanced’ game, rather than be necessarily realistic. To achieve this objective, their values will be subject to minor adjustments without notice; in any case, this should *not* affect your program which should under no circumstances have these constants’ values hard-coded at any place.

2.2 Command line parameters for the *OCEAN* class

The synopsis for invoking the `udriver` (or `test_drive`) program is:

```
udriver [-u  $x_u$   $y_u$   $v_u$   $\theta_u$ ] [-e  $x_e$   $y_e$   $v_e$   $\theta_e$ ] [-c  $X_{max}$   $Y_{max}$ ] [-s  $r$ ] [-w] [opte]
```

where the `-u` option sets the UBoat’s initial position (x_u, y_u) and velocity $(v_u; \theta_u)$ (and similarly for the `-e` option); the `-c` option sets the channel size. The `-w` option suppresses the animation window.

Default values are $X_{min} = Y_{min} = 0$, $X_{max} = X_{max_def}$, $Y_{max} = Y_{max_def}$, $x_u = x_e = (X_{min} + X_{max})/2$, $y_u = y_e = (Y_{min} + Y_{max})/2$, $v_u = v_e = \theta_e = \theta_u = 0$, and `opte = 0` (no EBoat). With `opte = 1`, the EBoat Driver will ignore the UBoat (and continue in a straight line from its initial velocity). Further values for `opte` will be published on the course Notice Board.

The above options are useful for testing.

If the `-s` option is used (incompatible with the `-u`, `-e` and especially the `-c` options), the initial conditions are set according to Section 2.1. The random number seed $r \neq 0$ is used to generate different initial conditions.

2.3 A Tale of Two Drivers

One (arguably) intuitive way of writing a driver program for a boat is to implement the algorithms in terms of ‘procedures’ extending over several time-steps, much like a navy boat driven by people would have. In the case of the UBoat driver, these might be:

- ‘head for the destination’ (providing the channel and path ahead is clear)
- ‘avoid collision with the walls’ (when the walls are close)
- ‘avoid an approaching EBoat’ (when the UBoat has a sonar contact, but the EBoat does not)
- ‘evade the charges of an attacking EBoat’

The basic components of all such procedures could be to turn the boat by a set angle’ and to drive the boat forward by a set distance’.

However, once such a procedure is entered, there will be circumstances in which it should be aborted, as is indicated above. An *alarm* can be given to such a procedure, which can be used to determine when it should be aborted (e.g. when unforeseen circumstances arise); then a more appropriate procedure for the current circumstances is chosen.

Implementing such procedures using Eiffel methods is the natural way to model this. For each method, an appropriate *alarm* object is tailored and may be passed down to any sub-procedures. The current *state* of the procedure is then simply represented by where in the method execution is currently occurring, and transitions between states are achieved by normal Eiffel control structures.

This is the recommended implementation strategy for the UBoat driver. However, to achieve the necessary UBoat-Ocean-EBoat interactions which occur on every time-step, it turns out to be impossible to do this as well for the EBoat driver if only one thread of execution for the whole simulation is permitted (which is a constraint imposed upon us by our implementation of SmallEiffel). Thus, the EBoat driver *EDRIVER* (and for that matter, the Ocean) has to be implemented in a different fashion: it is coded to manipulate the EBoat object one step at a time, with *state variables* explicitly representing the procedure (and sub-parts of the procedure) currently active, with explicit code to determine state transitions by manipulating these variables.

2.4 Simulation Security Mechanisms

As computational objects are not subject to any physical laws (as are the real-world objects they are intended to represent), it is important that they are not ‘corrupted’, either accidentally or otherwise. An example of this is a UBoat object moving more than 100m per time step, which implies a speed which no known U-boat is capable of! In particular, the driver classes are not to be trusted!

The information hiding afforded by the interfaces to the *BOAT* classes, together with their **invariant** checks (which, together with any internal features of the classes, may be changed or extended without notice), is thought to be sufficient for most of these purposes. The *OCEAN* class (which is trusted) also checks that UBoat object was initialized as requested by its *make* method (it creates the EBoat object directly), and the boats notion of time is kept synchronised with its own.

However, this will be put to the test in this assignment!

3 Setting Up

Copy the files from `/dept/dcs/comp1120/public/assign3:`

```
loadpath.se Makefile test_drive.e boat_drive.e boat_alarm.e udriver.e
```

into your own sub-directory. The latter 3 files are templates to which you will complete for this assignment.

Note that the file `loadpath.se` results in the Eiffel compiler finding different versions of the following classes from Assignment 2:

- *TWOVEC*: added *add_v* feature, convenient for performing vector calculations within expressions.
- *SONAR*: is extended to have a *detect_explosion* feature.
- *EBOAT*: can have a *Void OCEAN* attribute, in which case does not attempt to call its methods (this is similarly done in the *UBOAT* class).

A void attribute is necessary for the EBoat used by its driver when co-routines are not available. It also permits a ‘dummy’ EBoat to be created, so that EBoat parameters can be accessed.

- *OCEAN* now assumes that the boat calling its method is a UBoat (for purposes of the logging events and the display window).

4 Part 1: A Generic Boat Drive Library

The class *BOAT_DRIVE* provides a library for basic procedures to drive a *BOAT* object. It turns out that a satisfactory implementation of these turns out to be less trivial than one might first think. Rather than by extending a *BOAT* class, this can be done by simply applying operations on the (reference) objects which are parameters to the methods in the library.

The provided file `boat_drive.e` is a template for the library; your first task is to give an implementation to the *course*, *turn* and *move* methods. Their required behaviour is specified informally through the comments under the heading of each procedure; note that the **ensure** clauses do *not* fully specify their required behaviour; however, they do provide some useful checking. You should not change these clauses or any other features of this class affecting its external interface (however, you should add **check** clauses to check for any extra post-conditions, or intermediate assertions, that you think appropriate).

course is fairly simple: it requires setting the boat’s controls appropriately and invoking the boat’s *step_time* method a sufficient number of times.

turn is in principle similar; a helper function *diff_angle* is provided which can be used to determine the amount which the boat has turned. However, it has some more stringent requirements: due to the discreteness of the simulation, the boat object can ‘over-shoot’ its turn by up to *Turn_max* degrees if the rudder setting is kept constant. While a similar phenomenon occurs for *course*, this is not acceptable for *turn*, as in time a small deviation in direction can result in a large deviation in the boat’s position. A solution is to use a reduced rudder setting for the last or first time-step; the supplied method *adjusted_rudder* can calculate such a setting.

The final procedure *move* can be achieved by a call to *turn* and *course*.

The provided file `boat_alarm.e` implements a simple implementation of a boat alarm. It only has two options: an alarm which never gets raised, and one that gets raised when the boat object is getting too close to the channel walls, as defined by its *ocean* attribute.

The provided test program `test_drive.e` should be used to test the *BOAT_DRIVE* library: it uses command line parameters (here the `-u` and possibly the `-c` options, with $opt_e = 0$, are appropriate, see Section 2.2), followed by a series of commands from the standard input. It can be compiled by the command `make test_drive`.

5 Part 2: Systematic Testing of the library

Provide a set of test cases that are sufficient to perform a testing of the procedures that you have added to *BOAT_DRIVE*. Note that it is *not* necessary in this case to test that **require** clauses correctly identify invalid inputs, as these have already been provided to you.

The test cases should include both the input (command line parameters followed by a (short) sequence of commands from standard input) and the expected result. In each sequence of commands, there should be a good reason if it includes more than invocation to a *BOAT_DRIVE* routine; rather, you should chose your initial conditions to immediately start with the situation of interest.

The test cases should also include a description of the expected output.

Place these in the text file `report.txt`, under the heading **Part 2**. Include a justification that these are sufficient to perform an adequate *unit* and *structural* testing of the procedures that you have added to *BOAT_DRIVE*.

6 Part 3: The UBoat Driver

From the template file `udriver.e`, which defines the root class *UDRIVER*, implement the following broad strategies:

- **UDuck**: drive to boat towards its destination, with a procedure to avoid the walls when necessary.
- **UAvoid**: attempt to avoid detection from an approaching EBoat (still at a range $> EBOAT.Range_{max}$) by changing course appropriately.

Note that EBoat's projected distance of closest approach when it is first detected, plus its relative speed, are relevant parameters for this situation; it may not always be possible to avoid detection, even assuming the EBoat continues in a straight line.

- **UEvade**: attempt to evade destruction when the EBoat is close enough to threaten the UBoat. Probably continuing in straight line at the same speed will not be an effective tactic in this situation!

Program termination can occur if the UBoat reaches its destination or if it is destroyed.

The suggested implementation strategy is to use the *BOAT_DRIVE* class and an extended *BOAT_ALARM* class (it can be extended any way provided its existing external interface remains the same). For the latter, alarms such as gaining/losing a contact, and the EBoat reaching a critical distance, might be required. However, you can implement these (or parts of these) in another other fashion, and you will be asked to justify your choice.

You should break down the main tasks into suitably designed and sized methods. Include in your methods any sensible **require**, **ensure** and **check** assertions (preferably, if and where possible, of the fashion used in the *BOAT_DRIVE* library).

Include, under a header of **Part 3.1**, in `report.txt` a justification of the implementation strategy that you have chosen.

Include also a description of the underlying *tactics* you have used for the above three broad strategies, including why you think they might be effective.

6.1 An Alternative: Breaking Object Security

An alternative broad strategy to **UAvoid** and **UEvade** is **UCheat**: if possible, get the UBoat to achieve its objective by corrupting the given UBoat object from within Eiffel. For example, this

might be done by making an instantaneous change of position, by altering *Speed_max*, or even by resetting *is_destroyed*. Corruption of the ocean object may also be possible.

However, *UDRIVER* must use an object of the un-altered *UBOAT* class, and not any of its sub-classes. To be fully equivalent to both *UAVoid* and *UEvade*, any mechanisms used in *UCheat* should not be affected by internal changes to the *UBOAT* and *OCEAN* classes, including increased checking of the integrity of the data structures and objects.

6.2 Testing

Include, under a header of **Part 3.2**, in `report.txt` a set of test cases to test the above three directives. Note the $\text{opt}_e = 0$ is appropriate to test *UDuck*, and $\text{opt}_e = 1$ is useful to test *UAVoid*. The cases should comprise of a brief description, the command line parameters to the `udriver` program (do *not* use simulation mode!), and a description of the expected and actual results.

6.3 Hints

For avoidance and evasion, *tracking* the EBoat may be necessary to make appropriate evasive action. In this case, tracking means to use 2 different sonar readings of the EBoat to not only calculate its present position, but also its present velocity.

In any case, use simple tactics; these are more likely to be successful (and far less time consuming!) than complex ones. No strategy is ‘perfect’ (in that it can guarantee the UBoat remains undetected / un-destroyed), and perfection is *not* required. Any reasonably effective tactic with a sensible motivation will be good enough.

7 Part 4: Further Reflections

Add under a heading of **Part 4** in your report answers to the following questions:

1. how useful were the **ensure** clauses provided in the *BOAT_DRIVE* in debugging its methods? Could such assertions provide an aid to or even obviate the systematic testing of Part 2?
2. the class *EDRIVER* has been implemented as a front-end to a C library which actually does all the work. Comment on the possible advantages and disadvantages of doing so.
3. based on your experience with this system, comment on how effective Eiffel is for providing security against accidental or deliberate object corruption.
4. give a brief critique on the overall software architecture of the system, and in particular on idea of providing an ‘ocean’ object attribute to one of the boats objects to effect an interaction between a boat on its environment. Also comment on the usefulness of having a library like *BOAT_DRIVE* and (a suitably extended) *BOAT_ALARM*. Are there alternate (arguably better) ways of going about this?
5. give any other pertinent thoughts you have on the simulation system.

8 Submission and other details

The assignment will be marked out of 30, with the following break-up of marks:

`boat_drive.e` and `boat_alarm.e`: 5, `udriver.e`: 12, `report.txt` (testing and answers to questions): 13

Note that good programming style as discussed in lectures, and readable report (including good grammar, layout and spelling) are important. See the web page for COMP1800 for more details on this; you are expected to use the same conventions.

It is due by 17:00 on Friday Jun 6. Submission is via the command:

```
mark comp1120 assign3 boat_drive.e alarm.e udriver.e report.txt
```

We estimate the assignment should take something like 20 to 30 hours per person; it may take much more if you want to put in more effort than is essential. How much time you wish /can afford to spend on this assignment is left to you.

Appendix: customizing the Display Window

You may find it useful to vary the animation speed and size of the window. This can be done via setting Unix environment variables in the shell that you invoke the program from. Details of how to do this will be posted on the course Notice Board when this becomes available.