ACTORS AND THEIR GOAL(S)

Actor	Goal(s)
Customer	- Request an order
Clerk	- Registering an item - Registering a customer - Receiving an order - Printing and sending the invoice to the storage area
Worker	- Picking up the invoice - Retrieving the item(s) from the shelves and packing them - Updating the item stock(s) after shipment

Use Case 1: Register Item

Preconditions: Product is not already registered in the system.
Postconditions: Product is successfully registered and its information is saved.

Table 1.1: Use case Register Item

Actions performed by the actor	Responses from the system
1. **Clerk** issues a **request** to register a new **product**.	
	2. **System** asks for the **product's name, amount and price.**
3. Clerk enters the required data into the system.	
	4. System reads new product's name, then creates a new product. Creates an unique **id for the product**, then adds it to the **item list**.
	5. Outputs a success message with new product's id.

Use Case 1: Register Item(continued)

Table 1.2: Use case Register Item's Extensions

Extensions
4a. Invalid data type in a field when entering the data 1. System throws an error message and asks the clerk to correct that field. 2. Clerk changes that field.
4b. Missing data field when entering the data 1. System throws an error message and asks the clerk to enter that field again. 2. Clerk enters that field again.

Use Case 2: Register Customer

Preconditions: The customer is not already registered in the system and at least 18 years old.
Postconditions: The customer is successfully registered and his/her information is saved.

Table 2.1: Use case Register Customer

Actions performed by the actor	Responses from the system
1. **Clerk** issues a **request** to register a new **customer**.	
	2. **System** asks for the **customer's data**(name, surname, address, date of birth, social security number(ssn)…).
3. Clerk enters the data into the system.	
	4. System reads new customer's data, then creates a new customer according to that data. Adds the customer to the **customer list**.
	5. Outputs a success message with new customer's ssn.

Use Case 2: Register Customer(continued)

Table 2.2: Use case Register Customer’s Extensions

Extensions
4a. Invalid data type in a field when entering the data 1. System throws an error message and asks the clerk to correct that field. 2. Clerk changes that field.
4b. Missing data field when entering the data 1. System throws an error message and asks the clerk to enter that field again. 2. Clerk enters that field again.

Use Case 3: Receive Order

Preconditions: The customer (who creates the order) is already registered in the system.
Postconditions: The order is successfully received, invoice is created and sent to storage area.

Table 3.1: Use case Receive Order

Actions performed by the actor	Responses from the system
1. The **customer** notifies the **clerk** of the **items** he/she wants to buy from the item list.	
2. Clerk issues a **request** to receive a new **order**.	
	3. **System** asks for the **customer's ssn**.
4. Clerk enters customer's ssn into the system.	
	5. System asks for the items that the customer wants to buy.
6. Clerk enters the **items' data** into the system.	
	7. System checks the data and stocks, creates an **invoice** for the item(s) which are in **stock**, saves the other items(which aren't in stock at the moment) to **waiting list**(their invoice will be created once they are received from a **manufacturer**).
8. Clerk prints the invoice for available items and sends it over to storage area.	

Use Case 3: Receive Order(continued)

Table 3.2: Use case Receive Order's Extensions

Extensions
5. Customer is not registered in system 1. System throws an error message. 2. System asks for the customer ssn again.
7a. An item is not registered in the system 1. System throws an error message and notifies the clerk that the item isn't registered. 2. Clerk notifies the customer and asks if he/she wants to purchase other item(s) in the list. 3. If the answer is yes, clerk asks the system to continue and create an invoice for remaining items. If the answer is no, clerk asks the system to stop the process and exit.
7b. If the item's stock amount is not enough 1. System puts the unavailable part(the amount which isn't in stocks at the moment) to waiting list. 2. System creates an invoice when item comes from manufacturer.

Because we added an item list and the customer will choose the items that he/she wants to buy from list, therefore we won't encounter this extension.

Use Case 4: Process Order

Preconditions: The invoice of the item(s) that will be shipped is already created, printed and sent to storage area.
Postconditions: Order is successfully processed and the items are shipped. Items are updated as shipped in system.

Table 4.1: Use case Process Order

Actions performed by the actor	Responses from the system
1. **Worker** takes the **invoice** which is printed and sent by the clerk.	
2. The worker reads the invoice, takes the **items** from **shelves** and packs them.	
3. The worker ships the items and invoice to customer.	
4. The worker requests the system to mark the **order** as having been shipped.	
	5. System asks for **invoice's id**.
6. Clerk enters the invoice's id into the system.	
	7. The system updates itself by recording the information. Also updates the stocks.

Use Case 4: Process Order(continued)

Table 4.2: Use case Process Order’s Extensions

Extensions
7. Order can’t be found in system 1. System throws an error message and asks the clerk to check invoice’s id again. 2. Clerk checks the id and re-enters it.

Noun Analysis for the use cases

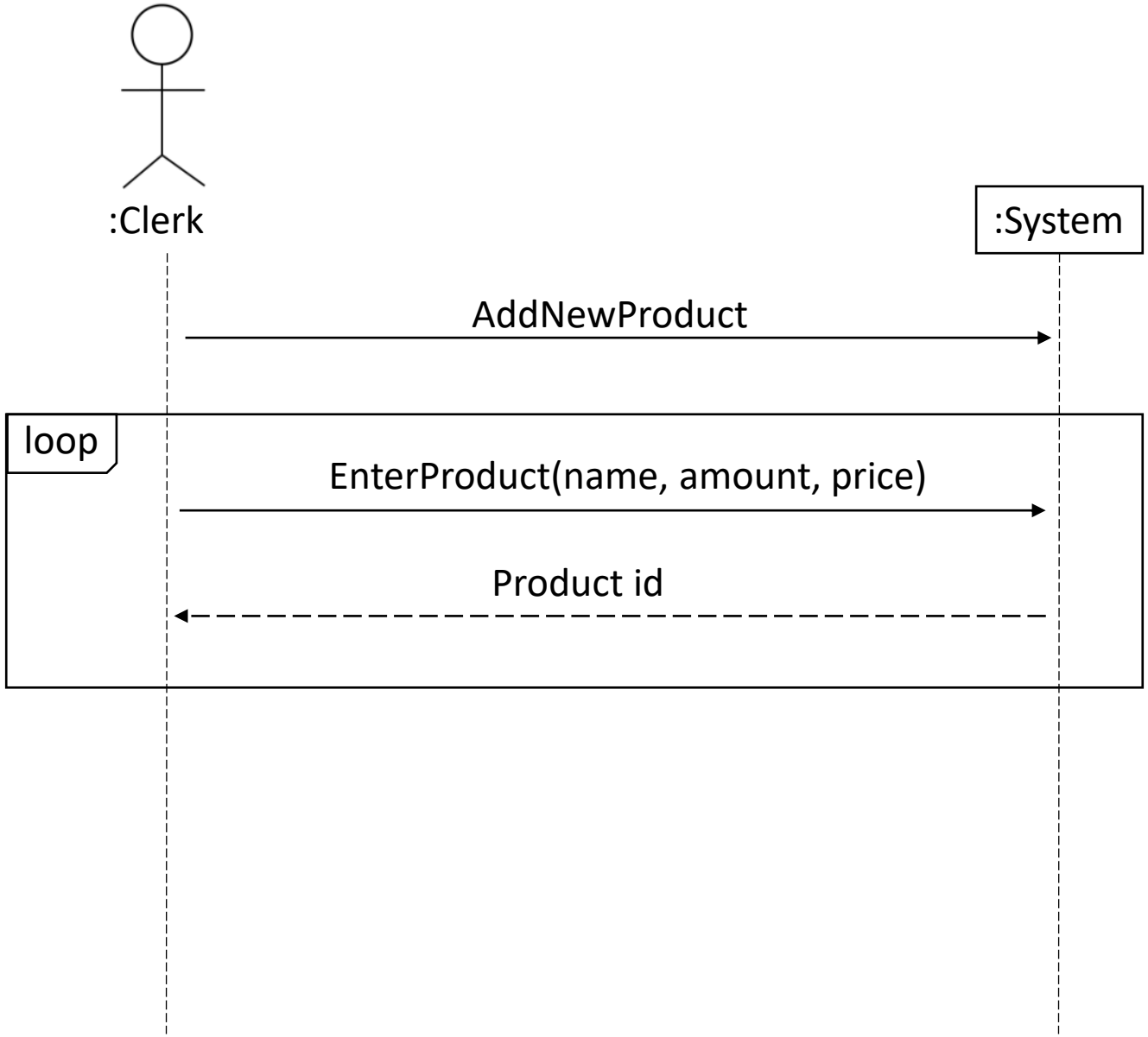
- Clerk
- ~~Request~~ → Is just a menu item so won't be a part of the data structures.
- ~~Product~~ → Item
- ~~System~~ → Not a class
- ~~Product's name~~ → Will be an attribute of item class
- ~~Id for the product~~ → Will be an attribute of item class
- Item List
- Customer
- ~~Customer's Data~~ → Will be the attribute of customer class.
- ~~Id for the customer~~ → Will be an attribute of customer class.
- Customer List
- Item
- Order
- ~~Customer's ssn~~ → Will be an attribute of customer class.
- ~~Items' data~~ → Will be seperated into parts(such as name, id, manufacturer…) and be attributes of item class.
- Invoice
- ~~Stock(item stock)~~ → We will use the warehouse class instead.
- Waiting list
- Manufacturer
- Worker
- ~~Shelve~~ → Would be a redundant class.
- ~~Invoice's id~~ → Will be an attribute of invoice class.

Another Conceptual Classes which aren’t part of noun analysis

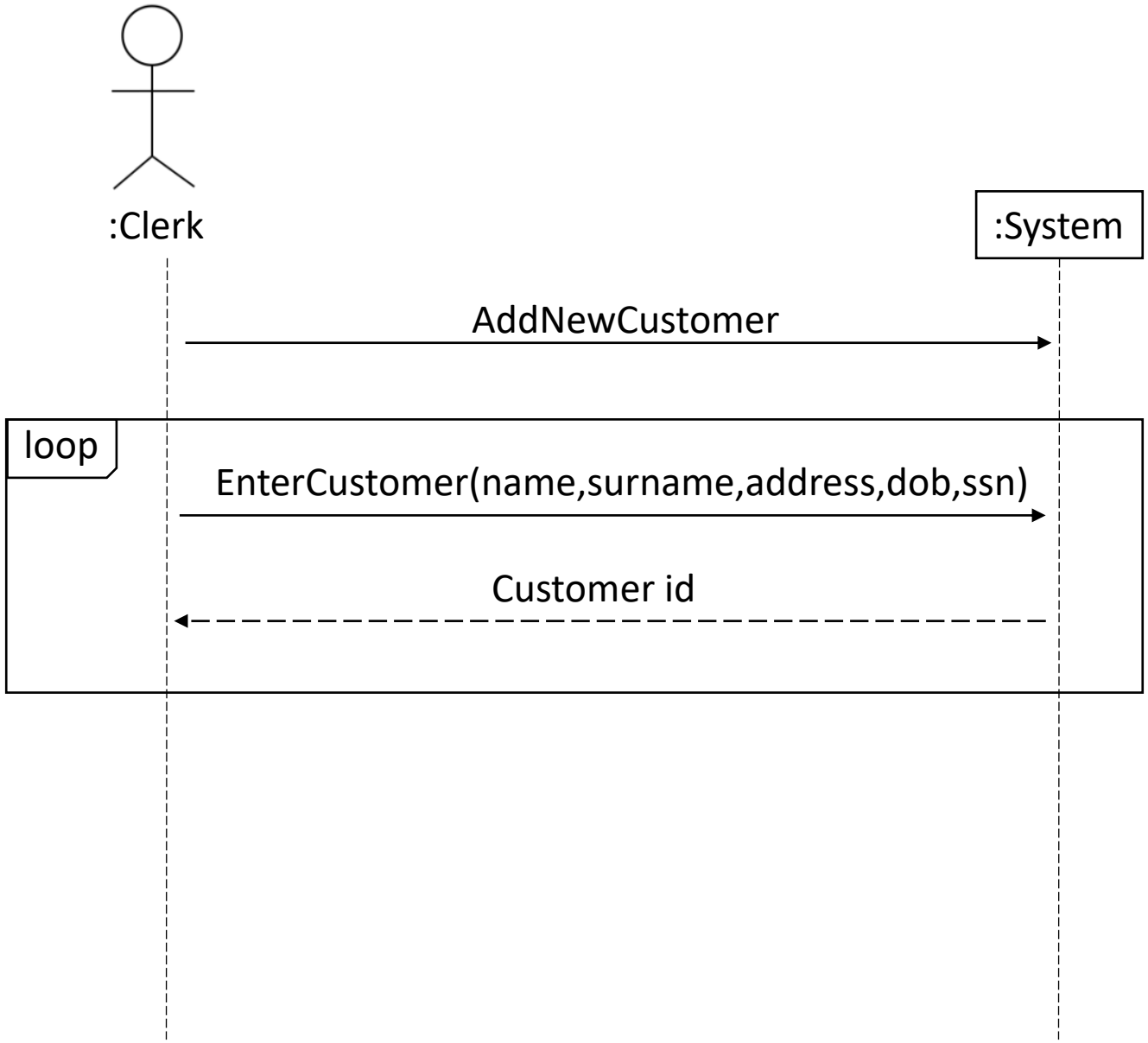
- Sales LineItem(Order LineItem) → Will record the sale of items.
- Warehouse → Will be like the controller class in a sequence diagram.

Deriving the System Sequence Diagrams of written use cases

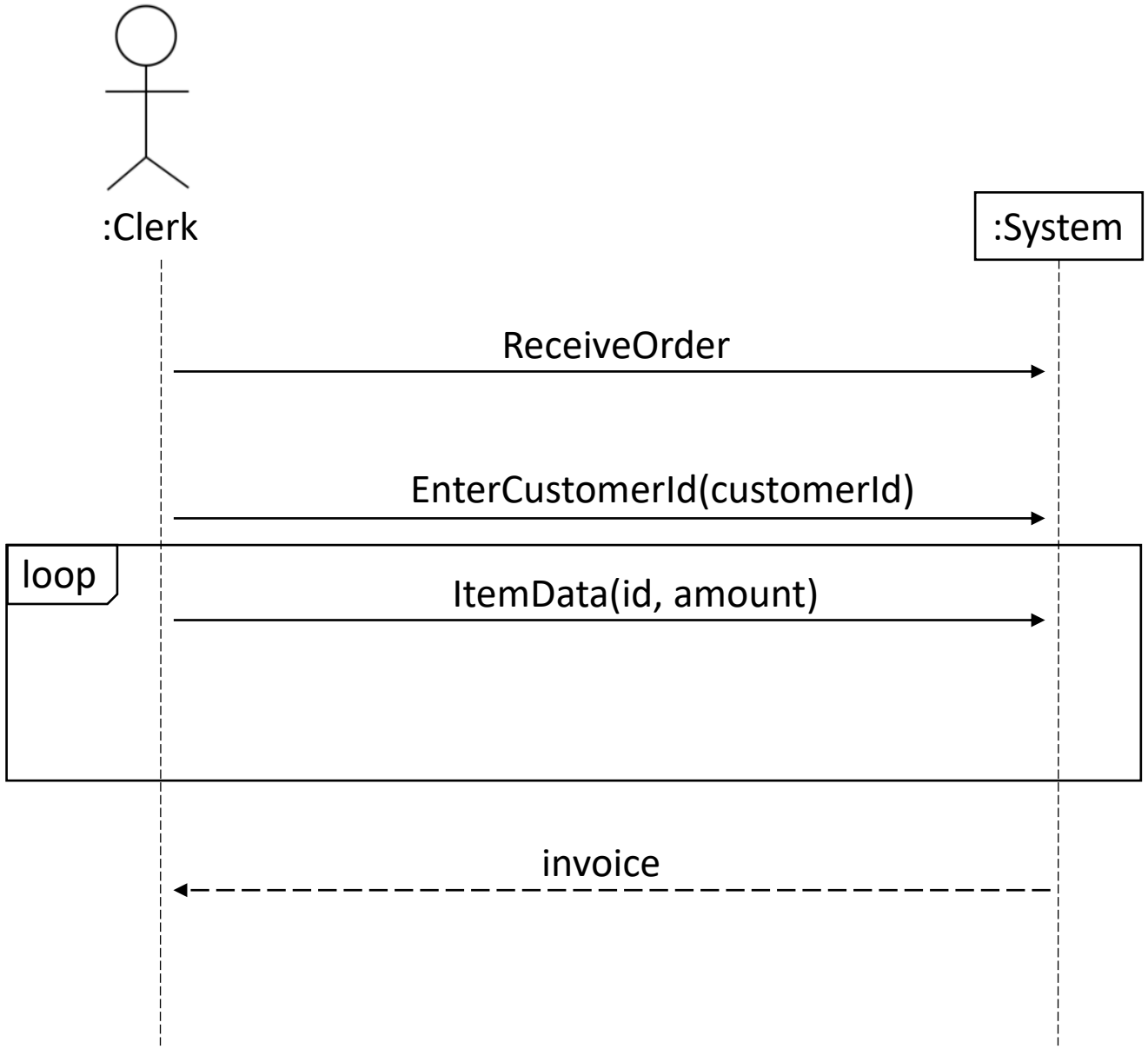
Register Item Scenario



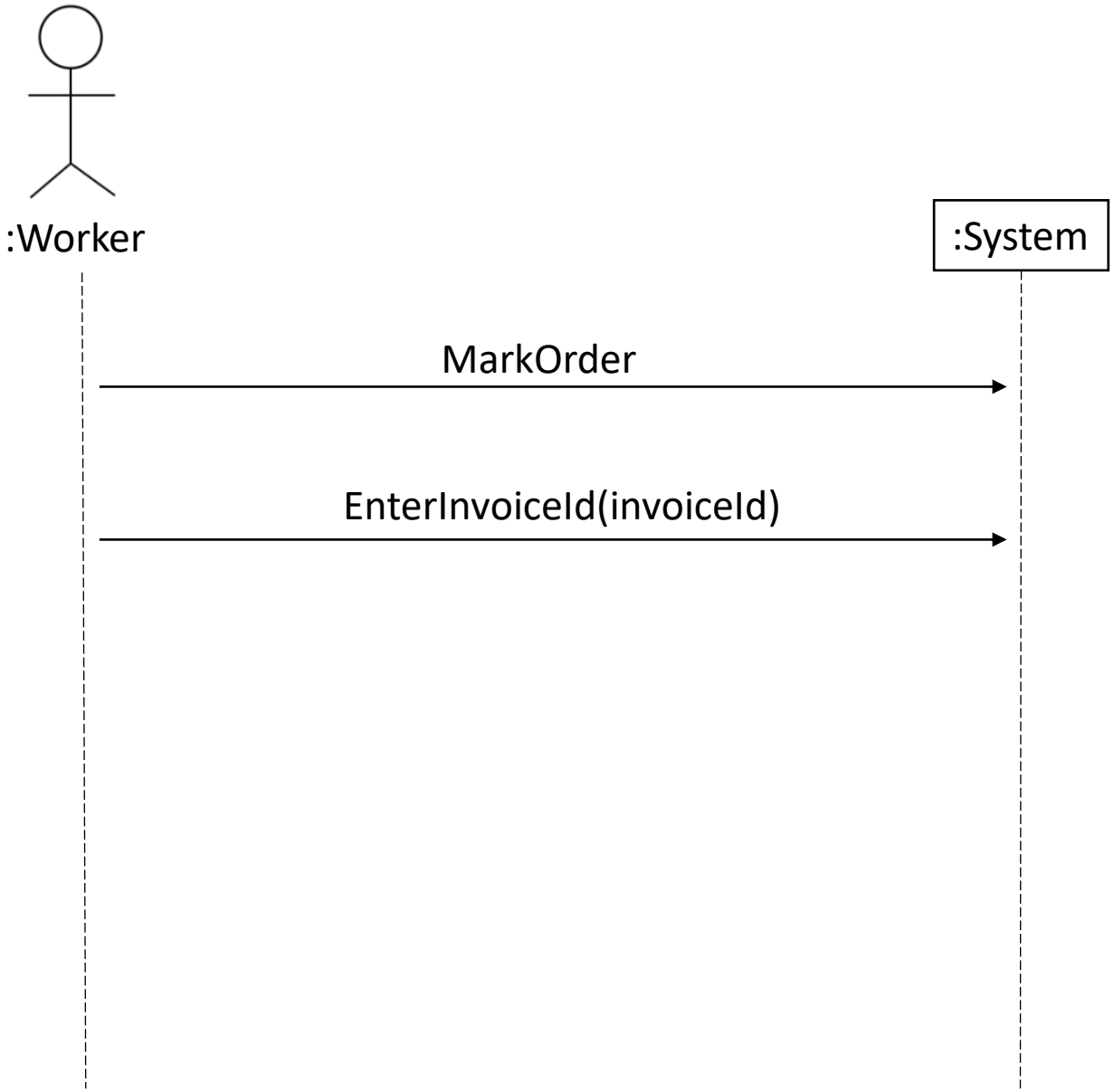
Register Customer Scenario



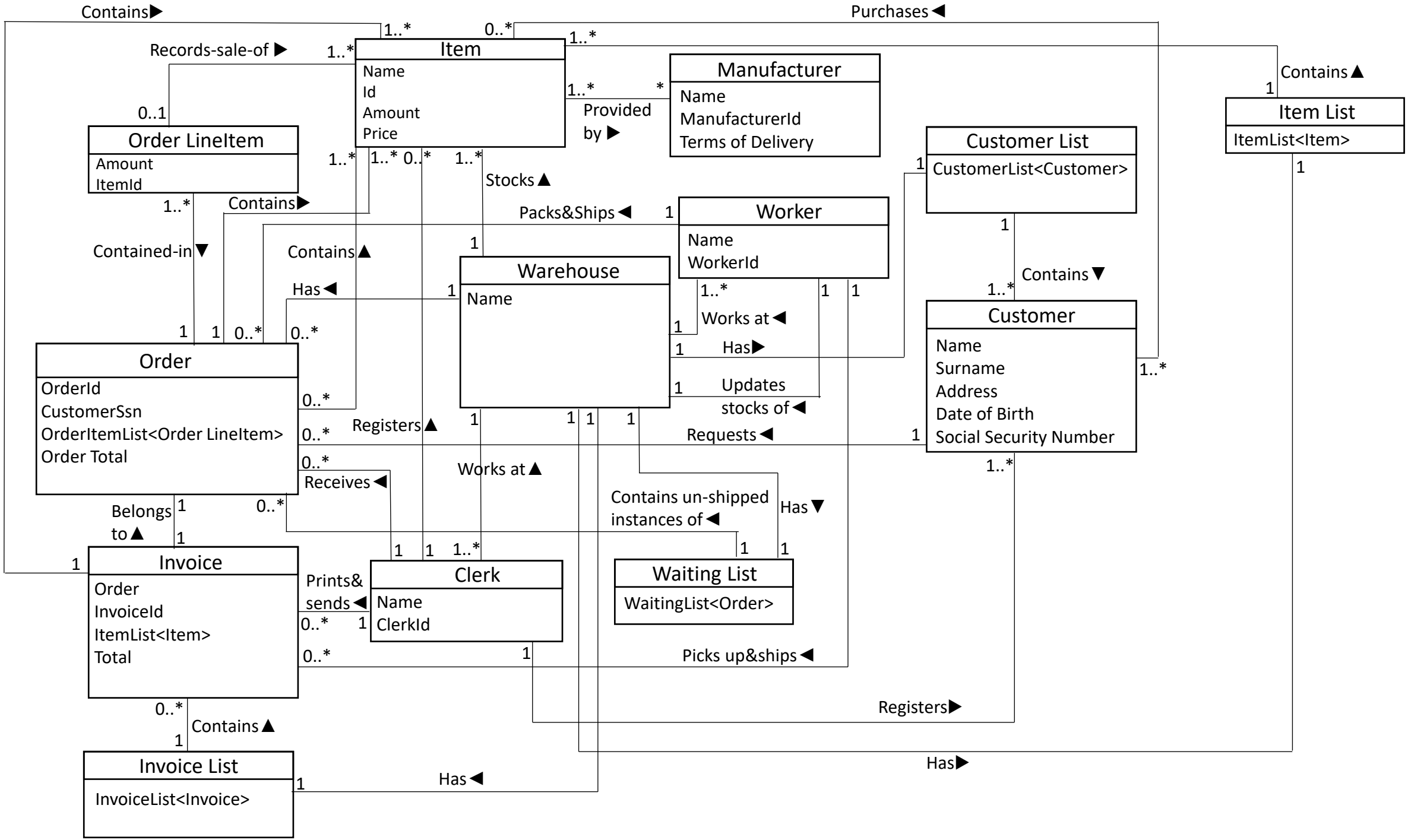
Receive Order Scenario



Process Order Scenario



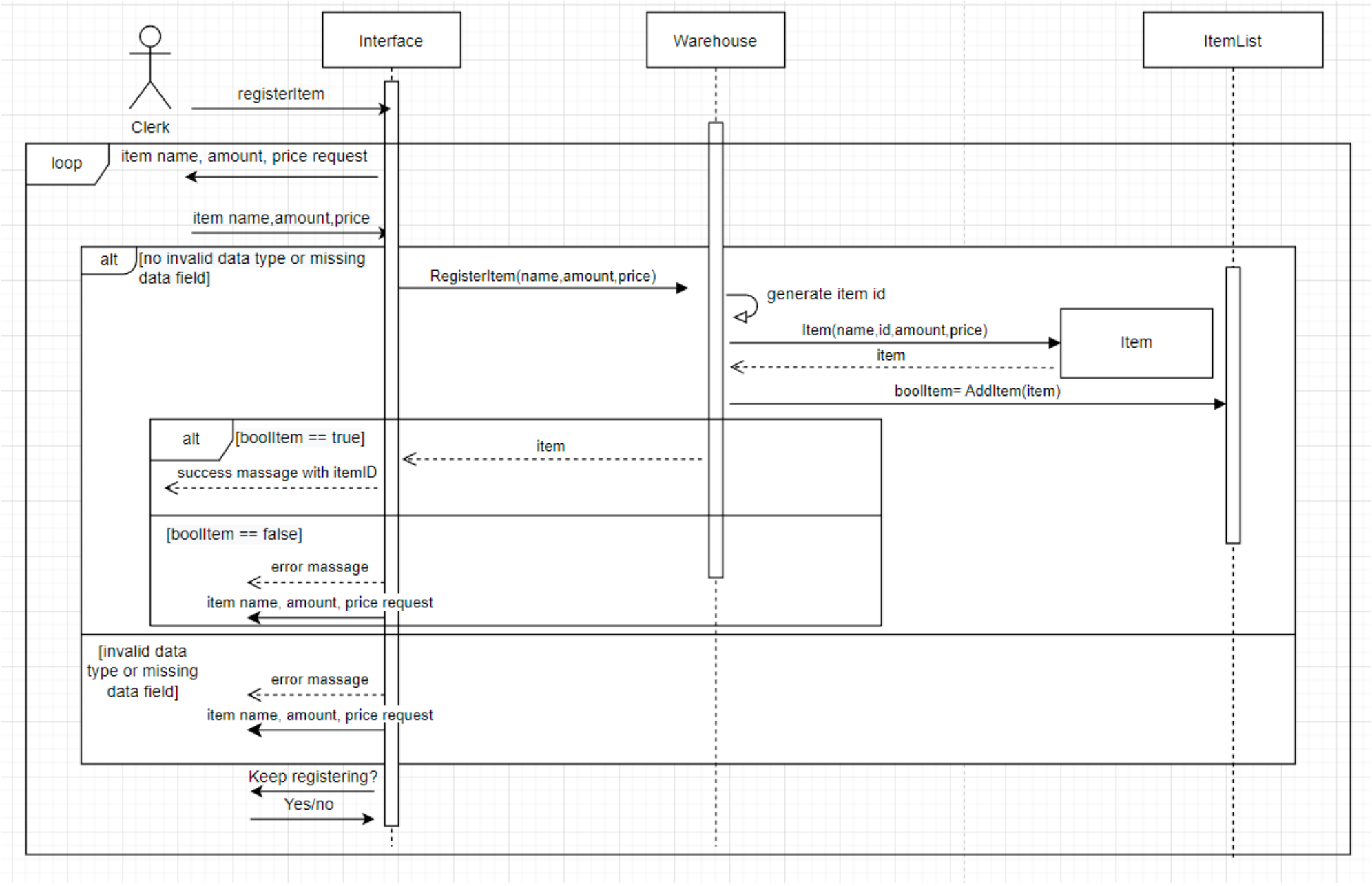
Constructing the domain model using documented use cases


Item
Name
Id
Amount
Price
Manufacturer
Name
ManufacturerId
Terms of Delivery
Order LineItem
Amount
ItemId
Item List
ItemList<Item>
Customer List
CustomerList<Customer>
Worker
Name
WorkerId
Warehouse
Name
Customer
Name
Surname
Address
Date of Birth
Social Security Number
Order
OrderId
CustomerSsn
OrderItemList<Order LineItem>
Order Total
Invoice
Order
InvoiceId
ItemList<Item>
Total
Clerk
Name
ClerkId
Waiting List
WaitingList<Order>
Invoice List
InvoiceList<Invoice>
Contains▶
Purchases◀
Records-sale-of ▶
Provided by ▶
Contains▲
Stocks▲
Contains▶
Packs&Ships◀
Contained-in▼
Contains▲
Has◀
Contains▼
Works at◀
Has▶
Updates stocks of◀
Registers▲
Requests◀
Works at▲
Receives◀
Belongs to▲
Contains un-shipped instances of◀
Has▼
Prints& sends◀
Picks up&ships◀
Registers▶
Contains▲
Has◀
Has▶

Designing the use cases developed in previous pages

Diagrams are made with draw.io

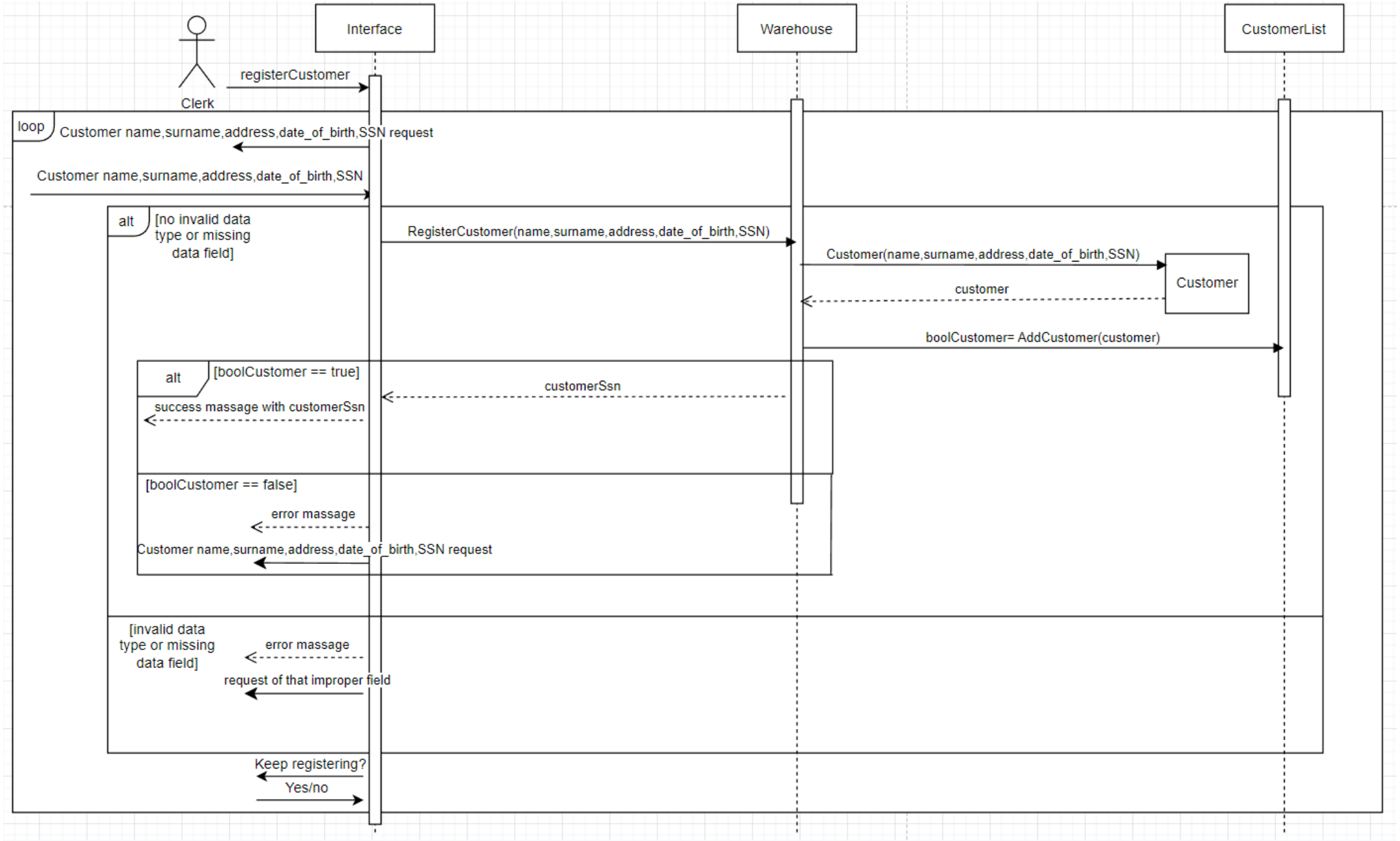
Sequence Diagram for the Use Case Register Item



GRASP Analysis for Register Item Sequence Diagram

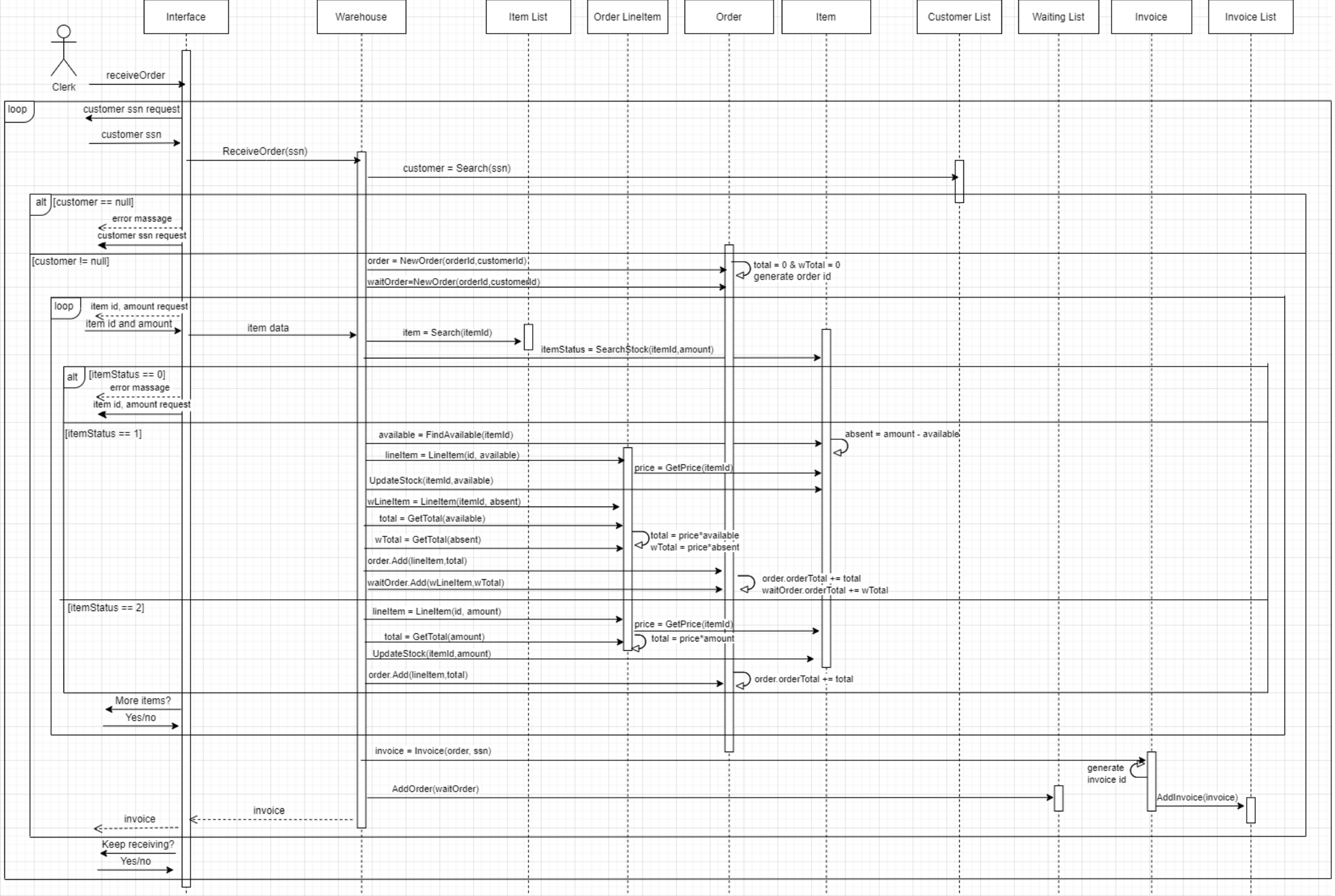
- **Controller:** `Warehouse` is the controller class in our sequence diagram because it seperates the interface and business layer. In this system, it represents the whole system and coordinates all of the system operations. Also, when a request comes from the UI layer object, `Warehouse` helps to determine what is that first object that receives the message from the UI layer objects.
- **Creator:** `Warehouse` is the creator class because it holds the necessary information to create an item. And `Warehouse` has the initializing data for `Item` class.
- **Information Expert:** `AddItem` method is added to the `ItemList` class because it holds all the items.
- **High Cohesion:** `ItemList` holds the items so `AddItem` method is added to it.
- **Low Coupling:** `AddItem` method isn't coming from `Item` class, so unnecessary coupling is avoided.

Sequence Diagram for the Use Case Register Customer



GRASP Analysis for Register Customer Sequence Diagram

- **Controller:** `Warehouse` is the controller class in our sequence diagram because it seperates the interface and business layer. In this system, it represents the whole system and coordinates all of the system operations. Also, when a request comes from the UI layer object, `Warehouse` helps to determine what is that first object that receives the message from the UI layer objects.
- **Creator:** `Warehouse` is the creator class because it holds the necessary information to create a customer. And `Warehouse` has the initializing data for `Customer` class.
- **Information Expert:** `AddCustomer` method is added to the `CustomerList` class because it holds all the customers.
- **High Cohesion:** `CustomerList` holds the items so we added the `AddCustomer` method to it.
- **Low Coupling:** `AddCustomer` method isn't coming from `Customer` class to avoid unnecessary coupling.

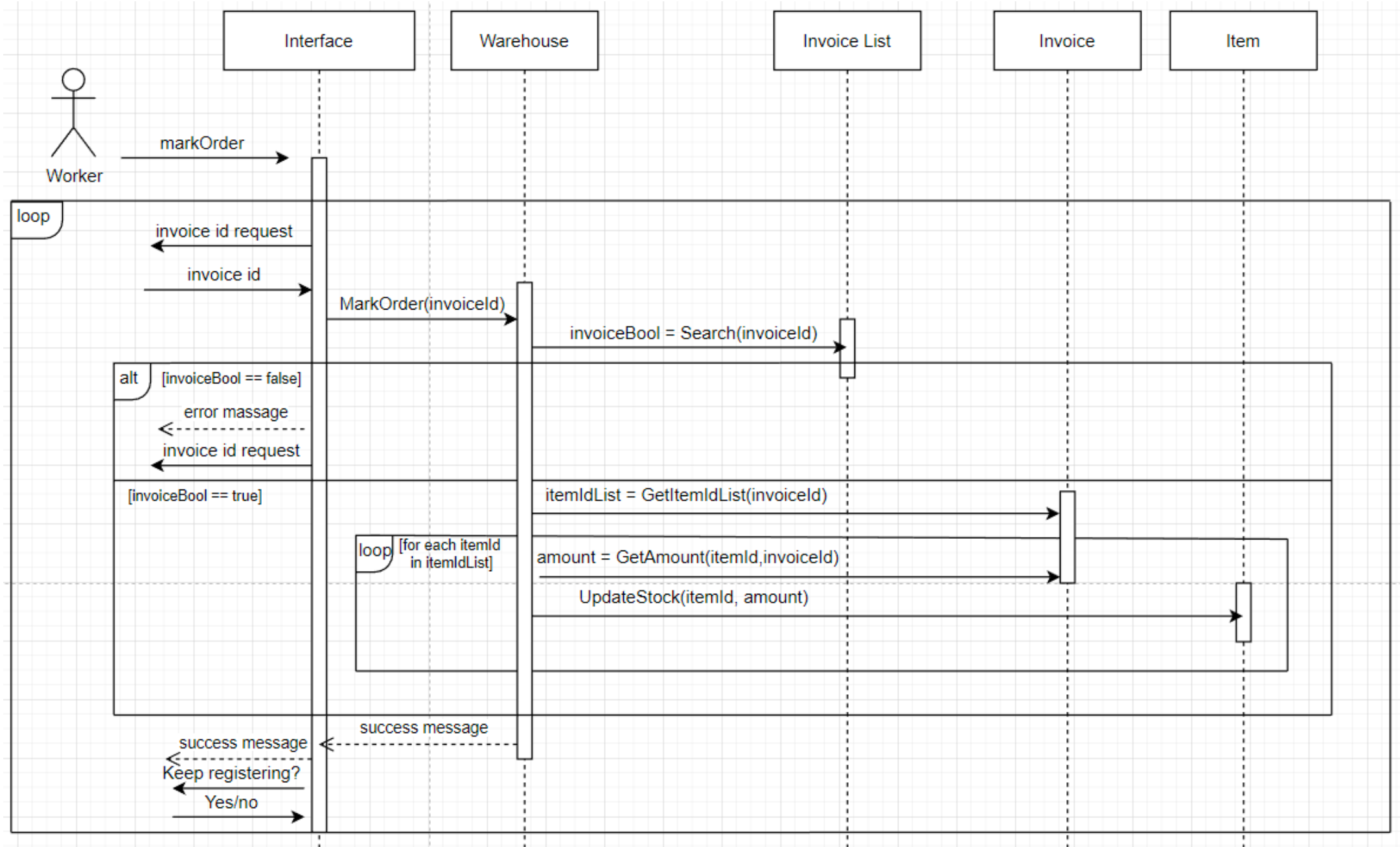


Sequence Diagram for the Use Case Receive Order

GRASP Analysis for Receive Order Sequence Diagram

- **Controller:** `Warehouse` is the controller class in our sequence diagram because it seperates the interface and business layer. In this system, it represents the whole system and coordinates all of the system operations. Also, when a request comes from the UI layer object, `Warehouse` helps to determine what is that first object that receives the message from the UI layer objects.
- **Creator:** `Warehouse` is the creator class because it holds the necessary information to create an `Order`, `Order LineItem` and `Invoice`. Also it closely uses `Order`.
- **Information Expert:** `Search` method is added to `CustomerList` because it holds all the customers. Another `Search` method is added to `ItemList` because it holds all the items. `Item` class has an amount attribute(which represents the stock amount of that item), so `SearchStock` and `UpdateStock` methods are added to `Item` class. Also, `FindAvailable` method is added to `Item` class to find the available stock amount of the item. `GetPrice` method is also added to `Item` class because `Item` class has an price attribute. `GetTotal` method is added to `LineItem` so it can calculate the total price of a line item in an order.
- **High Cohesion:** We are calculating the total amount in `Order LineItem` class because it has the required information for calculating the total(price and amount).
- **Low Coupling:** `UpdateStock` and `Add` methods are'nt coming from `Order LineItem` class to avoid unnecessary coupling.

Sequence Diagram for the Use Case Process Order

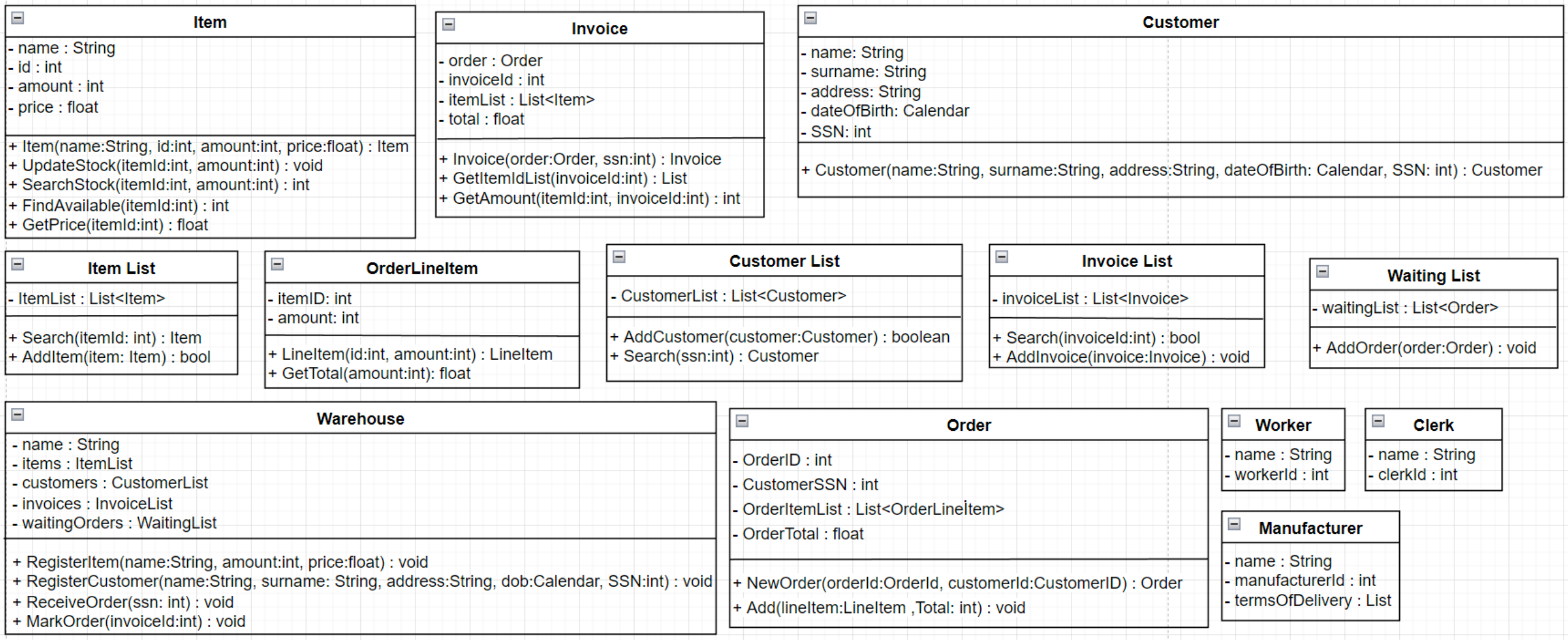


GRASP Analysis for Process Order Sequence Diagram

- **Controller:** `Warehouse` is the controller class in our sequence diagram because it seperates the interface and business layer. In this system, it represents the whole system and coordinates all of the system operations. Also, when a request comes from the UI layer object, `Warehouse` helps to determine what is that first object that receives the message from the UI layer objects.
- **Creator:** We have no creators in this diagram because we don't create any items.
- **Information Expert:** `Search` method is added to `InvoiceList` because it holds all the invoices. `GetItemIdList` method is added to `Invoice` class because it holds the invoices and invoices have `ItemList` in them. They also hold the items in an order so `GetAmount` method is also added to `Invoice` class.
- **High Cohesion:** `Item` class holds the item stock(amount attribute) so we added the `UpdateStock` method to `Item` class.
- **Low Coupling:** `UpdateStock` method isn’t coming from the `Invoice` class to avoid unnecessary coupling.

Deriving class model of the designed system

Class Model of Our System



Made with draw.io