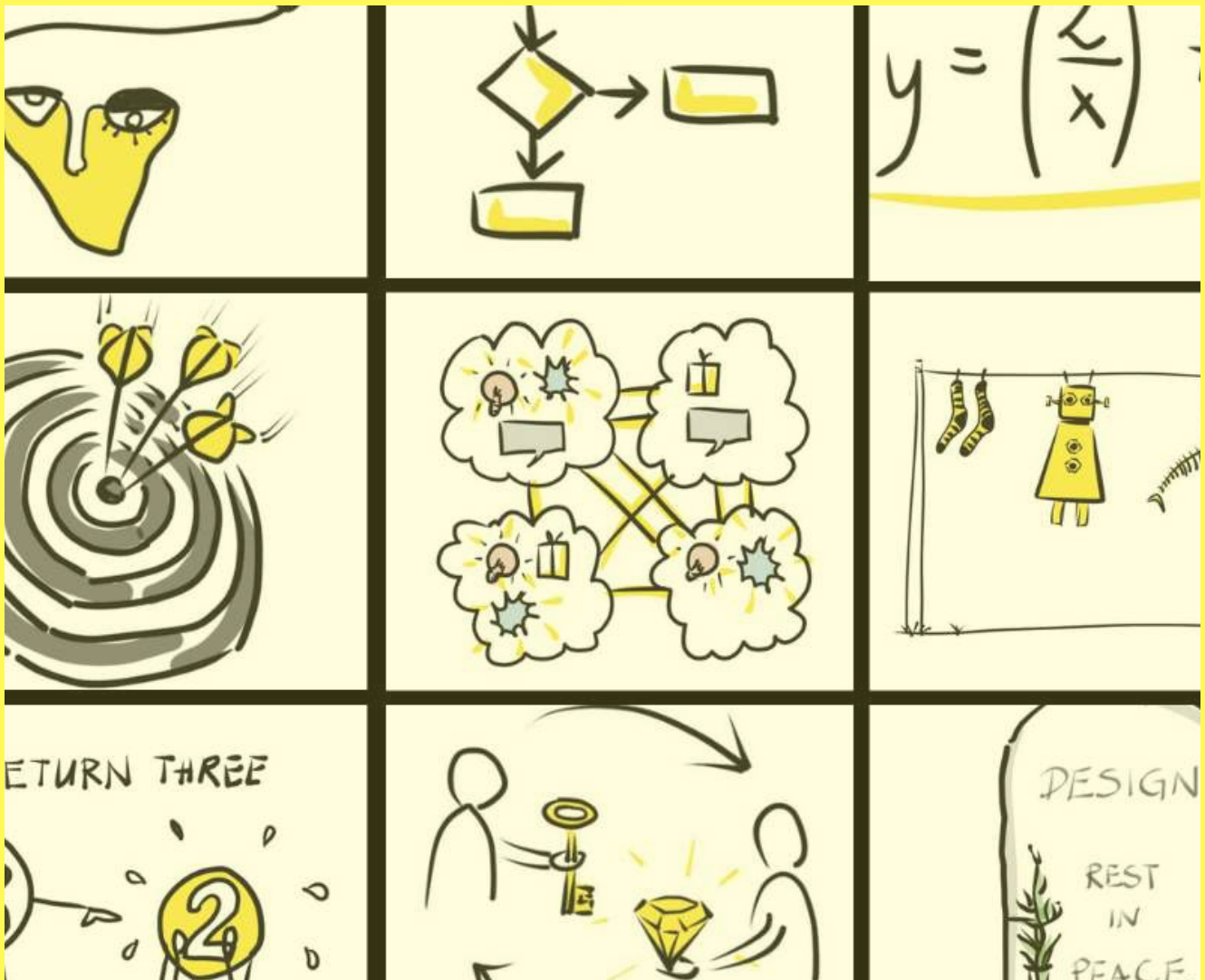


Why does software design rot and what do coupling and cohesion have to do with it?



TOMASZ STOLARCZYK



@stolarczyk



mrpicky.dev

Rotting design BINGO!	1
Symptoms of rotting design	3
Symptom no. 1 - rigidity	3
Symptom no. 2 - fragility	3
Symptom no. 3 - immobility	3
Symptom no. 4 - viscosity	3
What the hell is going on here?	4
Abstract concepts	5
When did it all begin?	6
Coupling	7
Types of coupling	7
Content Coupling	7
Common Coupling	8
External Coupling	9
Control Coupling	10
Stamp Coupling and Data Coupling	11
Decoupling	14
Cohesion	15
Types of Cohesion	16
Coincidental Cohesion	16
Logical Cohesion	16
Temporal Cohesion	17
Procedural Cohesion	18
Communicational Cohesion	18
Sequential Cohesion	19
Functional Cohesion	19
Nonlinearity	20
Stretch out for your cohesion	21
How can we cope with all that?	23
About Mr. Picky	24

Rotting design BINGO!

Some time ago within the confines of getting back to basics, I read "Design Principles and Design Patterns." by Uncle Bob. It is a publication that is almost 20 years old, and mostly you can find there a few design patterns called SOLID at that time. Today those patterns are probably known to all of us as SOLID principles, but that's a different story. What got my attention there was an idea of rotting design.

When we start a project, we usually spend a significant amount of time designing it within the team. Usually, we end up with a clear vision of how it should work. We fix the boundaries - allowing or disallowing some behaviors in our brand new system. As a result, we get a design that is state-of-the-art - at least to our current knowledge. All team members ritually put their signatures in blood on it and agree to respect it till the end of its days.

In the beginning, everything looks great. We develop with ease, first versions get to production, time goes by, and we are receiving first feature requests. Next to them appear little exceptions to our original agreements. Some tiny "hacks" here and there. Just because they are "temporary solutions" or because we "need to fix a bug quickly." Thus our design starts to rot, and it is not the bleeding edge anymore. At some point, it has nothing in common with our first approach, and someone says "we need to rewrite it." However, it can happen that not everyone would be so enthusiastic about doing so. Especially people that are not close to the code but have to approve such a decision and justify its business viability. The worst case scenario is that we stay with rotting design without the chance to redesign it. If so, any changes, even the smallest ones, are hard to implement. The situation becomes very frustrating for developers as well as product owners, managers and so on, up to the top of the food chain hierarchy.

That's usually the moment when there starts a witch-hunt and the temptation to blame the whole evil thing on constant changes or blurry requirements, customer's lack of business knowledge, lack of documentation or user stories, you name it. There are situations when it's true, but still, we are required to write code that is resilient and easily maintainable. Why? Because in most cases we work in agile environments, and by default, requirements will change. Our goal is to be one step ahead of customers. This is the world we live in.

Let's chill out a bit for a moment and play BINGO:

Rotting Design BINGO

changes hard to apply	simple change impacts numerous modules	implementing simple changes takes forever
changing one place harms another	fixing a bug causes x others	modules are not reusable because of their dependencies
rewriting a code instead of reusing existing one	easier to do "hacks" than go "by the book"	environment is slow and inefficient

Picture 1. Rotting design BINGO

So? BINGO? If you have won, I have some good and bad news. The bad news is that probably you are struggling with the rotting of the design in your application. The good news is that I'm going to shed some more light on its disease. However, before that, I would like to ask you to take a closer look at the examples on the bingo board. Each of them shows evidence of one out of four symptoms of rotting design.

Symptoms of rotting design

Symptom no. 1 - rigidity

We can notice it when a seemingly simple change in the code triggers a chain reaction of unexpected changes in the entire application. As a result, it becomes tough to introduce any changes as it is very time-consuming.

Real life example could be a situation when a product owner comes to a team and asks about the estimation of adding just this one, tiny checkbox to the form. After a while, the team estimates that two devs are going to do it in a two week period. Unbelievable? Oh, I assure you it's true ;).

As a result, non-critical issues are not being fixed, because Product Owner or manager are afraid of spending the team's time on fixing them, as they don't know how long it could take.

Symptom no. 2 - fragility

When after a change in one place, you get bugs in totally different parts of the system that are logically not connected with the original place, you are probably dealing with fragility.

You fixed the tax policy, but you also introduced three new bugs related to adding products to the cart, customers filtering in admin panel and error logging.

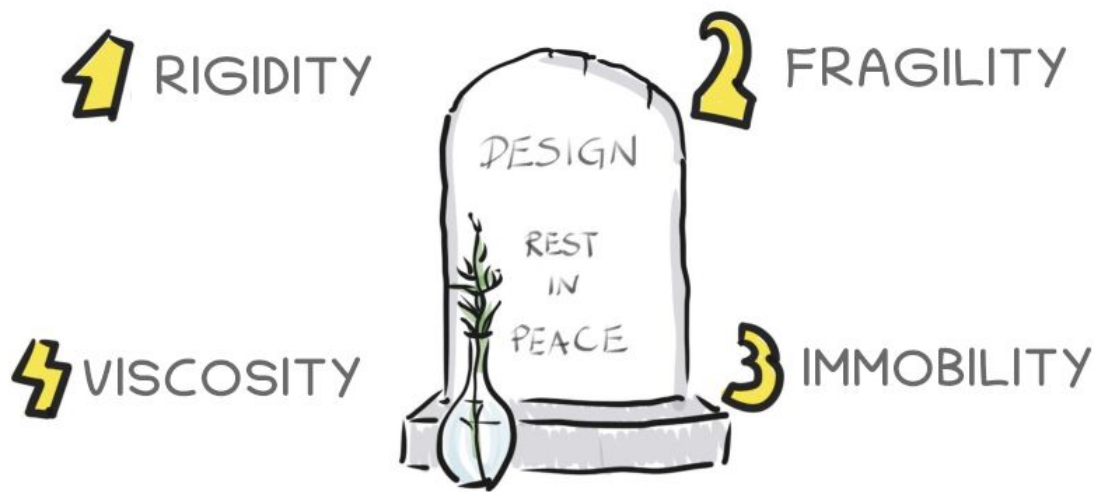
Once again fear appears because no one knows where and how many business functionalities break after bugfix. There is also a suspicion that developers have lost control over the code.

Symptom no. 3 - immobility

Now I want you to imagine a situation when you implement a new feature, and you recall a piece of code that does exactly what you need. You find it, analyze it and leave it there, rewriting it in your functionality. But why? Why can't you reuse it? Because the overhead of extracting it and satisfying all its dependencies is so huge that it simply doesn't make any sense.

Symptom no. 4 - viscosity

When we develop our application, we can do it in one of two ways: according to the current design or against it. In the case when it is easier to follow the latter approach, it probably means that viscosity is in attendance. We can easily link it to all those situations when we knew how something should be coded, but instead we did some little hack to deliver our feature quickly. To sum up, it is easier to do those tiny hacks than follow the design rules.



Picture 2. Four main symptoms of rotting design.

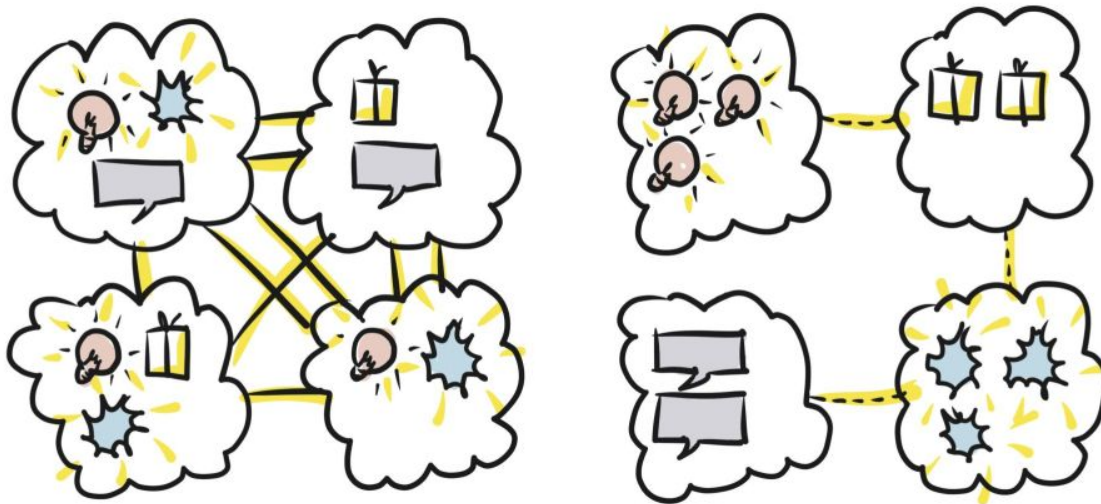
What the hell is going on here?

Now it's time to finally get to know what disease our design is suffering from. Once we have found out that it is not because of changing requirements, what else can it be? Well, the most common reason for that is the wrong dependency management in our code. If so, two old concepts come to the rescue: coupling and cohesion.

Coupling can be described as a strength of the relationship between classes or modules. Typically we talk about its two types: tight and loose coupling. However, as we all know, between two extreme values, there is always this gray area where we can place a lot of different cases, which is the case here, too. More about it later on.

Cohesion, on the other hand, is a degree of how elements of a specific class/module go together, how they match the class and each other. They all should effectively work towards a common goal. They should all naturally mesh with each other, without any "glue" used to connect them. When it comes to cohesion, we also usually describe it as low or high, but guess what: there is more than that.

Generally, we consider good design as one that it is easily maintainable, resilient to changing requirements and having loosely coupled, independent classes, that do exactly what they were created for. In other words, such a design is marked by loose coupling and high cohesion.



Picture 3. Strong coupling and low cohesion (on the left) vs. loose coupling and high cohesion (on the right).

Abstract concepts

From the very first time when coupling and cohesion were defined, there was one issue with both of them. Let me ask you a few questions:

- how can I recognize that one class is tightly coupled with another?
- how can I verify whether or not a specific class is cohesive?

Well, my class is definitely highly cohesive and loosely coupled with others, but a class of my colleague's... hmmm... he or she had better reconsider their future career as a developer.

It turns out that each of us can have different images of what is loosely/tightly coupled and what is or is not cohesive. What is most interesting, any of us can be right. Why is that so? Both concepts are highly abstract. How do we deal as human beings with such concepts? We look for analogies to our daily lives. You can quickly validate this by looking into the most highly rated answers to general questions about coupling and cohesion on Stack Overflow. You will find a ton of analogies. Just take a look:

- "Car and Spare parts analogy of coupling" (<https://stackoverflow.com/a/37993102/8854363>)
- "You and the guy at the convenience store." (<https://stackoverflow.com/a/39988/8854363>) - loose coupling
- "The cheese store." - high cohesion
- "You and your wife." - tight coupling

Analogies are great, and we like them. They help us to get the context and shed some more light on the problem. But do they really explain everything? I don't think so. Ok, so now it's time to dig into Wikipedia or any other pedia. What we see there are some classical definitions that I

more or less included earlier in this article. Then we scroll a bit down, and we notice something more as there are types of coupling ([https://en.wikipedia.org/wiki/Coupling_\(computer_programming\)#Types_of_coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming)#Types_of_coupling)) and types of cohesion ([https://en.wikipedia.org/wiki/Cohesion_\(computer_science\)#Types_of_cohesion](https://en.wikipedia.org/wiki/Cohesion_(computer_science)#Types_of_cohesion)). No one told us about them, so let's take a look at what more your search engine of choice returned. So now is the moment when real fun begins. There are a lot of publications and books. One author says that, for example, there are only two types of coupling, another says there are twelve. Someone else that all the others are wrong because they didn't consider this and that. At the same time, some of the publications even use different names; for example, cohesion is sometimes referred to as "functional binding" or "strength".

When did it all begin?

To answer this question, we need to go back in time to 1974, when Larry Constantine, Glenford Myers, and Wayne Stevens published in *IBM Systems Journal* their article called *Structured Design* (<https://ieeexplore.ieee.org/document/5388187>). They described coupling and cohesion there, but actually, Larry Constantine is considered to be the father of these concepts.

One year later, the book *Reliable Software Through Composite Design* by Glenford Myers was published. Quite a big part of it was dedicated to both concepts. On a side note, I can say that it's a great book; it's written in an accessible way, and the most fascinating thing about it is that while reading it you can have an impression that in the last 50 years little has actually changed when it comes to the challenges in software design and its modularity.

Then after another few years, Larry Constantine and Edward Yourdon published their book called *Structured Design* - the same as the article from 1974. They were also referencing Mayers' book, which goes to show that those guys for few years were doing a lot of research in the area of coupling and cohesion.

We could probably say now that those were the 70s, so it should be quite outdated now. Procedural programming was standard back then, they had modules, and now we do OOP, we have our cool, shiny classes. Why should we even care? Well, it turns out that even though paradigms have changed, coupling as well as cohesion are still with us and they feel pretty fine in OOP. When in the early 90s Timothy Budd published his book *An Introduction to Object-Oriented Programming* you could find there that counterparts of modules in OOP are classes. Personally, I would go further and say that both concepts are so universal, that I can easily imagine, for example, a loose/tight coupling between packages in Java or even between microservices.

Coupling

Now that we know when it all started, we should focus on details, so let's take a closer look at coupling. From what has already been said or based on experience, we know that this is a degree of the relationship between classes - the strength of connections between them. The value of this strength can tell us what a probability of the need for changes in other classes than the class that we are currently working on is. At the same time, it impacts the costs of development and maintenance.

Because of that, we should pursue loose coupling, so one would be able to easily analyze/debug/maintain any class, without detailed knowledge about other classes within the system. The looser the coupling is, the more independent the classes are. How can we define a degree of relationship between the two classes? To answer this question, we should take a look at communication between them. How do they do it? What does the message look like? When we know this, we can define what type of coupling it is and what it entails.

It was Glenford Myers who proposed the most common division of coupling into types in his book *Reliable Software Through Composite Design*. He defined six types, and now we are going to give them our attention. Let's go from those considered as the least to those that are the most welcome.

Types of coupling

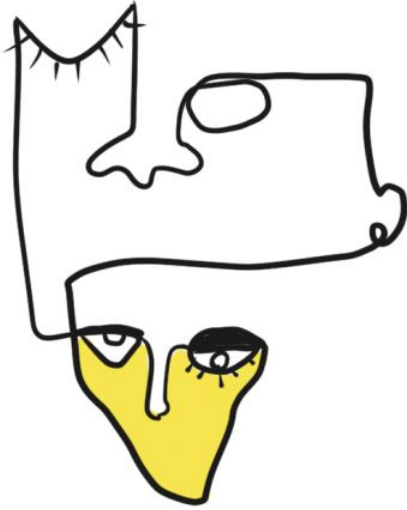
Content Coupling

Also known as "pathological coupling". That's because it usually occurs when one class directly uses private members of the other class. Objects of the first class are changing an internal state and behavior of objects of the second class. From the message's point of view, it looks like the sender forces the message on the recipient, who is unable to resist.

In our daily dev life, good examples of it could be:

- not respecting access modifiers,
- reflection,
- monkey patching.

The degree of how much we know about the other class is the highest as we know literally everything.



Picture 4. Content coupling – pathological – we reach and manipulate the internal state of objects.

Listing 1. The object of the `Image` probably would prefer to set its `description` field on its own.

```
Class<?> clazz = Image.class;
Object imageInstance = clazz.newInstance();

Field descriptionField = imageInstance
    .getClass().getDeclaredField("description");

descriptionField.setAccessible(true);
descriptionField
    .set(imageInstance, "Pen Pineapple Apple Pen");
```

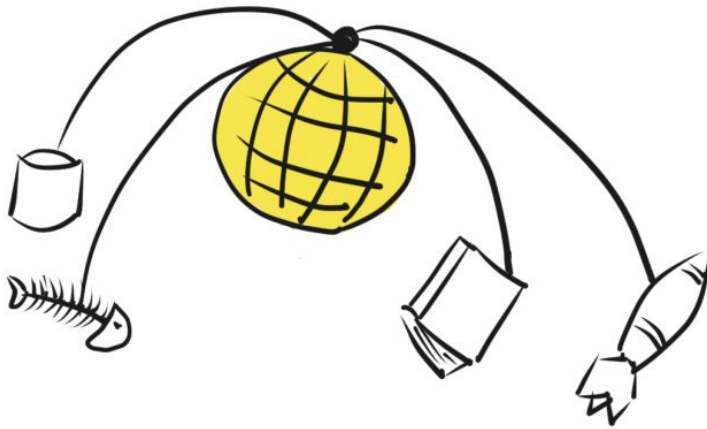
Common Coupling

When classes change and communicate via a shared global state, we probably have a brush with common coupling. It means that when the state is changed in runtime, we may have no idea what objects rely on it and how it would impact their behavior.

The name itself is derived from the `COMMON` statement used in Fortran, where it means a piece of memory shared between different modules. Because of that, they are coupled to each other through this common environment.

It's easy to notice that in this kind of coupling, we need to be extremely careful while changing the global state as it could cause changes in places that we would not expect. It requires an immense knowledge of all classes that depend on that state. However, it's not only about the

state but also about the data structure - any change in it causes changes in all dependent classes.



Picture 5. Common coupling – different classes in the system communicate via a global shared state.

External Coupling

Let's consider now the case when the structure of the message used in communication between classes comes from the outside of our system - the structure is external, so is coupling.

A good example could be using classes from third-party libraries: because of doing so, we become tightly coupled to them, and we lose control of any changes introduced in their next releases. Another example could be some external protocol or data format used in communication.

Listing 2. External coupling – let's imagine now removing the `getAmount()` method (that is used everywhere in our system) by an author of popular money class provider library.

```
import popular.money.class.provider.Money
import popular.money.class.provider.MoneyAmount
```

```
public class Foo {

    public MoneyAmount receive(Money money) {
        //method body
        return money.getAmount();
    }
}
```

Just to make it clear - we will always need third-party libraries and will always integrate with something outside of our system using some external standard or protocol. However, the most important thing is to disallow those libraries/integrations to spread all around the system. The best idea would be to use them to do their job but then to block them out using our abstractions.

The idea behind it is not to "rewrite everything" when, for example, some external library went into a maintenance-only mode, but to change/replace the implementation in some specific places.



Picture 6. External coupling – objects communicate via a protocol or structure that comes from the outside (third-party lib)

Control Coupling

Imagine the situation when you are writing a piece of code and you call a method with some kind of flag as a parameter. Moreover, based on the value of the flag, you can expect different behavior and result. You as a caller have to know quite well what's going on inside the called method, and the method/class itself is not a black box for you anymore. At this stage, you are more like a coordinator as you say what has to be done and what you expect in return. The described scenario is a typical example of control coupling, where one class passes elements of control to another one. What's important here, the class that passes those arguments does it deliberately as it wants to achieve specific results. The most common cases for control coupling are methods that take the above-mentioned flags or use switch statements.

In OOP, objects should decide what to do based on their internal state and received data and not on an external flag passed by someone with a view to doing something.



Picture 7. Control coupling – object passes to the other object elements of control that affect execution and a returned result.

Listing 3. Control coupling – it could be a good idea to split this method into two separate methods.

```
public void save(boolean validationRequired) {  
    if (validationRequired) {  
        validate();  
        store();  
    } else {  
        store();  
    }  
}
```

Stamp Coupling and Data Coupling

When communication between classes is not either pathological (content coupling) or through the shared global state (common coupling), neither via external protocol nor structure (external coupling) and it has no elements of control (control coupling), then we have probably ended up with one of the two types that put the message and its structure on the pedestal.

In the first type (stamp coupling), classes use data structure (our own and not from the third-part lib) in communication. Usually, it is some kind of DTO

(<https://martinfowler.com/eaaCatalog/dataTransferObject.html>). The recipient can decide whether he wants to use all data from the structure or just a part of it because the rest can be totally useless in a specific context.

For example, in different applications, we can often find classes whose name ends in "details" or "data". Does it resonate with you? So let's take a look at `EmployeeDetails` class, as you probably suspect you could find everything about an employee there - the sky is the limit. Let's assume now that we have the method that takes an object of this class as a parameter and

should return an employee's address, based only on the employee's id from inside of the whole structure. Now we could argue that there is no coupling between caller and recipient as we pass some set of data, the recipient takes what it needs, and that's it. However, there is coupling to the whole structure of the passed data. There could be a case when a change occurs in a part of the structure that is not used by the recipient, but some adjustments are still required.

This kind of coupling can also lead to the creation of artificial data structures, that would hold unrelated data. Then frivolous adding different items to such data "bags" could become a common practice in our code.

Let's think now how to avoid coupling to the whole data structure. The answer is quite clear - use just data items instead. Doing so, we should end up with the loosest type of coupling that is data coupling.

Basically, it means that the recipient (method) takes a list of arguments (values). Notice that here we only pass values that are key to method execution. In other words, there is no space for any unnecessary items. For example, instead of passing the whole `EmployeeDetails` structure, we could pass just an employee's id. In return, we also could get some single value, like a zip code.

Listing 4. Difference between stamp coupling and data coupling.

//Stamp coupling

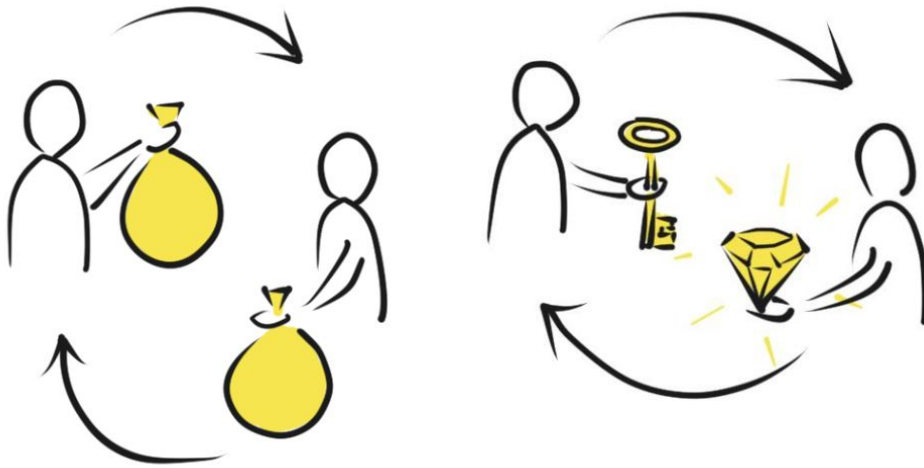
```
public EmployeeAddress findAddressFor(EmployeeData employeeData) {  
  
    EmployeeAddress address = repository  
        .findByEmployeeId(employeeData.getId())  
    //method body  
    return address;  
}
```

//Data coupling

```
public ZipCode findZipCodeFor(EmployeeId employeeId) {  
  
    EmployeeAddress address = repository  
        .findByEmployeeId(employeeId)  
    //method body  
    return address.getZipCode();  
}
```

We can say that a class/method is marked by data coupling when we can't notice any other types of coupling there. Why did I even mention it? Let's imagine a situation when we see a method that has a single parameter. We could potentially think that there is a stamp or data coupling, but it can turn out that actually the parameter is an element of control that is specific for control coupling. In such a situation, we should take a closer look at how a caller of the method sees it. If the caller uses a parameter as an element of control and expects some

specific behavior by setting it then indeed we have a case of control coupling, however, if the caller passes the value and treats it just like any other data we are probably dealing with data coupling.



Picture 8. Communication in stamp coupling (on the left) and data coupling (on the right)

That's it when it comes to types of coupling. I want to repeat here that this division into types is the original by Myers - the emphasis is put on communication between classes and the message itself. I'm mentioning it as there are also other types of coupling that are specific for OOP and those are probably the ones that we are most familiar with. What I mean by that is how classes are connected physically. Whether we create an object of some class directly, or use an interface, or, finally, if we may just emit some events without the knowledge who and how consumes them.

Myers defined types of coupling when procedural programming reigned supreme. I have an impression that then when OOP became popular, those types lost popularity and everybody's attention focused only on those specific for OOP. Why do I think so? When we take a look at most code examples on the Internet looking for "examples of tight/loose coupling", we mainly find code snippets that consider it only on a level of using interface (loose coupling) vs. specific implementation (tight coupling). But that's a totally different story for another discussion or ebook.

Decoupling

Now that we know what types of coupling we can expect and how to identify them in our code, it would be good to find some ways of getting those types that are usually more desired. To do that we can use decoupling, which can be any technique that helps us to achieve more independent classes.

Each of the types suggests some forms of decoupling and in general it means favoring types with looser coupling. For example, if we identify a type of control coupling in a method and we can see two flows with different results based on the passed element of control, maybe we could split the method into two separate methods. Another example could be a situation when we see that a method has a parameter that is data structure and uses just a few fields from it. In this case, maybe we could replace the whole structure with a short list of parameters.

The next technique could be to design classes in such a way as though communication between them could be done using queues. In this approach, we focus on what a class should do and what it needs to execute its logic. Moreover, the moment of execution becomes irrelevant.

A good practice could also be to use "local" (in other words, context-specific) data structure in communication between classes. Thanks to that, we don't allow them to spread all around the system but just to be used where they fit.

So that would be it for now when it comes to coupling. We know what it is, what types of coupling there are, how to identify them, and how to achieve more independent classes. Now it's time to move on and take a closer look at cohesion.

Cohesion

In the previous part, there was a lot about communication between classes, so now it's time to focus on the classes themselves and the elements they consist of. When do we know that a specific element fits into a certain class? Sometimes what we assume does, actually doesn't necessarily do it for our colleague. In such a case, cohesion comes to the rescue. In general, it's a measure of how elements go together.

Regarding cohesion, there was a problem with what to call it to start with. At the beginning it was named "intramodular functional relatedness", but this term was found to be a bit clumsy. Cohesion was also called "binding" or "functionality". In his book Myers calls it "module strength", which is also entirely accurate as it can tell us how tightly elements of a class are bonded in the context of the functionality they perform - to what extent they are related. Finally, the term "cohesion" was chosen by analogy to the cohesion of groups - a similar concept in sociology.

Glenn Vanderburg wrote a post where he explained why in his opinion programmers have a problem with understanding the whole concept of cohesion. He argues that the name is not as self-explanatory as is the case of coupling and it is not used so often in everyday life, either. Because of that, we can't refer it to our daily situations. To better visualize the term, he proposes comparing it to other words with the same root. For example, "adhesion", which describes stickiness. If something sticks to something else, it is a one-sided process, often aided by a third-party - glue in most cases. He also used an example of duct tape that doesn't have to fit into anything as due to its stickiness it sticks to almost anything. On the other hand, cohesive elements match each other perfectly, and there is no need to use any glue between them.

The better our understanding of a designed problem is, the higher cohesion we can end up with. This is why it is absolutely vital to spend time on getting familiar with the problem itself and the business domain. Once we know it quite well, we should focus on grouping elements in a way that reflects the problem. Thanks to that, we maximize relationships between them.

Now let's think what drives the grouping of elements into a specific class. To answer that, we should take a closer look at types of cohesion, starting with one with the lowest and ending up with the highest degree of it.

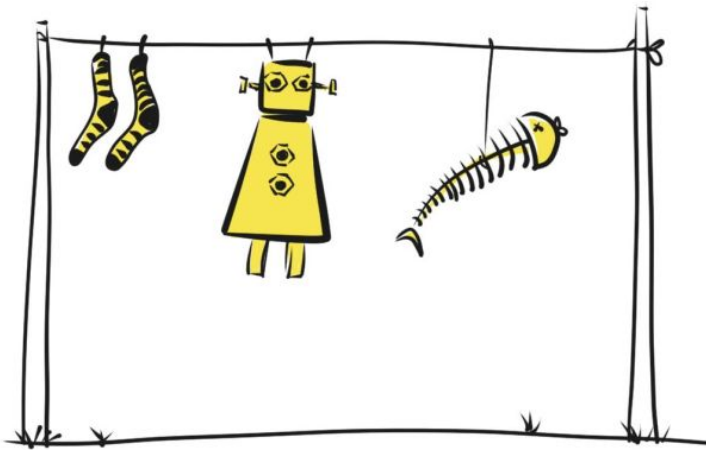
Types of Cohesion

Coincidental Cohesion

The first criterion for assigning elements to a specific class could be an actual lack of any criteria. It means that elements are grouped totally randomly - a real free-for-all. Such classes are created coincidentally and not in the process of design.

If we were to give this type any number on the scale of cohesion, we would probably give it a zero as there is no cohesion here at all.

Classes with this type of cohesion are quite rare but if we find any, they are probably artifacts of bad modularization. Usually, they are created because we find some sequence of methods' calls in a few different places in our application, and we decide to extract them to a single class. Extracting, as we know, is not a bad thing, but if single methods do totally different things and, worse still, if we do it without thinking about what those sequences do in each of the contexts, we probably should leave them as they are. Otherwise, we can end up with a class that we will have to modify when something changes in one of those contexts but not in others.

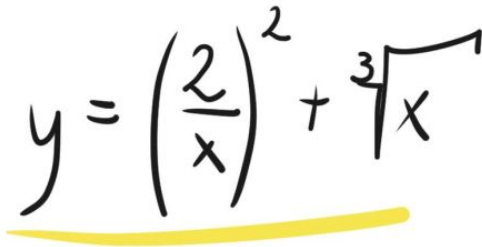


Picture 9. Coincidental cohesion – total randomness in grouping elements into a class.

Logical Cohesion

In logical cohesion, elements of a class are grouped because they solve problems from the same category. Logically there is something that they have in common even though they have different functionalities.

A good example here could be mathematical operations, as each of them can do various things, but often they are grouped in some Math or Utils class with... mathematical operations. The above-mentioned Utils classes are usually good examples of this kind of cohesion. One to rule them all!


$$y = \left(\frac{2}{x}\right)^2 + \sqrt[3]{x}$$

Picture 10. Logical cohesion – class' elements solve the same type of problems. For example, a class that aggregates all mathematical operations.

Temporal Cohesion

If logical cohesion is combined with another grouping criterion that is the moment of execution, we will probably get temporal cohesion.

The best examples of this type of cohesion would be classes/modules that initialize, stop, or clean something. Initialization methods are logically connected as they “initialize something”. What is more, those methods usually have to be executed at some specific point in time or period.

Quite often, such classes/modules need to have this type of cohesion, and any attempts to change it to more cohesive types are highly ineffective.



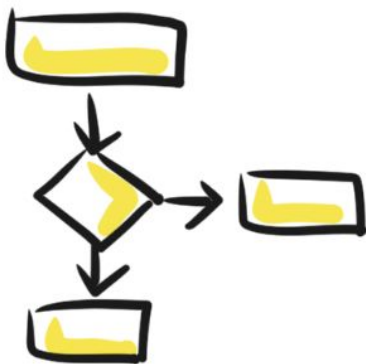
Picture 11. Temporal cohesion – elements of the class have to be executed within the same period.

Procedural Cohesion

The next type is procedural cohesion, and it usually occurs when instead of modelling a domain we focus on an algorithm of execution - the order of steps that have to be executed to get from state "A" to state "B". It may lead us to the approach where we end up modelling the algorithm itself and not the problem that we were supposed to solve.

Cohesion in this type is higher as elements are not only grouped because of their moment of execution like in temporal cohesion, but they also have to be executed in a specific order.

Loops, multiple conditions, and steps in the code can show evidence of procedural cohesion. Those are usually classes that, while being analysed, make us want to take a piece of paper, a pencil and start to draw a block diagram.



Picture 12. Procedural cohesion – usually when we model algorithm itself and not the problem that we were supposed to solve

Communicational Cohesion

The first type of cohesion that focuses on the modelled problem is communicational cohesion. The criterion for grouping elements into a class here is that they are communicationally connected, which means that they work on the same input or return the same type of output data. Thanks to that, it can be easier to reuse the whole class in different contexts.

Classes like this have origins in thinking about operations that can be done on some particular set of data, or on the other hand - operations that have to be executed to get that set of data.



Picture 13. Communicational cohesion – elements in the class are grouped because they are communicationally connected, so they use the same input data or return the same output data.

Sequential Cohesion

At the stage when we group elements because of their sequence of data processing, so that the output of one element is at the same time input of the next one we probably have a brush with sequential cohesion.

There could be only one small issue here. If we have such a sequence, we can cut it in different ways. It means that created classes can do more than one functionality, or quite the opposite - only some part of the functionality. That's the only reason why in the hierarchy of cohesion types it stands below the last one, which is functional cohesion.



Picture 14. Sequential cohesion – we group elements because of the sequence of data processed by them.

Functional Cohesion

We achieve it when all elements within a class are there because they work together in the best possible way to accomplish the goal - functionality. Each processing element of such a class is

its integral part and is critical for the functionality of the class. The class itself performs no less and no more than one functionality.

What's interesting, mathematical operations are often mentioned as an example of functional cohesion because they do exactly what they should and nothing more than that. I have also mentioned mathematical operations in logical cohesion, but there all operations were grouped in a Utils or Math class, and now I mean more creating a class per operation. So the responsibility of a single class would be only to perform a single operation.

It can be quite easy to judge whether a class has functional cohesion when we deal with a "small" functionality, but at higher levels of abstractions it can become a bit tricky. That's why we can try to do it by looking for signs of other types of cohesion - similar to the case of data coupling. A bit like a shell game. For that, chapter "Exercises" could be helpful. You will find there quite interesting high-level exercises that can help you with everyday work to define what type of cohesion your class has, so don't stop, keep reading :).



Picture 15. Functional cohesion – elements within a class are there because they work together in the best possible way to accomplish the goal - functionality.

Nonlinearity

When it comes to types of cohesion, one more thing is worth mentioning. In the book *Reliable Software Through Composite Design* as well as in *Structured Design*, the authors suggested assigning a weight to each of those types. Nevertheless, what is most important here they were quite reticent about it. That's because they insisted that it might be a bit too early for it and they didn't want it to influence future research into cohesion. They emphasized that values were chosen "based on educated guesses" and "these aspects of the model must be verified and refined based on data collected".

The only reason why they decided to assign those weights, was to show that in their opinion those weights have nonlinear values and for example, in *Structured Design*, they were proposed as follows:

- 0: coincidental
- 1: logical
- 3: temporal
- 5: procedural
- 7: communicational
- 9: sequential
- 10: functional

When we can identify what types of cohesion our classes have, and we can see some places for improvements, these weights can be helpful as they can show us where we should utilize our energy. If we, for example, see that some class has sequential cohesion, it means that it is already almost "ideal" and maybe it makes no sense trying to get to functional cohesion. Instead, it is better to take care of places where the gain is higher, for example, between temporal and procedural cohesion.

Stretch out for your cohesion

Now I have something of a warm-up. A few simple exercises that can help you define the type of cohesion that a class has. In *Structured Design* (for more details about the article itself go to the "When did it all begin?" chapter) the authors proposed a simple technique, which was to describe a module's (class in our case) function in a single sentence.

So now let's look into their guidelines:

1. "If the module is functional in nature, it should be possible to describe its operation fully in an imperative sentence of simple structure, usually with a single transitive verb and a specific non-plural object."

For example:

- Calculate VAT
- Calculate commission
- Read the temperature
- Retrieve the order info

2. "If the only reasonable way of describing the module's operation is a compound sentence, or a sentence containing a comma, or a sentence containing more than one verb, then the module is probably less than functional. It may be sequential, communicational, or logical in terms of cohesion."

For example:

- Update time spent on the task, employee's work time, and invoice based on time card. - probably communicational cohesion, as all elements are communicationally connected by a time card.

- Update the order and save it. - sequential cohesion.
3. "If the descriptive sentence contains such time-oriented words as "first," "next," "after," "then," "start," "step," "when," "until," or "for all," then the module probably has temporal or procedural cohesion; sometimes but less often, such words are indicative of sequential cohesion"
For example:
 - Before sorting, save data, remove duplicates, and check checksums - temporal cohesion as we assume that order is not essential here.
 - Download exchange rates, next send sales results to the sales department and new orders to the orders department. - probably procedural cohesion as we model the following steps where order is essential but output data of elements is not input data of those following them, so it's not sequential.
 4. "If the predicate of the descriptive sentence does not contain a single specific objective following the verb, the module is probably logically cohesive. Thus, a functional module might be described by "Process a GLOP." A logically bound module might be described by "Process all GLOPS," or "Do things with GLOPS.""
 5. "Words such as "initialize," "clean-up," and "housekeeping" in the descriptive sentence imply temporal cohesion."

Those guidelines might be considered too general, but in my opinion it is an excellent exercise sometimes to ask ourselves "what does that class actually do?".

What about a situation when we are considering adding a new element into an existing class? In *Structured Design* (the book), the authors give us a similar technique based on the principle of association. This time we should answer the question "why does the element fit there?". As a result, we could end up with the following statements:

- "Z is associated with this module containing X and Y, because X, Y and Z are all related by virtue of having the 'glop' property"
- "It's OK to put Z into the same module as X and Y, cause they' are all related in such-and-such a manner"
- "It's OK to put Z into the same module as X and Y, cause they're all members of the glop set."

Last but not least, a hint from me: model problems/features in a group. Why? Since the bigger our understanding of the problem, the higher the cohesion is. However, there is a tiny catch here. Each of us analyzing a problem has their own version of it in mind and the same goes for solutions to it. That's why it is so important to share this process with others. Through sharing, we build a common understanding of the problem within the team.

How can we cope with all that?

Imagine a situation when without going into any details your colleague tells you "that new class that you've just created has poor cohesion." I can imagine how adrenaline hits you, and at this stage, you probably can react in very different ways. You can flee, fight, or... debate.

Assuming that he or she is not a wild animal, and we are not going to end up as his or her dinner, I encourage you to take the third option. Maybe such a situation is just an opportunity to discuss and define what we understand as coupling or cohesion. Maybe we have a different understanding from our colleague. Next, we can also discuss what types of both terms we can identify in our system and whether we allow them there or not. It's not like some of them are right or wrong by default and we should avoid thinking in this way. There is no black or white here. As it is quite common in our profession - it all depends - usually on the case or context. Sometimes we will need strong coupling and trying to lose it would not make any sense. The most important thing for us is knowing all those types and knowing the good, the bad, and the ugly sides to them.

Let's take a look now at your BINGO board.

If you have "won," think for a second about going back to the roots and taking a look at dependencies between classes and the classes themselves - what they do and what their responsibilities are. You might be able to identify some types of coupling and cohesion, and you may find some places for improvement by changing those types. Thanks to that you could end up with looser coupling and higher cohesion.

On the other hand, if you have "lost," it means that everything should be fine with your code but I hope that by writing this post I will help you to pay attention to things that will allow you to enjoy your clean design forever.

About Mr. Picky

Mr. Picky is a developer who loves exploring different software designs and architectures. Is a big fan of DDD, software design in general, refactoring techniques and using the right tools for the job. Mr. Picky is very picky when it comes to choosing the approach to follow. Likes to know his alternatives with all their pros and cons. Mr. Picky will show you things from different angles to help you make the right decisions. So dare to be picky! And follow Mr. Picky, dear Watson!

twitter: [@stolarczyk](https://twitter.com/stolarczyk)

www: mrpicky.dev

mail: stolarczyk@gmail.com

