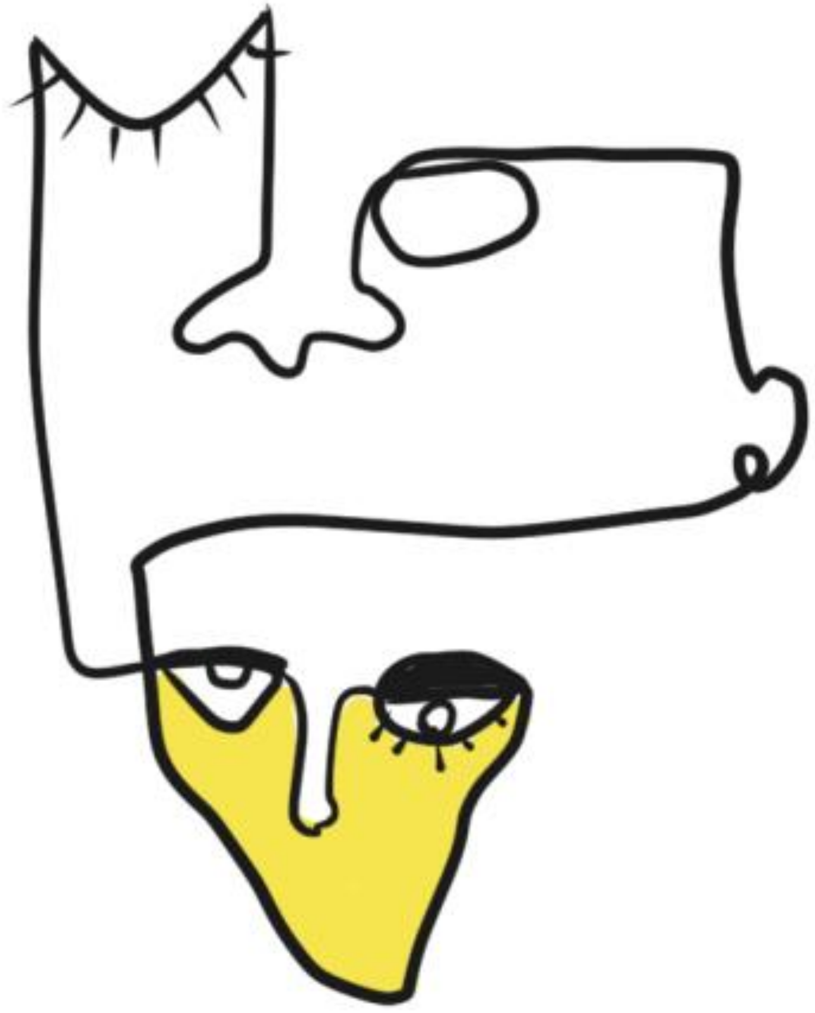
Types of Coupling

https://mrpicky.dev/six-shades-of-coupling/

Content Coupling

- Also known as pathological coupling
- 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.

Content Coupling



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.

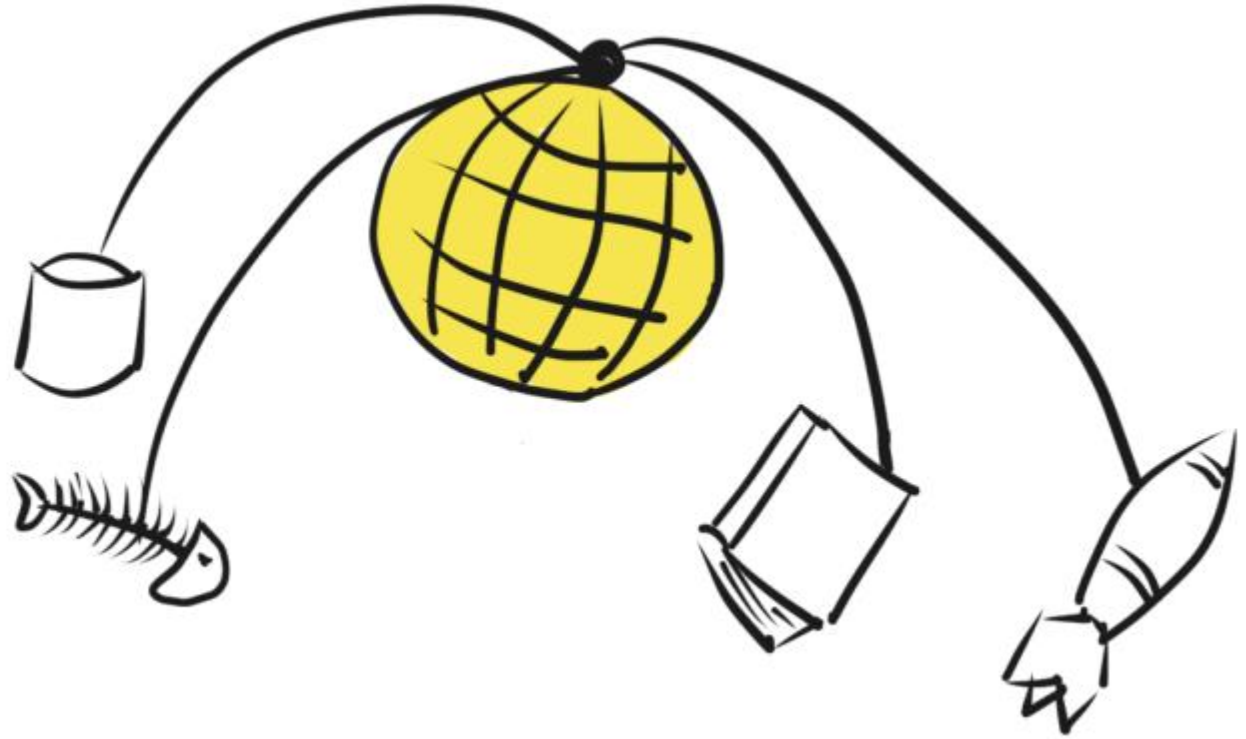
The object of the Image probably would prefer to set its description field on its own.

```
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.

Common Coupling



Common Coupling

- 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.

Common Coupling

- 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.

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.

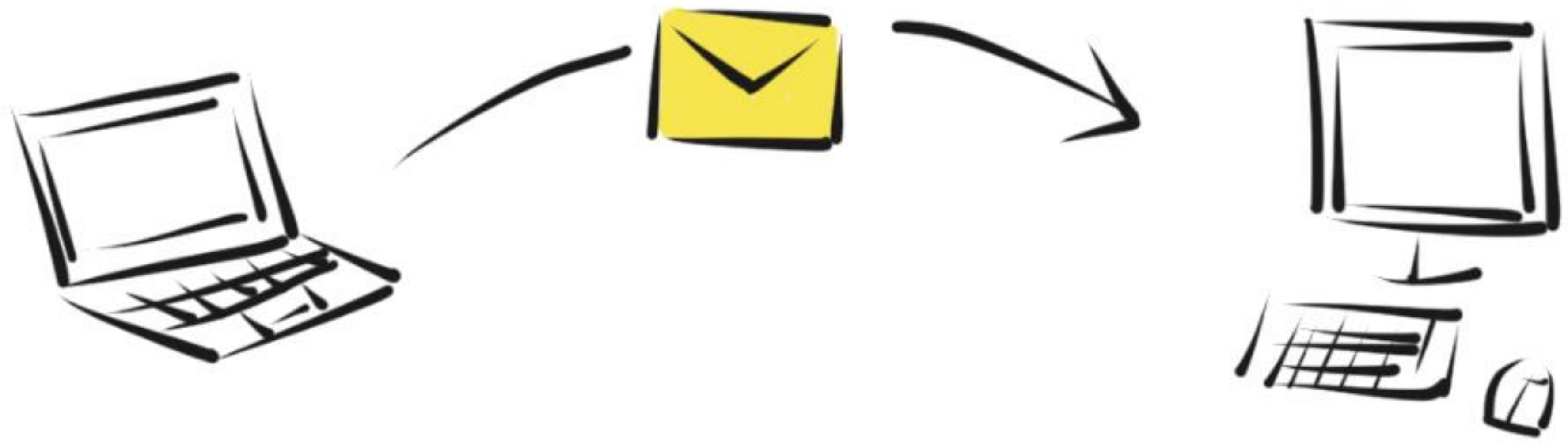
External Coupling

- 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.

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();
        }
}
```

External Coupling



External Coupling

- 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.

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.

Control Coupling

- 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.

Control Coupling

- 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.

Object passes to the other object elements of control that affect execution and a returned result.

RETURN THREE
K
2

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.

Stamp Coupling

- Classes use data structure (our own and not from the third-part lib) in communication.
- Usually, it is some kind of DTO.
- 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.

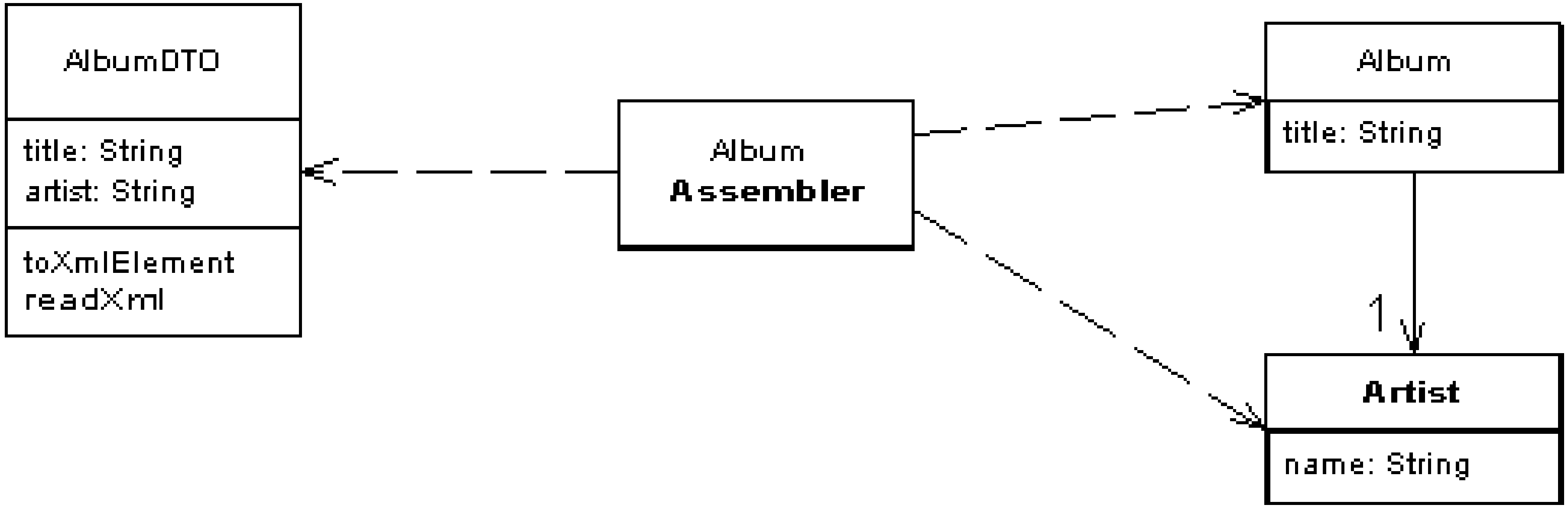
DTO (Data Transfer Object)

- An object that carries data between processes in order to reduce the number of method calls.

DTO

- When you're working with a remote interface, each call to it is expensive.
- As a result you need to reduce the number of calls, and that means that you need to transfer more data with each call.
- One way to do this is to use lots of parameters.
- However, this is often awkward to program - indeed, it's often impossible with languages such as Java that return only a single value.
- The solution is to create a Data Transfer Object that can hold all the data for the call. It needs to be serializable to go across the connection. Usually an assembler is used on the server side to transfer data between the DTO and any domain objects.

DTO


AlbumDTO
title: String
artist: String
toXmlElement
readXml
Album
Assembler
Album
title: String
1
Artist
name: String

Stamp Coupling

- 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.

Stamp Coupling

- 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.

Stamp Coupling - Drawback

- 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.

Data Coupling

- 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.

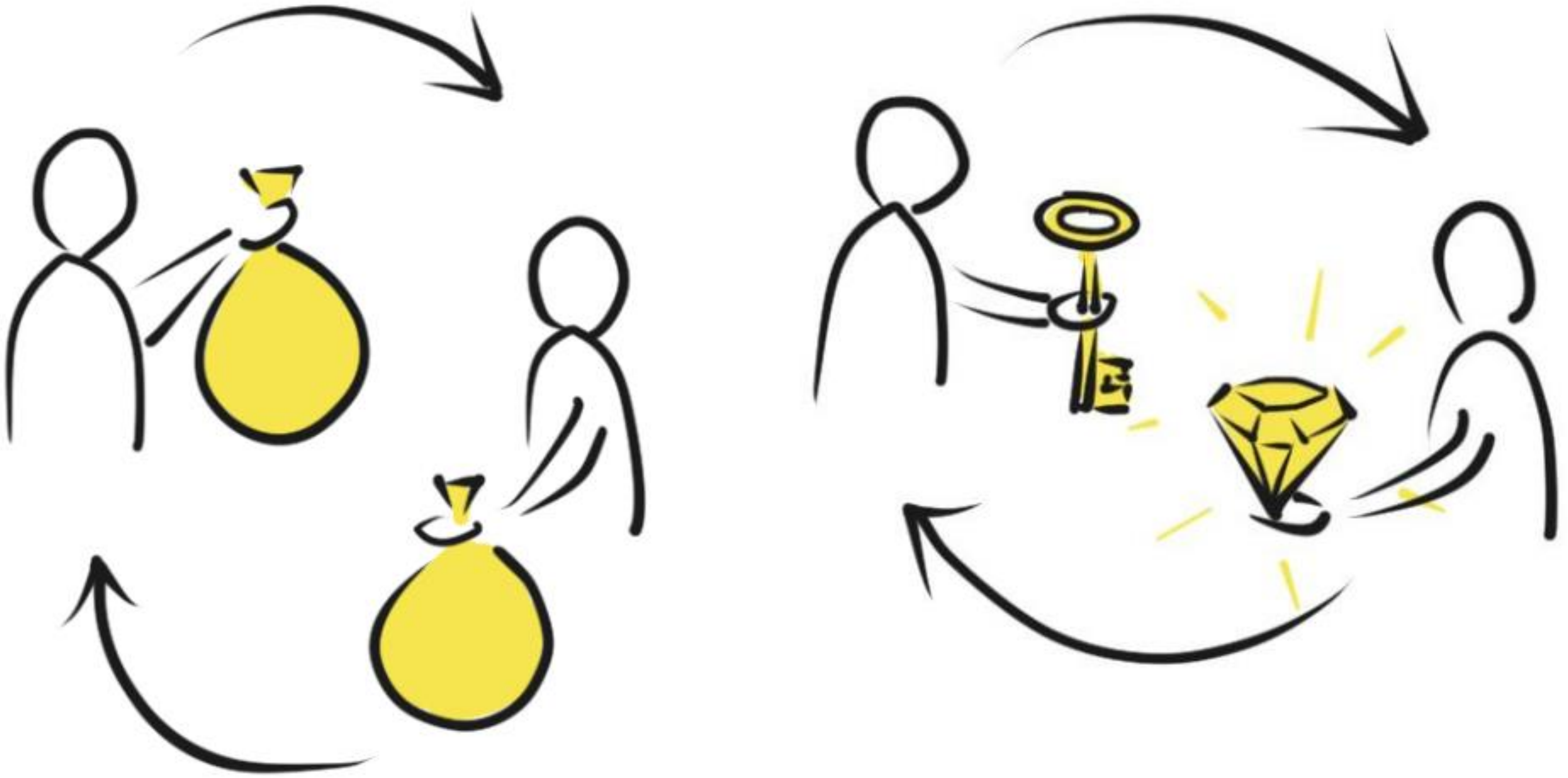
Data Coupling - Example

- 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.

Example

```
//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();
}
```

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



Decoupling method I

- 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.

Decoupling Method II

- 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.

Decoupling method III

- 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.

Decoupling Method IV

- 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.