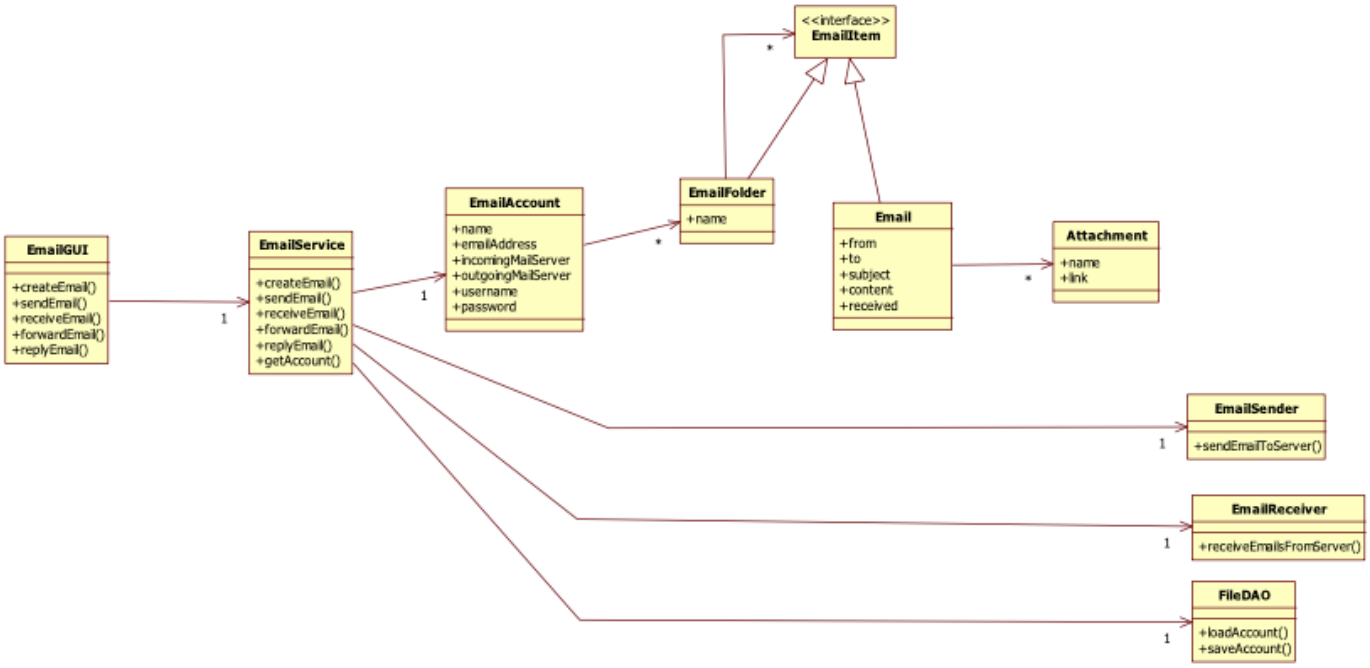
Question 1 [25 points] {30 minutes}

Suppose that you have to write an email client application according to the following requirements:

1. You can create an email
2. You can send an email. You do this by calling the outgoing mail server.
3. You can receive emails. You do this by calling the incoming mail server.
4. You can reply to an email
5. You can forward an email
6. The email application contains by default the following mail folders: **inbox**, **outbox**, **sent items**, **deleted items**, **spam** and **drafts**. You can add your own mail folders. Any mail folders can contain other mail folders. You can move email messages to any mail folder.
7. Emails can contain any number of attachments
8. Your design should have a clear separation between user interface, business logic and other technical classes
9. All emails and attachments are stored on your local file system.

Draw the class diagram that shows how your design works. Make sure your class diagram contains all the important information to communicate your design. Your class diagram should also contain methods that access the email server and the methods that load the emails from the file system and store the emails to the file system. Use the best practices we studied in the course. If you apply a design pattern, make sure this design pattern really gives an advantage to your design.

RESULT 1


<<interface>>
EmailItem
EmailAccount
+name
+emailAddress
+incomingMailServer
+outgoingMailServer
+username
+password
EmailFolder
+name
Email
+from
+to
+subject
+content
+received
Attachment
+name
+link
EmailGUI
+createEmail()
+sendEmail()
+receiveEmail()
+forwardEmail()
+replyEmail()
EmailService
+createEmail()
+sendEmail()
+receiveEmail()
+forwardEmail()
+replyEmail()
+getAccount()
EmailSender
+sendEmailToServer()
EmailReceiver
+receiveEmailsFromServer()
FileDAO
+loadAccount()
+saveAccount()
*
*
*
1
1
1
1
1

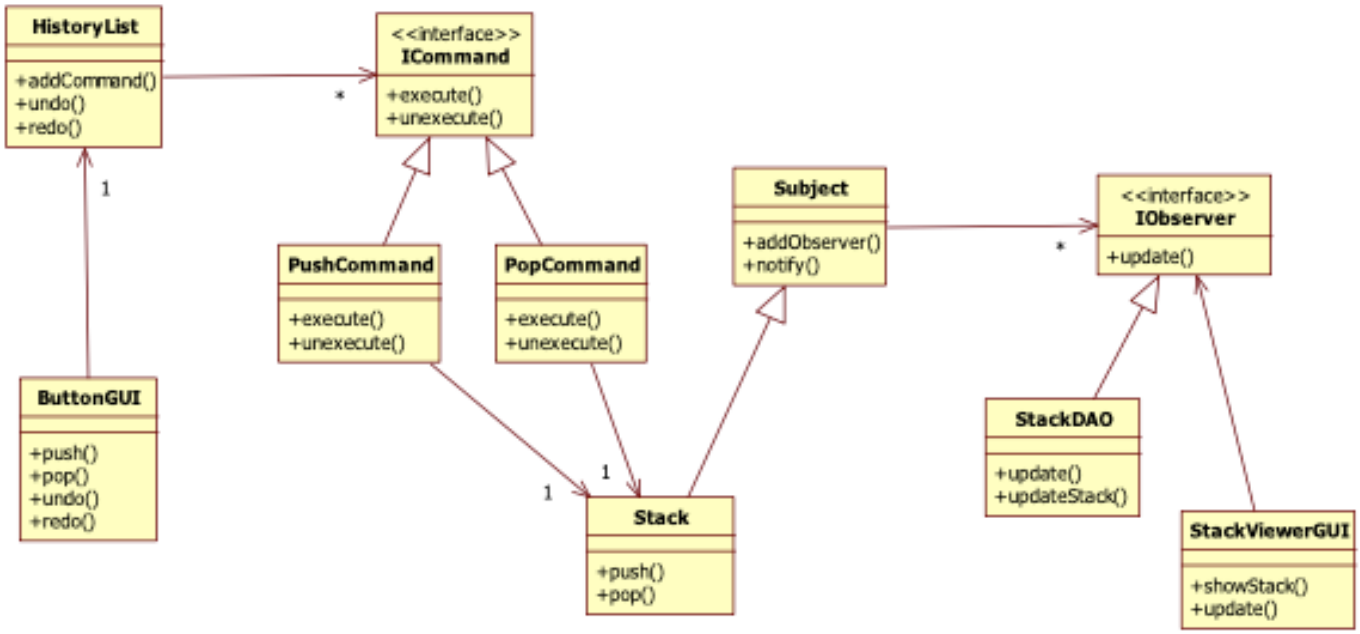
Question 2 [50 points] {60 minutes}

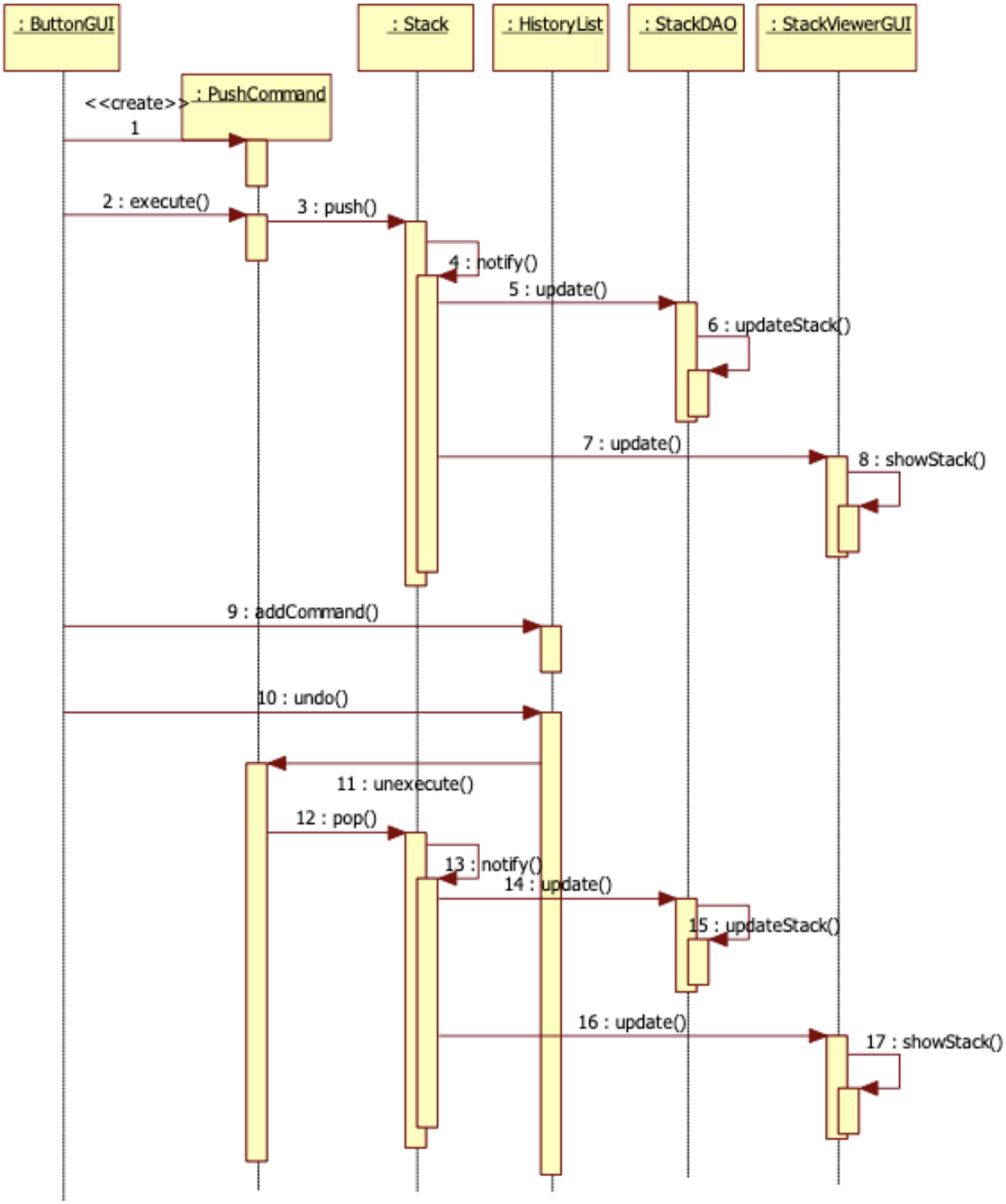
Suppose we have to design a simple Stack application:
In case you don't know what a Stack is: A Stack is a list of objects that work according the LIFO principle, which is Last In, First Out.
The 2 important methods are push(x) which adds the object x to the top of the stack, and pop() which takes the last added object off the stack.

- The Stack application has the following requirements:
- The Stack contains String objects.
- We have a GUI class which allows us to enter a String. Then this GUI class has 4 buttons: A **Push** button, a **Pop** button, an **Undo** button and a **Redo** button.
- We have a StackViewerGUI class which shows the content of the stack.
- It should be easy to add more GUI classes to this application that show the content of the stack in a different way.
- Every time we change the stack, the content of the stack is updated in the database.
- The stack should be reusable in other applications, so it should not depend on other classes.

a. Draw a **class diagram** of your design of the stack application. Make sure your class diagram contains all the important information to communicate your design.
b. Draw a **sequence diagram** showing the following sequence:
    1. First we push a particular String on the stack
    2. Then we click the undo button.

RESULT 2


HistoryList
+addCommand()
+undo()
+redo()
*
<<interface>>
ICommand
+execute()
+unexecute()
1
ButtonGUI
+push()
+pop()
+undo()
+redo()
PushCommand
+execute()
+unexecute()
PopCommand
+execute()
+unexecute()
1
1
Stack
+push()
+pop()
Subject
+addObserver()
+notify()
*
<<interface>>
IObserver
+update()
StackDAO
+update()
+updateStack()
StackViewerGUI
+showStack()
+update()


: ButtonGUI
: PushCommand
: Stack
: HistoryList
: StackDAO
: StackViewerGUI
<<create>>
1
2 : execute()
3 : push()
4 : notify()
5 : update()
6 : updateStack()
7 : update()
8 : showStack()
9 : addCommand()
10 : undo()
11 : unexecute()
12 : pop()
13 : notify()
14 : update()
15 : updateStack()
16 : update()
17 : showStack()

Question 3 [20 points] {20 minutes}

Suppose we call a method on class A, and as a result one or more different methods of other classes need to be called. There are 2 patterns that we studied that can be used to implement this requirement:

1. The Observer pattern:
2. The Chain of Responsibility (COR) pattern:

a. Explain clearly all differences you know between these 2 patterns.
b. When would you use the observer pattern and when would you use the COR pattern?

You do NOT get points for explaining the 2 patterns in isolation. You need to explain the **differences** between the 2 patterns.

RESULT 3

a.

Observer pattern	COR pattern
Observers don't know each other	Handler needs to know the next handler
All observers are called	A handler may decide not to call the next handler
We have a list of observers	We have a chain of handlers

b.

You use COR when only one class needs to handle the action, and you don't know which class.

You use the observer pattern when all observers need to be notified

QUESTION 4

Describe how the **Façade** pattern relates to one or more of the SCI principles you know. Your answer should be about half a page, but should not exceed one page (handwritten). The number of points you get for this questions depend on how well you explain the relationship between the Façade pattern and the principles of SCI.