



COMPUTING INTEGRITY

INCORPORATED

60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile



Object-Oriented Software Design Principles¹

23 January 2010

Thomas Mercer-Hursh, Ph.D.

In choosing design patterns and writing object-oriented code² there are a number of design principles which have been expressed over the years by writers in the OO world. The purpose of this document is to gather together some of these principles as a reference for use by OOABL programmers. Some of these principles may seem similar or overlapping, but they are talking about different viewpoints or contexts based on the same core values. This overlap should be seen as complimentary and reinforcing rather than redundant.

Separation of Concerns (SoC)

Some might say that the core principle in OO programming is Separation of Concerns, although it is a concept which is not unique to OO. Separation of Concerns is the idea of dividing an application into distinct components that overlap in functionality as little as possible. I.e., a “concern” is a focus or responsibility and the associated attributes and behavior. Separation of Concerns is achieved by modularization and specifically through encapsulation in OO programming. Layered designs are another aspect of Separation of Concern.

SOLID Principles

The next five principles taken together are known as the SOLID principles of Object-Oriented Design³ from the first initial of their names.

Single Responsibility Principle (SRP)

One could say that the Single Responsibility Principle is the logical compliment of Separation of Concerns, i.e., that each responsibility or concern should be a separate class. A responsibility is an axis of change, so any change to the responsibility should impact only one class. No change to any other responsibility, e.g., business rules outside this responsibility, UI, database schema, report format, or other part of the system should require this class to change. Some people call this the One Responsibility Rule and express it as “A class has a single responsibility: it does it all, does it well, and does it only.”⁴

¹ See Mercer-Hursh, Thomas - *Object-Oriented Vocabulary: An Introduction*, for definitions of the words used in this document. See Mercer-Hursh, Thomas - *Object-Oriented Design Patterns: An Introduction*, for some common design patterns used in OO development which reinforce and support the design principles discussed here.

² While the context of this document is intended for use in relation to Object-Oriented ABL, the principles involved should apply to any OO language.

³ This group and the two groups following on package cohesion and package coupling are derived from Robert Cecil Martin’s compilation at <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>.

⁴ Bertrand Meyer, *Object Oriented Software Construction*.

Open Closed Principle (OCP)

Classes, or other software components, should be open for extension, but closed for modification. I.e., ideally, one should never have to change an existing class, but rather should be able to extend it and re-use it in new contexts. The intention is that one should not have to modify software that is already working and thus potentially introduce bugs. Of course, if the definition of the responsibility which the class represents changes, it may be necessary or appropriate to change the class, but if the essential responsibility which the class represents has not changed, then the class should not change.

Liskov Substitution Principle (LSP)⁵

If a class X is a subtype of class Y, then any instance of class X should be substitutable in any context in which class Y was expected. In other words, any true subtype is a behavioral substitute for the parent and has at least the same knowledge as the parent.

Interface Segregation Principle (ISP)

The dependency of one class on another should be based on the smallest possible interface. I.e., components of a system should be decoupled to the extent possible and no class should need to know about methods which do not pertain to it. This makes it easier to refactor and change a system. Note, if an interface defines the invariants of a particular concern or subject matter, that is inherently the smallest possible interface for that concern or subject matter. I.e., the issue is less the number of elements in an interface than it is decomposition to intrinsic cohesive concerns in the problem space.

Dependency Inversion Principle (DIP)

High level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. These principles are intended to avoid fragile, rigid, and immobile designs.

Package Cohesion Principles

The next three principles relate to the cohesion of classes within packages, i.e., how does one decide what should be within the package versus what should not.

Reuse Release Equivalence Principle (REP)

The unit of reuse is the unit of release. In order to effectively reuse code, it must be available as a complete black box. Thus, users of the code are shielded from changes to the code and only need to modify their own code when and if they desire to take advantage of new functionality which has been added. It is sometimes called the Black Box Principle.

Common Closure Principle (CEP)

Classes that are tightly coupled with each other should be part of the same package; unless classes are tightly coupled, they should not be part of the same package. While individual classes within a package may not be tightly coupled one to the other, the classes in the package as a whole should be tightly coupled or they should not be a package. This is an extension of the idea of the cohesion within a class and its separation from other classes, but at the package level.

⁵ Defined by Barbara Liskov in 1988.

Common Reuse Principle (CRP)

The classes in a package should be reused together, i.e., as a unit. If you reuse one class in a package, you should be reusing them all. Just as an individual class can be a unit of reuse, a well constructed package will be a unit of reuse. It might be suggested that, if reuse is at the package level, then the package should be provided with an interface which hides the individual classes.

Package Coupling Principles

The next three principles relate to how one package should interact with another.

The Acyclic Dependencies Principle (ADP)

The dependency structures between packages must not contain cyclic dependencies, i.e., if package A depends on package B, there must not be a path of dependency that leads back from B to A.

Stable Dependencies Principle (SDP)

The dependencies between packages should be in the direction of increasing stability, i.e., a package should always be dependent on a package that is more stable than it is. By “stable” here, it is meant that a package is hard or unlikely to change. Thus, the more malleable parts of an application should depend on the less malleable parts.

Stable Abstractions Principle (SAP)

The more stable a package, the more abstract it should be; unstable packages should be more concrete. I.e., the less likely a package is to change, the more abstract it should be.

Commonly Referenced Principles

The next three principles are very commonly cited in discussions about Object-Oriented.

Program to the Interface

Since an interface is the abstract representation of the behavior of a class, orienting one's programming to the interface ensures that one is focusing on the abstraction, not the implementation and simultaneously enables the class to work with any other class which implements the same interface. See *Design by Contract*⁶

Favor Composition over Inheritance

When dealing with a complex object, there is often a choice of whether to use inheritance to segregate specialized data and behavior into subclasses or whether to create a delegate or some other associated class to contain the variations in behavior for a particular aspect of the whole. Thus, in general, we should prefer moving sets of behavior into another class over making a complex inheritance hierarchy with multiple, overlapping areas of specialization. This follows from wanting to separate responsibilities and create cohesive classes that have closely focused scope.

Problem Space Abstraction

One of the primary characteristics of OO development is Problem Space Abstraction. Every object and package in an OO application abstracts an identifiable entity in some problem space. The application may deal with multiple problem spaces and the entities may be purely conceptual, but

⁶ Mercer-Hursh, Thomas. *Object-Oriented Design Patterns: An Introduction*.

they are readily identified by any domain expert familiar with the problem space. Objects are directly identified with individual problem space entities and Classes are directly identified with the common knowledge and behavior of a group of such problem space entities. Not only does this imply a very direct correspondence between the application design and the problem space, but it insures that change will be minimally disruptive since the structure of the application will only change when the structure of the problem space changes. Every other change will be internal to one of the entities.

Summary

As in many areas of endeavor, if one picks a different source, one will get a different list of principles. We could harvest from multiple sources and get more principles to add, but many would overlap with or repackage the principles above in different forms and emphasis. One author may state the principle as an absolute, another as a goal, and a third as dubious, often based on some specific aspect. Generally, the concepts behind the rule are still agreed, but the details of their interpretation can vary widely. Rather than merely extend this list with more principles, let us consider some of the common themes which these principles represent.

Common Themes

One of the pervasive themes in these principles is a sharp distinction between what belongs in a class or package and what is outside of it. This theme is core to OO thinking and is expressed strongly in Separation of Concerns and Single Responsibility Principle. Each class should be about one thing and should contain all of the attributes and behavior related to that thing. Each class should be strongly separated from other classes. This concept is what is meant by the term

Encapsulation.

Moreover, there is an emphasis on the idea that any one class exposes a “contract” to the outside world and nothing outside of the class should be aware of how that contract is implemented. This is called **Abstraction**. Together, these concepts are central to realizing the benefits of OO design since they mean that an application is divided into units, each of which can be modified internally without regard to other units and each of which contains all of the data and behavior about that particular piece of the application. This promotes re-use, stability, maintainability, and clarity. The same principles apply to packages, but more loosely.

A second pervasive theme in these principles is the notion that classes which have a relationship should be connected only by what is essential to the relationship. Packages, subsystems, and layers are *usually* **Loosely Coupled** because an entity in one does not even know what entity in the other will respond to its message, much less what they will do and how they will do it. However, objects within such units are more **Tightly Coupled** because they know exactly who they are collaborating with and what properties they have. This promotes re-use, stability, and maintainability.

The combination of Encapsulation and Loose Coupling leads one to speak of a Class as a **Black Box** whose interior is unknown and to which one connects only according to the contract. Related to this is the idea of **Substitutability** expressed in the Liskov Substitution Principle, i.e., if two classes share a part of their contract by virtue of having a common parent or interface, any other class can work with objects of either class according to that contract without having any awareness of the specific type of the object.

Generalization or Inheritance is sometimes spoken of as a **White Box** relationship since the implementation of the superclass is available to the subclass. This awareness of implementation contrasts with Composition or Delegation as Black Box relationships since those two objects interact solely according to their externally visible contracts. Thus, Composition is favored over Inheritance because there is a better separation of responsibility. Note that a Generalization Hierarchy as a whole is still a Black Box to an external client, although the designer of the client must be aware of the responsibilities exposed at each level of the hierarchy.

Inherent to these themes is the principle that any one attribute or behavior should only be in one place. Otherwise we would not have correctly subdivided the universe of data and behavior to create cohesive, unified units. Students of relational database systems will recognize this as the principle of **Normalization**.

In a number of these principles, the statement is about a principle in terms of properties of the design, but one should remember that every design is intended to be a model of a problem space. Thus, if one has analyzed the problem space correctly, the design principle follows naturally. For example, in Interface Segregation Principle, if one has defined interfaces in terms of cohesive, irreducible concerns in the problem space, by definition one has also created the minimal appropriate interface. In Stable Dependencies Principle, one could re-interpret the requirement as defining the interface capturing the invariants of the problem space at the client's level of abstraction. In Stable Abstractions Principle, if one defines interfaces in terms of invariants of the problem space, the more stable will be the definition.

Conclusion

While there is some difference of opinion about some details of some of the principles documented here, there is broad consensus in the OO world about the underlying principles on which they are based. Adhere to these principles and one can expect to more readily enjoy the benefits of object-orientation; violate the principles *without good reason* and those benefits may be denied.

There are reasons why one may not wish to adhere strictly to every principle in every circumstance. In particular, one might decide that there are characteristics of OOABL which indicate a pattern which seems to violate one of these principles and yet is a good manifestation of the underlying principles, but the choice should be made knowingly and carefully. There are often many ways to solve a given problem and it may be that more than one of these ways would seem to be an expression of one or the other of the principles reviewed here. Arriving at the best solution requires balance, judgment, and consideration of how the whole will function together, not just verification that each element can be defended according to some principle.