CocaCola supply chain management system

Features A & B: Supply chain management with Point of sale system

Tom Lynch (540621) and Anthony Miller (636550)

cocacolasupplychain.herokuapp.com/CocaColaSupplyChain

Login details:

Distribution Centers
username: melbdc, password: a
username: syddc, password: a
username: brisdc, password: a
username: perthdc, password: a

Retailers
username: colescoll, password: a
username: colesfitz, password: a
username: colessyd, password: a
username: colesperth, password: a

Bottling Plants
username: melbbot, password: a
username: sydbot, password: a

Factory
username: hq, password: a

Introduction

This document describes the architecture and design decisions made in the implementation of both the internal and client facing portions of the supply chain management software produced for Coca Cola.

Feature A encapsulates Coca Cola's point of sales systems, which allows distributors to manage their inventory, ship pallets to buyers and manage their accounts. It also provides a portal for retail buyers to purchase pallets of either vanilla, zero or regular coke cans.

Feature B covers Coca Cola's internal supply chain mechanisms, allowing Factories, Bottling Plants (Bottlers) and Distribution Centers to request, produce and ship product.

Design decisions - Feature A

N-tier architecture

Abstraction and encapsulation is one of the core principles of software design. It allows for modules of code to communicate without having to know exactly the other modules are behaving inside. Each module (in this case in a layered 3 tier architecture) only needs to know about the layers above and/or below it.

The *presentation layer* facilitates user control of the system, where the user inputs data to the business domain layer and is returned the relevant information.

The *domain layer* controls the business logic and behaviour of the system, taking a request from the user, querying a persistent database for the appropriate data, transforming it into easily digestible and relevant knowledge for the user.

Data source layer maintains a rigid structure of business entities and the relationships between them. These are not necessarily structured to reflect the domain on which they are based, but for efficiency and integrity.

These layers are further decoupled by a service layer between the presentation and domain layers, and data mappers between the domain and data layers.

Service Layer

Service layer lies between the domain layer and the presentation layer. And aggregates what data is displayed to whom and rules by which they can interact with the system.

This can be implemented through templating languages which are widely used on the web, such as ejs or in CocaCola's case jsp. It allows for a variety of different information to be displayed to a range of users with differing use cases from one standardised template.

This is important for the point of sales system, as the use cases are diametrically opposed, as there is a give and take transaction between two parties, with one spending money and gaining product and the other making money and removing product from the inventory.

Despite the opposite use cases they share the same interface with different values rendered.

Domain model

A domain model is a way of representing the internal state of the database in the domain layer, as opposed to calling the database each time data is needed to be presented to the user which results in repeated calls, which is the opposite of reuse. This pattern is not constrained by the rigidity of relational databases and can be implemented in a way that better represents the desired business logic.

For example, in the CocaCola Supply chain management system transactions are kept in an account book so they can be aggregated in the domain layer as they would on a balance sheet. When in reality the transactions are linked directly to distribution centers and buyers through foreign keys.

Data mapper

A data mapper establishes a facade between the domain layer and the data source layer decoupling the two even further.

This also ensures the single responsibility (**S**OLID) principle where the domain object is merely responsible for representing a structural element in the data layer and the mapper is responsible for updating both the data and domain representations of the object, based on the desired behaviour.

All classes that represent a structural element of the data layer have mappers in the CocaCola Supply chain network.

However, in the CocaCola Supply chain system rather than calling the database directly, the mappers first call the identity map which will call the database if the value has not been called before.

Foreign key mapping

Aims to represent the underlying structural relations of the data layer by linking a table to its relatives. This is essentially a way of representing a one-to-many table relation, where the many relatives store the key of their shared relative.
This has its drawbacks as it strongly couples the domain and data source layers, however it is a simple algorithmic process that is mechanical and relatively easy to understand.

This pattern is used to map the distribution centers to the head Coke HQ offices, so that they can monitor the revenue of the distribution centers, thus combining them will give the total revenue

Unit of work (UOW)

However, all this decoupling has some drawbacks, especially under conditions where networks are relatively slow or unreliable. Even under circumstances where connections are fast, a db connection must be created with each call which is cumbersome in comparison to other operations in the system.

Unit of work demonstrates an elegant solution to this problem, where you are relying on a physical connection that is out of your control. It does this through keeping a list of items that have been modified since being read from the database, making sure this change is reflected in the data layer eventually. It keeps a list of new objects that have flowed down through the layers and eventually need to be added to the database. UOW also keeps track of items that the user has selected for deletion and are to be removed from the database.

Some implementations include a clean list of items that haven’t been modified since being removed from the db. However, this seems somewhat redundant as it can be assumed that any item that is not dirty, new or to be deleted is an accurate portrayal of its structure in the data layer.

Identity map

Identity map acts as a cache making sure that the data source layer is only called when necessary. This not only improves performance by not having to make external calls to a potentially busy database, but also ensures the integrity of the database. This integrity is ensured by making sure that no two objects in the domain represent the same object in the data source layer. If somehow two items in the domain layer relate to the same table in the database, then if one of these objects is changed there will arise a dilemma as to which is the real representation of the table.

This is used for all data pulled from the database into the domain layer in CocaCola's supply chain.

Lazy load (lazy initialisation)

Much of the design decisions in this project have been implemented with the goal of representing a structured relational database in a mode that represents domain specific behaviour. However, these objects need not be loaded in completely, and portions of the object need only be loaded when needed.

This includes a minor adjustment to the class' getter methods, whereby the method checks if the local variable has been initialised, and if it has it merely returns it as usual. However, if it has not been initialised it queries the db to initialise and return the variable.

This means that many of the objects in the domain are only partially represented, however it is efficient as is only calls the database when absolutely necessary. Yet there may be the issue of the db being called more than needed to, however this is necessary in Coke's case, for instance CocaCola HQ doesn't care how much inventory all distribution centers have on hold, when calculating their annual revenue.

Identity field

One of the most simple ways connect an object in the domain class is to have an id variable that is identical to its id in the data table. It is simple but can be done in a number of ways.

We decided to go with auto generation as the domain does not need to know the value of the key before it is added because the transactions do not need line items, just the number of pallets purchased as the distributors only sell in quantities of pallets, so only the amount of product and not the product themselves need to be attached to the transaction tables. Thus we decided to outsource this logic to the database's software, instead of scanning the table for a unique identifier or handling the iteration ourselves.

Association table mapping

As programming allows for rich data structures like maps and collections, which represent sets and lists in an intuitive way, there can arise the problem of circular dependencies. For example with the many-to-many relationship between distribution centers and retailers in the data layer, would result in a circular dependency with buyers having a list of distribution centers that they can buy from, while distribution centers would have a list of buyers they sell to… which have a list of distributors they buy from which have a list of buyers they sell to and so on…

We mitigated this by having an intermediate mapper that maps a single entity to the foreign keys of its many associates, while its associates conversely have a pointer to its many relations of the object in question.

Embedded value

One-to-one relations make can make sense intuitively and use tables to in essence represent some kind of data type. For example in the transaction class there was thought of having the transaction compartmentalised having a one-to-one mapping with a contract (a table that specified the buyer and the seller and the amount monetary agreed to). However, we decided against this and embedded this one-to-one relation in the transaction table.

This was decided because the use case for transactions would primarily be auditing and business tracking, thus these transactions would be displayed primarily in row/column form (akin to a spreadsheet), thus it was deemed redundant to include a foreign key for this one-to-one relationship.

Concrete class inheritance

All users of the system will be transacting. Currently in *feature A* these transactions are commercial in nature, where goods are exchanged for money. However in feature B, many internal transactions will be taking place in CocaCola's vertically integrated supply chain, where a mega syrup factory will ship syrup to bottlers who will create pallets that will be sent to distribution centers to be finally sold for profit.

Although all parties are transactors, the portal will not manage the buyer's inventory for a variety of reasons, and thus the buyers and distributes diverge at this point, keeping the transactional logic consistent across all actors but separating concerns.

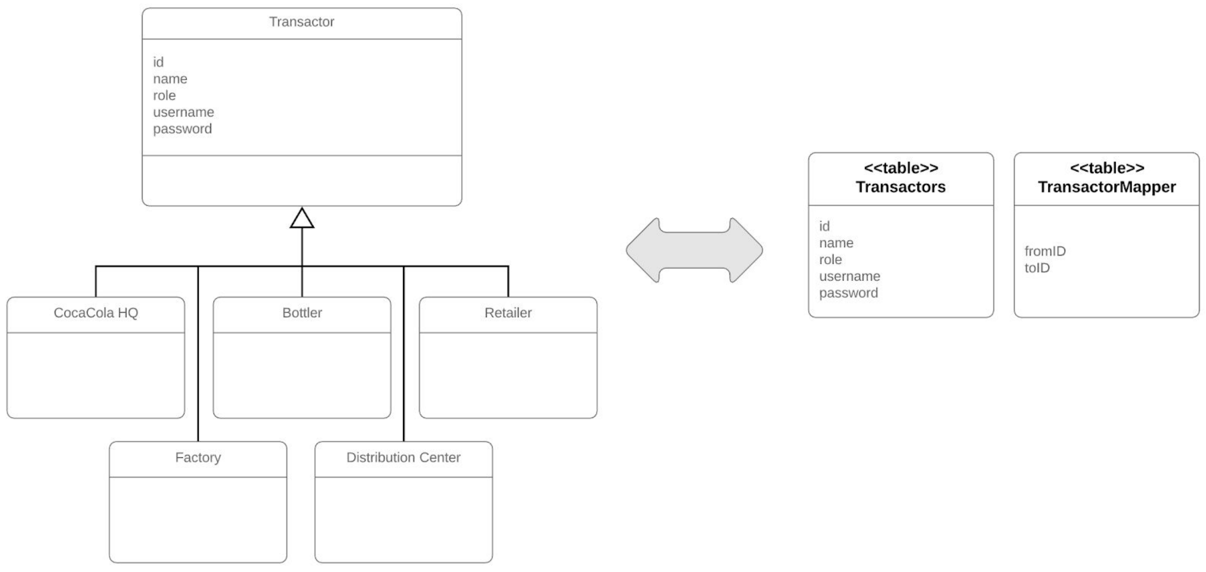
Abstraction has been an important component of feature A, due to an entire supply chain being abstracted to a PalletFactory class, that is all the distribution centers care about is receiving and selling pallets, not how the pallets are created. This will allow us to extend the functionalities of the system in a straightforward manner with minimal disruption as we implement the internal supply chain logic.

Concrete class inheritance will allow us to only store real world actors in the database, while maintaining transactional logic.

Design decisions - Feature B

Single table inheritance

While object-to-relational structural design patterns were to be implemented in part 2 (feature A), we have since began using single table inheritance for our transactors. This is because we realised we could use a TransactorMapper table in the database to pair transactors, when one produces a product and sends it to the other. This means we can have all transactors in a single table, with a role field that covers what their specific role is. The separation of these roles is done in the domain layer by querying on the role type. This can be seen in the figure below.



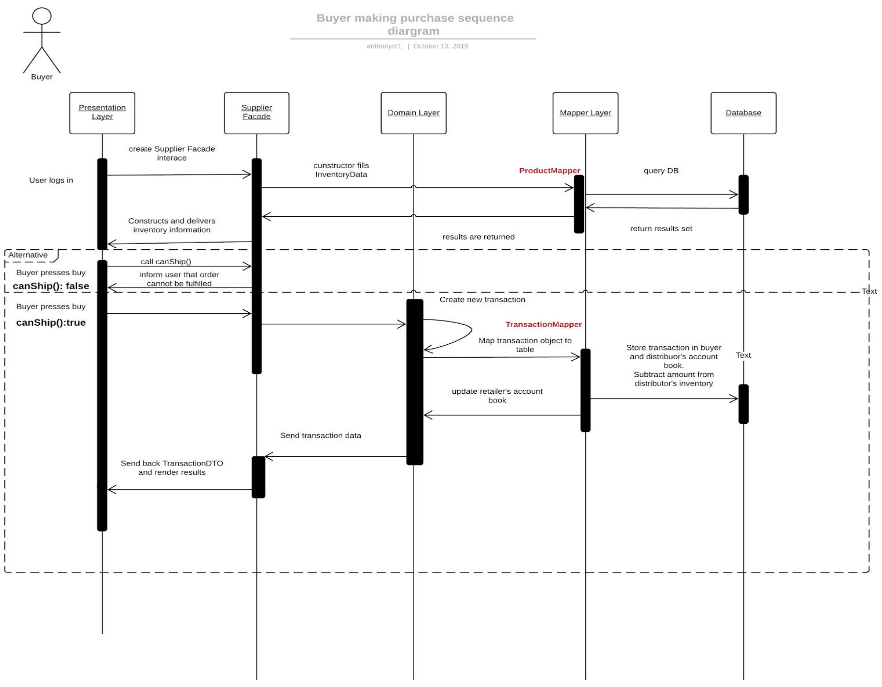
Page controller

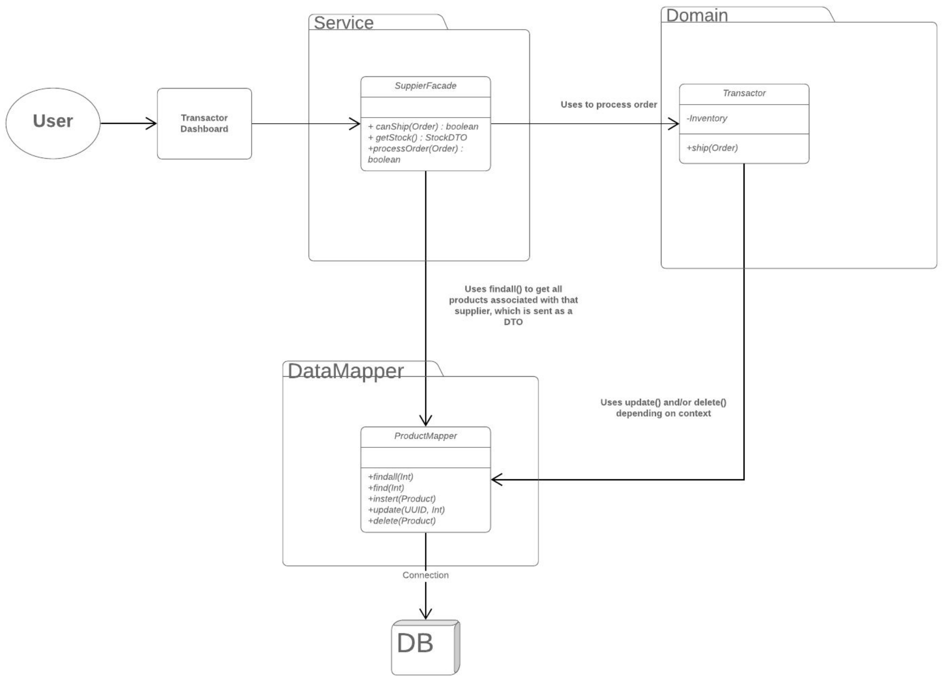
Our pages are controlled using the page controller pattern. These controllers extract data via a Post method, invoke the appropriate methods before determining which view to be displayed. Page Controller was determined to be the better option over the Front Controller pattern due to its simplicity. Front Controller may be more extensible, but we decided at this

	stage of the project, without another feature being added extensibility was not a huge concern. Instead, using page controller we do end up with some duplicate code, however since there is a small number of page controllers we deemed this not to big of an issue. This allowed greater flexibility and development speed.
Template view	JSP will be used to allow data to be embedded into HTML pages. Similar to our reasoning for choosing Page Controller over Front Controller, we have selected Template view over Transform or Two Step View for simplicity. Admittedly, maintainability is more limited, however the speed of development was to be significantly increased due to the simplicity in designing and programming the pages. Another con of this pattern is the fact that it is difficult to test. However due to the time sensitive nature of this project and the fact that formal automated testing methods are not required nor are we intending on creating them, it was decided that simplicity again trumps the portability of Transform View and the maintainability of Two Step View.
Client side session state	The session will be stored in a non-persistent cookie. This was chosen as the only need for a session in our system is authentication for all users and the coke shopping cart for retailers. Thus no persistence was needed. Cookies were chosen as they are widely used and both team members have experience with them. It is unlikely that the tool will be used in incognito mode (disables cookies in most browsers) by the staff of CocaCola.
Implicit offline-pessimistic lock	The pull based nature of the CocaCola supply chain means that rather than items being pushed through the chain, when a supply chain entity runs out of inventory they request more to be sent by the preceding entity. Thus entities will be adding to their own inventories and subtracting from their preceding entity's inventory. What's more is that multiple downstream entities can pull from the same upstream entity (eg. many retailers can buy from one distributor), and thus the POS system for that distribution center must be locked during the purchasing process, and the upstream suppliers of the DC's inventories must also be locked.
Data transfer object (DTO)	A data transfer object is helpful in decreasing the amount of data sent over the wire, at the detriment to cohesion as data that is not that closely related may be sent together. In our case, a Stock DTO is used to represent the stock of the different types of CocaCola. This data is gathered and then converted into a json. It is sent in partnership with the SupplierFacade, a remote facade that we explain next. Transactions are also sent as DTO's as they contain an order object and are rendered in bulk from the dashboard view.

Remote facade

We have used a remote facade for the supplier of the orderer. This allowed us to abstract away the details of the supplier which the receiver doesn't need for a transaction. The only details the retailer needs whether the supplier has the required stock or not. Thus, it sends the StockDTO to the receiver which allows it to gauge whether a transaction will be successful (there will be enough stock to fulfill the order). It also processes the transaction through this facade. See the diagrams below for implementation details.

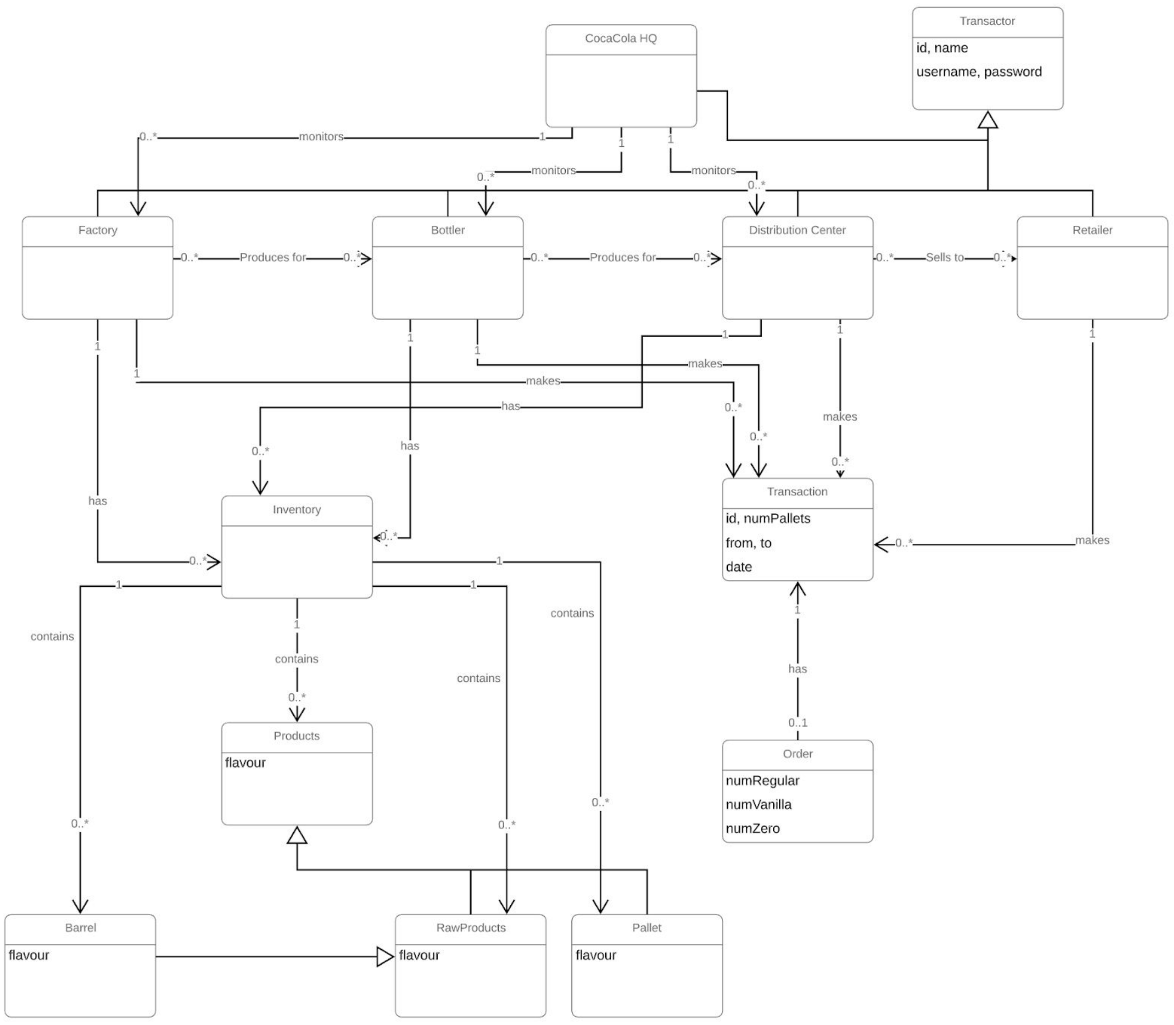

Buyer making purchase sequence diargram
anthonym1 | October 13, 2019
Buyer
Presentation Layer
Supplier Facade
Domain Layer
Mapper Layer
Database
create Supplier Facade interace
User logs in
cunstructor fills InventoryData
ProductMapper
query DB
Constructs and delivers inventory information
results are returned
return results set
Alternative
call canShip()
Buyer presses buy
inform user that order cannot be fulfilled
canShip(): false
Text
Create new transaction
Buyer presses buy
canShip():true
TransactionMapper
Map transaction object to table
Store transaction in buyer and distribuor's account book.
Subtract amount from distributor's inventory
Text
update retailer's account book
Send transaction data
Send back TransactionDTO and render results



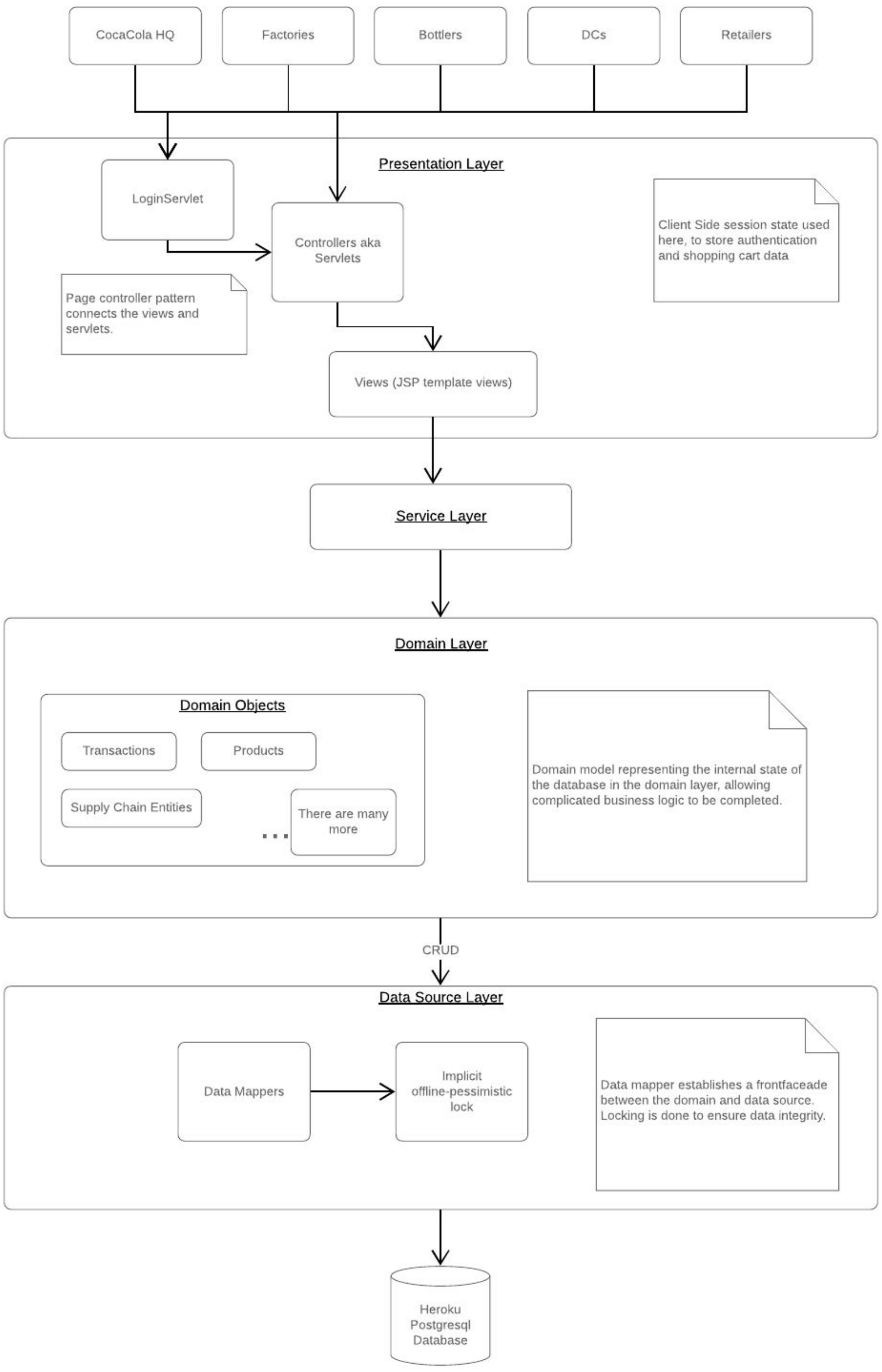
Authentication enforcer	Apache Shiro has been used for authentication, and it follows the authentication enforcer pattern, which handles auth for all users in a centralised way. Shiro is an external library, which we have decided to use as 'rolling your own' by implementing a proprietary approach is generally frowned upon due to the complexities of the process and the amount of different ways small errors can be made which an attacker can exploit this. Tried and tested methods are considerably safer in security related areas. Centralised auth will improve security and maintainability as well as reducing code reuse. COmbined with a secure pipe (mentioned later), it reduced the risk of an attacker gaining access to sensitive information.
Authorisation enforcer	Once a user is authenticated, they also must be authorised to only have access to the specific parts of the enterprise system they are meant to be able to access. Implementing this pattern with Apache Shiro, we are able to authorise users in a centralised manner, allowing us to give them fine-grained access to different functions. This ensures our different user types, from the CocaCola HQ to retailers each can only access and perform permitted actions while decreasing the amount of endpoints needed.
Intercepting validator	Again, thanks to using Shiro all requests and validated and cleansed before passing through to the domain. This ensures no malicious code will be run. Implementing this manually was an option but would require each input to be validated which could easily suffer from the forgetfulness problem, that is that the developer could easily forget to validate one of the many inputs. Having it all done by Shiro is much more convenient.

Secure pipe	As we are deploying with Heroku which uses TLS, it is not necessary to implement the secure pipe pattern ourselves - it’s already been done for us!

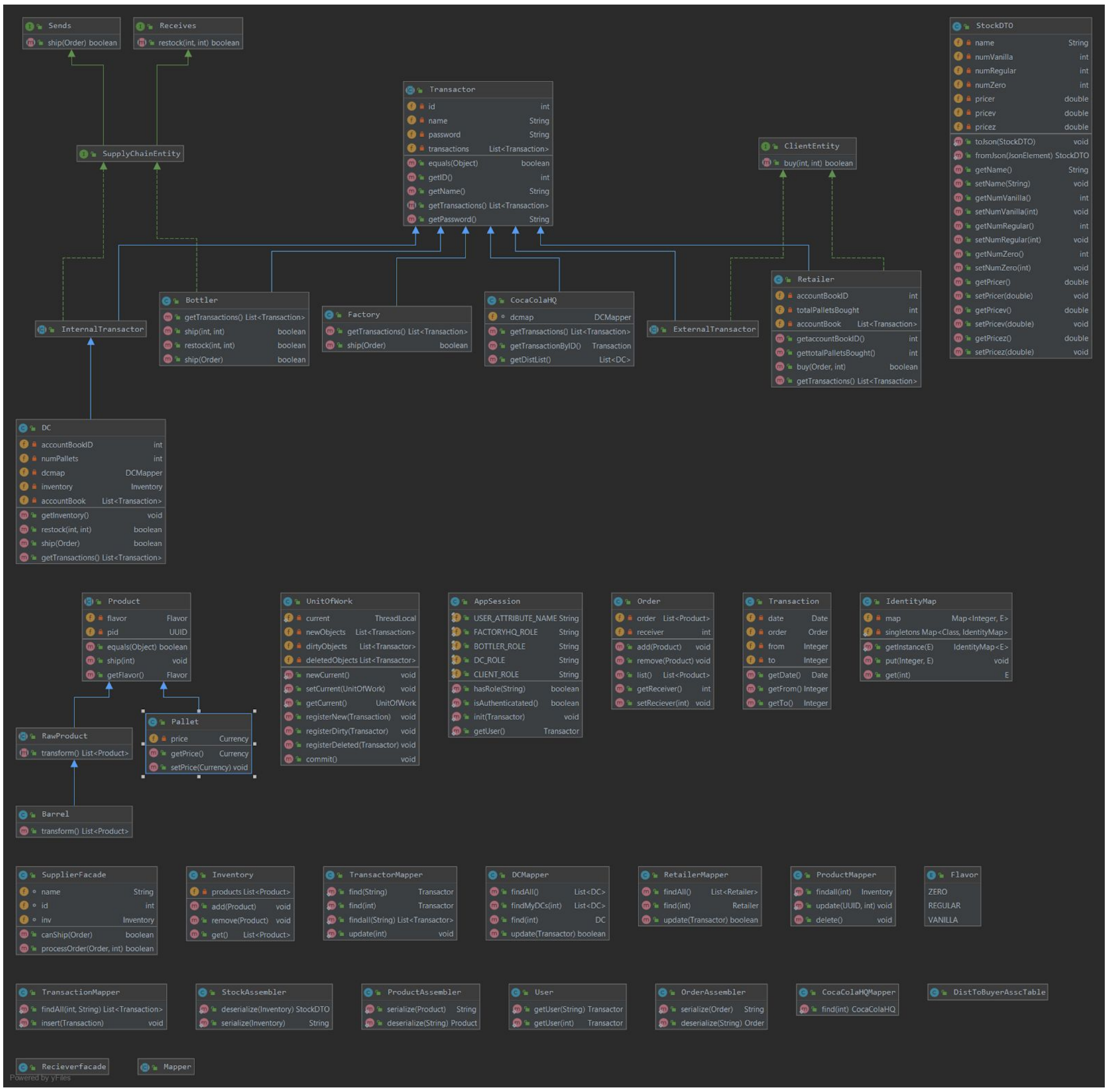
Domain Model



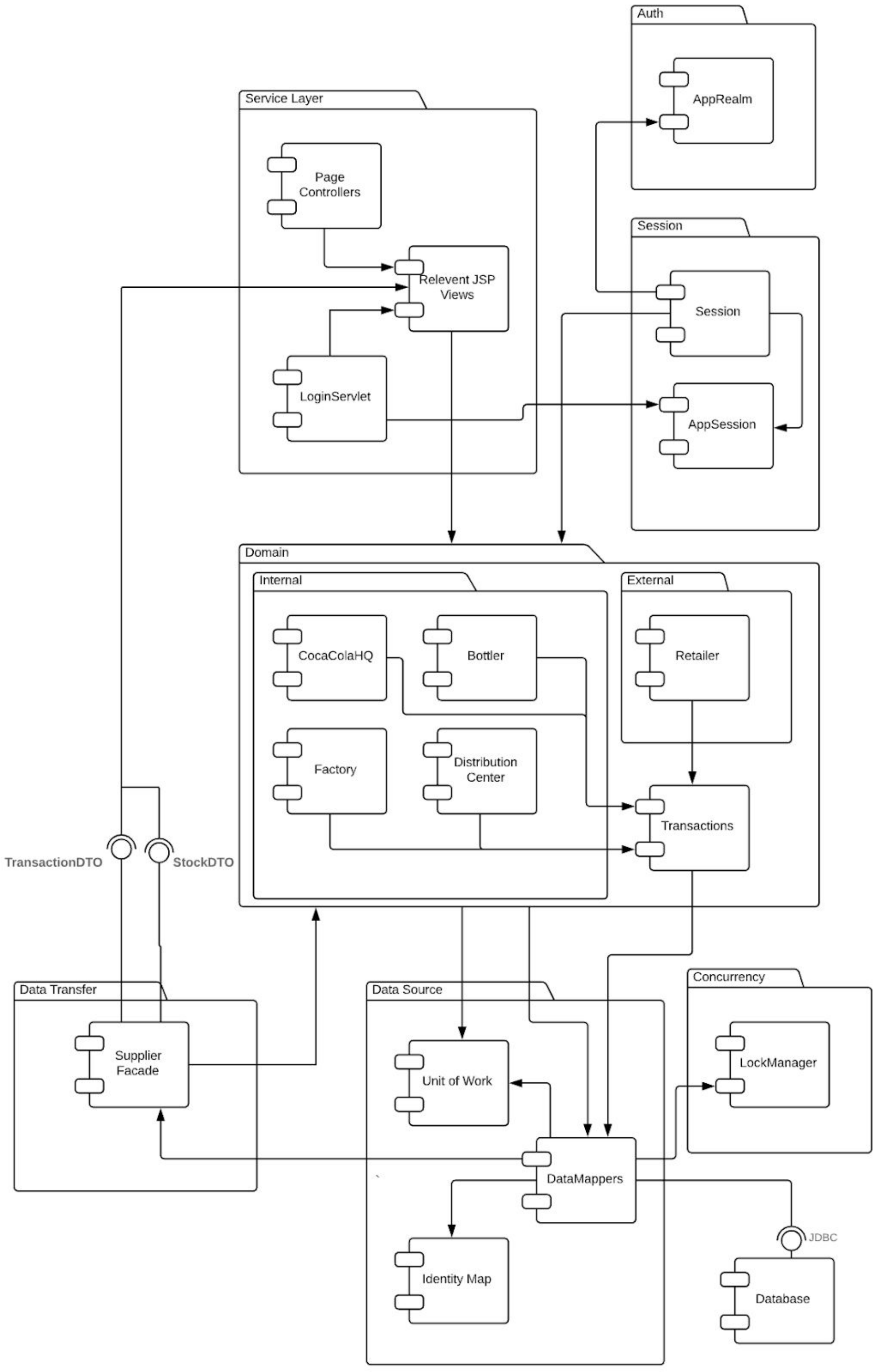
High level architecture diagram


CocaCola HQ
Factories
Bottlers
DCs
Retailers
Presentation Layer
LoginServlet
Controllers aka Servlets
Client Side session state used here, to store authentication and shopping cart data
Page controller pattern connects the views and servlets.
Views (JSP template views)
Service Layer
Domain Layer
Domain Objects
Transactions
Products
Supply Chain Entities
There are many more
Domain model representing the internal state of the database in the domain layer, allowing complicated business logic to be completed.
CRUD
Data Source Layer
Data Mappers
Implicit offline-pessimistic lock
Data mapper establishes a frontfaceade between the domain and data source. Locking is done to ensure data integrity.
Heroku Postgresql Database

Class Diagram

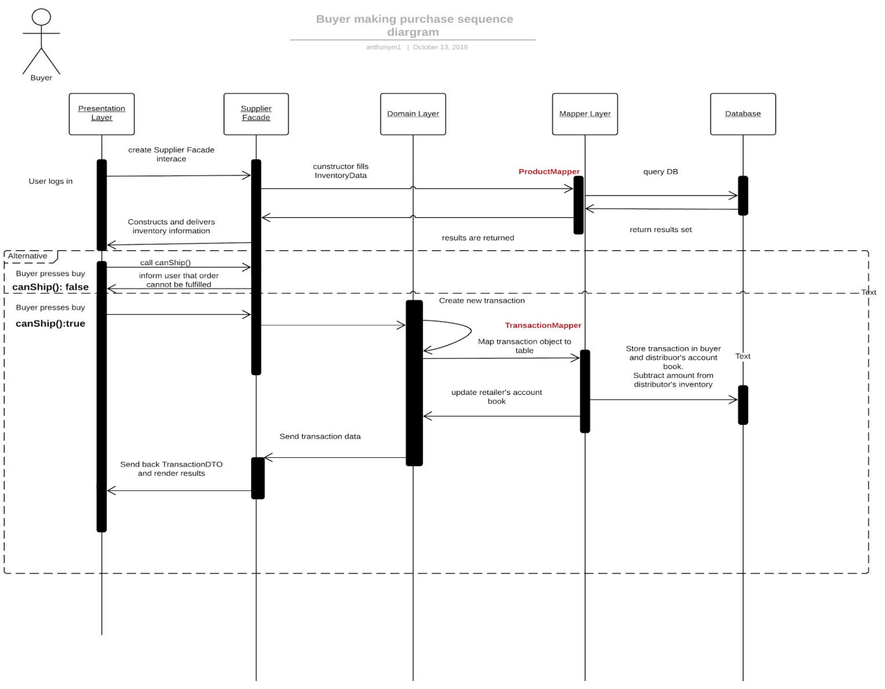


Component Diagram

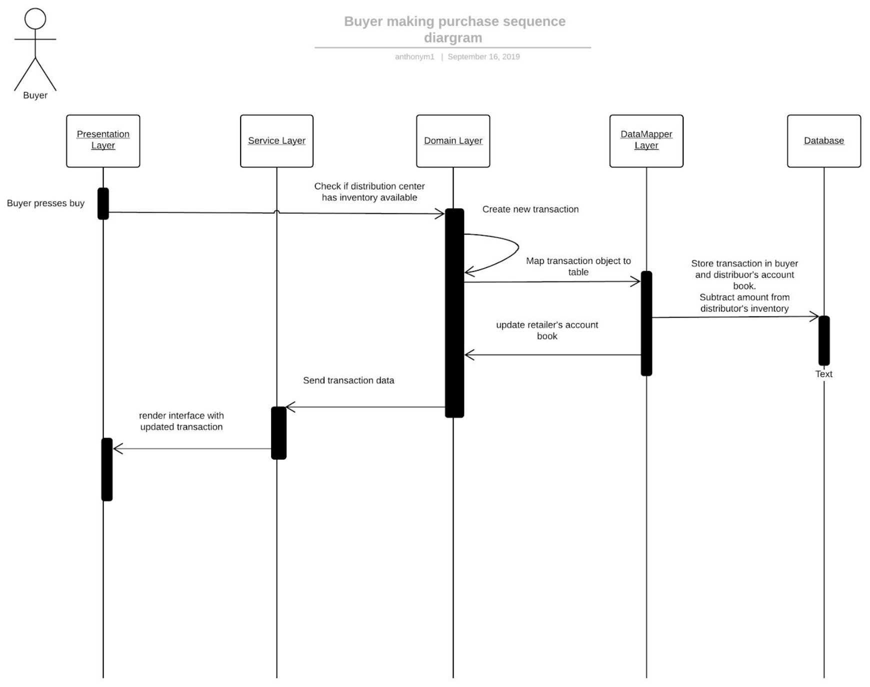


Sequence diagram

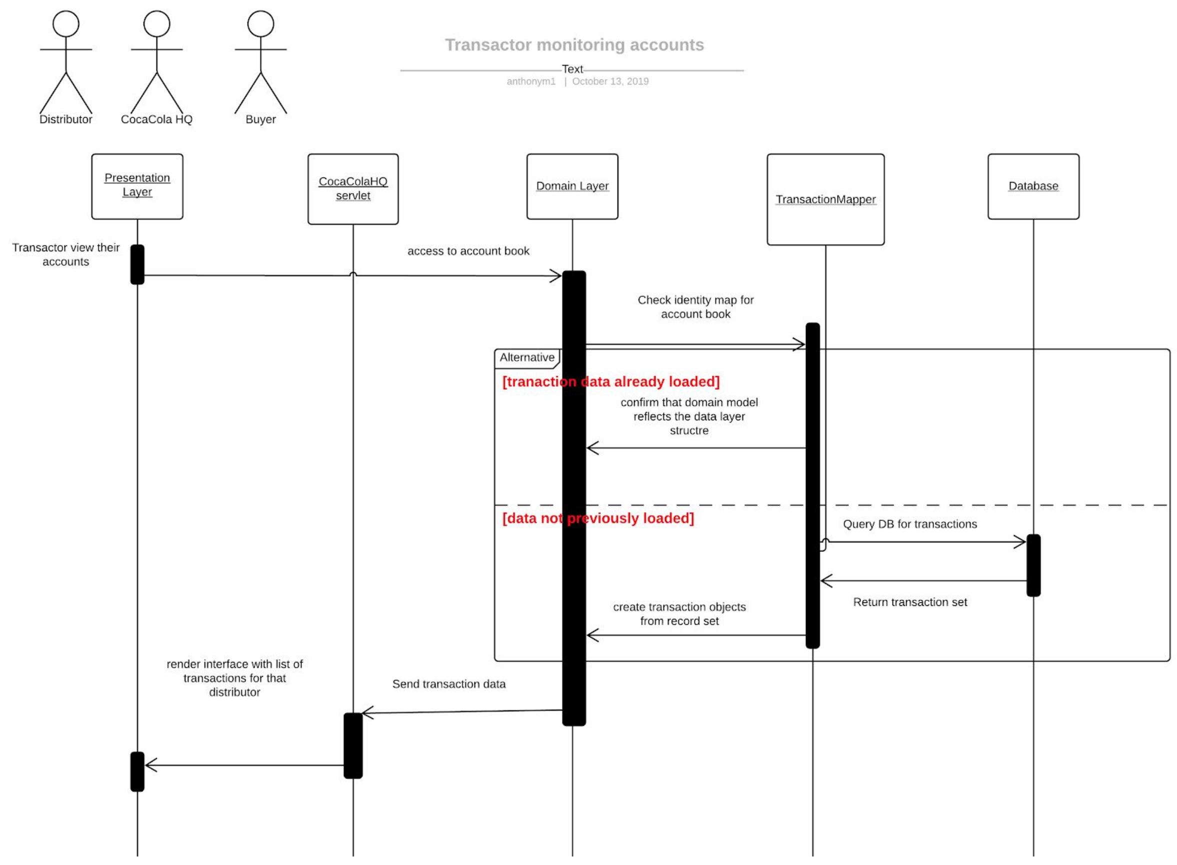
1. buying/restocking with Feature B


Buyer making purchase sequence diargram
anthonym1 | October 13, 2019
Buyer
Presentation Layer
Supplier Facade
Domain Layer
Mapper Layer
Database
create Supplier Facade interace
User logs in
cunstructor fills InventoryData
ProductMapper
query DB
Constructs and delivers inventory information
results are returned
return results set
Alternative
call canShip()
Buyer presses buy
canShip(): false
inform user that order cannot be fulfilled
Text
Buyer presses buy
canShip():true
Create new transaction
TransactionMapper
Map transaction object to table
Store transaction in buyer and distribuor's account book. Subtract amount from distributor's inventory
Text
update retailer's account book
Send transaction data
Send back TransactionDTO and render results

2. Retailer buying with feature A functionality


Buyer making purchase sequence diargram
anthonym1 | September 16, 2019
Buyer
Presentation Layer
Service Layer
Domain Layer
DataMapper Layer
Database
Buyer presses buy
Check if distribution center has inventory available
Create new transaction
Map transaction object to table
Store transaction in buyer and distribuor's account book. Subtract amount from distributor's inventory
update retailer's account book
Text
Send transaction data
render interface with updated transaction

3. Transactors viewing transactions


Transactor monitoring accounts
Text
anthonym1 | October 13, 2019
Distributor
CocaCola HQ
Buyer
Presentation Layer
CocaColaHQ servlet
Domain Layer
TransactionMapper
Database
Transactor view their accounts
access to account book
Check identity map for account book
Alternative
[tranaction data already loaded]
confirm that domain model reflects the data layer structre
[data not previously loaded]
Query DB for transactions
Return transaction set
create transaction objects from record set
render interface with list of transactions for that distributor
Send transaction data