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 diagramms(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] 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*

1.3 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*.

1.4 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:

-
-

1.5 Chapters

1.5.1 Structured Analysis

This chapter contains:

- The dataflow diagramm of the e-Broker system.
- Dataflow diagramms 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 diagramm of the system.

▹ Class diagramm of the system.

▹ State-machine diagramm 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 diagramm.
- ▹ A detailed Class diagramm.
- ▹ A Sequence Diagram.
- ▹ A Communication Diagram.

1.5.3 Structured Design

This is the chapter where a systematic methodology using

- a Programm Structure diagramm 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 diagramm.

determines the design specifications of the application.

2 Structured Analysis

2.1 e-Broker Dataflow diagramm

a

2.2 Procedure decomposition Dataflow diagramms

b

2.3 Process decomposition Tree

c

2.4 Specifications

d

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

bb

2.4.3 Log in

cc

2.4.4 Command Purge

dd

3 UML

In this chapter, all neccessary UML diagramms 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

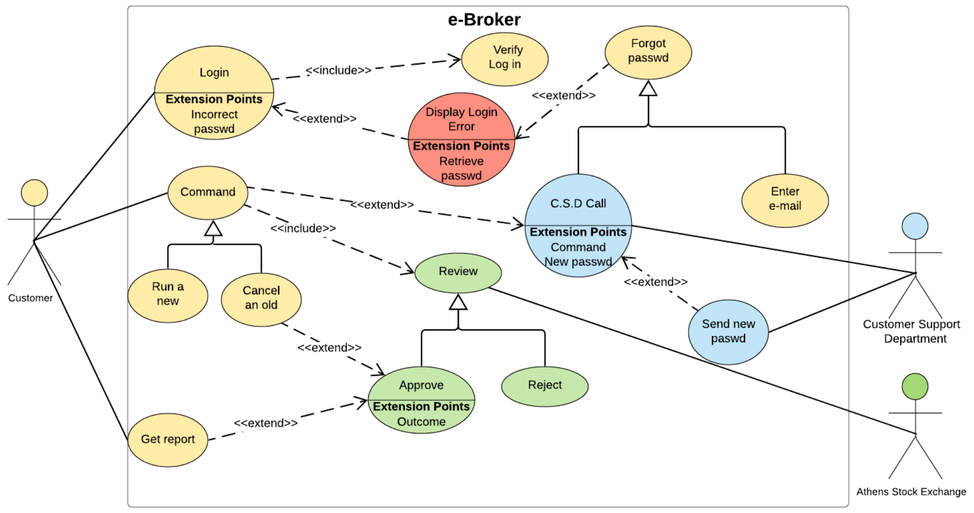
After taking a good look at the description of the system, we came up with the following actors:

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

and the corresponding use cases. Each customer can:

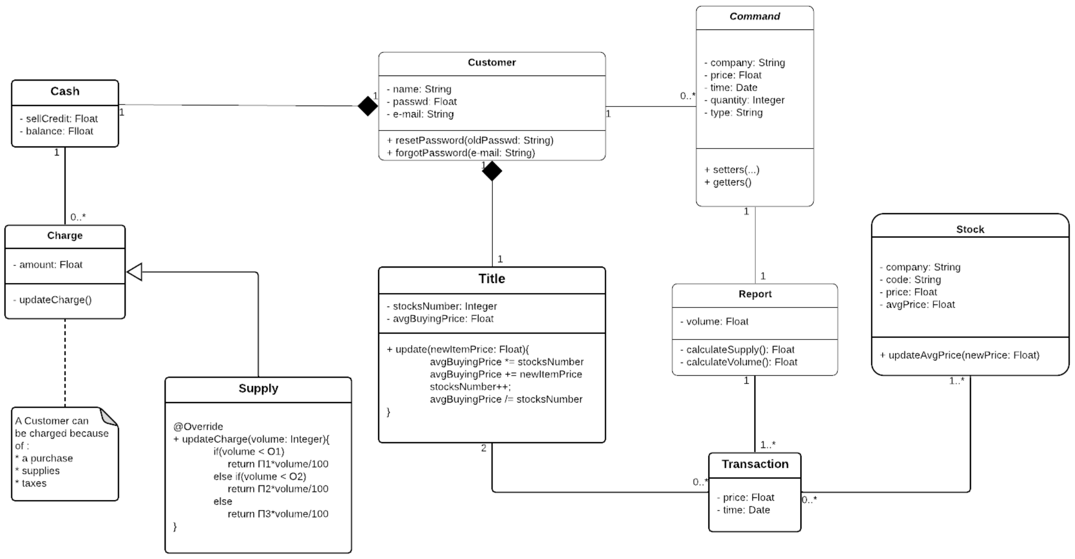
- **log in**
  This action will **always** trigger a password check, and it **might** lead to an error message. In this case a **generic** "forgot password" request **might** happen, which can be either a password reminder via an e-mail, or a call to the *C.S.D* for a new password or/and a command to 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 diagramm


e-Broker
Login
Extension Points
Incorrect passwd
<<include>>
Verify Log in
Forgot passwd
<<extend>>
Display Login Error
Extension Points
Retrieve passwd
<<extend>>
Command
<<extend>>
C.S.D Call
Extension Points
Command
New passwd
Enter e-mail
<<include>>
Review
Run a new
Cancel an old
<<extend>>
Send new paswd
<<extend>>
Approve
Extension Points
Outcome
Reject
Get report
<<extend>>
Customer
Customer Support Department
Athens Stock Exchange

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 diagramm.



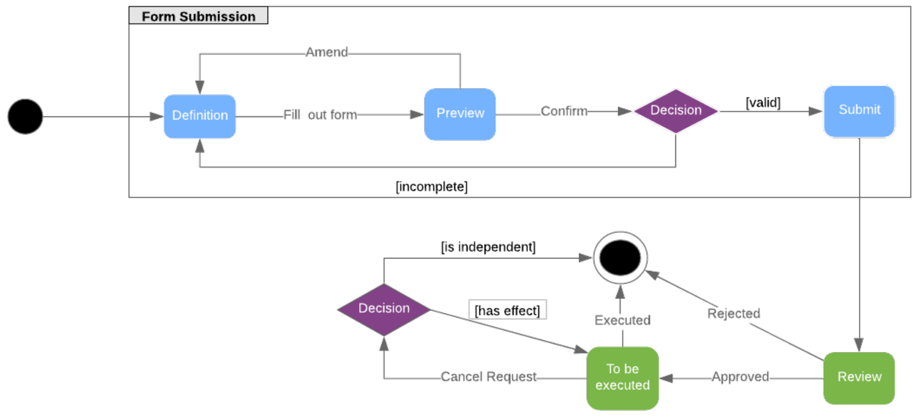
Let us now explain what is happening on the diagramm above.

- 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.
- 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 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 diagramm 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 diagramm 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 diagramm 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.)

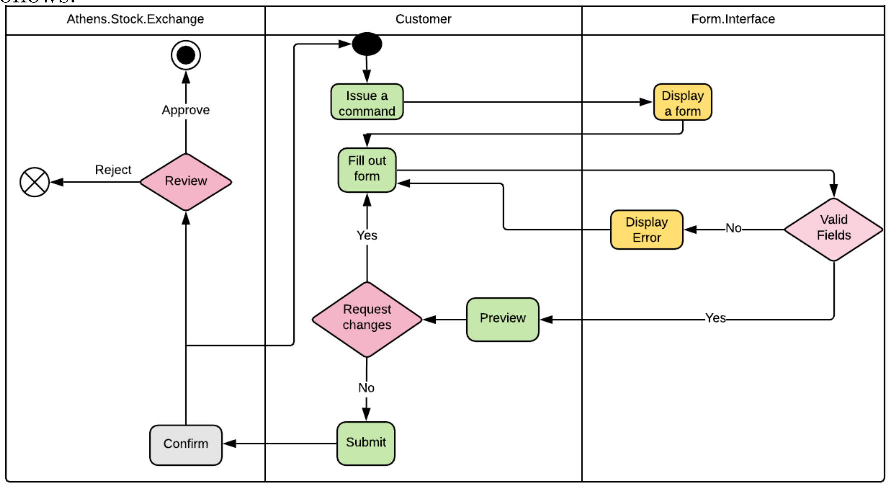
4. A.S.C receives request

   (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 Diagramm

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 diagramm follows:

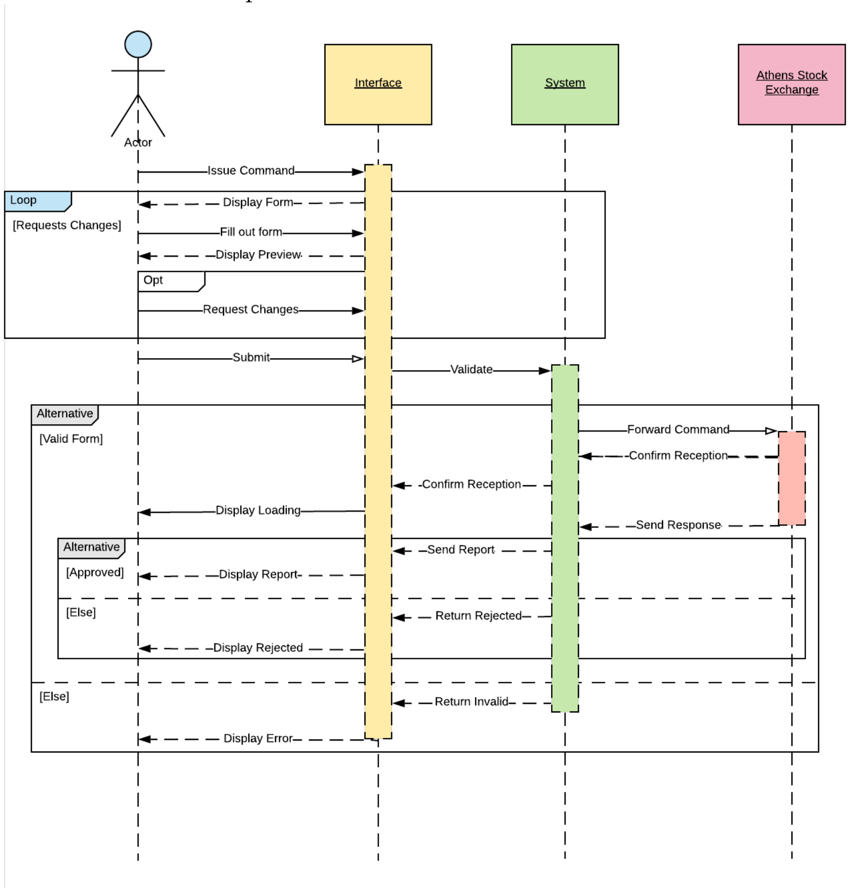


3.4.3 Detailed Class Diagramm

bb

3.4.4 Sequence Diagramm

This diagramm, 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 Diagramm

dd

4 Structured Design

ee

5 Epilogue

k t