CS 247 — Software Engineering Principles

Kevin James

Spring 2014

---

1 Object-Orientation

Code which follows **object-oriented** patterns has better affinity between user-defined types and real-world types, can be more easily reused (inheritance, composition, as-is, polymorphism), allows for encapsulation and information hiding, decomposes extremely effectively (and thus seperates concerns), and allows for abstraction through ADTs (Abstract data types), interfaces, etc.

1.1 ADTs

An ADT is a user-defined data type. They are composed with a range of legal values and functions that maniplate variables of a given type. By providing compiler support for these type restrictions, we turn programmer errors into type errors (which are checked by the compiler).

One of the main motivations for designing an ADT is to ensure safety of any client code. Other motivations include evolvability and scalability. ADTs also tend to improve code efficiency by limiting range checks to constructors and mutators.

An ADT constructor initializes the new object to some legal value and throws an error if the passed-in value is illegal. Accessors and mutators provide restricted read/write access to one of the values in the object. It is best practice to ensure legality within each mutator and use `const` references whenever possible.

We use the **outside-in** method fo development: first we determine what the user wants out of it, then we design and implement it.

Function Overloading is syntactic sugar which allows us to define a function with the same name as some other function, so long as this new function has different parameters. It is generally best practice to only overload functions when the purpose of the function is the same for each. For example: `void print(int)` and `void printfloat`.

Another option is to use **default arguments**. Defulat arguments tend to be used when a given argument should be optional. For example: `void print(int, outputStream=cout)`. Default arguments must appear only in the function declaration.

We can also overload operators. For example, we could define the sum of two classes as some other class, some modified version of one class, or even a standard data type of some sort with `MyClass operator+(const MyClass&, const MyClass&)`. Though widely used, this practice should be used sparingly, if at all.

Nonmember functions are critical ADT functions which are declared outside of the class. This leads to ebtter encapsulation and more flexible packaging. Additonally, certain functions are required to be nonmember functions (e.g. `operator>>`). Streaming operators should be nonmember functions so that they can accept a stream as a first operand and thus chain stream operations.

A class can have `private`, `protected`, and `public` data members and functions (in addition to several less-often used flags). It is best practice to use the most secure flag as possible (generally: `private`); though the `public` flag is sometimes necessary, the `protected` flag should be avoided. We can also use the `friend` flag to create a `private` method which is accessible to a given other class.

1.2 Inheritance

Inheritance allows us to crreate classes (**derived classes**) which include the data members and functions of a **base class**.

1.3 Polymorphism

When we create a derived class, that class can inherit pointers from its parent.

Object slicing occurs when only base class fields are copied from a subclass. It can occur as a side-effect of intereactions between a subclass and a superclass: passing the sublcass by value, assigning the subclass to the superclass, or suing the superclass assignment operator between subclass instances.

Function binding occurs at compile time. When a base class pointer is given a reference to a subclass after its declaration, the subclasses methods will not be called.

Virtual functions are a method of getting around this. A virtual function delays function lookup to runtime, and thus will ensure the subclass function will be called even when using a pointer to the base class.

When having an instance of a base class doesn't make sense, we can make a **pure virtual function**. Pure virtual functions need no definitition and prevent class instatiation. They also ensure all subclasses must have a definition for that function.

1.4 Overloading

Overloading occurs when two functions with the same name have a different set (amount or type) of parameters.

1.5 Overriding

Overriding occurs in subclasses, when a subclass function has the same signature as a superclass function. When this function is called from the derived class, the derived class' version of the function will be called.

1.6 Friendship

The **friend** command allows us to declare a class or function which can access the private members of the class containing the friend declaration.

For example:

```
class Base {
    int priv;
public:
    friend void print(Base);
}

void print (Base esab) {
    cout << esab.priv << endl;
}
```

1.7 Helper Functions

The best practice is to hide helper functions as `private` methods or within a namespace. Helper functions modularize our code, but should not pollute the gloal namespace.

2 Entity vs Value Objects

Entity Objects are the digital embodiment of a real-world entity. Each object has a distinct identity and objects with the same attribute values are not equal. **Value Objects** simply represent a value of an ADT. Value objects with the same attribute values are considered to be identical.