CS 246 Fall 2021 — Tutorial 8

November 3, 2021

Summary

1 `virtual` and `override`

- When working with inherited classes, we will often want to define the methods so that it will work differently for separate subclasses:

```
// Full example at animals/animals.cc
struct Animal {
    virtual bool fly() const {
        return false;
    }
};

struct Bird : public Animal {
    bool fly() const override {
        return true;
    }
};

struct Goose : public Bird {
    bool fly() const override {
        cout << "THANK MR. GOOSE" << endl;
        return true;
    }
};
```

- Note that we have declared `fly()` as a `virtual` method.
- **Note:** the virtual method will only be called when dealing with objects *through pointers/references*. This does not work when using objects directly.

  - For example, what are these lines of code actually doing?

```
Animal a = Bird{};
a.fly();
```

- Using the keyword `override` tells the compiler to check that the method is actually overriding a `virtual` method in a superclass. This causes a compiler error if it can't find a `virtual` function of the same signature.

2 Pure Virtual Methods

- *Pure Virtual methods* are methods that subclasses will need to provide an implementation for if they want to be instantiable.
- Most of the times, pure virtual methods will not have an implementation.
  - Pure virtual methods can have an implementation.
- We declare a method as *pure virtual* when we add `virtual` to the front and `= 0` to the end of its declaration in the class definition. For example:

```
class A {
    virtual void someFunction() = 0;
};
```

- Typically, pure virtual methods are used if it does not make sense for a method to have an implementation in the base class, or if we want to make the class *abstract*.

3 Abstract and Concrete Classes

- A class is *abstract* if it has one or more pure virtual methods. A class is *concrete* if it has no pure virtual methods.
- This means that all classes must be either abstract or concrete, but not both.
- Abstract classes are not *instantiable*. This means that you cannot create objects of abstract classes, i.e. you can only create objects of concrete classes.

```
class A {
    int a;

public:
    explicit A(int a) : a{a} {}
    virtual void foo() = 0;
};
```

```
class B {
    int b;

public:
    explicit B(int b) : b{b} {}

    void bar() {
        cout << "This is not pure virtual" << endl;
    }
};

int main() {
    A a{1};
    B b{2};
    b.bar();
}
```

- The purpose of an abstract class is to allow subclasses to inherit from a base class containing information that is common to all subclasses, but it doesn't make sense to have an instance of the base class.
- A subclass of an abstract class is also abstract unless it implemented *all* pure virtual methods of the base class, in which case it becomes concrete.

```
class C : public A {
    int c;

public:
    C(int a, int c) : A{a}, c{c} {}
    void foo() override {
        cout << "Overriding foo" << endl;
    }
};

int main() {
    // A a{1};
    B b{5};
    b.bar();
    C c{1, 2};
    c.foo();
}
```

4 Destructors Revisited

- Now with inheritance and (pure) virtual methods, we need to revisit the destructor.

1. If your want the class to be inherited, the destructor should always be `virtual`. Why?
2. They must always have an implementation; even if they are pure virtual. Why?

```
class B {
  public:
    virtual ~B() = 0;
    virtual string hello() = 0;
};

class A : public B {
    A *arr;
  public:
    A() : arr{new A[5]} {}
    ~A() { delete[] arr; }

    string hello() override {
        return "Bonjour!";
    }
};

B::~B() {}
```

- Class A inherits from the abstract class B which has a pure virtual destructor.

Let's take a look at another example of the *Decorator design pattern*, since it relies heavily upon abstract base classes, polymorphism, and virtual methods to work properly.

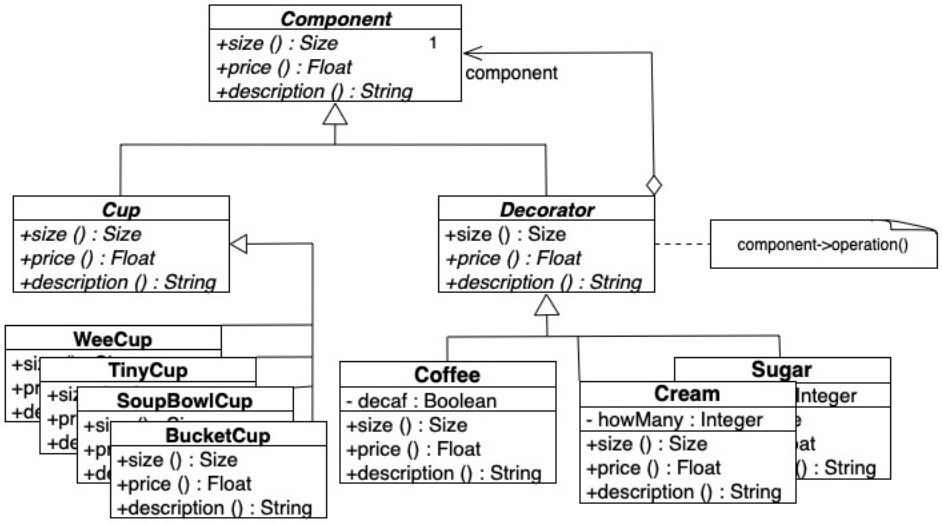
5 Decorator

- Lets us add (and remove) features/functionality at *run-time*.
- Often used when creating a subclass for every possible combination is impractical.
- Structurally, we have a linked-list of decorations connected to a final, concrete object.
- Work is done through *delegation*, passing information up and down the linked-list, by calling the methods on the linked-together objects.

5.1 Example

Let's consider another example of something where the Decorator design pattern could fit. Consider a cup of coffee. We could have multiple sizes, types of coffee, and additions. We'll start with the

following features:



Question: Why isn't the `size()` method in the `Decorator` class a pure abstract method?