Engineering Method Integrative Task 1

Samuel Navia Quiceno (A00405006)
Simón García (A00371828)
Juan Camilo Criollo Cifuentes(A00402515)

Phase 1: Problem identification

Description: A toy store in the city has had a substantial increase in sales and is in need of a system which allows the employment of an online store where their products as well as their users can manage everything. The store has suggested making a preliminary desktop version so they can make validations of the program operation. The validation will be done by a single user which should be able to navigate through the model and cause all possible events. Afterwards, the system will need to be uploaded to a webpage so it can generate the expected value the toy store is looking for. The problem must be solved since a delay in shipments, loss of information or faulty hand checked information could affect the store efficiency with delivering orders which could potentially drive the toy store off the market. As sales are increasing, it is easier for workers who are keeping track of all products, users and sales do commit errors which could translate into some major production issues later on.

Causes: The major causes to this problem rely on the fact that the sales are growing and more users seem to be interested in the store's products, but the current system is outdated and could lead to potential mistakes in the production process. As the user base grows along with the amount of sales, manual labor will tend to cause many mistakes as the influx increases over time. Keeping track of information in the local systems, or physical invoices would tend to get messy and sometimes even confusing.

Symptoms: With the mentioned causes above, the store could encounter problems like loss of information of customers, products and the order of sales. This could cause users to wait for longer periods of time than they should've since the priority system should be present at the moment the sales are done. Additionally, manually performing this type of work is prone to errors, which could result in sending the wrong packages, incorrect quantities (either too much or too little) or, in the worst case, failing to send an order altogether. While local systems may suffice for small stores, a growing market demands an online presence for the toy store to remain competitive against other stores that already have established, fully operational websites.

Problem definition: A toy store is staying behind the market's demand, and in order to keep up with the sales they are in need of an online webpage which would help them keep track of customers, products and sales.

Requirement Specifications:

Client:	Toy Store
Users:	Toy Store Customer, Toy Store manager and employees
Functional requirements:	● Req#1: Register clients ● Req#2: Create products in the product catalog ● Req#3: Delete products from the catalog ● Req#4: Update product information in the catalog ● Req#5: Undo actions ● Req#6: Place orders ● Req#7: Manage order shipments
Context of the problem:	● A toy store has come to need a web page since the sales are increasing and their current methods to manage customers, products and sales has become inefficient. The store has asked for a system that uses various data structures that allow the customers to place their orders and the store employees to keep track of their customers, products and the shipments. The system must be able to register new clients, create, delete and update products. The program must also allow users to undo actions, place new orders and manage the order of shipments efficiently taking into account the product priority and the order the products came through so the store can act appropriately.
Nonfunctional requirements:	● Usability: The program must be user friendly so clients are able to place their orders easily and for employees to edit the catalog or attend the shipments efficiently. ● Scalability: Since the store is experiencing a growth in sales, the system must be able to grow according to the store needs. ● Efficiency: The system must be able to undertake all its operations quickly and in a precise manner. Information shouldn't be lost in the process.

Identifier and Name:	Req#1 Register clients		
Summary:	The system is able to register a new customer with the necessary information: name, address, email and password. After the information is saved, a “successful registration” message will be printed. If any error were to happen in the process a generic “error” message will be printed instead.		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	Name	String	● Mandatory
	Address	String	● Mandatory
	Email	String	● Must be a unique email ● Mandatory
	Password	String	● Mandatory
Results or postcondition	The system allows a customer to register himself after the name, address, email and password have been entered correctly. The system will then use a Hash Table using the division method in order to create a key where all the information “customer type” will be saved and it can be accessed later. After the information is saved, a message stating “the customer has been registered successfully” will be printed. If any error were to occur during the process a general “error message” will be displayed.		
Outputs	**Output Name**	**Data type**	**Format**
	Successful registration	Text	“The customer has been registered successfully.”

	General Error	Text	“There has been an error in the registration process.”

Identifier and Name:	Req#2 Create products in the product catalog		
Summary:	Create Products in the Product Catalog is to allow the toy store to add new products to the catalog, ensuring that all relevant information is captured and the entered data is validated to maintain inventory integrity and organization. This will facilitate product management and the shopping experience in the future.		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	productCode	String	● Must be a unique identifier. ● Mandatory
	productName	String	● Mandatory
	productDescription	String	● Mandatory
	productPrice	double	● Mandatory
	productPriority	int	● Number between 1 and 5 ● Mandatory
Results or postcondition	After creating a product, it is successfully added to the catalog. All the necessary information must be added including the code, name, description, price and priority. The system then will add the product to a hash table to contain the newly added product.		

Outputs	**Output Name**	**Data type**	**Format**
	Success Message	String	Message that states "The product has been added successfully"
	Error Messages	String	Message stating "The product couldn't be added because of duplicated code"

Identifier and Name:	Req#3 Delete products from the catalog		
Summary:	is to allow users to remove products from the inventory using the product code. If the code is valid and exists, the product is deleted. If not, an error message will appear.		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	productcode	String	• Must Exist within the system • Mandatory
Results or postcondition	After attempting to delete a product, if the productcode is valid and exists in the catalog, the product is successfully removed from the catalog. If the productcode is invalid or does not exist, an error message is displayed, and the catalog remains unchanged.		
Outputs	**Output Name**	**Data type**	**Format**

	successMenssage	String	Message stating “the product has been deleted successfully”
	errorMessage	String	Message stating “The product has been deleted successfully”

Identifier and Name:	Req#4: Update product information in the catalog		
Summary:	Updates product information in the catalog to allow users to modify the details of existing products.Users can update information such as product name description,price, stock and category		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	Modification type	int	● Must be one of the printed options of possible modifications to an existing product.
	productcode	String	● Must exist in the catalog ● Mandatory
	newProductCode	String	● Cannot exist in the catalog already. ● Mandatory
	productName	String	● Mandatory

	productDescription	String	● Mandatory
	productPrice	double	● Mandatory
	Product priority	int	● Mandatory
Results or postcondition	First the system will ask to enter the code of the product that wants to be changed. If it exists, then it will ask to enter a type of modification it wants to be done to the product (change code, name, description, product, price or priority). Afterwards the user must enter the new value for that product. If the code doesn't exist and error message will be displayed. In the case the code wanted to be changed and the entered new code already exists within the catalog then an error will also appear.		
Outputs	**Output Name**	**Data type**	**Format**
	successMessage	String	"The information was added successfully"
	errorMessage	String	"There was an error finding the code you entered.
	DuplicateNewCode	String	"The entered code for which you want to replace the code of the product already exists."

Identifier and Name:	Req# 5: Undo actions
Summary:	allows users to revert the last action taken on the system, such as adding, updating or deleting a product. This feature helps prevent errors and allows users to easily correct them.

Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	actionId	String	-Must not be empty. -Must be a valid ID of the last action.
Results or postcondition	After attempting to undo an action, it will depend on the Stack that contains strings that identify each action taken. Depending on the string the system will revert whatever action the user did right before selecting that option. The reversible actions include: ● Adding a customer ● Adding a product ● Modifying a product ● Deleting a product ● Placing an order		
Outputs	**Output Name**	**Data type**	**Format**
	successMessage	String	Simple text ("Action undone successfully.")
	errorMessage	String	Simple text ("No action to undo or invalid ID.")
	currentCatalogState	Object or List	Object or list representing the current state of the product catalog after the undo action.

Identifier and Name:	Req# 6: Place orders

Summary:	Allows users to create and submit orders for products in the catalog. The order will have the email of the user, the date and the products that are ordered will be ordered in terms or produc priority.		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
	customerEmail	String	● Mandatory
	productList	List or Array	● Mandatory
Results or postcondition	After the user enters the email the system will validate that the email exists. If it does, then it will print all the products that are in the catalog. Afterwards the user will be able to write down one by one the code of the products he wants to add to the order. After pressing 0 then the program will continue to register the order. If the email doesn't exist an error will be thrown (likewise if the product list is sent empty).		
Outputs	**Output Name**	**Data type**	**Format**
	orderConfirmation	String	"The order was entered successfully"
	Non Existent email	String	"There was an error entering the order. The entered email doesn't exist."
	Empty code list	Object or List	"There was an error entering the order. The list of product codes was left empty"

Identifier and Name:	Req# 7: Manage order shipments		
Summary:	It allows te workers of the toy store to know which product must be produced and packed and where it must be shipped with all the order information corresponding to the customer.		
Inputs	**Input name**	**Data Type**	**Conditions for valid values**
Results or postcondition	After the		
Outputs	**Output Name**	**Data type**	**Format**
	Next Product	String	The massage will contain the product and the order that is belongs to. EX: “The next product in line up is X and it belongs to the order Y”.
	Empty orders	String	“There are no placed orders so nothing is needed to be produced. Try again later”.

Phase 2: Gathering the necessary information

Stacks: Is a type of dynamic set that allows to delete the element from the set that was most recently inserted. This means that if the numbers {1, 2, 3} were added in order into the stack, the delete option will get rid of number 3 since it was the most recent addition to the set. This order of operations makes stacks famous for the implementation of the LIFO policy which stands for “last-in, first out”. There are two main operations in a stack: “push” and “Pop”. Push would mean insert an element into the stack, while pop would delete the last element from the stack. One additional difference between push and pop is based on the parameter of the operation. A push element must specify the value of the element, while pop receives no parameter, and will always remove the last-in element (Cormen et. al., 2009).

Graphic representation of a stack and basic pseudocode:

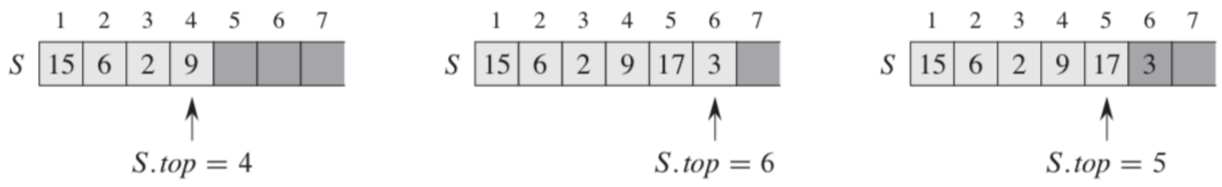


Figure 1. (Cormen et. al., 2009).

Figure 1 represents a stack S that holds up to 7 elements. S.top makes reference to a pointer informing which was the last added element.

```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top - 1
4      return S[S.top + 1]
```

Pseudocode 1. (Cormen et. al., 2009).

The pseudocode above represents a general idea of the 3 main operations that should make part of a stack.

Queues: Is a type of dynamic set that allows to delete the element from the set that has been the longest inside the set, in other words, the first one to be added. In contrast with stacks, if the number {1, 2, 3} were added in the same order inside the queue, the delete option will get rid of number 1 since it was the first added and it has spent the longest inside the structure. This makes the queue apply the FIFO policy, which stands for “first-in, first-out”. Similarly to stacks, queues hold two main operations “enqueue” (insert) and “dequeue” (delete). Every new element that is enqueued will be added to the back of the queue structure, and it will only be dequeued when all the other elements in front were dequeued first (Cormen et. al., 2009).

Graphic representation of a queue and basic pseudocode:

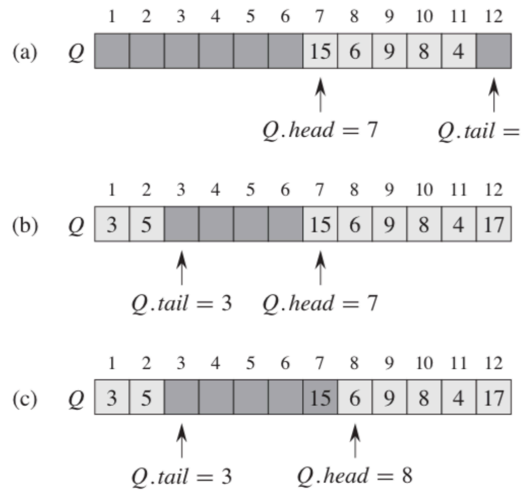


Figure 2. (Cormen et. al., 2009).

Figure 2 represents a Q with 12 elements which uses a circular reference. Q.head represents the oldest element in the queue and Q.tail points towards the last added element. When adding a new element into the queue, since it has a circular configuration, the last elements will go into the first empty spaces.

```
ENQUEUE($Q, x$)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.length$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$

DEQUEUE($Q$)
1  $x = Q[Q.head]$
2  if $Q.head == Q.length$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

Pseudocode 2. (Cormen et. al., 2009).

The Pseudocode above has the basic instructions for the two main operations in a queue.

Priority Queue: Similar to a regular queue, a priority queue is a dynamic data structure that allows the removal of the first-in element in the structure. The difference relies on the fact that each element carries an arbitrary priority value called a “key” which lets the structure identify the “most important element” and which should be dequeued first. Adding the number {1, 2, 3} into the priority queue while assuming a max-priority queue, the elements would need to be reorganized into {3, 2, 1} since the most “valuable” element is 3, followed by 2 finally by 1. The most usual algorithm to sort these elements in the priority queue is heap sort, with either max-heap or min-heap, depending on the desired order and contexts (Cormen et. al., 2009).

Pseudocode taking into account a max-priority queue using heapsort:

```
HEAP-MAXIMUM($A$)
1  return $A[1]$
```

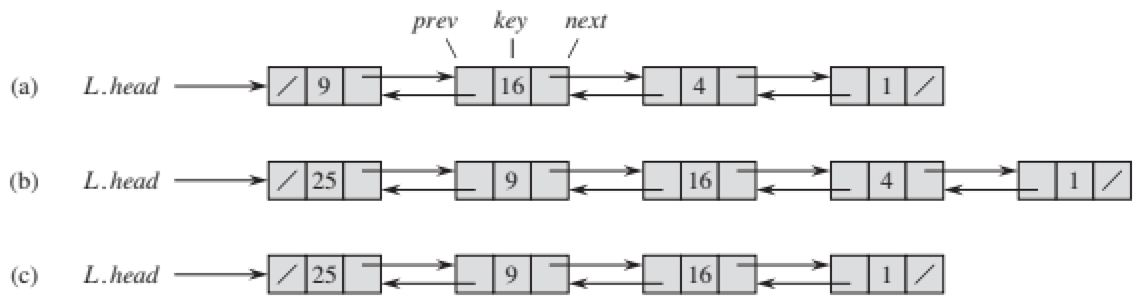
```
HEAP-EXTRACT-MAX($A$)
1  if $A.heap\text{-}size < 1$
2      error "heap underflow"
3  $max = A[1]$
4  $A[1] = A[A.heap\text{-}size]$
5  $A.heap\text{-}size = A.heap\text{-}size - 1$
6  MAX-HEAPIFY($A, 1$)
7  return $max$
```

```
HEAP-INCREASE-KEY($A, i, key$)
1  if $key < A[i]$
2      error "new key is smaller than current key"
3  $A[i] = key$
4  while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5      exchange $A[i]$ with $A[\text{PARENT}(i)]$
6      $i = \text{PARENT}(i)$
```

```
MAX-HEAP-INSERT($A, key$)
1  $A.heap\text{-}size = A.heap\text{-}size + 1$
2  $A[A.heap\text{-}size] = -\infty$
3  HEAP-INCREASE-KEY($A, A.heap\text{-}size, key$)
```

Pseudocode 3. (Cormen et. al., 2009).

Linked List:
A linked list is a data structure in which objects are arranged in linear order. However, unlike a matrix, where the linear order is determined by the matrix indexes, the order in a linked list is determined by a pointer to each object.



As shown in Figure 10.3, each element of a doubly linked list L is an object with
an attribute key and two other pointer attributes: next and pre. The object may
also contain other satellite data. Given an element x in the list, x:next points to its
successor in the linked list, and x:pre points to its predecessor. If x:pre D NIL,
the element x has no predecessor and is therefore the first element, or head, of
the list. If x:next D NIL, the element x has no successor and is therefore the last
element, or tail, of the list. An attribute L:head points to the first element of the
list. If L:head D NIL, the list is empty.
A list may have one of several forms. It may be either singly linked or doubly
linked, it may be sorted or not, and it may be circular or not. If a list is singly
linked, we omit the pre pointer in each element. If a list is sorted, the linear order
of the list corresponds to the linear order of keys stored in elements of the list; the
minimum element is then the head of the list, and the maximum element is the
tail. If the list is unsorted, the elements can appear in any order. In a circular list,
the pre pointer of the head of the list points to the tail, and the next pointer of
the tail of the list points to the head. We can think of a circular list as a ring of elements. In the
remainder of this section, we assume that the lists with which we
are unsorted and doubly linked.

Searching a linked list
The procedure LIST-SEARCH.L; k/ finds the first element with key k in list L
by a simple linear search, returning a pointer to this element. If no object with
key k appears in the list, then the procedure returns NIL.

```
LIST-SEARCH($L, k$)
$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return $x$
```

To search a list of n objects, the LIST-SEARCH procedure takes ,.n/ time in the
worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose key attribute has already been set, the LIST-INSERT
procedure “splices” x onto the front of the linked list.

```
LIST-INSERT($L, x$)
$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL
```

Deleting from a linked list
The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE($L, x$)
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$
```

Hash Tables:

Definition of Hash Tables

Hash tables are data structures that work like maps or dictionaries, where unique keys are linked to values. They use a hash function to calculate an index in an array, where key-value pairs are stored.

Structure of a Hash Table

1. Keys and Values: Each entry in the hash table has a key and an associated value.
2. Hash Function: A function that takes a key and gives back an index in the table. The quality of this function is very important for the performance of the hash table.
3. Array: The hash table is implemented as an array where the elements are stored.

Features

- **Fast Access:** The main advantage of hash tables is that they allow fast access to values, usually in O(1) time for searching, inserting, and deleting.
- **Collisions:** These happen when two different keys produce the same index. They need to be managed well to keep the performance of the table.

- **Resizing:** When the table gets full, it can be resized to maintain good performance, often by doubling its size and reinserting existing entries.

Types of Collision Handling

1. **Chaining:** Each index in the table contains a list of all entries that are mapped to that index.
2. **Open Addressing:** When a collision occurs, the next available index in the table is searched.

Basic Operations

- **Insertion:**
  - To add a key-value pair, the index is calculated using the hash function, and the value is placed in that position. If there is already a value for that key, it can be updated.
- **Searching:**
  - To find a value using its key, the hash function is used to get the index, and the stored value is checked. In case of a collision, the list is followed (in chaining) or the next indices are searched (in open addressing).
- **Deletion:**
  - To remove a key-value pair, the corresponding index is found, and the value is deleted. In chaining, the element is removed from the list, and in open addressing, the space is marked as empty.

Bibliographic references:

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). Massachusetts Institute of Technology.

Phase 3: Looking for creative solutions

Method to produce original ideas: Attribute listing. An attribute list, in this context, would be made by gathering and ordering all the possible information we can think of about the product that has been asked to make. The attribute is a breakdown of all the components which are essential to this product. Afterwards there should be an idea about how to change or innovate for each attribute.

Subject: Desktop version of a program to manage customers, products and sales of a toy store.

General attributes:

- **User Interface:** The app should have a simple ui which allows the user to navigate the functionalities easily and operate his needs without much effort.

    - If the user is a customer, it should be able to select every available option in order to process his orders.
    - If an operator of the toy store is using the program, he should be able to manipulate the product catalog with various options, and see the production queue depending on order and priority.
- **Data Storage and Management:** The program should be able to store the information of the products, customers and the orders.
    - The customers and the products should be saved in the form of a TAD that is able to create information for new customers and products.
        - The TAD should be specific specially for the products at the moment of explaining the procedures that allow them to edit the catalog of products.
        - The TAD could then be added in the form of a hash table to access the information easily by the users.
        - One option to work the list of customers and products is through a simple or double linked list which is then saved in a generic structure of each type.
        - Another option could be independent ArrayList could be used to save the information of each object, since each has their own specific characteristics and can be instantiated.
    - There should be another process to store the orders different to the products and customers since they need to be processed in the same order they came in, and the priority should also be taken into account.
        - A queue structure is specially useful in this case since the specified order of dispatch was clear as a first-in, first-out operation.
        - Additionally, the queue should have a priority attribute that allows more "valuable" products to be dispatched first than the rest.
        - Another option could be a different linked list that is reordered based on the priority criteria.
        - The linked list could point to the first added product so then it can be removed, and the pointer can move to the next candidate for shipment.
- **Undo functionality:** The program was specified to have an undo functionality which allows it to take back the last action taken by the user.
    - A useful structure to hold this type of information is a stack, since the undo process was specified as a last-in, last-out process.
    - The stack should be able to support all the actions listed above, creating a type of action history.
    - There could be a level of additional depth to the number of actions that can be undone.
        - iPhones Undo button only work for the last immediate change, while in docs document for example, there is a large amount of changes that can be backtracked using ctrl + z.
    - An alternative option could be using a double linked list that holds the last version of all the elements at a specific moment.

        - If the undo option is pressed, then the previous element becomes the last element and the other one is erased from the list.
- **Order Management:** The order management is a quality of the program that should allow the employees of the toy store work in delivering the most important product.
    - The relative importance of the product is determined by which product has been in queue the longest, and which product has the highest priority to be dispatched.
    - The structure, as discussed above, should be a priority queue that is able to manage multiple orders without losing track of which must be shipped first.
    - One alternative to the queue could be worked through a linked list that adds at the end the last element added and the remove option takes the first element out and the head is then moved to the subsequent position.
    - Another way of attempting this queue is by making an ArrayList that holds all the products that have been added in the queue.
        - This would make each customer to have his own ArrayList of products and then those would have to be reordered in a larger one.
        - When an element is erased then every element in the ArrayList must be moved to the position behind them so the structure doesn't have any empty places.

Creative solution 1: Lists and Simple Linked Lists

- Create independent simple linked lists which hold the information of customers and products.
    - The simple linked list would allow you to create an object, delete the object or search for the object.
- Use a different double linked list to hold all the information of every action taken by the user so it can be then saved and have access to the previous state of the program.
- Use a List (ArrayList) to save the orders done by the users, which then will have to be rearranged so it has the right order taking into account priority and the time a customer made the order.
    - The arrayList shouldn't have any specified type so it can receive all types of products. Or use some type of inheritance to fix this criteria.

Creative solution 2: Hash-Table and ArrayList

- Create a hash-table which will hold the information of customers
    - The hash table will have the ability to enter the customers information such as their products whose search is O(1) and with this they have the capacity to access or modify the information of the customers, especially if there is a large list of clients and the method to handle the collision will be through chaining
    - The hash table will be useful too to save all the products from the toy store.

- Create a double LinkedList that will store all the user's actions, because if the user wants to revert the previous state of an action, they can do so.
- Create a ArrayList which hold the information of products
    - This structure is ideal for dynamically adding, updating and removing products.

    - Add, delete, modify products. The ArrayList allows you to access products through their index, which makes management easier in a store with a large catalog.
- Create ArrayList for Orders with Priority Queue:
    - Each customer's orders will be stored in an ArrayList within each customer hash table entry. This ArrayList will contain the products requested by the client.
    - With this implementation the workers will be able to ship orders based on the time the order was placed and the size of the order
    - The ArrayList will reorder the elements taking into account the priority of the ordered products.

Creative solution 3:Hash Table and Stack

- Create a hash-table will be hold the information of the customers
- Create a hash-table with products sold by the toy store
- Create stacks that will save the decision of the customers in the program
- Create a queue to save the order in which the customers place their orders.
- Create a priority queue with urgent and non-urgent shipments

Phase 4: Transition from the formulated ideas to preliminary designs

Discarded idea: *Creative idea 1:* The idea, even though with very careful programming could work, there are several issues concerning efficiency, error prevention and scalability.

Issues with efficiency: Using a simple linked list to hold the information of products and customers would make any function like creating, removing or searching O(n) since the entire list would be traversed for any process (Adding could be O(1) with some additional criteria). Utilizing a List to save the orders made by the customers, and sorting those orders with various comparisons every time an order is added, would prove inefficient.

Issued with error prevention: Using a general List where any object can be added would make it impossible for the program to check for non-compatible objects (A customer could be added to the product list). Another factor could be that having a normal double linked list, if not handled properly the states of the program could cause various mistakes including deleting a position that was not supposed to be deleted, problems adding the new state, etc.

Issues with scalability: The program should be able to hold different types of objects, not only products. Using generics with the implemented lists can limit the scalability if the toy store is looking to expand the functionalities or type of data being saved.

Possible creative ideas that could be implemented:

Creative 2:

The aim of this idea is to manage the number of customers using a hash table with chaining, as it seeks to achieve maximum efficiency while optimizing memory usage and ensuring excellent error handling. The related products will be stored as part of the customers' attributes within the hash table, maintaining quick access. It's important to note that the hash table index will have a limited size to preserve the O(1) efficiency, which is crucial for consistent performance.

Additionally, a doubly linked list will be used to store the customer's decisions, regardless of the type of decision, allowing for flexible and efficient tracking or reversal of actions. An ArrayList will be used to manage products, enabling the easy addition, updating, and deletion of specific products, providing a dynamic and easily accessible structure.

Lastly, an ArrayList with a priority queue will be implemented to manage the flow of orders based on both the size of the order and the time it was placed. This will allow orders to be processed in an organized manner, following predefined criteria and optimizing the logistical flow.

Together, this creative idea emphasizes maximizing efficiency through the hash table and ensuring easy accessibility with the ArrayList implementations, resulting in a well-structured and agile system.

Creative 3:
The third creative idea wants to improve efficiency and error prevention by improving the handling of multiple data structures. For customers, a hash table with linked lists is used while maintaining O(1) efficiency. This structure ensures consistent access to customer data and related products stored as attributes. All created orders will be saved in a hash table and will also hold the information of the customer. Inside the order there will be a priority queue that will hold the products in the order of priority and additionally based on the FIFO logistics.

For decision tracking, a single stack is implemented. Every action taken by the customer will be tracked with a unique ID so it can afterwards be undone if needed. This allows more fine-grained control when undoing actions, providing flexibility by allowing the reversal of specific actions without affecting others.

Product management is maintained in a different HashTable. Since products should be easily extracted, edited, etc. The hash table is a great option for the toy store to have a mutable structure for its old and upcoming products.

To manage the orders in which they were placed inside an order and added into a queue structure which is useful to ensure that the first customer that made an order is the first one to be served. Queue is an ideal structure because of its FIFO logistics, which works in the same way as a usual waiting line.

This idea focuses on increasing efficiency through the use of hash-table and improved stack management, while ensuring better error prevention with validation layers in the order flow.

Phase 5: Evaluation and selección of the best solution

Criteria:

A. The creative idea utilizes the structures asked by the client:
    a. [5] = The idea implements all the structures asked by the client.
    b. [3] = The idea implements some of the structures asked by the client.
    c. [1] = The idea implemented none of the structures asked by the client.

B. The efficiency of the structures that are implemented:
    a. [5] = The structures used have a time complexity of O(1)
    b. [4] = The structures used have a time complexity of O(logn)
    c. [3] = The structures used have a time complexity of O(nlogn)
    d. [2] = The structures used have a time complexity of O(n)
    e. [1] = The structures used have a time complexity of O(n^2)

C. Error prone during implementation:
    a. [5] = The probability of implementing the idea with errors is low
    b. [3] = The probability of implementing the idea with errors is high
    c. [1] = The probability of implementing the idea with errors is inevitable

D. Flexibility and scalability of the general solution:
    a. [5] = The program is extremely flexible and scalable
    b. [3] = The program is moderately flexible and scalable
    c. [1] = The program is not flexible and scalable

	Criteria A	**Criteria B**	**Criteria C**	**Criteria D**	**Total**
Solution 2	3	2	3	3	11
Solution 3	5	5	5	5	20