OO Primer

(Object Oriented Principles Explained)

Arnon Rotem-Gal-Oz

[This whitepaper is based on a series of blog posts that first appeared in Dr. Dobb's Portal www.ddj.com/dept/architect]

Object Orientation has been with us for quite a few years. Yet it is relatively hard to find solid information on the basic OO principles. Sure everybody knows about the GoF patterns but there are a few basic principles that are easy to follow and can greatly enhance your designs.

The material in this paper is not new or original (except maybe JEDUF) it just an attempt to provide a primer to the works of people like Bertrand Meyer, Robert C. Martin, Barbara Liskov, Martin Fowler, Tom Demarco, David Parnas, and probably a few others I don't know since some of these principles are 20 years or more old.

The Principles

It's sad to say, but poor design is everywhere. Over the years I had a chance to review many system designs and it is not often that I see a design that is really brilliant.

Trying to think of why and how this happens, it seems at least some of the problem can be traced to how object orientation is taught. Programming courses usually focus on the syntax and mechanics of OO in general, and the programming language being taught in particular. Attending class, you hear a lot about objects, classes, state, methods, and constructors. Courses also teach you about inheritance, polymorphism, and encapsulation. Still, the focus is mostly on what it is and how to do it in *insert your favorite language here* programming language, but not on the motivations. Another problem is that courses usually talk about synthetic examples (shapes, pets, etc.), not real-world problems.

The same is true to "design courses." Often, these courses only teach UML syntax, not how to design or analyze. Or they stress design patterns and their implementation but not the motivations for the solution; for example, use cases is a functional decomposition technique. I've seen many developers "analyze" use cases and come up with a class model where the logic of what should be a single class is fragmented in per-use case classes.

This is a generalization, and I am sure there are some excellent courses out there. Moreover, I probably haven't seen enough courses to really pass judgment. However, I have had the chance to work with many developers and the "knowledge patterns" described above are recurrent.

Anyway, there is no point crying over spilled milk. Instead of just complaining, I thought I'll take the same path I took with the fallacies of distributed computing and cover some of the principles that do lead to better designs. Specifically this paper will touch:

Protected Variation/Open Closed Principle

Cohesion--Single Responsibility Principle
Interface Segregation principle
Dependency Inversion principle
Inversion Of Control
Design-by-Contract/Liskov substitution principle
YAGNI
JEDUF
DRY

7 Deadly Sins of Design

Before we delve into the "what to do" rules – lets take a quick look at the 7 deadly sins of design – the design problems we all want to avoid. Circumventing the principles that will be discussed later will, more often than not, result in committing one or more of these sins.

(The first four can be found in Robert C. Martin's paper Principles and Patterns*)

1. Rigidity. Make it hard to change, especially if changes might result in ripple effects or when you don't know what will happen when you make changes.

2. Fragility. Make it easy to break. Whenever you change something, something breaks.

3. Immobility. Make it hard to reuse. When something is coupled to everything it uses. When you try to take a piece of code (class etc.) it takes all of its dependencies with it.

4. Viscosity. Make it hard to do the right thing. There are usually several ways to work with a design. Viscosity happens when it is hard to work with the design the way the designer intended to. The results are tricks and workarounds that, many times, have unexpected outcomes (esp. if the design is also fragile).

5. Needless Complexity. Over design. When you overdo it; e.g. the "Swiss-Army knife" antipattern. A class that tries to anticipate every possible need. Another example is applying too many patterns to a simple problem etc.

6. Needless Repetition. The same code is scattered about which makes it error prone.

And in closing the list, the 7th Deadly Sin of Software Design is (the obvious):

7. Not doing any design

Protected Variation/Open Closed Principle

On one hand, the Open Closed Principle (OCP), defined by Bertrand Meyer nearly 20 years ago, means that classes or components should be open for extension and adaptation. On the other hand, OCP means they should be closed to avoid cascading of changes to existing clients of the code.

In his paper "Protected Variation: The Importance of Being Closed", Craig Larman says that OCP demonstrated the original intent behind the object-oriented concept of information hiding.

OCP is explained both in Larman's paper as well as by "The Open-Closed Principle" by Robert C. Martin. I won't attempt to compete with them in terms of explaining OCP's meaning, but I would like to highlight some points and issues in regard to OCP.

It is easy to see the benefit of having a class that answers this principle: When you need to add a requirement, instead of breaking dependent code (and tests) you just extend it somehow and everything is nice and dandy. Furthermore, violating OCP can result in Rigidity, Fragility, and Immobility.

But how do you do that? The obvious (and naïve) answer is inheritance. Every time something needs to change just add a sub-class. The parent class is not changes and voilà. However, if you add sub-classes all the time you'd get "lazy classes" or freeloaders--sub-classes without a real reason for existence not to mention a maintenance nightmare.

Thus, sub-classing is an option but we need to consider carefully where to apply it. Other (more practical) OCP preserving steps include:

- Marking class fields as private
- Not adding getters/setters **automatically**. You may want to read "Why getter and setter methods are evil" by Allen Holub, and Martin Fowler's rebuttal/refinement in "Getter Eradicator".
- Inversion of Control strategies
  - TemplateMethod pattern
  - Strategy Pattern
  - Closures (Ruby/Python) or Anonymous delegated (C#)
- Also don't forget to hide implementation details behind (stable) interfaces

To sum up, OCP is an important principle, Following this principle does results in better design. There are several practical and common steps we can take to help keep this principle and handle changes better.

Cohesion – The Single Responsibility Principle

The Single Responsibility Principle (SRP), also known as "high cohesion" is, like OCP mentioned above, a fundamental design principle.

The Encarta Dictionary defines cohesion as:

> Sticking or working together: the state or condition of joining or working together to form a united whole, or the tendency to do this.

The same holds true for classes. A class should be cohesive; that is, have only a single purpose to live and all its methods should work together to help achieve this goal. Or in Robert C. Martin's words:

> There should never be more than on reasons for a class to change.

When a class has more than one responsibility (that is, reason to change) these responsibilities are coupled. This makes the class more difficult to understand, more difficult to change, and more difficult to reuse (rigidity, immobility and needless complexity). Cohesion should also be applied at the method level--and for the exact same reasons.

You can also take a look at David Chelimsky's blog where he talks about not fragmenting that single responsibility between classes.

The challenge with SRP (and with David's extension of it) is getting the granularity of a responsibility right. For example, if I have a class that represents a business entity, should it also save itself to the database? Is handling all aspects of the customer including its persistency a single responsibility or is handling its business aspects one and persisting the second? In "Patterns of Enterprise Application Architecture" Martin Fowler et al. present the "ActiveRecord" pattern where a class wraps a database row (the first option described above). On the other hand, Alistair Cockburrn describes the Hexagonal architecture which talks about separating domain from transformation--persisting to a database (the second approach described above). I'll discuss additional aspects of this particular issue shortly when I examine architectural dilemmas.

Sometimes it is easier to see the responsibilities are unrelated. For instance, consider the same customer entity mentioned above. Should it also contain the logic needed to render it on the GUI? Here it is easier to see that the GUI representation should be handled in a different class. GUIs change a lot (relatively), and the entity may have several views on the UI (a grid, a specific form etc.). The entity and its representation may need to run on completely different tiers etc.

Sometimes the second responsibility may seem so trivial you may be tempted to leave it anyway. For example, creating a (thread safe(!)) singleton in C# (if you don't want lazy initialization) is as simple as:

```
sealed class Singleton
{
    private Singleton() {}
    public static readonly Singleton Instance = new Singleton();
    // other business related code goes here
}
```

Why bother separating the responsibility to have a single instance (singleton) from the business responsibility?

The reason is the coupling this induces on the consumers of the class which is very hard to remove. I recently had to do just that. I worked with a group that had a few singletons and code with dependencies on them. It was basically some application-level protocol layer (for communicating with external devices) the idea was to reuse it for a sister project that had to talk with the same devices. They said they had some problems reusing the code and asked me to take a look to see if I can help. I thought that since we're talking about the same devices, plus some of the code is generated it would be a breeze, maybe we'll do some refactoring and everything will be okay. It turned out we had to do a major redesign to uproot all the application specific singletons in to make the core code more mobile. You can read more on why singletons are bad on the C2.com Wiki and on Scott Densmore's blog.

One last point regarding SRP: If you cannot separate the responsibilities into separate classes, at least consider separating them to different interfaces. This is known as Interface Segregation Principle.

Interface Segregation Principle

ISP is a sort of the poor man's Single Responsibility Principle (SRP). While SRP addresses high cohesion in the class level, ISP is about high cohesion in the interface level. In other words, you create client (type) specific interfaces so that users will not have to depend on functionality they don't need.

ISP is a special case of a wider principle called "Separation of Concerns" in which each interface deals with a specific aspect of the behavior.

Here is a simple example for applying ISP: Consider a drawing program where you build a rectangle class that has some logic (e.g. calculate area) as well as drawing logic (e.g. render itself on the screen using DirectX). If you also have some algorithms that also need the area calculation of the triangle, the algorithms are now dependent on DirectX. When applying SRP this would be refactored into two classes, while with ISP it would be refactored into two interfaces and the algorithm would only depend on mathematical oriented one. Robert C. Martin described ISP in detail in Interface Segregation Principle.

ISP can be used as a solution where a class violates SRP from some reason (to help protect the class users). Another common use of this principle is as a second step for increasing cohesion and separation of unrelated parts. Interestingly determining what client specific interfaces you should have can be done using similar principles mentioned in a post I made on DDJ blog called Consumer Driven Contracts. There are several key differences though. Services granularity is very different from classes' granularity and more important ISP promoted fine-grained interfaces while contracts promote coarse-grained ones.

Dependency Inversion Principle

One of the simplest, yet most fundamental, object-oriented design principles is Dependency Inversion Principle - not to be confused with Dependency Injection or Inversion Of Control (see below).

The principle simply states that:

> Higher level modules should not depend on lower level modules. Both should depend on abstractions (interfaces or abstract classes).
>
> Abstractions should not depend on implementations.

The principle is simple since unlike some of the other principles it is structural and doesn't require a lot of thinking (vs. other principles like YAGNI, OCP, ISP, etc.).

The principle is fundamental as it is one of the main differences from procedural programming (if you remember what that is). In procedural programming, you had a program that depended on its modules which then depended on its functions. If a detail in a function changed, it could have severe ripple effects on all the hierarchy that depended on it. Applying DIP means that both the program and the classes depend on interfaces (and the whole interface-based programming idea). Other implications of DIP are things like layers and the afore mentioned Dependency Injection pattern.

Applying DIP helps avoid the above mentioned problems by increasing loose coupling. Abstract interfaces are usually more stable (compared with their implementation) so it is easier to update implementation without affecting the class consumers. This also helps increase testability (easier isolation) and decrease the rigidity of the design

Dependency Injection

While not a principle in itself, Dependency Injection is a very valuable technique to achieve Dependency Inversion – So it is worth spending a few paragraphs on it as well.

What's dependency injection (DI)? In a nutshell, DI means that a class does not instantiates any other classes. Instead, it depends on the classes that are "somehow" injected into it.

This is probably best explained with an example. (While this code is in C#, it is generally generic):

```
public class Controller
{
  IView view;

  //old way

  public Controller()

  {

  this.view = new View();

  }
```

In the above (very simplified) MVP example the controller creates the view. This means that the controller needs to have a reference to the specific view implementation.

Using DI the same class would look something like this:

```
  //new way - Using Inejection

  public Controller(IMyView tView)

  {

  this.view = tView;
```

```
   }

}
```

This time the controller expects some outside "assembler" class to hand it a view that adheres to the IMyView interface. The controller does not control the life cycle of the view nor is it aware of the specific implementation (instead it depends on an abstraction – the interface i.e. conforms to DIP). The example above is a constructor injection it is also possible to use a setter method (the dependency is injected after the class is instantiated). Yet another option is to use getter injection--but that requires using aspects to weave the dependency into the class. You can read a more thorough explanation of this at Martin Fowler's site.

Why do you need DI? For one thing, it helps introduce loose coupling--using DI, the Controller class above is only dependent on the **IMyView** interface and not on the view implementation. Using DI also means we can supply different views (provided they adhere to the IMyView interface) and the controller won't have to change.

Note that when you use DI, the injected class loosens control on the lifetime of the dependency (in the example above, the Controller no longer instantiate the view). If you need the object lifetime control you can inject an abstract factory instead of injecting the target class.

Furthermore DI makes classes more testable (which is why DI works so well with TDD). In the "old" example above, it is hard to have the Controller as the unit under test since we also need the view. If we use DI, the test fixture (or test method) can give the unit under test (the Controller in the example) a Test Double (dummy, fake, stub or mock) so that the unit under test can be as small as you like.

If you want to use DI, you don't have to build the infrastructure (factories, assemblers, etc.) to support DI by yourself. There are several frameworks that do this for you: spring and spring.net are the most famous, but there are few others like picocontainers, objectbuilder, and so on.

Inversion of Control

Another OO principle is "Inversion of Control" (IoC). While not as basic as some of the others (such as the Single Responsibility Principle or Liskov Substitution Principle), IoC has been in wide use as techniques that implement it (Dependency Injection, for instance) become more prevalent.

IoC means that an object that wants some functionality from another object (let's call it "provider") gets called by the provider; rather than the usual flow of logic where the object calls the provider when it needs it (hence the "inversion" of the control since it is the provider that controls who gets called where). Say, for example, that you are writing a GPS-based navigation program and you want to draw a car symbol on a map to show the current position. One way to do that would be that every time the car position changes, you call the map object and tell it to draw the car at that new position. Another way to do it would be for the map object to call the car object whenever it is convenient for the map (x times a second, for instance) and ask for its position. This second option uses Inversion of Control.

Why do this? In the scenario above, one motivation might be the load. The GPS might update its position tens of times per second, but for a smooth movement on the map you only need to update it 5-10 times a second. In the more general case, it is just simpler to get the functionality you want if you use IoC. Again, for the example above, instead of understanding the ins-and-outs of the map object, you just implement an interface to return the symbol of the car and retrieve the position, and then let the map object worry about the rest.

This is why IoC is a popular technique used in frameworks (see also Framework vs. Libraries), or as Martin Fowler said, it is the almost the defining characteristic of frameworks.

Some people confuse IoC with the Dependency Inversion Principle; for example, see the IoC definition in Wikipedia. However, they are quite different. DIP talks about type dependency (classes should depend on abstractions) while IoC talks about flow of logic.

Another misconception about IoC is that it is equal to Dependency Injection; see the IoC definition in HiveMind, an IoC-based container. I think that the source of this (and the previous) misconception comes from Dependency Injection enabling containers like Spring or Hivemind.

DI containers utilize IoC. The container, for example, controls the lifecycle of the objects involved. Another example is that the container also performs the injection on its discretion. The end result (of the dependency injection) is that the two (or more) objects only have a dependency on interface (that is, the object that got the dependency injected into it only knows the "dependency", the other object, by its interface). The end result is an implementation of the Dependency Inversion Principle (only depend on abstractions). To top that, the words "injection" and "dependency" occur too much between those three terms "**Dependency** Injection," "**Inversion** of Control", and "**Dependency Inversion** Principle".

You may also want to read Martin Fowler's article "Inversion of Control Containers and the Dependency Injection Pattern" which explains the difference in more detail.

Design-by-Contract/Liskov substitution principle

Liskov Substitution Principle (LSP) is yet another object-oriented principle. However, unlike the Dependency Inversion Principle I recently discussed it is much more subtle. Still, it is just as fundamental.

LSP was defined back in 1988 by Dr. Barbara Liskov. The original definition is a little hard to follow, but goes like this:

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

In plain English this means that any subclass of a class should always be usable instead of its parent class which makes Subclassing a **public** behavior preserving operation. Another way to look at this principle is known as design by contract:

A sub class should Honor the contracts made by it parent classes

Pre-conditions can only get weaker

Post-conditions can only get stronger

The classic examples for violation of this principle are an ellipse that inherits a circle or a square that inherits a rectangle (hint: the rectangle has a setWidth and SetHight methods). I'll try to show that by an example of how Microsoft violates LSP within .NET class hierarchy

.NET (1.1) has ContextBoundObject (which inherits from a MarshalByRef object). A nice feature of ContextBoundObjects is that .NET is that the .NET CLR provides a transparent proxy to class derived from it, intercepts the call and even lets you extend its behavior by providing custom sinks and adding them to the chain called by the interceptor. (This feature has been used to create simple AOP solutions in C# (you can see an example here. Note that using this for AOP not recommended by Microsoft). So far so good. Another useful class in .NET is ServicedComponent (your primary and almost single gateway to COM+ services in C# (before COM+ 1.5 Services without components or System.Transactions)). As it happens ServicedComponent is a subclass of ContextBoundObject, indeed Microsoft uses the proxy and sinks to add the different COM+ services to objects that inherit from ServicedComponent. The LSP violation is that while ContextBoundObjects lets developers add their own sinks ServicedComponents do not... I once sat for a day trying to figure out what went wrong in a piece of code that worked perfectly when we used "plain" ContextBoundObjects and suddenly failed miserably with Enterprise Services.

Violations of LSP results in all kind of mess--can get superclass unit test to fail, a lot of unexpected or strange behavior (e.g. the example above) and also a lot of OCP violations--as you have if-then statements to resolve the correct subclass. LSP is the reason that it is hard to design and create good deep hierarchies of sub classes and the reason to consider using composition over inheritance. It is sometimes hard to see the problems that will be caused by violating LSP - and you may be tempted to think everything would be okay. But it will come back to haunt you and since you need to dig in into the internal working of the specific subclass it is hard to track these problems.

JEDUF

Agilists tend to down play traditionalists reliance on Big Design Up Front (BDUF) and they are right. More often than not, Big Designs without underlying code that demonstrate it actually works and integrates well prove to be a waste of time.

However, that does not mean not to do any design up front--as indeed several people have articulated this (see, for example, Lidor Wyssocky, Keshav, Robert Watkins, Jonathan Kohl and others).

Now some projects allow for no design up front (for example, see Bowling example by Robert C. Martin and Bob Koss). However, I think many projects will greatly benefit from some design up front. How much design you ask? Ah, that's easy--just enough. Hence, Just Enough Design Up Front (JEDUF).

Of course, at the application level it depends. If you are building a safety critical DO-178B certified project, that can mean quite a lot. Nevertheless, normal IT projects don't need that much up front design. You should probably spend some time thinking about the system's quality attributes. Analyze them until you can think about them as scenarios in your application; for instance, a performance requirement can be written as "Under normal operation, perform a database transaction in 100 milliseconds" or "Under normal or stress conditions, a critical alert generated by the system will be displayed to the user in less than 1 second." The nice thing about expressing quality attributes as scenarios is that it is relatively easy to code a proof-of-concept (or an XP Spike) to check options for actually fulfilling them. The Up Front design occurs when you try to prioritize, integrate, and balance the (sometimes conflicting) requirements of multiple quality attributes to create an integrated architecture that fits your project.

When it comes to class level design, I agree with Lidor Wyssocky. I think you should be pragmatic and apply common sense (well, if you are just starting out with TDD, you probably want to be a little more dogmatic about it so that you would acquire the habit). Also if you solved a similar problem in the previous iteration or project you can apply a similar solution. The same can be true at the package or module levels.

The point is that is value in design and there is no need to tackle every problem as if it is the first time you see it. You can save time by relying on your past experience.

YAGNI

Since some requirements are bound to change over time, Big Design Up Front (BDUF) can end up being a waste of time.

Another problem with BDUF is that it can be overly complicated (mentioned in my "7 Deadly Sins of Design") and/or "goldplating". The balancing force on the road to Just Enough Design Up Front is the "You Ain't Gonna Need It" (YAGNI) principle.

the eXtreme Programming version of YAGNI is:

> **Always** implement things when you **actually** need them, **never** when you just **foresee** that you need them.

I have to say that my view is a little different than that - I think (as evident from the JEDUF principle above) that there are things that you **know** you going to need and that are **hard to add** later on (for example, if you design object for local use and then later you suddenly decide they should reside on separate tiers). However, these thing are relatively few and it is well worth your time thinking whether you know you going to need something or you just foresee it and if the thing being designed (or coded) is something that would be easy to add later or not. When something can be added later just as easily it **is** better to postpone it unless it is needed right now.

The result of applying YAGNI is simpler designs which really focus on the problem at hand without "future proofing"

DRY

The last OO principle I am going to discuss in this paper is "Don't Repeat Yourself" (or "DRY" for short). I've also seen it referenced as "Single Point Control".

The principle simply states that:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Actually saying that this is an "object-oriented principle" does a disservice to this principle because it is a fundamental principle for software engineering. For example, consider:

> Database normalization rules. Redesigning tables so that data will only be defined once.
> Refraining from magic numbers. Adding constants, variables.
> Parameterizing algorithms. Instead of rewriting them for new sets of requests.
> Writing functions instead of repetitive code.
> Even go-to statements.
> etc.

For object orientation DRY means things like using inheritance, template methods etc. DRY is also related to several refactorings like extract class, extract method, extract supereclass, etc.

The most obvious benefit of DRY is increasing maintainability and testability--you have one place to go when you need to change something and you only have to test once for all. if you fail to apply DRY a maintainer/tester will have to repeat their efforts as well (plus the overhead of understanding what is it exactly that happening and why the results are different).

The pitfall of DRY is trying to generalize too much which can result in over complication (remember YAGNI mentioned above).

Summary

This paper covers few of the basic principles of good OO design. You need to remember that real life applications usually need to handle issues that are not OO in nature (like distribution or concurrency) which complicate things. Another thing that you need to remember is that the principles are just that - principles. These are not commandments that you must follow although following them will help you avoid the "design sins" (read design problems). The last thing you should keep in mind is that on a given context some principles cannot all be applied together (e.g. making sure OCP is met you will need to make the design more complex (and potentially violate YAGNI) this is OK – your job is to balance the principles and determine which ones are more important in the situation (hell if the job was simple they would have given it to the janitor and not you…☺)

I hope you find this paper useful and that it helps you sort out the different principles and their benefits.