Vladyslav Bogdantsev

vladproduction.java@gmail.com

*project link: https://github.com/vladproduction/My-Pet-Projects/tree/main/Tasker*

The Tasker project is a task management system designed to allow the scheduling and execution of various types of tasks. It supports immediate tasks, delayed tasks, scheduled tasks at a specific time, and repeated tasks that run at specified intervals.

Scheduling management systems

Tasker

**Contents**

[**Preconditions** 2](#_Toc186489602)

[**1)** **Requirements** 3](#_Toc186489603)

[**Functional requirements** 3](#_Toc186489604)

[**Non-Functional requirements** 3](#_Toc186489605)

[**2)** **Use Cases** 5](#_Toc186489606)

[Diagramming Use Case #1 (Create a Now Task) 5](#_Toc186489607)

[Diagramming Use Case #2 (Create a Delay Task) 6](#_Toc186489608)

[Diagramming Use Case #3 (Create a Repeated Task) 7](#_Toc186489609)

[Diagramming Use Case #4 (Create a Schedule Task) 8](#_Toc186489610)

[Diagramming Use Case #5 (Cancel a Scheduled Task) 9](#_Toc186489611)

[Diagramming Use Case #6 (View Task Execution Logs) 10](#_Toc186489612)

[**3)** **Objects, Classes and Relationships** 11](#_Toc186489613)

[**Objects** 11](#_Toc186489614)

[**Classes** 11](#_Toc186489615)

[**Relationships** 13](#_Toc186489616)

[**4)** **CRC Cards** 14](#_Toc186489617)

[**5)** **Class Diagrams** 16](#_Toc186489618)

[**6)** **Java code** 17](#_Toc186489619)

# **Preconditions**

This document presents a detailed design and implementation plan for the Task Management System, known as Tasker, which is designed to efficiently schedule and manage various types of tasks. It aims to provide users with a robust and flexible framework for immediate, delayed, scheduled, and repeated task execution.

The document outlines both functional and non-functional requirements to ensure that the system aligns with user needs and meets expected performance criteria. It describes various use cases that detail interactions between users and the system, facilitating a clear understanding of how users will engage with the Tasker.

Key components of the document include an identification of essential objects and classes, along with an overview of their relationships within the system architecture. A class diagram will be provided to visually represent these relationships and the overall structure of the system.

Moreover, optional Class-Responsibility-Collaboration (CRC) cards are included to clarify the responsibilities and collaborations of each class, enhancing understanding of the system design.

To support the path to development, this document also suggests Java code implementations for the defined classes, providing a clear and actionable framework for developers. By adhering to the guidelines and requirements outlined herein, the Tasker project aims to deliver a reliable, maintainable, and user-friendly task scheduling solution.

# **Requirements**

## **Functional requirements**

1. ***Task Types:***

* NowTask: Executes an action immediately.
* DelayTask: Executes an action after a specified delay.
* RepeatedTask: Executes an action at regular intervals, starting from a specified date.
* ScheduleTask: Executes an action at a specified future date and time.

1. ***Task Scheduling:***

* Users should be able to create tasks with a Runnable action.
* The system handles scheduling based on the task type: immediate, delayed, periodic, or scheduled for a specific time.

1. ***Task Processing:***

* Tasks must be processed asynchronously using a ScheduledExecutorService to avoid blocking operations.

1. ***Error Handling:***

* The system must handle exceptions during task execution and log errors appropriately using a logging framework (SLF4J).

1. ***Task Cancellation:***

* The ability to cancel scheduled tasks if needed.

1. ***Graceful Shutdown:***

* The system must support safe termination, allowing all scheduled tasks to complete before shut down.

## **Non-Functional requirements**

1. ***Performance:***

* The system should efficiently handle multiple tasks with a thread pool (ScheduledExecutorService) of size 10, reducing the likelihood of bottlenecks during task execution.

1. ***Scalability:***

* The design should allow for easy addition of new task types and processors without significant rework. The ability to handle an increasing number of tasks efficiently is important.

1. ***Reliability:***

* The system must ensure that tasks are executed as expected. The introduction of error handling helps maintain reliability despite task execution failures.

1. ***Maintainability:***

* The code is organized into separate classes for each task type and processor, following the single responsibility principle, which aids in easier maintenance and understanding of the codebase.

1. ***Usability:***

* The API allows users to create and manage tasks easily. However, additional enhancements could improve usability through helper methods or a builder pattern.

1. ***Error Handling:***

* The implementation includes error logging for task execution failures. The custom TaskProcessingException provides a clear way to handle and report issues.

# **Use Cases**

**Use Case 1: Create a Now Task**

***Description:*** A user creates a task that executes immediately.

***Actors:*** User

***Preconditions:*** User is authenticated and has access to the task management interface.

***Postconditions:*** A new Now task is created and executed immediately. The user is notified of successful execution.

***Main Flow:***

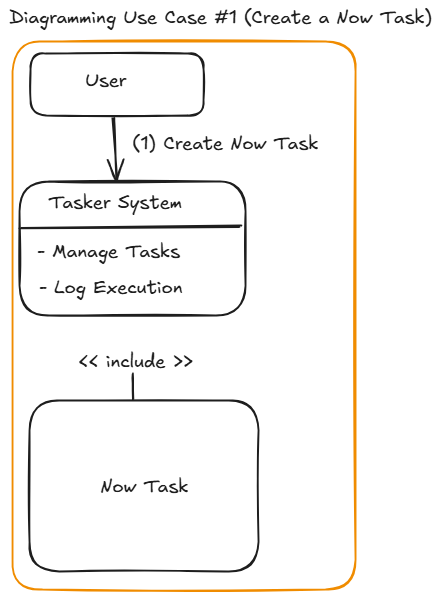
1. User selects the option to create a Now task.
2. User inputs the action (a Runnable).
3. The system creates the Now task and immediately executes the action.
4. The system logs the execution and notifies the user.

## Diagramming Use Case #1 (Create a Now Task)

Actors: User (The individual interacting with the system)

System: Tasker System (The system that processes the tasks)

Use Case: Create Now Task (The action performed by the user, leading to the immediate execution of a task)



**Use Case 2: Create a Delay Task**

***Description:*** A user creates a task that executes after a specified delay.

***Actors:*** User

***Preconditions:*** User is authenticated and has access to the task management interface.

***Postconditions:*** A new Delay task is created and scheduled for future execution.

***Main Flow:***

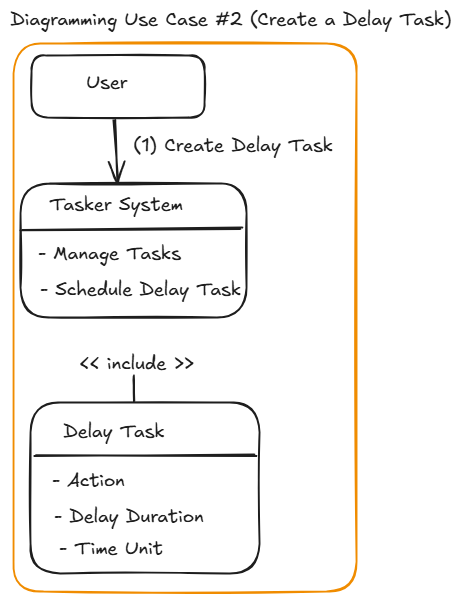
1. User selects the option to create a Delay task.
2. User inputs the action (a Runnable) and specifies the delay duration and time unit.
3. The system creates the Delay task.
4. The system schedules the task for execution after the specified delay.
5. The system logs the scheduling action and notifies the user.

## Diagramming Use Case #2 (Create a Delay Task)

Actors: User (The individual interacting with the system)

System: Tasker System (The system that processes the tasks)

Use Case: Create Delay Task (The action performed by the user to create a task that executes after a specified delay)



**Use Case 3: Create a Repeated Task**

***Description:*** A user creates a task that executes repeatedly at defined intervals.

***Actors:*** User

***Preconditions:*** User is authenticated and has access to the task management interface.

***Postconditions:*** A new Repeated task is created and scheduled for recurring execution.

***Main Flow:***

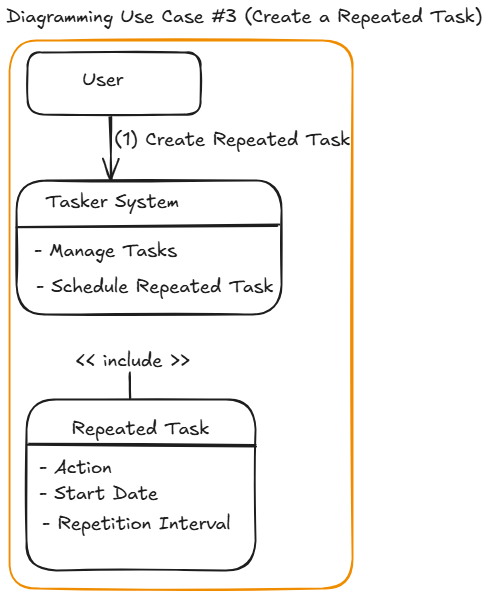
1. User selects the option to create a Repeated task.
2. User inputs the action (a Runnable), specifies the start date, and sets the repetition interval in seconds.
3. The system creates the Repeated task.
4. The system schedules the task for execution at the specified intervals.
5. The system logs the scheduling action and notifies the user.

## Diagramming Use Case #3 (Create a Repeated Task)

Actors: User (The individual who wants to create a repeated task)

System: Tasker System (The system that processes the tasks)

Use Case: Create Repeated Task (The action performed by the user to set a task that executes at regular intervals)



**Use Case 4: Create a Schedule Task**

***Description:*** A user creates a task that executes at a specific future date and time.

***Actors:*** User

***Preconditions:*** User is authenticated and has access to the task management interface.

***Postconditions:*** A new Schedule task is created and scheduled for future execution at the specified time.

***Main Flow:***

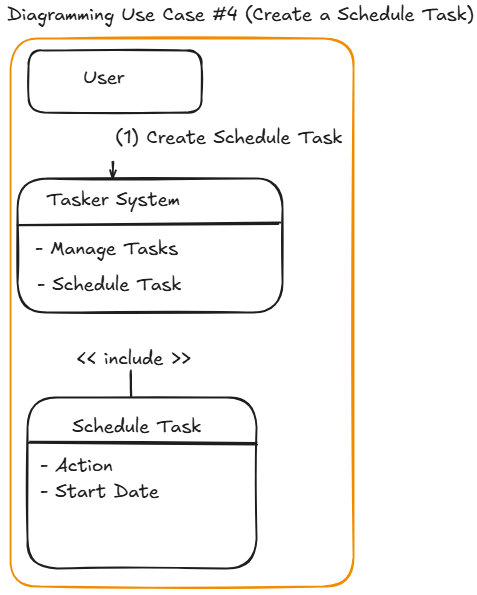
1. User selects the option to create a Schedule task.
2. User inputs the action (a Runnable) and specifies the exact start date and time.
3. The system creates the Schedule task.
4. The system schedules the task for execution at the specified time.
5. The system logs the scheduling action and notifies the user.

## Diagramming Use Case #4 (Create a Schedule Task)

Actors: User (The individual who wants to create a repeated task)

System: Tasker System (The system that processes the tasks)

Use Case: Create Schedule Task (The action performed by the user to create a task that executes at a specific date and time)



**Use Case 5: Cancel a Scheduled Task**

***Description:*** A user cancels an existing scheduled task.

***Actors:*** User

***Preconditions:*** User is authenticated, has access to the task management interface, and there are scheduled tasks available.

***Postconditions:*** The specified task is canceled, and it will no longer execute.

***Main Flow:***

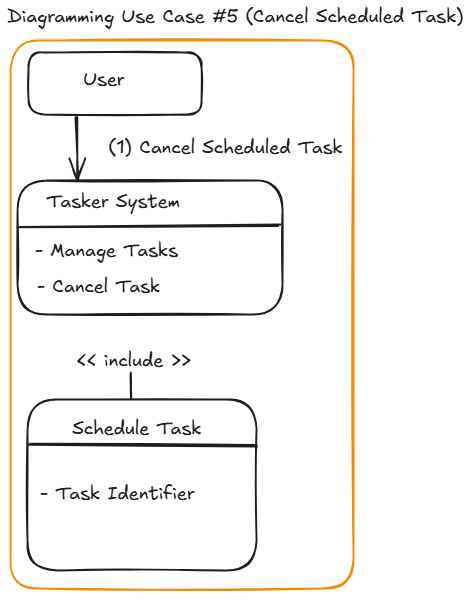
1. User selects the option to view scheduled tasks.
2. User selects a specific task to cancel.
3. The system cancels the task, if possible.
4. The system logs the cancellation and notifies the user of the successful cancellation.

## Diagramming Use Case #5 (Cancel a Scheduled Task)

Actors: User (The individual who wants to cancel an existing scheduled task)

System: Tasker System (The system responsible for managing tasks)

Use Case: Cancel a Scheduled Task (The action performed by the user to cancel a previously scheduled task)



**Use Case 6: View Task Execution Logs**

***Description:*** A user views the logs of previous task executions and any errors that occurred.

***Actors:*** User

***Preconditions:*** User is authenticated and has access to the log interface.

***Postconditions:*** The user can view a list of task executions, including success and error logs.

***Main Flow:***

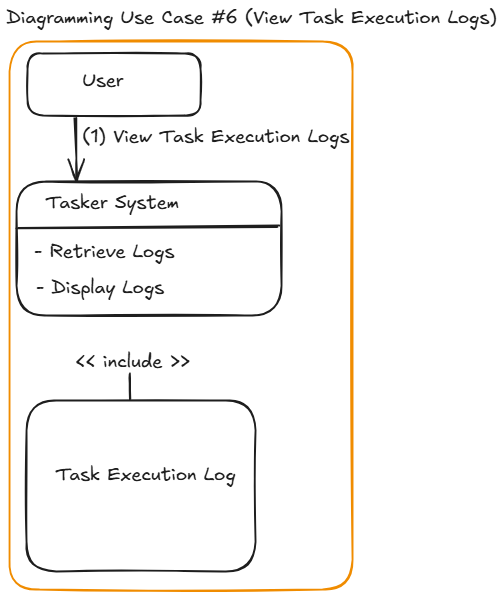
1. User selects the option to view task execution logs.
2. The system retrieves and displays the logs of past task executions.
3. The user scrolls through the logs and can filter or search based on criteria (e.g., date, task type).

## Diagramming Use Case #6 (View Task Execution Logs)

Actors: User (The individual who wants to view task logs)

System: Tasker System (The system responsible for logging and displaying task execution results)

Use Case: View Task Execution Logs (The action performed by the user to view logs of executed tasks)



# **Objects, Classes and Relationships**

## **Objects**

* **Runnable**

Purpose: Represents the action to be performed by the task (the code that executes when the task is run).

Significance: Provides a flexible way to define task actions as a lambda expression or method reference.

* **Instant**

Purpose: Represents a specific point in time, used to define start dates for scheduled and repeated tasks.

Significance: Enables precise scheduling of tasks, taking into account different time zones and timestamps.

* **TimeUnit**

Purpose: Represents the unit of time for delay specifications (e.g., seconds, minutes, hours).

Significance: Offers flexibility in defining delays for DelayTask and intervals for RepeatedTask.

* **Duration**

Purpose: A temporal amount representing a time interval, used for calculating the time between instances.

Significance: Facilitates the calculation of delays and waiting times for tasks.

* **ScheduledFuture**

Purpose: Represents the result of a delayed or periodic computation that can be cancelled or checked for completion.

Significance: Allows for tracking the status of scheduled tasks and managing their lifecycle effectively.

* **ScheduledExecutorService**

Purpose: An interface for managing and scheduling tasks for execution in a background thread pool.

Significance: Ensures efficient and concurrent execution of tasks, enhancing application performance.

## **Classes**

**TaskType (Enum)**

* Description: Represents the different types of tasks in the system.
* Values: NOW, DELAY, REPEATED, SCHEDULE
* Relationships: Used by the Task class to define the type of task.

**Task (Class)**

* Description: Serves as a base class for different task types.
* Attributes:
* Runnable action: A reference to the action to be executed.
* TaskType taskType: The type of the task (NOW, DELAY, REPEATED, SCHEDULE).
* Methods:
* Getters and setters for action and taskType.
* Relationships: Inherited by classes NowTask, DelayTask, RepeatedTask, and ScheduleTask.

**NowTask (Class)**

* Description: Represents a task that executes immediately.
* Relationships: Inherits from Task.

**DelayTask (Class)**

* Description: Represents a task that executes after a specific delay.
* Attributes:
* long delay: The amount of time to wait before execution.
* TimeUnit timeUnit: The unit of time for the delay.
* Relationships: Inherits from Task.

**RepeatedTask (Class)**

* Description: Represents a task that executes repeatedly at defined intervals.
* Attributes:
* Instant startDate: The start date for the repetition.
* long nextRunInSeconds: The interval between repetitions in seconds.
* Relationships: Inherits from Task.

**ScheduleTask (Class)**

* Description: Represents a task that executes at a specific future date and time.
* Attributes:
* Instant startDate: The specific date and time for execution.
* Relationships: Inherits from Task.

**TimeService (Class)**

* Description: Provides shared utility methods for time calculations and handling.
* Methods:
* long calculateAmountOfSeconds(Instant startDate): Calculates the number of seconds between the current time and a specified start date.

**TaskProcessor (Class)**

* Description: Handles the processing of tasks based on their type.
* Attributes:
* ScheduledExecutorService executorService: Manages task scheduling and execution.
* Methods:
* void process(Task task): Processes a task based on its type.
* static void stopProcessing(): Gracefully stops the processing of tasks.

**Individual Task Processors (Classes)**

* Description: Handles the execution of specific task types.
* Classes:

**NowProcessor**

**DelayProcessor**

**RepeatedProcessor**

**ScheduleProcessor**

* Relationships: Each processor class handles its specific task type and contains logic for task scheduling and execution.

**Main (Class)**

* Description: The entry point of the application that demonstrates how to utilize the Tasker system.
* Relationships: Increases interaction with Task classes and TaskProcessor.

## **Relationships**

* **Inheritance:**

NowTask, DelayTask, RepeatedTask, and ScheduleTask inherit from the Task class. This establishes a common interface and functionality for all task types.

* **Association:**

TaskProcessor uses Task instances for processing tasks, creating an association where the processor handles various task activities.

Individual task processor classes (e.g., NowProcessor, DelayProcessor) relate to their respective Task classes for processing logic.

* **Dependency:**

TimeService provides support methods that are used by multiple task classes (for calculating delays and times).

* **Aggregation:**

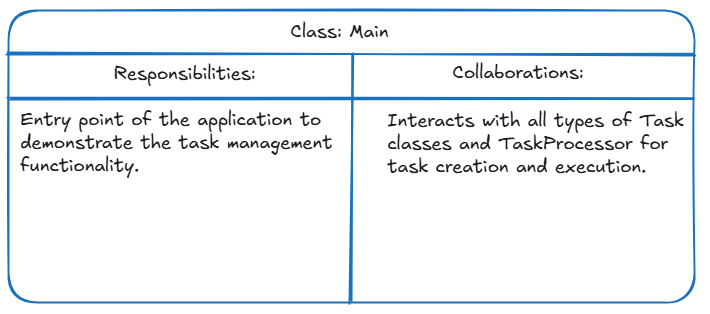
The TaskProcessor class has a collection of scheduled tasks that are managed through a ScheduledExecutorService.

# **CRC Cards**

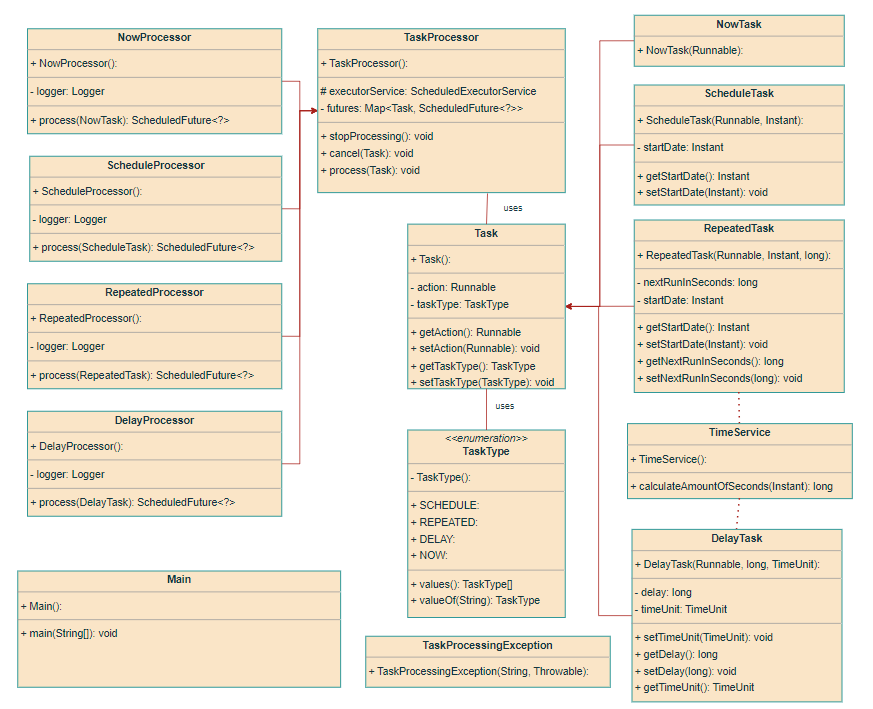
Table format succinctly outlines the key classes in the Tasker:

| **Class** | **Responsibilities** | **Collaborations** |
| --- | --- | --- |
| **TaskType** | Define and represent the different types of tasks. | Collaborates with the Task class. |
| **Task** | Serve as the base class for all task types. | Collaborates with subclasses: NowTask, DelayTask, RepeatedTask, ScheduleTask. Used by TaskProcessor. |
| **NowTask** | Execute an action immediately. | Inherits from Task. Collaborates with TaskProcessor. |
| **DelayTask** | Execute an action after a specified delay. | Inherits from Task. Used by DelayProcessor. |
| **RepeatedTask** | Execute an action repeatedly at defined intervals. | Inherits from Task. Collaborates with RepeatedProcessor. |
| **ScheduleTask** | Execute an action at a specified future date and time. | Inherits from Task. Works with ScheduleProcessor. |
| **TimeService** | Provide utility methods for time calculations. | Utilized by DelayTask and RepeatedTask. |
| **TaskProcessor** | Process various types of tasks and manage their execution. | Interacts with Task subclasses during processing. Utilizes specific processors like NowProcessor, DelayProcessor, RepeatedProcessor, ScheduleProcessor. |
| **NowProcessor** | Handle the processing of NowTask. | Collaborates with TaskProcessor. |
| **DelayProcessor** | Handle the processing and scheduling of DelayTask. | Collaborates with TaskProcessor. |
| **RepeatedProcessor** | Handle the scheduling and repetitive execution of RepeatedTask. | Interacts with TaskProcessor during task processing. |
| **ScheduleProcessor** | Manage scheduling and execution of ScheduleTask. | Collaborates with TaskProcessor. |
| **Main** | Entry point of the application to demonstrate functionality. | Interacts with all types of Task classes and TaskProcessor. |

Example of the card could look like this one:



# **Class Diagrams**



# **Java code**

public class Task {  
 private Runnable action;  
 private TaskType taskType;  
  
 public Runnable getAction() {  
 return action;  
 }  
  
 public void setAction(Runnable action) {  
 this.action = action;  
 }  
  
 public TaskType getTaskType() {  
 return taskType;  
 }  
  
 public void setTaskType(TaskType taskType) {  
 this.taskType = taskType;  
 }  
}

public enum TaskType {  
 *NOW*, *SCHEDULE*, *DELAY*, *REPEATED*}

public class ScheduleTask extends Task{  
 private Instant startDate;  
 public ScheduleTask(Runnable action, Instant startDate) {  
 this.startDate = startDate;  
 setAction(action);  
 setTaskType(TaskType.*SCHEDULE*);  
 }  
 public Instant getStartDate() {  
 return startDate;  
 }  
 public void setStartDate(Instant startDate) {  
 this.startDate = startDate;  
 }  
}

public class RepeatedTask extends Task{  
 private Instant startDate;  
 private long nextRunInSeconds;  
 public RepeatedTask(Runnable action, Instant startDate, long nextRunInSeconds) {  
 this.startDate = startDate;  
 this.nextRunInSeconds = nextRunInSeconds;  
 setAction(action);  
 setTaskType(TaskType.*REPEATED*);  
 }  
 public Instant getStartDate() {  
 return startDate;  
 }  
 public void setStartDate(Instant startDate) {  
 this.startDate = startDate;  
 }  
 public long getNextRunInSeconds() {  
 return nextRunInSeconds;  
 }  
 public void setNextRunInSeconds(long nextRunInSeconds) {  
 this.nextRunInSeconds = nextRunInSeconds;  
 }  
}

public class NowTask extends Task{  
 public NowTask(Runnable action) {  
 setAction(action);  
 setTaskType(TaskType.*NOW*);  
 }  
}

public class DelayTask extends Task{  
 private long delay;  
 private TimeUnit timeUnit;  
 public DelayTask(Runnable action, long delay, TimeUnit timeUnit) {  
 this.delay = delay;  
 this.timeUnit = timeUnit;  
 setAction(action);  
 setTaskType(TaskType.*DELAY*);  
 }  
 public long getDelay() {  
 return delay;  
 }  
 public void setDelay(long delay) {  
 this.delay = delay;  
 }  
 public TimeUnit getTimeUnit() {  
 return timeUnit;  
 }  
 public void setTimeUnit(TimeUnit timeUnit) {  
 this.timeUnit = timeUnit;  
 }  
}

public class TaskProcessor {  
 protected static ScheduledExecutorService *executorService* = Executors.*newScheduledThreadPool*(10);  
 private Map<Task, ScheduledFuture<?>> futures = new ConcurrentHashMap<>();  
 public void process(Task task){  
 ScheduledFuture<?> future;  
 switch (task.getTaskType()){  
 case *NOW* -> future = new NowProcessor().process((NowTask) task);  
 case *DELAY* -> future = new DelayProcessor().process((DelayTask) task);  
 case *REPEATED* -> future = new RepeatedProcessor().process((RepeatedTask) task);  
 case *SCHEDULE* -> future = new ScheduleProcessor().process((ScheduleTask) task);  
 default -> throw new IllegalArgumentException("Unsupported task type");  
 }  
 futures.put(task, future);  
 }  
  
 public void cancel(Task task){  
 ScheduledFuture<?> future = futures.get(task);  
 if(future != null){  
 future.cancel(false);  
 }  
 }  
  
 public static void stopProcessing() throws InterruptedException {  
 *executorService*.shutdown(); // Prevent new tasks from being scheduled  
 if (!*executorService*.awaitTermination(15, TimeUnit.*SECONDS*)) {  
 *executorService*.shutdownNow(); // Force shutdown if not completed  
 }  
 }  
}

public class ScheduleProcessor extends TaskProcessor {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(ScheduleProcessor.class);  
  
 public ScheduledFuture<?> process(ScheduleTask scheduleTask){  
 Instant startDate = scheduleTask.getStartDate();  
 long delay = TimeService.*calculateAmountOfSeconds*(startDate);  
 return *executorService*.schedule(() -> {  
 try {  
 scheduleTask.getAction().run();  
 } catch (Exception e) {  
 *logger*.error("Error executing ScheduleTask: {}", e.getMessage(), e);  
 throw new TaskProcessingException("ScheduleTask execution failed", e);  
 }  
 }, delay, TimeUnit.*SECONDS*);  
 }  
}

public class RepeatedProcessor extends TaskProcessor{  
 private static final Logger *logger* = LoggerFactory.*getLogger*(RepeatedProcessor.class);  
 public ScheduledFuture<?> process(RepeatedTask repeatedTask){  
 Instant startDate = repeatedTask.getStartDate();  
 long delay = TimeService.*calculateAmountOfSeconds*(startDate);  
 long nextRun = repeatedTask.getNextRunInSeconds();  
 return *executorService*.scheduleAtFixedRate(() -> {  
 try {  
 repeatedTask.getAction().run();  
 } catch (Exception e) {  
 *logger*.error("Error executing RepeatedTask: {}", e.getMessage(), e);  
 throw new TaskProcessingException("RepeatedTask execution failed", e);  
 }  
 }, delay, nextRun, TimeUnit.*SECONDS*);  
 }  
}

public class NowProcessor extends TaskProcessor {  
  
 private static final Logger *logger* = LoggerFactory.*getLogger*(NowProcessor.class);  
  
 public ScheduledFuture<?> process(NowTask nowTask){  
 Runnable runnable = nowTask.getAction();  
 return *executorService*.schedule(()->{  
 try {  
 runnable.run();  
 }catch (Exception e){  
 *logger*.error("Error executing NowTask: {}", e.getMessage(), e);  
 throw new TaskProcessingException("NowTask execution failed", e);  
 }  
 }, 0, TimeUnit.*SECONDS*);  
 }  
}

public class DelayProcessor extends TaskProcessor {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(DelayProcessor.class);  
  
 public ScheduledFuture<?> process(DelayTask delayTask) {  
 Runnable runnable = delayTask.getAction();  
 return *executorService*.schedule(() -> {  
 try {  
 runnable.run();  
 } catch (Exception e) {  
 *logger*.error("Error executing DelayTask: {}", e.getMessage(), e);  
 throw new TaskProcessingException("DelayTask execution failed", e);  
 }  
 }, delayTask.getDelay(), delayTask.getTimeUnit());  
 }  
  
}

public class TimeService {  
 public static long calculateAmountOfSeconds(Instant startDate){  
 Instant now = Instant.*now*();  
 Duration duration = Duration.*between*(now, startDate);  
 return duration.getSeconds();  
 }  
}

public class TaskProcessingException extends RuntimeException{  
  
 public TaskProcessingException(String message, Throwable cause) {  
 super(message, cause);  
 }  
}

public class Main {  
 public static void main(String[] args) throws InterruptedException {  
  
 // Create task examples  
 Instant startNextTask = Instant.*now*().plus(10, TimeUnit.*SECONDS*.toChronoUnit());  
  
 Task nowTask = new NowTask(() -> System.*out*.println("Now task executed"));  
 Task delayTask = new DelayTask(() -> System.*out*.println("Delay task executed"), 20, TimeUnit.*SECONDS*);  
 Task scheduleTask = new ScheduleTask(() -> System.*out*.println("Scheduled task executed"), startNextTask);  
 Task repeatedTask = new RepeatedTask(() -> System.*out*.println("Repeated task executed"), Instant.*now*(), 5);  
  
 TaskProcessor taskProcessor = new TaskProcessor();  
  
 try {  
 // Process tasks  
 taskProcessor.process(nowTask);  
 taskProcessor.process(delayTask);  
 taskProcessor.process(scheduleTask);  
 taskProcessor.process(repeatedTask);  
 } catch (Exception e) {  
 System.*err*.println("Error processing tasks: " + e.getMessage());  
 }  
  
 // Optionally, wait for a while before shutting down to see task outputs  
 try {  
 Thread.*sleep*(30000); // Wait enough time to see execution of scheduled and repeated tasks  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 } finally {  
 // Clean up and shut down the processor  
 try {  
 TaskProcessor.*stopProcessing*();  
 System.*out*.println("Task processing stopped successfully.");  
 } catch (InterruptedException e) {  
 System.*err*.println("Error shutting down task processing: " + e.getMessage());  
 }  
 }  
  
 }  
}