COMP 2150 – Object Orientation

Course Description

Calendar entry

Design and development of object-oriented software. Topics will include inheritance, polymorphism, data abstraction and encapsulation. Examples will be drawn from several programming languages. Prerequisite: COMP 2160; and one of COMP 2140 or COMP 2061.

General Course Description

By this point students have learnt the basics of object-oriented programming, the differences between instance and class variables, and how to build simple class hierarchies in Java. Now it is time to dive deeper into the concepts of object orientation and understand how they work in multiple different object-oriented languages (Java, C++ and a dynamic OO programming language such as Ruby or JavaScript).

We use Java as a familiar object-oriented language. C++ offers low-level access to the mechanisms used to implement object-oriented techniques, as well as a wider variety of those techniques compared to other languages. A dynamic and dynamically-typed programming language introduces a different programming paradigm, and run-time polymorphic behaviours such as duck typing.

In this course we teach students how to use different object-oriented tools such as abstract classes, polymorphism, different forms of inheritance, shadowing, overriding, overloading, encapsulation, Java interfaces, and multiple inheritance. Students also learn the rudiments of good software design and the production of maintainable software.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Understand the basic concepts related to objects (instance and class variables/methods, access, and non-access modifiers).
- Manipulate objects in memory using pointers.
- Implement data structures such as lists, stacks, and binary trees in a high-level language.
- Build multi-file projects using automated tools.
- Perform well-defined and structured tests on code.
- Use a debugger to inspect program state and step through code line-by-line.

Course Goals

By the end of this course students will:

- Understand the benefits, in terms of code readability and maintainability, of compartmentalizing code into different objects and hierarchies of classes.
- Write large object-oriented programs, divided into multiple files, in Java, C++ and a dynamic programming language.
- Make use of object-oriented concepts like abstract classes, polymorphism, encapsulation and overriding in an efficient manner.
- Begin to understand the advantages and disadvantages of different programming languages.

Learning Outcomes

Basic OO concepts

Students should be able to:

1. Differentiate between classes, records, objects, and abstract data types.
2. Describe the meaning of object-oriented concepts such as hierarchies of classes, single/multiple inheritance, polymorphism, dynamic class binding and dynamic method-message mapping.
3. Explain how object-oriented concepts such as inheritance, polymorphism, and dynamic binding/messaging are implemented.

Basic OO concepts in Java and C++

Students should be able to:

1. Define and use classes using appropriate syntax for access modifiers, variables, methods, and constructors.
2. Use abstract classes and methods.
3. Build a simple hierarchy of classes.
4. Use polymorphism by defining virtual methods and reverse polymorphism by downcasting a variable. C++ only.
5. Write standard C++ code for file input/output operations.

Inheritance and methods

Students should be able to:

1. Describe 9 different forms of inheritance: specialization, specification, construction, generalization, extension, limitation, containment, variance, and combination.

2. Use substitution where an object and an object of a subclass are used interchangeably.
3. Distinguish between the behaviors of shadowing, overriding, redefinition and refinement of methods.
4. Define class methods and class variables in C++.

OCCF

Students should be able to:

1. Explain the Orthodox Canonical Class Form: standards ensuring that the same low-level operations work similarly for all objects.
2. Define a null constructor, an assignment operator, a destructor, and a copy constructor in a C++ class.
3. Use reference parameters and the reserved word `const` in C++.

Encapsulation

Students should be able to:

1. Explain and use the concept of information hiding.
2. Understand the differences between the public, protected and private access modifiers and use them appropriately.
3. Explain the ramifications of using friendship in C++.
4. Define their own Java packages and understand the "blank" access modifier (package protection) in Java.
5. Make use of the object pointer.

Java interfaces

Students should be able to:

1. Force a class to implement abstract methods by having it implement an interface.
2. Use interfaces as variable types, parameter types and return value types.
3. Build classes that implement multiple interfaces at the same time.
4. Define interfaces and hierarchies of interfaces, including extending multiple interfaces at the same time.
5. Add default and static methods inside an interface.

OO in a dynamic language

Students should be able to:

1. Write standard code in a dynamic language in the same manner as they would in Java and C++.

2. Instantiate objects of the same or similar types using a language specific mechanism, such as classes or prototypes.
3. Define objects as a collection of behaviours rather than a fixed type ("duck typing").
4. Apply metaprogramming for reflection and self-modification.
5. Compare the object-oriented capabilities (e.g., abstract classes, inheritance, shadowing, overriding, refinement and polymorphism) of a dynamic language with those of static and statically-typed languages.

Multiple inheritance

Students should be able to:

1. Build a class that implements multiple interfaces in Java.
2. Explain the ramifications of using multiple inheritance in C++ and how mitigate the impacts.

Introduction to design patterns

Students should be able to:

1. Understand the importance of design patterns in software development.
2. Distinguish between basic design patterns (adaptor, iterator, etc.).