CS 247 Midterm Review

CS 247

University of Waterloo

cs247@student.cs.uwaterloo.ca

June 25, 2014

Overview

Inheritance

Inheritance lets us reuse code with an "is a" relationship between objects. An object which publicly inherits from another gets all of the public and protected methods and variables of the base class as its own but does not have direct access to private methods and data members.

Inheritance in Constructors

How are base class parts of derived classes instantiated in a derived class constructor?

Inherited Methods and Overriding

If a base class and derived class define a function with the same name, when would base class methods be called? When are derived class methods called?

Inheritance and Static Methods/Variables

Are static methods and variables inherited?

Inheritance in Destructors

How are base class parts of derived classes destroyed in a derived class destructor?
Example : example.cpp

Exceptions

Exceptions allow us to seperate error handling code from normal code and prevent errors from being ignored. The best practice when creating exceptions is usually to create an exception class that inherits from std::exception and implements

```
virtual const char* what()
```

(you need to include $<$ *exception* $>$ to do this.)

Header Guards

Header guards prevent errors from the same things being defined mulitple times when files are included in more than one other file in a project.

No "Using"

Do not include "using" directives in header files. When you do so, any files that include the header will also have the effect of "using" which can lead to unexpected name clashes.

Make

Make is a tool for generating executables that allows us to compile large software projects easily and save time on compilation by compiling incrementally.

```
CXX = g++
CXXFLAGS = -Wall -MMD
OBJECTS =
DEPENDS = ${OBJECTS:.o=.d}
EXEC =

${EXEC} : ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

clean :
    rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}

-include ${DEPENDS}
```

Passing by Constant Reference

When programming in C++ we often pass objects to functions. When we pass by value the object's copy constructor is called, usually copying the entire object. This can be quite inefficient so we try to avoid it by passing a pointer which will only require 32 bits (or 64) to be copied. However, when we pass a pointer we have to use dereferencing syntax to actually get to the object so we instead pass a reference which allows us to use the object as though it were the actual object. Now that we are using a reference, if the object changed in the function the change will propagate back to the original object since it is a reference to the original object. To avoid this, as well as to allow passing literals, we make the reference constant.

Entity Based vs Value Based

When designing an ADT one of the first questions that must be asked is whether the ADT is entity based or value based.

Entity Based

Entity based ADTs usually:

- Prohibit assignment and the copy constructor
- Prohibit type conversion
- Avoid equality
- Are mutable

Value Based

Value based ADTs usually:

- Implement equality and other comparison operators
- Include a copy constructor and assignment operator
- Are immutable (instead of changing the value a new value is created)

Mutable vs Immutable

Whether to make an ADT mutable or immutable is often decided by whether it is entity based or value based. Mutable ADTs should include functions that allow them to be mutated while immutable ones should not.

Design Patterns

The design patterns covered were :

- Singleton
- Template Method
- Facade
- Adapter
- Strategy
- Observer
- Model-View-Controller

Singleton

What is the singleton pattern?

Singleton

The singleton pattern restricts the instantiation of a class to one object. We can enforce this by keeping the constructors private preventing the client from creating instances of the class without going through class methods.

Template Method

What is the template method pattern?

Template Method

The template method pattern promotes code reuse by providing a skeleton of code for an algorithm while deferring some steps to subclass methods.

Facade

What is the facade pattern?

Facade

A facade is an object that provides a simplified version of a larger interface of code. This is done to reduce dependency on the inner workings of a library and to make libraries easier to use.

Adapter

What is the adapter pattern?

Adapter

The adapter pattern is a way of allowing the interface of an existing class to be used by another interface. A class is defined that translates between the two incompatible interfaces.

Strategy

What is the strategy pattern?

Strategy

The strategy pattern is a way of making algorithms interchangeable at run-time. This is accomplished by providing an abstract base class as an interface for the algorithms and creating concrete derived classes that with different implementations of the algorithm.

Observer

What is the observer pattern?

Observer

One object, the subject, maintains a set of dependents, the observers. When the state of the subject changes it notifies the observers usually by calling one of their methods.

MVC

What is MVC?

Model-View-Controller

Technically not a design pattern but a software architectural pattern (broader scope than a design pattern). MVC is used when designing user interfaces, separating the application into three parts: model, view and controller. The model takes care of the logic of the application, updating the view when it changes. The controller takes user input and sends commands to the model. The view uses the observer pattern to interact with the model, recieving a notification each time the model is updated then requesting information from the model that it uses to generate a representation for the user.

UML Exercise

Try drawing a UML diagram for the strategy pattern.

Object Oriented Programming Principles

The principles covered in class were :

- Open Closed Principle
- Favour Composition over Inheritance
- Single Responsibility Principle
- Dependency Inversion Principle
- Liskov Substitutability Principle
- Law of Demeter

Open Closed Principle

What is the Open Closed Principle?

Open Closed Principle

A module should be open for extension but closed to modifcation. This means you provide an abstract base class which gives the interface for the client to interact with and do not modify it, then you still have the freedom to extend the functionality through concrete classes without breaking client code.

Favour Composition Over Inheritance

Why? When do we still use inheritance?

Favour Composition Over Inheritance

We choose composition over inheritance because it is possible to modify the component at run-time while we are able to use it in a similar way to inheritance by delegation of methods. We still need inheritance when we need a type hierarchy and it is still useful when using the entire interface of an existing class.

Single Responsibility Principle

What is the Single Responibility Principle?

Single Responsibility Principle

Each changeable design decision should be encapsulated in a separate module.

Dependency Inversion Principle

What is the Dependency Inversion Principle?

Dependency Inversion Principle

High level modules and low-level modules should depend on abstractions rather than concrete classes. Abstractions should not depend on details, details should depend on abstractions. Essentially, high level code should only use low level code indirectly.

Liskov Substitutability Principle

What is the Liskov Substitutability Principle?

Liskov Substitutability Principle

In order for a subclass to be substitutable for the base class it must:

- Accept the same messages (Should have all the methods of the base class with matching signatures)
- Require no more and promise no less than the base class in its methods (Weaker or same precondition, stronger or same postcondition)
- Match the observable behaviour of the base class (invariants, performance)

Law of Demeter

What is the Law of Demeter?

Law of Demeter

An object should not go through chains of objects to get to the information it needs.
In a class A with method m, A::m should only call methods of:

- A
- A's data members
- A::m's parameters
- Objects created by A::m

The End