37. What is coupling?

Brief summary: Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

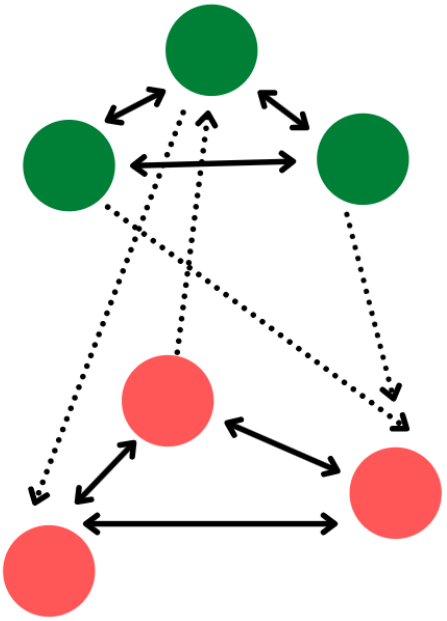
Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

Have you ever needed to introduce a small change in a class, but it actually forced you to also introduce changes in many other classes? Well, it seems like those classes were highly coupled with each other. It made them brittle - they got broken and needed to be fixed when a change was introduced somewhere else.

The high (or "strong") coupling means that one class knows too much about what is going on under the hood of another class.

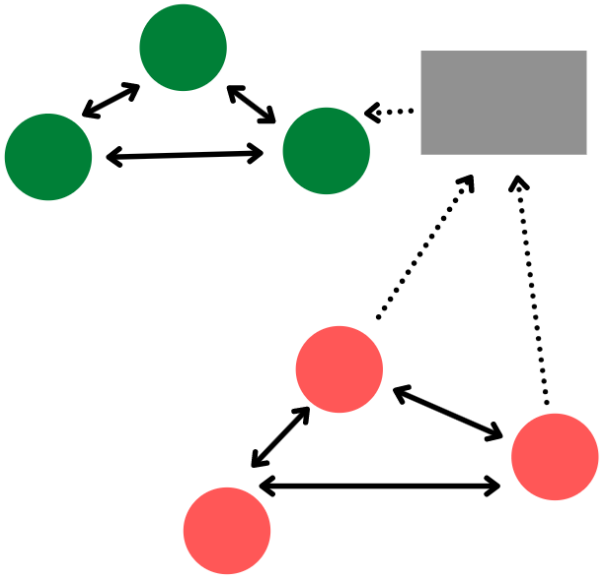
Low (or "loose") coupling is a desirable trait of classes.

This illustration shows the strong coupling between two classes:



Those classes, although separate, know way too much about each other, and they communicate directly between themselves. To reduce coupling, we should

introduce a simple, well-defined, and abstract interface, that will be the channel through which they communicate.



This way, if something changes in one of the classes, the other will not be affected, as long as the interface doesn't change. And remember, the implementation details change much more frequently than interfaces.

Let’s see some strongly-coupled classes.

```
public record Subscriber(string Email, bool IsPremium);

2 references
public class Subscribers
{
    public Subscriber[] Items;
}
1 reference
public class NewsletterSender
{
    private Subscribers _subscribers;

    0 references
    public NewsletterSender(Subscribers subscribers)
    {
        _subscribers = subscribers;
    }

    0 references
    public void SendTo(bool premiumSubscribersOnly)
    {
        for (int i = 0; i < _subscribers.Items.Length; i++)
        {
            if(!premiumSubscribersOnly ||
                _subscribers.Items[i].IsPremium)
            {
                Console.WriteLine(
                    $"Newsletter sent to " +
                    $"{_subscribers.Items[i].Email}");
            }
        }
    }
}
```

At first glance, it may look all right. But notice how the SendTo method (and thus the whole NewsletterSender class) depends on implementation details of the Subscribers class. It is not only aware that it holds a very concrete type of collection (an array) but it could even modify its elements. Let’s see what would happen if I wanted to change the collection that the Subscribers class use from an array to HashSet:

```
public class Subscribers
{
    4 references
    public HashSet<Subscriber> Items { get; }
    0 references
    public Subscribers(HashSet<Subscriber> items) => Items = items;
}
```

The SendTo method breaks:

```
public void SendTo(bool premiumSubscribersOnly)
{
    for (int i = 0; i < _subscribers.Items.Length; i++)
    {
        if(!premiumSubscribersOnly ||
            _subscribers.Items[i].IsPremium)
        {
            Console.WriteLine(
                $"Newsletter sent to " +
                $"{_subscribers.Items[i].Email}");
        }
    }
}
```

I changed an **implementation detail** in the Subscribers class and it **shouldn't affect** any other classes. It did, which proves that our code is brittle.

Let's fix it. The Subscribers class should only expose an abstract collection of items - let's make it IEnumerable. The consumers of this class don't need to know whether is an array, a HashSet, or anything else:

```
public class Subscribers
{
    3 references
    public IEnumerable<Subscriber> Items => _items;
    2 references
    private HashSet<Subscriber> _items { get; }
    0 references
    public Subscribers(HashSet<Subscriber> items) => _items = items;
}
```

Now, let's adjust the code in the NewsletterSender class:

```
public void SendTo(bool premiumSubscribersOnly)
{
    foreach(var subscriber in _subscribers.Items.Where(s =>
        !premiumSubscribersOnly || s.IsPremium))
    {
        Console.WriteLine(
                $"Newsletter sent to " +
                $"{subscriber.Email}");
    }
}
```

Great. Now the NewsletterSender class is not aware of any implementation details of other classes. As far as it's concerned, the Subscribers class only provides a collection that can be enumerated. Whether it's an array, a List, or anything else is irrelevant, and can change without the NewsletterSender class even knowing.

You can recognize high coupling by observing the following:

- One type uses another type directly, without having any abstraction in between.
- Even a small change in a class leads to a cascade of changes all around the project.
- Classes are not independent. To make some object work, we need to set up some state in other objects. This is particularly visible in testing - when setting up a test, you must do a lot of work on other objects than the one that you actually want to test.

The question is, what can we do when we observe that our code is tightly coupled? The best solution is to simply reduce the direct connections between concrete types.

Let me illustrate it like this: let's say I want to go for a trip by the sea. If I am tightly coupled with the Car class and I only accept it as the mean of transportation. It may mean that my weekend will be ruined if my car breaks down or, for example, my driving license expires. On the other hand, if I would only depend on some IMeanOfTransport service, it would mean that I am not coupled with any concrete type implementing it, and I could easily switch whatever I use to a plane or a train. And my weekend would be saved. I wouldn't depend on the technical details of the mean of transport. I would only need to be provided with something I can use to travel, and what it is or how it works under the hood, I don't really care as long as it takes me to the beach.

As you can see, to reduce coupling we should have different types communicate over interfaces, not directly. If you know the Dependency Inversion Principle from the SOLID principles, you can see that its main purpose is reducing coupling: according to this principle, types should not depend on concrete implementations, but rather on abstractions. By following this principle, we remove the direct way of communication between classes, making them more independent from each other.

The perfect classes and modules should be highly cohesive and loosely coupled.

Let's summarize. Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

Bonus questions:

- "**How to recognize strongly couples types?**"
  One type uses another type directly, without having any abstraction in between. We often recognize strong coupling the hard way: when we see that even a small change in a class leads to a cascade of changes all around the project. It proves that the types are not independent.

- "**Which of the SOLID principles allow us to reduce coupling?**"
  The Dependency Inversion Principle, which says that classes shouldn't depend on concrete implementations, but rather on abstractions. When following this principle we remove the direct way of communication between classes, making them more independent from each other.