Mastering Inheritance and Composition

Although inheritance is one of the fundamental constructs of object-oriented development, many developers are increasingly turning away from inheritance in lieu of other design strategies.

Regardless, both inheritance and composition are mechanisms for reuse. *Inheritance*, as its name implies, involves inheriting attributes and behaviors from other classes.

Composition, also as its name implies, involves building objects by using other objects.

Reusing Objects

Perhaps the primary reason that inheritance and composition exist is object reuse. YOu can build classes by utilizing other classes via inheritance and composition, which in effect , are the only ways to reuse previously build classes.

Inheritance represents the is-a relationship

Composition involves using other classes to build more complex classes--a sort of assembly.

Composition represents a has-a relationship.

Over time the luster of inheritance has dulled a bit. In fact, even in some early discussions, the use of inheritance itself is questioned. Many early object-based platforms did not even support true inheritance. As Visual Basic evolved into Visual BAsic .NET, early object-based implementations did not include strict inheritance capabilities. Platforms such as the MS COM model were based on interface inheritance.

Today, the use of inheritance is still a major topic of debate.

In actuality, both inheritance and composition are valid class design techniques, and they each have a proper place in the OO developer’s toolkit. And, at least, you need to understand both to make the proper design choice.

The fact that inheritance is often misused and overused is more a result of a lack of understanding of what inheritance is all about than a fundamental flaw in using inheritance as a design strategy.

Inheritance and composition are both important techniques in building object-oriented systems.

Generalization and Specialization

The idea of Inheritance is to go from the general to the specific by factoring out commonality.

Design Decisions

Factoring out as much commonality as possible is great. however , as in all design issues, sometimes it really is too much of a good thing. Although factoring out as much commonality as possible might represent real life as closely as possible, it might not represent your model as closely as possible. The more you factor out, the more complex your system gets. So you have a conundrum: Do you want to live with a more accurate model or a system with less complexity? You have to make this choice based on your situation, for there are no hard guidelines to make the decision.

There will be instances in your design when the advantage of a more accurate model does not warrant the additional complexity.

Deciding whether to design for less complexity or more functionality is a balancing act. The primary goal is always to build a system that is flexible without adding so much complexity that the system collapses under its own weight.

Current and future costs are also a major factor in these decisions.

Composition

It is natural to think of objects as containing other objects.

Whenever a particular object is composed of other objects, and those objects are included as object fields, the new object is known as a *compound*, an *aggregate*, or a *composite*.

From the author’s perspective, there are only two ways to reuse classes--with inheritance or composition.

Representing Composition with UML (Unified Modeling Language)

Using too much composition can also lead to more complexity. A fine line exists between creating an object model that contains enough granularity to be sufficiently expressive and a model that is so granular that it is difficult to understand and maintain.

Why Encapsulation is Fundamental to OO

When talking about a class, the primary design decision revolves around encapsulating both the data and the behavior into a well-written class.

How Inheritance Weakens Encapsulation

Encapsulation is the process of packaging classes into the public interface and the private implementation.

The problem is that if you inherit an implementation from a superclass and then change that implementation, the change from the superclass ripples through the class hierarchy. This rippling effect potentially affects all the subclasses. A rippling effect such as this can cause unanticipated problems.

If the subclass were truly a specialization of the superclass, changes to the parent would likely affect the child in ways that are natural and expected.

A Detailed Example of Polymorphism

Designing a class for the purpose of creating totally independent object is what OO is all about. In a well designed system, an object should be able to answer all the important questions about it. As a rule, an object should responsible for itself. This independence is one of the primary mechanisms of code reuse.

Polymorphism literally means *many shapes*. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. HOwever, because each subclass is a separate entity, each might require a seprate response to the same message.

Object Responsibility

Polymorphism is one of the most elegant uses of inheritance.

Although Rectangle and Circle are both shapes, they have some differences. As shapes, their area can be calculated. Yet the formula to calculate the area is different for each. Thus, area formulas cannot be placed in the shape class.

Here is the important point regarding polymorphism and an object being responsible for itself: The concrete classes themselves have responsibility for the drawing function.