Good Programming Practise

Four Object Orientated Programming Principles

Abstract

Abstract principle is about defining the structure of your program by breaking it down and removing the unnecessary information. Abstract principle is about the thought process behind identifying roles and responsibilities and collaborations for objects. This is performed by creating a UML diagram.

Inheritance

Inheritance principle is about having a generalised base class that has a specialized derived class sharing what the object knows(fields) and what the object does(methods) This saves repeating code for similar objects.

Encapsulation

Encapsulation principle is about what an object knows and does. Also encapsulation is about what things outside the object know depending if the object has a property or not or if the method/field is using protected, private or public types.

Polymorphism

polymorphism is the process of child or specialised classes possessing methods from the parent/Generalised class and modifying them to perform different behaviours. Polymorphism heavily depends on inheritance as classes need to inherit the methods to be able to change them. There are 4 different types of polymorphism. Subtype, Adhoc, Parametric and Coercion. Subtype is the main type of polymorphism as it is using derived classes and base classes to change methods. Parametric gives a way to execute code for different types. adhoc allows you to use methods to act differently for different types. Coercion is used when an object is cast into another object.

Program design

Keep the program simple, not having too much in one object, break it up into components by delegating tasks as much as possible.

Classes should limit the amount of information that can be accessed outside the class as much as possible to reduce bugs.

Child/Derived classes have similar conditions and constraints to the parent/base class so the child is a good substitute for the parent by collaborating through abstrctions.

Use UML diagrams to create visualization the design of you program. You can use Structure diagrams to emphasize the methods, fields and properties that belong to the class. It also shows their relation between different objects and which have children/parent structure to inherit certain methods and fields from the generalized class. Behaviour diagrams show the step to step operation of you program, showing the order in how it will run. Interaction diagram displays what objects methods access other objects methods to share data through the program.

Programs require good resource management as any object created must also be destroyed or it would lead to memory leaks.

Design Patterns should be used to increase productivity. Roles and responsibilities should be factored into classes, appropriate inheritance hierarchies must be defined, appropriate delegation and collaboration must be established. All this should be done before you start writing your code.

Code writing

Always comment and document methods and fields with XML so anyone can know what each components purpose is.

Use proper naming conventions such as Camel case for fields and methods. Also for consistency have local private/protected fields starting with an underscore and properties with full caps.

Always use a consistent indent style for the specific language so it makes it easy to read and understand code.

Learn proper syntax for the language you are writing in, each language has different ways of doing the same things.

Development practises

When creating a new component of your program always test it before you move on or you could create bugs and not know the source.

Use Unit Testing to test features, constructors and values of you objects to make sure they are all working properly.

Reflecting on how these practises are demonstrated in the unit

The code writing was demonstrated in all of the tasks as documents should always meeting conventions such as naming and formatting. This promotes good practise so when you come back and look at the code or others look at the code they can easily understand what everything does.

Development practises started on task 6 when we had to do unit testing on the shape task to make sure all the methods were performing the task they should be. From there on we used unit testing to make sure everything was working in most tasks like the case study, shape drawer, spells and Planet rover tasks. In C++ we used cppunit to test the program. Unit testing is Important to do so you can confirm that all methods are working correctly.

Abstract

An example of using the abstract was when we needed to create a UML diagram for pass task 12 when we needed to make a UML diagram for the swinwarts game to include the spellbook, spell, invisibility, teleport and heal class. Also for pass task 15 we needed to create the Planetary rover UML diagram so we know what the Rover, battery, and devices know such as fields and what they can do such as Methods. Abstract type was very useful for planning what the objects in the program would do for this unit. It would greatly help make sure that an object had all the components needed before coding so that when you are actually writing the code, you can read off the plan and know what methods and fields to write.

Inheritance

An example of Inheritance was task 11 shape drawer. In the shapeDrawer we created a shape class that generalised as a base class. Then the task had me making different types of shapes such as triangle, square and line. They all shared the x and y positions and the abstract drawShape method. Inheritance in this unit is very useful for reusing code when you have similar objects that can share methods and fields.

Encapsulation

Encapsulation was used in all the tasks for this unit. It was very important for the unit as it is the basis of every object in a program. It was very useful in the unit to block access to the fields for an object, outside the class so the values couldn't be changed by another object accidentally.

Polymorphism

An example of using polymorphism in the tasks was the Shape Drawer Program. In the shape drawer program each shape required to be drawn so we created a DrawShape method in the parent class. Polymorphism was needed to make the class abstract so we could overrule the method in the child classes so they could each perform SwinGame's unique drawcircle, drawrectangle and drawline methods. Polymorphism was great to use in the unit so if I needed to modify methods in derived/child classes, I could easily change what the pre-existing method performs without the need to recreate new methods for every derived classes.