# Comp 4603

# Advanced C++

|  |  |
| --- | --- |
| Assignment | 5 |

Student Name: Alisher Shamayev

BCIT ID: A01182685

Task 1: To implement four classes (Animal, Cat, Dog, Poodle) to showcase virtual functions and pure virtual functions. Submit your .cpp files and .hpp files into the drop-box

Task 2: To explain each principle in SOLID and make your own example for each

1. Single Responsibility Principle

Explanations:

|  |
| --- |
| This principle is for times when we have a class with one single responsibilities and that responsibilities used within that class. |

Example in C++ code:

|  |
| --- |
| **class** Car {  **public**:  string brand;  **int** price  string model;  };  **class** CarMarket {  **public**:  **void** postCar(Car car){}  **void** deletePost(Car car){}  }; |

1. Open-Closed Principle

Explanations:

|  |
| --- |
| This principle basically means that any class or modules or functions and etc. Must be open for extending it however can not be modified, so basically the source code should not be touched but can have a new added functionality. |

Example in C++ code:

|  |
| --- |
| **class** Shape {  **public**:  **virtual** **void** draw() = 0;  };  **class** Circle : **public** Shape {  **public**:  **void** draw() **override** {  cout<<"Draws a circle"<<endl;  }  }; |

1. Liskov Substitution Principle

Explanations:

|  |
| --- |
| This principle in short saying that a derived class should be able to substitute for objects of base class without causing any errors in the system. |

Example in C++ code:

|  |
| --- |
| **class** Shape {  **public**:  **virtual** **void** draw() = 0;  };  **class** Circle : **public** Shape {  **public**:  **void** draw() **override** {  cout<<"Draws a circle"<<endl;  }    };  **void** liskovSubstitution(**const** Shape& shape) {  shape.draw();  }  **int** main() {  Circle Circle;  liskovSubstitution(Circle);  **return** 0;  } |

1. Interface Segregation Principle

Explanations:

|  |
| --- |
| This principle is for times when we want software, classes to use interfaces they need instead of making bunch of classes use interfaces that they don’t need. In short we want interfaces that serves certain tight purposes that classes can use, rather than making one big interfaces with multiple tasks. |

Example in C++ code:

|  |
| --- |
| **class** Shape {  **public**:  **virtual** **double** area() **const** = 0;  **virtual** **double** perimeter() **const** = 0;  };  **class** Circle : **public** Shape {  **public**:  **double** radius;  **const** **double** pi = 3.14;    **double** area() **const** **override** {  **return** pi \* radius \* radius;  }  **double** perimeter() **const** **override** {  **return** 2 \* pi \* radius;  }  }; |

1. Dependency Inversion/Injection

Explanations:

|  |
| --- |
| Inversion principle is when we want make high level modules not depend on low level model, and also trying to make both depend on abstraction.  Injection helps us to change dependencies during compile or run time and not more needed dynamically typed dependencies. |

Example in C++ code:

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  **class** Uber {  **public**:  **virtual** **void** uberService(string message) = 0;  };  **class** UberEats : **public** Uber {  **public**:  **void** uberService(string message) {  cout << message << endl;  }  };  **class** User {  **private**:  Uber\* uberUser;  **public**:  User(Uber\* uberUser) : uberUser(uberUser) {}  **void** orderFood(string cafe) {  uberUser->uberService("placing order: " + cafe);    }  };  **int** main() {  UberEats uberEats1;  User User(&uberEats1);  User.orderFood("Wendy's");  **return** 0;  } |