Object-Oriented Concepts and Design Principles

Signature

- Specifying an object operation or method involves declaring its name, the objects it takes as parameters and its return value. Known as an operation signature

Interface

- Set of all signatures defined by an objects operations is known as the objects interface

Type

- A type denotes a particular interface, e.g. Window, Shape, Stack
- directly supported in Java & C# with the notion of an ***interface*** as opposed to a class
- A GUI object can be of types: JFrame and ActionListener, could easily be extended to MouseListener, Shape etc.
- A type is part or all of the interface
- Widely differing objects may share a type, e.g. both the traffic lights and shop example both inherit the Frame interface and implement the ActionListener interface

Subtype

- A type is a **subtype** of another if it contains its interface contains that of its **supertype**

Dynamic Binding and Polymorphism

- dynamic binding doesn't commit an objects response to a message to a particular implementation until run-time
- can write programs that expect a particular interface and know that any object with that interface will respond to the message
- allows substitution at run time of objects which share the same interface
- known as polymorphism
- Polymorphism:

  simplifies definition of clients

  decouples from each other

  allows them to vary their relationships at run time

Class

- defines an objects implementation, specifies internal data and representation and defines the operations an object can perform
- consequently an objects response (the method it invokes) to a message is fixed at compile time

Abstract Class

- a class whose main purpose is to define a common interface for its subclasses
- defers some or all of its implementation to its subclasses
- cannot be instantiated
- only way C++ can come close to what an interface class does in Java

Class Inheritance and Interface Inheritance

- difference between an object's class and type: type only refers to set of messages to which object can respond while class defines object's internal state and operations
- since a class defines an object's operations, it also specifies its interface or type
- class inheritance defines an object's implementation in terms of another's, i.e. is a mechanism for code reuse and representation sharing. Interface inheritance or ***subtyping*** defines when an object can be substituted in another's place at run-time
- Java has another type of inheritance for specifying interface inheritance only, namely the ***interface*** *class* construct.
- In C++ and Eiffel inheritance means both interface and implementation inheritance.
- In Smalltalk, inheritance means implementation inheritance Pure interface inheritance can be achieved in C++ by inheriting publicly from pure abstract classes

Inheritance: Raison d'Etrè

- Implementation reuse is only half the reason for inheritance.
- Its ability to define objects with identical interfaces is the basis for polymorphism (proper use!)

Benefits of programming to an interface:

1. clients remain unaware of the specific types of objects they use as long as the objects adhere to the interface expected by the client
2. clients remain unaware of the classes that implement the objects

- greatly reduces implementation dependencies
- leads to principle

Program to an interface and not an implementation

- don't declare variables to be instances of concrete classes if possible

Class Inheritance versus Composition

- 2 most common techniques for reusing functionality in object-oriented systems,
- inheritance – allows object to extend its functionality by adding to and/or modifying the data and operations it inherits from its parent class - **white box** reuse
- composition - obtain new functionality by assembling or composing objects to get more complex structure - **black-box** reuse

Inheritance

Advantages

- defined statically at compile-time and easy to use. Directly supported by programming languages
- makes it easy to modify the implementation being reused – operation overriding
- provides a natural or intuitive implementation of specialisation/generalisation relationships, e.g. *BankAccount* and *CurrentAccount*
- inter-object behaviour at run time may be easier to follow as more of the program structure is decided at compile time

Disadvantages

- can’t change inherited implementation of operations at run-time as they are fixed at compile-time
- even worse, parent classes often define at lease part of their subclasses’ physical representation which leads to tight coupling between parent class and its subclasses
- implementation of a subclass is so tightly bound to that of its parent class that any modification of parent class will force change in subclass
- this breaks the spirit of encapsulation, violating a basic Object-Oriented principle
- one way around this is to inherit from abstract classes (or interface classes in Java)
- A subclass cannot rely on testing of parent class, it must be fully tested itself
- With inheritance, extra, optional features for an object would require extensive and unnecessary subclassing (an individual subclass for each option) which would complicate the code and class hierarchy unnecessarily

Composition

- object composition defined dynamically at run-time trough objects acquiring references to other objects, any object can be replaced at run-time by another once it has the same type
- requires that objects respect each others' interfaces, thus not breaking encapsulation
- because object implementation is written in terms of object interfaces, there are much less implementation dependencies
- favouring composition over inheritance :
  - helps keep classes encapsulated and focused on one task – high cohesion
  - leads to smaller and more manageable classes and class hierarchies
- on the other hand composition:
  - will give rise to a greater number of objects
  - system behaviour will depend on their interrelationships instead of on defined in class structures

Design Principle

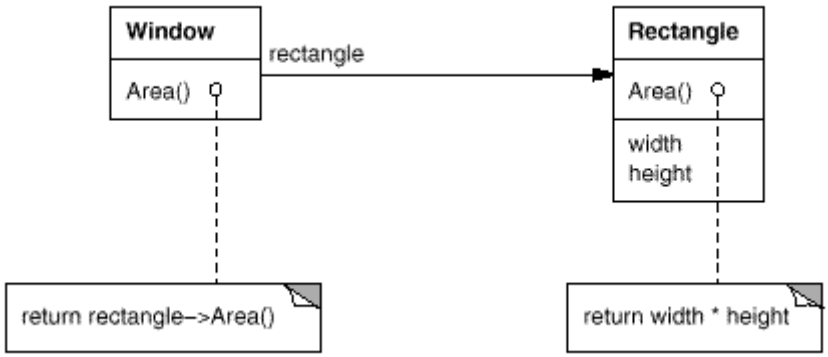
Favour object composition over class inheritance

- Ideally, should be able to get all functionality by assembling existing components. Not so in practice, usually set of existing components not rich enough. Reuse by inheritance makes it easy to create new components which can be composed with old ones.

- Thus inheritance and object composition work together

Delegation

- a way for making composition as powerful for reuse as inheritance
- For example, instead of making Window a subclass of Rectangle, Window reuses behaviour of Rectangle by keeping a rectangle instance variable and delegating rectangle specific behaviour to it



- delegation makes it easy to compose behaviours at run-time and to change the way they are composed. E.g. change a Window to oval at run-time by substituting the rectangle reference to an oval reference assuming they both share a same type such as ***Shape***
- harder to understand than more static software
- run-time inefficiencies, not serious criterion
- use only when it simplifies more than it complicates