ArticleS. UncleBob.

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.*

Wed, 11 May 2005 20:27:08, Shane, Buy the Book! Expand All | Collapse All

I would encourage anyone reading and thinking "I need to know more about this stuff" to buy the book that Bob wrote and refers to (PPP).

Thu, 12 May 2005 10:37:50, Henrik Huttunen, Some praise Expand All | Collapse All

These principles are taught at our university in a course, where we take second big step toward OO-programming i.e. learning to program with design patterns. I consider the first five to be very important for to get deeper understanding of programming. They've changed my thinking somewhat, and it's nice to check your solution against those principles and see what they might reveal.
Also, I have read many articles of yours, and I like them very much. The way you present the problems and different methods to handle them, are clear and profound. And you have made good points about why to write tests before any code; it surely makes programming less painful, when need of debuggin is decreased alot.

It's a shame you haven't lately written large articles -- at least don't know any. But at least this blog and newgroup discussing are active.

Sincerely, Henrik

Thu, 12 May 2005 11:41:21, Uncle Bob, Writing Large Articles. Expand All | Collapse All

Henrik,

I have a regular column in Software Development magazine. It's called "The Crafstman". In it I write a lot about TDD, Principles, Patterns, and life on a starship. You can see a list of all these article at: http://www.objectmentor.com/resources/listArticles?key=topic&topic=Craftsman

I also write feature articles for this magazine from time to time, and for other magazines as well. You can keep track of them, and all the articles the Object Mentors write at http://www.objectmentor.com/resources/articleIndex

Thu, 12 May 2005 12:00:20, Henrik Huttunen, Articles Expand All | Collapse All

Robert,

yes thank you, I'm aware of those. I just wondered you hadn't done any article this year, that's all.
Btw. I'm buying a new book about refactoring/TDD/design patterns in general, and would like to hear recommendations. Refactoring to Patterns by Joshua Kerievsky is often recommended, but someone said it has too application specific examples. If someone has read it, I'd like to hear comment on that.

Sun, 15 May 2005 05:13:44, Mauro Marinilli, Praise again and a hint for Henrik Expand All | Collapse All

I use to read so much about OOP, but your advice always sounds proven and deep. It's something else!
Keep up your good, honest work.

OOP is too flexible. You can use it for everything and the opposite of everything. That's why it got so popular, perhaps. Principles then are always controverse, though. What happened for example to the fundamentalists of encapsulation (those that refuse to design or even use setter/getters)? Also structured programming isn't a silver bullet, and exaggerating with it make things worse (at least this is what I try to teach to my students.. Shame on me!). That's why your advice is so important: because it is empirically grounded.

Henrik: as of Refactoring to Patterns, I'd suggest you to have a look at an earlier draft (http://scholar.google.com/url?sa=U&q=http://www.tarrani.net/RefactoringToPatterns.pdf) that contains half of the patterns but gives you a feeling of the overall approach (sorry if I use this space to reply to another reply).

Mon, 16 May 2005 07:32:54, Henrik Huttunen, Refactoring to Patterns Expand All | Collapse All

Mauro, thanks for the link :).

Fri, 20 May 2005 19:30:44, Mark Dalrymple, Refactoring to Patterns Expand All | Collapse All

Refactoring to Patterns is awesome. The examples do deal with real domain-specific code, but the principles are applicable to other domains.

Fri, 20 May 2005 23:15:19, JDCarroll, Almost right Expand All | Collapse All

You begin by talking about OOD and by the end of the first paragraph have made the most critical of errors: equating OOD with OOP. OOA/D is about THINKING. OOP is about DOING. The two are separate.

- *It took me over half a decade to realize that this wasn't true. OOP and OOD are inseparable. OOA is undefined.* **- UB**

And the only criteria by which a OO programming language should be judged is the ability to move from one to the other.
Then you move into the notion of Dependency Management as if the way that we think about and write the programs will make it right. Sorry. But whether you live in a Structured world or an Object world the notion of dependancies doesn't change. Folks write code that you depend on. You write code that other folks depend on.

Until that changes, we're stuck.

- *No, we aren't stuck. Dependencies are manageable.* **- UB**

Tue, 7 Jun 2005 02:46:09, cheng jing, when and how add database Expand All | Collapse All

I read your wonderful book < agile software development >, At chapter 19, u implement the payroll system.
Now I design a true PayRoll[?] system, but i don't known when and how can i add database detail ? i can add UI first then meet the client's requirement ?

- I recommend that you concentrate on the business rules first. Don't worry about the UI or the Database at first. Get the business rules (taxes, deductions, etc) to work. Then you can add a database. And finally you can add a UI. - **UB**

Sun, 12 Jun 2005 04:31:44, Brad Appleton, Where does The Law of Demeter fit? Expand All | Collapse All

Hi Bob!
Granted, the "Law of Demeter" is a style guideline rather than a principle. Nonetheless, I would expect it to be something that could be readily derived from a handful of these principles. Do you agree? if so, how would you derive it?

- LOD is a matter of dependencies. A statement like system.trunk.line.lineCard.connect() concentrates too much information into a single place. That one line of code knows about four classes! Large knots of dependencies like this are a violation of the OCP. Any change to the data structures causes changes to that line of code.

Mon, 13 Jun 2005 02:35:41, , Expand All | Collapse All

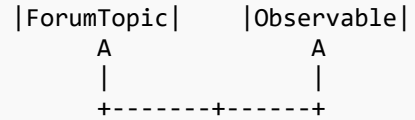
Fri, 17 Jun 2005 07:27:50, Denis Krukovsky, SRP with Observable model? Expand All | Collapse All

Can we start a discussion here? Let's say we have a Model of some entity, say a ForumTopic[?]. We give it the responsibility to represent a forum topic. Now a classic case - we want our Models to be Observable. So we add a responsibility to notify its Observers on state change. SRP violation.

How to make ForumTopic[?] respect SRP while apply Observer pattern here?

Denis Krukovsky
http://dotuseful.sourceforge.net/
http://dkrukovsky.blogspot.com/

- You could use the *class form* of the *Adapter Pattern* (i.e. Multiple Inheritance) as follows:



```
                |
        |OservableForumTopic|
```

- **UB**

Mon, 20 Jun 2005 19:59:43, Don, ISP and accessing multiple non-cohesive interfaces Expand All | Collapse All

Thanks for your Agile Software Development book - a great resource. I have a question concerning ISP. The second paragraph explicitly states that some objects require non-cohesive interfaces. So I have an object O with two non-cohesive interfaces A and B. My question is this - when a client has access to O as an instance of A, and then at some point needs access to O as an instance of B, how is that managed? My instinct tells me to use a factory for the conversion so that the knowledge that A and B are coresident in O is isolated (allowing adapters and other alternatives) but I am not confident. Any advice welcome. Thanks again.

Sat, 9 Jul 2005 00:40:56, Jay Levitt, Principles same w/dynamic languages? Expand All | Collapse All

I'm reading through PPP tonight, and it's a fascinating book. But the examples you use to illustrate design smells seem to be dependent on the use of a statically-typed language like C++, with solutions involving virtual base classes and/or MI.

I'm playing around with Ruby, which has none of the above, and at first blush it seems to me that a lot of the "binding" issues simply don't exist there; LSP is inherently enforced by "duck-typing", and classes can be extended repeatedly in different source files, providing ISP. As such, SRP and OCP just don't seem to matter. Have there been any discussions or articles about how these principles apply to dynamic languages?

Fri, 30 Sep 2005 08:03:26, Brijesh, I finished reading ur book PPP Expand All | Collapse All

hi uncle bob,
I am great fan of yours, i completed reading the book PPP , after reading that book, the way I look at the solution for a given problem is changed , I mean my thought process is changed when i start thinking about the design, thanks a lot for ur marvellous work on this area.

thanks Brijesh

Tue, 4 Oct 2005 21:39:38, JeanW[?], Responsibility Expand All | Collapse All

The book is great. I do have some deeper questions. The one I have been struggling with lately is defining a responsibility. You write, "In the context of the SRP, we define a responsibility to be 'a reason for change.'"

However, the word "responsibility" suggests that the root meaning has to do with managing some sort of feature, and that change is just a *symptom* of managing that feature. Other responsibilities lie undiscovered until a change exposes them. But I've had trouble defining what it is.

I mean, if all you wanted to convey was "a reason for change," then you could have called it the "Single Reason to Change" principle. So why did you use the word "responsibility"? And is there a deeper definition than "a reason for change"?

Thu, 6 Oct 2005 09:41:53, Uncle Bob, The Single Reason to Change Principle Expand All | Collapse All

The principle comes from the work of Tom DeMarco[?]. I don't know if he's responsible for the name or whether the name is an accident of some kind. What I can tell you is that the definition of the principle has evolved over the last thirty years. It began as:

- A module should do one thing, do it well, and do it only.

And it has finally transformed into its current form:

- A module should have one and only one reason to change.

Are these two phrases really synonyms? I think so. I think a reasponsibility boils down to a reason to change. For example, consider a payroll module that pays an employee. We could say that the responsibility is "pay employee". However, over time the payment calculation rules change, even though the printed format of the paycheck does not. Later, the format of the check changes but the calculation doesn't. Clearly there are two different responsibilities. Calculation, and format.

Mon, 3 Apr 2006 01:58:28, Amir Khan, Liskov principle related *Expand All* | *Collapse All*

Hi Uncle Bob,

I read with great interest the articles written by you. In fact we cover one technical article every monday morning. Today we did Liskov's Substitution Principle. Please forgive me if my question sound too naive. You talked about Factoring as being one way of making classes LSP compliant. For exampe you created LinearObject[?] and sub-classed Line and LineSegment[?] from it. I was wondering what if the derived classes have some methods that cannot be foctored out to a base class. For example if I factor Circle, Rectangle, and Square to make the Shape class and put the Draw method in it. What if I also need to rotate Rectangle and Square but not the Circle. So what would I do? Should I put the Rotate method in the base Shape class and write a degenerative Rotate method in Circle and solid implementations in Rectangle and Square?

In general what should I do if my to-be-factored classes have some common behavior yet also have some unique behavior.

I shall look forward to hear from you.

Regards

Amir Khan
Karachi, Pakistan

Front Page | User Guide
root (for global !path's, *etc.*)
SetUp[?]TearDown[?]