UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

CS4125 - Systems Analysis & Design
Group Project Report - SEM1AY18/19

Group: Jabronis - Team Members:

15159558	Alex Hutt
15183769	Michael Meskell
16172639	Henry deLongue

Contents

1. Narrative Description

For our project we chose to work on an online retail store with emphasis on stock management and allowing for customers to access the system also.
The system works as follows:

The system keeps track of the inventory within the store. The stock is checked as the program launches and is done in the background. If transactions are made, the system keeps track of which items are being bought/returned and increments/decrements the stock levels accordingly. If an item in the stock falls below the threshold in place, the system sends an automatic order for new stock items.

Employees are split into two types: Manager and Stock Employees. Stock employees are given options to order stock manually, view current stock levels, or add/remove stock from the database manually. When an order is made by the employee, it gets sent to the manager for approval.

The manager can order stock without need for approval. As the manager oversees the store, they are given access to employee and customer databases. They can choose to add or remove employees from the database and do the same with customers. There is a final option to view order requests made by the stock employees and approve or disapprove them.

Customers do not have access to any of the databases apart from the products database. They are given the option to either buy goods or return items. They are asked to give their customer ID which allows the system to find their details in the database, meaning they do not need to input their credit card number each time they wish to buy something.

The customers are shown a list of products available in the store and can choose which ones they would like to buy. A voucher system was implemented to allow customers to receive

discounts on products based on how frequently they shop in the store. The system will then read through the customers order and tally up the prices to find the total amount that they will have to pay. They system then increments the store account accordingly.

2. Software Lifecycle Model

For our project we looked at a few different software development lifecycle models.

2.1. Waterfall

The waterfall model was a system we were used to working in: Start the work, finish it, move on. However, its lack of flexibility was a point against it, as was its inability to enable developing further features after finishing a phase. In the end, we decided that it was too rigid for us to want to use.

2.2. Rational Unified Process (RUP)

RUP was the second model we looked at, however it was quickly discounted due to the heavy amount of modelling involved. Given that we had not tackled this type of project before, we felt it was best to keep to a more flexible timeline, as opposed to restricting ourselves heavily with too much up front design.

2.3. Agile

After looking at these options, we decided Agile would fit the project best. We liked the iterative process and more spread out planning of work flow. The flexibility to give parts of the project more time for developing or bug fixing was invaluable during our development.

3. Project Plan

3.1. Planning Summary

We decided early on to meet 1-2 times a week, and set our several short term plans, using the project plan timetable shown in the specification. After we set up GitHub, we agreed on communicating through Facebook messenger and decided to use the Java Eclipse IDE to develop our code.
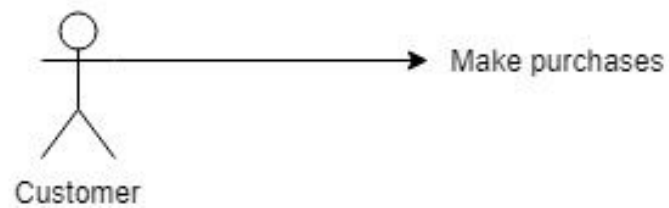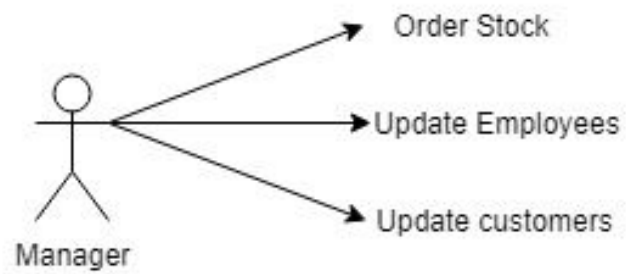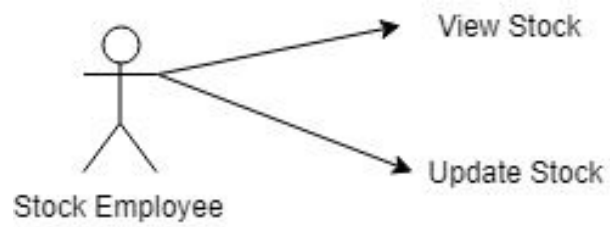
3.2. Project Plan: Timetable

Week	Workflow
3	Setup team roles, agree on scenario, learn to use Github, research on existing projects
4	Requirements
5	Analysis
6	High level architecture
7	Coding Iteration 1: Basic infrastructure and 2 key use cases
8	Coding Iteration 2: 2 more use cases
9	Coding Iteration 3: Another use case and MVC (GUI)
10	Coding Iteration 4: Another use case and Added Value
11	Overrun
12	Architecture and Design Recovery

3.3 Project Roles

	Role	Description	Member
1	Project Manager	Sets up group meetings, gets agreement on the project plan and tracks progress.	Michael
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report	ALL
3	Requirements Engineer	Responsible for requirements.	ALL
4	Architect	Defines system architecture.	Michael
5	Systems Analyst	Creates conceptual class model.	Alex
6	Designer	Responsible for recovering design time blueprints from implementation.	Michael
7	Technical Lead	Leads the implementation effort.	Alex
8	Programmers	Each team member to develop at least 1 package in the architecture	ALL
9	Testers	Testing of software	ALL
10	Dev Ops	Must ensure each team member is competent with development infrastructure, e.g. GitHub, Eclipse, etc.	Harry

4. Use Case Diagrams

Stock Employee
- View Stock
- Update Stock

Manager
- Order Stock
- Update Employees
- Update customers

Customer
- Make purchases

5. Use Case Descriptions

5.1. Use Case Description - Add Employee

USE CASE		Add Employee
Goal in Context		For the manager to add an employee from the database
Scope and level		Company
Preconditions		Employee is hired
Success End Conditions		Manager adds Employee from the system
Failed End Conditions		Manager is unable to add an employee
Primary, Secondary, Actors		Manager, Employee, Company
Trigger		Manager has hired an employee
Description	Step	Action
	1.	Manager hires a new Employee
	2.	Manager gives new Employee an Employee ID
	3.	Manager then adds Employee ID to the database of Emps.
Variations	Step	Branching Action
	1.	Employee may be fired/Manager wishes to remove them from the database
Priority		Top
Performance		5 minutes
Due Date		Release Version 1.0

5.2. Use Case Description - Add Customer

USE CASE		Add Customer
Goal in Context		Manager to add customer from loyalty card system
Scope & Level		Company
Preconditions		Customer asks to be Added
Success End Conditions		Manager adds the customer
Failed End Conditions		Customer is not added
Primary, Secondary, Actors		Manager, Customer, Company
Trigger		Customer expresses wish to be added to the loyalty card system
Description	Step	Action
	1.	Customer receives email about Loyalty card system
	2.	Customer wishes to be added
	3.	Manager receives customers wishes
	4.	Manager adds as per customers wishes
Variations	Step	Branching Action
	1.	Customer may wish to be removed from the loyalty card system.
	2.	Customer may not wish to be added
Priority		Top
Performance		5 minutes
Due Date		Release 1.0

5.3. Use Case Description - Make Purchases

USE CASE	Make Purchases
Goal in Context	Customer is making an in-store purchase
Scope and Level	Company
Preconditions	Customer has sufficient funds for goods
Success End Conditions	Customer has goods, we have money for the goods
Failure End Conditions	Customer does not have goods, we do not have money
Primary, Secondary Actors	Customer
Trigger	Customer walks past sensors with items

Description	Step	Action
	1.	Customer walks past sensors with items
	2.	Sensors detect items & relays information to system
	3.	Sensors detect items & relays information to system
	4.	System bills customer and subtracts relevant stock from database.
	5.	System sends message to customer with receipt
Variations	Step	Branching Action

	1.	Customer may pay with cash
	2.	Customer may pay with loyalty points
Priority		Top
Performance		Instant with electronic payment, 5 minutes with cash
Frequency		200/day
Due Date		Release 1.0

6. Non-Functional Requirements

- Usability: The end user should be able to navigate the software quickly and without any difficulty.
- Extensibility: If new features are required, they should be able to be added to the current software without
- Data Retention: Data should be safely stored by the program.

6.1. Tactics to support quality attributes

To support good usability, we decided to implement our user interface to be minimalist, with good description of what is expected of the end user at every stage of use.

For our project to be highly portable, we decided to program through Java. This allows our project to be run on any machine equipped with a Java Virtual Machine without any extra work being put in.

To allow for good extensibility we made heavy use of interfaces. If, for instance, we needed to add a new type of employee to our system, developing the UI would be quick and easy by simply implementing from our existing UI interface.

7. GUI Prototypes

Login		Transactions	
ID	[＿＿＿＿]		Add return
			Add Sale
Password	[＿＿＿＿]		View Transaction

Logged in as: Stock Employee		Logged in as: Manager	
	Order Stock		Update Customers
	Update Stock		View Stock Order Requests
	Add New Stock		Update Employees
	View Stock		

8. System Architecture

8.1 Package Diagram
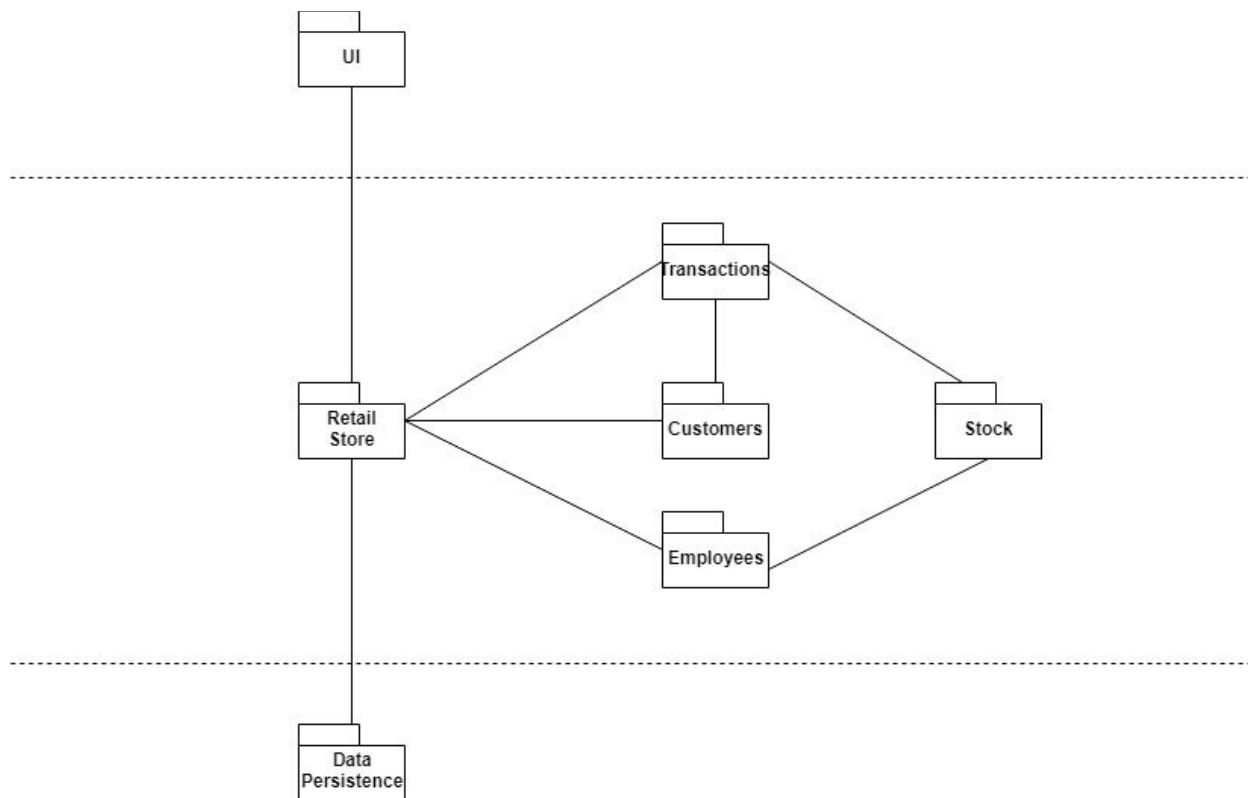
8.2 Architecture Decisions

Due to time constraints during development, we were unable to implement MVC as originally planned. For the core architectural design of this project, we attempted to utilise the Layered architectural design pattern, and for the perspective(view) layer resorted to displaying our UI through the console window. This layer employed a pipe-filter style of architecture, passing data about the current store employee from one UI class into the next and filtering the relevant data accordingly.

8.3 Design Patterns Implemented

- **Singleton Pattern:** We used the singleton design pattern on our Account class. This ensures only one account would be instantiated.
- **Facade Pattern:** The Facade pattern was used with our Store class. We used this pattern to mask the load-in of our data behind the StoreFacade class.
- **Criteria/Specification Design Pattern:** This was implemented for the Employee class. It allowed us to quickly & easily check if the functionality a user was trying to gain access to was suitable for their employee type, as well as making data persistence easier.
- **Factory Method Design Pattern:** This is how we created instances of Transactions and its subclasses, Sales and Returns.

9. Candidate Classes

Before	After
- Stock ~~Shop~~ - Employee - Manager - Stock Employee ~~Cashier~~ - Customer ~~Phone~~ - Account - Order - Voucher - Transaction - Sales - Returns ~~Warehouse~~ ~~Distributor~~	- Stock - Store - Employee - Manager - Stock Employee - Transaction - Sales - Returns - Customer - Account - Voucher - Order

9.1 Reasoning:

- **Shop:** Renamed to *Store*.
- **Cashier:** Unnecessary, the mechanics for sales are fully automated.
- **Phone:** Would not be used, so made into an attribute of the Customer class.
- **Warehouse:** Unnecessary, as most stock would be handled by store.
- **Distributor:** Unnecessary, as we decided to order products directly from the company.

9.2 Class Descriptions

- **Stock:** Renamed to *StockItem*, this object tracks a type of product's ID, quantity, price and use-by date.
- **Store:** This class launches the program, reads the information in from the files (through the Facade), and calls the data persistence class on exit.
- **Employee:** Eventually changed to implement Criteria Design pattern rather than a single employee interface, this object holds the information about each employee.
- **Transactions:** The parent class for both transaction types (sales and returns), this object holds all the information about the transactions, and is constructed via the *TransactionsFactory* method.
- **Customer:** Holds all information about each customer.
- **Account:** Holds & updates bank balance, and implements singleton design pattern.
- **Voucher:** Holds information about vouchers awarded to customers.
- **Order:** Holds information regarding the ordering of new stock.

10. Analysis

10.1. Communication Diagram
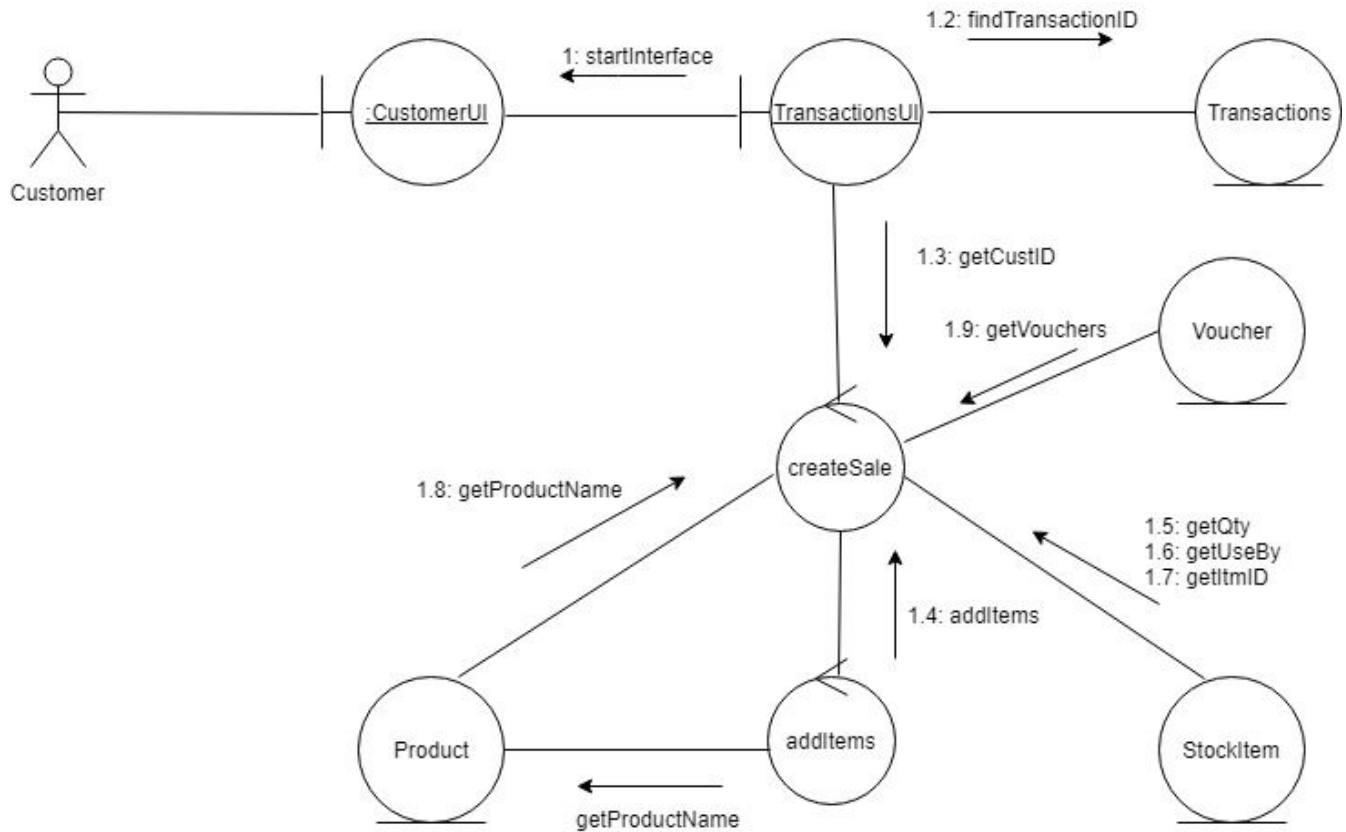
10.2. Entity-Relationship Diagram
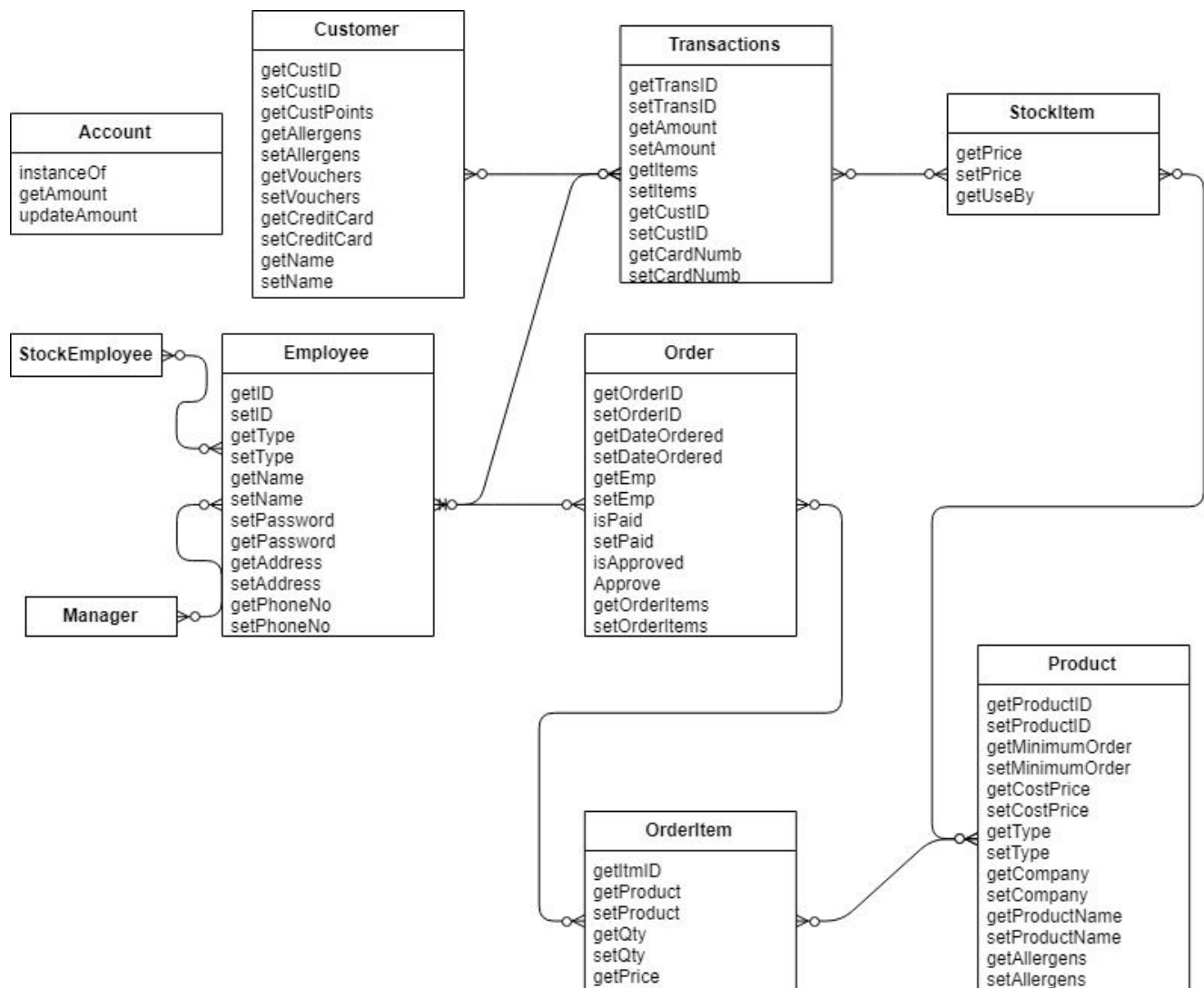
Final E-R diagram
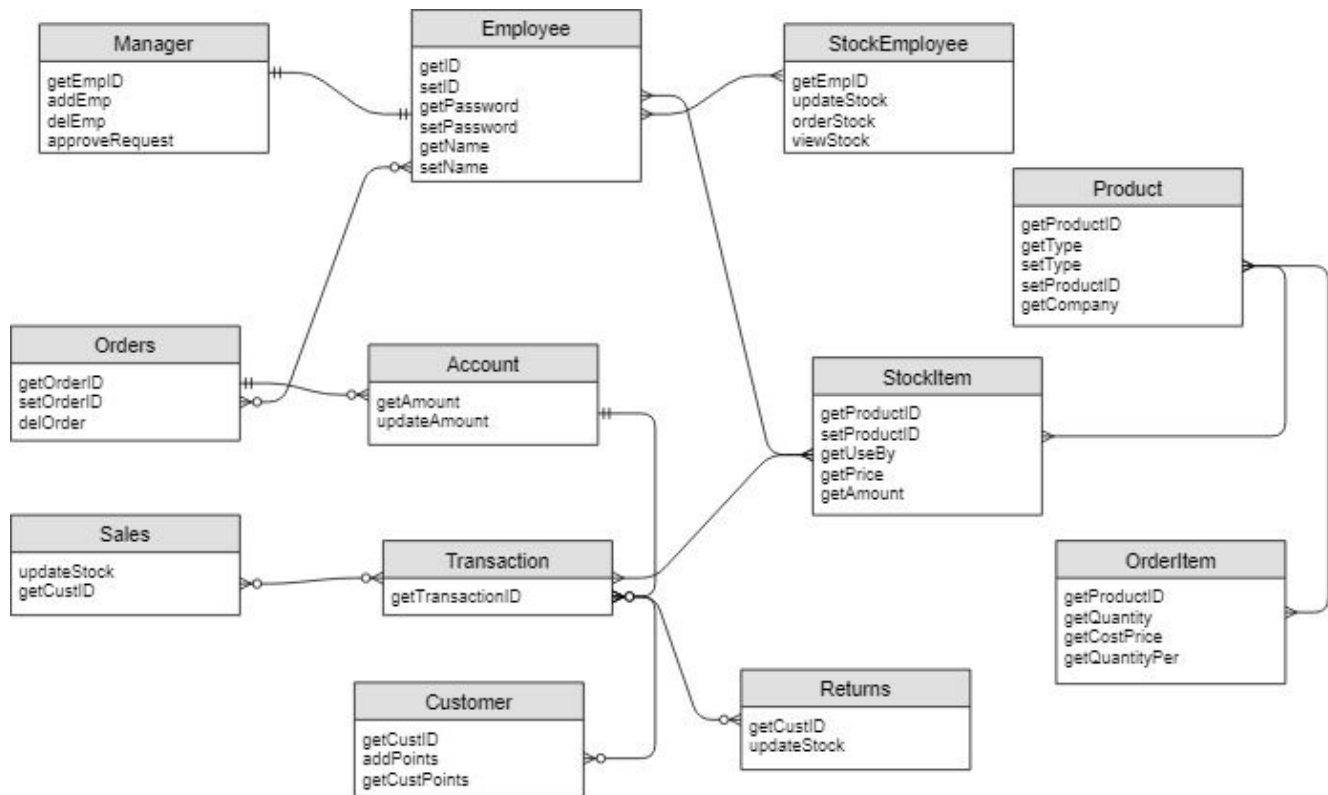
Old diagram from Week 5 - Analysis

10.3. Analysis Class Model
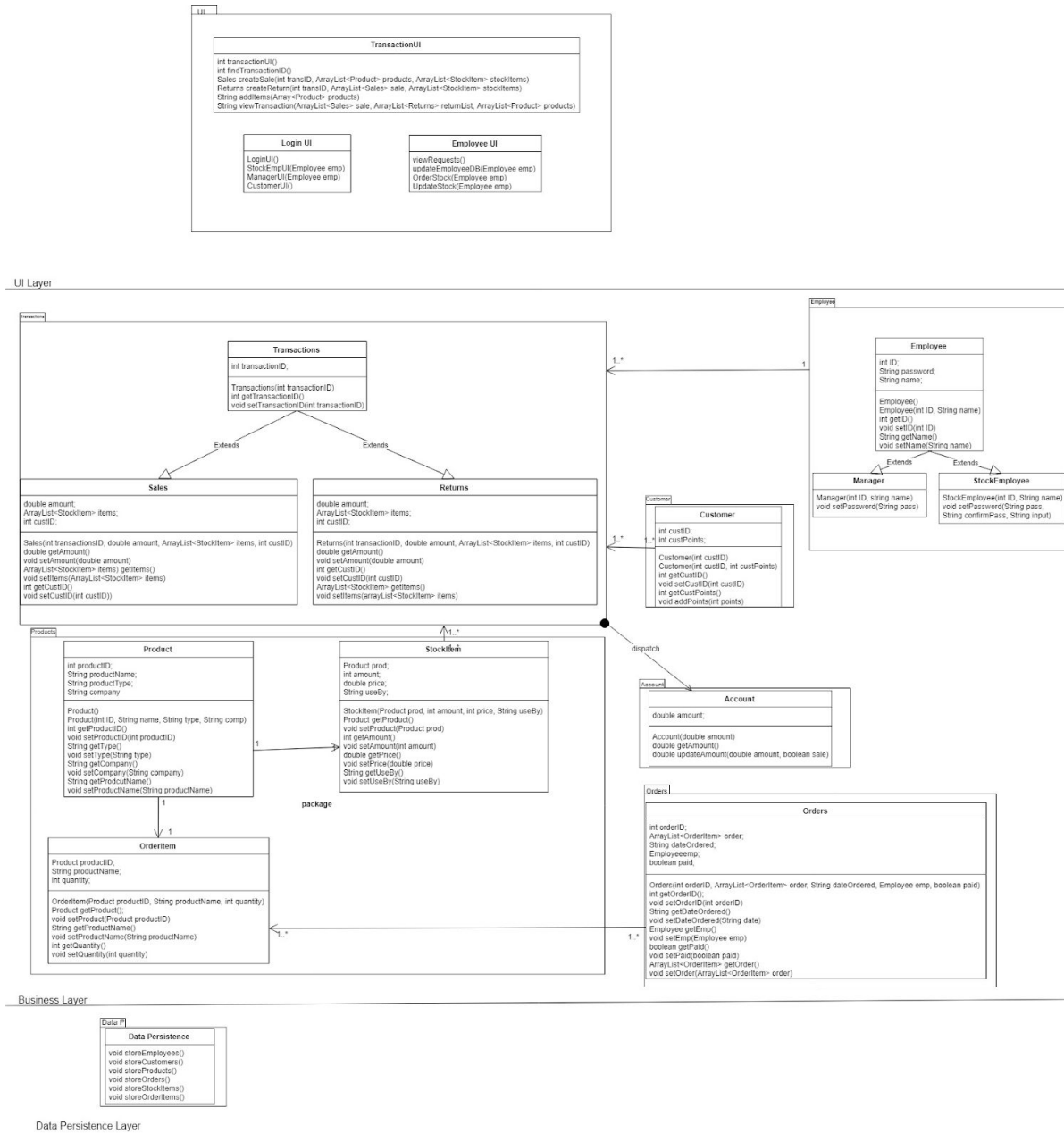
Original diagram from Week 5 - Analysis

11. Lines of Code

Package	Method/File	Name	Lines of Code
customer		Alex	79
dataPersistence	run()	Alex	40
	employeesToFile()	Michael	32
	customersToFile()	Alex	46
	productsToFile()	Alex	33
	ordersToFile()	Michael	34
	stockItemsToFile()	Alex	21
	orderItemsToFile()	Michael	21
	salesToFile()	Michael	28
	returnsToFile()	Michael	27
	accountToFile()	Alex	9
employee	Criteria	Alex	9
	CriteriaManager	Alex	20
	CriteriaStockEmployee	Alex	20
	Employee	Harry	70
retailStore	Account	Alex	35
	Store	Michael	17
(StoreFacade)	runs()	Alex	22
	readEmployees()	Michael	30
	readCustomers()	Michael	40

	readProducts()	Michael	31
	readStockItems()	Michael	20
	readOrderItems()	Michael	21
	readSales()	Alex	26
	readReturns()	Alex	21
	readAccount()	Alex	9
	checkStockLevels	Michael	62
stock	Order	Harry	85
	OrderItem	Michael	42
	Product	Michael	82
	StockItem	Michael	27
transactions	Returns	Alex	12
	Sales	Alex	12
	Transactions	Alex	67
	TransactionsFactory	Alex	24
	Vouchers	Alex	29
UI	customerUI	Alex	11
	LoginUI	Michael	109
(ManagerUI)	startInterface()	Michael	61
	updateCustomers()	Michael	102
	viewRequests()	Michael	57
	updateEmployees()	Alex	127
(StockEmployeeUI)	startInterface()	Michael	56
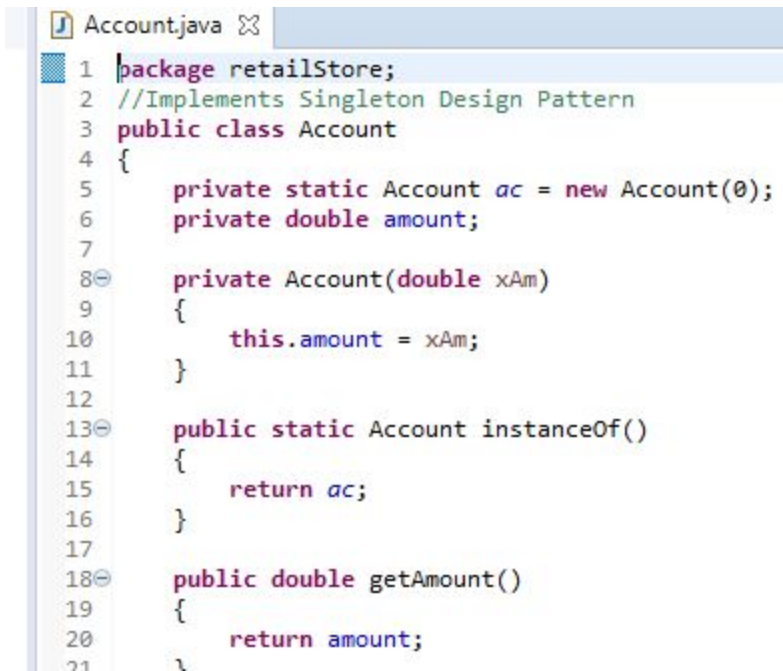
	orderStock()	Michael	48
	updateStock()	Michael	73
	viewStock()	Michael	16
	viewProducts()	Michael	15
UI	StoreUI	Michael	44
	TransactionsUI	Alex	348
	UI	Michael	7

Total Lines of Code

Names	Lines of Code
Alex Hutt	1020
Michael Meskell	1102
Harry deLongue	155
Total	2277

12. Code Fragments

12.1 Singleton Design Pattern

```java
Account.java ⊠
 1  package retailStore;
 2  //Implements Singleton Design Pattern
 3  public class Account
 4  {
 5      private static Account ac = new Account(0);
 6      private double amount;
 7
 8      private Account(double xAm)
 9      {
10          this.amount = xAm;
11      }
12
13      public static Account instanceOf()
14      {
15          return ac;
16      }
17
18      public double getAmount()
19      {
20          return amount;
21      }
```

We decided the best way to implement our account class was with the Singleton design pattern, to ensure only a single account could be instantiated.

12.2 Factory Method Design Pattern

```java
1  package transactions;
2
3  import java.util.List;
4
5
6  //Implements factory method for transactions
7  //Author: Alex
8  public class TransactionsFactory {
9
10     public Transactions getTransactions(String transactionsType, int transID, double amount, List<StockItem> items, int custID, String cardNumb)
11         transactionsType = transactionsType.toLowerCase();
12         if (transactionsType == null)   {
13             return null;
14         }
15         else if (transactionsType == "sales")   {
16             return new Sales(transID, amount, items, custID, cardNumb);
17         }
18         else if (transactionsType == "returns") {
19             return new Returns(transID, amount, items, custID, cardNumb);
20         }
21         else
22             return null;
```

The factory design pattern, implemented for the *Transactions* class and its subclasses, *Sales* and *Returns*.

12.3. Facade Design Pattern

```java
public class Store {

    public static void main(String[] args)

        // For testing purposes
        System.out.println("- Login info -\
        System.out.println("For testing man

        StoreFacade f = new StoreFacade();
        f.run();
    }
}
```

```java
//Runs facade
public void run() throws IOException, ParseException {

    readEmployees();
    readCustomers();
    readProducts();
    readStockItems();
    readOrderItems();
    readOrders();
    readSales();
    readReturns();
    readAccount();

    checkStockLevels();

    //Launches UI
    UI store = new StoreUI();
    store.startInterface();

    //Reads lists into files
    DataPersistence d = new DataPersistence();
    d.run();
}
```

Excerpts from the *Store* class and the *StoreFacade* class, showing the Facade pattern implementation.

26

12.4 Criteria Design Pattern

```java
6  public class CriteriaManager implements Criteria    {
7
8⊖     @Override
9      public List<Employee> meetCriteria(List<Employee> employees) {
0          List<Employee> managers = new ArrayList<>();
1
2          for (Employee emp : employees)  {
3              if (emp.getType().toLowerCase() == "manager")   {
4                  managers.add(emp);
5              }
6          }
7          return managers;
8      }
9
```

The criteria design pattern implemented in for the employee class. Not shown - the *CriteriaStockEmployee* class, which is used to check if the current employee is a stock employee.

13. Additional Value
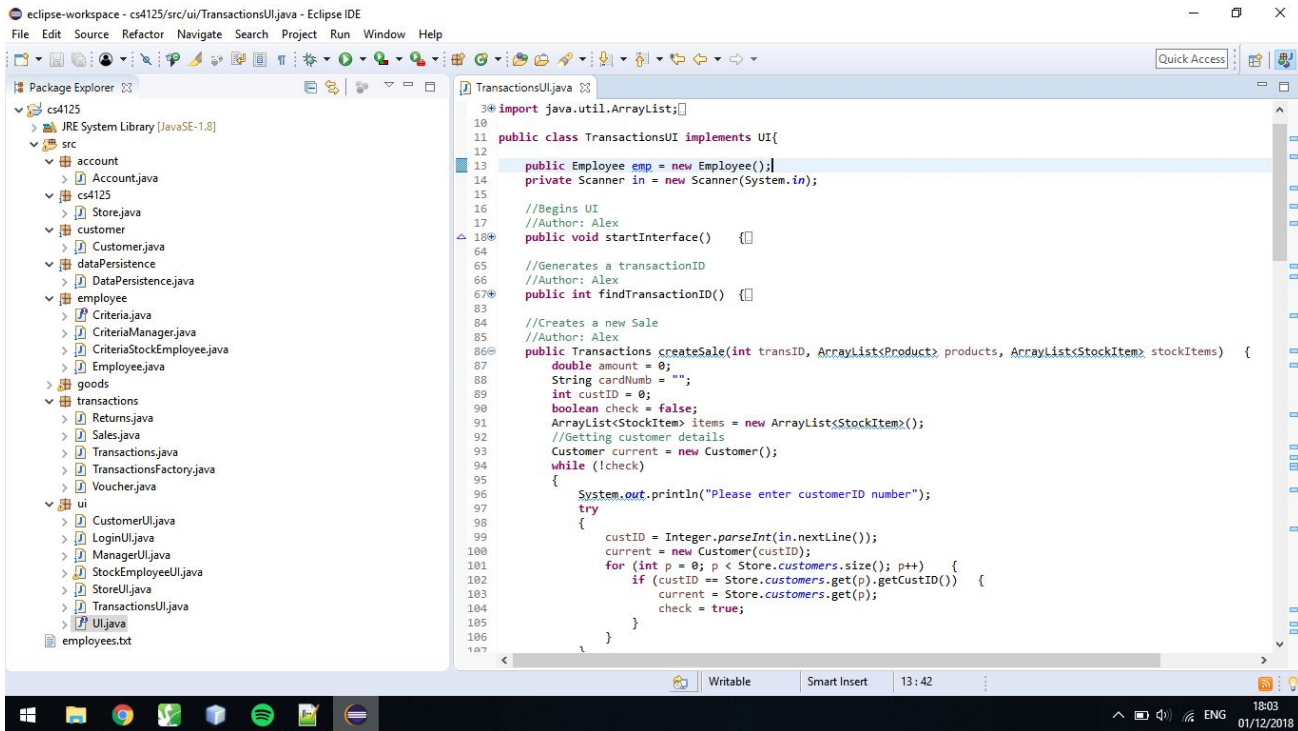
Prior to Sonar Lint
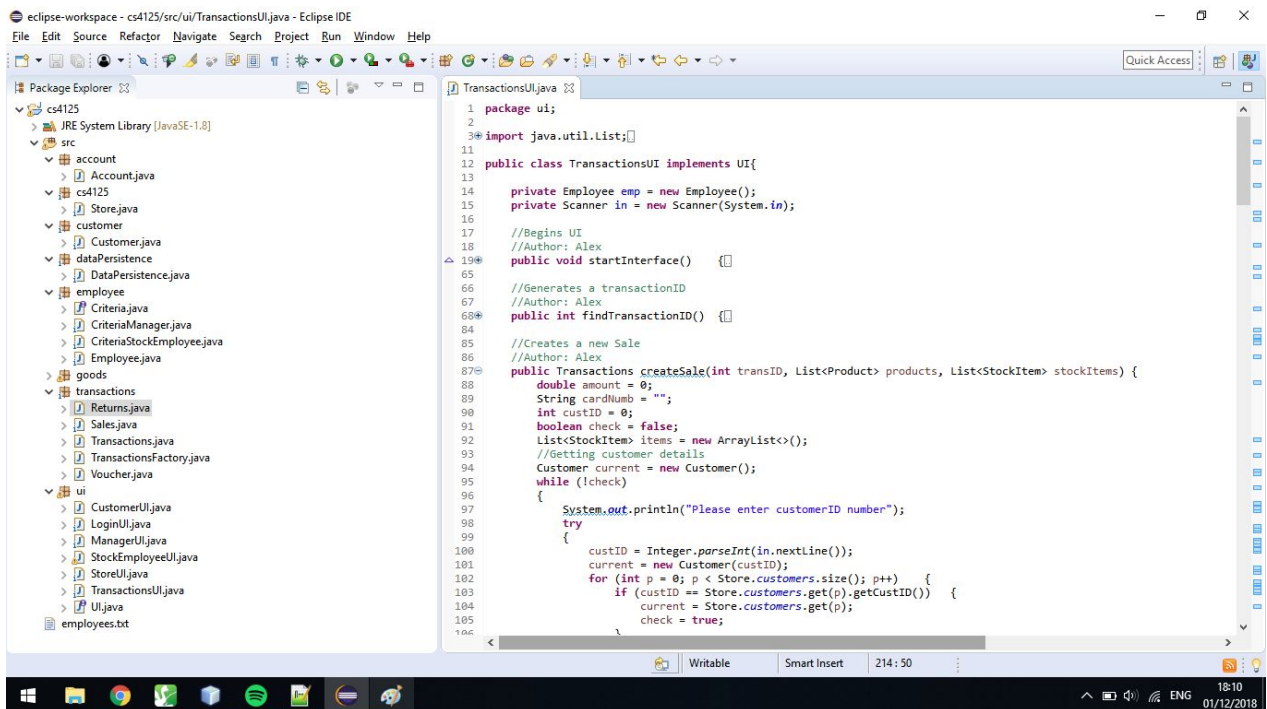
After Sonar Lint

```
}
BufferedWriter empWriter = new BufferedWriter(new FileWriter(emp));
empWriter.write(eachLine);
empWriter.close();
}
try (BufferedWriter empWriter = new BufferedWriter(new FileWriter(emp));)
    empWriter.write(eachLine);
}
```

Example of improved code from SonarLint

We decided to use the SonarLint plugin for Eclipse to inspect our code for bugs and code smells. It was especially useful in the example above when implementing the BufferedWriters in our Data Persistence class.

| Branch: master ▾	New pull request			Create new file	Upload files	Find file	Clone or download ▾
🔲 michaelmeskell Add files via upload ...						Latest commit 72fcc7a 16 hours ago	
📁 cs4125			Add files via upload				16 hours ago
📄 README.md			Initial commit				2 months ago
📄 TransactionsFactory.java			Add files via upload				12 days ago
📄 TransactionsUI.java			Add files via upload				12 days ago
📄 cs4125.rar			Add files via upload				13 days ago
📄 employees.txt			Add files via upload				12 days ago
📄 orderItems.txt			Add files via upload				12 days ago
📄 products.txt			Add files via upload				12 days ago
📄 stockItems.txt			Add files via upload				12 days ago

As it is the industry standard, we decided on Github as our repository for this project.

14. Testing

For our testing process, we decided to have all group members act as manual testers for both their own code and each others. This allowed for flexibility in workflow as well as constant testing of the project on both the full scale program and individual methods or code blocks. This allowed testing to be as localised to the issue as it needed to be.
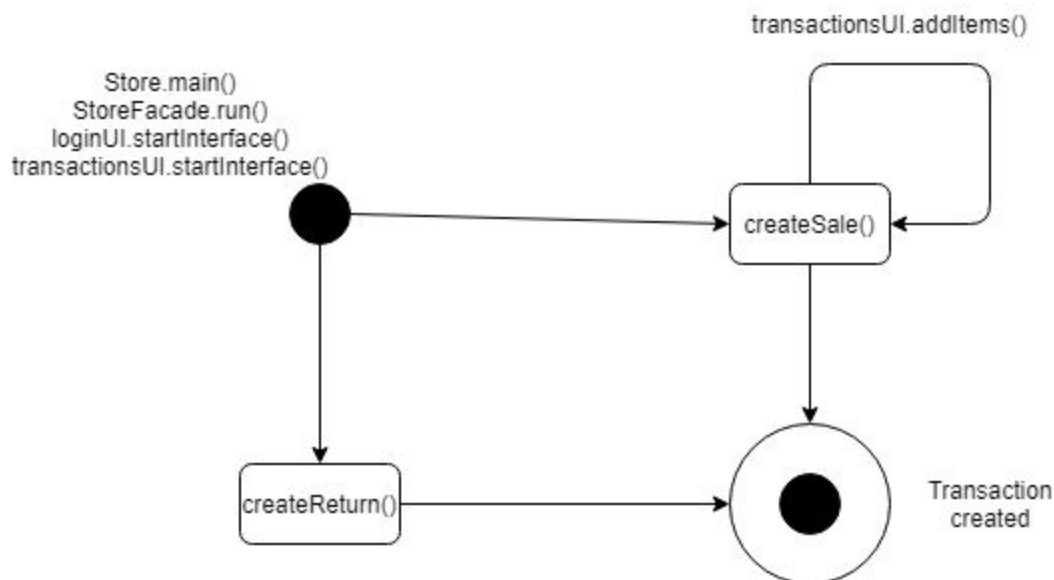
15. Design and Architecture Recovery

15.1. Architectural Diagram
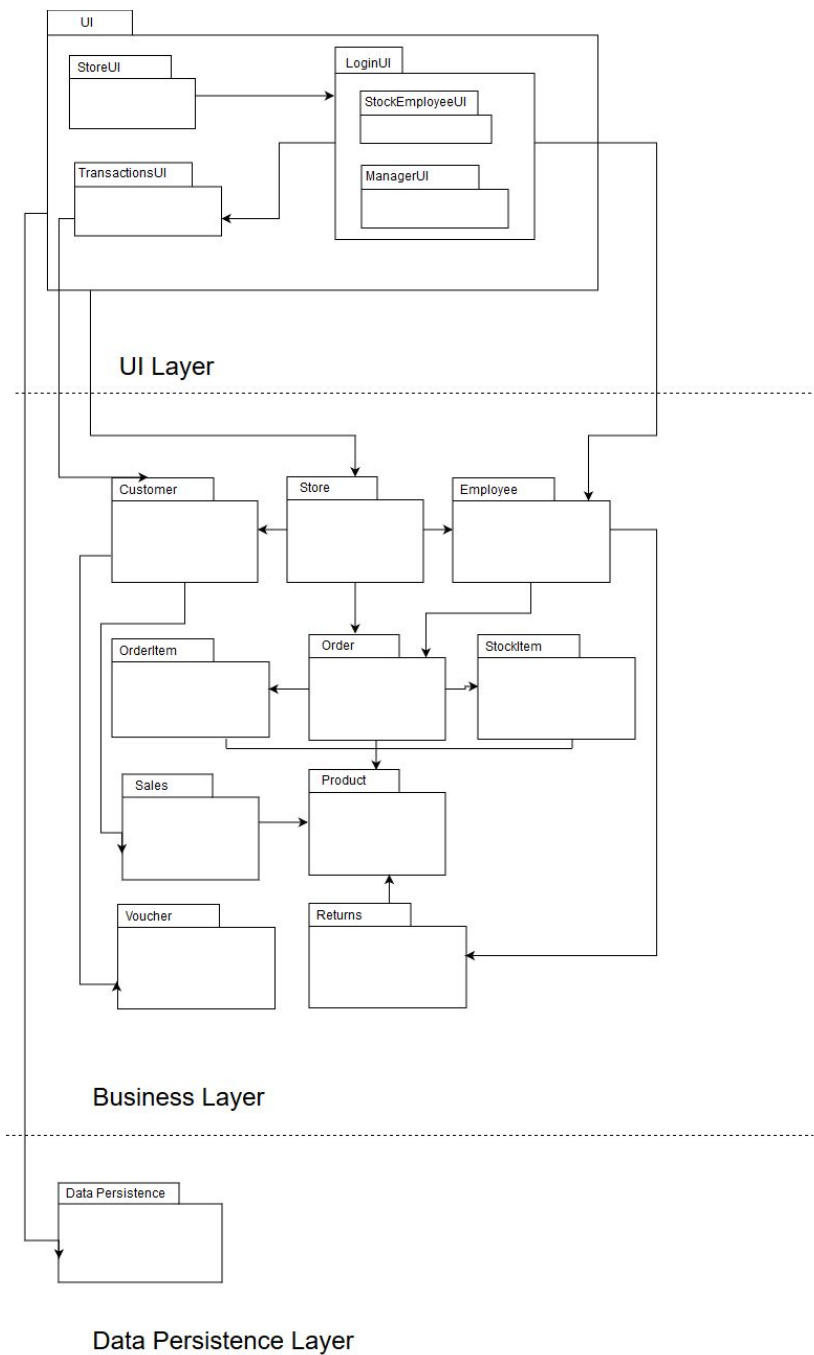
15.2. Design-Time Class Diagram

15.3. State Chart

State chart for the creation of a transaction by the user

16. Critique and Contrast: Analysis Sketches vs Blueprints

Going into the project we were unsure to what extent the final implementation would differ from our original design. Once we began implementation, it became clear that our original design was not at all detailed enough or deep enough to satisfy the goals of the project.

The first major difference is between the analysis-time class diagram and the design-time class diagram. The first major difference is in our UI layer. After implementing the UI interface in full, there is a stark difference in the two layers, with design-time being both more extensive and better designed. The package layout in our final design is also far better, with the singular packages for Orders and Account being replaced with more cohesive packages in Stock and

retailStore respectively. Finally, our design-time diagram containts better representation of the design patterns we used, specifically the StoreFacade and TransactionsFactory classes being implemented.

The major difference in our two Entity-relationship diagrams is the final implementation of the code, including the restructuring of the Transactions, Orders and Employees classes.

17. References

Websites: draw.io for sketch/diagram creation
IDE: Java Eclipse