THE ENGINEERING DESIGN METHOD

PHASE 1: PROBLEM IDENTIFICATION

The system should address the management of tasks and reminders in a way that allows users to have a good time management and better performance according to the completion of their commitments. The statement mentions that a hash table should be used to store tasks and reminders, taking into account a unique identifier and a description, allowing the user to add, modify and delete tasks and reminders. It should allow to manage priorities so the classification of tasks is "Priority" or "Non-priority", this will be carried out priority queues. In addition, an "undo" function should be implemented to allow users to revert actions performed in the system.

Need:

Users need to manage tasks and reminders, the hash table is taken into account to store the identifier and description of the task. For the management of “Priority” or “Non-priority” tasks, the use of priority is required, it is based on the fact that the tasks are stored in the priority queue depending on the importance, where the most important or priority one should be allowed to be. first and non-priority are counted in order of arrival. The user can add, modify or delete so an undo method is created that works with a priority queue.

Definition of the problem:

The objetive of the system is to provide users with a way to store, organize and access their tasks and reminders clearly and efficiently.

PROBLEM SPECIFICATION TABLE

CUSTOMER	Jeison Mejia
USER	All public
FUNCTIONAL REQUIREMENTS	RF1: Store tasks and reminders RF2: Edit tasks and reminders RF3: Delete tasks and reminders RF4: Show tasks in order of arrival. RF5: Show action history RF6: Undo action
CONTEXT OF THE PROBLEM	The system is based on task and reminders management allowing users to manage and organize their pending tasks efficiently. Users can add, modify, and delete tasks, as well as view an ordered list of tasks and reminders. In priority management, tasks

	can be categorized into “Priority” and “Non-priority.” The system also has the ability to undo actions.
NON-FUNCTIONAL REQUIREMENTS	The user interface should be easy to use so that users can add, modify, and delete tasks efficiently. It must be quick and efficient when it comes to managing tasks and reminders. The system must be able to adapt to an increase in the number of users and the number of tasks managed.

Functional requirements analysis table

Name or identifier	Store tasks and reminders		
Summary	The system uses a hash table to store task and reminder information.		
Tickets	Entry name	Datatype	Selection or repetition condition
	Title	String	
	Task description	String	
	Deadline	date	
	Priority	int	1. Priority 2. Not priority
Result or postcondition	The task is successfully stored in the hash table.		
Departures	Entry name	Datatype	Selection or repetition condition
	message	String	Added successfully or An error occurred while adding

Name	Edit task		
Summary	The system allows you to edit a task.		
Tickets	Entry name	Datatype	Selection or repetition condition
	id	String	Identifier of the task to edit
	option	int	1. Identifier 2. Title 3. Description 4. Priority date
	newValue	String	
Result or postcondition	The task has been edited successfully. A confirmation message is printed. In case of error, an error message is printed.		
Departures	Entry name	Datatype	Selection condition or
	message	String	"Edited successfully" or "An unexpected error occurred"

Name	Delete tasks		
Summary	The system allows you to delete a task that has already been created		
Tickets	Entry name	Datatype	Selection or repetition condition
	id	String	
Result or postcondition	The task has been successfully deleted. A confirmation message is printed. In case of error, an error message is printed.		
Departures	Entry name	Datatype	Selection or repetition condition
	message	String	"It was deleted correctly" or "An unexpected error occurred"

Name	Show tasks in order of arrival.

Summary	The system allows you to print the current task queue.		
Tickets	Entry name	Datatype	Selection or repetition condition
Result or postcondition	The system prints the list of tasks in the queue.		
Departures	Entry name	Datatype	Selection or repetition condition
	printqueue()	String	Text string representing the queue

Name	Show action history		
Summary	The system allows you to print the current task queue.		
Tickets	Entry name	Datatype	Selection or repetition condition
Result or postcondition	The system prints the action history		
Departures	Entry name	Datatype	Selection or repetition condition
	printHistory()	String	Text string representing action history

Name	undo action		
Summary	The system prints the last action performed on the system and gives the user the option to undo it.		
Tickets	Entry name	Datatype	Selection or repetition condition
Result or postcondition	Depending on the user's choice, the last action can be undone. A message is printed indicating whether the action was successfully undone.		
Departures	Entry name	Datatype	Selection or repetition condition

	printLastAction()	String	

PHASE 2: COLLECTION OF THE NECESSARY INFORMATION

For this task storage system we need to define different practices and approaches for the design, the necessary structures for the collection of information should be taken into account, this to know how it works and how each one can be interpreted, in this case the following was used:

Hash table:
A hash table is a data structure that allows the association of unique keys with values, like a dictionary. This is accomplished using a hash function that maps each key to an index on a table. The hash function takes a key as input and returns an index into the table, where the value associated with that key is stored. The hash table allows quick access to values through their keys, making it suitable for efficient data search and retrieval.

Linked list:
A linked list is a linear data structure consisting of nodes, where each node contains a value and a reference to the next node. The nodes are linked sequentially, allowing elements to be stored sequentially. Linked lists can be used to maintain a collection of elements in a specific order, and are especially useful when you need to efficiently add or remove elements in the middle of the sequence.

Queue:
A queue is a data structure that follows the "First In, First Out" (FIFO) principle, meaning that the first element to be added is the first to be removed. It is used to manage elements in a sequential order, where elements are added to the end of the queue and removed to the beginning of the queue. Queues are ideal for tasks that must be completed in the order in which they were added.

Priority Queue:
A priority queue is a data structure that stores elements along with an associated priority value. Items are organized so that items with a higher priority are processed or deleted before those with a lower priority. This means that, unlike a normal queue where the first element in is the first out (FIFO), in a priority queue, the order of processing is based on the priority assigned to each element.

Stack:
A stack is a data structure that follows the "Last In, First Out" (LIFO) principle, where the last element added is the first to be removed. It is used to manage items in a reverse order to

which they were added, making it suitable for tracking recent tasks or providing "undo" functionality where the last action performed can be efficiently undone.

PHASE 3: SEARCH FOR CREATIVE SOLUTIONS

To start looking for different solutions, we must be clear about what was requested in the statement of the integrative task, so we begin with:

1. To use the hash table, you must use an array that will have a node in each position. Each node inside will implement tasks with unique identifiers; to know the position of each node with the hash function, that identifier is used. When creating this system with arrays, it must be taken into account that it does not have infinite positions, so if we enter an identifier and it is not empty, a linked list is created.

2. The java.time library will be used for dates and times.

3. As for the priority queue, the tasks were organized in a heap that is an array of fixed size, it would be in order of arrival, each time a task is added it would enter the array and a method would be called that arranges the order according to the priority. When the capacity of the array is full, a method is called that enlarges the array with the data it already had.

4. To track the actions, it was thought to manage it with a stack, every time a task is created, edited or deleted from the controller, a task-type object would be created and added to the stack.

5. For non-priority tasks, we sought to directly use the queue, the first task that would enter would be the first that would leave, so it would directly remain in the queue.

PHASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS

Three different analytical, simulation and physical methods are used in this design process:

Analytical Models: Analytical models based on data structures such as heap, linked lists or priority queues will be considered. These models will allow a quantitative analysis of the performance and efficiency of each structure in the context of task management. Mathematical calculations will be performed to evaluate, for example, the time complexity of insertion, deletion and search operations in these structures.

Simulation Models: Simulation models will be developed to recreate the operation of the task management system in a virtual environment. This will include testing of typical use cases and usage scenarios. For example, simulations where high priority task are added could simulate and observe how the system responds. The simulation will provide information on how the system behaves under varyng conditions and help identify potential bottlenecks or challenges implementation.

Physical Models: Physical models, such as prototypes or mock-ups, will be considered to represent key system components. These tangible models will be useful for visualizing and testing concepts before implementation, which Will contribute to more informed decision making.

PHASE 5: EVALUATION AND SELECTION OF THE BEST SOLUTION

As the design process evolves, proposed solutions are subjected to rigorous evaluation. In this phase, the selection of the most appropriate solution to address the problem in question is sought, considering a variety of factors. Evaluation criteria include accuracy, efficiency, comprehensiveness, and adaptability. Furthermore, choosing the best solution can be based on economic, social and environmental factors, in addition to technical considerations.

Choice Type 1: Effectiveness and Efficiency:

Criterion A. Effectiveness of the Solution:
[2] Exact and precise solution.
[1] Approximate solution.

Criterion B. Efficiency:
[4] Constant efficiency.
[3] Efficiency greater than constant.
[2] Logarithmic efficiency.
[1] Linear efficiency.

Choice Type 2: Comprehensiveness and Adaptability:

Criterion C. Comprehensiveness and Adaptability:
[3] Comprehensive and highly adaptable solution.
[2] Various applications, but not comprehensive.
[1] Unique and not very adaptable solution.

Choice Type 3: Implementation and Compatibility:

Criterion D. Ease of Implementation:
[2] Supports basic hardware operations.
[1] Not fully compatible with basic hardware operations.

	Critrion A	Critrion B	Critrion C	Critrion D	Total
Hash Table	2	2	2	2	8
Linked lists	1	4	2	1	8
Stacks	2	3	3	2	10
queue	2	2	2	2	8
Priority Queue	2	3	3	2	10

	Critrion A	Critrion B	Critrion C	Critrion D	Total
Analytical Models	2	2	3	2	9
Simulation models	1	3	2	1	7
Physical models	2	1	2	2	7

PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS
For this phase of the system we must create the design documentation, including specifications of the data structures used, class diagrams and descriptions of the system functionalities. Technical reports are explaining the design and implementation of the system are also prepared. (They will be found in the documents folder)

PHASE 7: DESIGN IMPLEMENTATION
Phase 7 tells us that the system must be implemented in accordance with the proposed design. Extensive testing must be performed to ensure that the system works as intended.

Temporal analysis:

First Algorithm

Function enqueue(item: T)

newNode = New Node<T>(item) → O(1) = 1

If isEmpty() Then → O(1) = 1

```
        front = newNode → O(1) = 1
        rear = newNode → O(1) = 1
    Else
        rear.setNext(newNode) → O(1) = 1
        rear = newNode → O(1) = 1
    End If
End Function
```

The "enqueue" function has mainly constant-time (O(1)) operations, which means that its execution time does not depend on the size of the queue. However, it is important to note that in the worst case, if the entire queue needs to be traversed to find the last element, the operation could take linear time (O(n)), where "n" is the number of elements in The tail.

Second Algorithm:

```
Function search(key: K) -> V
    index = FunctionHash(key) → O(1) = 1
    If table[index] is null then → O(1) = 1
        Return null → O(1) = 1
    Else
        currentNode = table[index] → O(1) = 1
        While currentNode is not null do → O(n) = n
            If currentNode.getKey() is equal to key then → O(n) = n
                Return currentNode.getValue() → O(1) = 1
            End If
            currentNode = currentNode.getNext() → O(n) = n
        End While
```

```
Return null → O(1) = 1

End If

End Function
```

The time complexity of the "search" function is dominated by the while loop that loops through the collision list and has a time complexity of O(n) in the worst case, where "n" is the number of elements in the collision list. collisions. The rest of the operations are constant time, O(1).

Spatial Analysis

First Algorithm:

```
Function enqueue(item: T)
        newNode = New Node<T>(item)
        If isEmpty() Then
front = newNode
rear = newNode
        Else
rear.setNext(newNode)
rear = newNode
        End If
End Function
```

Guy	Variable	Number of Atomic Values
Entrance	item	1
Assistant	newNode	1
Exit	-	-

Spatial Complexity Total = Input + Auxiliary + Output = 2 = O(1)

Spatial Complexity Auxiliary = 1 = O(1)

Spatial Complexity Auxiliary + Output = 1 = O(1)

Second Algorithm:

```
Function search(key: K) -> V
    index = FunctionHash(key)
    If table[index] is null then
        Return null
    Else
        currentNode = table[index]
        While currentNode is not null do
            If currentNode.getKey() is equal to key then
                Return currentNode.getValue()
            End If
            currentNode = currentNode.getNext()
        End While
        Return null
    End If
End Function
```

Guy	Variable	Number of Atomic Values
Entrance	key	1
Assistant	index	1
	currentNode	n
Exit	-	-

Spatial Complexity Total = Input + Auxiliary + Output = 1 + 1 + n = O(n)

Spatial Complexity Auxiliary = 1 + n = O(n)

Spatial Complexity Auxiliary + Output = 1 + n = O(n)

Design:

Name	stack
abstract object	Stack = ((e1, e2, e3, ..., en), top)
Invariant	0 ≤ n ∧ Size(Stack) = n ∧ top = en

Operation	**Tickets**	**Departures**	**Operation type**	**Specification**
push()	item <T>		Modifier	Add an item to the stack
pop()		item <T>	Modifier	Removes and returns the top element of the stack
peek()		item <T>	Analyzer	Returns the top element of the stack without removing it
isEmpty()		Boolean	Analyzer	Check if the stack is empty
size()		int	Analyzer	Returns the number of elements in the stack

Name	queue
abstract object	Queue = ((e1, e2, e3, ..., en), front, back)
Invariant	0 ≤ n ∧ Size(Queue) = n ∧ front = e1 ∧ back = en

Operation	Tickets	Departures	Operation type	Specification
enqueue()	item <T>		Modifier	Queue an element to the end of the queue
dequeue()		item <T>	Modifier	Dequeue and return the element at the front of the queue
peek()		item <T>	Analyzer	Returns the top element of the queue without deleting it
rear()		item <T>	Analyzer	Returns the top element of the queue without deleting it
isEmpty()		Boolean	Analyzer	Check if the queue is empty
size()		int		Returns the number of elements in the queue.

Name	HashTable
abstract object	size, table, functionHash
Invariant	{key ¡= table(key)}

Operation	Tickets	Departures	Operation type	Specification
insert()	Key <K> Value<V>		Modifier	Insert a new node to the hash table
search()	Key <K>	Value<V> either null	Analyzer	Find a value associated with a key in the hash table
delete()	Key <K>	Boolean	Modifier	Delete a node with the specified key in the hash table

Name	PriorityQueue
abstract object	Hash= {size, hashFunction, keyEqualityFunction, table}
Invariant	

Operation	**Tickets**	**Departures**	**Operation type**	**Specification**
insert()	Task		Modifier	Insert a new task into the priority queue
remove()		Task	Modifier	Delete and return the task with the highest priority from the priority queue
isEmpty()		Boolean	Analyzer	Check if the priority queue is empty
ensureCapacity()			Modifier	Ensures there is sufficient capacity in the array
bubbleUp()			Modifier	Rearrange the array after adding an element to maintain priority queue ownership
bubbleDown()			Modifier	Reorganize the array after deleting an element to maintain priority queue ownership