Three Sources of a Solid Object-Oriented Design

Design heuristics, scientifically proven OO design guidelines, and the world beyond the beginning

By: Gene Shadrin

Object-oriented design is like an alloy consisting of a solid grounding in the object-oriented (OO) approach and implementing the best OO practices heavily laced with how to sidestep the OO pitfalls. The design process isn't only a matter of applying basic OO principles. One should go further and achieve a reasonable tradeoff between OO design principles and applicable design patterns. This article will discuss the relationship between three sources of a solid OO design and offer a starting point to understanding what is, in effect, a complex process. It will also serve as a review for experienced designers.

The OO Design Pyramid

When developers first turn to object-oriented development, they try to grasp the basics. It may take several project lifetimes. Getting a full sense of object orientation is a big shift in people's minds, especially for those who come from the procedural world. Such principles are like ABC books since they define the foundation of the OO world.

Once people get a sense of the basic principles, they start feeling the power of the OO approach. In our experience this is a dramatic moment for every OO designer and developer. The euphoria of possessing a powerful methodology can make them feel completely self-sufficient and they stop and don't try for perfection. It takes time to realize that there are significant issues beyond the basics that can be found in the accumulated wisdom of the OO developer community and are expressed in a set of design heuristics and scientifically proven OO design guidelines. Some OO designers jump to using design patterns everywhere like a cookie recipe. Although design patterns are important and useful tools in building OO systems, they're just templates that define a common design language and significantly improve design quality, when properly used.

To build contemporary real-world enterprise-class systems, OO designers should (a) be proficient in basic OO principles, (b) master the principles of OO design, and (c) understand design patterns. Otherwise the chance of design flaws and total cost of system ownership increases. And, with growing adoption of Service Oriented Architecture (SOA), the robustness of the SOA services can depend directly on the solidity of the OO design of underlying components.

The pyramid in Figure 1 represents three sources of a solid OO design. It also defines a systematic view of design and uncovers dependencies between these sources.

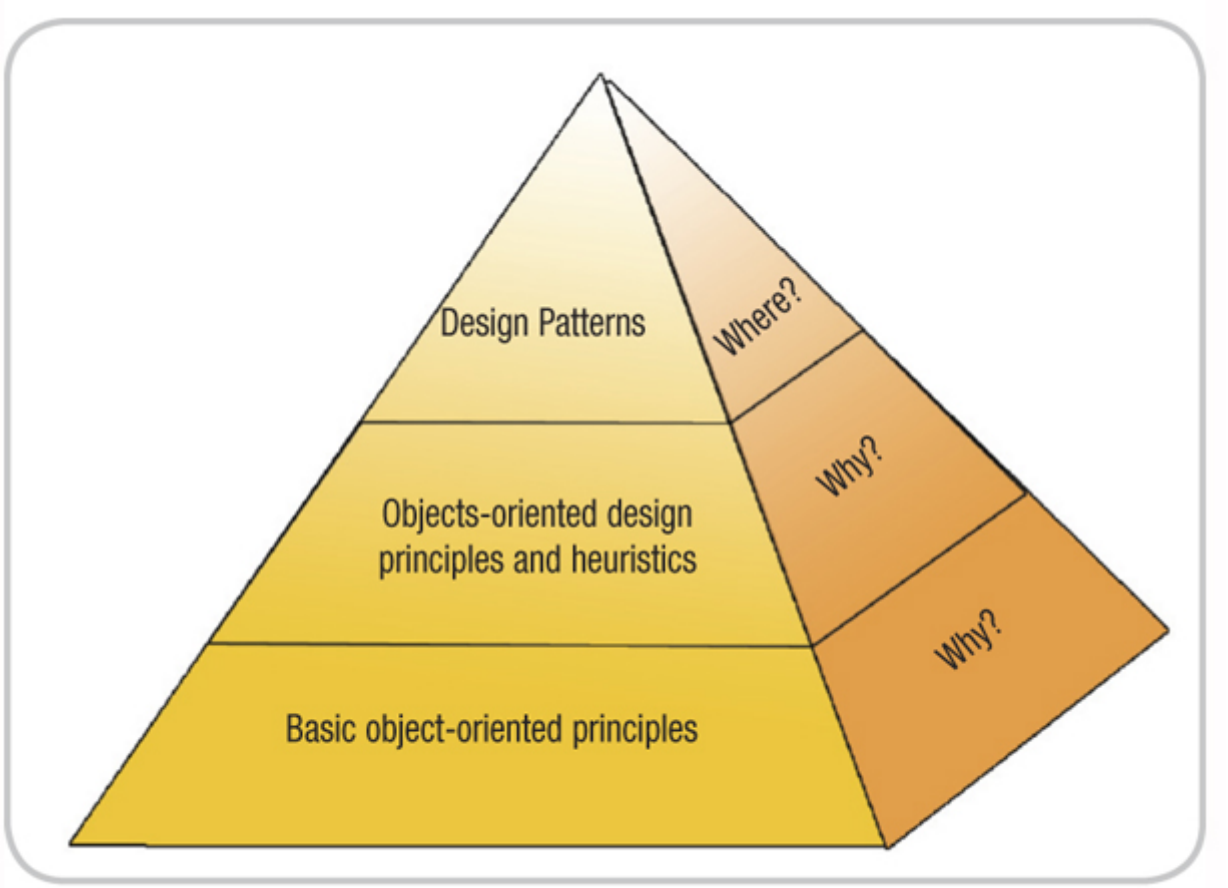

Figure 1 Design pyramid

Now let's discuss each level of this pyramid from an architectural and design point of view. Of course there's no way to explain these topics completely in such short article, but we'll try to highlight the necessity of paying attention to the dependencies and their impact on the quality of OO design.

Basic OO principles

The first level of the OO design pyramid is formed by a set of basic OO principles.

The most basic OO principles include encapsulation, inheritance, and polymorphism. Along with abstraction, association, aggregation, and composition, they form the foundation of the OO approach. These basic principles rest on a concept of objects that depicts real-world entities such as, say, books, customers, invoices, or birds. These classes can be considered templates for object instantiation, or object types. In other words, class specifies what an object can do (behavior) and defines patterns for its data (state). Modeling the real world in terms of an object's state and behavior is the goal of the OO approach. In this case, state is represented by a set of object attributes, or data, and behavior is represented by the object's methods.

Now we'll discuss the application of basic OO principles to modeling.

Encapsulation encloses data and behavior in a programming module. Encapsulation is represented by the two close principles of information hiding and implementation hiding. *Information hiding* restricts access to the object data using a clearly defined "interface" that hides the internal details of the object's structure. For each restricted class variable, this interface appears as a pair of "get" and "set" methods that define read-and-write access to the variable. *Implementation hiding* defines the access restrictions to the object methods also using a clearly defined interface that hides the internal details of object implementation and exposes only the methods that comprise object behavior. Both information and implementation hiding serve the main goal - assuring the highest level of decoupling between classes.

Inheritance is a relationship that defines one entity in terms of another. It designates the ability to create new classes (types) that contain all the methods and properties of another class plus additional methods and properties. Inheritance combines interface inheritance and implementation inheritance. In this case, *interface inheritance* describes a new interface in terms of one or more existing interfaces, while *implementation inheritance* defines a new implementation in terms of one or more existing implementations. Both interface inheritance and implementation inheritance are used to extend the behavior of a base entity.

Polymorphism is the ability of different objects to respond differently to the same message. Polymorphism lets a client make the same request of different objects and assume that the operation will be appropriate to each class of object. There are two kinds of polymorphism - *inheritance polymorphism*, which works on an inheritance chain, and *operational polymorphism*, which specifies similar operations for non-related out-of-inheritance classes or interfaces. Because inheritance polymorphism lets a subclass (subtype) override the operation that it inherits from its superclass (supertype), it creates a way to diversify the behavior of inherited objects in an inheritance chain, while keeping their parent-objects intact. Polymorphism is closely related to inheritance as well as to encapsulation.

Naturally, some OO principles are controversial in the sense that they contradict one another. For example, to be able to inherit from a class, one should know the internal structure of that class, while encapsulation's goal is exactly the opposite - it tries to hide as much of the class structure as possible. In real life the tradeoff between these two principles is a fine line that can't be established without stepping up to the next level in the design pyramid.

OO Design Principles and Heuristics

OO design principles and heuristics form the second level in OO design pyramid.

There are about a dozen OO design principles and four dozens OO design heuristics identified over the years by OO evangelists such as Grady Booch, Bertrand Meyer, Robert C. Martin, Barbara Liskov, and others. OO design principles define the most common scientifically derived approaches for building robust and flexible systems.

These approaches proved to be the best tools in solving numerous OO design issues that can't be captured by fundamental OO principles.

The class structure and relationships group consists of the following design principles: the Single Responsibility Principle (SRP), the Open/Closed Principle (OCP), the Liskov Substitution Principle (LSP), the Dependency Inversion Principle (DIP), and the Interface Segregation Principle (ISP).

The *Single Responsibility Principle* specifies that class should have only one reason to change. It's also known as the cohesion principle and dictates that class should have only one responsibility, i.e., it should avoid putting together responsibilities that change for different reasons.

The *Open/Closed Principle* dictates that software entities should be open to extension but closed to modification. Modules should be written so that they can be extended without being modified. In other words, developers should be able to change what the modules do without changing the modules' source code.

The *Liskov Substitution Principle* says that subclasses should be able to substitute for their base classes, meaning that clients that use references to base classes must be able to use the objects of derived classes without knowing them. This principle is essentially a generalization of a "design by contract" approach that specifies that a polymorphic method of a subclass can only replace its pre-condition by a weaker one and its post-condition by a stronger one.

Dependency Inversion Principle says high-level modules shouldn't depend on low-level modules. In other words, abstractions shouldn't depend on details. Details should depend on abstractions.

The *Interface Segregation Principle* says that clients shouldn't depend on the methods they don't use. It means multiple client-specific interfaces are better than one general-purpose interface.

The package cohesion group consists of the following design principles: the Reuse/Release Equivalency Principle (REP), the Common Closure Principle (CCP), and the Common Reuse Principle (CRP). This group deals with the principles that define packaging approaches based on class responsibilities (i.e., how strongly related the responsibilities of classes are).

REP makes release granularity equal to reuse granularity, *CCP* says classes that change together belong together, and *CRP* says classes that aren't reused jointly shouldn't be grouped together.

The package coupling group consists of the following design principles: the Acyclic Dependency Principle (ADP), the Stable Dependency Principle (SDP), and the Stable Abstractions Principle (SAP). This group deals with principles that define packaging approaches based on the packages' collaboration (i.e., how much one package relies on or is connected to another).

ADP prohibits forming cyclic dependencies among packages, *SDP* says package dependency should be allowed to reinforce package stability, and *SAP* says stable packages should be abstract packages.

Design heuristics derive from the practical experience of OO developers. They normally lie on top of design principles but can interrelate with them or even underlie design principles. Heuristics can extend design principles to several specific implementations. That's why they're on the pyramid along with design principles.

As design principles, heuristics are grouped by their application: class structure, object-oriented applications, relationships between classes and objects, inheritance relationships, association relationships, etc. Heuristics are less fundamental than design principles, but they clarify, explain, and expand design principles.

Both design principles and heuristics can be controversial. In other words, some design principles and heuristics have internal dissension, while others contradict each other. For example, conforming to the Open/Closed Principle can be expensive and lead to unnecessary complexity - the class model should be pertinent to a specific context. What do designers get if they achieve conformance? The answer is the greatest benefits of the OO paradigm - flexibility, reusability, and maintainability. Another example - the Liskov Substitution Principle restricts the use of inheritance while the Open/Closed Principle embraces it.

But for all the controversy inherent in design principles and heuristics, they still give the OO designer such a powerful and systematic basis for robust design that the resulting OO model created with their "help" is of immeasurably better quality than the ones built on just basic OO principles.

Design Patterns

The top level of OO design pyramid represents design patterns.

There are 23 basic design patterns identified by "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), 21 core J2EE patterns identified by the Sun Java Center, 51 patterns of enterprise application architecture identified by Martin Fowler et al., and 65 enterprise integration patterns. There are also a lot of patterns specific to particular problem domains.

Design patterns represent common structured solutions to design problems solved in a particular context. They're a guide to good design practices and span a wide range of solutions from general topics like object lifecycle and structure to more specific themes such as presentation, business, data, and integration tiers, data transfer, state management, message construction, routing, channels, and transformation. Each pattern describes intent, motivation, applicability, structure, participants, collaborations, consequences, and implementation, using one of the common notations (most often UML).

To manage design patterns better and simplify their application to real systems, all patterns are categorized. Categories reflect the approach to group them together. For example, the "Gang of Four" (GoF) patterns generally considered fundamental to all other patterns were categorized by their authors into three groups: Creational, Structural, and Behavioral. Java practitioner Steven John Metsker categorized these same patterns into different groups: Interfaces, Responsibility, Construction, Operations, and Extensions.

The rule of thumb is to try to apply patterns where application design would benefit from performance and flexibility. Sometimes, however, designers have to choose patterns based on just one "benefit." For example, to improve performance, designers should always apply the core J2EE patterns of Data Access Object, Front Controller, Session Façade, Service Locator, Transfer Object, Transfer Object Assembler, Value List Handler, and Composite Entity.

In most cases, it's possible to come up with appropriate design patterns for particular problems. Still, there are situations where design patterns either don't exist or offer an inefficient solution. In that case, the solution may rest with design principles and heuristics.

Putting It All Together

The sections above described the content of each OO design pyramid level. Interdependencies between pyramid levels are represented by a complex graph that confirms their controversial semantics. As seen in Figure 2, which represents relationships between the first and the second levels (basic OO principles and OO design principles), some OO design principles depend on multiple basic OO principles while others are drawn based on other considerations.

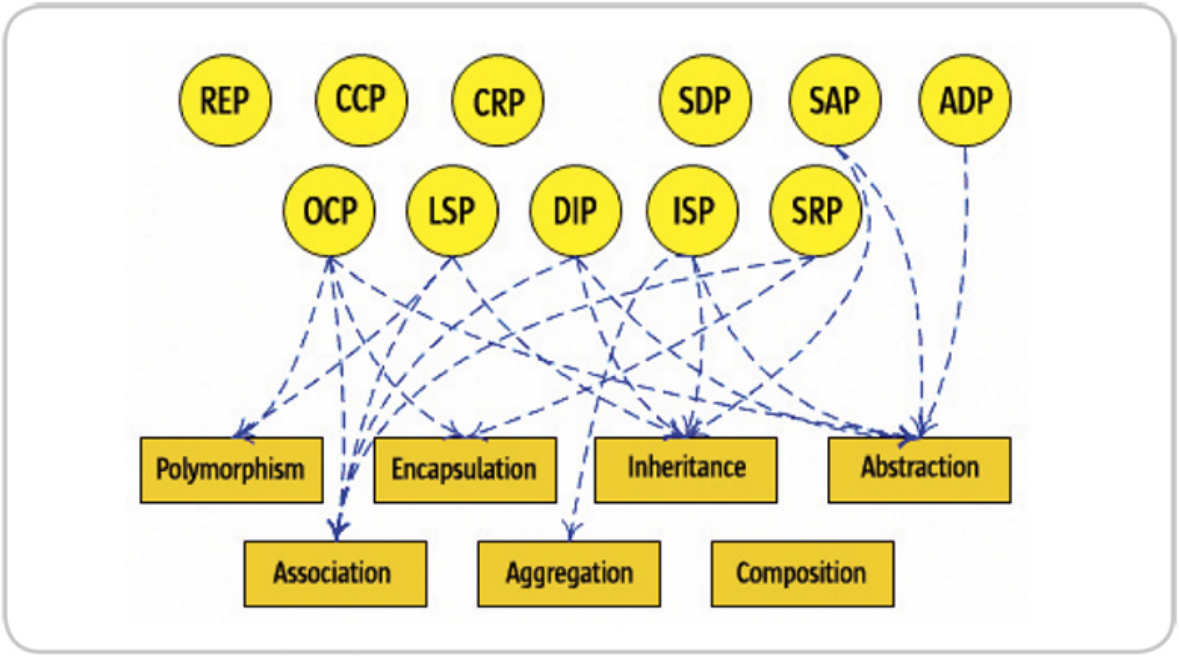

Figure 2 Dependencies between OO design principles and basic OO principles

As such, trying to draw the dependencies in such a graph is impossible. So a more appropriate methodology would be viewing these design pyramid levels and their relationships from the context of their application. Using this methodology, we can represent such relationships with this simile. If we take a car engine domain as an example, then we can think of the basic OO principles as the ones defining reciprocal-to-rotary motion conversion principles. This answers the "**why**" question and explains why the system works the way it works.

We can think of OO design principles and heuristics as answers to the "**what**" question because they unfold what to do to achieve design harmony. This is similar to the answer on what to do to build an internal-combustion engine.

At the top level of the OO design pyramid, we find the most effective approaches to solving generic and specific problems in certain contexts. We can think of this level as the one answering the "**where**" question. It's like picking the most efficient engine constructions for particular consumer requirements.

So different levels of the OO design pyramid tackle different aspects of OO design. Experience shows that one can't leave any of them out without running the risk of loosing something important. Applying the elements from the upper levels without aligning them to the lower levels can lead to design flaws, while applying elements from the lower levels without knowing the upper levels can increase design time (potentially at a lower quality), since designers would be forced to "reinvent the wheel" (see Figure 3).

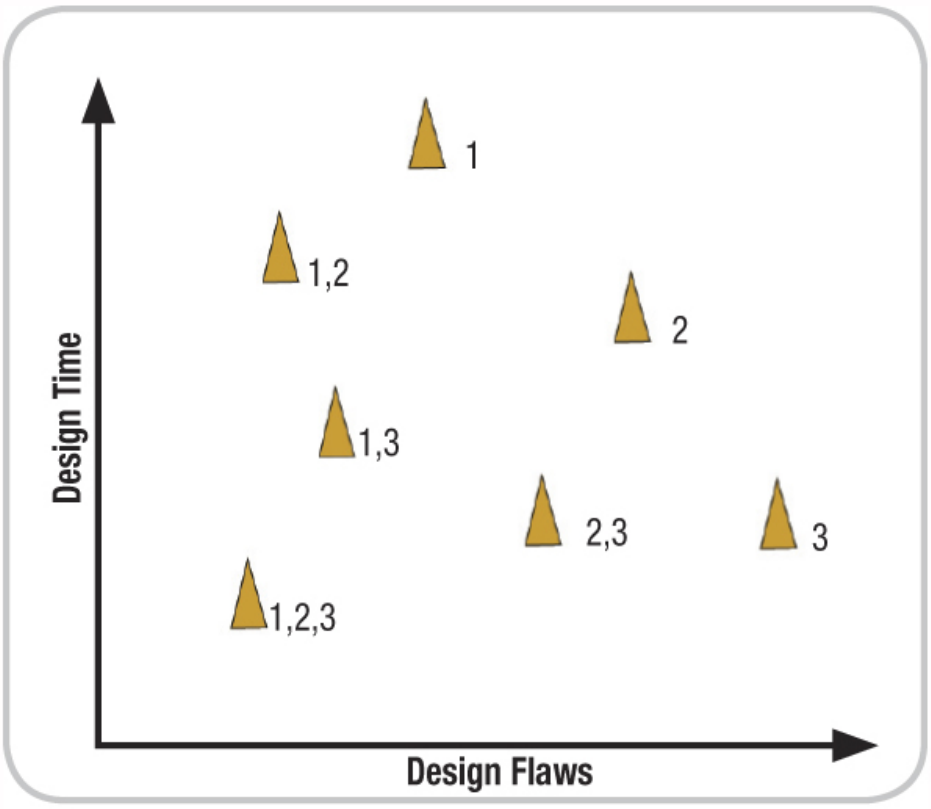

Figure 3 Design time vs quality (1 = basic OO principles, 2 = OO design principles, 3 = design patterns)

Conclusion

Because OO design elements focus on different aspects of OO design, coupled with their controversial and generic nature, knowing or even mastering only one or two levels of the OO design pyramid isn't enough to develop contemporary, robust, flexible, extensible, and stable software. OO designers and developers have to take all levels of the pyramid into account and apply them in a systematic manner.

Applying OO design levels in a top-down approach (from design patterns to design principles and heuristics to basic OO principles) saves design time and increases design quality by ensuring that none of the important dependencies are lost.

References

- ***Thinking in Java, Third Edition*** by Bruce Eckel, Prentice Hall PTR, 2002, ISBN: 0131002872
- ***Agile Software Development, Principles, Patterns, and Practices*** by Robert Cecil Martin, Prentice Hall, 2002, ISBN: 0135974445
- ***Object-Oriented Design Heuristics*** by Arthur J. Riel, Addison-Wesley Professional, 1996, ISBN: 020163385X
- ***Design Patterns*** by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Professional, 1995, ISBN: 0201633612
- ***Design Patterns Java Workbook*** by Steven John Metsker, Addison-Wesley Pub Co, 2002, ISBN: 0201743973
- ***Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*** by Deepak Alur, Dan Malks, John Crupi, Prentice Hall PTR, 2003, ISBN: 0131422464
- ***Patterns of Enterprise Application Architecture*** by Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford, Addison-Wesley Professional, 2002, ISBN: 0321127420
- www.objectsbydesign.com/tools/umltools_byCompany.html for a list of design tools.