Zeus Solutions

CS4125 Systems Analysis and Design Project

Warehouse Distribution Software

Shane Craven	12141194
Jonathan Hickey	10109889
Emmylou Flores	12132403

Table of Contents

Project Plan and Allocation of Roles 4

Narrative 5

Overview of current market 5

Zeus Solutions Product 5

Software Lifecycle Model 6

Waterfall Model: 6

V-Model: 6

Agile Software Development: 6

Requirements 7

Use Case Diagrams 7

Use Cases Description 10

Process Incoming Order 11

Use Case Realizations 16

GUI Prototypes 19

Log In 19

Log Out 19

Collect Items 20

Item Collected 20

Search for Order 21

Dispatch Order 21

Register New Product 22

Generate Item ID 22

Stock Items 23

Item Stocked 23

Generate Report 24

Discussion on Tactics to Support Quality Attributes 25

System Architecture 26

Architecture: 26

Implementation Language: 26

Design Patterns: 27

Server Fail Over 27

UML Workbench 27

Analysis 28

Data Driven Design: 28

User: 28

Product: 28
Item: 28
Order: 28
Priority: 28
Cubby: 29
Shelve: 29
Sector: 29
Class Diagrams 30
Interaction Diagram 32
Entity Relationship Diagram 33
Added Value 34
GitHub 34
Compcon – Customer feedback loop 37
Zeus Server Selector – Server Failover 38
Design Blueprints 40
Architectural Diagram 40
Class Diagram – coreclasses 41
Class Diagram – database 42
Class Diagram – server 43
Class Diagram – servercommunication 44
Class Diagram – client 45
Interaction Diagram - Login 46
State Chart 47
Descriptions of the patterns 48
Critique 49
Analysis Artefacts and Design – Compare and Contrast 49
Server and network communication observations 49
Database and Server side object observations 49
Implementation Criticisms 49
Server and network communications 49
Database 50
Core Class Package Communications 50
GUI Communicator Controller 50
Client 50
References 51

Project Plan and Allocation of Roles

Deliverable	Task	Assigned
Narrative		
	One Page Narrative	Jonathan
Software Lifecycle		
	One Page Software Lifecycle	Shane
Requirements		
	Use Case Diagrams	Jonathan
	Use Case Descriptions	Group
	NFRs	Emmylou
	Discussion on Tactics	Emmylou
	GUI prototypes	Jonathan
System Architecture		
	Two Page Discussion which includes Package Diagram	Shane
Analysis		
	Data Driven Design	Shane and Jonathan
	Class Diagram	Shane
	Communication Diagram Sequence Diagram	Jonathan
	Entity Relationship Diagram	Emmylou
Code		
	Code	Group
Discussion		
	Two Page Discussion of Added Value	Group
Design Blueprints		
	Architectural diagram	Shane
	Class Diagram	Shane
	Interaction Diagram	Jonathan
	One State Chart	Emmylou
	Descriptions of the design patterns used	Jonathan
Critique		
	Compare and Contrast Analysis vs Blueprints	Group

Role Title	Assigned
Project Manager	Jonathan
Requirement Engineer	Emmylou
System Architect	Shane
Developers	Group
Testers	Group

Narrative

Overview of current market

Over the last number of years internet shopping has become more popular, taking away business from local supermarkets and stores. Many of these supermarkets and stores have introduced online shopping to their customers, as well as their local outlets to be able to compete with the likes of Amazon.

This type of online shopping, has led to the development of various sizes of warehouses where these companies store their products. At these warehouses the staff process the customer orders. This involves the staff moving around the warehouse picking up products and bring them to packaging. Once at packaging the customer's postage information is attached to the product.

When new stock arrives at the warehouse, the staff have to place it onto the shelf specified. There are a number of rules that affect where the stock can be placed within the warehouse. Each business may have its own set of rules. For example: Frozen food will have to be placed in freezers. Very heavy items to be placed on lower shelves. Etc.

Zeus Solutions Product

This is where our solution comes in. Zeus Solutions wants to create an *automated* warehouse management system. This system will then be sold onto a number of Zeus Solutions clients. Each client may have a different operating system.

This *automated* system, will help the Client's staff with organisation in the warehouse. The Client's staff will still be needed within the warehouse. The main focus of this system is to increase the processing speed of the order, reduce human error and make the whole process easier to manage. These aims will be achieved by calculating the best routes that a staff member should take, which optimizes the staff productivity.

This system will also allow Client's managers to track employee work rate and stock levels within the warehouse. This is achieved by tracking the number of items handled by Client's staff throughout the order process.

With the system being developed in-house, each section of the documentation once approved by the required department cannot change. Unless the change has a sufficient impact on the overall system and can be agreed with all departments. The design of the system should take into consideration:

1. The system will need to have an easy use API for the Client so that they can integrate it into their current systems.

2. In the future Zeus Solutions clients can request new functionality to the product.

Software Lifecycle Model

We discussed three potential software lifecycle models to use for our project. Upon researching the Waterfall, V-Model and Agile development we found that both the Waterfall and V-Model had characteristics which were not optimal for our project.

Waterfall Model:

- The waterfall model reduces our ability to adapt to change in our business domain and introduces high risk and associated high cost in doing so. Working software is delivered at later stages of the lifecycle therefore incurring high cost in making any changes to the requirements set out at the beginning of this project should the need arise in the implementation step.

- Testing is performed as the last step before shipping the software using this model. Since our project encompasses automated test cases it would not be optimal to only execute such tests at the final stage of development, which could lead to the exposure of high level system bugs at this point in the process that could have been dealt with earlier at a reduced cost.

V-Model:

- Very rigid and inflexible model for software development that would make it more difficult to respond to any changes to the defined requirements.

- Since software is only developed during the implementation phase of the V-Model no early prototypes are created. As the complexity of modern systems grows we feel that having a model that allows for continuous iterative improvements is crucial.

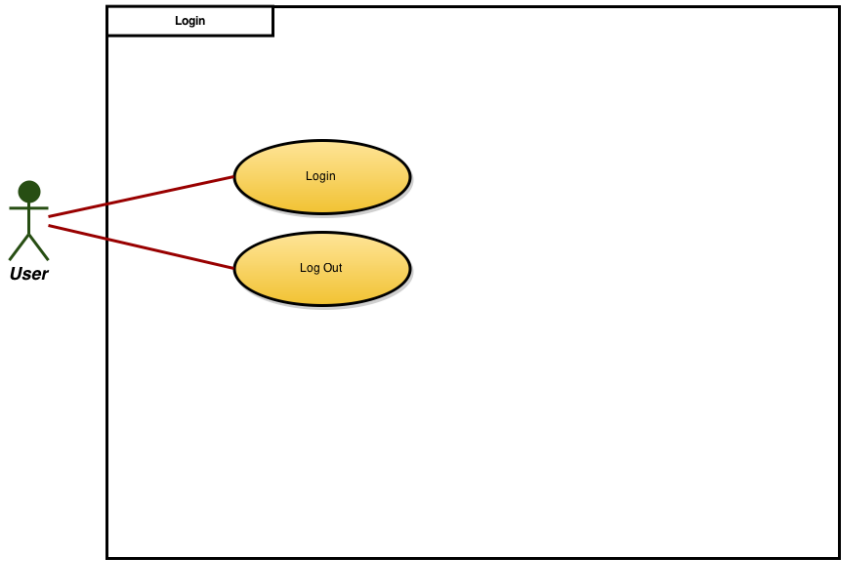
Agile Software Development:

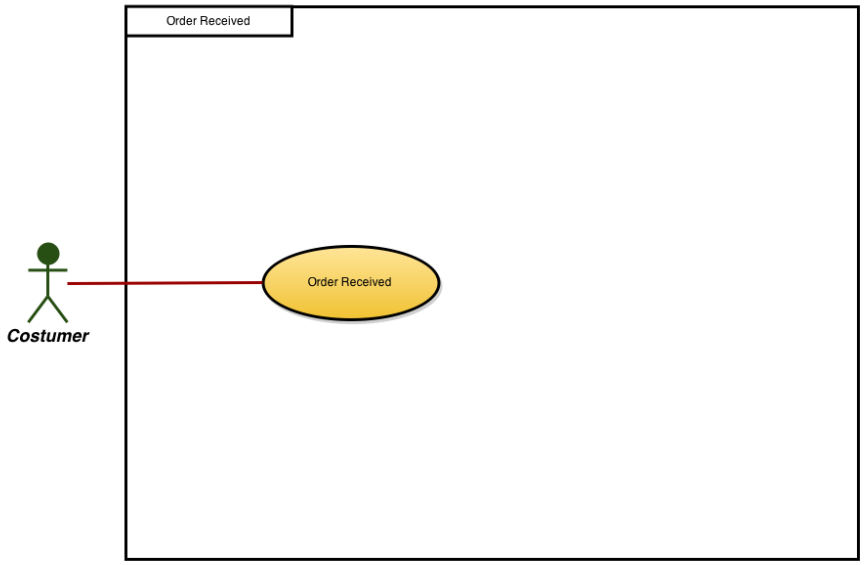
A paradigm shift is occurring in how customer demand for products is fulfilled. Building a system that can quickly respond to change as new technologies and processes come into scope is a key consideration for Zeus Solutions. As the number of people using online product procurement methods increases, the way we store a vast stock of material goods in storage warehouses and how the logistics of such an operation is managed will need to scale. We need a software development model that provides an iterative development approach to achieve our goal of constant software improvements as the business domain expands.

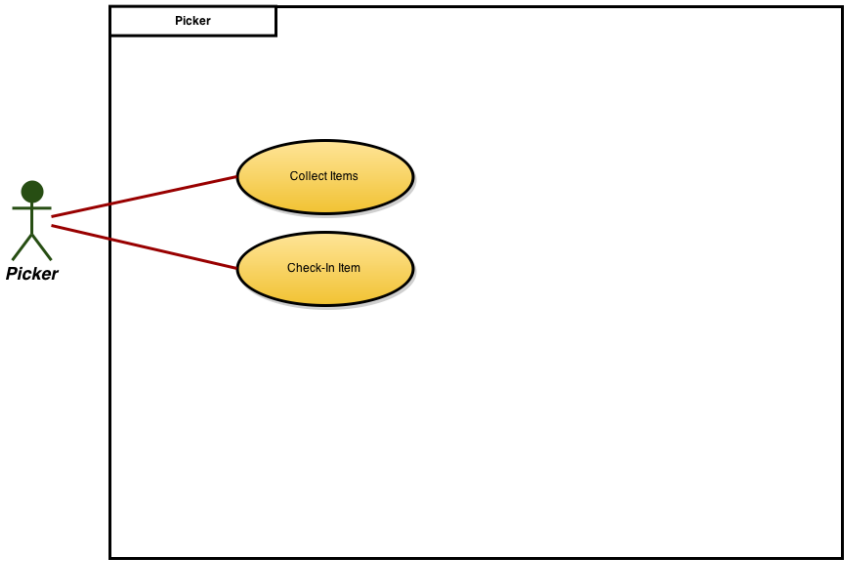
The agile software development model provides this key approach for Zeus Solutions. It provides us with the ability to take an iterative and incremental approach to software development. As the complexity of material storage and fulfilment increases we can incorporate feedback from our customers and with each iteration of our software provide them with the best possible experience and ensure that our system can scale as our customers expands their operation.

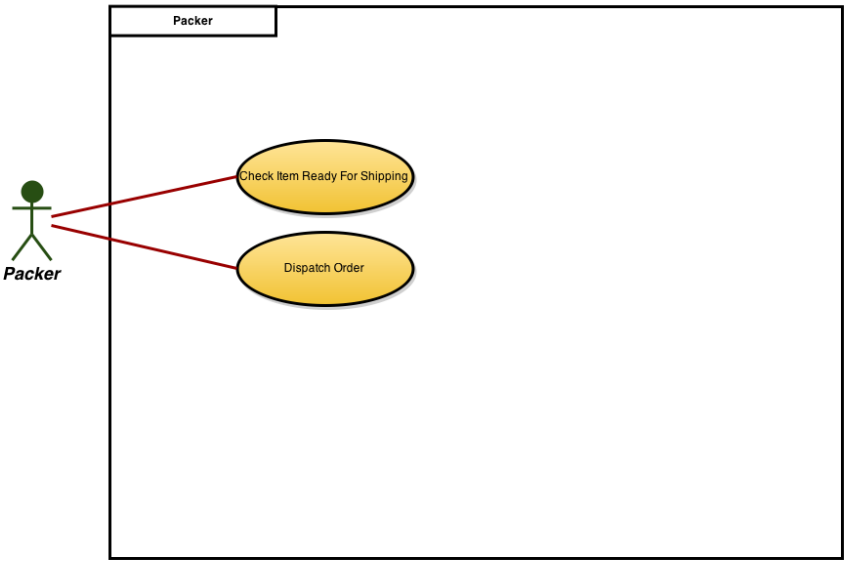
Requirements

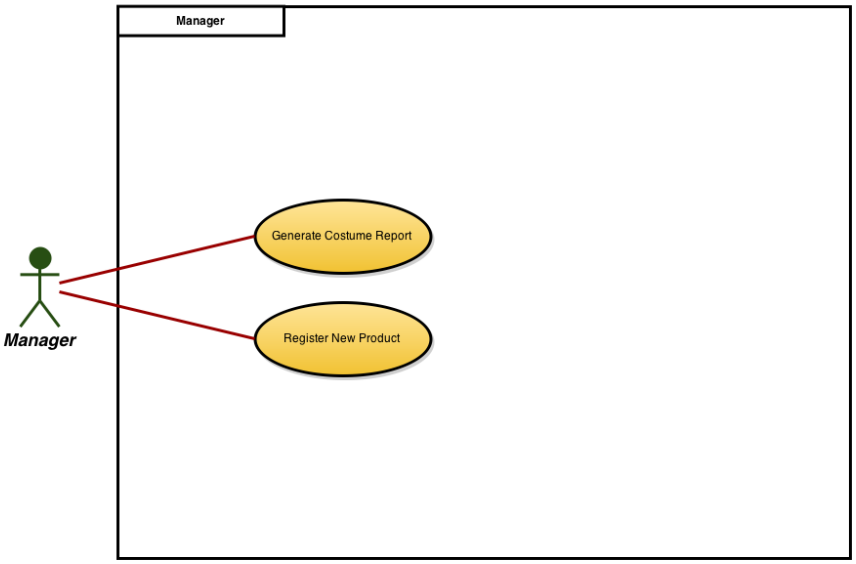
Use Case Diagrams

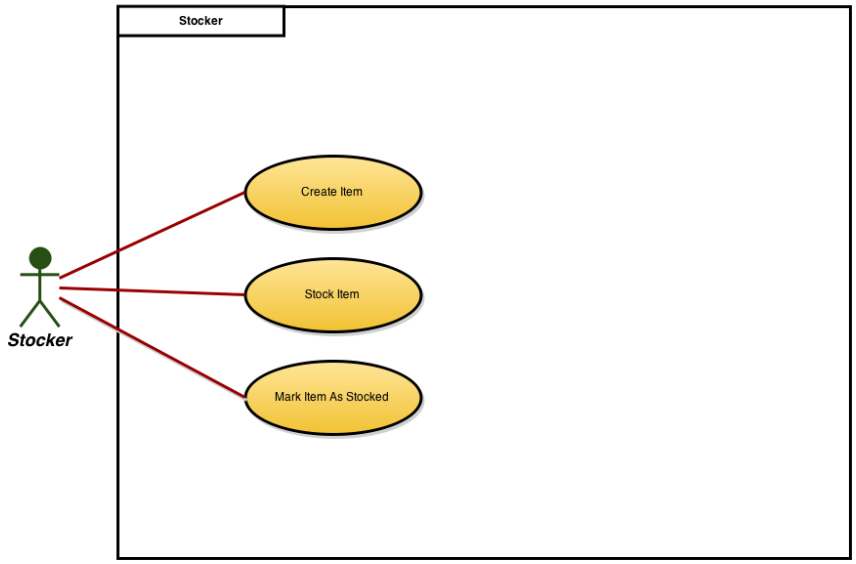





Picker
Collect Items
Check-In Item
Picker


Packer
Check Item Ready For Shipping
Dispatch Order
Packer


Manager
Generate Costume Report
Register New Product
Manager


Stocker
Create Item
Stock Item
Mark Item As Stocked
Stocker

Use Cases Description

Use Case: 001	Login
Actor Action	System Response
1. User enters credentials (username, password) and clicks login.	2. System validates user log in credentials. System displays user's main screen.
Alternative Route	
1. User enters credentials (username, password) and clicks login.	2. System cannot validate user log in credentials. Display Error message.

Non-Functional Requirement: Security

The system should encrypt the user's information and store it securely.

Use Case: 002	Log Out
Actor Action	System Response
1. User clicks log out.	2. System displays log out screen.
Alternative Route	

Non-Functional Requirement: Security

The system should end the session duration and log out the user securely.

Use Case: 004	Collect Items
Actor Action	System Response
1. Picker clicks Collect Items.	2. System displays a list of items for the picker to collet. System asks picker if they are ready to collect the items.
3. Picker clicks ready.	4. System binds the items to the picker. System removes the items from the sector queue.
Alternative Route	
3. Picker clicks the back to main screen.	4. System returns the pick to main screen. Items are not bound to the picker.

Non-Functional Requirement: Usability

The user main screen should be easy to interact with and the functions are easy to understand.

Use Case: 005	Item Collected
Actor Action	System Response
1. Picker inputs an Item ID and clicks Item Collected.	2. System validates the Item ID. System changes Item status.
Alternative Route	
1. Picker inputs an Item ID and clicks Item Collected.	2. System cannot validate the Item ID. System alerts Picker of Error.

Non-Functional Requirement: Accuracy

System should display accurate information regarding the item ID that has been searched.

Use Case: 003	Process Incoming Order	
Description	An Incoming order is received by the system. This order is then processed and the items which make up the order are sourced in the warehouse. The system then fills the relevant queues with the items that need to be picked for order fulfilment	
Preconditions	1: Order has shipping information 2: Order contains one or more valid products	
Post Conditions:	1: Order has been processed successfully.	
Success End Condition	Order has been processed successfully.	
Failed End Condition	The order has not been processed successfully	
Actors	Customer	
Trigger	Order has been received through the API	
DESCRIPTION	Step Number	Action
	1	Order has been received through the API
	2	Order is placed into an order queue
	3	The system pops off an order from the order queue
	4	The system locates all available items for a product within the warehouse
	5	The system selects an item from the list of items, based on a calculated priority
	6	System updates order with selected item information
	7	After selecting the item, the item is passed into the sector queue.
	8	Item State is changed from “Available” to “Pickup”
	9	Steps 2 to 6 are repeated for the each of the products within the order
	10	The order status will be changed from “New Order” to “Processing Order”
EXTENSIONS	Step Number	Branching Action
	4.a	No items are available to fulfil a product request: Return error message and stop order processing
VARIATIONS	Step Number	Branching Action

Non Functional Requirement: Extensibility

The algorithm for selecting which items should be used to fulfil a product in the order should be modifiable by the customer.

Use Case: 006	Search for Order
Actor Action	System Response
1. Packer inputs Item ID and clicks Search for Order.	2. System returns the Order ID that the item belongs to. System makes Order complete if all items have been processed. System asks the Packer if they are ready for label to be printed.
3. Packer clicks Print Address	4. System prints address for shipping. Screen is reverts back to normal.
Alternative Route	
1. Packer inputs Item ID and clicks Search for Order.	2. System returns an error if the Item ID does not belong to an order.
1. Packer inputs Item ID and clicks Search for Order.	2. System returns the Order ID that the item belongs to.

Non-Functional Requirement: Reliability

System should return appropriate errors if necessary.

Use Case: 007	Dispatch Order
Actor Action	System Response
1. Packer clicks Dispatch tab.	2. System changes the Search for Order GUI display to Dispatch GUI display.
3. Packer inputs Order ID and clicks Dispatch Order.	4. Systems changes Order status to order shipped. System displays a message informing the user that the order has been dispatched.
Alternative Route	
3. Packer inputs Order ID and clicks Dispatch Order.	4. System returns an error if the Order ID is not complete.

Non-Functional Requirement: Performance

The change between GUI displays should be fast and responsive.

Use Case: 008	Register New Product
Actor Action	System Response
1. Manager clicks Register New Product tab.	2. System changes the Generate Report GUI display to Register New Product GUI display.
3. Manager enters Product ID, Name, Height, Width, Depth, Weight and Description of product. Picker clicks Register New Product.	4. Systems validates manager's input. System displays a message informing the user that the product has been added to the system.
Alternative Route	
3. Manager enters Product ID, Name, Height, Width, Depth, Weight and Description of product. Picker clicks Register New Product.	4. System returns an error if the information is missing or if the product is already existing.

Non-Functional Requirement: Performance

The change between GUI displays should be fast and responsive.

Use Case: 009	Generate Item ID
Actor Action	System Response
1. Stocker clicks Generate Item ID tab.	2. System changes the Stock Items GUI display to Generate Item ID GUI display.
3. Stocker enters Product ID, manufacture date and/or Expiry Date and clicks Create Item ID.	4. Systems validates stocker's input. System applies a priority to the Item. Item gets placed into a queue, where it waits until it is put into a cubby. System prints a label with unique ID for the Item.
Alternative Route	
3. Stocker enters Product ID, manufacture date and/or Expiry Date and clicks Create Item ID.	4. System returns an error if the information is missing or incorrect.

Non-Functional Requirement: Performance

The change between GUI displays should be fast and responsive.

Use Case: 010	Stock Items
Actor Action	System Response
1. Stocker clicks Stock Items ID tab.	2. System changes the Generate Item ID GUI display to Stock Items GUI display. System will display a list of items that are needed to put away. Along with the item information the system will display which cubby hole in the shelve the item is meant to be in.
3. Stocker clicks ready.	4. Systems removes items queue.
Alternative Route	

Non-Functional Requirement: Performance

The change between GUI displays should be fast and responsive.

Use Case: 011	Stock Items
Actor Action	System Response
1. Stocker clicks Stock Items ID tab.	2. System changes the Generate Item ID GUI display to Stock Items GUI display. System will display a list of items that are needed to put away. Along with the item information the system will display which cubby hole in the shelve the item is meant to be in.
3. Stocker clicks ready.	4. Systems removes items queue.
Alternative Route	

Non-Functional Requirement: Performance

The retrieval of requested information should be fast and responsive.

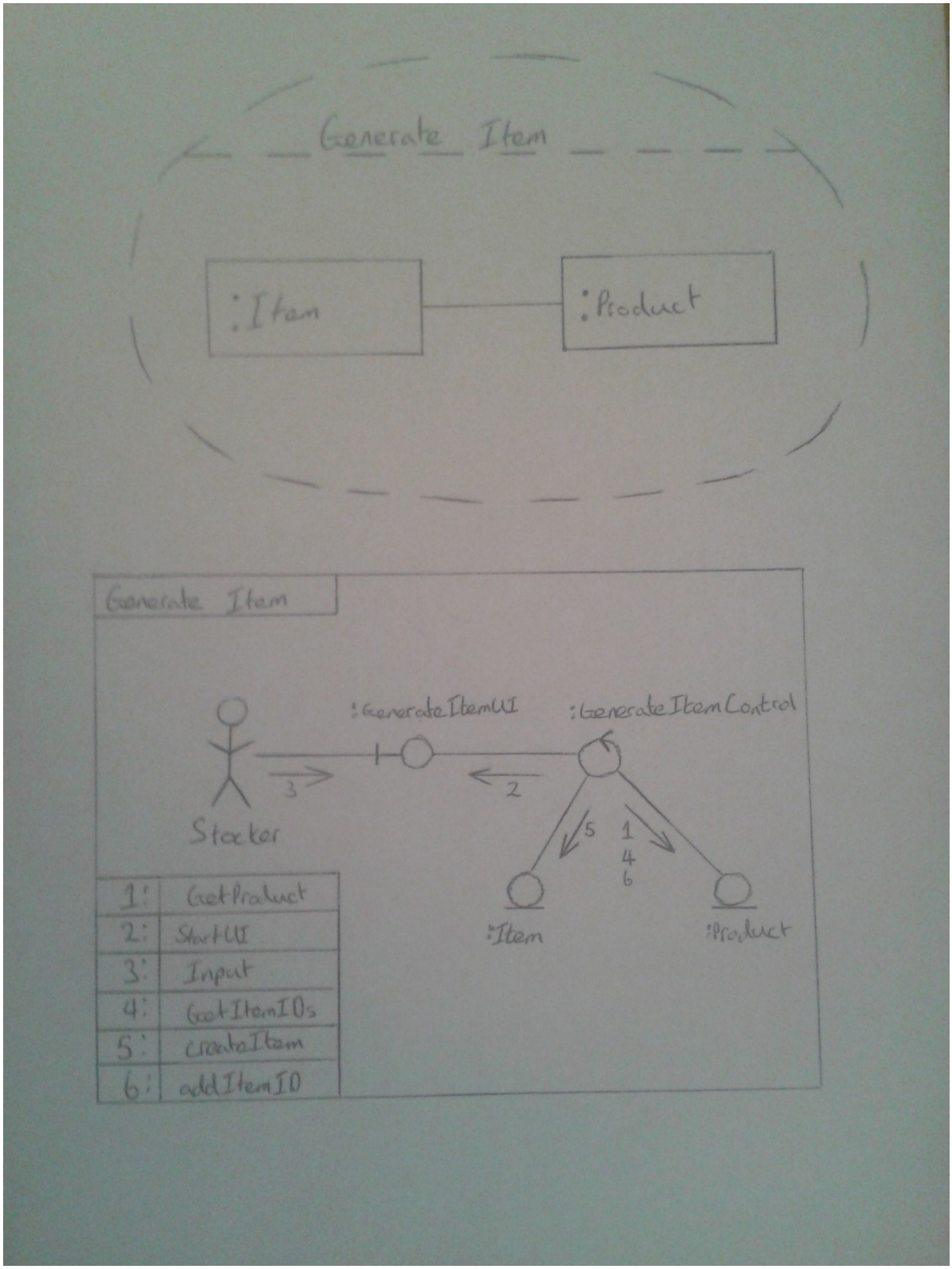
Use Case: 011	Item Stocked
Actor Action	System Response
1. Stocker clicks Stock Items tab.	2. System changes the Generate Item ID GUI display to Stock Items GUI display.
3. Stocker enters Item ID, Cubby ID and clicks Item Stocked.	4. Systems validates Stocker's input. System updates Item's location.
Alternative Route	
3. Stocker enters Item ID, Cubby ID and clicks Item Stocked.	4. System returns an error if there is information missing, or if there is invalid information.

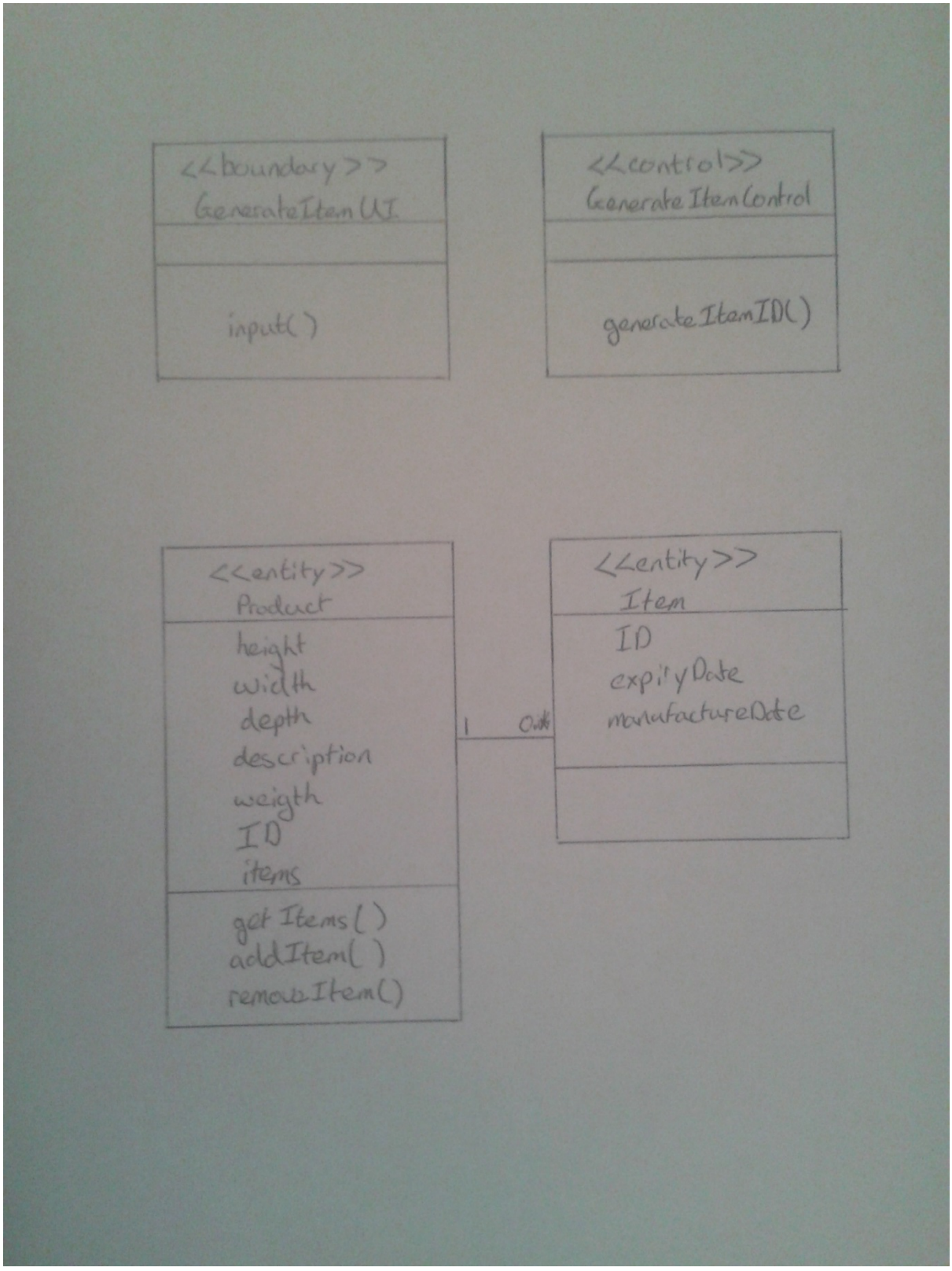
Non-Functional Requirement: Performance
The change made in information should be fast and responsive.

Use Case: 012	Generate Report
Actor Action	System Response
1. Manager clicks Generate Report tab.	2. System changes the Register New Product GUI display to Dispatch GUI display.
3. Manager enters start date, end date, one or more Product IDs and selects one or more types of graphs.	4. Systems validates manager's input. System displays a graph based on the information Manager has provided.
Alternative Route	
3. Manager enters start date, end date, one or more Product IDs and selects one or more types of graphs.	4. System returns an error if there is information missing, or if the system couldn't generate the report.

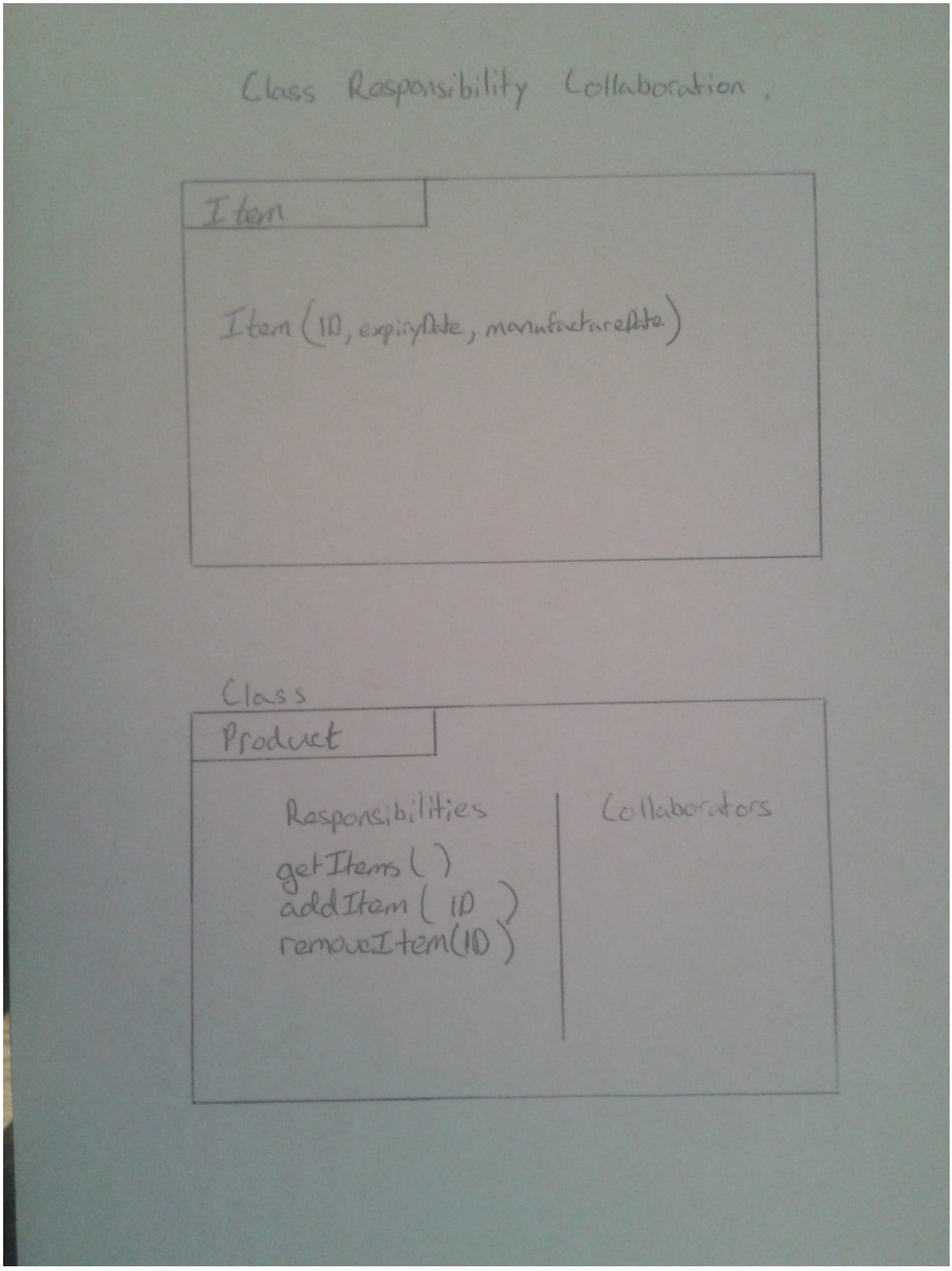
Non-Functional Requirement: Accuracy
The system should generate appropriate information as requested by the user.

Use Case Realizations



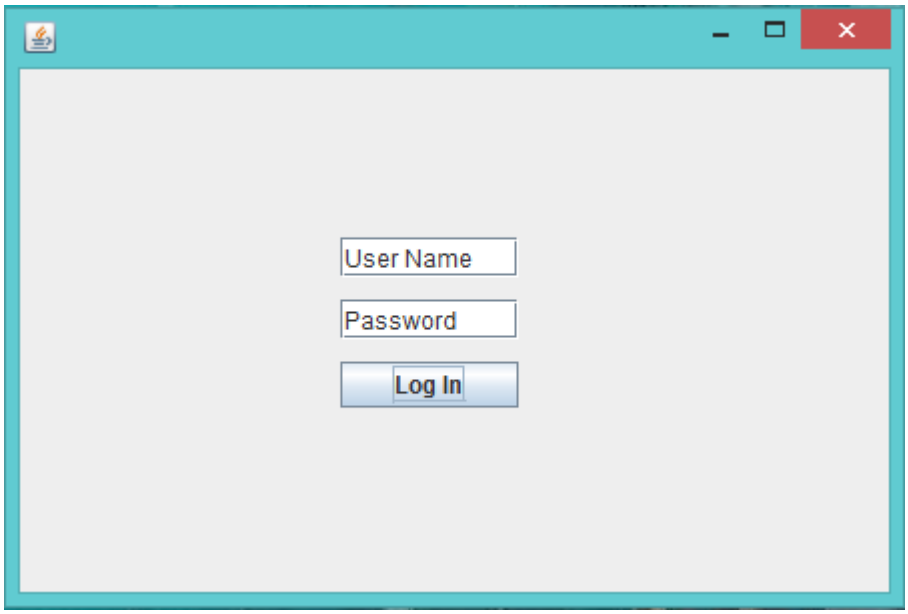

<<boundary>>
GenerateItemUI
input()
<<control>>
GenerateItemControl
generateItemID()
<<entity>>
Product
height
width
depth
description
weigth
ID
items
getItems()
addItem()
removeItem()
1
0..*
<<entity>>
Item
ID
expiryDate
manufactureDate

Class Responsibility Collaboration.

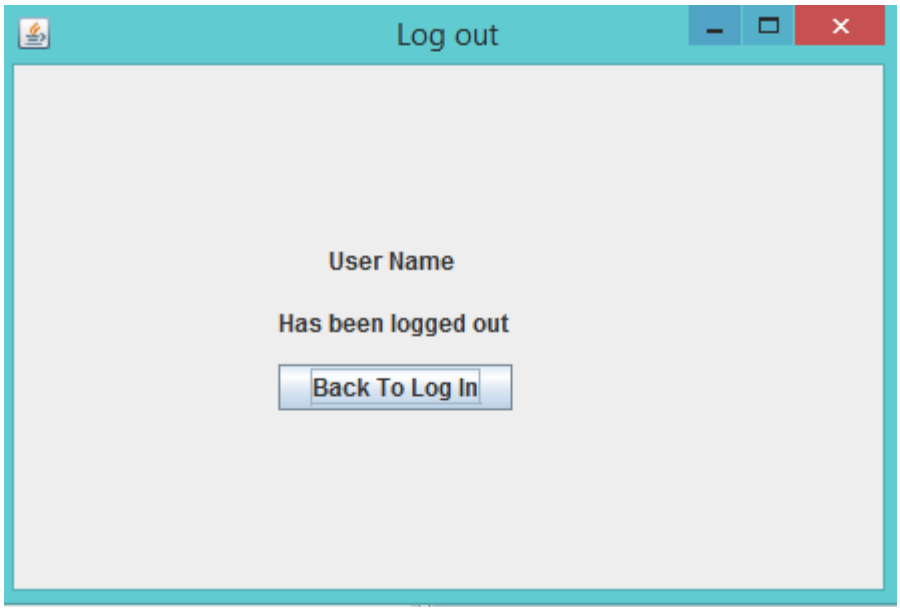


GUI Prototypes

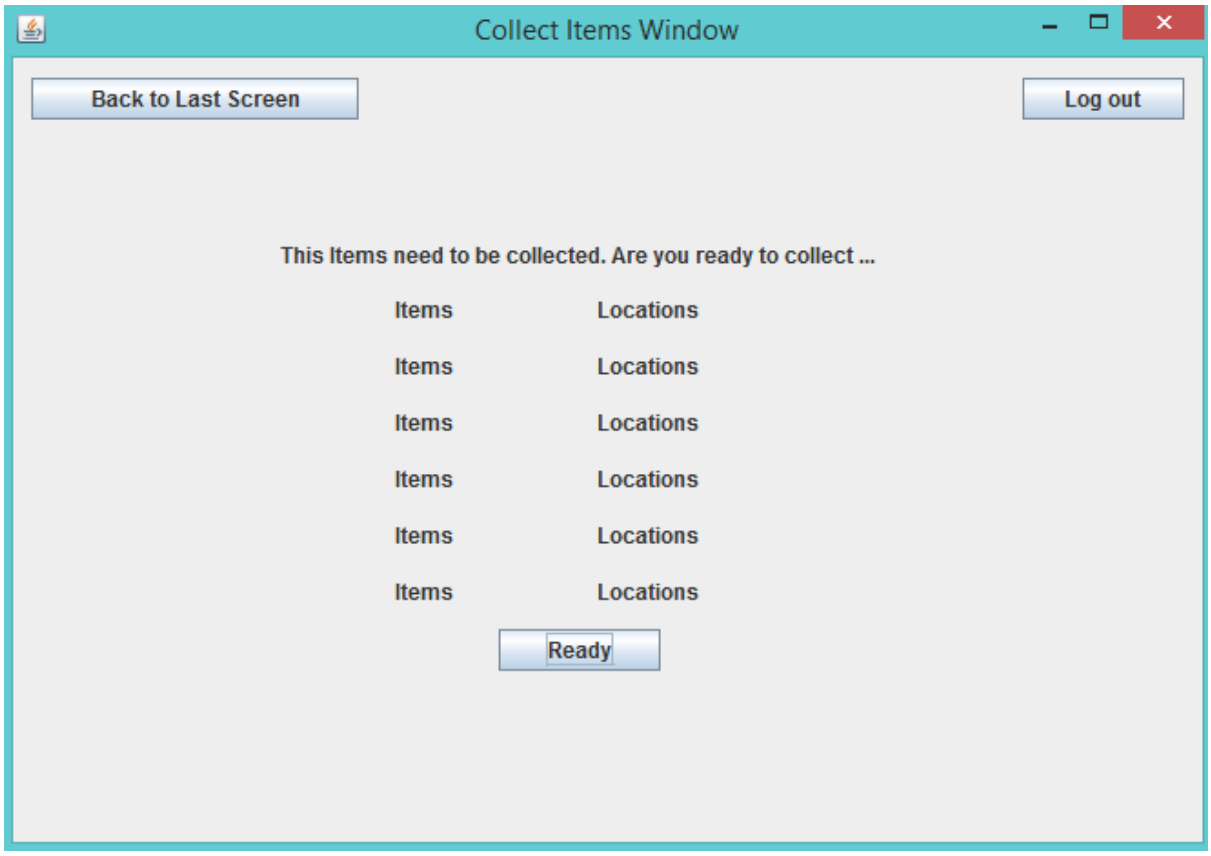
Log In



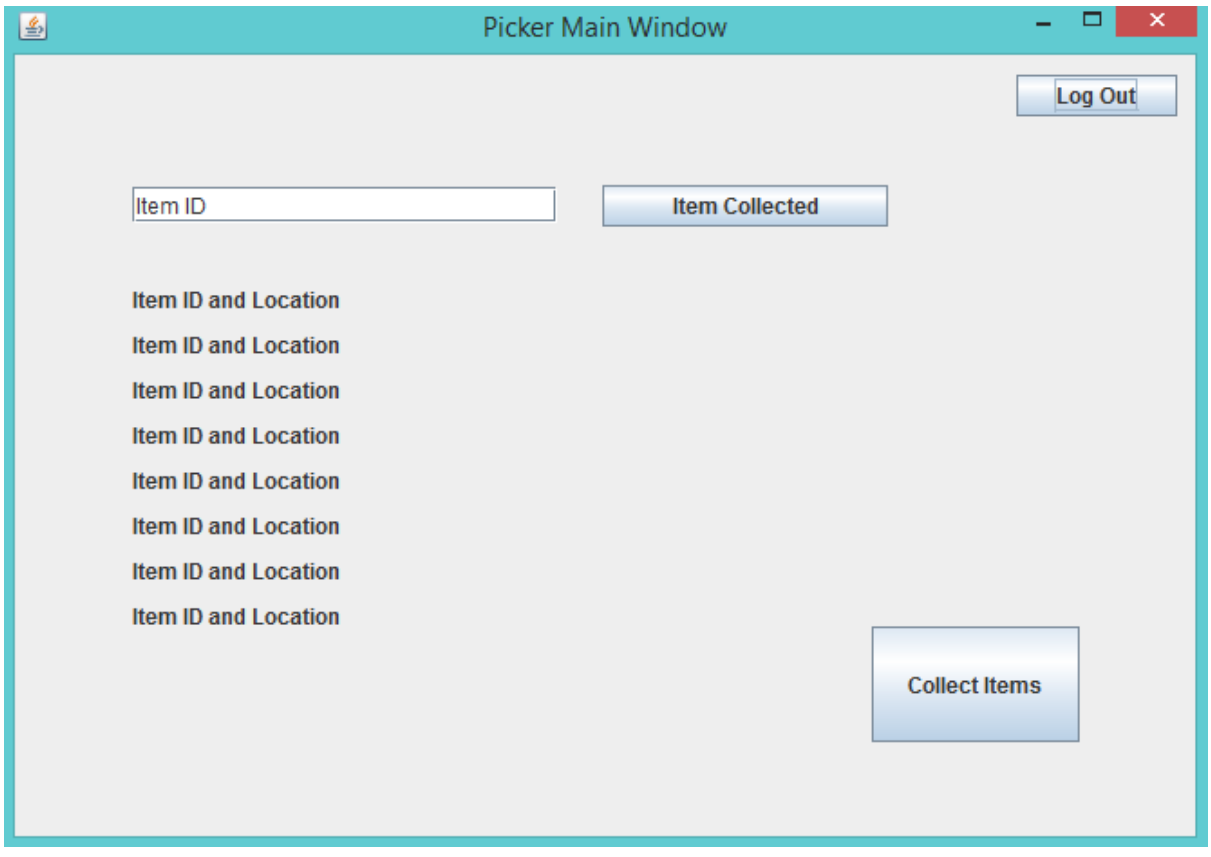
Log Out



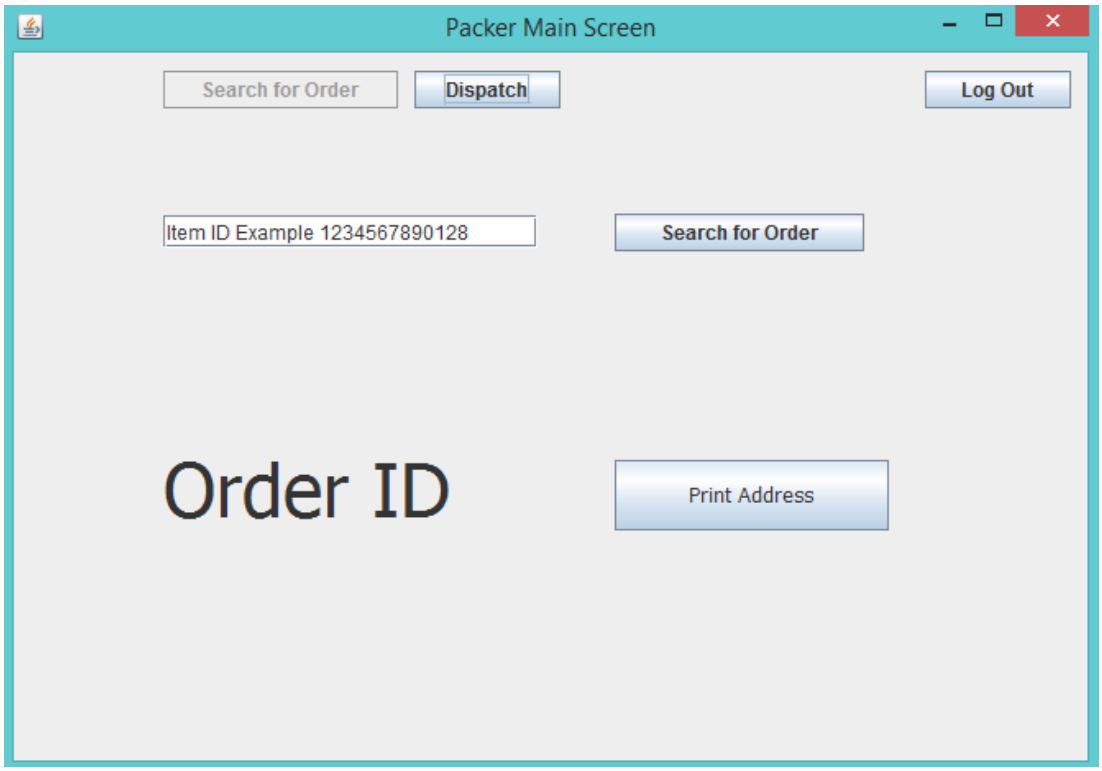
Collect Items



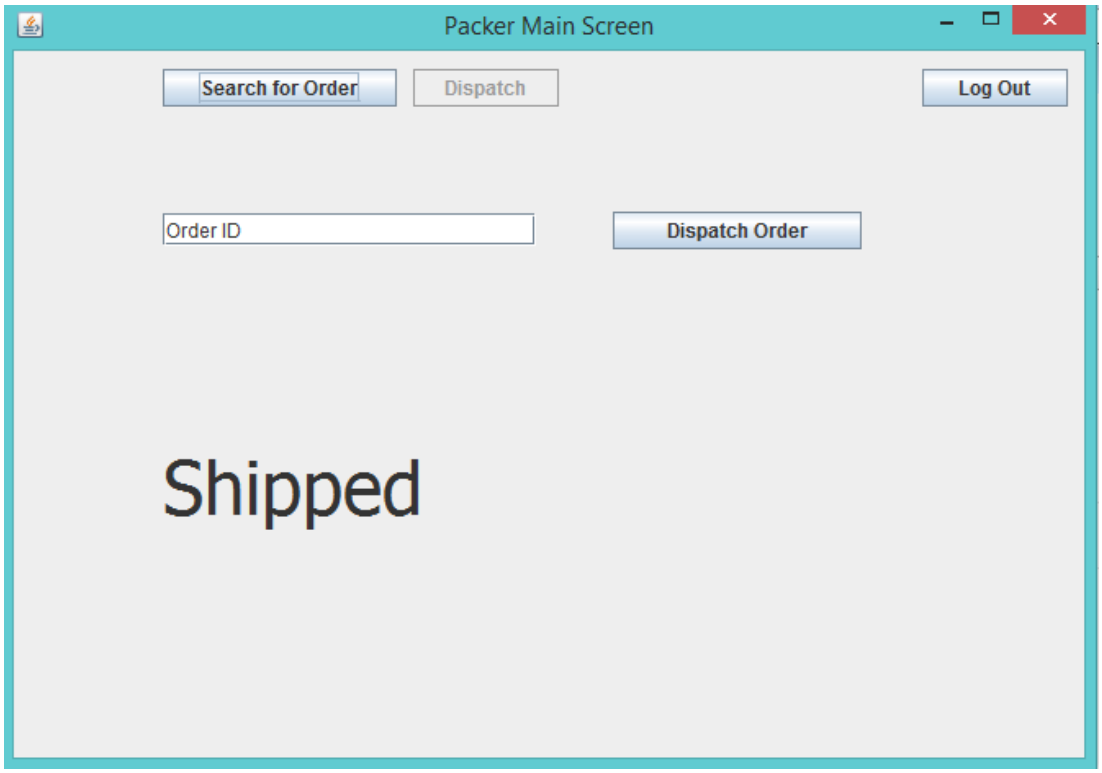
Item Collected



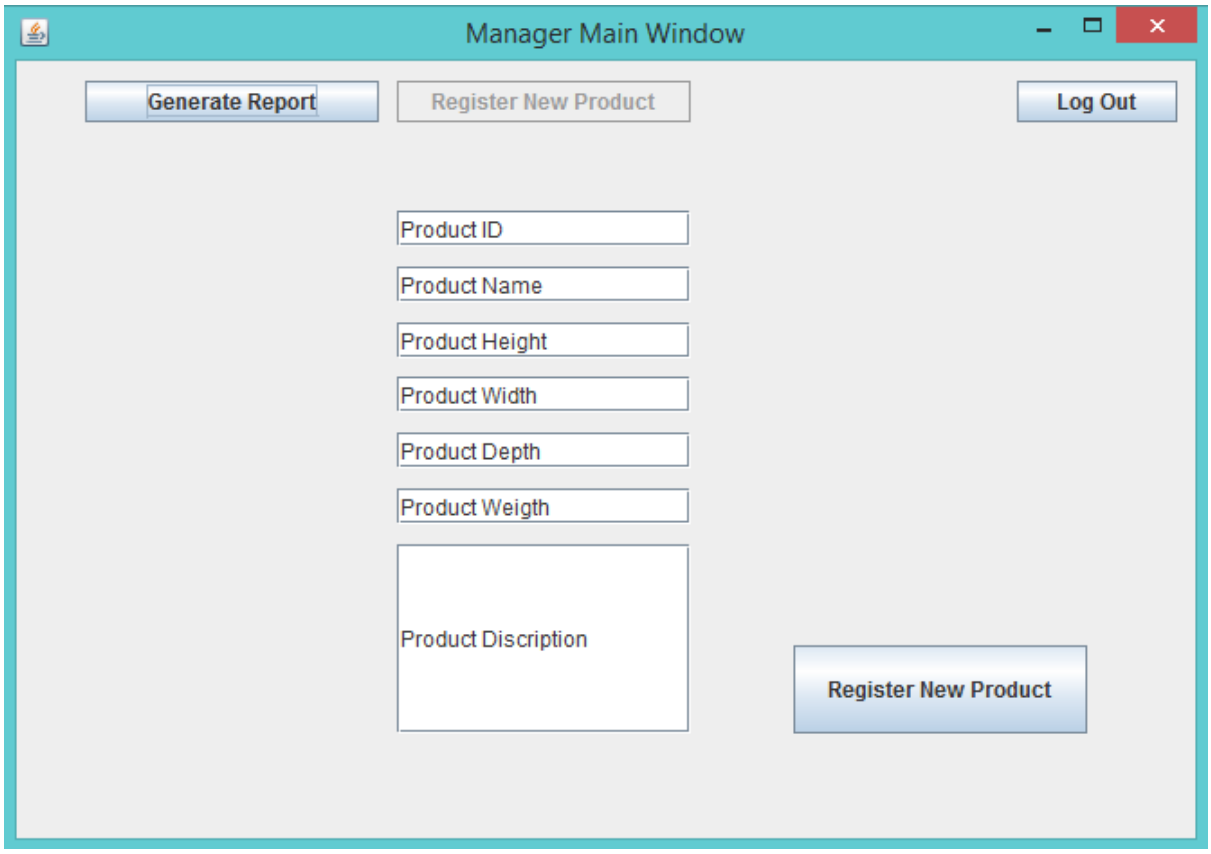
Search for Order



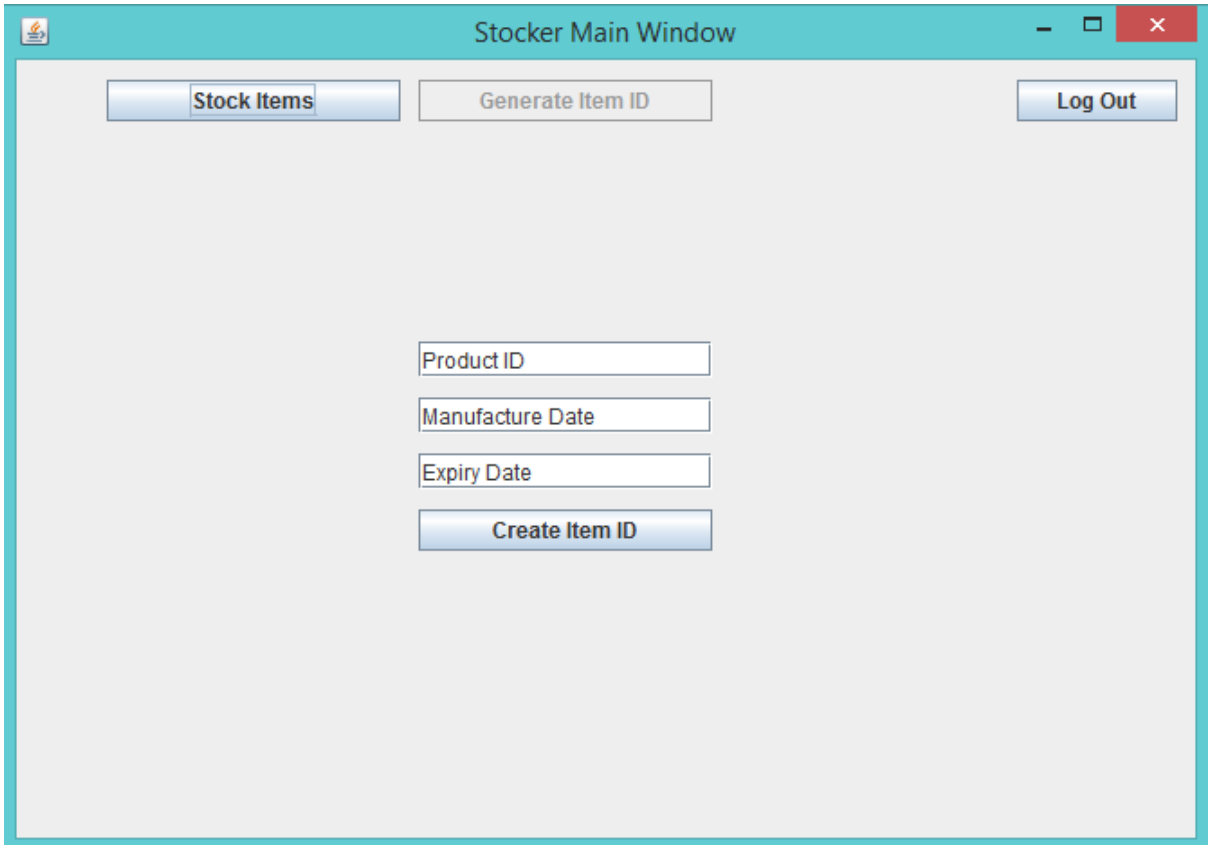
Dispatch Order



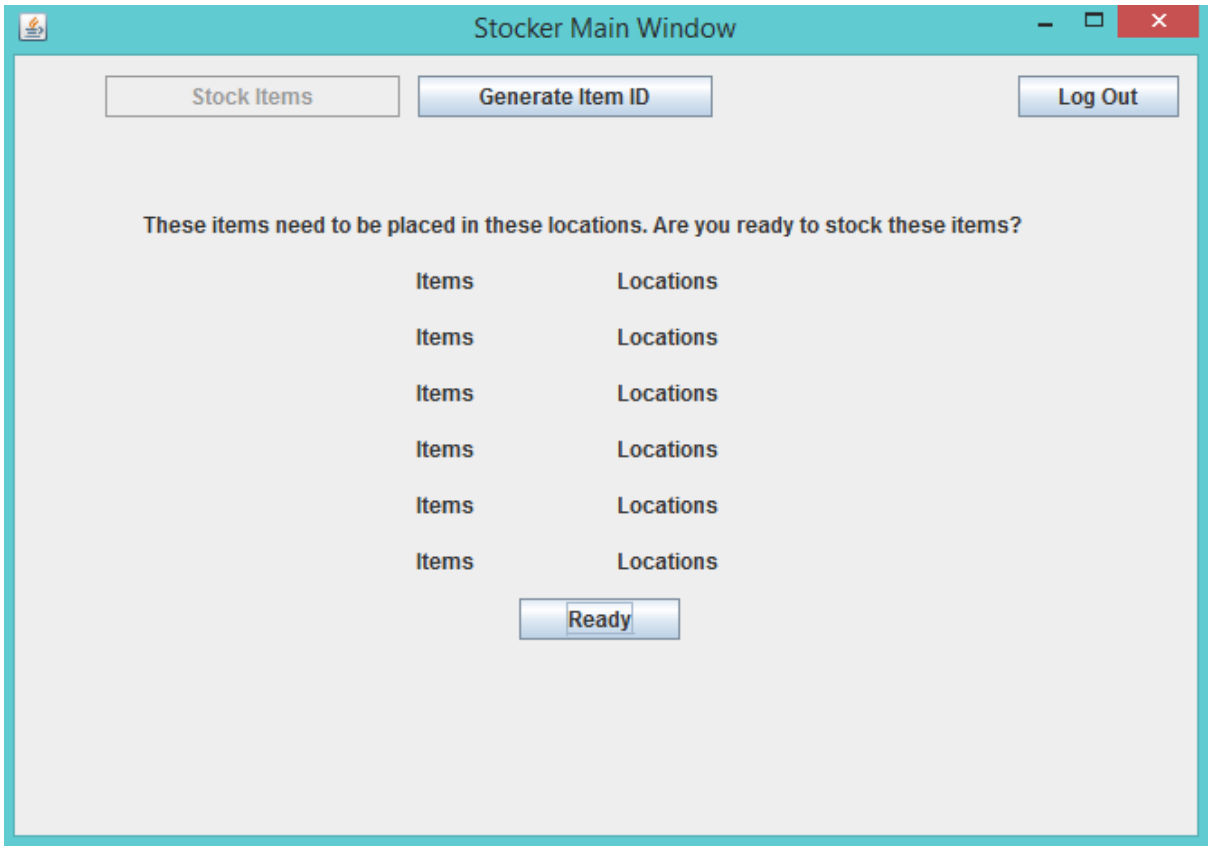
Register New Product



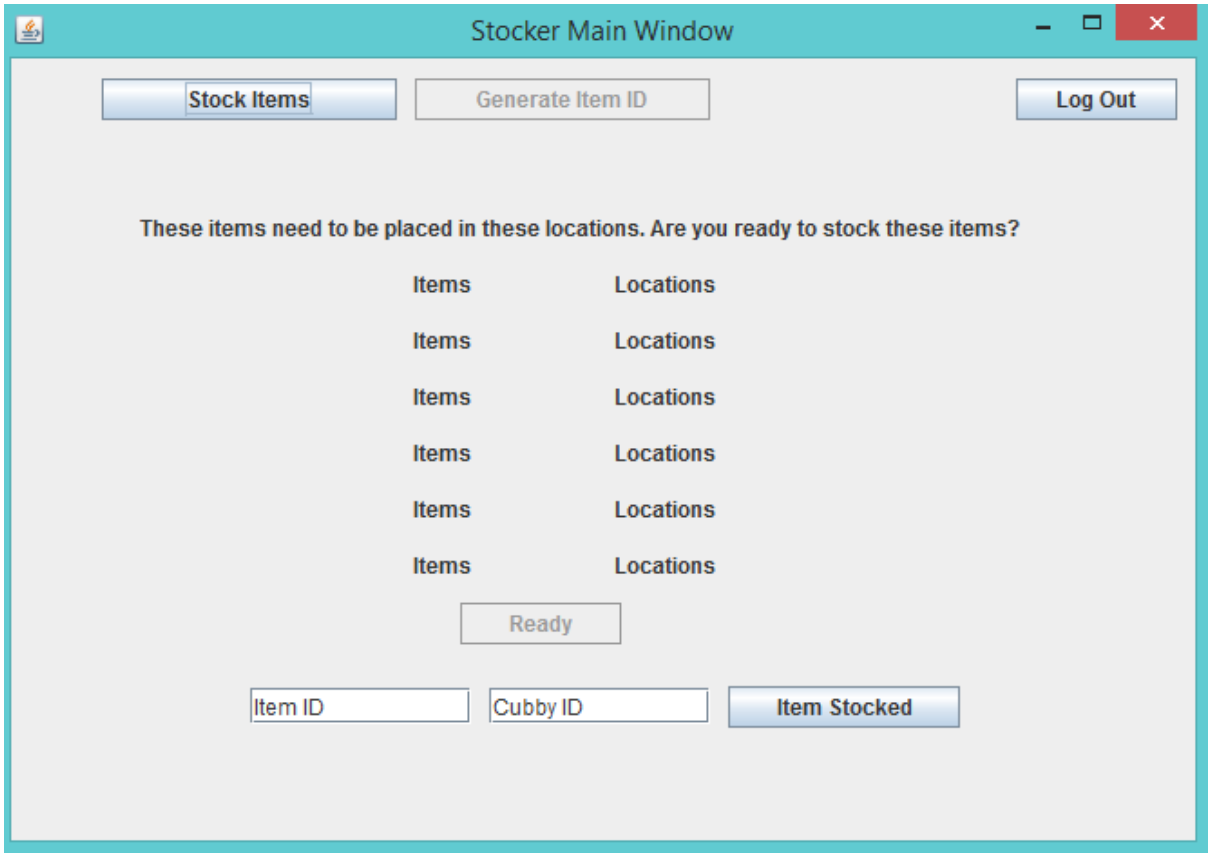
Generate Item ID



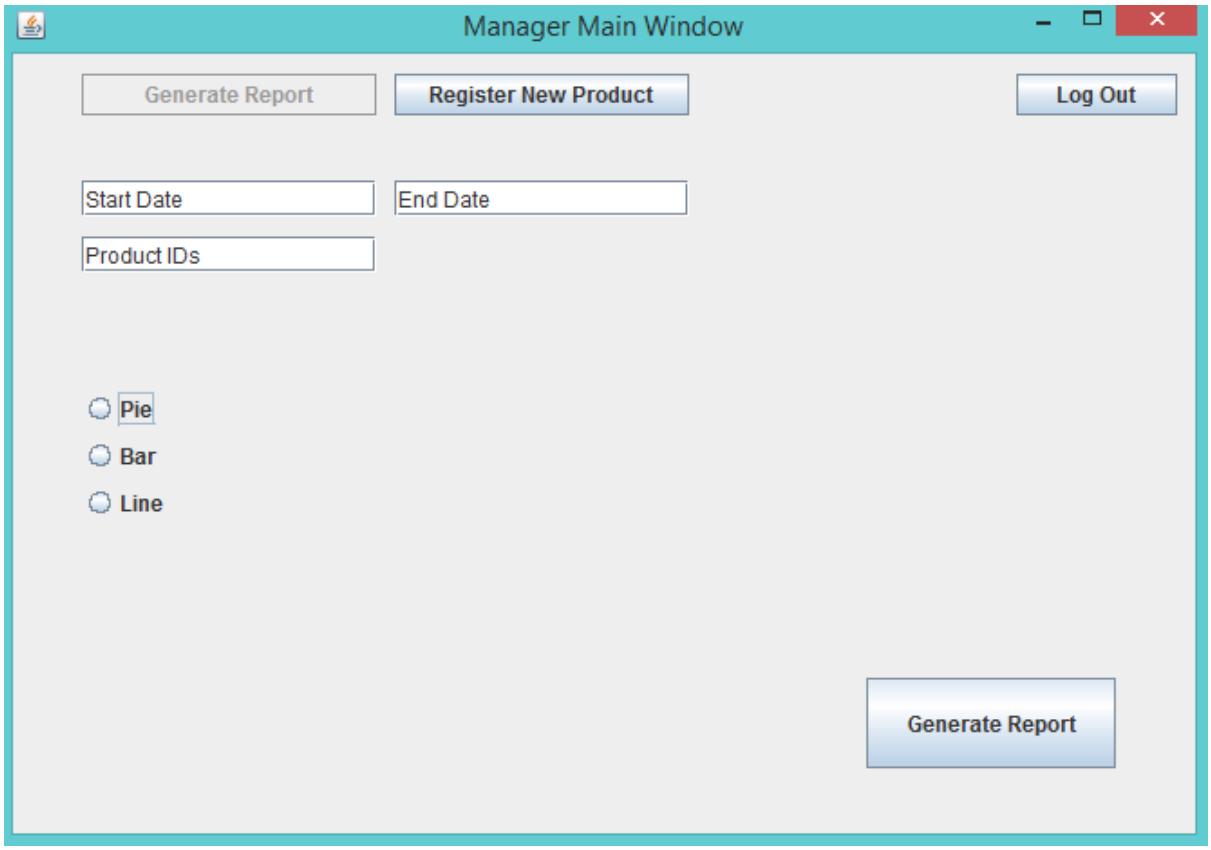
Stock Items

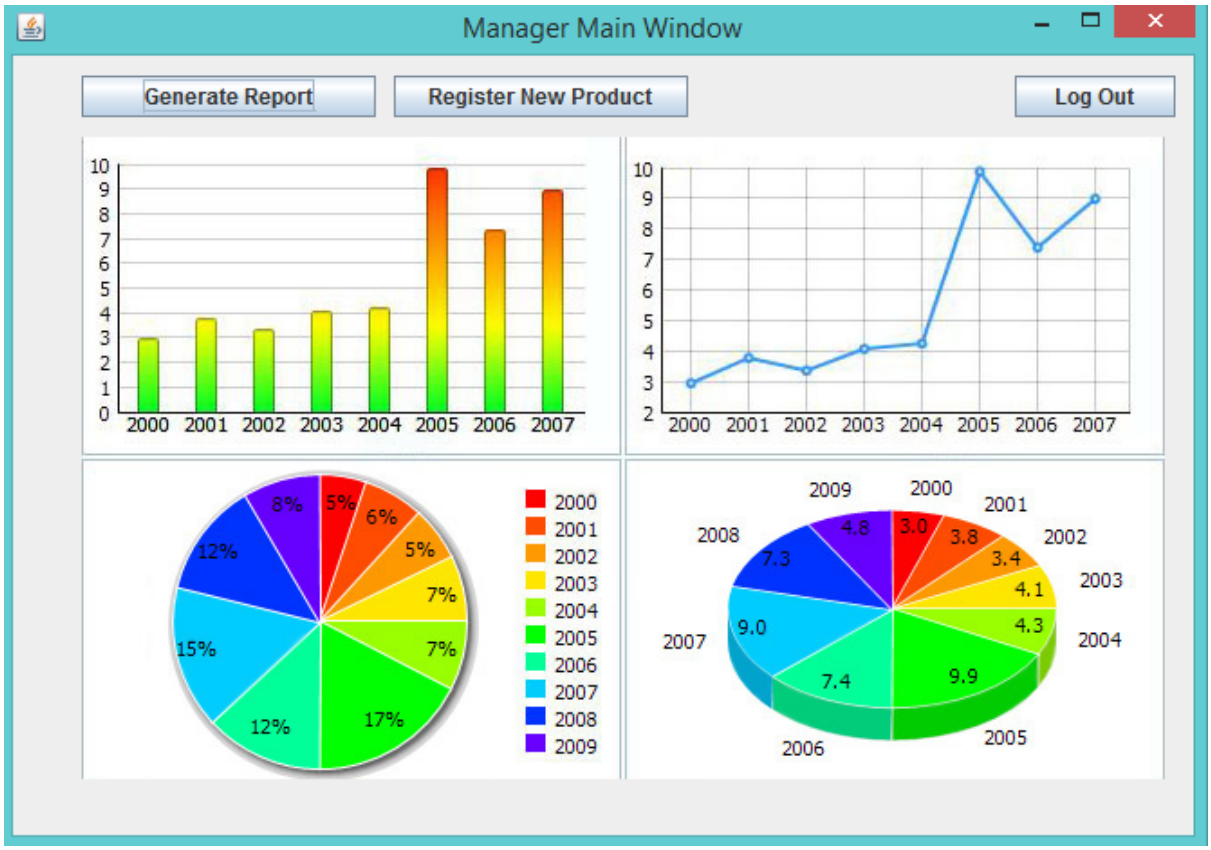


Item Stocked



Generate Report





Discussion on Tactics to Support Quality Attributes

As a team we decided to program our system with the use of the Java programming language as it allows great system portability. Based on research, the Java programming language is the most common language known to all software engineers. This will enable ourselves to find many other programmers to maintain and update our system.

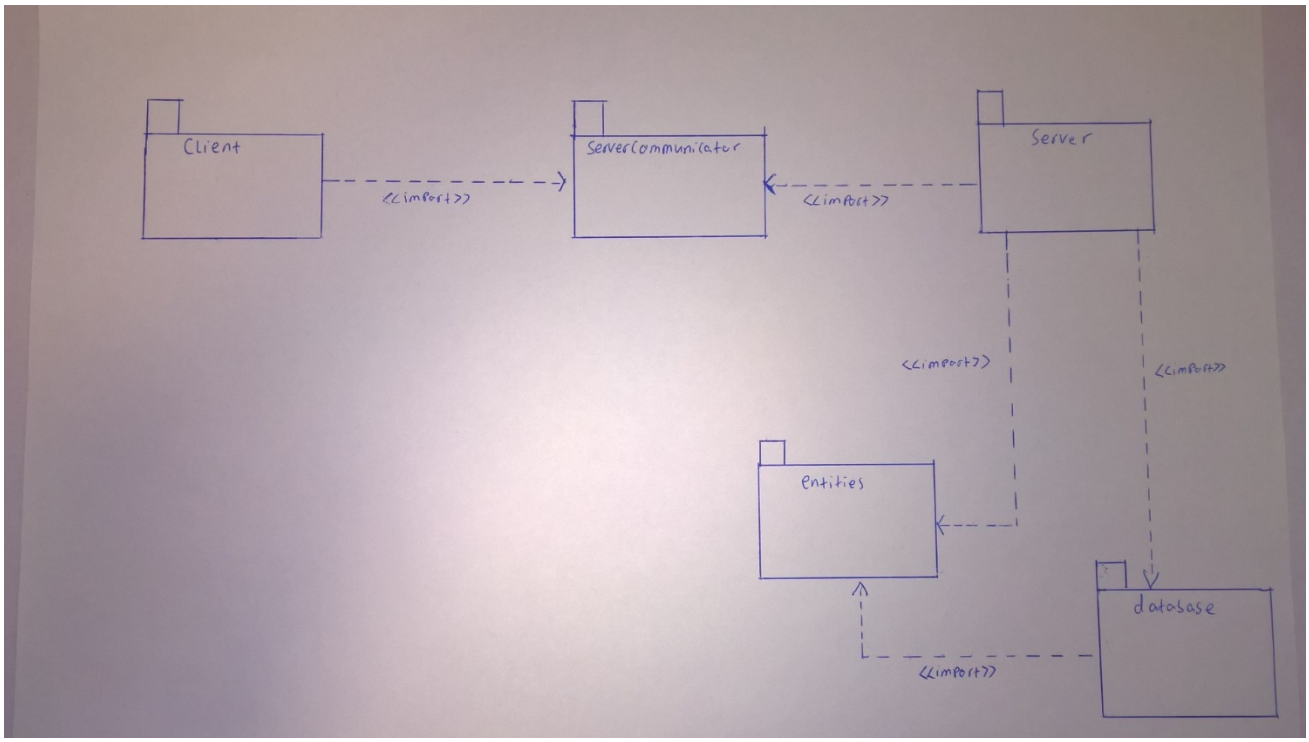
Our main aim is to deliver the highest of standards to the supply chain development of enterprises globally. To achieve this, we are taking the agile development approach and constantly improve our system. This would help maintain our system's reliability and modifiability at once.

Additionally, we want our users to be confident when using our system and have them reassured that the security of any private information is stored efficiently and kept confidential. To support this ability we are limiting access to the server functions to users with valid credentials and have the required access level.

In order for our system to be successful and unique, we have developed our own algorithms to ensure accuracy to all information and service we deliver to our users.

We wanted our users to be able to navigate through our windows and use its functions intuitively so we decided to design a simple, friendly to use graphical user interface that will enhance the users' recognition rather than recall of the system and learnability, which will lessen the training and time it will take for a person to use it.

System Architecture



Package Diagram – Hand Drawn

Architecture:

The architecture for this project is divided into three separate sub-systems. The client side, server side and database. Each sub-system runs independently from the other and strict rules are in place to determine which sub-systems can access the other.

The client sub-system communicates directly with the server which contains all business logic for the application. The client will send messages requesting an action to be performed, the required data needed to perform said action and will retrieve a result. The client side will have no direct access to the database layer and must do all processing through the server API. The idea behind this approach at this point in the project is reduce maintenance costs since all clients must use the same API.

The server sub-system contains all business logic in the application and is the layer responsible for performing actions. It communicates directly with the database and returns results of its processes to the caller. It is also responsible for ensuring that requests are authenticated before providing the requested service.

The database sub-system would, in a full scale implementation run on its own server, possibly exposing functionality through a RESTful API for expansion of database consumption in the future. For the purposes of this project the database layer will be simulated due to time constraints.

Implementation Language:

Due to its portable nature we chose java as our programming language of choice for both client and server sub-system implementation. Java is built on the premise of “Write Once Run Anywhere” essentially allowing our software to run on a wide spectrum of differing hardware configurations and

operating systems as long as the Java Virtual Machine is supported. Since all team members had previous experience working with this language it also felt like the optimal choice to reach maximum productivity during the implementation stage.

Due to the nature of our system and its application in enterprise the scale at which it's deployed and the volume of requests it receives from clients may become problematic. This may introduce performance issues as the customer needs expand. Due to java's extra overheads concerned with the virtual machine and automatic garbage collection this may become a problem, in particular on the server side. One way to alleviate this problem would be to implement the server side in C++ however we then encounter a trade-off between portability and performance. This performance issue could be mitigated to a point if the system is sufficiently distributed. By distributing our system efficiently into components we can spread load across multiple instances and thus improve performance through those means. Although a C++ implementation would provide increased performance server side we don't feel the benefits are sufficient when measured against the costs.

Design Patterns:

At this point in the project we have considered numerous design patterns to improve the extensibility and maintainability of our code and to better support key design principles in the three sub-systems of the application. In the class diagrams that follow this section you will see an example of the Factory design pattern which we plan to use to construct Cubby objects of differing sizes. Using the factory design pattern will allow us to encapsulate and localise object creation which makes code easier to maintain in the future.

It's apparent at this point that due to the client abstraction from the underlying backend implementation on both the server layer and database layer the server will need a flexible method in which to add new features to its API which is available to clients. To fulfil this need, I have been researching the Command design pattern which I believe can be combined with the strategy pattern to provide this flexibility and allow easier extensibility of the API in the future. Both the Command and Strategy patterns are members of the behavioural design pattern family. In addition we plan to use the Observer Design pattern in the GUI implementation using the Swing library.

Server Fail Over

In addition to the three tier architecture discussed in this section I believe this project would be an excellent candidate for an external service that informs the client application what instance of a Zeus Solution Server to connect to. This would provide a number of key benefits which I feel contribute significant value to the overall design. The ability for a server to fail, yet this failure does not affect the client end is clearly a feature worth implementing. If this feature does get implemented for our proof of concept implementation it will be writing in c# using the .NET framework due to past experience working in that environment when constructing web applications.

UML Workbench

As part of our research at the beginning of this project we have decided to use, for blueprints, Visual paradigm. This workbench has a healthy feature set and provides easy to use tools for creating professional UML diagrams. For this purposes of this implementation we are using the Community Edition of Visual paradigm,

Analysis

Data Driven Design:

Noun Identification technique

User	System	Picker	Packer	Stocker	Manager
Order	Warehouse	Product	Customer	Queue	API
Item	Sector	Basket	Status	Screen	Button
Priority	Report	Stock	Frame	Time	Graph
Process	Notification	ID	Details	Credential	Address
Information	Shelve	Height	Width	Depth	Description
Name	Username	Password	Error	Message	Input
Click	Cubby				

Heuristics:

1. Out of Scope
2. Attribute
3. To vague or specific.

The filtered list:

User	Picker	Packer	Stocker
Manager	Order	Product	Customer
Item	Sector	Screen	Priority
Report	Shelve	Cubby	

User:

A User is a general term that for anyone that uses the system. All users share common attributes, name, user ID and password. Users can also log into the system and log out.

Product:

Product is what is being stored within the warehouse. A Product can be described with a unique product identifier, height, width, depth and weight.

Item:

Item is an extension of Product. A Product may have many Items however an Item can only have one Product. An Item can be described with a unique item identifier, Product identifier, location, priority, expiration date, manufacture date and a status.

Order:

Order is a collection of products or items. An Order can described by a unique order identifier, shipping address and contents of Order.

Priority:

The Priority of an Item changes daily. Priority can be calculated by difference by current date and manufacture date, or it could be calculated by difference between expiry date and current date. Priority can be described by last update, priority number.

Cubby:

Cubby is a collection of Items. Cubby can be described by a unique cubby identifier, location, height, weight, depth and contents of cubby.

Shelve:

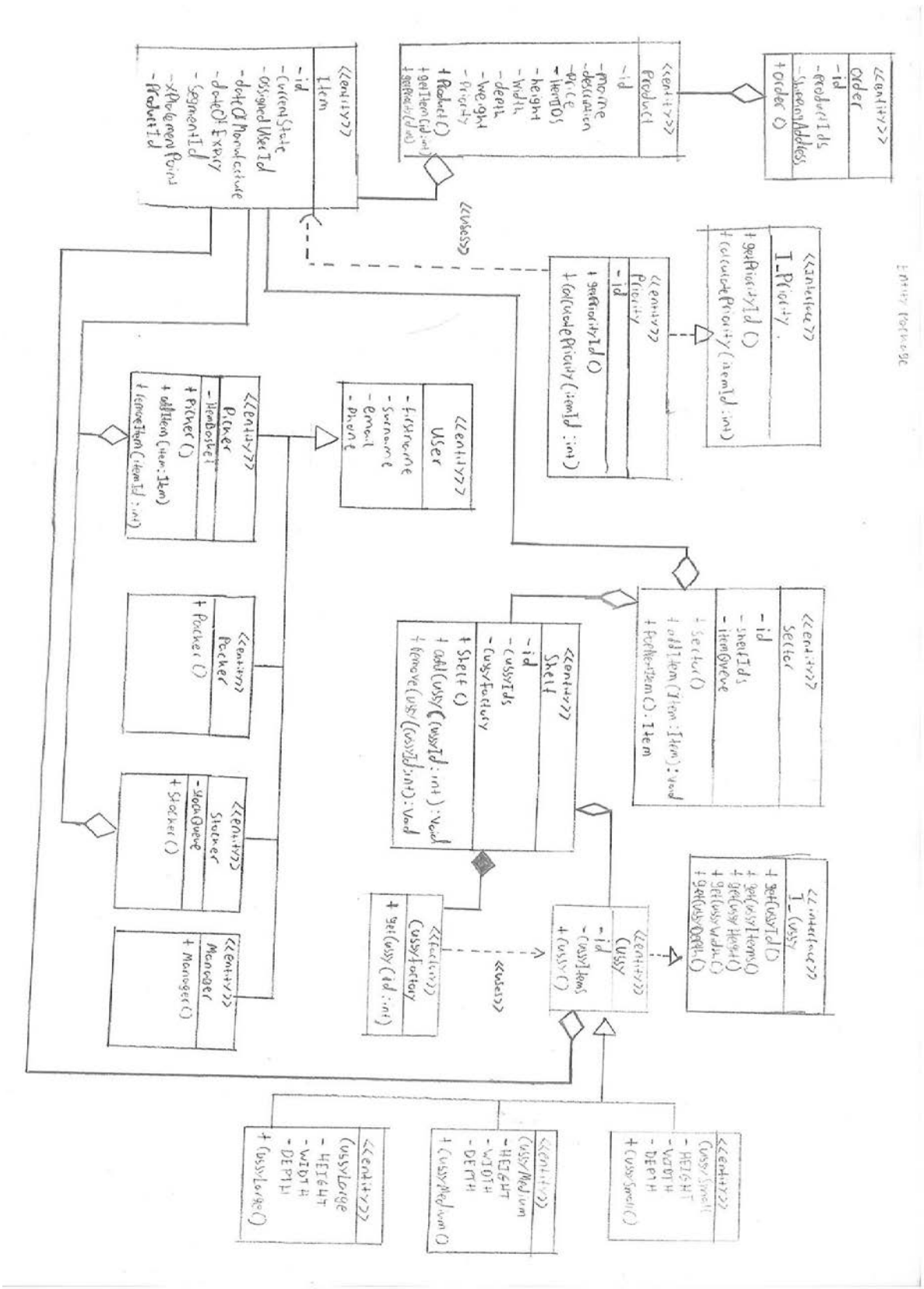
Shelve is a collection of cubbies. Shelve can be described by a unique shelve identifier, location, height, weight, depth and contents of shelve.

Sector:

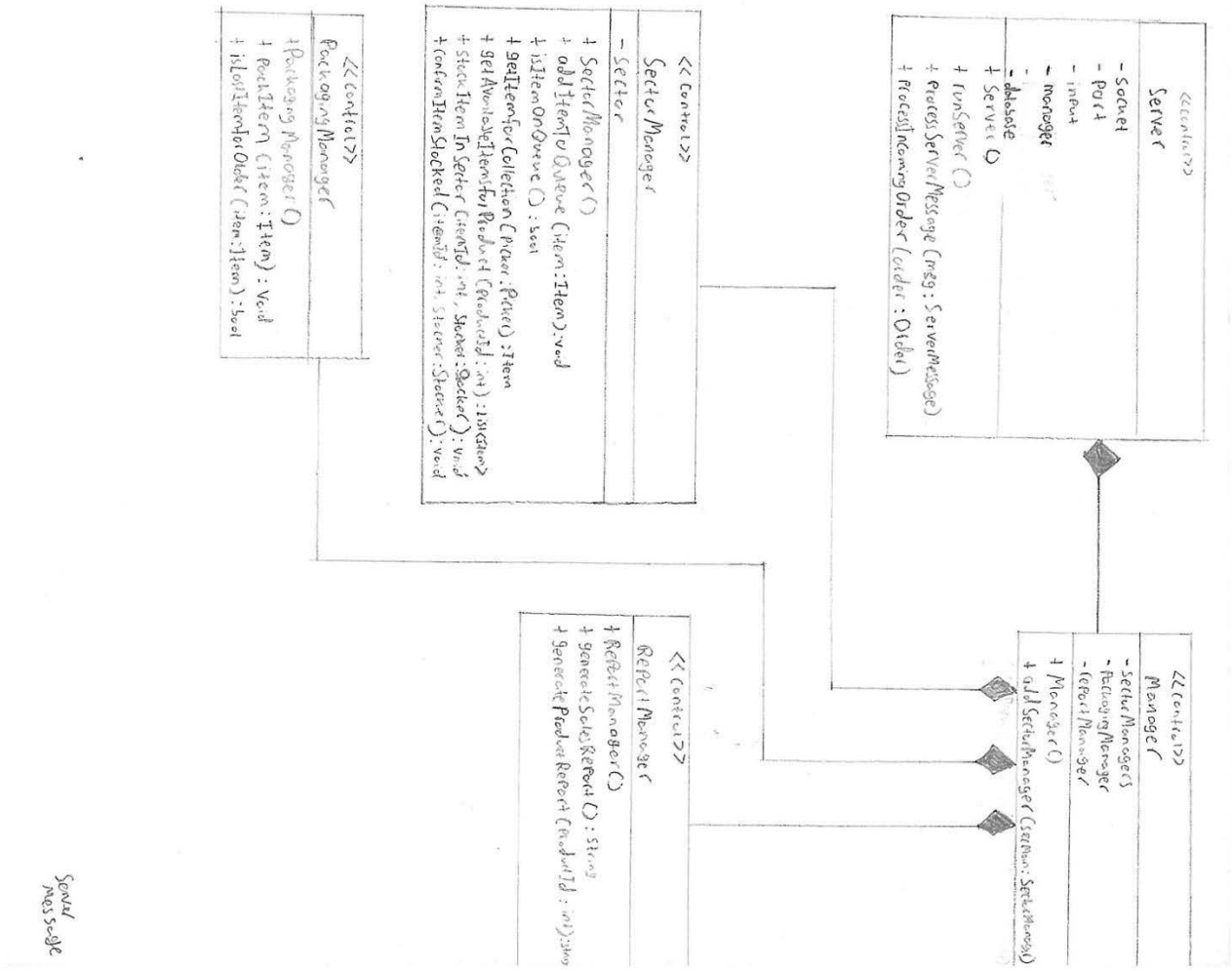
Sector is a collection of Shelves. Sector can be described by a unique sector identifier, location, height, weight, depth and contents of Sector.

Class Diagrams

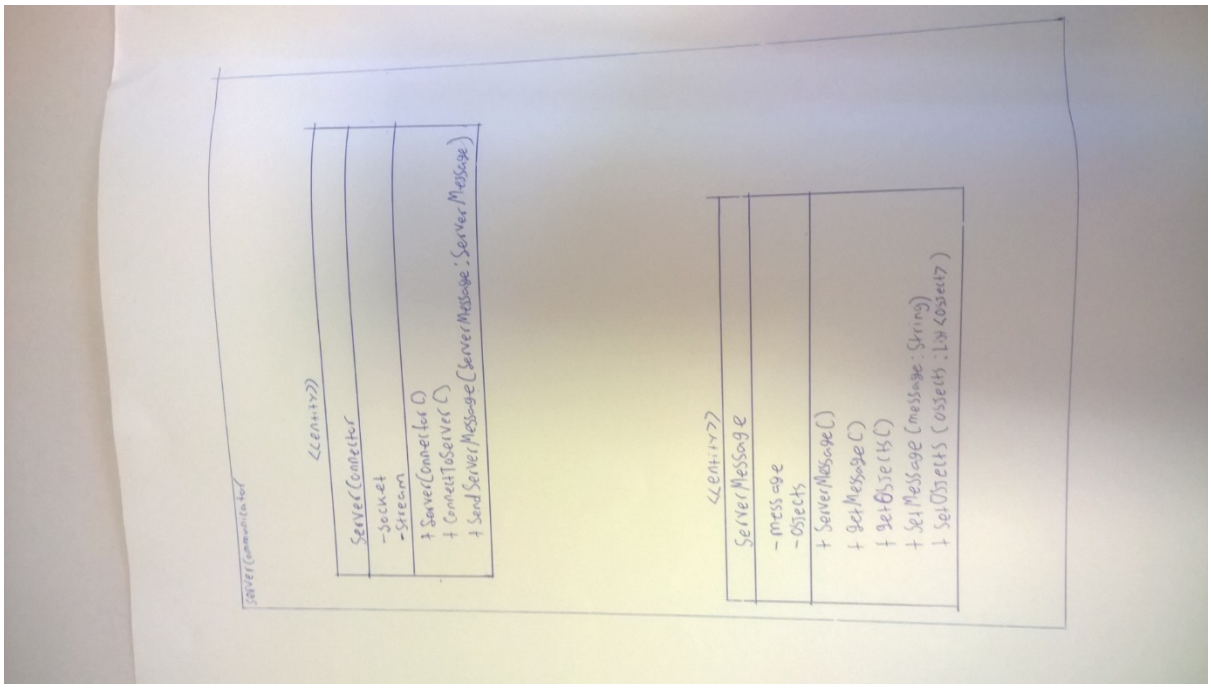
Due to the size of the system it was not possible to get all classes on a single A3 page. Database and Client are omitted.



Entity Package – Class Diagram

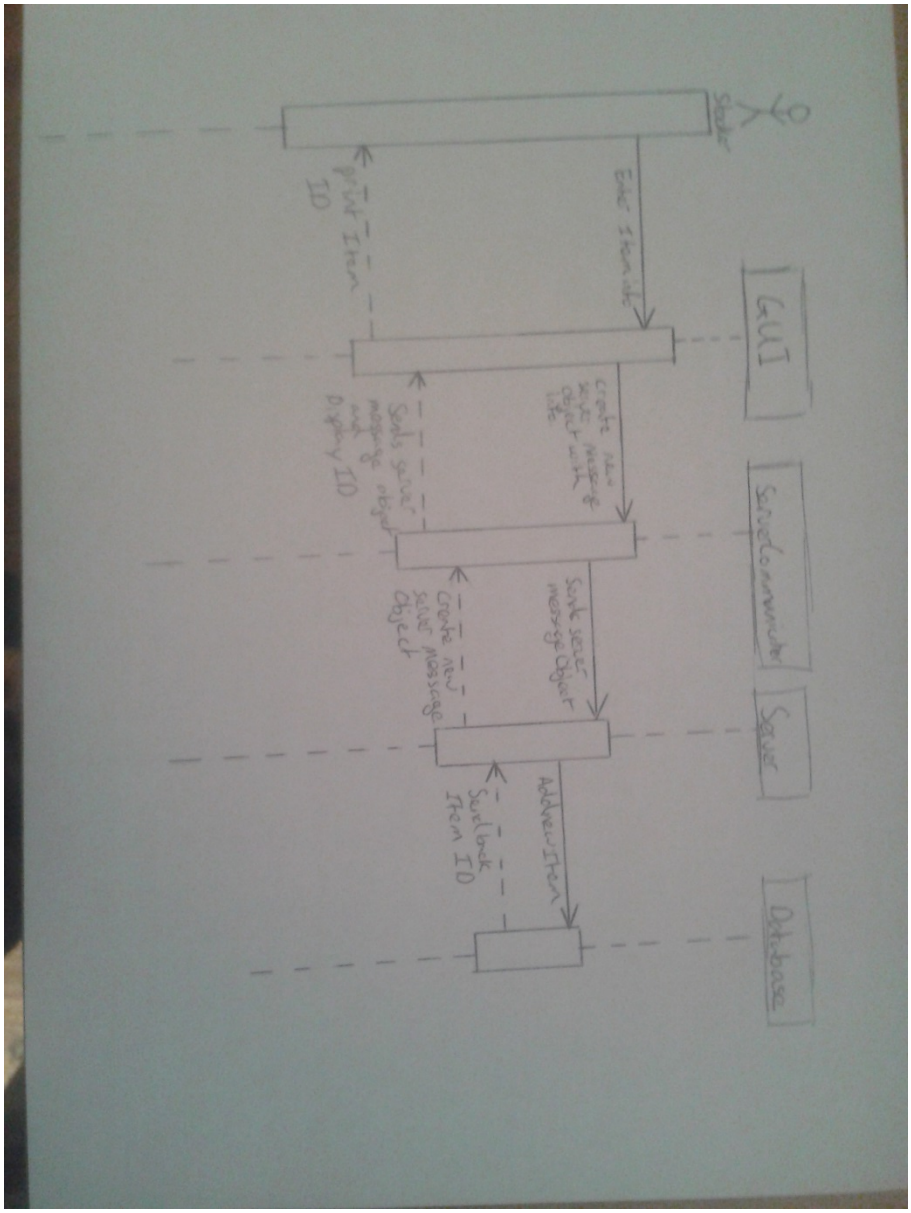


Server Package – Class Diagram

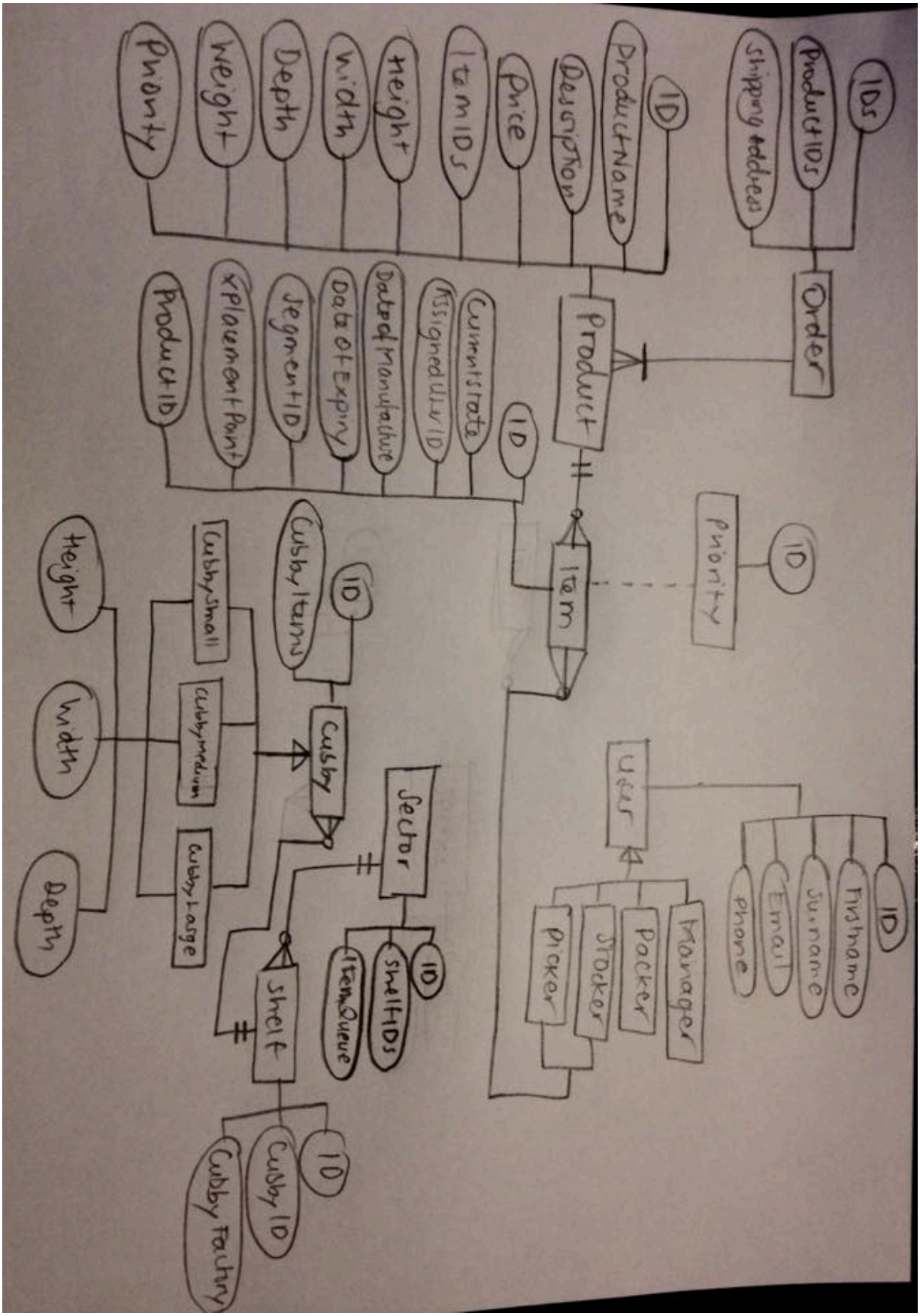


ServerCommunicator – Class Diagram

Interaction Diagram



Entity Relationship Diagram



Added Value

GitHub

We used GitHub during our project implementation phase. GitHub is a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git.

Git is a distributed version control system. It usually runs at the command line of your local machine. It keeps track of your files and modifications to those files in a "repository" (or "repo"), but only when you tell it to do so. (In other words, you decide which files to track and when to take a "snapshot" of any modifications.)

Through the use of GitHub, there we published our Git repositories and collaborated with each other while at the same time tracking each other's progress by syncing.

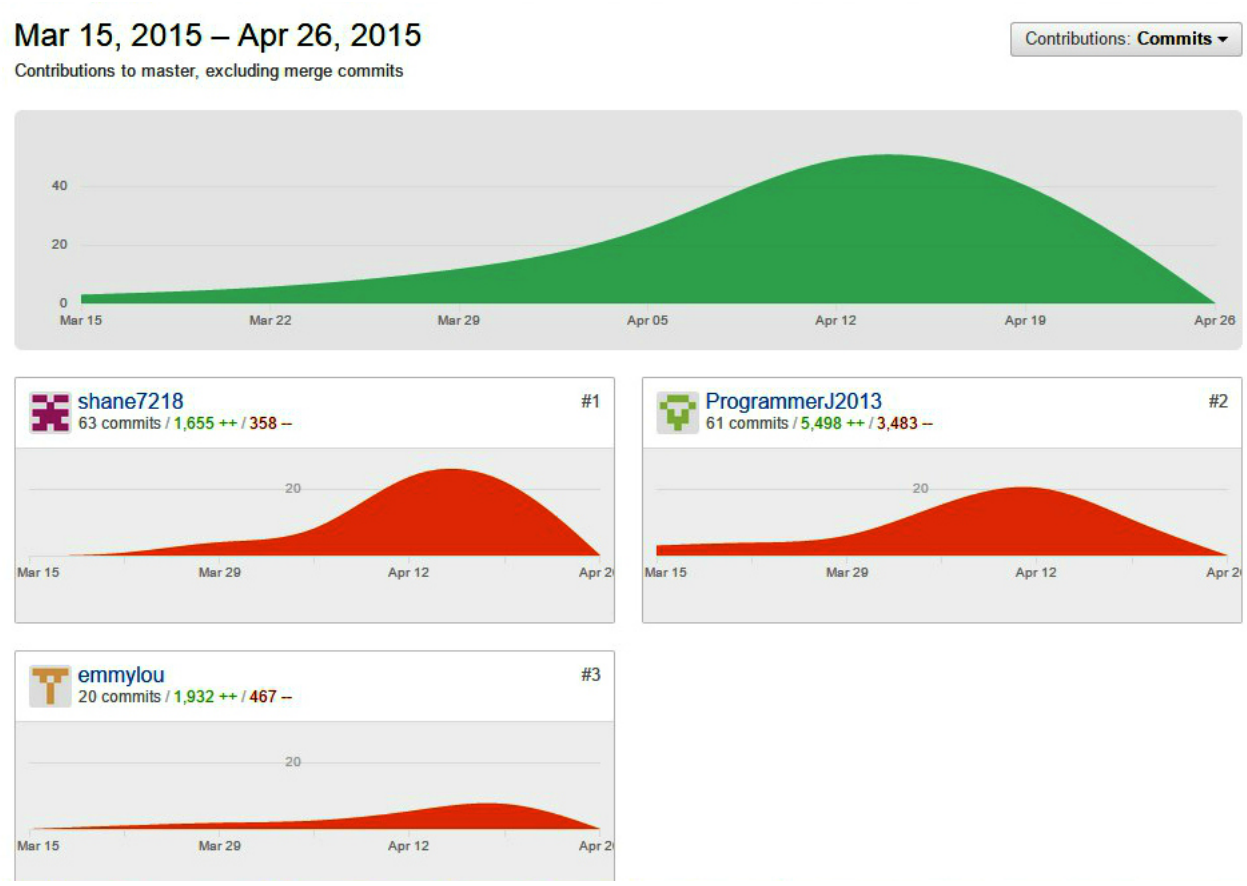
Each of us in the team had our own branches in which we locally commit our changes and subsequently push our commits from our branches to a main "*master*" branch to update our overall project.

We only pushed commits that are error-free from our branch to the master branch.

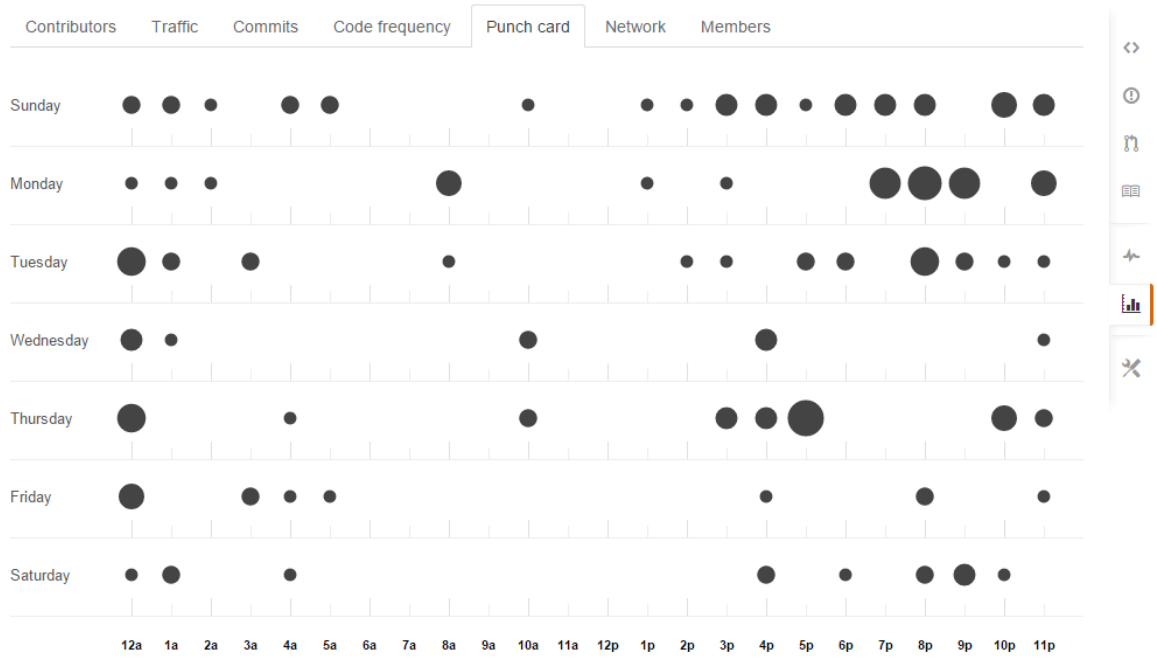
Since each and every one in the team can access each other's source code means that we had a chance to better understand how each of our program functions and more importantly help each other to fix any problems we see or add new features which we thought would be of great benefit.

We see GitHub as an added value to our project as it allows;

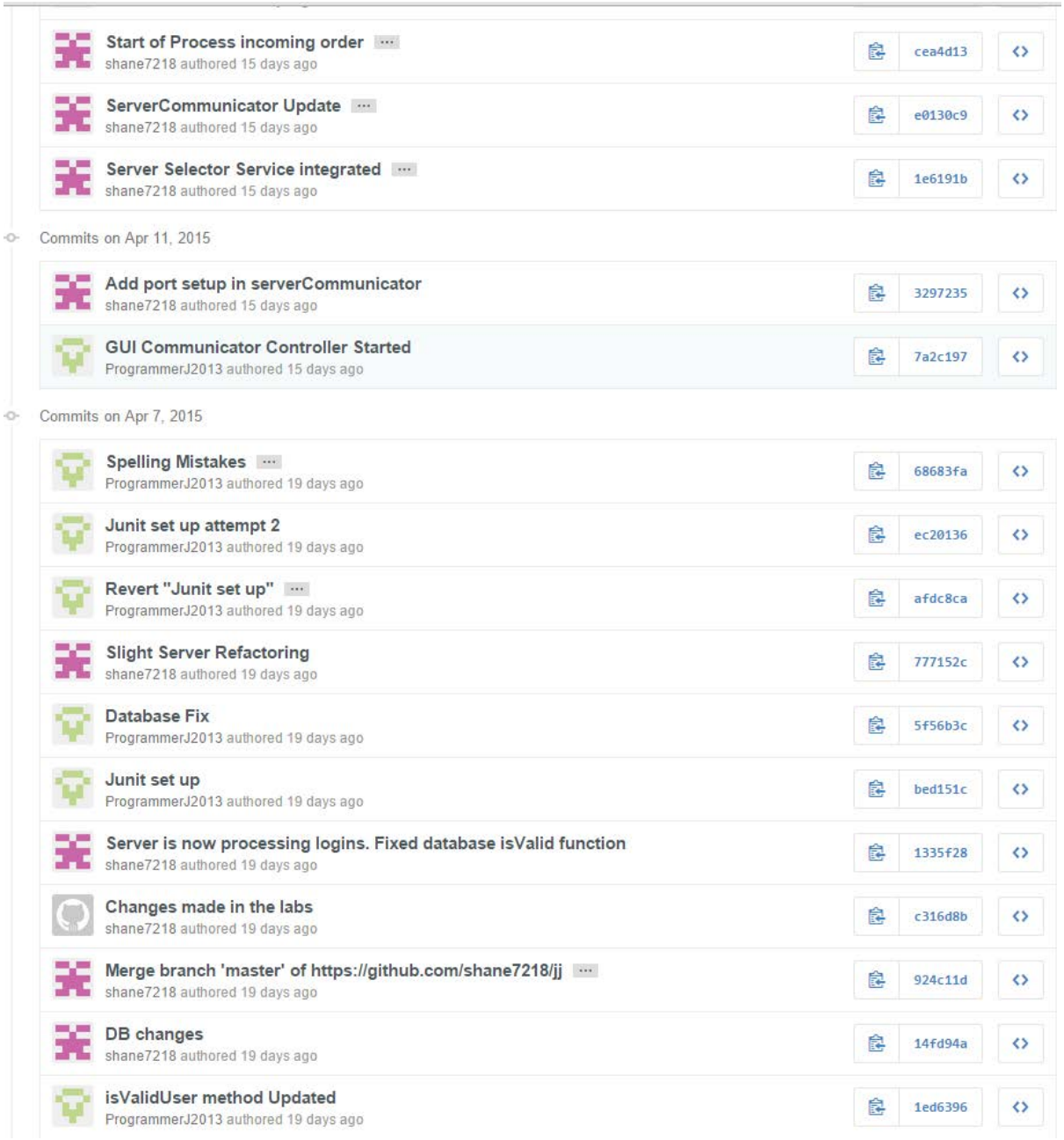
- **Integrated issue tracking** to let each of us stay on top of bugs and focus on features. Keyboard shortcuts make issue assignment and labelling fast. Only team-mates and collaborators can create and view issues on private repositories.
- **Collaborative code review** is very important. After creating a branch and making one or more commits, a Pull Request (Code + Issue + Code comments) starts the conversation around the proposed changes. Additional commits are commonly added based on feedback before merging the branch.
- **Syntax highlighted code and rendered data**
- **Revert changes to previous state** allows each of our member to cancel any merged Pull Requests and roll back any changes made.
- **Branch merging** which enables everyone in the team to merge our individual branch to *master* so that each of our contributions become part of the main body of work.



Commits – Contributions to master



Commits – Punch Card



Git commit - Examples

Compcon – Customer feedback loop

At this project's inception we chose to follow the agile software lifecycle model, with the intention of providing iterative improvements to our software based on feedback we receive from our customers. In order to ensure that we receive feedback in a consistent and constructive way as well as providing a platform for developers to directly communicate and take part in public discussions with customers we elected to use the Compcon [http://compcon.shanecraven.info] platform.

The Compcon platform provides us with live chat facilities for direct and private communication with customers and also public forums which provide public discussion capabilities. Having private chat facilities allows us to participate in one-on-one interactions with our customers which provide us with insight into issues that they would like to see addressed in future releases. Also having the ability to participate in public discussions is also a welcome addition and provides opening dialogue between users and developers. We believe that having this sort of interaction been customer and developer aid in keeping feature requests relevant to customer demands.

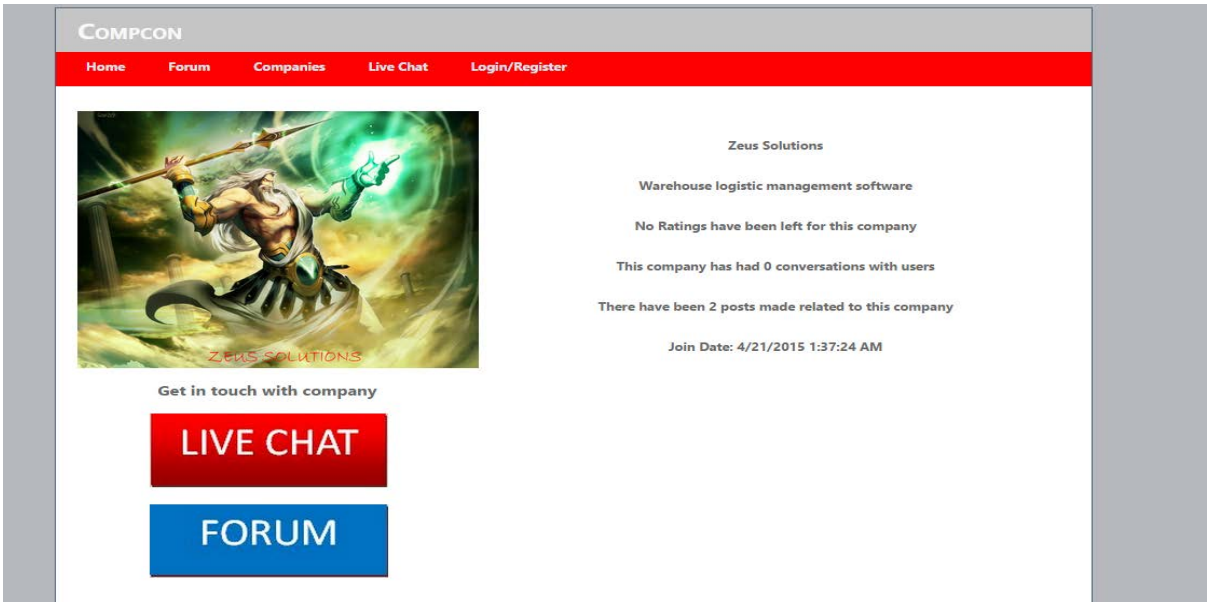


Fig 1: Zeus Solutions – Company Page

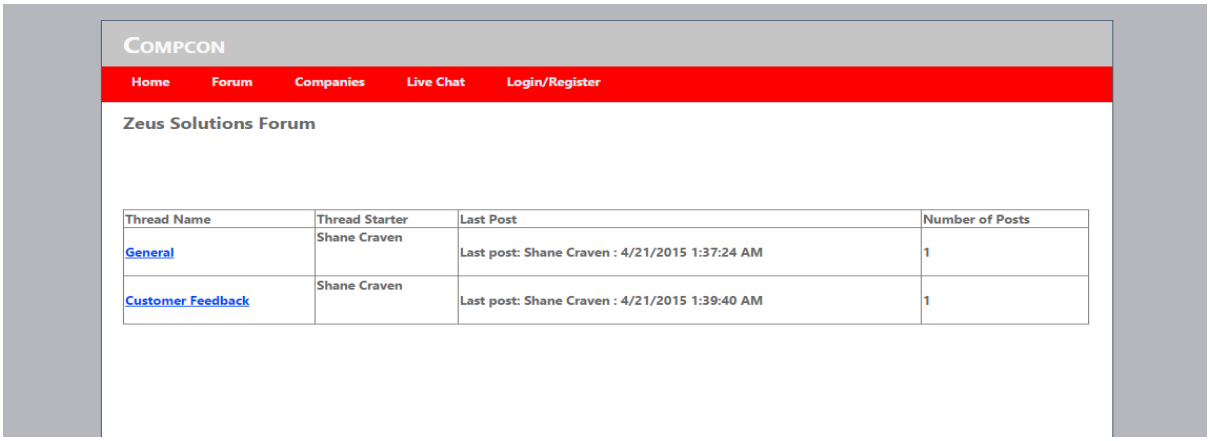
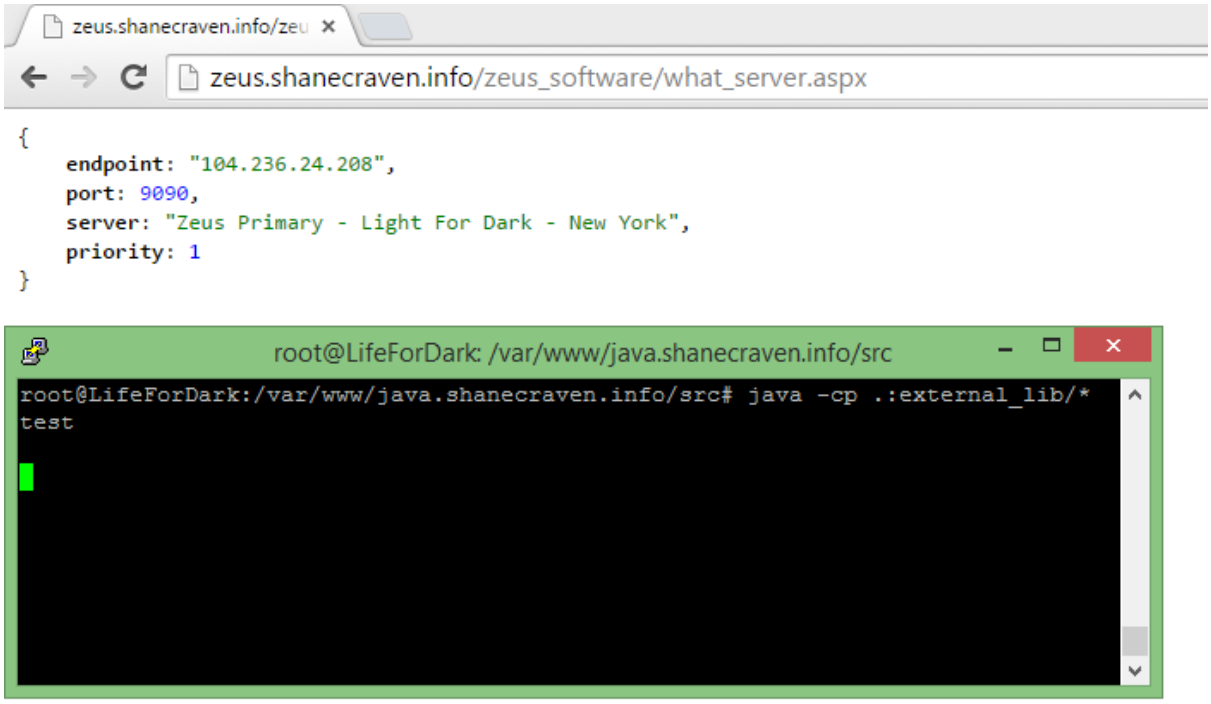


Fig 2: Zeus Solutions – Company Forum

Zeus Server Selector – Server Failover

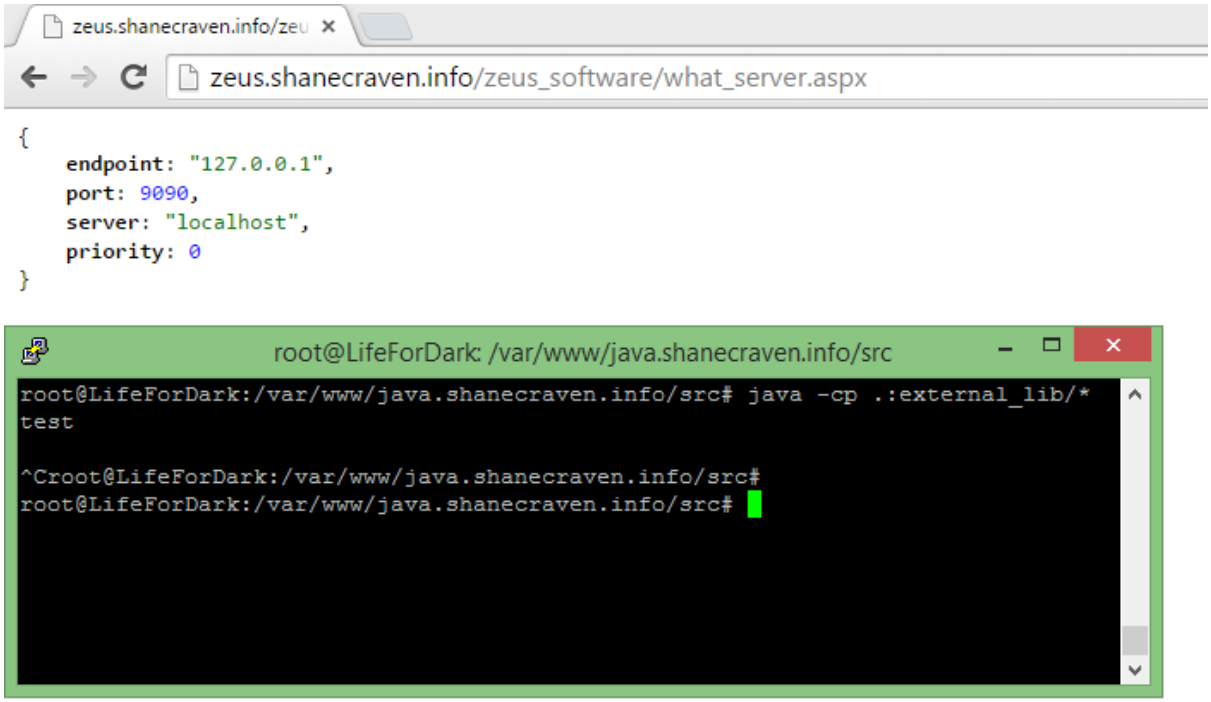
When implementing our application we tried to come up with features that would provide better deployment options in an enterprise environment. One such feature aims at improving maintainability at the scope of system deployment within an enterprise environment. If we have our server software running in a company and the host in which the server is executing on needs to be taken down for scheduled maintenance or the host was to fall over due to unforeseen circumstances we want to provide customers with automatic redundancy that is invisible to clients of the application. This is particularly crucial for large production deployments as even a short disruption to a system such as this can cause major financial loss to our customer and negative impact to customers of the company that is using our software.

Our solution to this problem is the Zeus Server Selector [http://zeus.shanecraven.info]. This system is implemented using the .NET framework and is designed using restful API architecture with JSON as the data format. For optimal reliability this system should be deployed on a separate host to Zeus server software deployments. This system knows what hosts the Zeus server software may be running on in the current environment, as well as the endpoints and ports. Each host has a designated priority which is used to determine which one should be selected to fulfil client requests. When the Zeus client application is executed it makes a call to this service requesting information needed to open a connection to a server and the service returns the information requested. This essentially abstracts the process of determining what server to use for processing requests from the client application and places it in a separate online service.



Zeus Server Selector – Main Server Online

As can be seen in the above image, the service is providing JSON data which contains the required information to open a socket to the Zeus Server Software to begin using the client application. In the below image the main server has been shut down.

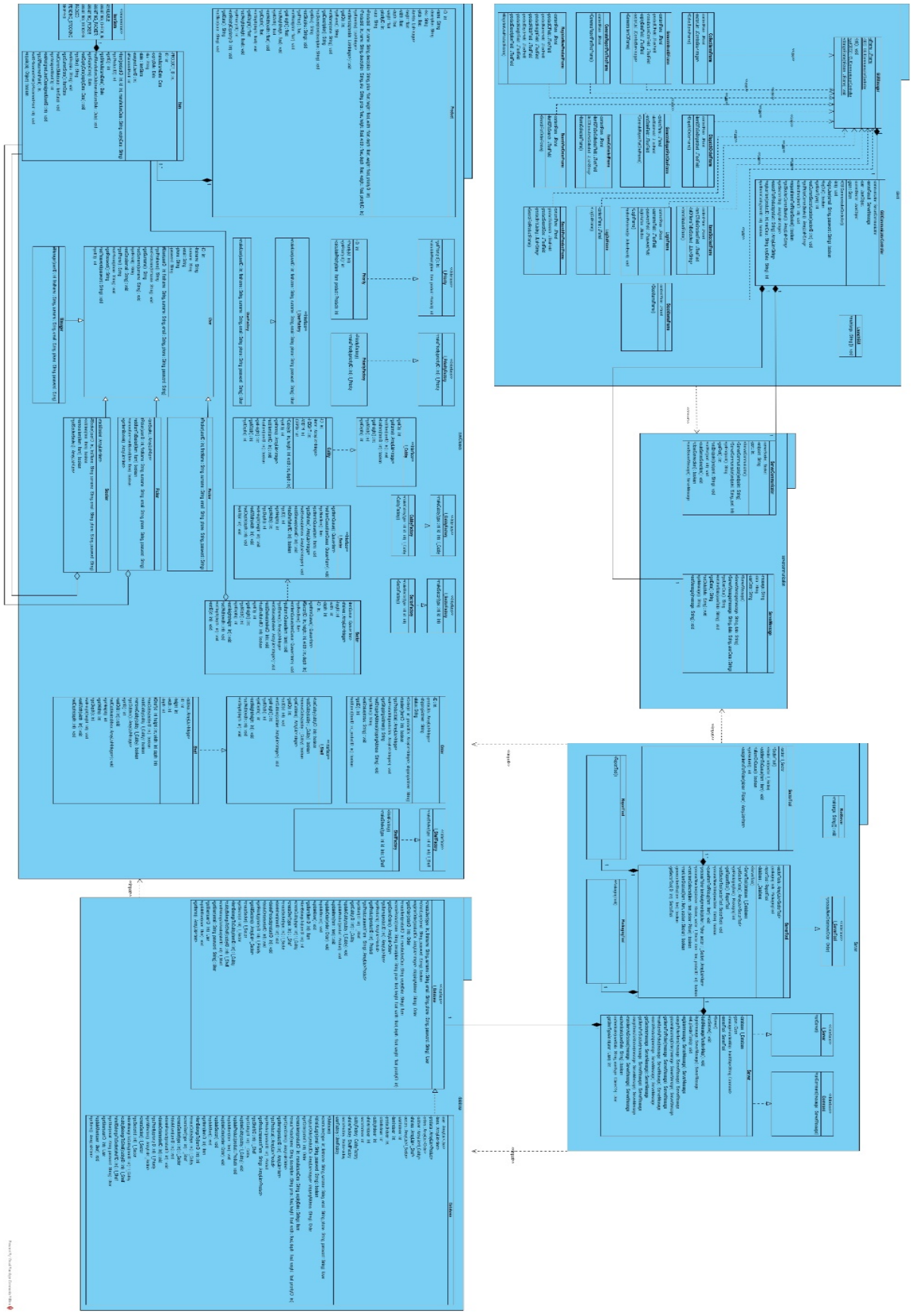


Zeus Server Selector – Main Server Offline (For the purposes of this project localhost is a backup server. In a production deployment of this application this would not be the case)

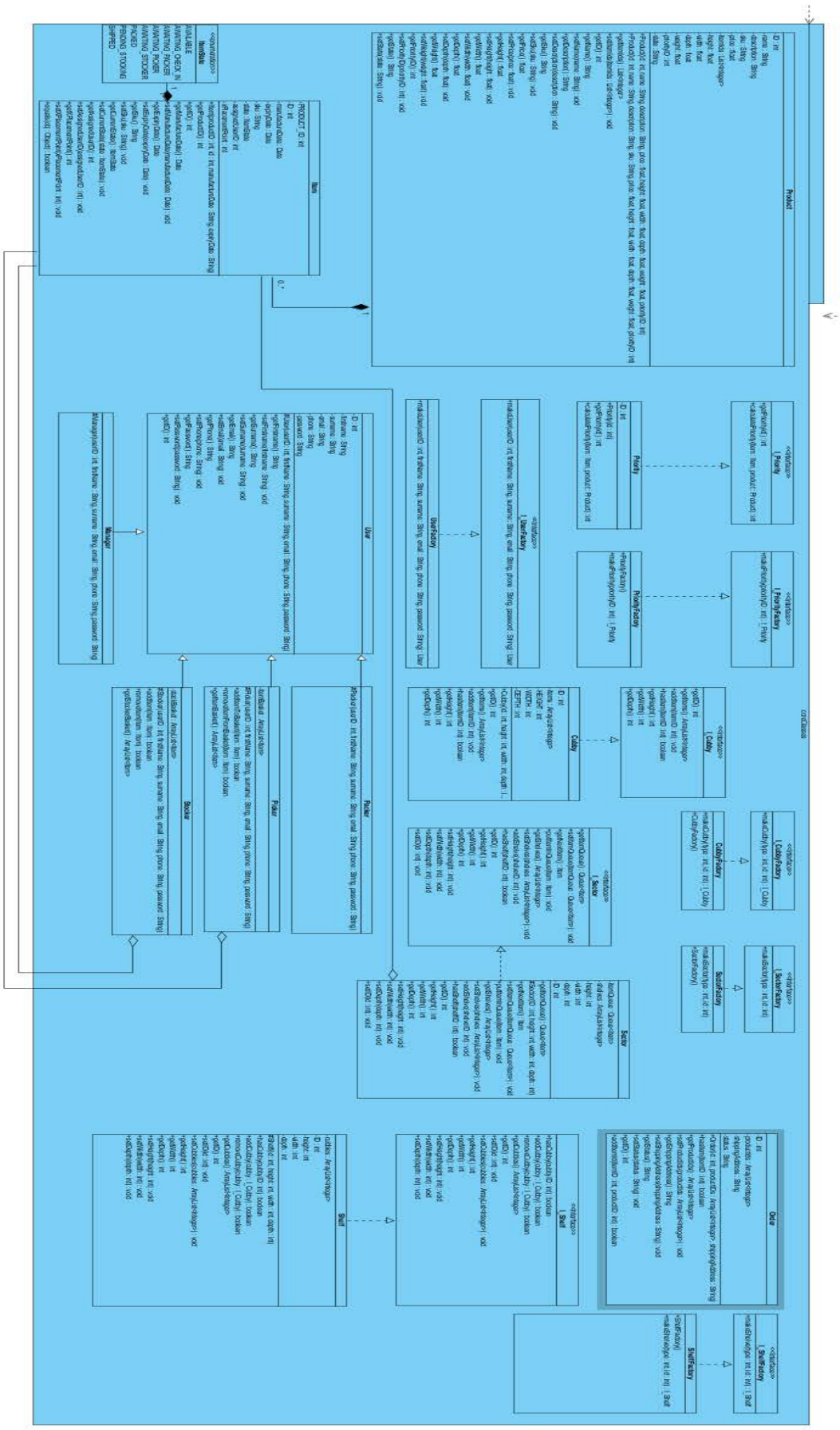
As can be seen in the above image when the main server was disabled the Server Selector returns a different endpoint. Since this single API provides all necessary information and automatic redundancy capabilities the client is ignorant to the underlying changes in the backed. This service could also be extended to provide more calculated choosing of endpoints based on environmental factors of a deployment and such changes would not require changes to the client implementation.

Design Blueprints

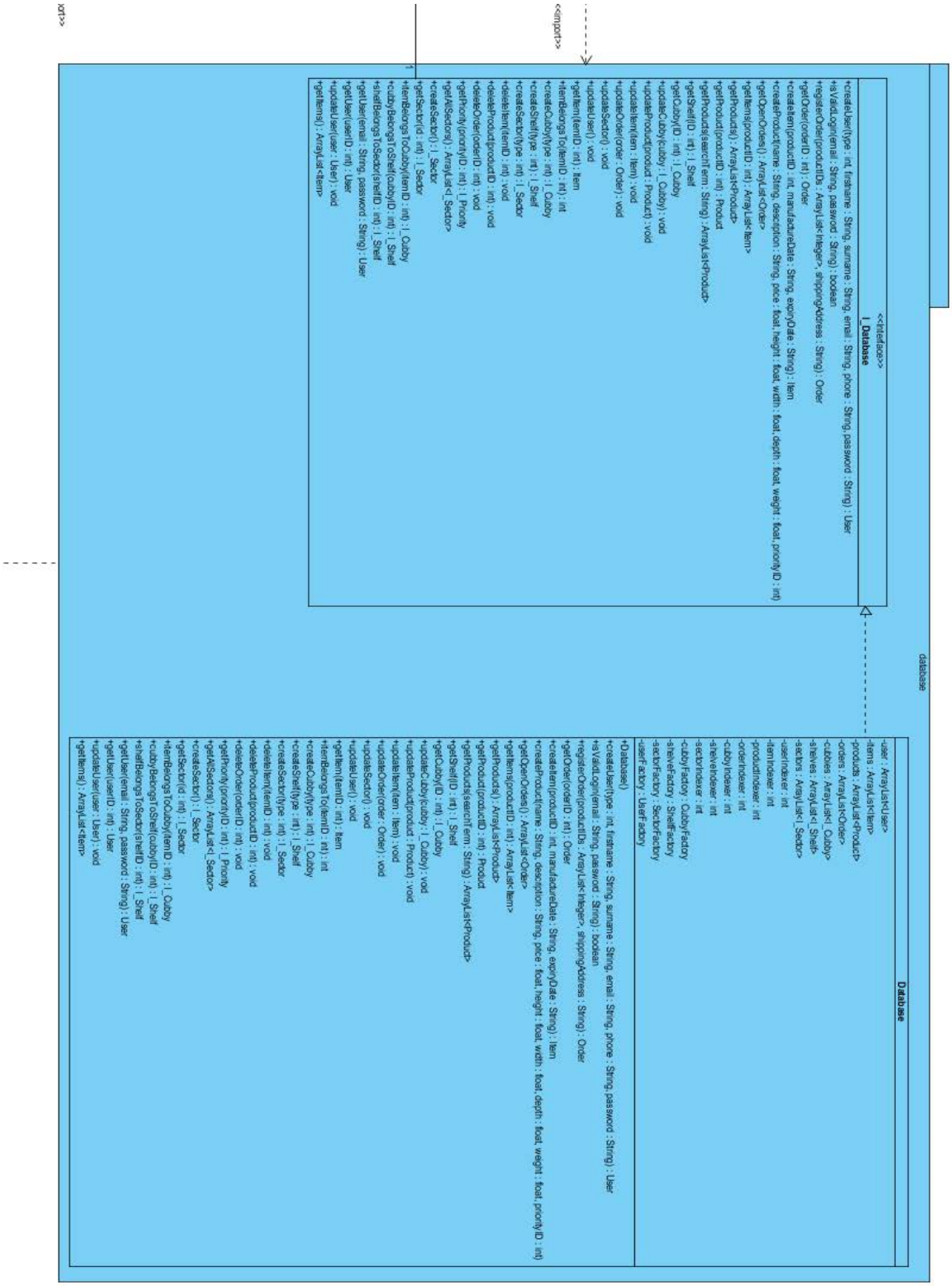
Architectural Diagram



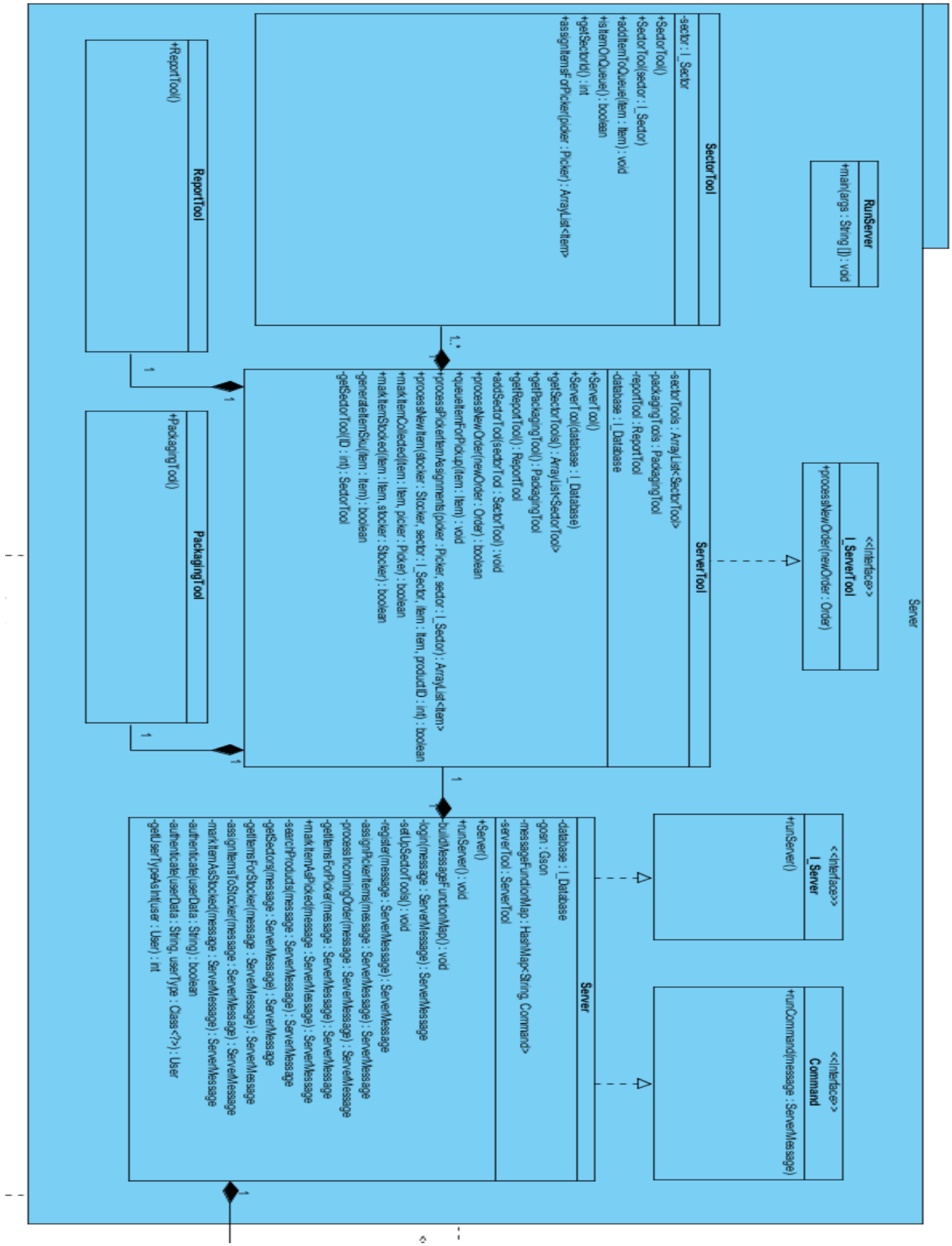
Class Diagram – coreclasses



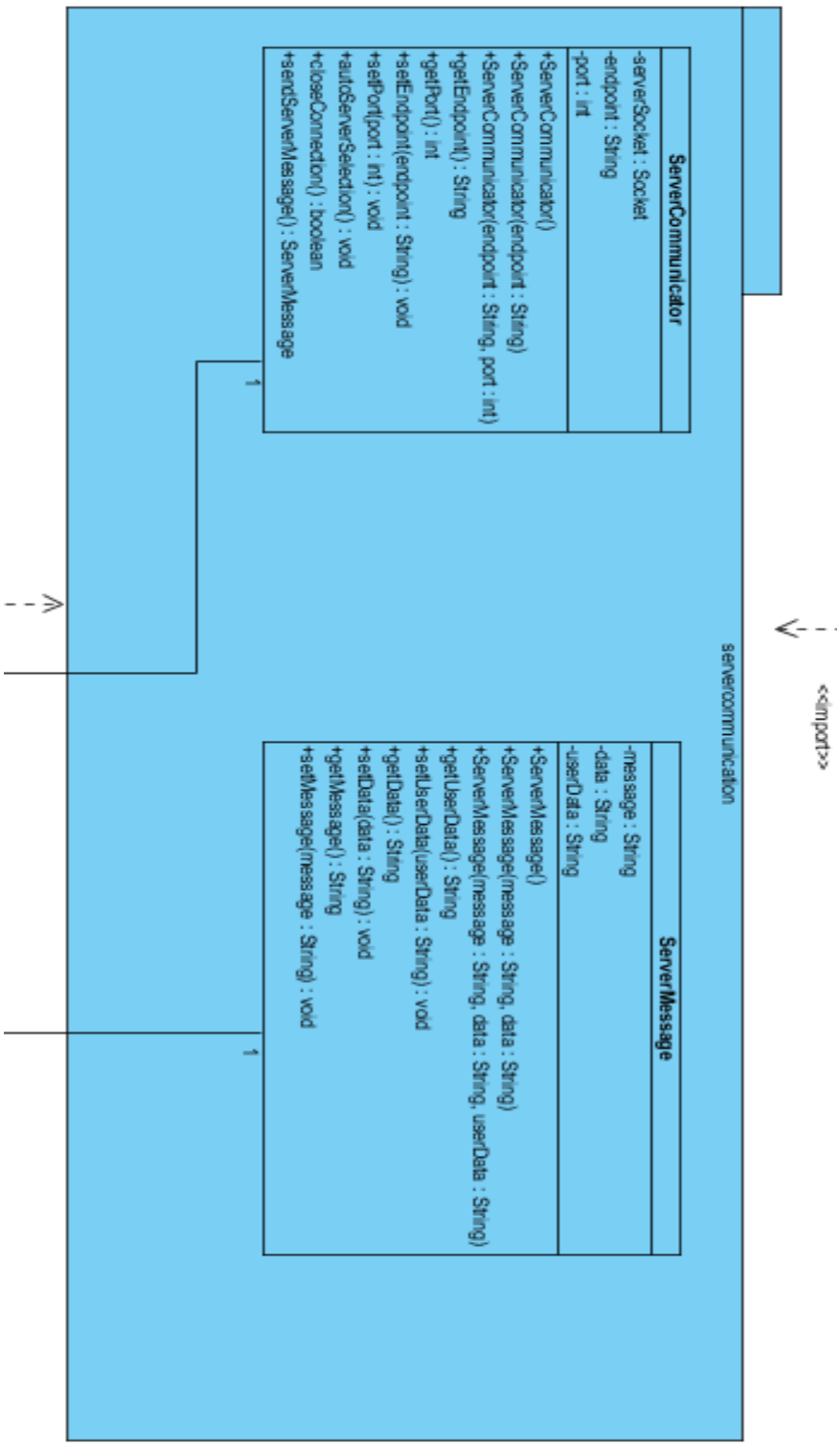
Class Diagram – database



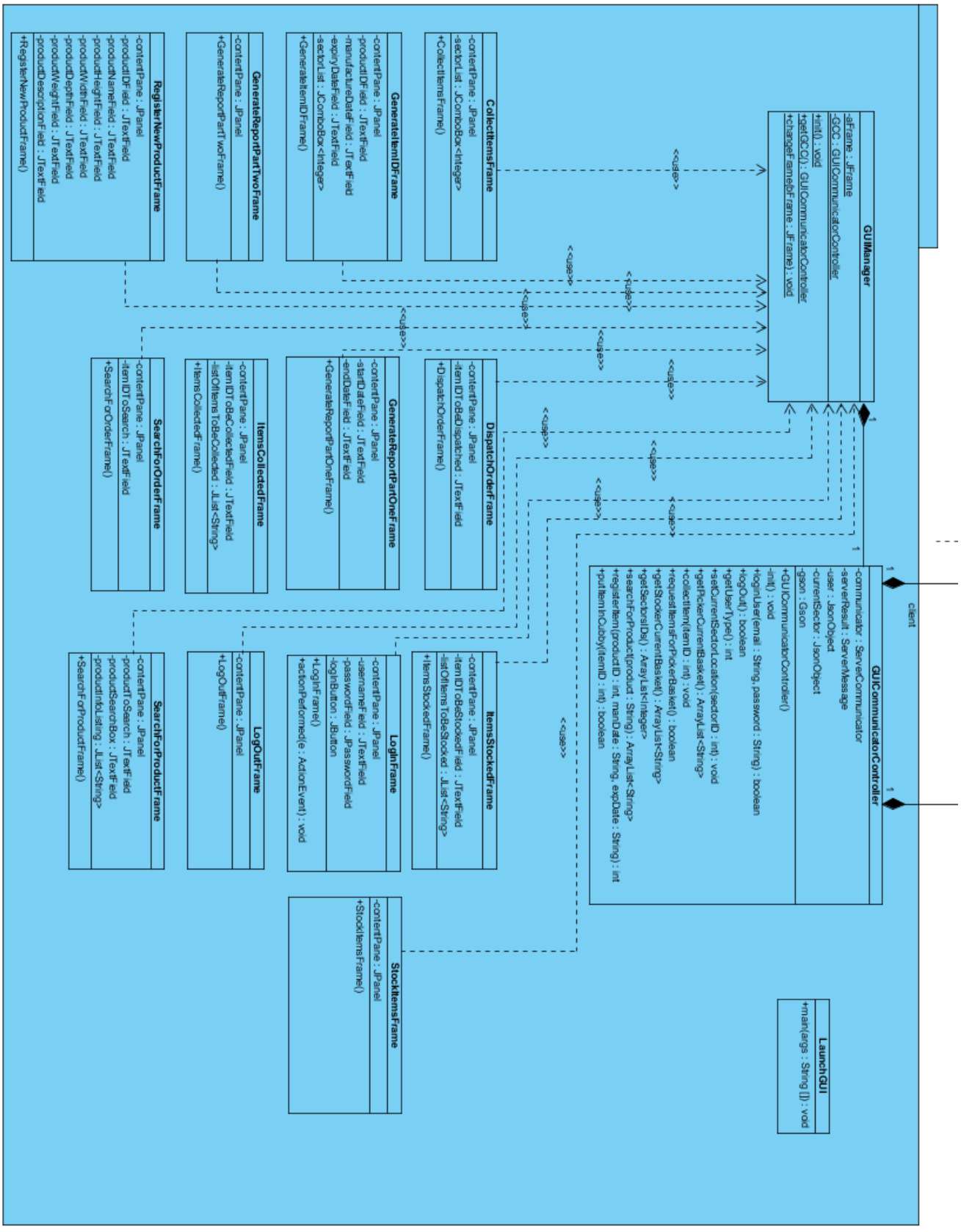
Class Diagram – server



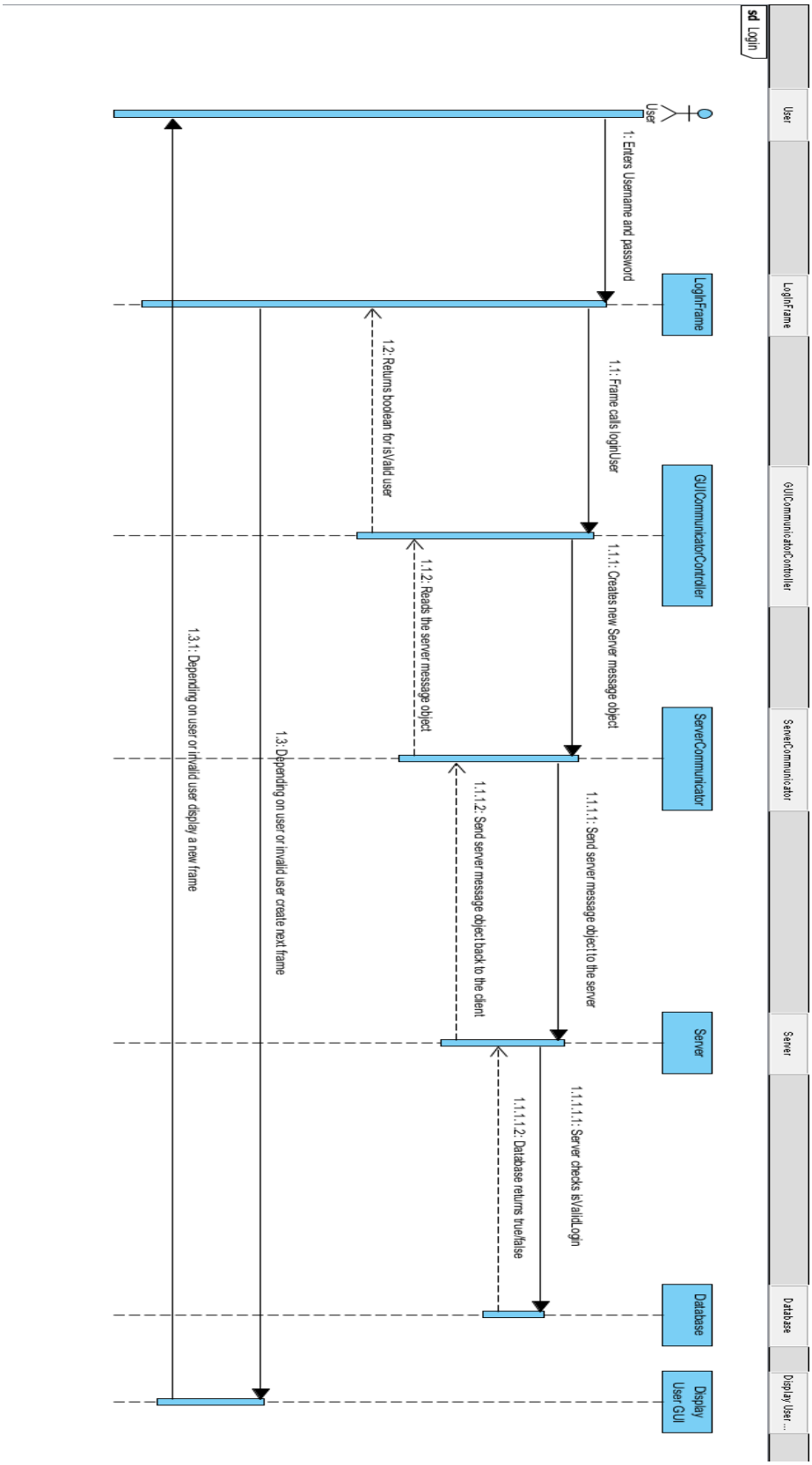
Class Diagram – servercommunication



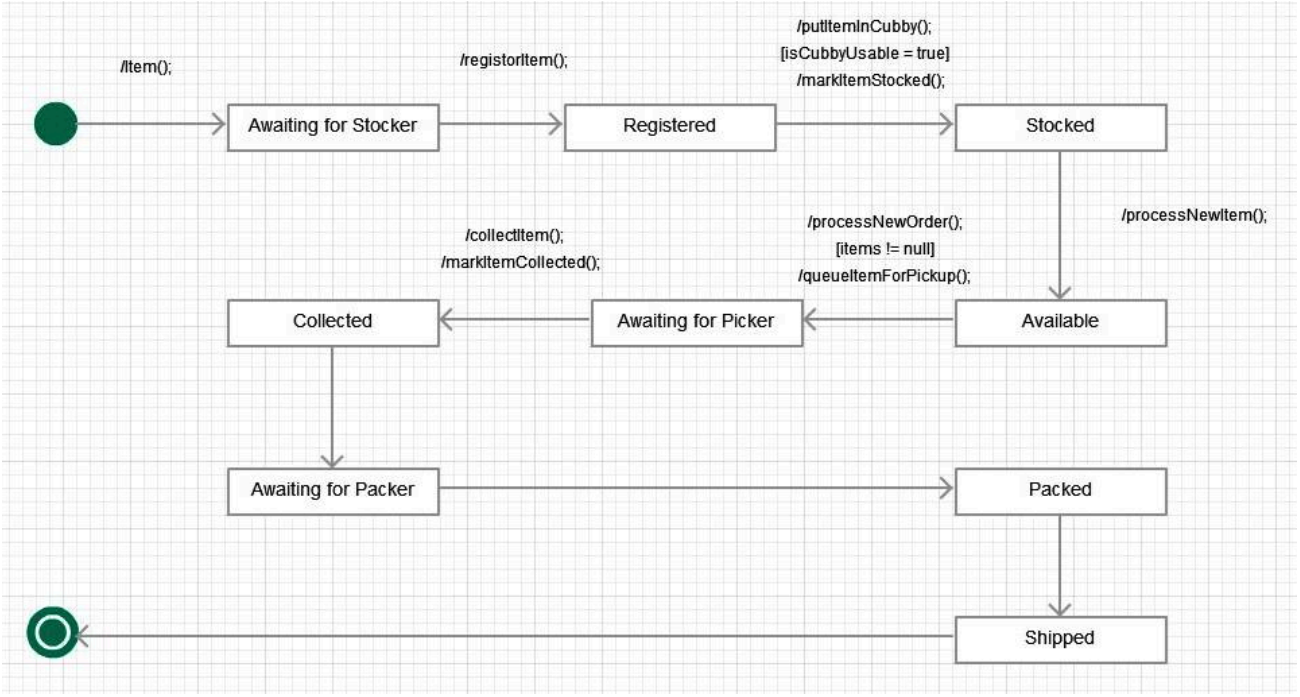
Class Diagram – client



Interaction Diagram - Login



State Chart


/Item();
Awaiting for Stocker
/registorItem();
Registered
/putItemInCubby();
[isCubbyUsable = true]
/markItemStocked();
Stocked
/processNewItem();
/processNewOrder();
[items != null]
/queueItemForPickup();
Available
/collectItem();
/markItemCollected();
Awaiting for Picker
Collected
Awaiting for Packer
Packed
Shipped

Descriptions of the patterns

Creational Patterns

These patterns are used to deal with class and object instantiation.

Singleton

Singleton design pattern provides the benefit that only one instance of an object can be created within the system. We used this pattern in the database class.

Factory

The main benefit to the factory design pattern is that it localizes the creation of objects within the system. This makes it easier to maintain and modify the system as the system grows. In our project we use a number of factory classes (examples: CubbyFactory, ShelfFactory, UserFactory). These factors create concrete objects based on what type passes into the make function inside the factory.

Behavioural Patterns

These patterns are used to deal with the communication between objects.

Command

The main benefit to the command design is that we can encapsulate a request as an object. In our project we use this pattern in the server implementation. Each command object has a method called runCommand which has to be overwritten.

Observer

The main benefit to the observer pattern is that we can attach multiply observers to a subject. In our project we only have one observer per subject being used, although we know we could have used more. The subject in our case being the Jbutton and the observer being the ActionListener. When the Jbutton is pressed a notifyAll is sent out to all the ActionListeners subscribed to that Jbutton. When the observer receives notify from the subject, the observer performs and action if need.

Critique

Analysis Artefacts and Design – Compare and Contrast

When comparing and contrasting the analysis artefacts from that of the design artefacts we noticed that the overall structure of the system remained somewhat familiar although extensively refined at the design stage. This was due to our original approach to the analysis stage where we focused too heavily on what the finished system would look like and not on following the analysis process correctly. We found that by trying to think of the finished product without first breaking down the problem into fragments caused all sorts of issues and resulted in Paralysis by analysis. We believe this highlights the importance of following the prescribed approach.

Server and network communication observations

The Server side implementation received a number of improvements during the implementation phase. The analysis design did not account for how the data received from the client would be processed on the server side. During the implementation it was determined that placing data parsing logic in the Server itself was the best course of action.

The command design pattern was added during the implementation step. This allowed the server's API to be extended without the need for excessive branching logic. It also simplified the logic needed by the server to accept incoming connections from clients thereby increasing extensibility and reducing required maintenance.

Database and Server side object observations

In the analysis stage we had a CubbyFactory which had the planned use case of allowing different size cubbies to be constructed. This turned out to be unnecessary and was removed during the implementation phase. Instead the CubbyFactory itself sets the size of the cubby object at creation. Interfaces and factories were added to other objects which made server side implementation much more flexible. We also deemed it appropriate to introduce an enumerator to keep track of the item state as it passes through the system.

Implementation Criticisms

Server and network communications

The use of sockets as the communication median between client and server could become problematic as the system expands into the future. Since socket programming is done at a lower level the cost of implementing and maintaining the code that goes with this is much higher. On reflection building the system around a protocol such as HTTP would provide a cleaner and more extendable implementation. We could expose the functionality of the server through a restful API. Implementing the system in this way would eliminate the need to deal with sockets and also significantly reduce coupling since a proprietary object (ServerMessage in the case of this implementation) would not be needed, the approach would also provide language independence, and would make client implementations easier.

I also feel the current separation of logic in the server is not sufficient and could be refactored further. The original idea was that the Server class itself would deal with JSON parsing and would then call the relevant ServerTool to handle the processing of the parsed data. On reflection I believe this could be broken down further into a more modular approach to parsing of requests. This would create a much more extendable and maintainable server side implementation.

The Zeus Server selector is currently used to provide fail over functionality for the customer. In a realistic distributed enterprise environment this would not be a suitable way to determine what server a client should communicate with. This functionality could, in theory be carried out by a load-balancer which would provide the functionality the Zeus Server selector provides while also providing load balancing of client requests and ensuring that the customer's infrastructure is being utilized to its full potential.

The implementation of the algorithm to determine what shelf an item should be placed on when it arrives in the warehouse is, in its current form, poorly primed for extending in the future. This would benefit greatly by following the same approached used to determine what Item to use to fulfil a customer order. By providing a Priority interface we can implement custom logic to determine the optimal location to store a project using the Strategy Design pattern.

Due to time constraints functionality within the ReportTool and PackagingTool was not completed. In a full scale implementation the ReportTool would provide methods that would generate various reports. The PackagingTool is responsible for providing methods for the last stage of order fulfilment related to shipping completed orders to customers and related processes.

Database

Currently the database layer is virtualized. In future implementations this class would be interacting with a live database server. If the database is updated this should not affect the implementation to the server as long as the new database class implements the database interface (I_Database) or another interface that extends the I_Database.

Core Class Package Communications

One issues which we did not rectify in the implementation was our coreclasses package. This package should have been broken down into clearer and more meaningful packages for consumption. In its current state it simply acts as a container for classes that do not fall into the other packages.

GUI Communicator Controller

This class should be implementing the singleton design pattern. Currently it is possible to create a number of instances of the object, each instance of the object would require a server socket. Also the class name may want to be changed.

Client

The client implementation in its current form is highly coupled and poorly architected. When completing the analysis phase of the project not enough time was spent discussing the architecture that would be used on the client side. This resulted in a sub-par implementation that would, for a full scale implementation, be completely overhauled. Currently the client spawns a new JFrame for each screen that user can view. A better approach would have been to have a central Panel factory that generated the required panel that would contain GUI controls needed to perform the desired action and this would be inserted into a single JFrame in a pluggable fashion. The GUI should have been able to deliver a user interface that is at a better standard that what it is now. Some of the functions' arrangements within pages were poorly placed within the page and some were not consistent. More functions were added in the GUI throughout the implementation phase. Some of the functions were not clearly labelled. An example of this would be when a user chooses which *sector* they are in. Even though this function was not in the previous design, the function itself was not clearly defined in the user interface. The location of the GUI window itself is not centred to the screen which lowers the standard of what it should have been.

References

Cite it Right, A guide to the Harvard referencing system: http://www3.ul.ie/referencing/miniSite/citeItRight.htm

Anon., 2015. *blog/wp-content/uploads/2010/06/charts_screen.JPG.* [Online]
Available at: http://www.dhtmlx.com/blog/wp-content/uploads/2010/06/charts_screen.JPG

Anon., 2015. *Study/Bachelor/Year_2/Object_Oriented_Modelling/Summary/Object-Oriented_Design_Process.* [Online]
Available at:
http://wiki.msvincognito.nl/Study/Bachelor/Year_2/Object_Oriented_Modelling/Summary/Object-Oriented_Design_Process

Banas, D., 2015. *Playlist of Design Patterns Video Tutorial.* [Online]
Available at: https://www.youtube.com/watch?v=vNHpsC5ng_E&list=PLF206E906175C7E07

canstockphoto, 2015. [Online]
Available at: http://ec.l.thumbs.canstockphoto.com/canstock22363682.jpg

CanStockPhoto, n.d. *canstockphoto.* [Online]
Available at: http://www.canstockphoto.com/zeus-wielding-thunderbolt-circle-retro-22363682.html
[Accessed 8 March 2015].

Hailu, S., 2014. *Design Patterns Explained Simply.* 1st ed. s.l.:Sami Hailu.

Molich, R. a. N. J., 1990. *Improving a human-computer dialogue, Communications of the ACM 33.* s.l.:s.n.

Simon Bennett, S. M. R. F., 2010. *Object-Oriented Systems Analysis and Design.* 4th ed. New York: McGraw-Hill Gigher Education.

VisualParadigm, 2011. *youtube.* [Online]
Available at: https://www.youtube.com/watch?v=UzUUZRK_Q6Y
[Accessed 25 April 2015].

VisualParadigm, 2011. *youtube.* [Online]
Available at: https://www.youtube.com/watch?v=18_kVlQMavE
[Accessed 25 April 2015].

Wikipedia, 2015. *Java (programming lanaguage).* s.l.:s.n.