There are two terms called coupling and cohesion. These concepts can help you understand more about your code and how your actions affect the code. They define relations between and within classes. Knowing them can help you create a better structure for your code, making it more flexible and purposeful. Let's learn about them; starting with coupling.

**Coupling**

**Coupling** is the degree to which software components depend on each other. There are two basic levels of coupling: **tight** and **loose**.

Loose coupling means less interdependence between classes, whereas in tight coupling it's the opposite. The difference is that in tight coupling, one class strongly depends on another and knows too much about its methods. But, in loose coupling one class knows little about object creation and methods in another class. This makes these classes less interdependent on each other.

The simplest way to loosen coupling is by implementing an interface in your code. There are also other ways to loosen coupling, like dependency injection.

**Tight coupling example**

Let's assume you're building a robot. A central part of this robot is an engine that cannot be easily replaced. You would have to deconstruct your robot and tweak it to use your new engine. It's a tightly coupled dependency which you can depict as:

class Engine is

method run() is

class Robot is

Engine eng = new Engine()

eng.run()

Here, the Robot class is dependent on Engine and needs to create an instance of it. Your dependent class knows too much about methods in this case, which tightens this coupling. Any changes in Engine will force you to change Robot as well. So, if the engine doesn't work, the whole robot will stop functioning.

**Loose coupling example**

Let's look at an example of loose coupling. Here, you can build a robot with a default paint scheme. It can be either silver or gold. Either way, you have to change your robot from scratch. It's a loosely coupled dependency.

interface Paint is

method paint()

class Silver implements Paint is

method paint() is

print("silver")

class Gold implements Paint is

method paint() is

print("gold")

class Robot is

Paint col = new Gold()

col.paint()

The paint classes are exposed to the Robot class through the Paint interface. In this instance, you can inject paint class methods through the interface. This means your Robot class knows little about methods in Silver and Gold classes. The Robot class can also use them separately. This means you can use other classes in case one doesn't work.

**Cohesion**

**Cohesion** is a representation of class functions. It is a measure of how closely related the responsibilities of a component are. Low cohesion suggests that a component is trying to perform too many unrelated tasks, making it harder to maintain and understand. In short, the more a class is able to do, the lesser its cohesion. In contrast, high cohesion indicates that a component performs a single task or a group of related tasks, leading to more understandable and maintainable code.

For example, when you're working on an application that represents a robot building factory. You can make a single class called RobotFactory. This class contains methods for robot creation, factory logistics, staff management, and much more.

class RobotFactory is

method createRobots() is ...

method maintainLogistics() is ...

method manage() is ...

This is an example of low cohesion where a single class has many different purposes. It is difficult to reuse and test classes like this. But, if you try to distribute methods with different purposes in different classes, you can increase your cohesion.

class BuildDepart is

method createRobots() is

...

class Logistics is

method maintainLogistics() is

...

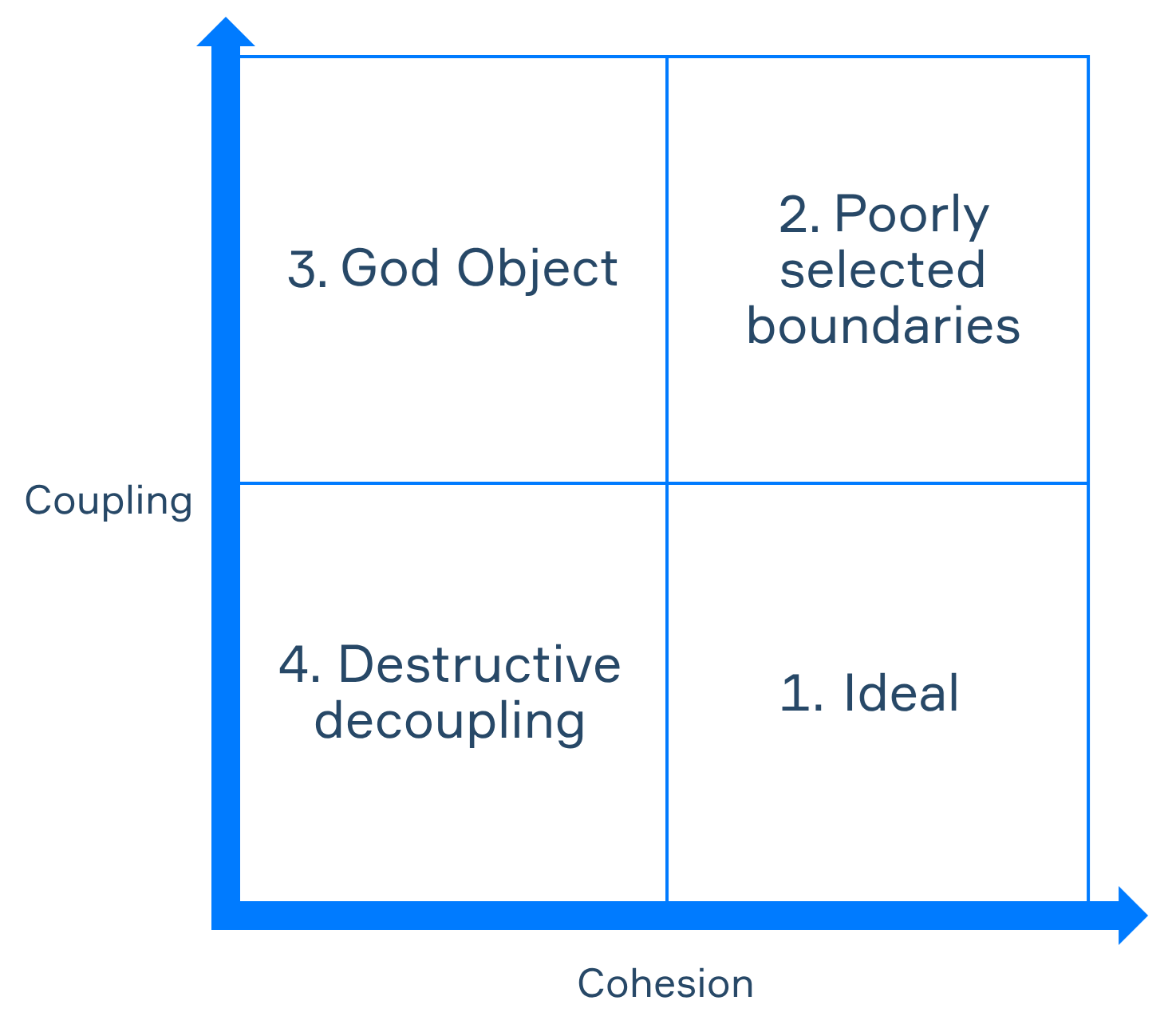
class Management is

method manage() is

...

**Working with cohesion and coupling**

The chart below illustrates the possible consequences based on the degree of cohesion and coupling.



Ideally, your objects should have low coupling and high cohesion. In other circumstances, you'll probably neglect some basic design principles like the single responsibility principle.

There are also situations when you completely forget about other elements while trying to achieve the ideal state, i.e. low coupling and high cohesion. This can lead to one of the following:

* Making a god object (low cohesion, high coupling) — one object performs all functions.
* Poorly selected boundaries (high cohesion, high coupling) — fragments of code have boundaries, but they also contain classes that shouldn't be included in them.
* Destructive decoupling (low cohesion, low coupling) — parts of code have the lowest coupling possible, but that leads to a lack of focus on your code.

When trying to lower your coupling, you shouldn't forget about increasing your cohesion and vice versa.

**Conclusion**

Let's summarize what we know about coupling and cohesion.

* You should try to achieve loose coupling. Loose coupling makes parts of your code less interdependent on each other. The easiest way you can achieve it is with interfaces and dependency injection.
* Parts of your code should have high cohesion, so that they can be more focused. This will make them more reusable and easier to test.

Ideally, you should try to maintain the highest cohesion and lowest coupling levels. It will make your code more readable and accessible.

Start practicing

So now the difference comes down to **"is-a" vs "can-do" vs "has-a":**

1. **Inheritance (is-a):**  
   Use class inheritance when something *is a* more specific version of another thing.  
   Like Engineer is a Person.
2. **Interface (can-do):**  
   Use interfaces to express **capabilities or behavior**, not identity.  
   Like Flyable or Serializable. So Engineer implements Flyable means *Engineer can fly*.
3. **Dependency (has-a):**  
   Use composition (and possibly dependency injection) when something *has* or *uses* another thing.  
   Like Robot has a Paint.

Match the type of coupling or cohesion with its description.

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose.

 **Tight coupling** → one class strongly depends on another and knows too much about its methods

 **Loose coupling** → one class knows little about object creation and methods in another

 **High cohesion** → class has a clear purpose

 **Low cohesion** → class has many purposes which are different from each other

**cohesion means how well the *parts of a class* work together toward *one clear goal*.**

So:

* **High cohesion**: all the methods and data in a class are tightly focused on doing one thing well. They're "connected" because they're working toward the *same purpose*.
* **Low cohesion**: the class is all over the place—maybe one method handles logging, another sends emails, another manages invoices. They're *not* connected in purpose.

What is the combination of low cohesion and tight coupling called?

God object

What is the ideal combination of coupling and cohesion?

High cohesion, loose coupling

Which statements about loose coupling are correct?

interdependent = dependent on other things, low cohesion = low focussed, loosely coupled = not dependent on other things

In loosely coupled code, one class knows little about object creation in another

You need to achieve loose coupling to make your objects less interdependent

Which coupling do these classes have?

class Car is

method drive()

class Ferrari is

Car ferrari = new Car()

ferrari.drive()

​Tight coupling

Which chunk of pseudocode has higher cohesion?

Example 1:

class MakeCar is

method makeWheels() is

return wheels

method makeEngine() is

return engine

method installEngine()

...

method installWheels()

...

Example 2:

class Wheels is

constructor of Wheels is ...

method installWheels() is ...

class Engine is

constructor of Engine is ...

method installEngine() is ...

class makeCar is

Engine eng = new Engine()

Wheels wh = new Wheels()

eng.installEngine()

wh.installWheels()

**Answer: Example 2 has higher cohesion.**

### Why:

* In **Example 2**, each class focuses on **one specific responsibility**:
  + Wheels deals only with wheels.
  + Engine deals only with the engine.
  + MakeCar coordinates the process. → This follows the **Single Responsibility Principle** → **High cohesion**.
* In **Example 1**, the MakeCar class does everything: it makes and installs both wheels and the engine. It mixes different responsibilities, leading to **lower cohesion**.

So, **Example 2 = better structure = higher cohesion**.

You are working on an application, which simulates car creation and driving. This app constructs a car from different parts and gives it to a driver. Here's a part of the application in pseudocode:

class Engine is ...

class Wheel is ...

class Car is

Engine eng = new Engine()

Wheel whl = new Wheel()

constructor Car() is

this.eng = eng

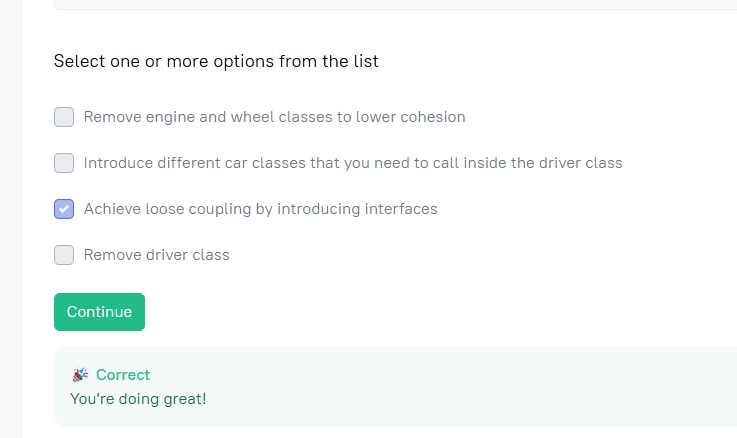
this.whl = whl

method drive() is ...

class Driver is

Car car = new Car()

car.drive()

****