Abstract Factory pattern :

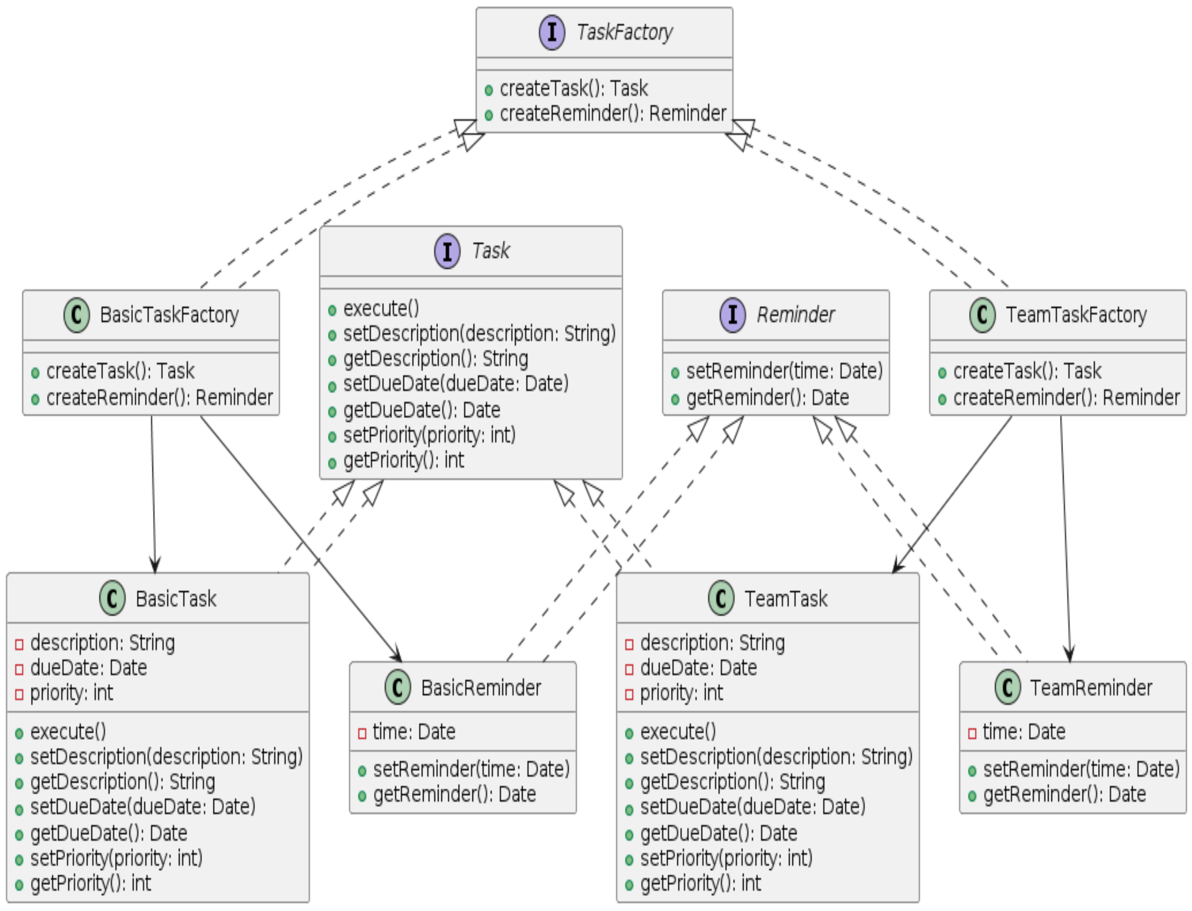
Aim :

To create a task management app that can manage different types of tasks and users by using the Abstract Factory design pattern.This allows for easy scalability and flexibility by decoupling the creation of objects from their usage.

Description:

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern involves creating an abstract factory interface with methods for each type of product and implementing concrete factories to create specific product instances.

In a task management app, this pattern can be used to create different types of tasks (e.g., SimpleTask, ComplexTask)and users (e.g., SimpleUser, AdminUser). The client code uses the factoryinterface to create these objects, ensuring that the creation logic is encapsulated and easily extendable.

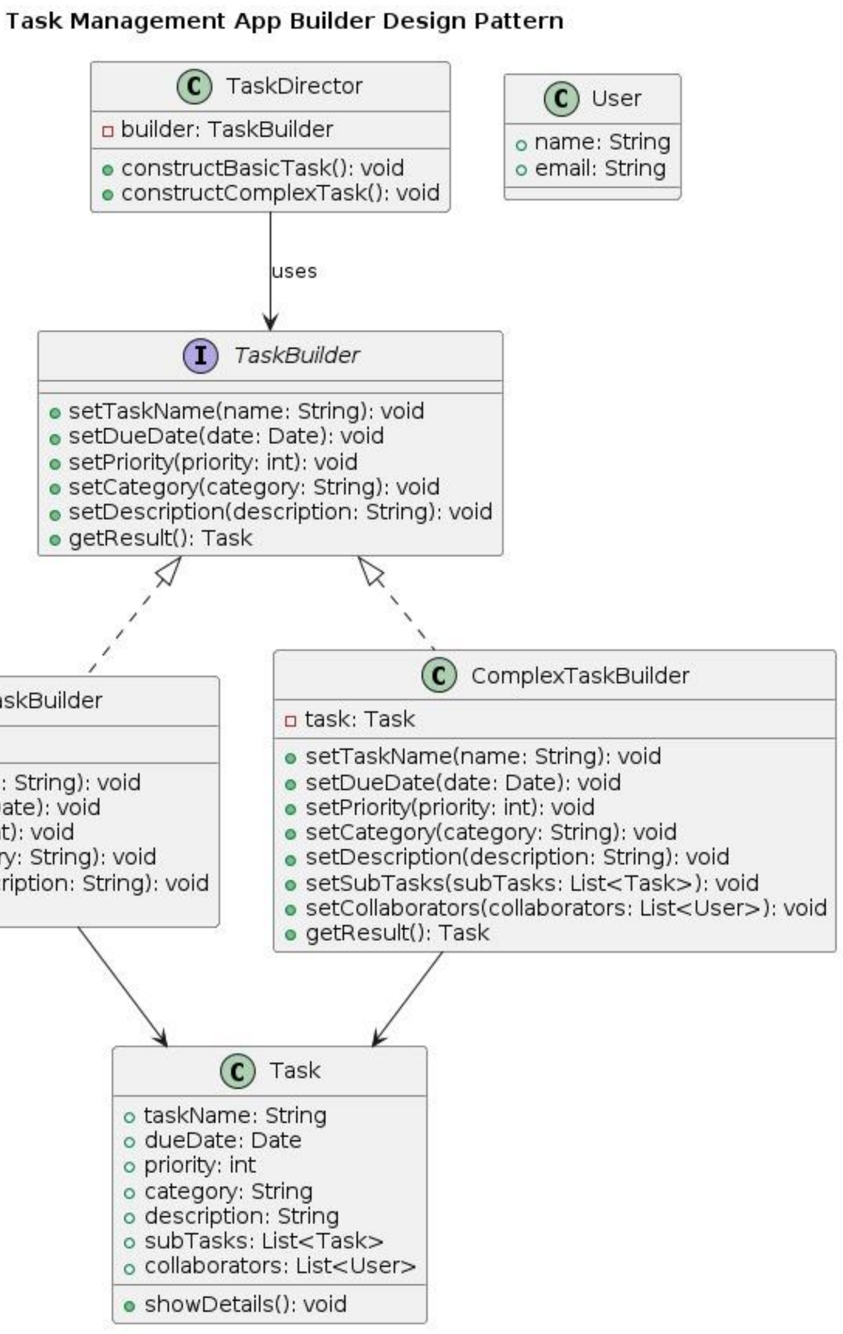


design Builder design pattern :

Aim: To design a task management app using the Builder design pattern, which will allow for the creation of complex task objects step by step, providing a clear and flexible way to construct different types of tasks with various configurations.

Description:The Builder design pattern is a creational pattern that provides a way to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

In the context of a task management app, the Builder pattern can be used to create tasks with various attributes such as title, description, priority, due date, and assigned user. This pattern is particularly useful when an object needs to be created in multiple steps, or when there are many optional parameters.


Task Management App Builder Design Pattern
TaskDirector
builder: TaskBuilder
constructBasicTask(): void
constructComplexTask(): void
User
name: String
email: String
uses
TaskBuilder
setTaskName(name: String): void
setDueDate(date: Date): void
setPriority(priority: int): void
setCategory(category: String): void
setDescription(description: String): void
getResult(): Task
BasicTaskBuilder
task: Task
setTaskName(name: String): void
setDueDate(date: Date): void
setPriority(priority: int): void
setCategory(category: String): void
setDescription(description: String): void
getResult(): Task
ComplexTaskBuilder
task: Task
setTaskName(name: String): void
setDueDate(date: Date): void
setPriority(priority: int): void
setCategory(category: String): void
setDescription(description: String): void
setSubTasks(subTasks: List<Task>): void
setCollaborators(collaborators: List<User>): void
getResult(): Task
Task
taskName: String
dueDate: Date
priority: int
category: String
description: String
subTasks: List<Task>
collaborators: List<User>
showDetails(): void

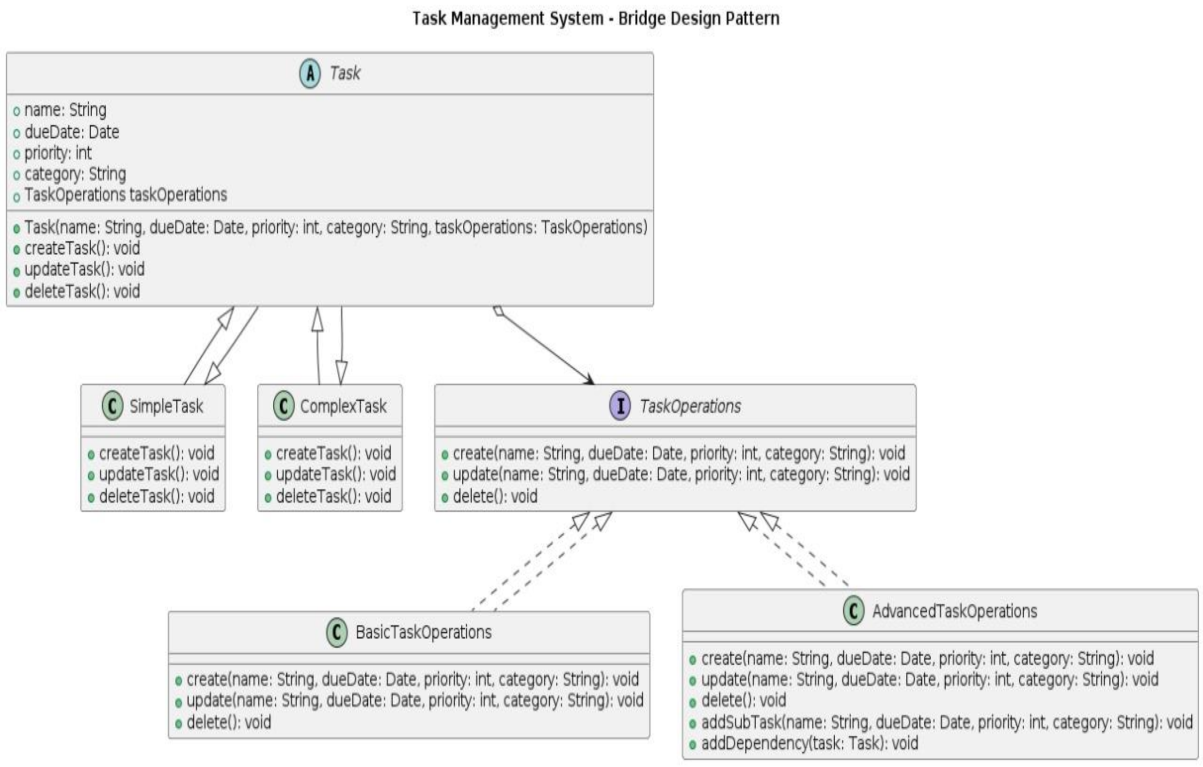
design Bridge design pattern:

Aim: To design a task management app using the Bridge design pattern, which will allow for the separation of an abstraction from its implementation, enabling the two to vary independently.

Description:

The Bridge design pattern is a structural pattern that decouples an abstraction from its implementation so that the two can vary independently. This pattern involves an interface (or abstract class) that defines the abstraction, and another interface that defines the implementation. Concrete implementations of both interfaces can be changed independently without affecting each other.

In the context of a task management app, the Bridge pattern can be used to separate the abstraction of a Task from the various implementations of how tasks are handled or displayed, such as in a web interface or a mobile interface.

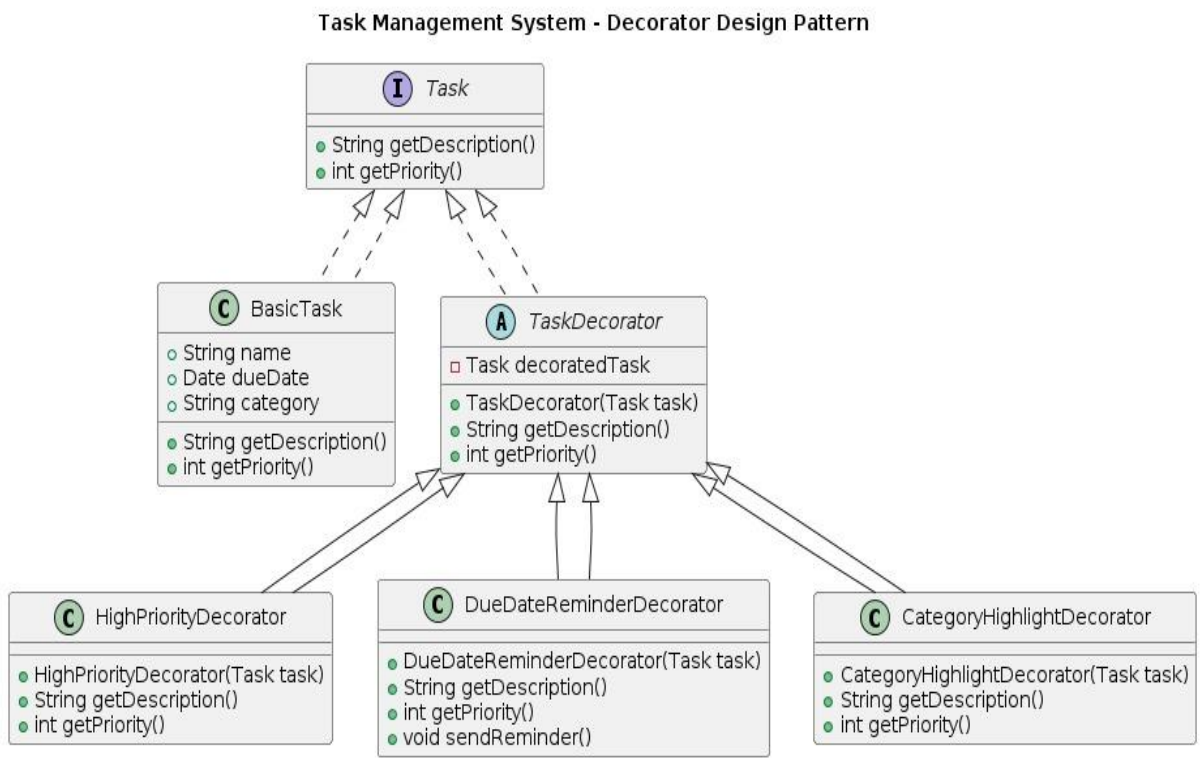

Task Management System - Bridge Design Pattern
A Task
name: String
dueDate: Date
priority: int
category: String
TaskOperations taskOperations
Task(name: String, dueDate: Date, priority: int, category: String, taskOperations: TaskOperations)
createTask(): void
updateTask(): void
deleteTask(): void
C SimpleTask
createTask(): void
updateTask(): void
deleteTask(): void
C ComplexTask
createTask(): void
updateTask(): void
deleteTask(): void
I TaskOperations
create(name: String, dueDate: Date, priority: int, category: String): void
update(name: String, dueDate: Date, priority: int, category: String): void
delete(): void
C BasicTaskOperations
create(name: String, dueDate: Date, priority: int, category: String): void
update(name: String, dueDate: Date, priority: int, category: String): void
delete(): void
C AdvancedTaskOperations
create(name: String, dueDate: Date, priority: int, category: String): void
update(name: String, dueDate: Date, priority: int, category: String): void
delete(): void
addSubTask(name: String, dueDate: Date, priority: int, category: String): void
addDependency(task: Task): void

design Decorator design pattern:

Aim: To design a task management app using the Decorator design pattern, which allows for the dynamic addition of responsibilities to objects without modifying their structure.

Description:

The Decorator design pattern is a structural pattern that lets you dynamically attach additional responsibilities or behaviors to an object. It provides a flexible alternative to subclassing for extending functionality. In a task management app, this pattern can be used to add extra features to tasks, such as notifications, deadlines, or priority levels, without altering the core task class.


Task Management System - Decorator Design Pattern
I Task
String getDescription()
int getPriority()
C BasicTask
String name
Date dueDate
String category
String getDescription()
int getPriority()
A TaskDecorator
Task decoratedTask
TaskDecorator(Task task)
String getDescription()
int getPriority()
C HighPriorityDecorator
HighPriorityDecorator(Task task)
String getDescription()
int getPriority()
C DueDateReminderDecorator
DueDateReminderDecorator(Task task)
String getDescription()
int getPriority()
void sendReminder()
C CategoryHighlightDecorator
CategoryHighlightDecorator(Task task)
String getDescription()
int getPriority()