

SCHEDULER PATTERN

Design Defects & Restructuring

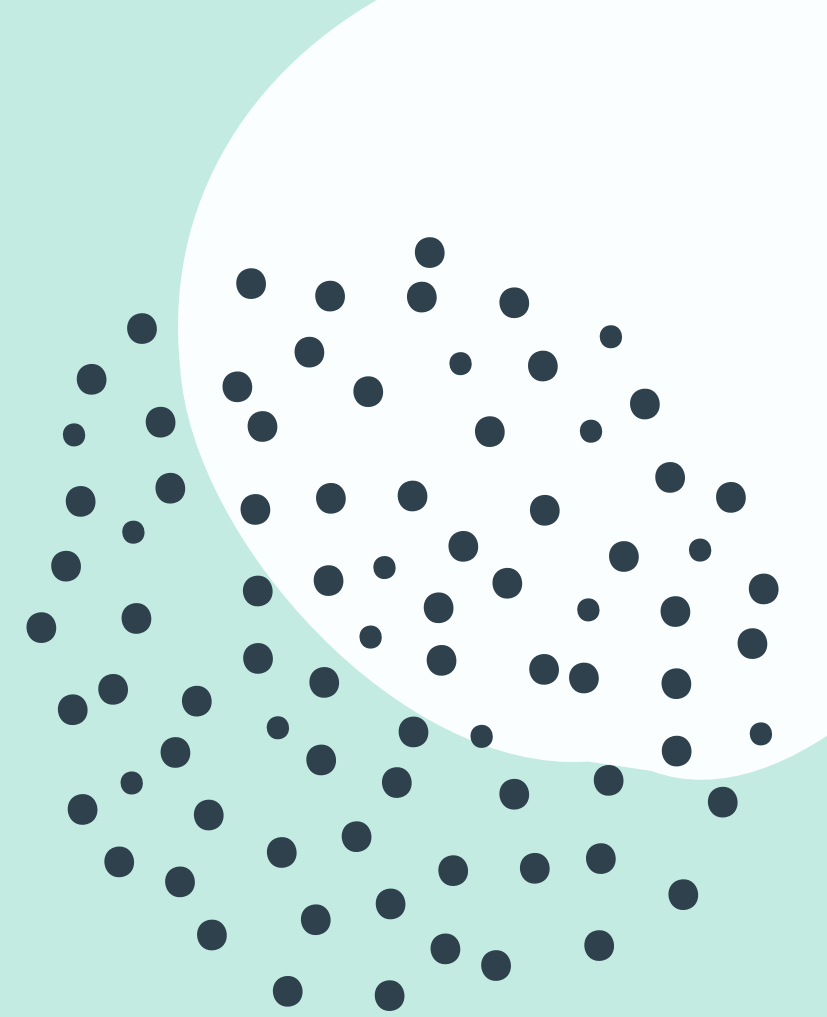


Team Members

Sarim Sohail 19k-0153

Mubashira Khan 19k-0239

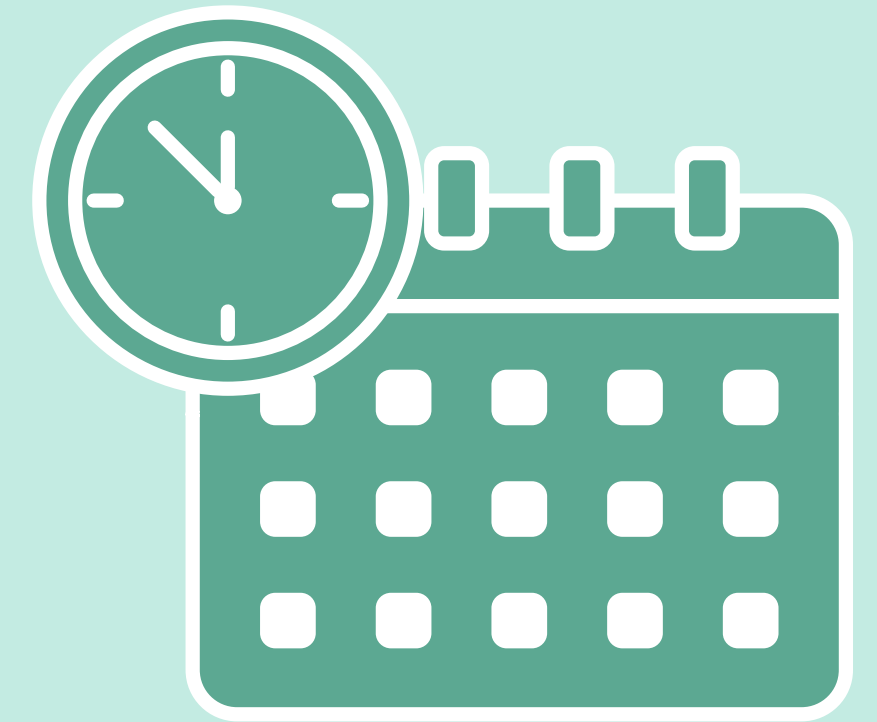
Hassan Jamil 19k-1292



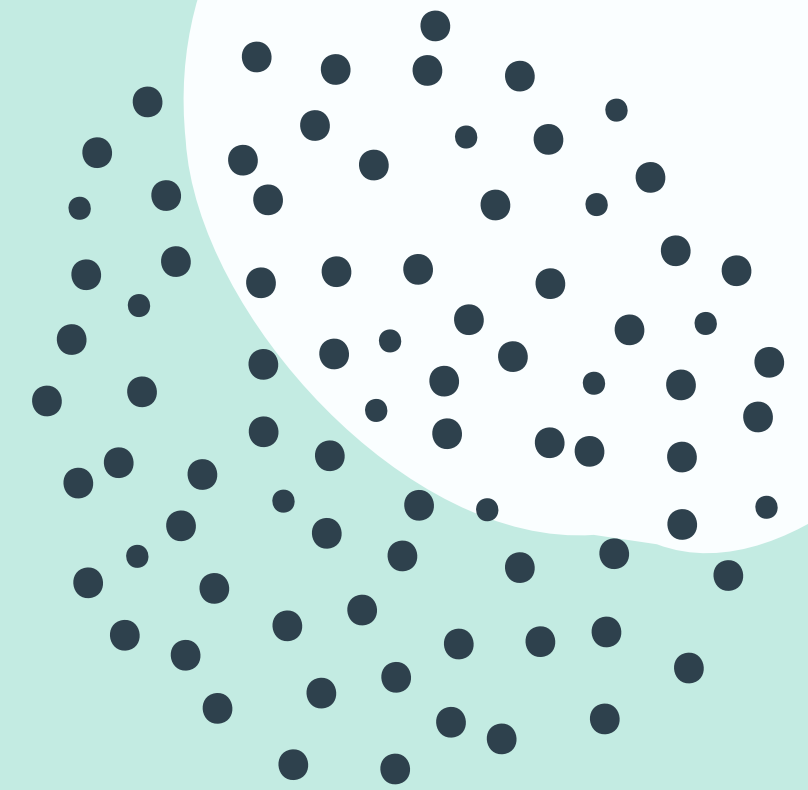


- The scheduler design pattern is a software design pattern that is used to schedule the execution of tasks.
- This can be useful in a variety of contexts, such as scheduling the execution of background tasks, scheduling the execution of tasks at a later time, or scheduling the execution of tasks at regular intervals.
- It falls under the category of Concurrency patterns.

Introduction

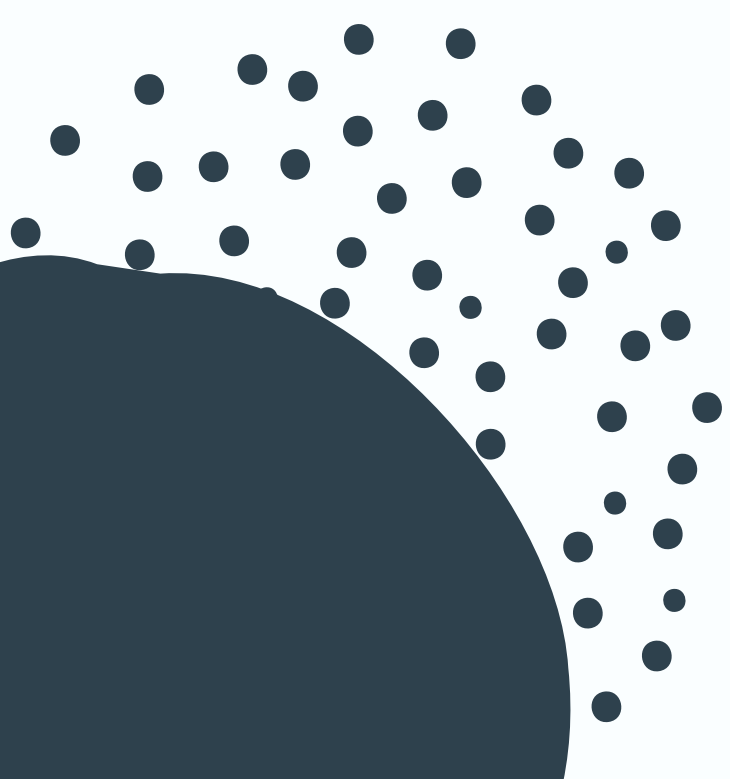


Background



The scheduler design pattern is a type of software design pattern that falls into the category of creational patterns. Creational patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

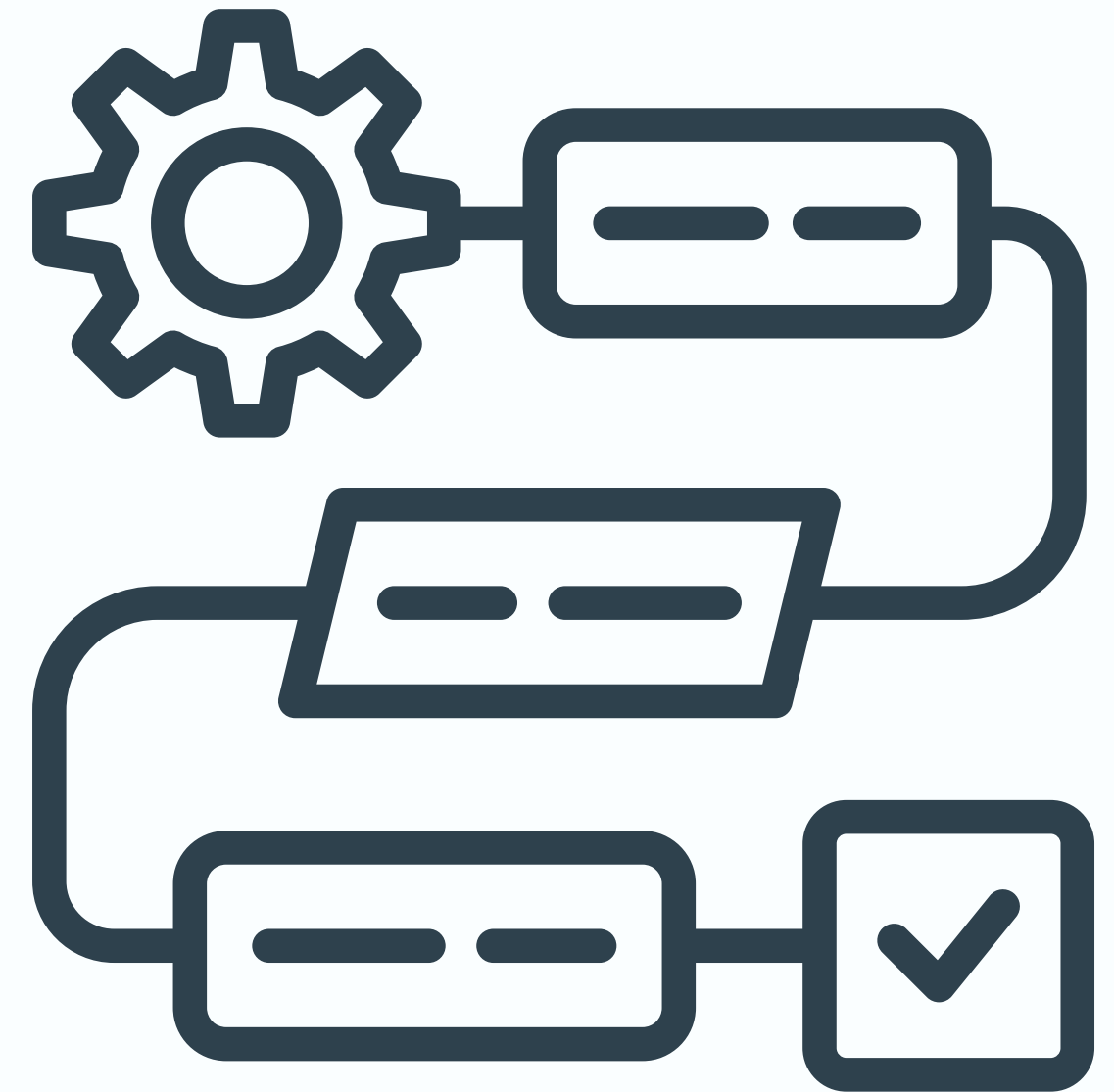
Motivation



In general, the scheduler design pattern aims to decouple the scheduling of tasks from the execution of those tasks, allowing for more flexibility and modularity in the design of the system. This can make it easier to add, modify, or remove tasks from the schedule, as well as to change the schedule itself.

Methodology

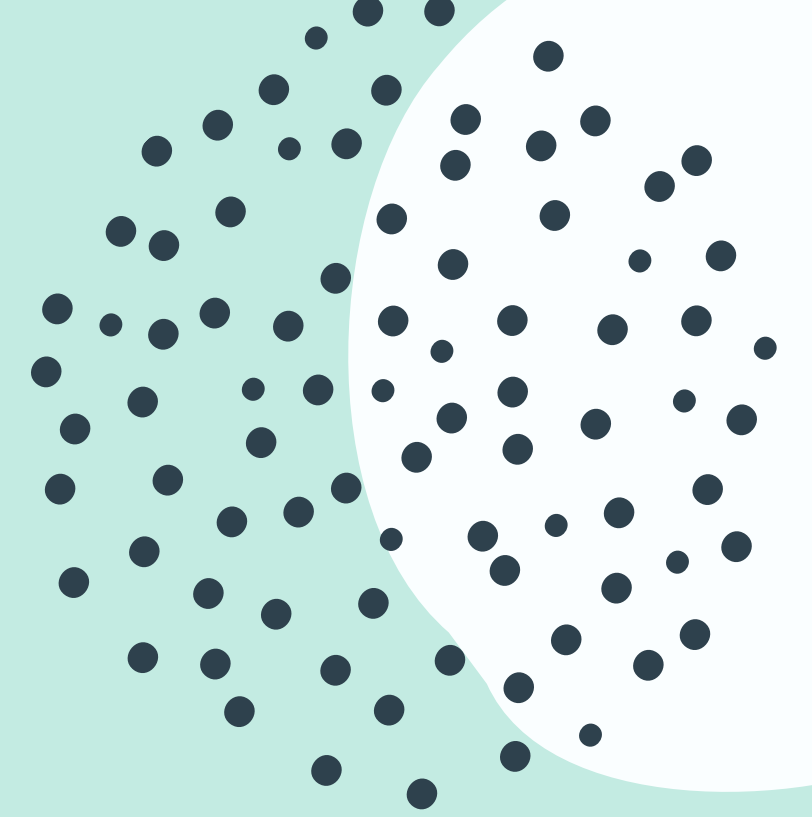
- There are several different approaches to implementing a scheduler, depending on the specific requirements of the task.
- Some common approaches include using a timer or a separate thread to trigger the execution of the task, or using a dedicated scheduling system such as a job scheduler or a message queue.



SCHEDULER.CS

```
1 using System;
2 using System.Collections.Generic;
3 using System.Threading;
4
5 class Scheduler {
6     private readonly Timer timer;
7     private readonly Dictionary<Action, int> tasks;
8
9     public Scheduler() {
10         tasks = new Dictionary<Action, int>();
11         timer = new Timer(Tick, null, Timeout.Infinite, Timeout.Infinite);
12     }
13
14     public void AddTask(Action task, int interval) {
15         tasks[task] = interval;
16         timer.Change(0, Timeout.Infinite);
17     }
18
19     public void RemoveTask(Action task) {
20         tasks.Remove(task);
21     }
22
23     private void Tick(object state) {
24         var now = DateTime.Now;
25         foreach (var task in tasks) {
26             if (now.Ticks % task.Value == 0) {
27                 task.Key();
28             }
29         }
30         timer.Change(1000, Timeout.Infinite);
31     }
32 }
33
34 class Program {
35     static void Main() {
36         var scheduler = new Scheduler();
37
38         // Add a task that prints "Hello, world!" every 2 seconds
39         scheduler.AddTask(() => Console.WriteLine("Hello, world!"), 2000);
40
41         // Wait for 10 seconds
42         Thread.Sleep(10000);
43     }
44 }
```

Code Explained



This scheduler uses a Timer object to trigger the execution of tasks at regular intervals.

The `AddTask` method adds a new task to the schedule, and the `RemoveTask` method removes a task from the schedule.

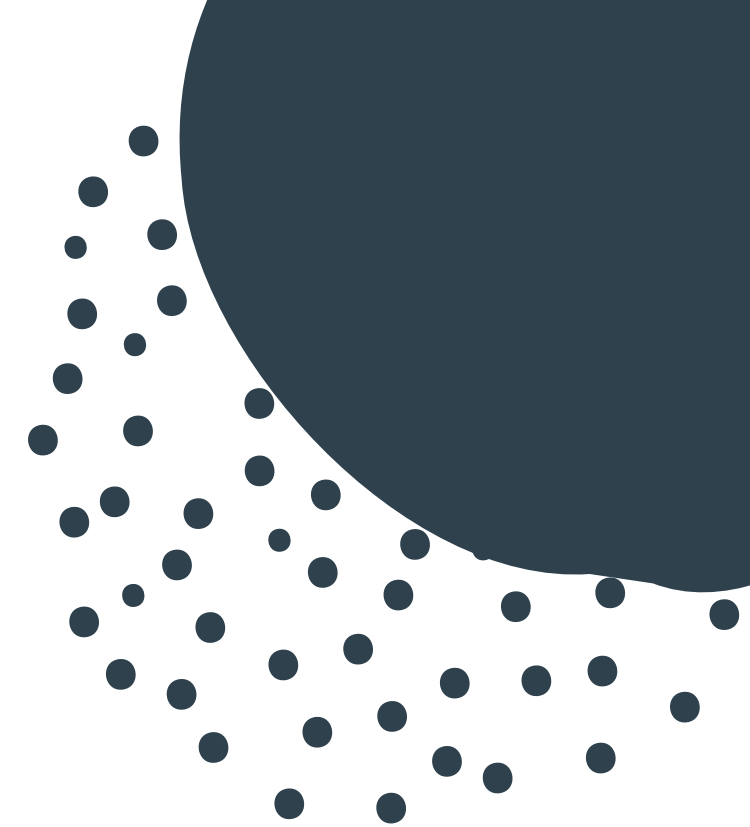
The `Tick` method is called by the timer and checks the schedule of tasks to see which tasks are due to be run. It then calls the `Key` method of each due task, which is the task itself.

The timer is set to run every 1 second, and the interval of each task is specified in ticks.

SCHEDULER.PY


```
1  import time
2
3  class Scheduler:
4      def __init__(self):
5          self.tasks = []
6
7      def add_task(self, task, interval):
8          self.tasks.append((task, interval))
9
10     def run(self):
11         while True:
12             for task, interval in self.tasks:
13                 task()
14                 time.sleep(interval)
15
16 # Example usage
17 def say_hello():
18     print("Hello!")
19
20 scheduler = Scheduler()
21 scheduler.add_task(say_hello, interval=5)
22 scheduler.run()
```

Code Explained



This scheduler runs a loop in which it executes each task in the tasks list and then sleeps for the specified interval before moving on to the next task. The `add_task` method allows you to add new tasks to the scheduler, and the `run` method begins the scheduling process.

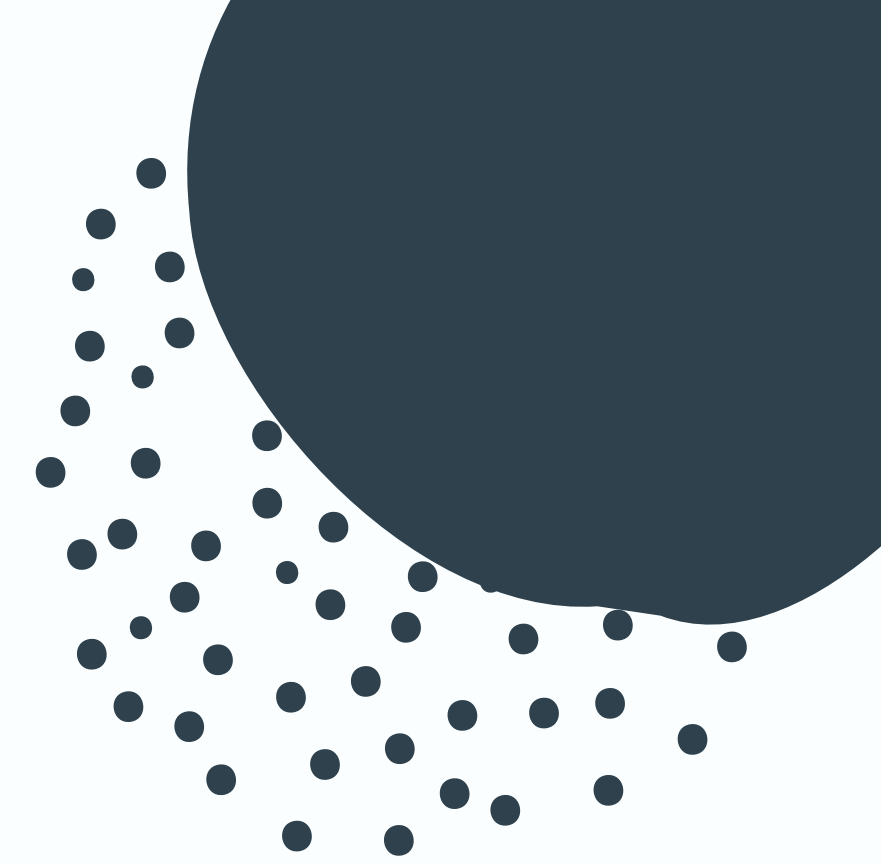
In this example, the `say_hello` function is added to the scheduler with an interval of 5 seconds, so it will be called every 5 seconds. You can add additional tasks to the scheduler by calling `add_task` with the desired task and interval.




One of the main benefits of using the scheduler design pattern is that it allows you to decouple the scheduling of tasks from the execution of those tasks.

This can make it easier to modify the schedule or add new tasks to the system, as you don't need to change the code that actually executes the tasks.

It also makes it easier to change the way that tasks are scheduled, such as switching to a different scheduling algorithm or using a different scheduling system.



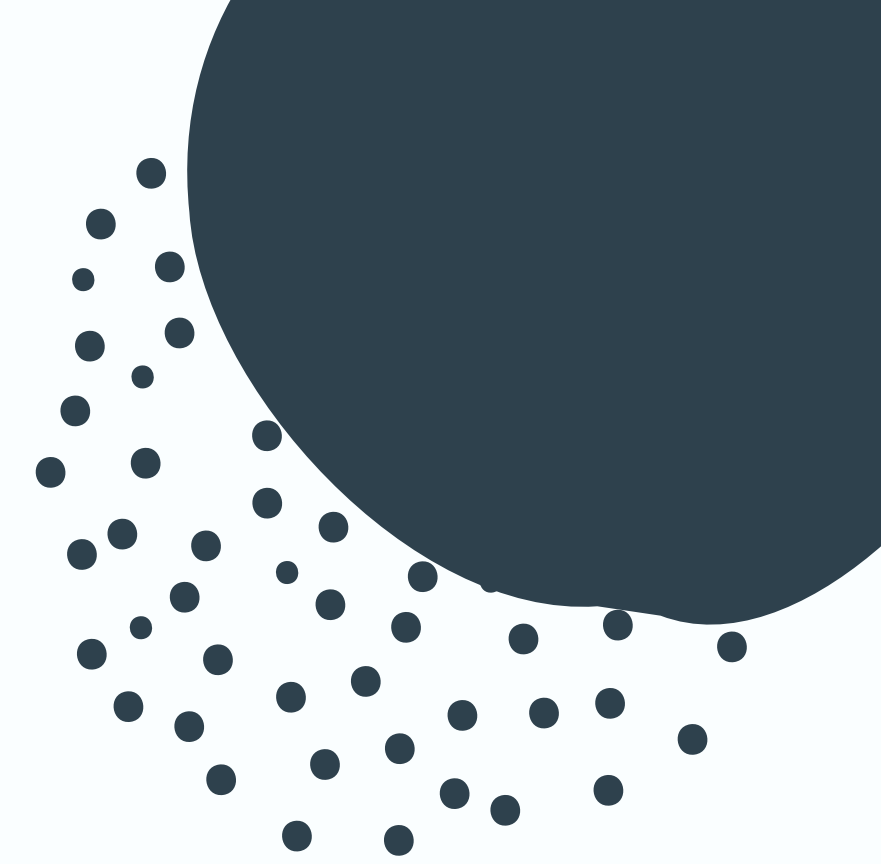
Advantages



Firstly, it can add complexity to the system, as you need to implement and maintain the scheduler itself.

Additionally, if the scheduler is not implemented correctly, it can cause problems such as missed or duplicate tasks, or delays in the execution of tasks.

Finally, if the scheduler relies on a separate thread or process to trigger the execution of tasks, it can add overhead to the system and potentially impact performance.



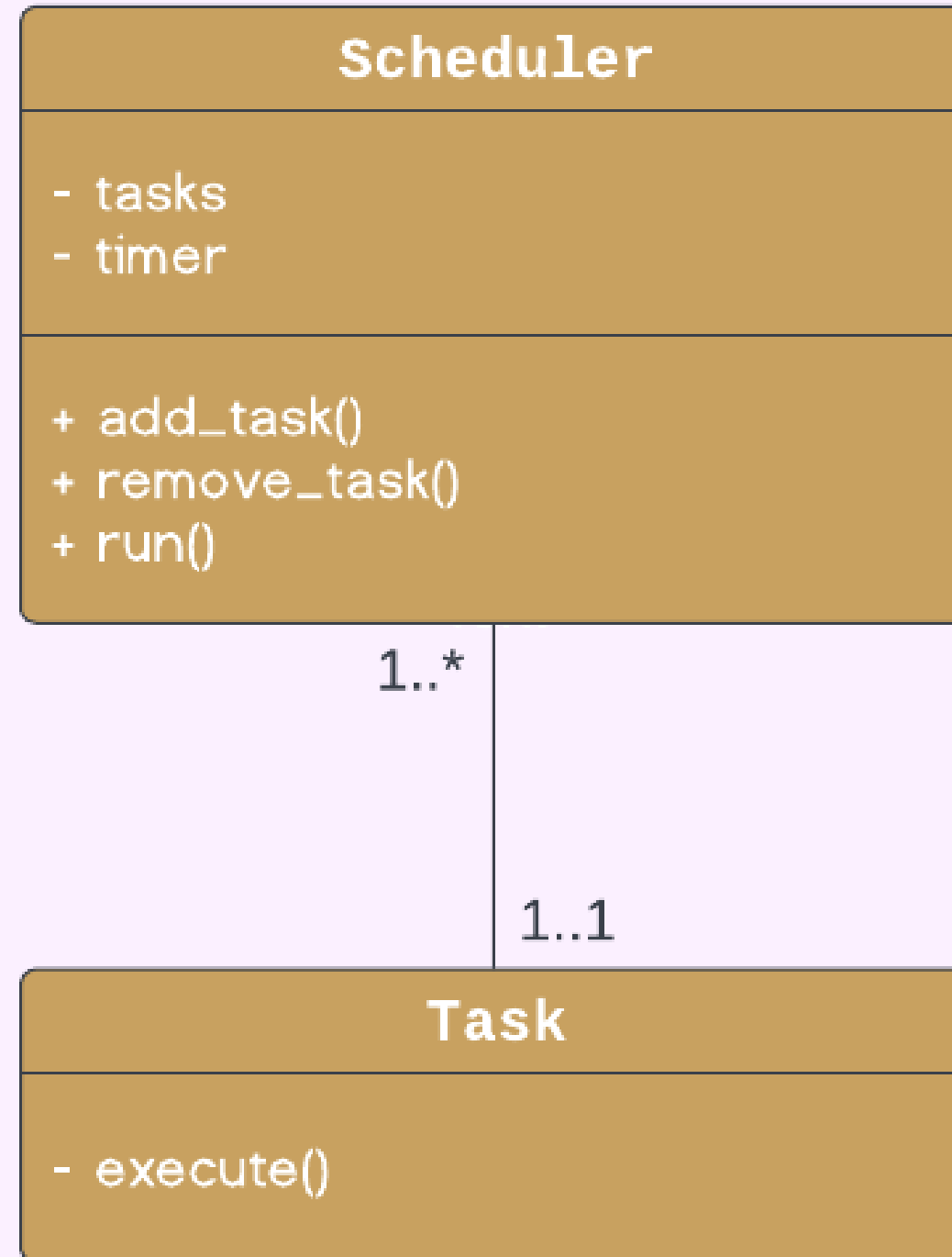
Disadvantages



Comparison with other patterns

- The scheduler design pattern can be further classified as a special type of factory pattern, which is a creational pattern that deals with the creation of objects without specifying the exact class to create.
- In the case of the scheduler design pattern, the factory is responsible for creating and scheduling the execution of tasks, without specifying the specific details of those tasks.

CLASS DIAGRAM





THE END.