(9.4) Software Reuse

Software Reuse Through Inheritance

Inheritance is one of the key principles of object-oriented programming (OOP), and it plays a significant role in promoting software reuse. By allowing new classes (called subclasses) to inherit the behaviour (methods) and properties (fields) of existing classes (called superclasses), inheritance enables developers to avoid duplicating code, create more efficient programs, and ultimately save time and reduce errors.

1. Avoiding Redundant Code:

Code Reusability: One of the biggest advantages of inheritance is that it allows subclasses to reuse code from the superclass. Inheritance allows a subclass to extend or modify the functionality of a superclass. A subclass can override methods from the superclass to provide specialized behaviors and attributes, which reduces the need for entirely new implementations or rewriting that functionality.

2. Faster Program Development Time

Building on Existing Code: Instead of creating a new class from scratch, we can create subclasses that inherit from pre-existing superclasses. This saves considerable time, since the code that has already been tested and debugged, the functionality defined in the superclass is already available for use. The subclass can focus on additional, specific behavior rather than re-implementing features already handled in the superclass.

For example, consider an application that models different types of vehicles. A general class Vehicle might have properties like color, speed, and methods like startEngine(). Then, specific types of vehicles like Car, Truck, and Motorcycle can inherit from Vehicle and automatically gain all of the shared functionality without needing to repeat the code for each subclass.

3. Reduced Errors and Improved Reliability

Centralized Changes: Inheritance promotes maintainability. When shared code is written in a superclass, any changes to that code only need to be made in one place. These changes automatically applies to all subclasses, ensuring consistency and reducing the chances of errors due to inconsistent updates.

For example, if a superclass Shape defines a method calculateArea(), and its subclasses Circle and Rectangle inherit from Shape, updating the method in Shape (for instance, to handle edge cases better) ensures that all subclasses benefit from the update without needing individual modifications.

Preventing Errors through Polymorphism: Inheritance promotes polymorphism, which allows a superclass reference to point to an object of a subclass. This enables more flexible, reusable, and error-free code. For example, you could use a method in the superclass to call a specific method in any subclass that overrides it, without needing to know the exact subclass type. This reduces the risk of introducing errors by hardcoding specific types and behavior.

Well-Defined Interfaces: By using inheritance, superclasses can define abstract methods (in abstract classes or interfaces), which require subclasses to implement them. This forces subclasses to provide the necessary functionality, leading to more predictable and structured behavior in the program.

**(9.5) Student Inheritance Hierarchy**

Student

UndergraduateStudent

**Freshmen**

GraduateStudent

**Sophomore**

**Sophomore**

**Sophomore**

**Sophomore**

**MasterStudent**

**DoctoralStudent**

Relationships:

Student (Superclass):

This is the base class for all types of students in the hierarchy. Some common attributes might include student ID, name and GPA. It serves as a foundation for all student types, and methods like enrollInCourse() and withdrawFromCourse() could be common to all students.

UndergraduateStudent (Subclass of Student):

Inherits from Student and represents students who are pursuing a bachelor's degree. Under this category, we have further classifications based on the student's year in school.

GraduateStudent (Subclass of Student):

Inherits from Student and represents students pursuing advanced degrees like a master's or doctoral degree.

Freshman, Sophomore, Junior, Senior (Subclasses of UndergraduateStudent):

These classes further specialize undergraduate students based on their year in school. A Freshman is typically a first-year student, a Sophomore is in their second year, a Junior is in their third year, and a Senior is in their final year. These classes might include additional attributes or methods that deal with year-specific behaviors or requirements (e.g., credit hours required).

DoctoralStudent, MastersStudent (Subclasses of GraduateStudent):

These are subclasses of GraduateStudent. A MastersStudent is pursuing a master's degree, and a DoctoralStudent is pursuing a doctoral degree. These subclasses can have specific attributes or methods related to their respective programs, such as thesis requirements for master's students or dissertation requirements for doctoral students.

Key Relationships:

**Is-A Relationship:** Each class in the hierarchy represents a type of student, so there is an "Is-A" relationship between the subclasses and their parent classes. For example, a Freshman is an UndergraduateStudent, which in turn is a Student.

**Generalization:** Student is a general class, and all other classes are more specific types of students.

**Specialization:** Each subclass represents a more specialized type of student with additional attributes or behaviors.

**9.6) Shape Inheritance Hierarchy**

**Shape**

**Pyramid**

**Cylinder**

**Cube**

**Sphere**

**Circle**

**Cone**

**Quadrilateral**

**Polygon**

**ThreeDimensionalShape**

**TwoDimensionalShape**

**Tetrahedronee**

**Triangle**

**Square**

**Rectangle**

**Ellipse**

**Parallelogram**

**Rhombus**

(9.7) **Quadrilateral Inheritance Hierarchy**

**Quadrilateral**

**Parallelogram**

**Trapezoid**

**Rectangle**

**Square**

**1. Class Point**

This class represents a point in a 2D coordinate system.

Instance Variables:

x (the x-coordinate of the point)

y (the y-coordinate of the point)

Methods:

Getter and setter methods for x and y.

**2. Quadrilateral Class (Superclass)**

This class represents a generic quadrilateral, which is a four-sided figure.

Instance Variables:

Four Point objects representing the four corners of the quadrilateral.

Methods:

A method area() to calculate the area, but this will be overridden by subclasses, as the area calculation depends on the specific type of quadrilateral.

**3. Trapezoid Class (Subclass of Quadrilateral)**

A trapezoid is a type of quadrilateral with one pair of parallel sides.

Instance Variables:

Inherited from Quadrilateral.

height (the perpendicular distance between the parallel sides).

Methods:

area() method calculates the area using the formula:

Area = 0.5 × (Base1 + Base2) ×Height.

**4. Parallelogram Class (Subclass of Quadrilateral)**

A parallelogram is a quadrilateral where opposite sides are parallel and equal in length.

Instance Variables:

Inherited from Quadrilateral.

base (the length of one of the sides).

height (the perpendicular distance from the base to the opposite side).

Methods:

area() method calculates the area using the formula:

Area = Base × Height.

**5. Rectangle Class (Subclass of Parallelogram)**

A rectangle is a special type of parallelogram where all angles are right angles.

Instance Variables:

Inherited from Parallelogram.

The base and height are used as the length and width of the rectangle.

Methods:

area() method calculates the area using the formula:

Area = Length × Width.

**6. Square Class (Subclass of Rectangle)**

A square is a special type of rectangle where all sides are of equal length.

Instance Variables:

Inherited from Rectangle.

Since all sides are equal, only one side length needs to be specified.

Methods:

area() method calculates the area using the formula:

Area = Side 2.