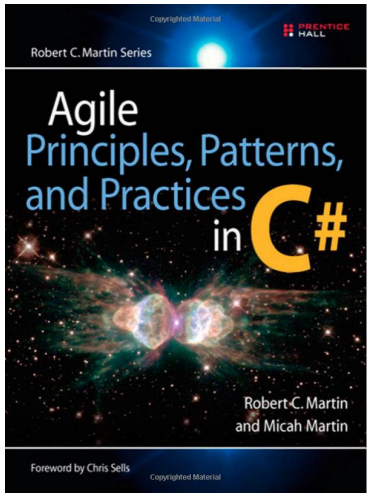
Principles of Package & Component Design

Saturday, January 4, 2020 7:52 PM

Book Review: Agile Principles, Patterns, and Practices in C#
Publicerat avEric Bäckhage mars 23, 2019

I have been reading Robert C. Martin's and Micah Martin's book, Agile Principles, Patterns, and Practices in C#, for quite a while now. The reason it has taken me so long to finish is that it is packed with so much information and covers so many aspects of software development, enough for at least 3-4 different books of their own.

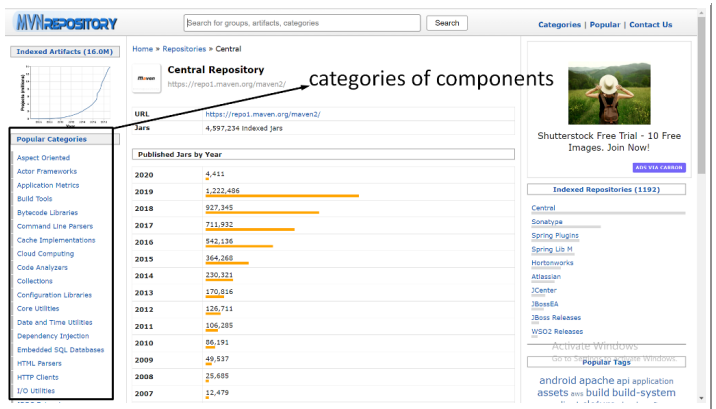


Book division In 4 sections:

1. Section on Agile Development which covers topics such as Agile Practices, Extreme Programming, Planning, Testing, and Refactoring.
2. A section on Agile Design where the famous SOLID principles are covered and also covers UML and how to effectively work with diagrams.
3. In the third section a number of design patterns are introduced and the practices learnt so far are put into practice in a case study.
4. The fourth and final section covers Principles of Package and Component Design (REP, CRP, CCP, ADP, SDP, and SAP) and introduces several other design patterns.

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs to be:

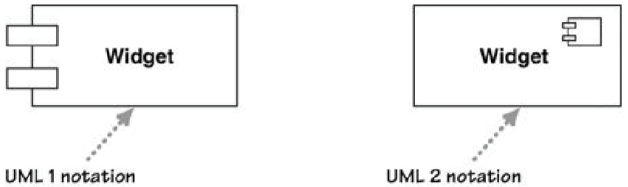
- fully documented
- thoroughly tested
- robust - with comprehensive input-validity checking
- able to pass back appropriate error messages or return codes
- designed with an awareness that it *will* be put to unforeseen uses
- **If the SOLID principles tell us how to arrange the bricks into walls and rooms, then the component principles tell us how to arrange the rooms into buildings. Large software systems, like large buildings, are built out of smaller components.**



Component Diagrams

A debate that's always ranged large in the OO community is what the difference is between a component and any regular class. This is not a debate that I want to settle here, but I can show you the notation the UML uses to distinguish between them.

UML 1 had a distinctive symbol for a component (Figure 14.1). UML 2 removed that icon but allows you to annotate a class box with a similar-looking icon. Alternatively, you can use the «component» keyword



As I've already said, the issue of what is a component is the subject of endless debate. One of the more helpful statements I've found is this:

> Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy.

Ralph Johnson, http://www.c2.com/cgi/wiki?DoComponentsExist

The important point is that components represent pieces that are independently purchasable and upgradeable. As a result, dividing a system into components is as much a marketing decision as it is a technical decision, for which [Hohmann] is an excellent guide. It's also a reminder to beware of overly fine-grained components, because too many components are hard to manage, especially when versioning rears its ugly head, hence "DLL hell."

When to Use Component Diagrams

Use component diagrams when you are dividing your system into components and want to show their interrelationships through **interfaces** or the breakdown of components into a lower-level structure.

Book cover

The book begins with a section on Agile Development which covers topics such as Agile Practices, Extreme Programming, Planning, Testing, and Refactoring. It continues with a section on Agile Design where the famous SOLID principles are covered and also covers UML and how to effectively work with diagrams. In the third section a number of design patterns are introduced and the practices learnt so far are put into practice in a case study. Finally, the fourth and final section covers Principles of Package and Component Design (REP, CRP, CCP, ADP, SDP, and SAP) and introduces several other design patterns. It ends with a lot of code examples where database (SQL) support and a user interface is added to the application introduced in section three.

Even though the book is over 10 years old it is still highly relevant. Agile software development, good practices and principles, and patterns for OOP, are skills that all software developers today will benefit from educating themselves on. There are tons of online material, classes, and other books that covers these topics, but I don't know of any other resource that have all of it in the same place.

With that said, I highly recommend this book. But to get the most of it you need to be prepared to put a lot of time and focus on reading it and really understanding the reasoning behind the principles and patterns. Personally I had to re-read some sections and take notes while I was reading, or I felt like I didn't get all the details. It might be helpful to buy some copies to your workplace and run it as a book circle so that you get to discuss the contents with other developers.

Verdict: Highly recommended!

Principles of Package and Component Design
Publicerat avEric Bäckhage mars 3, 2019

One of the most known set of principles regarding software design is probably SOLID. But an important part of structuring software that **SOLID** does not cover is how to group classes into packages and **components** in a way that makes it scale, both when the application itself grows but also when the number of **teams** and **developers** working with the code grows.

Fortunately this is also something that Uncle Bob has thought of and written down a number of principles on. In the next set of blog posts I will cover these principles briefly, but I do encourage you to buy a copy of either "Agile Principles, Patterns, and Practices in C#" or "Clean Architecture" so that you can really read up on them and look at a larger example on how they can be applied.

Uncle Bob has written down six different principles on package and component design. Three of them targets package cohesion, and the other three governs package coupling. They are called:

☑ The Reuse/Release Equivalence Principle (REP) ☑ The Common Reuse Principle (CRP) ☑ The Common Closure Principle (CCP)	**Package Cohesion**
☑ The Acyclic Dependencies Principle (ADP) ☑ The Stable-Dependencies Principle (SDP) ☑ The Stable-Abstractions Principle (SAP)	**Package Coupling**

I will write about them in that order, starting with REP and ending with SAP.

Oh, and by the way. In .NET a component is called an **Assembly** and is usually carried in a DLL.

The Reuse/Release Equivalence Principle

Publicerat av Eric Bäckhage mars 3, 2019

What?

When we group classes into components we strive towards making some of them reusable so that we can potentially use them for other purposes and also other teams in the organization may use them if they want to. The Reuse/Release Equivalence Principle (REP) states that ”The granule of reuse is the granule of release”. In other words, if a component should be considered reusable it must be a releasable unit.

Why?

The reasons for why we should group classes into releasable components are mostly political. It has to do with the expectations other people have on the components and the support they require if they are going to reuse the code.

The users will expect the components to have version numbers, good documentation and support, etc

How?

When grouping classes into components, reuse should be considered. Either all the classes in a component are reusable, or none of them are. Do not put classes that should be reused and classes that should not be reused into the same component.

The reusable components should also ”target the same audience” i.e. they should have a common theme or use case. Think of it this way: If the component is reused, all of the classes in the component should be reused. Do not put classes that have nothing in common into the same reusable component.

The Common Closure Principle
Publicerat avEric Bäckhage mars 6, 2019

What?

The Common Closure Principle (CCP) states: *”The classes in a component should be closed together against the same kind of changes. A change that affects a component affects all the classes in that component and no other components.”*

To put it in other words, a component should not have multiple reasons to change. This is the Single Responsibility Principle (SRP) from SOLID restated for components.

Why?

The goal of CCP is to increase maintainability. If a change to requirements always triggers changes in a lot of the components it will be harder to make the change, each change will lead to a lot of re-validation and redeploying, and parallelizing different changes will be harder.

You would prefer the changes to occur in one component only.

How?

As with SRP, full closure is not possible. There will always be changes that requires multiple components to be changed. The aim should be to be strategic and place classes that we, from experience, know often changes together into the same component.

The classes don’t have to be physically connected (i.e. are connected through code) but can also be conceptually connected.

The Common Reuse Principle
Publicerat avEric Bäckhage mars 4, 2019

What?

The Common Reuse Principle (CRP) states: *”The classes in a component are reused together. If you reuse one of the classes in a component, you reuse them all.”*

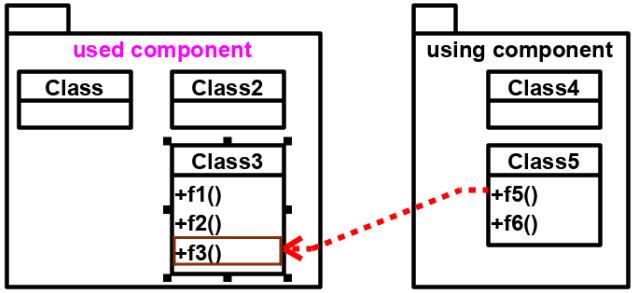
In the last post I wrote about the Reuse/Release Equivalence Principle (REP). It says that reusable components must be releasable components, with everything that comes with making it so. I finished with

writing that classes should have a common theme or use case.

CRP emphasizes this, by saying that the classes that make up a component should be inseparable. It should be hard or impossible to reuse only some of the classes of a component. The using component should be (directly or indirectly) be using all of them.

Why?

Components in .NET are made from assemblies carried in a DLL. If one component uses a class in another component, even if it only uses a single method in a single class, it has a strong dependency to that component.



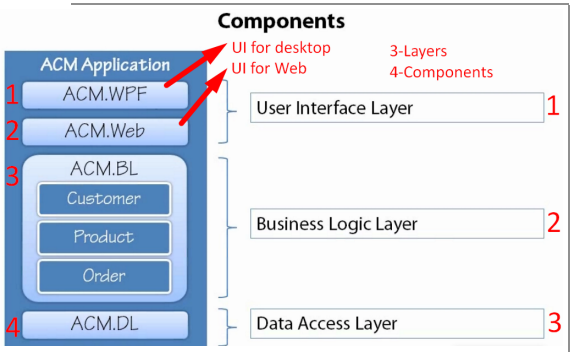
Using component **class5** only uses f3() and does not uses f1() and f2() from Used Component in spite of this anytime any changes made in f1() or f2() or in both will also affect the using component. So Using component has a strong dependency.

This means that every time someone makes a change to used component, even in code that the using component do not use and don't care about, **a new version of the used DLL is created and the using component must be re-validated against that new version.**

This is a waste of resources and effort and should be avoided.

How?

CRP says something about which classes to put in a component, but mostly it says which classes to **not** put in the component. The classes that make up a component should be tightly coupled and depend heavily on each other. It should not be possible to separate them easily. If a class does not fit into this description it does not belong in the component and should be put elsewhere.



When your find yourself creating general purpose functions which does not belongs to any of these components consider putting them in general purpose class. And if the resulting class is reusable throughout the current application & possibly other projects or applications you may build, consider creating a **reusable** library/package of general purpose classes. Code Libraries are convenient ways to package & reuse a set of classes. Most applications makes use of some common functionality. Maybe you have a common set of code for:

- **Logging**
- Or a **specialized set of data handling** routines/functions you reuse.
- Or you want to start to build up some **general purpose string handling routines/functions**.
- Many other scenarios

Such general purpose classes & methods are often **static** in nature, static classes & functions provides a shortcut to the operations or functionality in a class when creating object is unwanted.

- One of the key benefits of creating an **instance** is that it can manage the data also called state for a particular object created from a class. Obj1, obj2, obj3 all these objects have different state.

The Acyclic Dependencies Principle
Publicerat avEric Bäckhage mars 9, 2019

What?

The Acyclic Dependencies Principle (ADP) is the first of three principles that deals with the relationships between components.

It says: *"Allow no cycles in the component dependency graph."*

If you draw the components and the dependencies between them and you are able to follow a dependency back to a component you have already visited, then it is a violation to ADP.

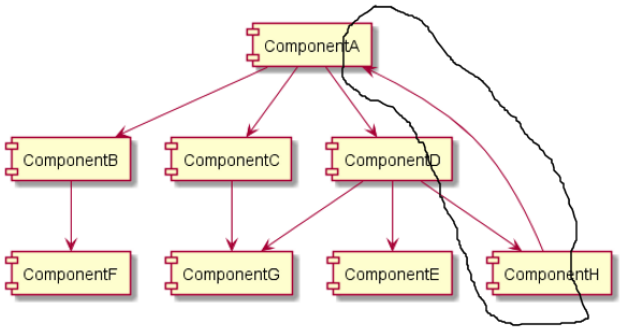


Figure 1. Violating ADP

Why?

So why should you care about cyclic dependencies? Because their existence makes it very hard to split up responsibilities and work on different tasks in parallel without stepping on each other toes all the time. Take a closer look at ComponentH in Figure 1 above. The dependency to ComponentA makes it depend on every other component in the system. Since ComponentD depends on ComponentH, it also makes ComponentD depend on every other component in the system.

This makes it really hard to work with and release components D and H. But what would happen if we could remove the dependency from ComponentH to ComponentA? Then it would be possible to work with and release ComponentH in isolation. The person, or team, working with ComponentD could lift in new releases of ComponentH when they needed, and wait with taking in the new release if they had other more prioritized tasks to solve before moving to the new release.

How?

How can we break a dependency such as the dependency from ComponentH to ComponentA? There are two options we can use:

1. Apply the Dependency-Inversion Principle (the D in SOLID) and create an abstract base class or interface in ComponentH and inherit it in ComponentA. This will invert the dependency so that ComponentA depends on ComponentH instead.
2. Create a new component – let's call it ComponentI – that both ComponentA and ComponentH depends on. See figure 2 below.

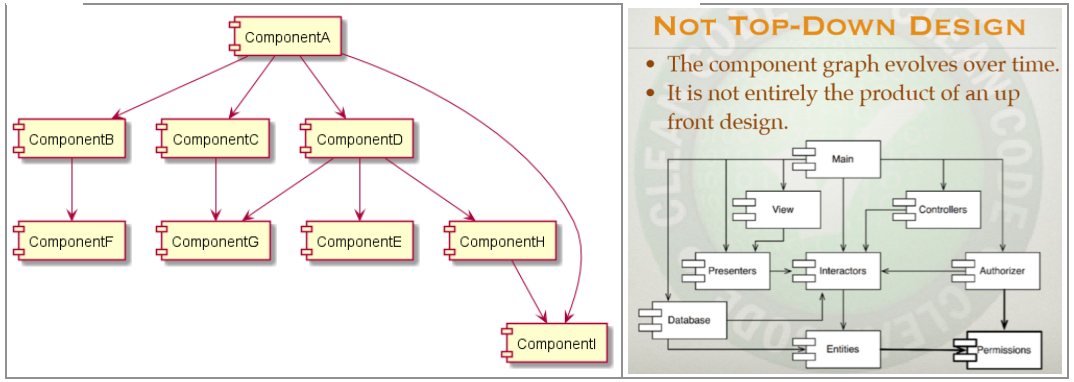


Figure 2. Conforming to ADP

It is easy to introduce cyclic dependencies. You need to keep track of the dependencies in your system and break cycles when they appear.

What?

The Stable-Dependencies Principle (SDP) says: *"Depend in the direction of stability."*

What this means is that the direction of the dependencies in the component dependency graph should point towards more stable components. To understand what this means we need to define *stability* for a component. Uncle Bob turns this around and defines a way to measure *instability*, *I,* using the formula:

I = Ce / (Ca + Ce)

where *Ca* is the number of classes outside this component that depend on classes within this component (afferent couplings) and *Ce* is the number of classes inside this component that depends on classes outside this component (efferent couplings). *I* has the range [0,1] where *I = 0* indicates a maximally stable component and *I = 1* indicates a maximally instable component.

A component that does not depend on any other component is maximally stable and a component that only depend on other components but have no components depending on it is maximally instable.

Why?

When we design our components there will be some components that we want to be easy to change. They contain classes that often needs to be modified when requirements change. If these components have many dependencies to them (are very stable; have a value *I* close to 0) they will be difficult to change since changing them will impact many other components.

Figure 1 shows a component diagram where SDP is violated.
The *Stable* component, that has an instability value of *I = 0.25* has a dependency to the *Volatile* component, that has an instability value of *I = 0.75.* This dependency makes the volatile component difficult to change.

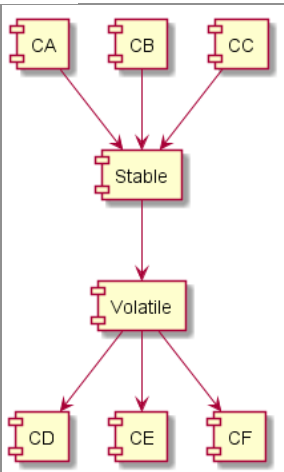


Figure 1. Violating SDP

How?

So how do we fix violations to SDP? The answer is, once again, inverting dependencies by applying the Dependency Inversion Principle (DIP). Look at Figure 1, to fix the violation we can create an interface or abstract base class and put in a new component, CG. We can then let the Stable component depend on CG and have the Volatile component implement the concrete classes. By doing that we end up with the dependency graph showed in Figure 2.

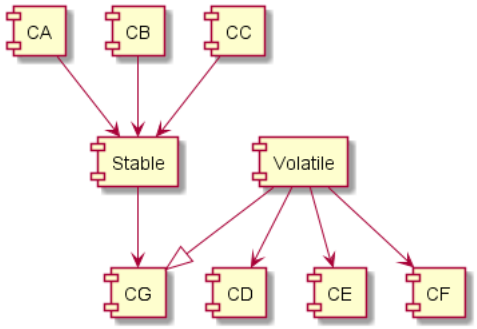


Figure 2. Conforming to SDP

Now with this change in place the volatile component has been made easier to change.

The Stable-Abstractions Principle

Publicerat av Eric Bäckhage mars 11, 2019

What?

This is the last of the Principles of Package and Component Design, the Stable-Abstractions Principle (SAP). It says: *”A component should be as abstract as it is stable.”*

In the Stable-Dependencies Principle (SDP) post we learned that stability, or in that case Instability, I, can be calculated using the formula *I = Ce / (Ca + Ce)* where *Ca* is the number of afferent couplings and *Ce* is the number of efferent couplings. A component that has many dependencies towards it, but depend only on a few external classes is considered stable, and vice versa.

Conforming to SAP leads to stable components containing many abstract classes, and instable components containing many concrete classes.

Why?

The goal of making stable components abstract is to allow for them to be easily extended and avoid them constraining the design.

Instable components on the other hand can contain concrete classes since they are easily changed

How?

Just as for instability we can define a metric that helps us understand how abstract a component is. The formula is very simple:

A = Na / Nc

where *Na* is the number of abstract classes in the component and *Nc* is the total number of classes in the component. A component with A = 1 is completely abstract while a component with A = 0 is completely concrete.

From <http://ericbackhage.net/clean-code/the-stable-abstractions-principle/>

The Stable-Dependencies Principle
Publicerat avEric Bäckhage mars 10, 2019

What?

The Stable-Dependencies Principle (SDP) says: *"Depend in the direction of stability."*

What this means is that the direction of the dependencies in the component dependency graph should point towards more stable components. To understand what this means we need to define *stability* for a component. Uncle Bob turns this around and defines a way to measure *instability*, *I,* using the formula:

I = Ce / (Ca + Ce)

where *Ca* is the number of classes outside this component that depend on classes within this component (afferent couplings) and *Ce* is the number of classes inside this component that depends on classes outside this component (efferent couplings). *I* has the range [0,1] where *I = 0* indicates a maximally stable component and *I = 1* indicates a maximally instable component.

A component that does not depend on any other component is maximally stable and a component that only depend on other components but have no components depending on it is maximally instable.

Why?

When we design our components there will be some components that we want to be easy to change. They contain classes that often needs to be modified when requirements change. If these components have many

dependencies to them (are very stable; have a value *I* close to 0) they will be difficult to change since changing them will impact many other components.

Figure 1 shows a component diagram where SDP is violated. The *Stable* component, that has an instability value of *I = 0.25* has a dependency to the *Volatile* component, that has an instability value of *I = 0.75.* This dependency makes the volatile component difficult to change.

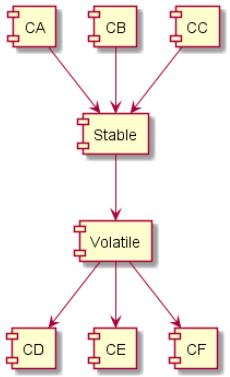


Figure 1. Violating SDP

How?

So how do we fix violations to SDP? The answer is, once again, inverting dependencies by applying the Dependency Inversion Principle (DIP). Look at Figure 1, to fix the violation we can create an interface or abstract base class and put in a new component, CG. We can then let the Stable component depend on CG and have the Volatile component implement the concrete classes. By doing that we end up with the dependency graph showed in Figure 2.

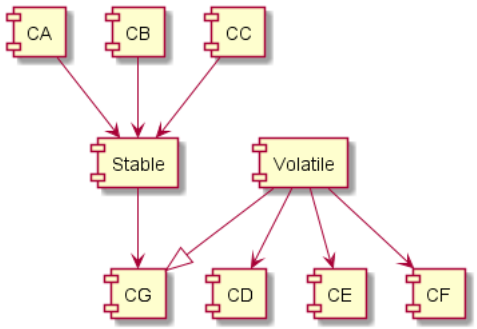


Figure 2. Conforming to SDP

Now with this change in place the volatile component has been made easier to change.