**Development of engineering method:**

1. **Problem identification.**

*Identification of the needs:*

* The client does not have a system that allows them to properly track the tasks and reminders that a potential user has.
* The client requires a system that allows the user to add, modify and delete its pending tasks and reminders.
* The tasks and reminders need to be stored on a data structure.
* The software must include an interface that allows the user to see his tasks and reminders ordered by a deadline or a quality for tasks that divides them in terms of priority.
* The user may do something or add a task incorrectly, that’s why is needed a function that allows him to undo the previous action. (Add, modify, or delete a task or reminder).
* The solution must guarantee that the user does not lose the order of priority of the task that were previously added.

*Definition of the problem:*

The client requires a software that allows the user to manage its pending tasks on the daily basis.

1. **Information gathering.**

*Hash table*

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

Font:

[**https://www.javatpoint.com/hash-table**](https://www.javatpoint.com/hash-table)

*Queues*

A queue is a sequence of items that are organized in a specific order and can be changed by adding items to one end of the sequence and removing items from the opposite end. Typically, the end where items are added is referred to as the "back," "tail," or "rear" of the queue, while the end where items are removed is known as the "head" or "front" of the queue.

The operation of adding an element to the rear of the queue is known as enqueue, and the operation of removing an element from the front is known as dequeue.

Font:

<https://en.wikipedia.org/wiki/Queue_(abstract_data_type)>

*Priority queues*

A priority queue is a specialized type of queue where elements are organized according to their assigned priority values. In this type of queue, elements having higher priority values are generally retrieved ahead of those with lower priority values. Each element in a priority queue is assigned a specific priority value, and when you insert an element into the queue, it is positioned within the queue according to its priority value.

Font:

[https://www.geeksforgeeks.org/priority-queue](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)

*Stacks*

A stack is a sequential data structure that adheres to a specific execution order for its operations. This order can be either Last in First Out (LIFO), where the most recently added element is the first to be removed, or First in Last Out (FILO), where the initial element added is the last to be removed.

A stack features 2 primary operations:

* Push: This operation involves adding an element to the collection.
* Pop: It entails the removal of the most recently added element that has not been previously removed.

Font:

<https://www.geeksforgeeks.org/stack-data-structure/>

[https://en.wikipedia.org/wiki/Stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

*Generics*

Generics, in essence, refer to parameterized types. The concept revolves around enabling types (like Integer, String, user-defined types, etc.) to serve as parameters for methods, classes, and interfaces. Using Generics, it becomes possible to construct classes that can function with various data types. Any entity, whether it's a class, interface, or method, operating on a parameterized type, is considered a generic entity.

In contrast, the Object class stands as the superclass for all other classes, permitting an Object reference to point to any object.

<https://www.geeksforgeeks.org/generics-in-java/>

*Node*

A node is a basic data structure that contains data and one or more links to other nodes. Nodes can be used to represent a tree structure or a linked list. In such structures using nodes, one can move from one node to another.

<https://www.codecademy.com/learn/cspath-cs-102/modules/nodes/cheatsheet>

1. **Finding creative solutions**

**Alternative 1 / AVL tree:**

An AVL Tree is a type of balanced binary search tree that is used to maintain a balance in the workload and enable efficient search of elements. Each node in the tree has a property called "balance factor", which measures the height difference between its left and right subtrees. Balance factors are maintained in a specific range, usually -1, 0, or 1.

The guaranteed balance of the AVL tree ensures that search, insert, and delete operations have logarithmic time complexity, making them efficient even for large sets of tasks.

Tasks are organized in a natural order, making it easy to find and retrieve tasks based on criteria such as due date or priority.

Keeping the AVL tree balanced requires careful management during inserts and deletes, which can introduce additional performance overhead compared to other data structures.

**Alternative 2/ JSON or XML Storage Files:**

JSON or XML file format: Storing tasks in structured files, such as JSON (JavaScript Object Notation) or XML (Extensible Markup Language), has several advantages:

Facilitates data import and export: JSON and XML formats are widely supported and used in a variety of applications and programming languages. This allows users to import and export tasks to and from other task management systems or productivity applications easily.

**Alternative 3 / Hash Table:**

Allows efficient access to elements through a hash function that associates a key with a value. In the task management system, you can use due date or priority as a key for each task.

Storing tasks in a hash table based on due date allows users to quickly search for all tasks that have a specific date. This is especially useful for remembering tasks that need to be completed soon.

Organizing by priority is also valuable, allowing users to quickly access the most important or urgent tasks.

The hash function calculates the storage location of a task based on its key (due date or priority) efficiently. This minimizes search and access time for tasks, improving the user experience when finding and managing their tasks.

We can use techniques such as linked lists to resolve these collisions and ensure data integrity.

1. **Preliminary Designs:**

Alternative 1 / AVL tree:

* **Efficient Search, Insert, and Delete Operations:** AVL trees are self-balancing binary search trees, ensuring a guaranteed height difference between left and right subtrees. This logarithmic time complexity makes them efficient for large tasks, especially for managing tasks and reminders.
* **Natural Order Organization:** The AVL tree's natural task organization simplifies task retrieval and querying, allowing users to efficiently manage their tasks based on criteria like due date or priority.
* **Balanced Workload:** The AVL tree maintains a balance in workload distribution, ensuring efficient tasks and reminders management, unlike unbalanced binary trees that can become lopsided and result in inefficient operations.
* **Performance:** AVL tree balance management requires careful handling during insertions and deletions, with rebalancing operations introducing additional performance overhead, causing users to experience slightly slower insert and delete operations.
* **Complexity:** AVL tree implementation and management can be more complex than simpler data structures, requiring more development effort, but it offers efficiency and organization advantages, but it's crucial to consider this trade-off.

Alternative 2 / Json or XML Storage Files:

* **Interoperability:** JSON and XML are widely supported in programming languages and applications, facilitating easy task import and export from various systems, productivity applications, and data processing tools, and are crucial for users integrating task management with other software or migrating data.
* **Structured Data:** JSON and XML are structured data formats that enable the creation of a schema for task data, ensuring a standardized and organized representation of tasks, thereby enhancing consistency and integrity in the task management system.
* **Version control:** Systems can be effectively utilized for tasks stored in JSON or XML files, enabling easy tracking of changes, and managing different versions of task data, making it beneficial for collaborative environments.
* **Complexity:** Human-readable files can enhance efficiency, but they can also increase complexity, particularly for tasks with extensive data, leading to larger, more challenging files to maintain.
* **Parsing overhead:** Performance-critical applications may face concerns due to slower parsing times when reading and writing JSON and XML files compared to binary formats.

Alternative 3/ Hash table:

* **Efficient Access:** Hash tables in task management systems enable efficient access to elements through a hash function, allowing quick retrieval of tasks associated with due dates or priorities, allowing users to find urgent tasks or prioritize their work based on importance.
* **Collision handling:** Hash tables can handle collisions where two tasks share the same key, ensuring data integrity. Techniques like linked lists can resolve these collisions, allowing for correct storage and retrieval of tasks with the same key.
* **Space efficiency:** Hash tables are space efficient as they don't require predefined storage for all keys and allocate storage dynamically based on the table's elements, making them suitable for managing diverse task sizes.
* **Flexibility:** Hash tables provide flexibility in key types and attributes, allowing for customization in task organization and access.
* **Order:** Hash tables don't automatically maintain task order, but if you need to sort tasks by due date or priority, you may need additional data structures or logic. That’s why we are implementing a priority queue that stores prior tasks or reminders that will be shown first when the user navigates through them.

1. Selection of the best Alternative:

* Criterium A: Assess how efficiently each alternative allows users to retrieve tasks based on attributes like due date, priority, or any other relevant criteria.
  + 1. Inefficient.
  + 2. Moderately efficient.
  + 3. Efficient.
  + 4. Highly efficient.
* Criterium B: Examine each alternative's handling of task data integrity, updates, insertions, and deletions without introducing errors or inconsistencies in the task management system.
  + 1. Inadequate handling.
  + 2. Adequate handling.
  + 3. Effective handling.
* Criterium C: The facility in algorithmic implementation assesses the ease of translating an algorithm or solution into executable code within a software system, considering various aspects of the implementation process for real-world application.
  + 1. Low Facility.
  + 2. Moderate Facility.
  + 3. High Facility.
* Criterium D: The complexity criterion evaluates the resource demands of a solution or algorithm, including time and space, to complete a task or operation. It assesses the efficiency and scalability of a solution, aiding in decision-making when choosing between multiple options.
  + 1. Suboptimal performance.
  + 2. Moderate performance.
  + 3. High performance

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterium 1 | Criterium 2 | Criterium 3 | Criterium 4 | Total |
| Alternative 1 | 3 | 3 | 2 | 2 | 10 |
| Alternative 2 | 2 | 2 | 3 | 1 | 8 |
| Alternative 3 | 4 | 3 | 2 | 3 | 12 |

According to the results, the chosen alternative will be the hash table, and priority queue to satisfy the requirements of the problem, since it got the best qualification among all the alternatives.

1. Preparation of reports and specifications:

Situation: Task and reminders management. (The operations that this concept implies).

function modifyTask(taskId, newTitle, newDescription, newDeadline, newIsPriority)

msg ← ""

task ← taskTable.get(taskId)

if task is not null then

if newTitle is not null and not isEmpty(newTitle) then

task.setTitle(newTitle)

msg += "\nTítulo modificado."

end if

if newDescription is not null and not isEmpty(newDescription) then

task.setDescription(newDescription)

msg += "\nDescripción modificada."

end if

if newDeadline is not null then

task.setDeadline(newDeadline)

msg + ←"\nFecha límite modificada."

end if

task.setPriority(newIsPriority)

msg +← "\nPrioridad modificada"

if not isEmpty(msg) then

msg ← "Tarea modificada exitosamente:" + msg

aftertask ← taskTable.get(taskId)

action ← new Actions("Modify task", task, aftertask)

actionStack.push(action)

else

msg ← "No se realizaron modificaciones."

end if

else

msg ← "La tarea con el ID " + taskId + " no existe."

end if

return msg

function addTask(id, title, description, deadline, priority)

msg ← ""

try

if id is null or title is null, or description is null or deadline is null then

throw new NullValueException("Error al agregar la tarea: Todos los campos deben ser proporcionados y no pueden ser nulos.")

else

task ← new Task(id, title, description, deadline, priority)

taskTable.put(id, task)

if priority >= 2 then

priorityTask.enqueue(task)

else

nonPriorityQueue.enqueue(task)

end if

msg ← "Tarea agregada exitosamente."

action ← new Actions("Add task", task)

actionStack.push(action)

end if

catch NullValueException e

msg ← e.getMessage()

end try

return msg

end function

function removeTask(taskId)

msg ← ""

if taskTable.containsKey(taskId) then

task ← taskTable.get(taskId)

taskTable.remove(taskId)

priorityTask.removeElement(task)

if task.getPriority() <= 1 then

nonPriorityQueue.dequeue()

end if

msg ← "Tarea eliminada exitosamente."

action ← new Actions("Delete task", task)

actionStack.push(action)

else

msg ← "La tarea con el ID " + taskId + " no existe."

end if

return msg

end function

function printListByPriority()

output("===== Tareas Prioritarias =====")

if priorityTask.isEmpty() then

output("No hay tareas prioritarias para mostrar.")

else

tempQueue ← new PriorityQueue<Task>()

while not priorityTask.isEmpty() do

task ← priorityTask.dequeue()

output(task.toString())

tempQueue.enqueue(task)

end while

priorityTask ← tempQueue

end if

output("===== Tareas No Prioritarias =====")

if nonPriorityQueue.isEmpty() then

output("No hay tareas no prioritarias para mostrar.")

else

tempQueue2 ← new Queue<Task>()

while not nonPriorityQueue.isEmpty() do

task ← nonPriorityQueue.dequeue()

output(task.toString())

tempQueue2.enqueue(task)

end while

nonPriorityQueue ← tempQueue2

end if

output("=================================================")

end function

1. **Design implementation. (The subroutine is constructed in the java programming language, below the definition of each one ).**
2. Add tasks and reminders.
3. Storage tasks and reminders.
4. Modify tasks and reminders.
5. Delete tasks and reminders.
6. Show list of tasks and reminders.
7. Priority management.
8. Manage action stack.
9. Register actions.
10. Undo last action.

Subroutine specification:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name: | *[RF0: Add tasks and reminders]* | | | |
| Resume | *The system must add a new task or reminder that the user wants to create to the hash table.*  *This must have the following information:*  *A title that represents the task or reminder, a short description of it, the date and time when the reminder ends from the created date and the priority of that reminder, which could be of three types: high, medium, and low.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| id | String | | *- The identifier must not be repeated*  *- There should not be an empty space at the entrance* |
| title | String | | - There should not be an empty space at the entrance |
| description | String | | *- There should not be an empty space at the entrance* |
| deadline | Date | | *- There should not be an empty space at the entrance*  *-Must have a date format:*  *dd/mm/yyyy* |
| priority | int | | *- There should not be an empty space at the entrance*  *-It will have a menu type format:*  *1.High*  *2.Medium*  *3.Low* |
| Postcondition | The system will save the information within the hash table | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| message | | String | “Your calendar or reminder was added successfully” |

Construction:

public class Task {

private String title;

private String description;

private Date deadline;

private Priority priority;

public Task(String title, String description, Date deadline, Priority priority) {

this.title = title;

this.description = description;

this.deadline = deadline;

this.priority = priority;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name: | *[RF1: Storage tasks and reminders]* | | | |
| Resume | The system, through a hash table implementation, must store the tasks and reminders that the user wants to enter. This will be achieved through a unique identifier that would be the access key and a value that would be the information that the agenda or reminder would contain.  The hash table will have as input the title, description, deadline, priority, and other information that the user wants to enter when the reminder has been added. | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| id | String | | *- The identifier must not be repeated*  *- There should not be an empty space at the entrance* |
| value | String | | *- There should not be an empty space at the entrance* |
| Postcondition | The system will save the information within the hash table | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| - | | - | - |

Construction:

public String addTask(String id, String title, String description, Date deadline, int priority) {

msg = "";

try {

if (id == null || title == null || description == null || deadline == null) {

throw new NullValueException(

"Error al agregar la tarea: Todos los campos deben ser proporcionados y no pueden ser nulos.");

} else {

Task task = new Task(id, title, description, deadline, priority);

taskTable.put(id, task);

if (priority >= 2) {

priorityTask.enqueue(task);

} else {

nonPriorityQueue.enqueue(task);

}

msg = "Tarea agregada exitosamente.";

}

} catch (NullValueException e) {

msg = e.getMessage();

}

return msg;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name: | *[RF2: Modify tasks and reminders]* | | | |
| Resume | The system must allow the user to modify a task or reminder already created in the system when required, that is, the information it contains:  The title, description, deadline, and priority contained above. | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| title | String | | *- There should not be an empty space at the entrance* |
| description | String | | *- There should not be an empty space at the entrance* |
| deadline | Date | | *- There should not be an empty space at the entrance*  *-Must have a date format:*  *dd/mm/yyyy* |
| priority | int | | *- There should not be an empty space at the entrance*  *-It will have a menu type format:*  *1. Priority*  *2. Not priority* |
| Postcondition | The system will modify the information changed by the user within the hash table | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| message | | String | “Your agenda or reminder was successfully modified” |

Construction:

public String modifyTask(String taskId, String newTitle, String newDescription, Date newDeadline,

int newIsPriority) {

msg = "";

Task task = taskTable.get(taskId);

if (task != null) {

if (newTitle != null && !newTitle.isEmpty()) {

task.setTitle(newTitle);

msg += "\nTítulo modificado.";

}

if (newDescription != null && !newDescription.isEmpty()) {

task.setDescription(newDescription);

msg += "\nDescripción modificada.";

}

if (newDeadline != null) {

task.setDeadline(newDeadline);

msg += "\nFecha límite modificada.";

}

task.setPriority(newIsPriority);

msg += "\nPrioridad modificada";

if (!msg.isEmpty()) {

msg = "Tarea modificada exitosamente:" + msg;

} else {

msg = "No se realizaron modificaciones.";

}

} else {

msg = "La tarea con el ID " + taskId + " no existe.";

}

return msg + "\n";

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name: | *[RF3: Delete tasks and reminders]* | | | |
| Resume | *The system will delete the task or reminder created in the hash table, the user will enter the key entered at the time it was created and it will be deleted from the hash table, if the system does not find the key it will throw an exception message.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| id | String | | *- The identifier must not be repeated*  *- There should not be an empty space at the entrance* |
| Result | The system will delete the task or reminder within the hash table | | | |
| Exits | **Entry name** | | **Data type** | **Format** |
| message | | String | “Your task or reminder was successfully deleted” |

Construction:

public String removeTask(String taskId) {

msg = "";

if (taskTable.containsKey(taskId)) {

Task task = taskTable.get(taskId);

taskTable.remove(taskId);

priorityTask.removeElement(task);

msg = "Tarea eliminada exitosamente.";

} else {

msg = "La tarea con el ID " + taskId + " no existe.";

}

return msg;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | *[RF4: Show list of tasks and reminders]* | | | |
| Resume | *The system will show the user a list of tasks and reminders created by the user ordered by deadline and priority.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| - | - | | *-* |
| Result | The system will display the list of tasks and reminders registered by the user. | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| list | | String | List of tasks and reminders ordered by priority and deadline. |

Construction

public void printListByPriority() {

System.out.println("===== Tareas Prioritarias =====");

if (priorityTask.isEmpty()) {

System.out.println("No hay tareas prioritarias para mostrar.");

} else {

PriorityQueue<Task> tempQueue = new PriorityQueue<>();

while (!priorityTask.isEmpty()) {

Task task = priorityTask.dequeue();

System.out.println(task.toString());

tempQueue.enqueue(task);

}

priorityTask = tempQueue;

}

System.out.println("===== Tareas No Prioritarias =====");

if (nonPriorityQueue.isEmpty()) {

System.out.println("No hay tareas no prioritarias para mostrar.");

} else {

Queue<Task> tempQueue = new Queue<>();

while (!nonPriorityQueue.isEmpty()) {

Task task = nonPriorityQueue.dequeue();

System.out.println(task.toString());

tempQueue.enqueue(task);

}

nonPriorityQueue = tempQueue;

}

System.out.println("=================================================");

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | *[RF5: Priority management]* | | | |
| Resume | *The system will have two different categories to order tasks according to their priorities, this will be implemented through a priority queue according to their level of importance. This will have two categories:*  *1. Priorities*  *2. non-priority*  *If the task is defined as a priority by the user at the time of its creation, they will be handled first as they are most important. If the task is not a priority, they will be managed after the priority ones and will be ordered according to their order of arrival.* | | | |
| Entries | **Entry name** | **Data Type** | | **Condition** |
| - | - | | *-* |
| Result | The system will order the tasks registered by the user by priority. | | | |
| Exits | **Exit name** | | **Data Type** | **Format** |
| - | | - | - |

Construction:

public String addTask(String id, String title, String description, Date deadline, int priority) {

msg = "";

try {

if (id == null || title == null || description == null || deadline == null) {

throw new NullValueException(

"Error al agregar la tarea: Todos los campos deben ser proporcionados y no pueden ser nulos.");

} else {

Task task = new Task(id, title, description, deadline, priority);

taskTable.put(id, task);

if (priority >= 2) {

priorityTask.enqueue(task);

} else {

nonPriorityQueue.enqueue(task);

}

msg = "Tarea agregada exitosamente.";

}

} catch (NullValueException e) {

msg = e.getMessage();

}

return msg;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | *[RF6: Create action stack]* | | | |
| Resume | *The system creates a stack for the purpose of keeping track of user actions, every time the user performs an action of adding, modifying, and deleting a task it records the action in the stack.*  *Each entry must contain information about the action performed and the details of the affected task.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| - | - | | *-* |
| Result | The system creates the action stack to track | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| - | | - | - |

Construction:

public class Action {

private Task afterChanges;

private Task withoutChanges;

private String actionType;

public Action(String actionType, Task afterChanges){

this.actionType = actionType;

this.withoutChanges = null;

this.afterChanges = afterChanges;

}

public class Stack<T> implements IStack<T> {

private INode<T, T> top;

private int size;

public Stack() {

top = null;

size = 0;

}

public class Controller {

private HashTable<String, Task> taskTable;

Queue<Task> priorityQueue;

Queue<Task> nonPriorityQueue;

private String msg;

private int taskIdCounter;

public Controller() {

taskTable = new HashTable<>(1000);

priorityQueue = new Queue<>();

nonPriorityQueue = new Queue<>();

msg = "";

taskIdCounter = 1;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | *[RF7: Register actions]* | | | |
| Resume | *The system every time it performs an action records the action within the stack, if the user adds a new task, it records the action of adding task along with the details of the task, otherwise, if the user modifies a task, it records modify task and the details of before and after modifying it. Everything will be recorded within the actions stack.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| - | - | | *-* |
| Result | El sistema registrara en la pila la acción realizada por el usuario. | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| - | | - | - |

Construction:

public String addTask(String id, String title, String description, Date deadline, int priority) {

Actions action = new Actions("Add task", task);

actionStack.push(action);

}

The last code was just a fragment of the whole add Method.

Actions action = new Actions("Delete task", task);

actionStack.push(action);

The last code was just a fragment of the whole delete Method.

if (!msg.isEmpty()) {

msg = "Tarea modificada exitosamente:" + msg;

Task aftertask = taskTable.get(taskId);

Actions action = new Actions("Modify task", task, aftertask);

actionStack.push(action);

} else {

msg = "No se realizaron modificaciones.";

}

The last code was just a fragment of the whole modify Method.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | *[RF8: Undo last action]* | | | |
| Resume | *. The system will allow the user to undo the last action performed by the user through a user interface.*  *This function will consist of unstacking the last action on the stack and rolling back the corresponding action based on the information stored on the stack.* | | | |
| Entries | **Entry name** | **Data type** | | **Condition** |
| - | - | | *-* |
| Result | El sistema eliminara de la pila la última acción hecha por el usuario | | | |
| Exits | **Exit name** | | **Data type** | **Format** |
| message | | String | “The last registered action has been successfully deleted” |

Construction:

public void undo() {

if (!actionStack.isEmpty()) {

Actions lastAction = actionStack.pop();

String actionType = lastAction.getActionType();

switch (actionType) {

case "Add task":

removeTask(lastAction.getAfterTask().getId());

break;

case "Delete task":

addTask(lastAction.getAfterTask().getId(), lastAction.getAfterTask().getTitle(), lastAction.getAfterTask().getDescription(),

lastAction.getAfterTask().getDeadline(), lastAction.getAfterTask().getPriority());

break;

case "Modify Task":

modifyTask(lastAction.getBeforeTask().getId(), lastAction.getBeforeTask().getTitle(),

lastAction.getBeforeTask().getDescription(), lastAction.getBeforeTask().getDeadline(),

lastAction.getBeforeTask().getPriority());

break;

}

}

}