

G52OBJ Lab Exercises 2

Steven Bagley

1. Introduction

These exercises serve two purposes, primarily to increase your familiarity with the **Decorator** pattern but also to highlight the similarities and differences between C++ and Java. To this end, this exercise uses the coffee shop **decorators** from lecture seven*. I'm providing you with the full Java implementation for these classes that you can use as a reference. I'm also providing you with some of the classes implemented in C++, and section 2 of this document spends sometime going over the differences between the two implementations.

The main exercise is outlined in Section 3. You are to implement the remaining classes in C++ using the provided Java as a guide. The final section illustrates why the **Decorator** pattern uses a subclass of the main type that all decorators inherit from by showing how we can use this to write decorators that can tell if they are decorating another decorator and modify their behaviour appropriately. This is done by getting you to implement another decorator for coffee shop, this time enabling clients to specify double sized drinks.

If you feel, that you are already familiar with C++ then I suggest skipping the first section and moving onto the second section which is implementing the remaining **decorators**. Even though this is, to some extent, just translating the Java to C++ doing the exercises should help to consolidate your knowledge of the **Decorator** pattern.

These set of exercises assume that the reader is familiar with the material covered in the first set of exercises. We shall not be covering how to compile the C++ or Java files under UNIX Note, we shall be assuming that you are familiar with compiling C++ and Java files under the UNIX systems. See the last set of exercises for examples of how to compile the C++ files, but remember you'll need to compile each .cpp file and then link them into an executable.

2. Converting the Java to C++

If you grab and extract `lab2.tar.gz` from the website (see the previous exercises for details of how to extract .tar.gz files if you are unsure), you should find that you'll have both C++ and Java versions of the classes implementing the **decorator** pattern for the coffee shop, in their respective directories. However, only some of the coffee and **decorator** classes specified in the Java have been implemented in C++. You will implement the remaining classes later.

Note for this section of the document we assume that you have both the Java and C++ files being discussed open in front of you.

2.1. The Beverage Class

The root of the object hierarchy in our coffee shop setup is the `CBeverage` class. The first difference to note between the C++ and Java implementation is that the C++ version is contained within two files (`Beverage.cpp` and `Beverage.h`) while the Java implementation is within a single file (`CBeverage.java`).

* Actually, both these exercises and the lecture were based on material from the O'Reilly *Head First Design Patterns* book and the **Decorator** chapter is available online at <http://www.oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf> if you want further reference material.

2.1.1. Inside Beverage.h

As we saw in the previous exercises, `.cpp` files contain the implementation of a class while `.h` files contain the declaration of the class. Therefore, the declaration for the `CBeverage` class can be found in the file `Beverage.h`. The first difference we notice is the following construct:

```
#ifndef __BEVERAGE_H__
#define __BEVERAGE_H__

...

#endif
```

in the header file to stop the header file being included multiple times and generating ‘class already defined’ errors at compile time (again, see previous exercises for more discussion of this). One thing to notice is that C++ needs you to put a semicolon after the closing brace of the class declaration if you forget this, you’ll generate sorts of unhelpful errors usually from completely unrelated parts of your program.

You should notice that the two languages are more similar than they are different. C++ doesn’t need classes declaring as `public` or `abstract` so these keywords are missing from the file. The major difference (other than the code being moved into a separate file) is that within the class definition the effect of the `public`, `private`, and `protected` access specifiers persists beyond the current keyword (as well as being suffixed by a colon). So for example, all of the methods between the `public:` and `protected:` keywords are publicly accessible methods.

Also note that the C++ version uses the `virtual` keyword to specify that a method might be overridden by a base-class. If you don’t specify this then C++ will always call the method for the class type rather than any subclasses that might be masquerading as the base-class. If you want to see this in effect, when you’ve finished the exercises, remove the `virtual` from the declaration of `getDescription()` in the header file and recompile. You’ll find that none of the decorators work properly. This is done because it makes methods that aren’t going to be replaced run slightly faster. The downside is that we have to remember to specify that a method is `virtual` if we are going to override it in a sub-class.

Other than that the declaration of the methods is more or less identical in both languages. `string` has a lower-case `s` in C++ compared to Java. Note that the abstract method `Cost()` (which returns the cost of the beverage) is specified using the `abstract` keyword in Java like this:

```
public abstract double cost();
```

and in C++ like this:

```
virtual double Cost() = 0;
```

Both tell the compiler that there is no implementation for this method provided in this class and hence that the class cannot be instantiated (although its subclasses can, if they implement the method).

Note that we are also specifying a destructor for the C++ class. This is so that when we come to implement the **decorators** we can ensure that the objects they wrap are deleted when the decorator is deleted. Since the destructor is going to be overridden by the sub-classes we need to declare it `virtual` so that the correct implementation is called at run-time. More on this when we discuss the implementation of **decorators**.

2.1.2. Inside Beverage.cpp

Moving onto consider implementation of the methods takes us into `Beverage.cpp`. Here we see the following at the top of the file:

```
#include <iostream>
#include <string>

using namespace std;
```

The first two lines include the header files for the default C++ libraries, this is similar to using the `import` keyword at the top of a `.java` file. The

```
using namespace std;
```

tells the compiler that we want to use objects and classes in the namespace `std`. This stops us having to prefix types such as `string` with `std::`. Lines like those above will be in pretty much every C++ you write. We then import any header files for any classes we use (in this case, we only include `Beverage.h` since we are implementing that class).

The rest of `Beverage.cpp` contains the implementation of `CBeverage` class's methods. The only difference between this C++ implementation and the implementation in Java and C++ implementation of the methods is that in C++ you need to prefix the method name with the name of the class using the `::` operator. This is because (self-evidently) the methods are no longer defined within the class declaration and so the compiler does not know which class they belong to unless we tell it. As an example, the constructor for defined: `CBeverage` is

```
CBeverage::CBeverage()  
{  
    m_description = "Unknown Beverage";  
}
```

The observant will notice that I've switched from using the old style C-strings (using `char *`) to using the C++ `string` objects. This should make the code more familiar to the Java you are used to.

2.2. Implementing Concrete Coffee

With the base-class implemented, we can turn our attention to one of the concrete coffee classes in this case, `CEspresso` although if you look at the provided implementation of `CDarkRoast` you'll see that it is virtually identical.

Comparing `CEspresso.java` with `Espresso.h` and `Espresso.cpp`, you should find that again there are very little differences between the two versions other than those we have already noted. The major new difference is how C++ specifies inheritance:

```
#include "Beverage.h"  
  
class CEspresso : public CBeverage
```

Firstly, while the Java compiler will go off and find the `.java` file for the base-class automatically, in C++ we need to specify it ourselves with an `#include`. This means that code that uses `CEspresso` objects exclusively without using the base-class does not have to manually include `Beverage.h`.

The next line tells the compiler that we are defining the class `CEspresso` and that it inherits from the class `CBeverage`. We use `: public` rather than `extends` as we would in Java to denote inheritance in C++. The reason for the difference is that C++ lets us change the access to methods and member variables inherited from the base-class. For the purposes of this course, we will always be using `public` here so that the access specifiers don't change.

If you wish, you can compare the implementations of `CDarkRoast`, however you will find that the similarities are identical.

2.3. Getting the Decorators in

Let's turn our attention to comparing the implementation of the **Decorators** themselves. These all inherit from the abstract class `CCondimentDecorator` which is a sub-class of `CBeverage`. This means that client code can treat the **decorators** as if they were normal `CBeverage` sub-classes.

Looking at the C++ implementation, we notice that there is no `.cpp` file, only a `CondimentDecorator.h`! This is not because I forgot to include it in the archive, but because there is no need for one. `CCondimentDecorator` is an abstract class (i.e. a class where all its methods are abstract) and so none of its methods are implemented. Hence, we do not need a `.cpp` file to contain the

implementation. Comparing the C++ and Java implementations, then you should see the same differences in implementation that we have already seen: the need to include header files explicitly, and the different operators for expressing inheritance and abstract methods.

2.3.1. Implementing the Mocha Decorator

Let's turn our attention to the `CMocha` decorator which implements a decorator for mochas. It's job is to add ' , Mocha' to the description and to add »0.20 to the cost. As you would expect by now, `CMocha` comes in two files with the declaration of the class in `Mocha.h` and the implementation in `Mocha.cpp`.

Most of the differences we have already seen. However, three main differences stand out that we've not yet come across. Firstly, variables containing references to objects are typed as `CBeverage` in Java, but `CBeverage *` in C++. This is because C++ is making it explicit to you that objects are represented as just a block of memory (as we saw in lecture 12).

The second main difference is that in C++ we call methods using `->` rather than the `.` operator that we use in Java. Again, this is because C++ is making the implementation of objects explicit.

The final difference is that we specify a destructor. This just deletes the pointer to the wrapped object that we store in a private member variable in the constructor. This ensures that the decorated object is deleted (and the memory it consumes made available for reuse) when the decorator is deleted.

2.4. Testing the objects

`starbuzz.cpp` and `StarBuzz.java` contain the implementation of code that construct a few beverages to test that our implementation is working properly (although you'll notice that the C++ implementation is cut-down and only tests the classes provided). Again, we have already seen most of the major differences between the two languages in this file. The only new difference we see is to do with how we print things. In Java, we use the `System.out.println` method to output strings, but in C++ we are using the rather more baroque[†]:

```
cout << beverage->GetDescription()  
    << " »"  
    << beverage->Cost()  
    << endl;
```

Basically, C++ reuses the `<<` operator to allow you to output data to a stream (in this case, the standard output represented by the global variable `cout`). We can concatenate multiple uses of `<<` to output multiple things as shown here as a shorthand for the more explicit:

```
cout << beverage->GetDescription()  
cout << " »"  
cout << beverage->Cost()  
cout << endl;
```

In the code quoted above, we are outputting the string returned from the `GetDescription()` method on the object pointed to by the variable `beverage`; a pound sign; the cost of the drink as returned by the `Cost()` method and finally a newline. C++ defines a global variable, `endl`, with the string necessary to generate a newline (it varied between operating systems).

Note that in the C++ file we also use the `delete` operator to delete the objects once we have finished with them.

3. Implementing the rest of the Coffee Shop

The C++ directory only contains implementations for the `CEspresso`, `CDarkRoast` and `CMocha` classes (as well as the relevant base-classes). For the next set of exercises, it is your task to implement the remaining classes and also to update `starbuzz.cpp` to test them out. You can use the Java

[†] See the section entitled *The March of Progress* at <http://www.horstmann.com/> for a humorous commentary on the 'development' of printing routines over the last thirty years...

implementations provided as reference. The classes you need to implement are:

Coffees
CDecaf
CHouseBlend

Decorators
CMilk
CSoy
CWhip

Some hints on what you will need to do:

- Create `.cpp` and `.h` files for each of the new classes, providing both a declaration for the class (in the `.h` file) and an implementation of the methods (in the `.cpp` file). I suggest you start by making a copy of the files for one provided classes (e.g. `CDarkRoast` or `CEspresso` for coffee classes or `CMocha` for decorators).
- Remember to add or modify the `#ifndef` etc. lines in the header file otherwise the file won't be including. The identifier after the directive doesn't matter, it just needs to be unique for each class.
- Add `#includes` to `starbuzz.cpp` to include your new classes' header files.
- Add code to `starbuzz.cpp` to create new and exciting coffee combinations. Again, you can use the existing code as a reference for how to implement this.
- Copy the code to print out the description and price of the newly created and drinks. This is literally a cut'n'paste job of the code already present in `starbuzz.cpp`, but remember to change the variable names where necessary. See the Java implementation (`StarBuzz.java`) for some examples you could use.

4. The Double size Decorator

In this part of the exercise, you are going to implement a new decorator that allows us to create double sized drinks. A double is charged at 1.75 times the price of a normal drink and the description is preceded by the word 'Double' e.g. 'Double Espresso'. With this implementation and your experience gained converting the other decorators from Java to C++, you should be able to implement the double decorator quite easily. The class should be called `CDoubleBeverage`.

Implement the code required for `CDoubleBeverage` and add code to create a `starbuzz.cpp` to create a both a (Double Espresso) with Milk, and Double (Espresso with Milk). Note I've used brackets to signify where the new decorator should be implemented in the chain of objects.

When you run your code, you'll find that the implementation prints identical output for both cases, which is not very clear. Let's modify this behaviour so that if the double decorator is wrapping another Decorator then it wraps the result of calling the wrapped object's `GetDescription` in brackets before prefixing it with 'Double'.

The code to implement this is very simple because we have made all decorators inherit from the abstract class `CCondimentDecorator`. Therefore, we can test to see if the decorated object is an instance of `CCondimentDecorator` and if it is then we can wrap the string in brackets. In Java, we could use:

```
if(m_beverage instanceof CCondimentDecorator)
{
/* ... */
}
```

In C++, we can get the same effect by attempting to convert the pointer to a `CBeverage` to a pointer to a `CCondimentDecorator` using the `dynamic_cast` operator. If this is unsuccessful (i.e. the object is not a sub-class of `CCondimentDecorator`) then it will return a pointer to `NULL` which we can test for like this:

```
if(dynamic_cast<CCondimentDecorator*>(m_beverage) != NULL)
{
    /* Insert code to handle case of a concrete coffee beverage */
}
else
{
    /* Insert code to handle case of a decorated coffee */
}
```

If you add that to your implementation of `CDoubleBeverage` and insert the necessary lines of code to generate the desired effect then you should find your code now correctly brackets the output if necessary.

Don't worry about needing to remember any of the type testing code for the coursework, you won't need to use it. The reason I mention it here is to illustrate why we implement an apparently unnecessary class to act as a base-class for the Decorators. As it stands none of the current Decorators would work any differently if we inherited them directly from `CBeverage`. However, by instead inheriting from a base-class (in this case, `CCondimentDecorator`) it is possible for us to perform tests to see whether we are wrapping a plain or decorated object and alter our actions appropriately.