Single Responsibility Principle

If you've ever heard of *SOLID* programming principles, the first letter in the acronym stands for the **Single Responsibility Principle** (SRP).

The reason is that we can concentrate better on important things instead of constantly switching our attention from one object to another.

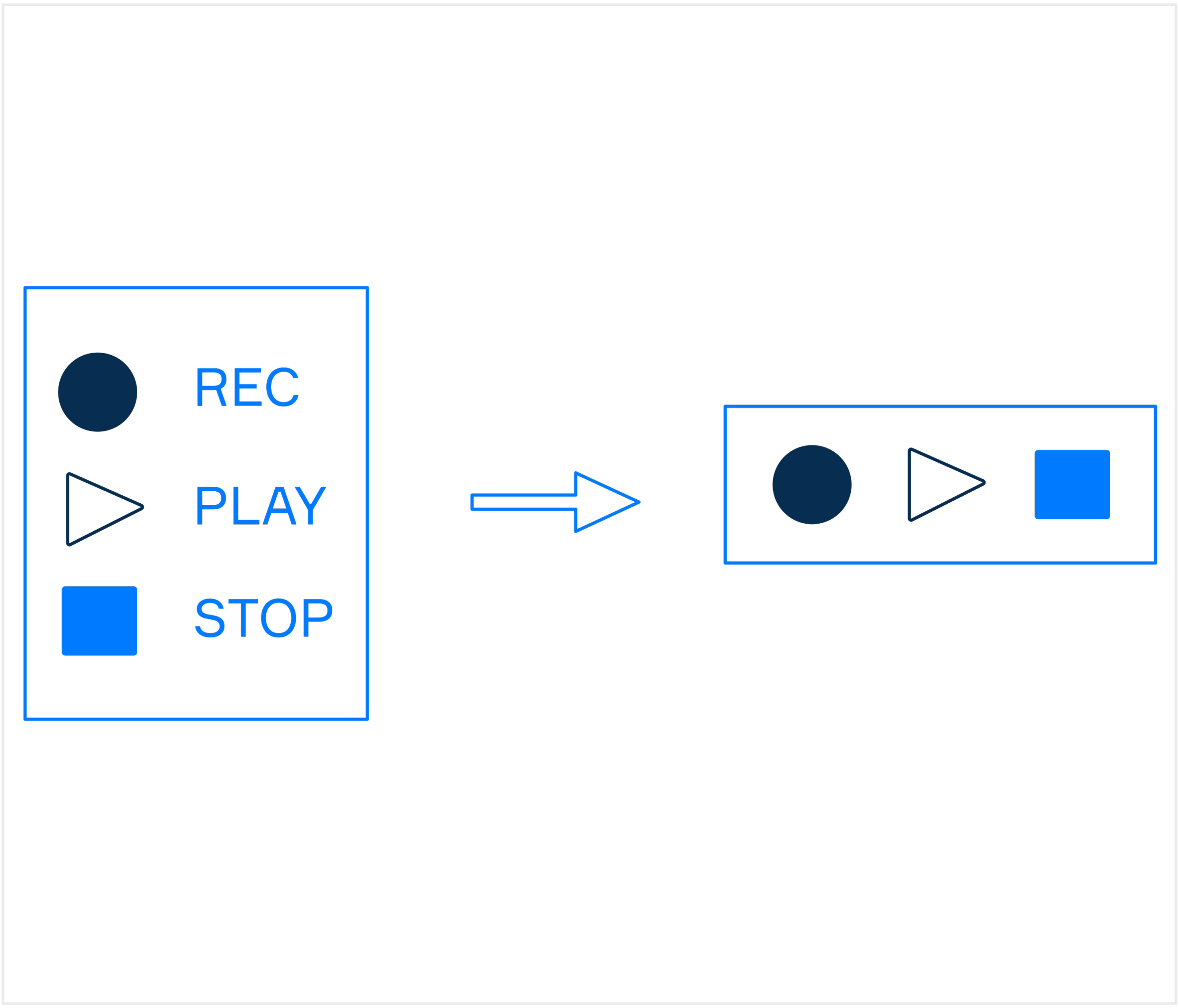
We use our smartphones to play music, send messages, and browse the internet. All this, often at the same time, but should our programs multitask? Does this idea contradict the "*one thing at a time*" rule?

To answer the first question about multitasking: *yes*. Your program should do what it has to, no matter how many parallel tasks there are. It seems reasonable because that's the way we create a complex application.

The answer to the second question is *no*. Let's take any application of your choice as an example. You expect some action from each part of it and you don't want these parts to do more than they have to.

How does it apply to the development process?

The program consists of some programming units like modules, classes, and functions. The Single Responsibility Principle requires your programming unit to have only one reason to change (i.e. a single responsibility). If you, as a developer, change the menu of a program, it should not affect the way other parts of the program work.



God object

Imagine we're working on a new application for smartphones. It has such functions as an alarm clock, reminders, and stopwatch. Because they are all related to time, they can be united in one class.

When we add the possibility to choose a ringtone for the alarm clock, can we use this for the stopwatch too? And if we add the possibility to save the top 5 results from the stopwatch, how can we use this function for the reminders? In trying to gather similar functions together we instead create a **God object**.

A **God object** is an object that *knows too much* or *does too much*.

In our case, it's better to isolate these three functions from each other, so we can update them independently without polluting the code with unnecessary details. Each part will be responsible for a specific thing and won't know about what the others are doing. This design is more likely to keep the code clean and clear.

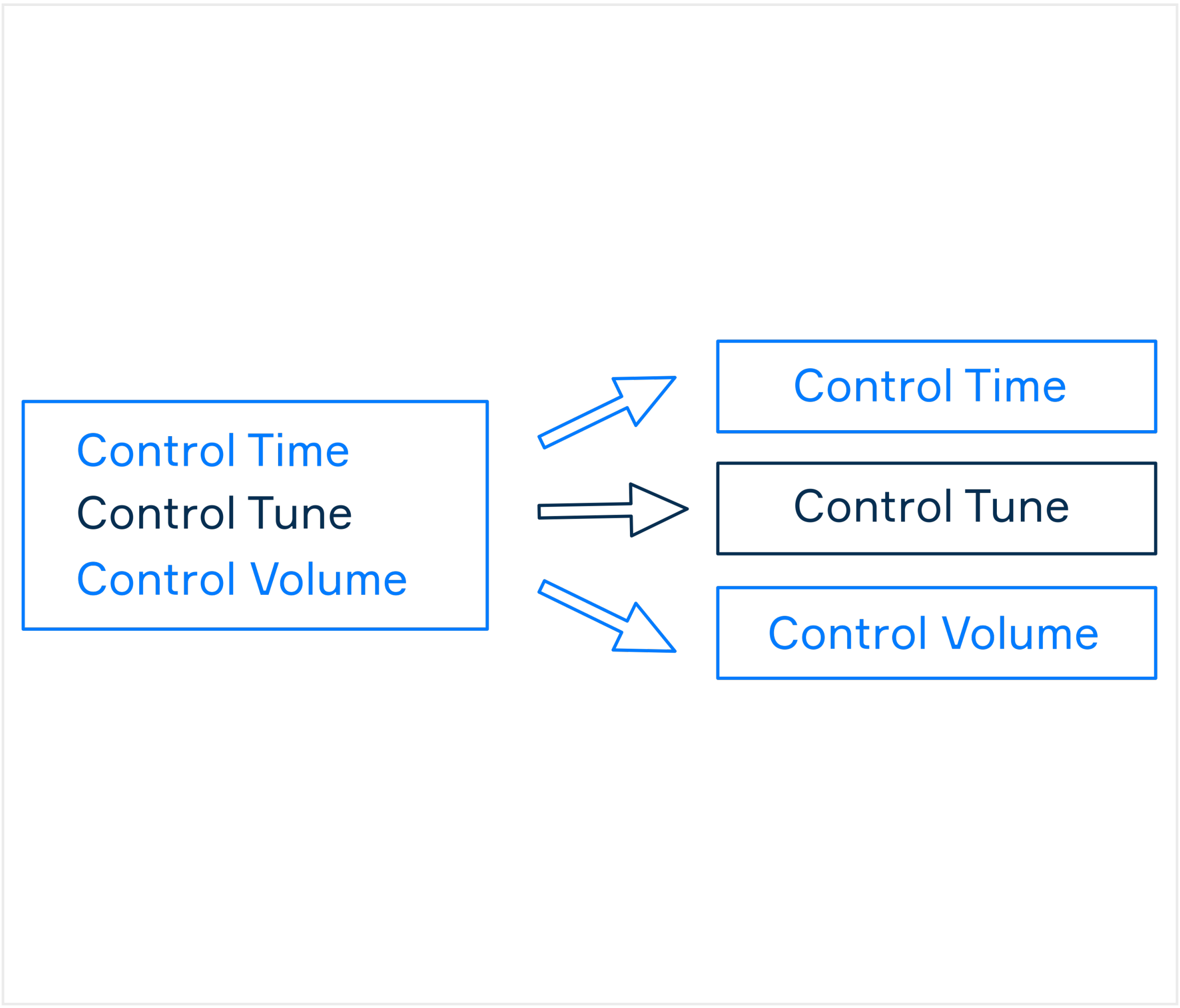
Responsibility boundaries

We decide to add an option to choose a ringtone for the alarm clock; we roll up the sleeves and start coding. We add methods LIST_TUNES, SET_TUNE, and PLAY_TUNE to the ALARM_CLOCK class.

It looks like we have a decent application. Users start downloading it, and in reviews they mention that it would be great to make a tune go louder when the alarm is set off. We add the method SET_ENHANCING_MODE, but then notice that we mixed up methods for different responsibilities in the same class: setting tunes, setting volume. This time our class has at least three reasons to change:

- time methods
- volume methods
- tune methods

Do we want the update of the volume methods to spoil the work of the time methods? Of course not. We should split the class into several parts and delegate responsibilities. You can ask, what will the ALARM_CLOCK do then? The answer is simple: it will call other classes and orchestrate their work depending on a user request.
Good news! We can also reuse TUNE and VOLUME classes for the REMINDER.



To draw boundaries for responsibilities, ask yourself: are these the same responsibilities for a user or not? If they are the same, bind the methods together. If they are not, divide them to create classes that have only *one reason to change*.

The core of the principle is not about breaking the code into small isolated pieces. You gather what belongs to the same responsibility in one place, and then separate units with different responsibilities from each other.

Read more on this topic in Testing Python Code 101 with PyTest and PyCharm on Hyperskill Blog.

Conclusion
The Single Responsibility Principle (SRP) prevents us from making God objects and encourages us to create classes with only one reason to change. Instead of gathering all methods in one place, we can separate methods with different responsibilities and use them anywhere we need.