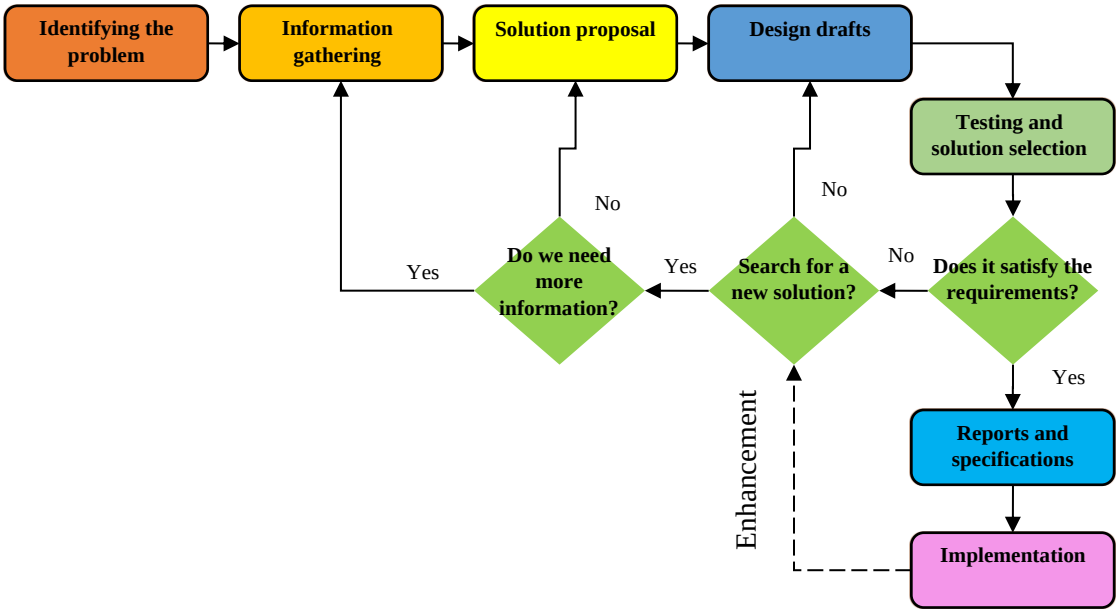
Context of the Problem

A videogame shop wants to innovate the way it offers its services, and requires a GUI based software which allows people from Cali to get familiar with the way the shop is intended to work.

Development of The Solution

To solve the proposed situation, the engineering method has been chosen to develop the solution following a systematic framework and according to the problematic given.

Based on the description of the engineering method given in the book *Introduction to Engineering* by Paul Wright, the following flux diagram was drawn, and will be followed according to the steps shown in it during the development of the solution.



Step 1: Identifying the Problem

Identifying symptoms and necessities

- People of Cali should be able to familiarize with the new shop's system and how it will work.
- The videogame shopping process should be simulated through software (the product).
- This software must have a GUI
- The software must inform the exit order of the clients, the value of each purchase and the order each client had their games packed up.

Definition of the Problem

The videogame store requires the development of a GUI software that simulates how the purchase process will work in the city of Cali.

Step 2: Information Gathering

With the purpose of being as clear as possible regarding the concepts involved in this process a search is done for the definitions of the informatic specific terms most closely related with the given problem. It is important to use renown and trustworthy sources for this search, to know which elements, take part in the problem and which ones don't.

Definitions

Sources:
https://www.informaticamilenium.com.mx/
https://softwareparatodo.com/
https://www.ecured.cu/

Software

Software is every application program and operative system that allow the computer can run smart tasks, directing the physical components or hardware with instructions and data through several different kinds of programs.

Simulation Software

A simulation software has the objective of facilitating or automating the modelling process for a real-world phenomenon, using mathematical formulas through programming. At its core, it is a program that allows the user to see what will happen after doing a specific action or set of actions, without having to do it in the real world.

Graphical User Interface (GUI)

A graphical user interface (GUI by its English acronym) is a program or environment that manages the interaction with the user basing itself on visual relations such as icons, menus, or pointers

Step 3: Creative Solutions Proposal

In this step, a handful of creative solutions should be proposed for the problem in question. However, because of the requirements of the problem given, the solution alternatives rely on a matter of form, because any solution proposed must be a simulation. The following solutions are proposed:

Alternative 1:

Ideally, the software will be divided in as many sections for the videogame purchase process as possible. In the statement given to us, there were four (4) sections. This process of division would be done via tabs or panes accessed by buttons on a menu. Furthermore, since the main process is a complex one with many abstract concepts like a payment queue, videogame shelves, and clients, the Object-Oriented Programming paradigm will be used for solving and modelling the problem.

The input will consist as a pane of its own and will take the specifications of each test case to be ran in the simulation. Each test case takes the number of cashiers available during the simulation, the number of clients and the number of shelves as main parameters for the simulation. Particularly, for each shelf an id and a number j of games must be provided, as well as the metadata related to the imputed game (Name, price, code, and stock). For each client that is added, a comma separated list

with the codes or names of the games (either or) is imputed, as well as the ID of each client. Finally, the user will decide which of the two provided sorting algorithms will be used to sort the list of stage 2.

The input will count as stage 1 in which the list of games from the catalogue for each client is provided. And the way they arrive at the tablet in stage 2 is determined by who is added first (which ever client gets added first is the first in line to use the tablet) and will also represent the amount n of minutes that will be added to each user (2^{nd} in line will get 2 minutes added). After clicking a submit or confirm button, each client is given a numeric random unique number which will be their code. This code will be used to identify each client. With this code the program will get the list of games for each client and sort it by distance to the tablet with the algorithm given in the input

With the ordered list done, the next stage is triggered, and a stack that represents the automated basket will begin stacking the games in the list. The length of the stack will be then get added to the client's total purchasing time (if they exit this stage with 3 games in the stack, 3 minutes will be added).

Finally, stage 4 begins. Up until now the clients have taken a quantity of $n + g$ minutes, where n is their position in the line to use the tablet (index 1) and g is the number of games they have in their basket. A client that used the tablet 4^{th} and gathered only 1 game will have 5 minutes to their name and will be placed before the clients who took **longer** or used the tablet **after** them, and before the clients who were **faster** or used the tablet **before** them. Using this rule, the clients will be queued in a single line to pay, and for each game in their basket they will take 1 minute paying for it. The way they exit follows the queue formed by this rule and will be displayed from top to bottom on screen, where the topmost client exited first, and the bottommost, last.

For each client in this screen a context menu will be triggerable which will show: their ID number, shop code, and the list of games they purchased, as well as the total price they paid for. On hover, or right clicking a game, the price, shelf, code, and name will be displayed, as well as the amount of it the client bought. This screen will be the final output and marks the end of the simulation. Additionally, this pane will show the real quantity of time the simulation took, as well as how many minutes the full interaction took for each client (from the moment they entered the store to the moment they exit it).

Alternative 2:

The software will be divided in four sections for the videogame purchase process. This process of division would be done via tabs or panes. Furthermore, since the main process is a complex one with many abstract concepts like a payment queue, videogame shelves, and clients, the Object-Oriented Programming paradigm will be used for solving and modelling the problem.

The input will consist as a text area of its own and will take the specifications of each test case to be ran in the simulation. Each test case takes the number of cashiers available during the simulation, the number of clients and the number of shelves as main parameters for the simulation. Particularly, for each shelf an id and a number j of games must be provided, as well as the metadata related to the imputed game (Name, price, code, and stock). Finally, the user will decide which of the two provided sorting algorithms will be used to sort the list of stage 2.

For each client on the other screen a pane with a text area will show: their ID number, shop code, and the list of games they purchased, as well as the total price they paid for, the games, the price, shelf, code, and name will be displayed, as well as the amount of it the client bought. This screen will be the final output and marks the end of the simulation.

Alternative 3:

The software will be divided in four sections for the videogame purchase process. Furthermore, since the main process is a complex one with many abstract concepts like a payment queue, videogame shelves, and clients, the Object-Oriented Programming paradigm will be used for solving and modelling the problem.

The input will consist of different lines submitted by console and will take the specifications of each test case to be ran in the simulation. Each test case takes the number of cashiers available during the simulation, the number of clients and the number of shelves as main parameters for the simulation. Particularly, for each shelf an id and a number *j* of games must be provided, as well as the metadata related to the imputed game (Name, price, code, and stock). Finally, the user will decide which of the two provided sorting algorithms will be used to sort the list of stage 2.

Later, for each client will be printed on console: their ID number, shop code, and the list of games they purchased, as well as the total price they paid for, the games, the price, shelf, code, and name, as well as the amount of it the client bought. This will be the final output and marks the end of the simulation.

Step 4: Transition from Ideas Formulation to Preliminary Designs

The first thing we do in this step is to discard ideas that are not feasible. In this sense we discard the Alternative 3 because it does not have graphical user interface (GUI), instead, it works with a command line interface (CLI) and the initial statement said that the simulation software must have a GUI.

Careful review of the other alternatives leads us to the following:

Alternative 1:

- It has multiple panes to be a complete interactive GUI simulation.
- It's possible to modify, create and delete all the objects with the different panes.
- It shows the process of all the stages and the final output in different panes.

Alternative 2:

- It only has two panes, one to put the input and the other one to show the final output or the solution.
- It only receives a single input and create the object, but it can't be modified later.
- It only shows the final output in a single pane.

Step 5: Evaluation and Selection of the Best Solution

Criteria

The criteria that will allow evaluating the alternative solutions must be defined and based on this result choose the solution that best meets the needs of the problem. The criteria we chose in this case are the ones we list below. Next to each one a numerical value has been established with the aim of establishing a weight that indicates which of the possible values of each criterion have the most weight (i.e., they are more desirable).

- *Criteria A.* Design quality GUI
  - [3] Good
  - [2] Fair
  - [1] Bad
- *Criteria B.* Completeness algorithm. Provides solution to any simulation's inputs
  - [3] Provide solution to any inputs
  - [2] Provide solution to some inputs
  - [1] Doesn't provide the right solution
- *Criteria C.* Ease of algorithmic implementation
  - [2] Simple implementation
  - [1] Complex implementation
- *Criteria D.* Process monitoring. Shows all the stages of the simulation
  - [2] Yes
  - [1] No

Evaluation

	Criteria A	*Criteria B*	*Criteria C*	*Criteria E*	*Total*
Alternative 1	*3*	*3*	*1*	*2*	*9*
Alternative 2	*1*	*3*	*2*	*1*	*7*

Selection

According to the previous evaluation, Alternative 2 should be selected since it obtained the highest score according to defined criteria.

Step 6: Preparation of Reports and Specifications

First, we establish the software requirements for a better understanding of the problem dividing them into two requirements: Functional and non-functional.

Definitions

Sources
https://reqtest.com

Functional Requirements

Any requirement which specifies what the system should do.

Non-functional Requirements

Any requirement that specifies how the system performs a certain function.

Establishment of Requirements

Functional Requirements

- The software must receive:
  - Store shells
    - Code of shell
  - Game catalog
    - Code of game
    - Amount of stock
    - Shell where it's located
    - Price
  - Store cashiers
    - Amount of cashiers
  - Client information
    - Identification
    - Game shopping list
      - Game's code
- The software must provide to client
  - Option to choose between two different sorting algorithms
- The software must show:
  - Client's exit in order
  - Client's value of each purchase
  - Client's packed games in order

non-functional Requirements

- The software must:
  - Register clients in arrival order
  - Use data structures such as:
    - Stack
    - Queue
    - Hash Tables
  - Use two different sorting algorithms with the following restriction:
    - Both need to have temporal complexity $O(n^2)$

Sort Algorithms

We select as our two sorting algorithms: Insertion Sort and Selection Sort. Then, we analyze their temporal complexity to determine if they truly have $O(n^2)$ as the requirement specifies.

Insertion Sort

Temporal Complexity Analysis

Algorithm	**Time**
Int n = arr.length;	1
If (n == 1 && getShelf(arr[0], shelf) == null)	C
return arr = new Integer[0];	1
For (int j = 1; j < n; j++)	n
Int key = arr[j];	n – 1
Int I = j - 1;	n – 1
String shelf1 = getShelf(arr[i], shelf);	C(n – 1)
String shelf2 = getShelf(arr[j], shelf);	C(n – 1)
If (shelf1 == null || shelf2 == null)	n – 1
arr[i] = (shelf1 == null) ? null : arr[i];	n – 1
arr[j] = (shelf2 == null) ? null : arr[j];	n – 1
Int compare = -1	n – 1
compare = shelf1.compareTo(shelf2);	n – 1
catch (NullPointerException e)	n – 1
while ((i > -1) && (compare >= 0))	(n(n+1)/2) + (n – 1)
arr [i + 1] = arr[i];	(n(n+1)/2)
i--;	(n(n+1)/2)
Arr [i + 1] = key;	n – 1
return arr;	1

$$T(A) = 3 + n + 10(n-1) + 2C(n-1) + C + 2\left(\frac{n(n+1)}{2}\right) + \frac{n(n+1)}{2}$$

$$T(A) = \frac{3n^2+25n+4nC-2C-14}{2}$$

Since C is a constant, we will replace it for number 1 in this case to get a simplified equation.

$$T(A) = \frac{3n^2+29n-16}{2}$$

The time complexity of this insertion sort is $O(n^2)$. Then, we can use this as our first sorting algorithm because it meets the time complexity restriction.

Spatial Complexity Analysis

Type	*Variable*	*Amount Atomic Values*
Input	arr	n
	shelf	n^2
Aux	n	1
	key	1
	i	1
	shelf1	1
	shelf2	1
	compare	1
Output	arr	n

$$Input + Aux + Output = n^2 + 2n + 6 = O(n^2)$$

The spatial complexity of this insertion sort is $O(n^2)$.

Selection Sort

Temporal Complexity Analysis

Algorithm	Time
Int n = arr.length;	1
If (n == 1 && getShelf(arr[0], shelf) == null)	C
return arr = new Integer[0];	1
For (int j = 0; j < n - 1; j++)	n
Int min = i;	n – 1
String shelf2 = getShelf(arr[i], shelf);	C(n – 1)
For (int j = i+1; j < n; j++)	(n(n+1)/2) + (n-1)
String shelf1 = getShelf(arr[j], shelf);	n(n+1)/2
Int compare = shelf1.compareTo(shelf2);	n(n+1)/2
If (compare < 0)	n(n+1)/2
Min = j;	n(n+1)/2
Int temp = arr[min];	n(n+1)/2
arr[min] = arr[i];	n(n+1)/2
arr[i] = temp;	n(n+1)/2
return arr;	1

$$T(A) = c + 3n + c(n-1) + 1 + \frac{8n(n+1)}{2} = \frac{8n^2+14n+2nc+2}{2}$$

Since C is a constant, we will replace it for number 1 in this case to get a simplified equation.

$$T(A) = 4n^2 + 8n + 1$$

The time complexity of this insertion sort is $O(n^2)$. Then, we can use this as our first sorting algorithm because it meets the time complexity restriction.

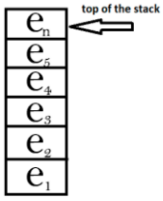
Spatial Complexity Analysis

Type	***Variable***	***Amount Atomic Values***
Input	arr	n
	shelf	n^2
Aux	min	1
	shelf1	1
	shelf2	1
	compare	1
	temp	1
Output	arr	n

$$Input + Aux + Output = n^2 + 2n + 5 = O(n^2)$$

The spatial complexity of this insertion sort is $O(n^2)$.

Stack ADT



$$Stack = \ll e_1, e_2, e_3, \dots, e_n >, top >$$

$$\{inv:\ 0 \leq n\ \wedge\ Size(Stack) = n\ \wedge\ top = e_n\}$$

$$Stack \rightarrow Stack$$
$$push:\ Stack\ x\ Element \rightarrow Stack$$
$$pop:\ Stack \rightarrow Stack$$

Stack

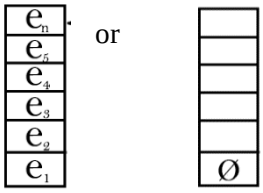
Builds an empty stack

$$\{pre:\ null\}$$
$$\{post:\ Stack\ s = \emptyset\}$$

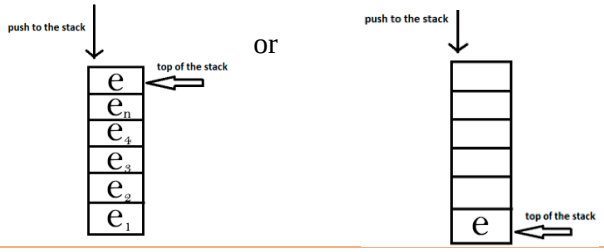
push

Adds the new element e to stack s

$$\{pre:\ Stack\ s = < e_1, e_2, e_3, \dots, e_n > and\ element\ e\ or\ s = \emptyset\ and\ element\ e\}$$



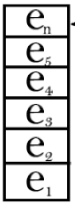
$$\{post:\ Stack\ s = < e_1, e_2, e_3, \dots, e_n, e > or\ s = < e >\}$$



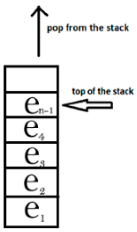
pop

Extracts from the stack s; the most recently inserted element.

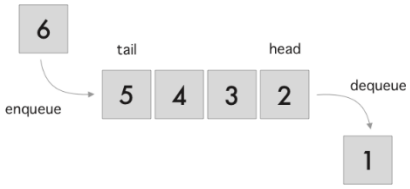
$\{pre:\ Stack\ s \neq \emptyset\ i.e.\ s = < e_1, e_2, e_3, \dots, e_n >\}$



$\{post:\ Stack\ s = < e_1, e_2, e_3, \dots, e_{n-1} >\}$



Queue ADT



$Queue = \ll e_1, e_2, e_3, \dots, e_n, front, back >$

$\{inv:\ 0 \leq n\ \wedge\ Size(Queue) = n\ \wedge\ front = e_1\ \wedge\ back = e_n\}$

$Queue \rightarrow Queue$
$enqueue:\ Queue\ x\ Element \rightarrow Queue$
$dequeue:\ Queue \rightarrow Element$

Queue

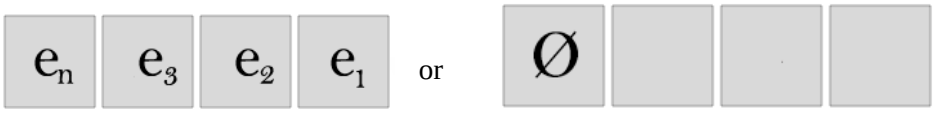
Builds an empty queue

$\{pre:\ null\}$
$\{post:\ Queue\ q = \emptyset\}$

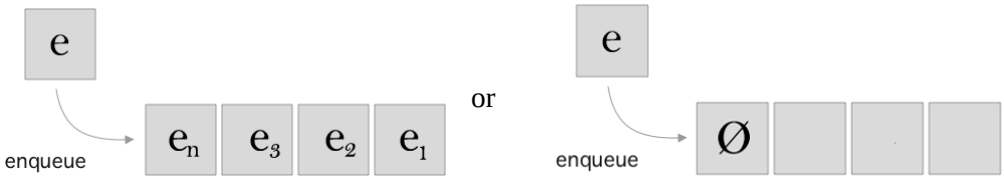
enqueue

Insert a new element e to the back of the queue q

$\{pre:\ Queue\ q = < e_1, e_2, e_3, \dots, e_n > and\ element\ e\ or\ q = \emptyset\ and\ element\ e\}$



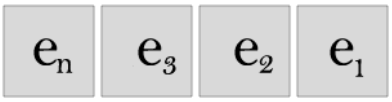
$\{post:\ Queue\ q = < e_1, e_2, e_3, \dots, e_n, e > or\ q = < e >\}$



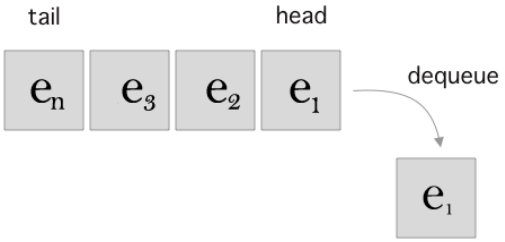
dequeue

Extracts the element in Queue q's front

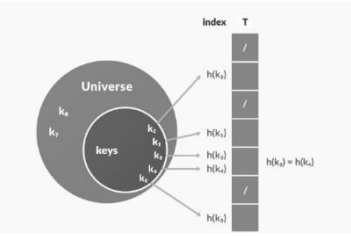
$\{pre:\ Queue\ q \neq \emptyset, i.e.\ q = < e_1, e_2, e_3, \dots, e_n >\}$



$\{post:\ Queue\ q = < e_1, e_2, e_3, \dots, e_{n-1} > and\ element\ e_1\}$



Hash Table ADT



$$HashTable = \ll < k_0, v_0 >, < k_1, v_1 >, < k_2, v_2 >, \dots, < k_n, v_n > \gg$$

$$\{inv:\ k_0 \in U \ \wedge\ h(k) \in A \mid \forall_k\, h(k) \neq A(v_0) \ \wedge\ h(k) \neq A(v_1)\ \}$$

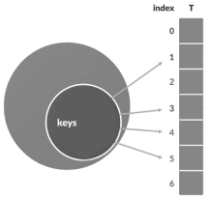
$$HashTable \rightarrow HashTable$$
$$insert: HashTable\ x\ Key\ x\ Value \rightarrow index$$
$$delete:\ HashTable\ x\ Key \rightarrow HashTable$$
$$search: HashTable\ x\ Key \rightarrow Value$$

HashTable

Creates an empty hash table

$$\{pre:\ null\}$$

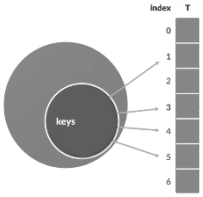
$$\{post:\ HashTable\ ht = \theta\}$$



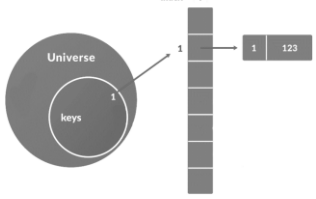
Insert

Inserts a new item into a table in its proper sorted, order according to the new item's search key.

$$\{pre:\ hashTable\ h = \ll < k_0, v_0 >, \dots, < k_n, v_n > \gg\ or\ h = \theta\ and\ Key\ k \neq \theta\}$$



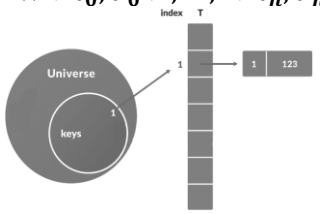
$$\{post:\ hashTable\ h = \ll < k_0, v_0 >, \dots, < k_n, v_n >, < k, v > \gg\ or\ h = < k, v >\ \}$$



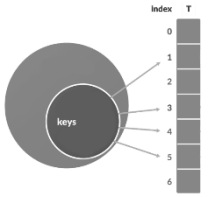
Delete

Deletes an item with a given search key from the table.

$\{\boldsymbol{pre}: \boldsymbol{hashTable\ h} = \ll < \boldsymbol{k_0}, \boldsymbol{v_0} >, \ldots, < \boldsymbol{k_n}, \boldsymbol{v_n} > \gg \boldsymbol{and\ Key\ k \neq \theta}\}$



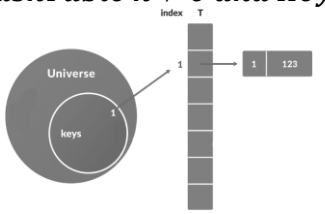
$\{\boldsymbol{post}: \boldsymbol{hashTable\ h} = \ll < \boldsymbol{k_0}, \boldsymbol{v_0} >, \ldots, < \boldsymbol{k_n}, \boldsymbol{v_n} >, \gg \boldsymbol{or\ h = \theta}\}$



Search

Retrieves an item with a given search key from a table.

$\{\boldsymbol{pre}: \boldsymbol{hashTable\ h \neq \theta\ and\ Key\ k \neq \theta}\}$



$\{\boldsymbol{post}: \boldsymbol{hashTable\ h \neq \theta\ and\ return} < \boldsymbol{k}, \boldsymbol{v} >\}$

Mockups

Main Pane

Shelves Pane

Games Pane

Clients Pane

Output Pane

Stages Pane

Test Cases

Scenarios Configuration

Name	Class	Scenario
setUp1	DefaultHashTableTest	vacío
setUp2	DefaultHashTableTest	• "Car", 1 • "Cat", 3 • "Dog"., 6

Test Goal: Check if a value is added to the hash

Class	Method	Scenario	Inputs Values	Result
DefaultHashTable	Insert	setUp1	"Car", 234	The object is added to the hash table
DefaultHashTable	Insert	setUp1	"Car", 234 "House", 375 "Dog", 426 "Cat", 426 "Lol", 426	The number of objects that the hash table allows are added after that it throws an exception.

DefaultHashTable	Insert	setUp1	“Car”, 234 “House”, 234 “Dog”, 234	Objects with the same key are added to the hash table.

Test Objective: Check if a desired object is removed from the hash table				
Class	**Method**	**Scenario**	**Inputs Values**	**Result**
DefaultHashTable	Delate	setUp2		It is verified that the index of key 3 was eliminated
DefaultHashTable	Delate	setUp2		It is verified that when deleting all the values of the table it does not overflow
DefaultHashTable	Delate	setUp1		It is verified that when deleting a non-existent value the table overflows

Test Objective: Check that the data can be retrieved from the hash table				
Class	**Method**	**Scenario**	**Inputs Values**	**Result**
DefaultHashTable	Search	setUp1	“Car”, 234	It is verified that the entered value is found in the table and can be retrieved
DefaultHashTable	Search	setUp2		It is verified that when searching for a value that does not exist in the table, a null value is returned
DefaultHashTable	Search	setUp1	“hello”, 3 “bye”, 3	It is verified that you can search for two values that have the same key

Name	**Class**	**Scenario**
setUp1	DefaultStackTest	vacío
setUp2	DefaultStackTest	• Client(1006016477, 102, 3) • Client(1006013423, 123, 4)

Test Goal: Check if a value is added to the hash				
Class	**Method**	**Scenario**	**Inputs Values**	**Result**
DefaultStack	Push	setUp1	Client(1006016477, 102, 3)	The object is added to the stack
DefaultStack	Push	setUp1	Client(1006014234, 110, 56) Client(1006015435, 106, 2) Client(1006013213, 1312, 231)	Objects are added in order and are saved by adding the last one at the beginning of the stack

Test Objective: Check if a desired object is removed from the hash table				
Class	**Method**	**Scenario**	**Inputs Values**	**Result**
DefaultStack	Pop	setUp2		Check that all elements of the stack can be removed
DefaultStack	Pop	setUp2		It checks that all the elements of the stack can be eliminated and if it can be eliminated and the stack is empty, it triggers an exception