Contents

1 Introduction

This section aims to clarify the goal of the project, as well as describing the philosophy and the tools used by this team to efficiently collaborate and achieve better results. In addition, any assunmptions made about the project are reported and a short description of the all content is given.

1.1 Goal of the project

Given the description of a software application, an e-broker[1] system, use a variety of diagrams(UML, dataflow etc.) in order to better analyze the requirements, functional or not, of the application.

1.2 How we worked

1.2.1 Trello

To better collaborate and organize our work, we used *Trello*, a flexible project management tool, by assigning weekly tasks to each member of the team.

1.2.2 Git

In order to exploit work parallelism, while keeping track of all progress the rest of the team made in the meantime, we used the famous VSC *git*[2] using *this* branching model. For those unfamiliar with git, it is a distributed version-control system, tracking changes in source code during software development to better coordinate work among programmers. Whenever a task was completed, the corresponding member requested a review. Then, the rest of the team was able to review the set of changes, discuss potential modifications, or even enhance the commit. Eventually, in most cases, the request was approved and merged into the main branch.

1.2.3 Discord

As in every team project, in order to optimize performance, meetings really were a necessity. We sure had some face-to-face meetings, but that was not always possible. So, another tool we used is *Discord*, a platform designed for video gaming communities, that specializes in text, image, video and audio communication between users in a chat channel. Using Discord's video-chatting and screen-sharing features helped us co-ordinate and co-operate easy and fast. As a team, we agreed on having short but frequent meetings(2 or 3, 10 minute meetings per day) in order to take full advantage of time, optimizing performance in general.

1.3 Submission

Work load was pretty well distributed amongst all members of this team throughout the whole process of deployment. The one member responsible of reviewing the final version of this pdf and submitting it to e-class is *Konstantinos Koyias*.

[1] An e-broker is a brokerage house that allows you to buy and sell stocks and obtain investment information from its Web site

[2] One can take a look at the source code of this project or even contribute, *here*

1.4 Assumptions

Despite the detailed and informative description of the project, there were a few points we had to intervene, make our own assumptions and decide on what is best to follow. The most important of those were:

- A newtwork partition needs to be included in possible scenarios.
  Take a look at *Command Transmission* for more.
- A client does not have to wait for a command report, it will be delivered at random time. Meanwhile a client can carry on navigating or issuing another command. The same holds for the e-Broker System forwarding a command to the Athens Stock Exchange. Both of these requests are asynchronous. Take a look at the UML *Sequence Diagram* for more.

1.5 Chapters

1.5.1 Structured Analysis

This chapter contains:

- The dataflow diagram of the e-Broker system.
- Dataflow diagrams for levels 1 and 2 of procedure decomposition.
- The process decomposition tree.
- Specifications for:
  1. The process of transmitting order in Structured English.
  2. The process of supply estimation using a Decision Table.
  3. The process of a client's log in the e-Broker system using a Tree.
  4. Removing a command from the Data Dictionary.

1.5.2 UML

In this chapter, we used the general-purpose, developmental, modeling language UML in order to visualize the design of the e-Broker system. Specifically, the following were included:

- ▹ Use Case diagram of the system.
- ▹ Class diagram of the system.
- ▹ State-machine diagram of the entity "Command".

To better model the stock selling/buying operations we included:

- ▹ A main success case Scenario and some alternative ones.
- ▹ An Activity diagram.
- ▹ A detailed Class diagram.
- ▹ A Sequence Diagram.
- ▹ A Communication Diagram.

1.5.3 Structured Design

This is the chapter where a systematic methodology using

- a Programm Structure diagram and
- a piece of pseudo-code that formulates the control unit of
  - the main transformation
  - the M3 transformation
  - the M4 transformation

  and the presentation unit of
  - the M2 transformation

  for the corresponfing Programm Structure diagram.

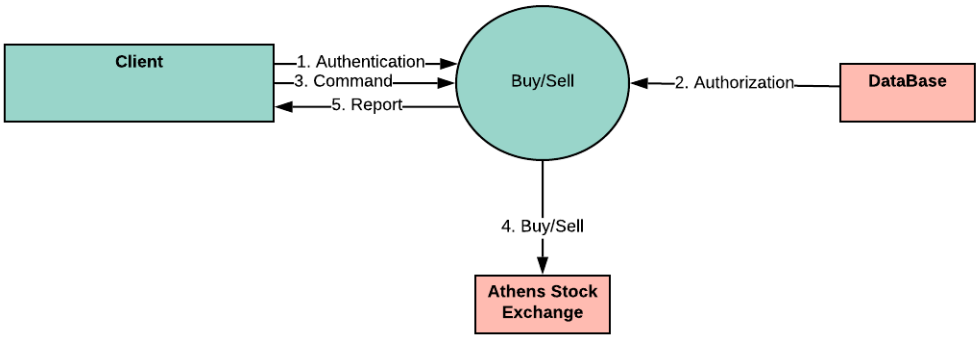
determines the design specifications of the application.

2 Structured Analysis

Let us try and understand how the system works in a logical, using a systematic, graphical approach, Structured Analysis.

2.1 e-Broker Dataflow diagram

A simple, overall, graphical representation of e-Broker is the one below

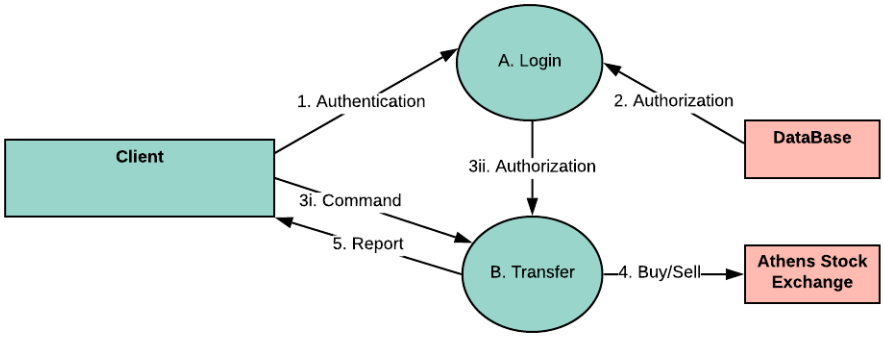


Level 0 is the simpliest of all dataflow diagrams. A client fills the log in form and his access rights are specified using the database on the right. Then he issues a command(buy/sell) which is forwarded to the A.S.T and as a result, he gets a report.

2.2 Procedure decomposition Dataflow diagrams

Now let us try and decompose the above diagram a bit, to better isolate each unit and process of the system, taking a more precise look on each of them seperately.

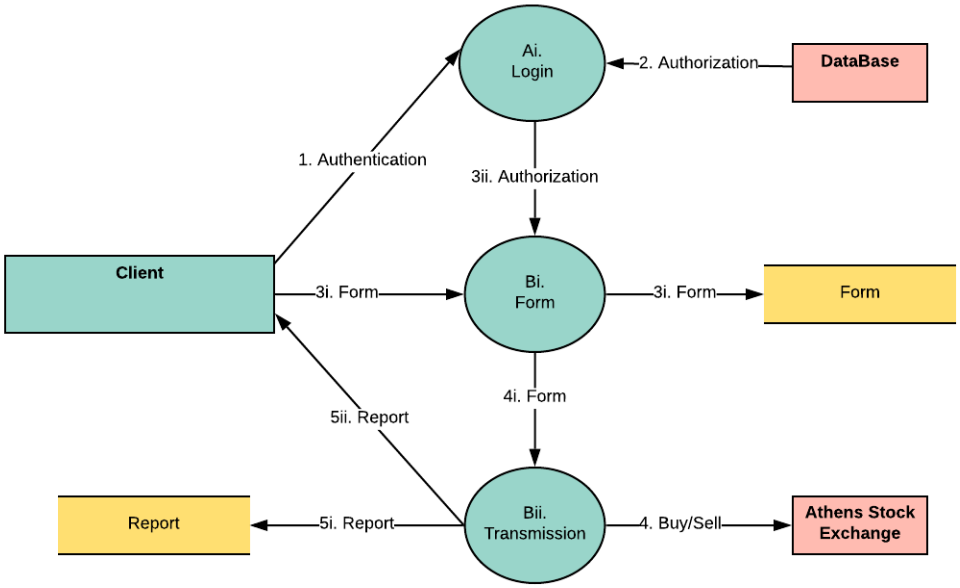
2.2.1 Single level decomposition



Notice how the Log in unit passes on the authorization data, which include all information about that particular client to the Transfer unit so that the transaction can be completed.

2.2.2 Double level decomposition

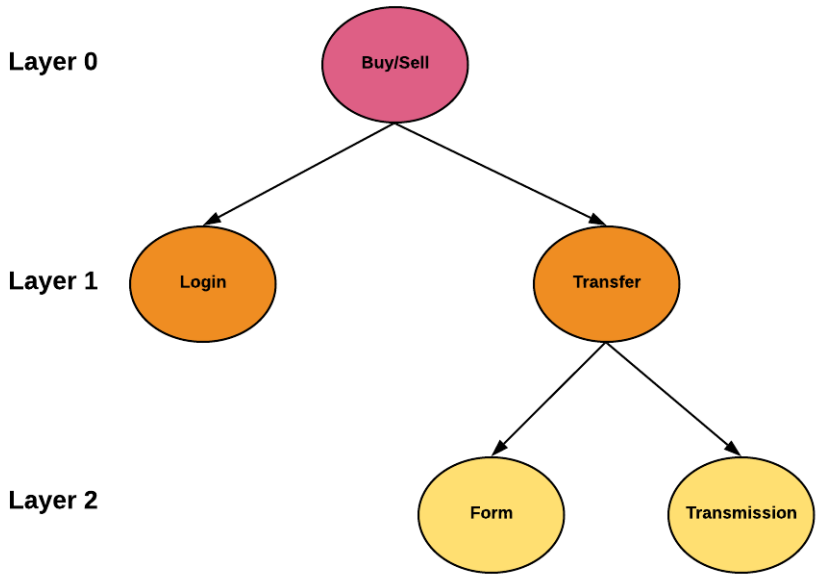
As you might thought by reading the description of this system, the Transfer unit can be broken down to a **Form** and a **Transmission** unit, which include what their names imply. So, the most detailed diagram version for the e-Broker system looks as follows



Notice how two **regular files** come into the game in this very last diagram. One for the form defining the transaction details and the other for the report describing the outcome of the trasnsaction.

2.3 Process decomposition Tree

Even though it is implied by the diagrams above, a nice graphical representation in the form of a tree of how complex processes decompose to smaller, isolated and more simple is certainly useful.



Notice how the leaves of this tree manage to get a **single** job done and can not be further decomposed.

2.4 Specifications

In this section, each specification is examined separately using a variety of Structured Analysis tools.

2.4.1 Transmitting Order

Using Structured English, the Transmitting Order process from the e-Broker system point of view looks as follows:

while Form Is Invalid OR Customer is Re-editing **do**
  Display Form
  On Submit do
  if Form Is Incomplete **then**
    Display "All fields required" message
  else if There is Field with Invalid Type **then**
    Display "Invalid field value" message
  else
    Display Preview
    Display "Back to Editing?" message
  end if
end while
Forward command to A.S.T
Get response
if Response is negative **then**
  Display "Rejected" message
else
  Display "Approved" message
end if

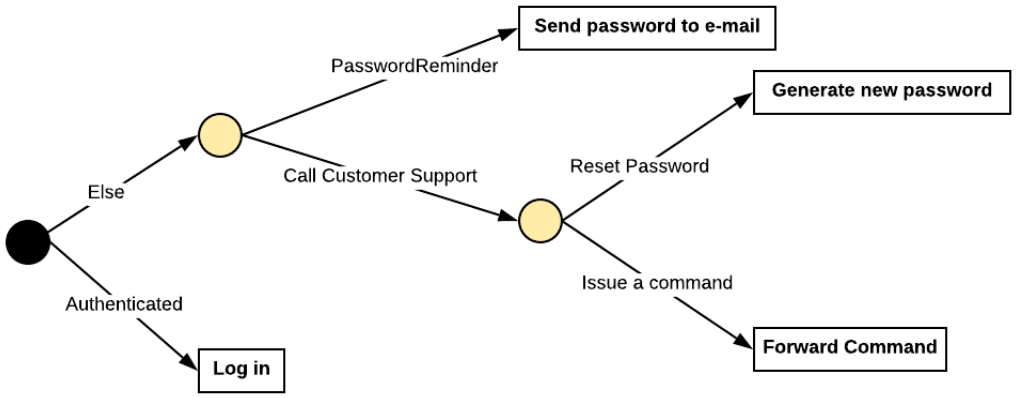
2.4.2 Supply Estimation

In this little section, we will use a decision table in order to help us simplify the complex process of calculating the exact charge of a Customer for a transaction.

Conditions	Rule 1	Rule 2	Rule 3
Volume	$0 - O_1$	$O_1 - O_2$	$O_2 - O_3$
Actions			
Set charge equal to	$P_1\% \times Volume$	$P_2\% \times Volume$	$P_3\% \times Volume$

2.4.3 Log in

Authentication is not always simple.
This is a detailed **decision tree** that proves just that.



2.4.4 Data dictionary of Command Report

For each element included in the Command Report, one can find information about its format data type and other meta-data in the table below.

Data Dictionary		
Name	**Description**	**Type**
SN	Stock Name	nvarchar(50)
TT	Transaction Kind	boolean
PR	Price	float
TM	Time	date
N	Stocks Number	integer
OPR	Total/Overall Price	float
ON	Overall Stocks Number	integer
SP	Supply Estimation	float

Notice that Transaction_Kind is of type boolean, where true corresponds to a buy and false to a sale.

3 UML

In this chapter, all neccessary UML diagrams for the e-Broker system will be created. In order to distinquish relationship types, we used this map:

{always: include, might: extend, generic: generalization}.

This gets a bit more clear in the following subsection.

3.1 Use Cases

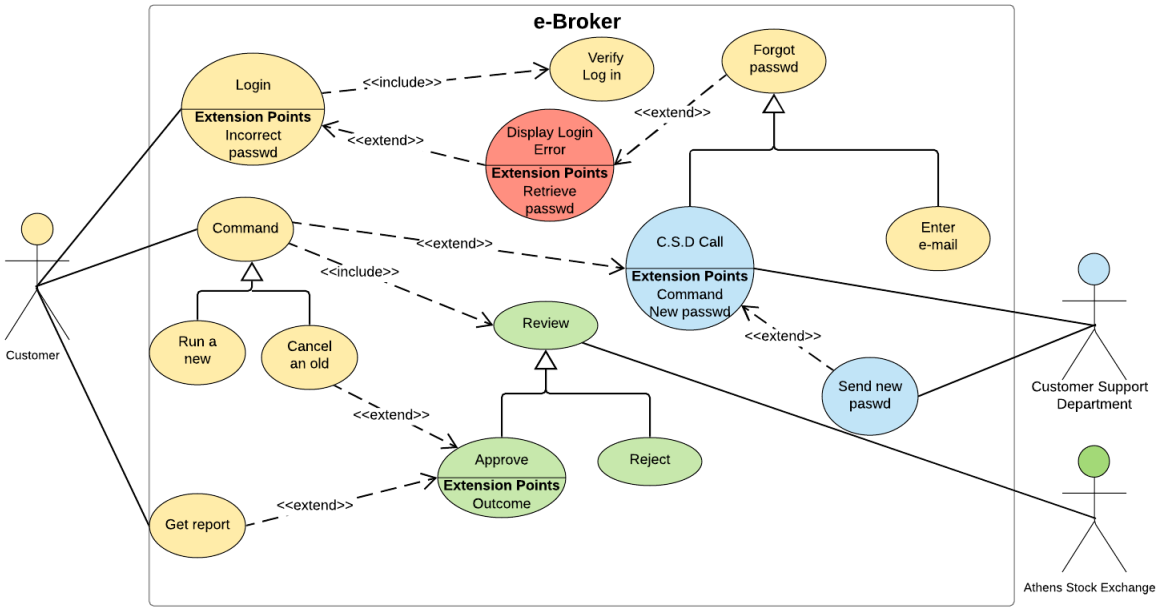
After taking a good look at the description of the system, we came up with the following actors:

- Customer(Primary)
- Customer Support Department(Secondary)
- Athens Stock Exchange(Secondary)

and the corresponding use cases. Each customer can:

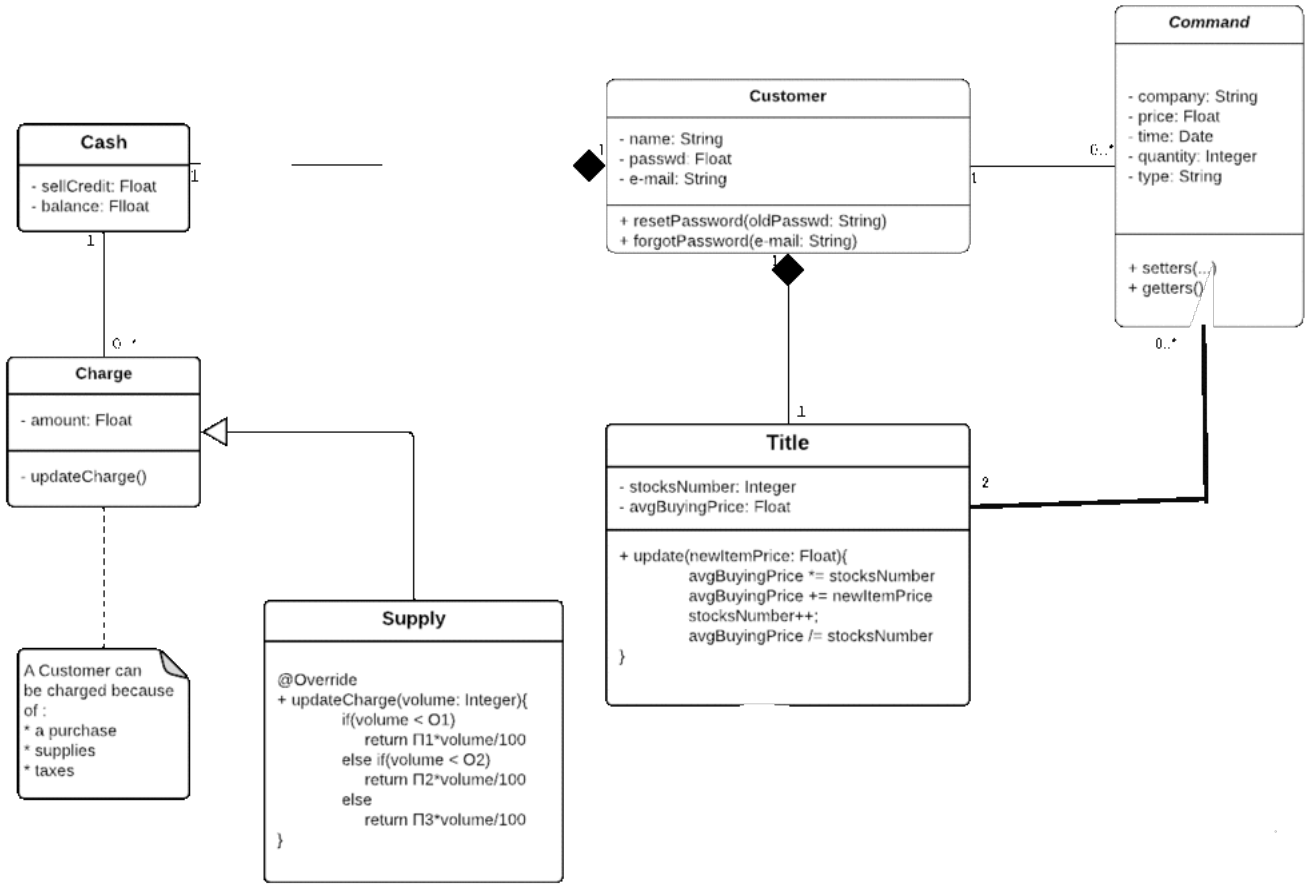
- **log in**
  This action will **always** trigger a password check, and it **might** lead to an error message. In this case a **generic** "forgot password" request **might** happen, which can be either a password reminder via an e-mail, or a call to the *C.S.D* for a new password or/and a command to be executed.
- **command**
  A **generic** command, can be either a creation or a cancellation, both of which will **always** have to be reviewed by the *A.S.E.* This **generic** review will either lead to an approval or a rejection. An approved command **might** at some point get cancelled, otherwise the *Customer* will
- **get a report**

so we came up with the following use case diagram



3.2 Classes

This is the section, where classification took place. Notice that C.S.D and A.S.E are just objects, like instances, not classes, so they have no place in the following diagram.



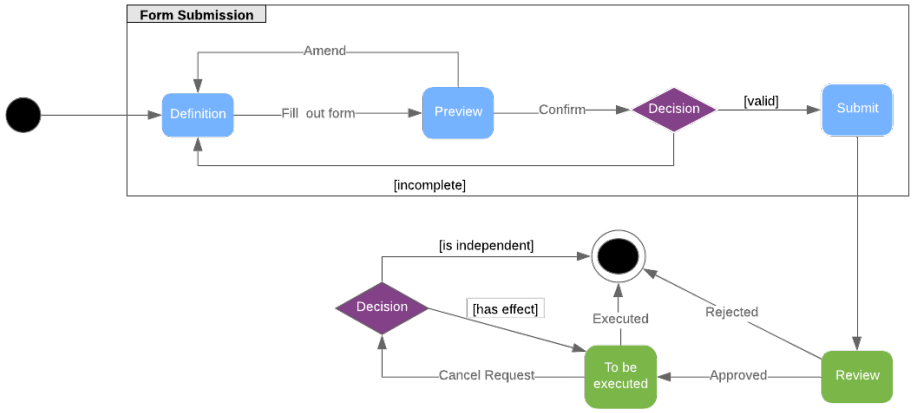
Let us now explain what is happening on the diagram above.

- First off, some data is kept for each **Customer**, a **Cash** and a **Title**. These, can not really exist without a Customer. This is called composition and we used a black diamond to denote it.
- Each Customer is able to request for as many **Command**s as he wishes, if any, to be executed or cancelled.
- Each Command, is associated with exactly two(2) Titles, one of them belongs to the buyer and the other, to the seller.

Notice that we could use three extra classes extending **Charge**, namely Purchase, Supply and Tax but other than Supply, a class with a different updateCharge aproach, all of them would not add any attribute or method up to their parent, so we decide not to define classes Purchase and Tax. Lastly, to avoid a huge diagram we omitted the following fields of Customer:
fathername, mothername, address, birthDate, id, tax_id and financialService.

3.3 Command Lifecycle

By reading the description of the system, one might notice that the lifetime of the *command* entity happens to be a realy interesting one. Well, in that case, a common technique is using a state-machine diagram to better visualize all possible outcomes of feature.
For a command to be executed, all properties it has, need to first be defined using a form.
After a successful submission, meaning all fields contain values of the right type, the command will be reviewed. In case of an approval, it will be executed as soon as possible. In the meantime, it can be cancelled. But, this in only possible in case of an independent command, meaning it has no recursive effect to any previously executed commands, because they can not be reverted now in any way. The diagram follows:



3.4 Command Transmission

3.4.1 Scenarios

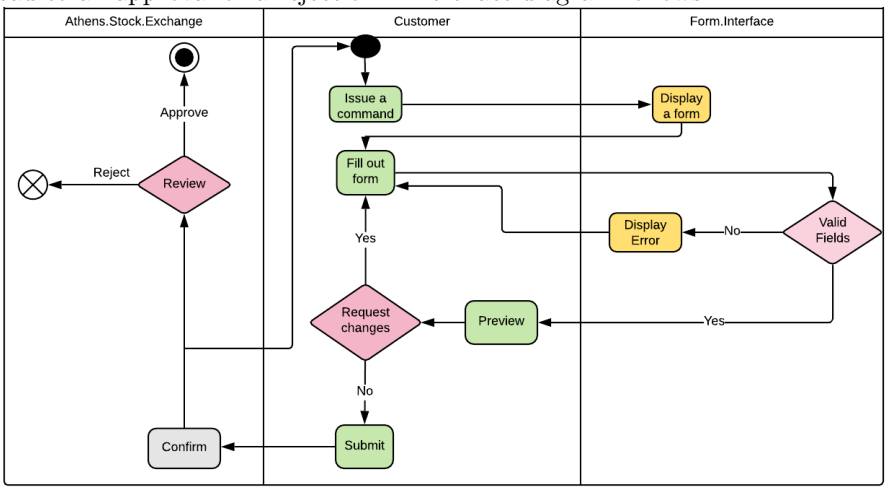
The transmission of a command, although it looks pretty straightforward, is actually a complex one. Let us take a closer look at it step-by-step in order to detect all points where something could go wrong and think about what happens in each of those cases. Steps should look as follows:

1. Customer fills out all attributes of the form
   (a) an incomplete form or
   (b) a field set with a non-matching type or out of range value
   (e.g “name” set to an integer, “time-target” refers to the past etc.)

   will take us back to step 1
2. Customer previews the form
   (a) customer decides to modify some field, back to step 1
3. Customer confirms and submits
   (a) a network partition occurs, action depends on system availability policy(e.g PACELC, PAVEL etc.)
4. A.S.C receives request
   (a) command execution is not possible, then command is rejected
5. A.S.C approves request, pushes command in the “To be executed” priority queue of e-Broker

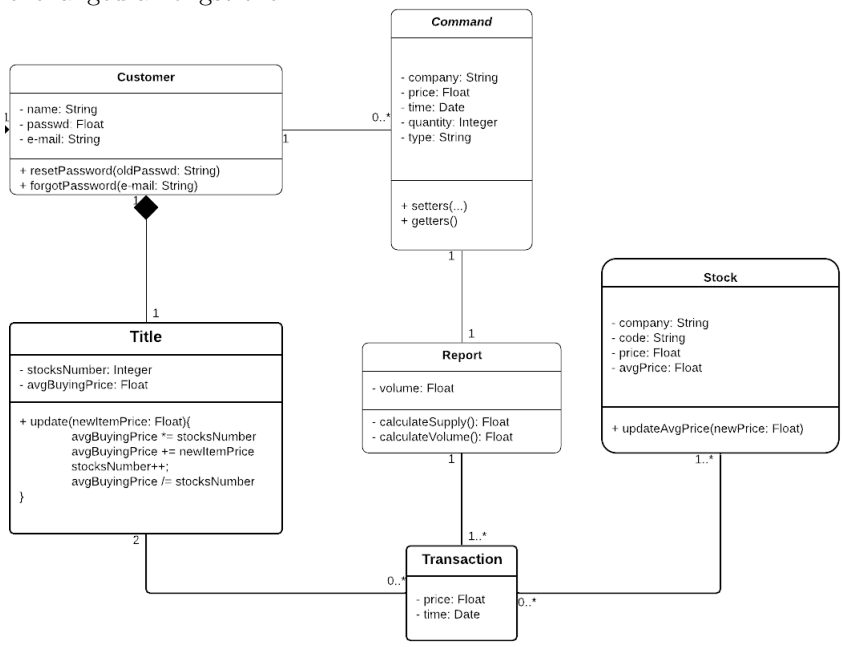
3.4.2 Activity Diagram

Once a Customer wishes to execute a command, a form to fill out opens up. It is not until it is complete and contains data of the right type, that the customer can preview it, perhaps request changes and then move on to submit it. At this point the A.S.T receives the submitted form, confirms of receiving it(now the customer is able to parallelly issue another command) and proceed with reviewing it. That will either lead to an approval or a rejection. The exact diagram follows:

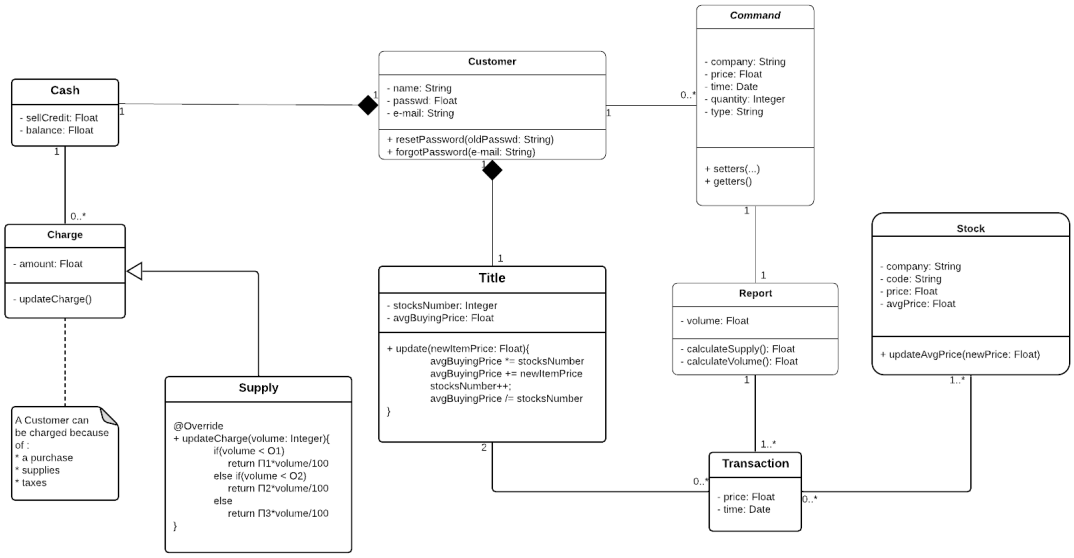


3.4.3 Detailed Class Diagram

Now, let us take a closer, more detailed look at all classes interacting during the Command phase as well as the messages exchanged amongst them.

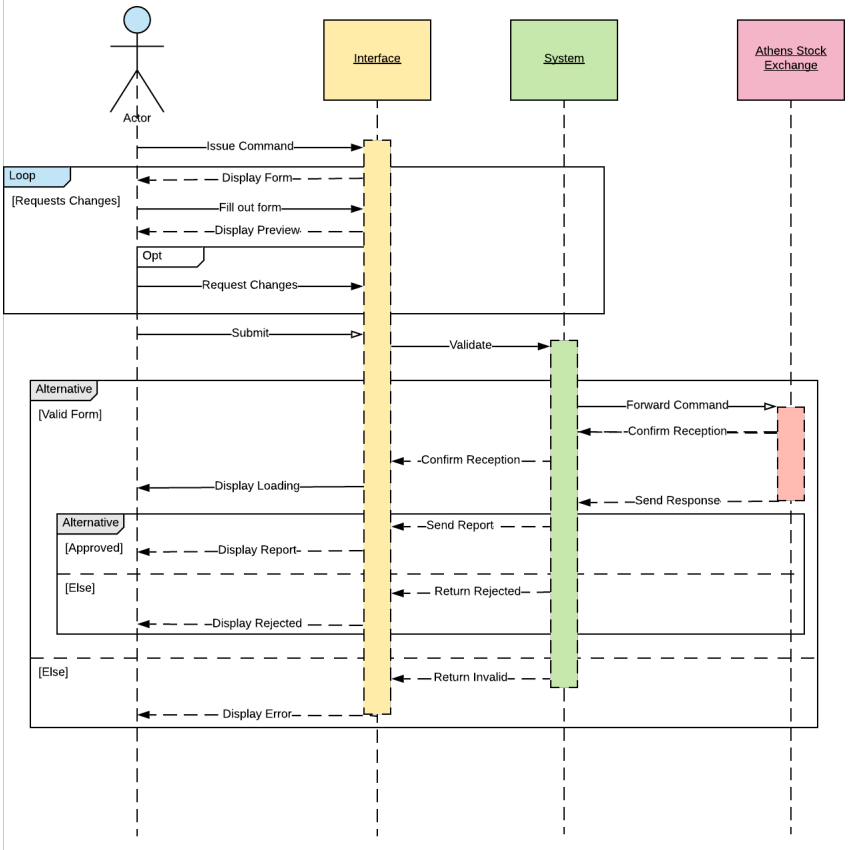


As we mentioned above, C.S.D and A.S.E are not classes, but just an instance of one, so they are omitted. The above diagram tells us that for each Command, there is a unique **Report**, associated with at least one(1) **Transaction** that involves at least one(1) **Stock** and is associated with exactly two(2) Titles, one for the buyer and one for the seller. So, a complete and very detailed class diagramm would look like this



3.4.4 Sequence Diagram

This diagram, in contrast to the Activity one above, analyzes a single scenario from a slightly different scope, using a closer to the implementation approach, intended for the development team to use.

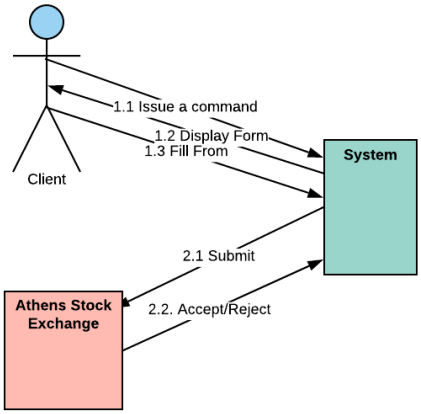


As you might did notice, a solid line denotes a request, while a dashed line denotes a response. Lastly, open arrow heads were used for asynchronous requests, notice that a Customer is able to carry on after

submission and the System will, obviously, not freeze until the Athens Stock Exchange responds after a Customer's command.

3.4.5 Communication Diagram

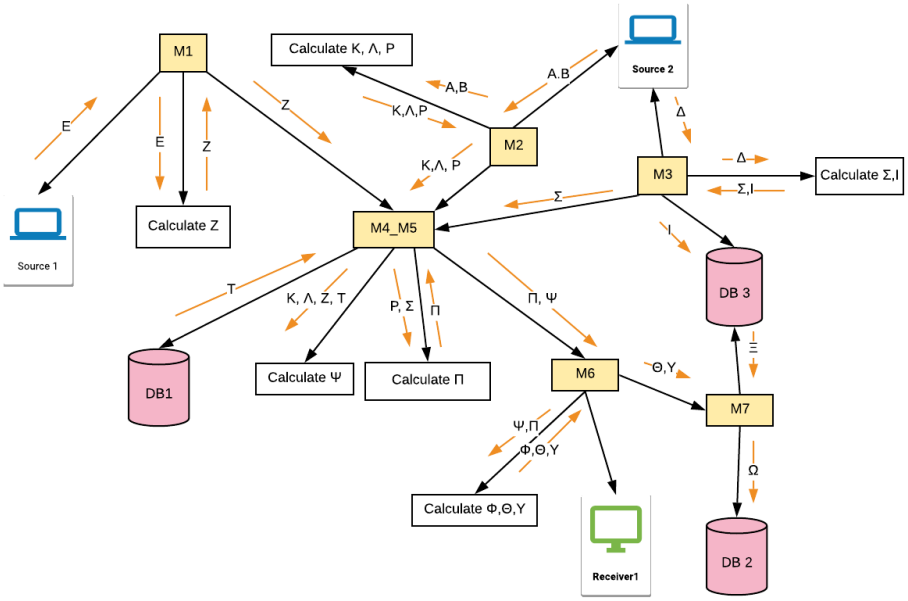
Now, let us focus on object relationships, rather than the sequence of the messages exchanged amongst them.



4 Structured Design

In this section, we attempted to better determine the design specifications of e-Broker by creating a Program Structure diagram and using pseudocode to better illustrate the functionality of some vital procedures for this application.

4.1 From Dataflow diagram to Program Structure diagram



4.2 Pseudocode Representation of Units

4.2.1 Central Transformation

PROCEDURE M2_M4
LOCAL_VAR T, Z, K, $\Lambda, P, \Psi, \Pi, \Sigma$
INITIALIZE T, Z, K, $\Lambda, P, \Psi, \Pi, \Sigma$
EXEC_M1(Z)
EXEC_M2(K, Λ, P)
GET_T(T)
CALL_M4(Z, K, Λ, Ψ)
CALL_M5(P, Π, Σ)
EXEC_M6(Ψ, Π)
END_PROCEDURE

4.2.2 M3 Transformation Control Unit

PROCEDURE M3(OUT: Σ)
LOCAL_VAR Δ, I, Σ
INITIALIZE Δ, I, Σ
GET_Δ(Δ)
CALL CALC_M3(Δ, I, Σ)
PUT_I(I)
END_PROCEDURE

4.2.3 M4 Transformation Calculation Unit

PROCEDURE CALC_M4(IN: Z, K, T , Λ, IN/OUT: Ψ)
...
//calculate Ψ
...
END_PROCEDURE

4.2.4 M2 Transformation Presentation Unit

The presentation unit of a trasnforamtion is the one component, responsible of communicating with all users and external devices/systems. As one can see, M2 is only communicating with one external device, Source 2. So, the pseudocode for the Presentation Unit of M2 looks as follows:

PROCEDURE GET_M2(IN/OUT: A, B)
read A, B from Source_2
END_PROCEDURE

5 Epilogue

5.1 Using Structured Analysis

Looking back at the Structured Analysis chapter, one might realize that a graphical represantation of a complex system can be very enlightening and informative both for a user and for a designer. This is what makes Structured Analysis a great tool, not just for software developmnet, but any kind of system analysis. To be more precise, in our case, e-Broker is pretty complex at first glance, but data-flow diagrams manage to give a clear picture of the system flow by dividing each process as much as needed in order to make the whole concept easily understandable by not only a designer but a common user as well. Therefore, Structured Analysis is definately a tool that we are going to use to decompose any complex project that we are assigned in the future, isolating each component of the system. This will definately make development an easier and much more enjoyable process.

5.2 Working with UML

Being CS students, using object-oriented languages like, for instance, C++ and Java, is pretty much a common case. That familiarity helped us absorb the whole usage and syntax of UML rather quickly. This why UML follows the object oriented concept in the first place. However, it did not take long to notice that UML is a lot more complex than it looks at first sight. Supporting a variety of different diagrams it was really hard to express ourselves and describe the same system over and over again from diffent aspects. One thing we realized using these tools is that avoiding too much detail is a pretty good idea. These are high-level tools, allowing us to grasp the concept and philosophy of a system, not the exact implementation. So, **KISS**(keep it simple stupid) would be our advice to anyone new to UML, trying to be as precise as possible, omitting exhaustive details which are entailed by the rest of the context and anyone can figure is what you want.

5.3 Conclusion

To sum things up, UML feels more natural to a software developer in contrast to Structured Analysis that seems to be the perfect fit for simple users to understand how a complex system works taking a really high-level look at the systems' components. As CS students, it makes more sense to go with UML. However, I think that Structured Analysis, a more simple approach to system analysis, is what we would rather use for projects that are not too large.