Faculty of Science, Engineering and Technology

Object Oriented Programming

Topic 7: Introducing C++

Resources

The following resources can help you with this topic:

- www.cplusplus.com Tutorials and Forums
  - Classes (I)
  - Default constructor and Destructor (stop at copy constructor)
  - Inheritance (skip friendship)
  - Polymorphism
- Tutorials Point C++ Programming Tutorials

Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the deadline (check Doubtfire for the submission deadline).

Pass Task 15 - Planetary Rover UML Class Diagram	2
Pass Task 16 - Planetary Rover Code	3
Pass Task 17 - Case Study: Iteration 3	7

Remember to submit your progress, even if you haven't finished everything.

After you have **discussed** your work with your tutor and **corrected any issues**, it will be signed off as complete.

Pass Task 15 - Planetary Rover UML Class Diagram

Before starting this exercise, install the C++ IDE, Compile Tools and Unit Testing Library as described in the ***OOP C++ Setup*** instructions (available on Blackboard).

For this exercise, you will develop code to model a simple robotic Rover that has a number of types of Device that it can deploy while exploring distant planets.

Use the following description to **draw** a **UML class diagram** of the Planetary Rover model:

★ A Rover has one or more Batteries, which it uses to power its Devices.

★ A Battery has an integer number of units of power (charge) that must be greater than zero.

★ A Rover has a number of Devices, which must be connected to a Battery to operate.

★ Devices can be connected to and disconnected from a Battery.

★ Rover can attach or detach a Device; when a new device is attached, the Rover connects it to the Battery with the greatest charge.

★ Rover can operate; when operated, it simply tells each of its attached devices to operate.

★ There are three different kinds of Device: Radar, Solar Panel, and Drill. When operated, each outputs a unique message and each has a different effect on the Battery to which it is connected

- Radar - when operated, the radar subtracts 4 units of power from the Battery to which it is connected. If successful, it outputs the message “PING" and randomly prints the message “PONG” (e.g. in 1 out of 4 times).
- Solar Panel - when operated, the solar panel adds 1 unit of power to the Battery to which it is connected. If successful, it outputs the message “Charging”.
- Drill - when operated, the drill first checks to see if its safety switch is enabled. If the safety switch is enabled, it prints the message “Safety First” and does nothing more. If the safety switch is not enabled, the drill subtracts 10 units of power from the Battery to which it is connected. If successful, it outputs the message “VRMM! VRMM!”.

Hint: Add a bool field _safetyEnabled to the Drill class. The constructor should initialise it to false. You can then check the behaviour of this switch when writing a test driver.

Pass Task 15 - Assessment Criteria

Make sure that your task has the following in your submission:

- UML class diagram that shows the Rover class, Battery class, Device class and its derived classes and all relationships. A photo of a sketch is fine, as long as it is readable.
- The Device class must be abstract.

Pass Task 16 - Planetary Rover Code

Create a Rover Tests file and add unit tests for attaching, detaching, and operating devices:

1. Edit **TestRover.h** and replace the two example tests

```
CPPUNIT_TEST(testMethod);
CPPUNIT_TEST(testFailedMethod);

[…]

private:
  void testMethod();
  void testFailedMethod();
```

with the declaration of a single unit test named **testAttach**.

2. Likewise, edit **TestRover.cpp** and replace the two example tests

```
void TestRover::testMethod() {
    CPPUNIT_ASSERT(true);
}

void TestRover::testFailedMethod() {
    CPPUNIT_ASSERT(false);
}
```

with a single **testAttach** method definition

```
void TestRover::testAttach()
{
    Rover* rover = new Rover();
    rover->attachDevice( new Radar() );

    CPPUNIT_ASSERT( rover->deviceCount() == 1 );
    delete rover;
}
```

This is the start of a test class that creates a *Rover* object and tests its *attachDevice* method.

3. Try to build the program; you will get a list of compilation errors related to there being no **Rover** or **Radar** classes. Define the Rover and Radar classes; for each class, use NetBeans to create a new C++ class

- select **File** -> **New File**.
- At **Step 1. Choose File Type**
- under **Categories** select **C++**
- under **File Types** select **C++ Class**
- click **Next >**
- In the **New C++ Class** pop-up window, specify the class name (Rover or Radar) and click **Finish**

4. Note that NetBeans creates both Rover.h (and Radar.h) in the **Header Files** and Rover.cpp (and Radar.cpp) in the **Source Files**.

5. Edit **TestRover.cpp** and, just under the

```
#include "TestRover.h"
```

add `#include` statements to include the two new header files

```
#include "Rover.h"
#include "Radar.h"
```

6. Edit **Rover.h** and **Rover.cpp** and add empty stubs for the **attachDevice** and **deviceCount** methods; e.g. in Rover.h add the *method declaration*

```
int deviceCount () const;
```

(in the public section) and in Rover.cpp add the *method definition*

```
int Rover::deviceCount () const
{
    return 0;
}
```

Note the use of the `const` keyword in the above declaration and definition.

This tells other developers (and the compiler) that this method does not modify the object that was used to call it. It is a promise that is enforced by the compiler. If any line of code inside the method definition attempted to modify an attribute/field of the class, the compiler would abort with an error message.

Note: Something is intentionally missing in the instructions!

What is the type of argument to attachDevice? Do you need to add another new class?

What are the relationships between this class and the Rover and Device classes?

7. After implementing the missing class, **run** the test again to see it compile and fail; e.g.

```
Test name: TestRover::testAttach
assertion failed
- Expression: rover->deviceCount() == 1

Failures !!!
```

8. To work around this failed test, change the deviceCount method to

```
int Rover::deviceCount () const
{
    return 1;
}
```

   then re-run the test and see it Pass. This indicates that one unit test is not sufficient.

9. Return to the **TestRover** code and add an additional unit test that attaches two devices and ensure that it fails when it asserts that deviceCount == 2.

10. Return to the Rover class and implement the *attachDevice* and *deviceCount* methods.
    - Use the Standard Template Library vector class to add a field called **_devices**
    - In *attachDevice* add the passed pointer to the collection using *push_back*.
    - In *deviceCount* return the size of the collection.

Hint: You will need a vector of pointers, such as

```
vector<Spell*> _spells;
```

(but of course not a collection of spells).

11. Run the unit tests again, and verify that both versions of **testAttach** now pass.

12. Use test-driven development to write unit tests and implement the remaining classes and missing functionality, including
    1. Battery, Solar Panel, and Drill classes; and
    2. Rover::operate and Device::operate methods
    3. Rover::connect and Device::connect (and disconnect) methods for connecting and disconnecting Battery objects

13. Decide how to deal with the error that should arise when a device tries to draw too much charge from a Battery; this occurs when the amount drawn by a Device would reduce the remaining charge to zero (or less). Should the Battery throw an exception or simply return an error code? How would you write a unit test to verify that an exception is thrown when it should be?

Hint: When implementing the random behaviour of the Radar class, use the Standard C Library `rand` and `srand` functions to generate random numbers, as described at http://www.cplusplus.com/reference/cstdlib/rand/

Pass Task 16 - Assessment Criteria

Make sure that your task has the following in your submission:

- The unit tests correctly check the features of all of the classes.
- The code must follow a self-consistent coding convention and be well documented
- The code must compile and the screenshot must show the tests passing.

Pass Task 17 - Case Study: Iteration 3

Over the next few week you will implement a larger object oriented program that demonstrates the use of all of the concepts covered so far. This will help you develop a deeper understanding, and create additional pieces of work to communicate this understanding.

1. For this week complete Iteration 3.

> **Note**: At this point there will not be a "program" as such, just a set of unit tests that help demonstrate that your solution is moving toward completion.

Once your tests are working correctly add additional details to your cover page. Remember to relate what you are doing to the unit's learning outcomes.

> **Tip**: Now would be a good point to reflect on anything you did in these iterations that was interesting or challenging. Elaborate on this in the cover page so that you can take advantage of this in your portfolio.

Pass Task 17 - Assessment Criteria

Make sure that your task has the following in your submission:

- Iterations 1 to 3 are implemented
- The new classes have XML documentation for all public features.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show the tests passing.