PrinciplesOfOod [add child]

The Principles of OOD

What is object oriented design? What is it all about? What are it's benefits? What are it's costs? It may seem silly to ask these questions in a day and age when virtually every software developer is using an object oriented language of some kind. Yet the question is important because, it seems to me, that most of us use those languages without knowing why, and without knowing how to get the the most benefit out of them.

Of all the revolutions that have occurred in our industry, two have been so successful that they have permeated our mentality to the extent that we take them for granted. Structured Programming and Object Oriented Programming. All of our mainstream modern languages are strongly influenced by these two disciplines. Indeed, it has become difficult to write a program that does not have the external appearance of both structured programming and object oriented programming. Our mainstream languages do not have **goto**, and therefore appear to obey the most famous proscription of structured programming. Most of our mainstream languages are class based and do not support functions or variables that are not within a class, therefore they appear to obey the most obvious trappings of object oriented programming.

Programs written in these languages may look structured and object oriented, but looks can be decieving. All too often today's programmers are unaware of the principles that are the foundation of the disciplines that their languages were derived around. In another blog I'll discuss the principles of structured programming. In this blog I want to talk about the principles of object oriented programming.

In March of 1995, in comp.object, I wrote an article that was the first glimmer of a set of principles for OOD that I have written about many times since. You'll see them documented in my PPP book, and in many articles on the objectmentor website, including a well known summary.

These principles expose the dependency management aspects of OOD as opposed to the conceptualization and modeling aspects. This is not to say that OO is a poor tool for conceptualization of the problem space, or that it is not a good venue for creating models. Certainly many people get value out of these aspects of OO. The principles, however, focus very tightly on dependency management.

Dependency Management is an issue that most of us have faced. Whenever we bring up on our screens a nasty batch of tangled legacy code, we are experiencing the results of poor dependency management. Poor dependency managment leads to code that is hard to change, fragile, and non-reusable. Indeed, I talk about several different *design smells* in the PPP book, all relating to dependency management. On the other hand, when dependencies are well managed, the code remains flexible, robust, and reusable. So dependency management, and therefore these principles, are at the foudation of the *-ilities* that software developers desire.

The first five principles are principles of *class design*. They are:

SRP	The Single Responsibility Principle	*A class should have one, and only one, reason to change.*
OCP	The Open Closed Principle	*You should be able to extend a classes behavior, without modifying it.*
LSP	The Liskov Substitution Principle	*Derived classes must be substitutable for their base classes.*
ISP	The Interface Segregation Principle	*Make fine grained interfaces that are client specific.*
DIP	The Dependency Inversion Principle	*Depend on abstractions, not on concretions.*

The next six principles are about packages. In this context a package is a binary deliverable like a .jar file, or a dll as opposed to a namespace like a java package or a C++ namespace.

The first three package principles are about package *cohesion*, they tell us what to put inside packages:

REP	The Release Reuse Equivalency Principle	*The granule of reuse is the granule of release.*
CCP	The Common Closure Principle	*Classes that change together are packaged together.*
CRP	The Common Reuse Principle	*Classes that are used together are packaged together.*

The last three principles are about the couplings between packages, and talk about metrics that evaluate the package structure of a system.

ADP	The Acyclic Dependencies Principle	*The dependency graph of packages must have no cycles.*
SDP	The Stable Dependencies Principle	*Depend in the direction of stability.*
SAP	The Stable Abstractions Principle	*Abstractness increases with stability.*