**Builder**

The application user has the option to send an email.

We chose this pattern because we have a complex product to produce (the email itself) without knowing in advance the exact components of the complex product.

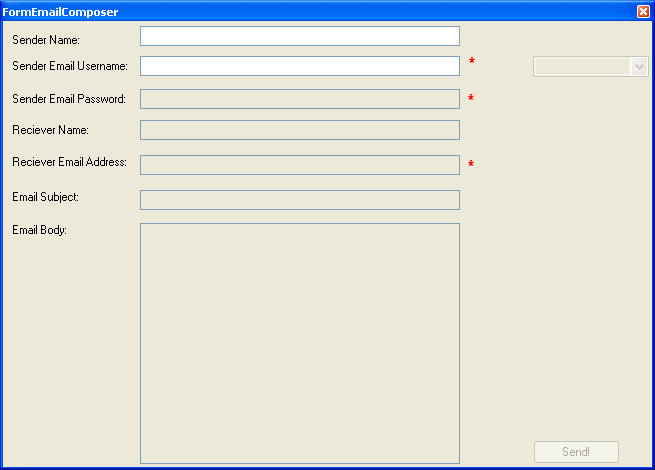
The builder pattern offers the possibility to receive each initialization parameters one by one and then returns the resulting constructed complex product (without knowing in advance the final configuration of the product).

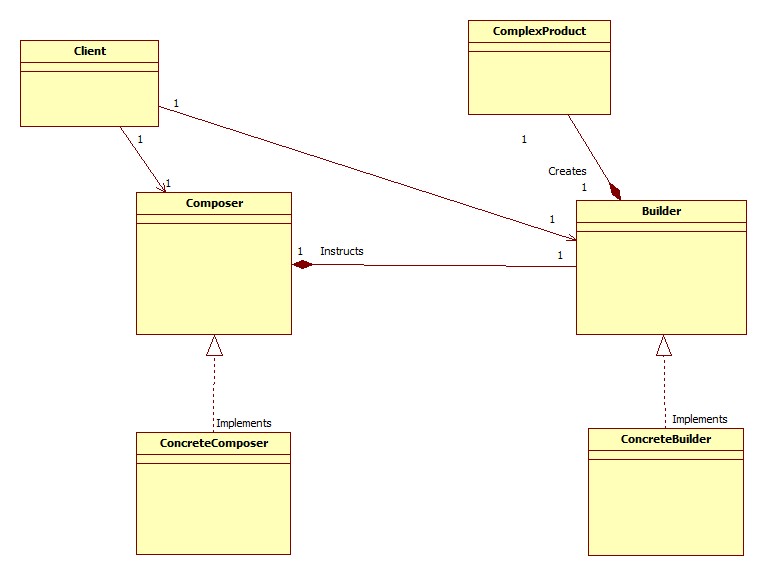
Using builder in our program:

Our implementation of this pattern creates an acquaintance between the client and the composer.

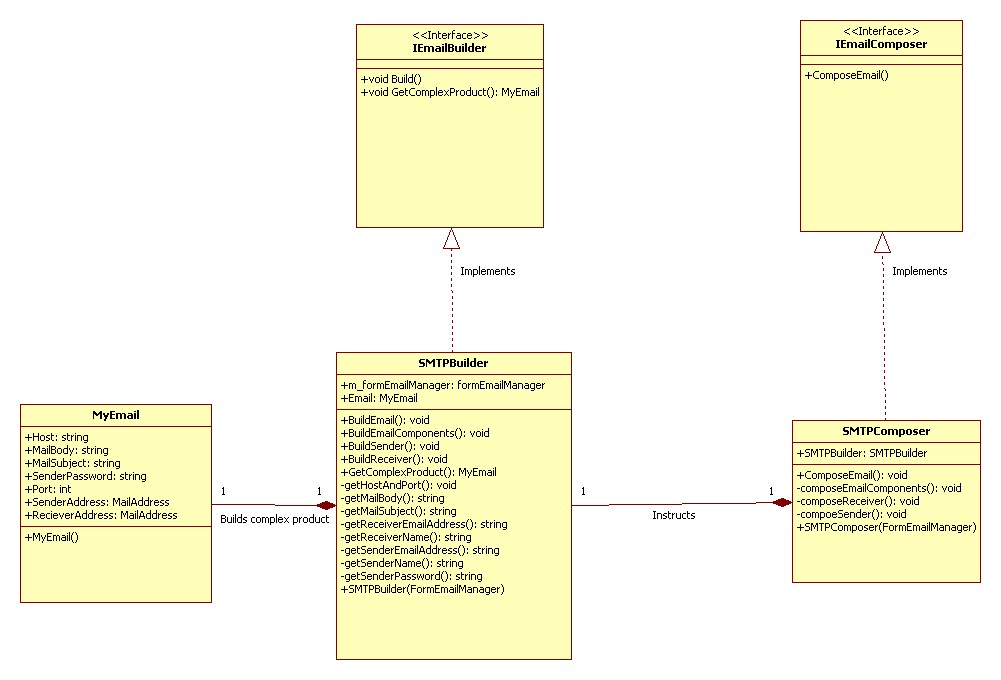
In our case, The “FormEmailManager” interacts with the “SMTPComposer” which interacts with “SMTPBuilder”. The SMTPBuilder implements an interface **I**EmailBuilder and SMTPComposer implements an interface **I**emailComposer. The SMTPBuilder will get the components one by one, will combine them, and return the complex product which is the email itself.

* While composing an email, there are few components to fill. Some are essentials and some are not, therefore we cannot know in advance the final configuration of the complex product.
* Using SMTP protocol for transferring the email through the internet, therefore we have two concrete components: SMTP Builder and SMTP Composer, which makes it more modular if ever in the future we would like to change the protocol.
* In this specific product there’s no importance in which component is being created first. But if ever needed in the future, this pattern is being implemented in a way that can support this feature.

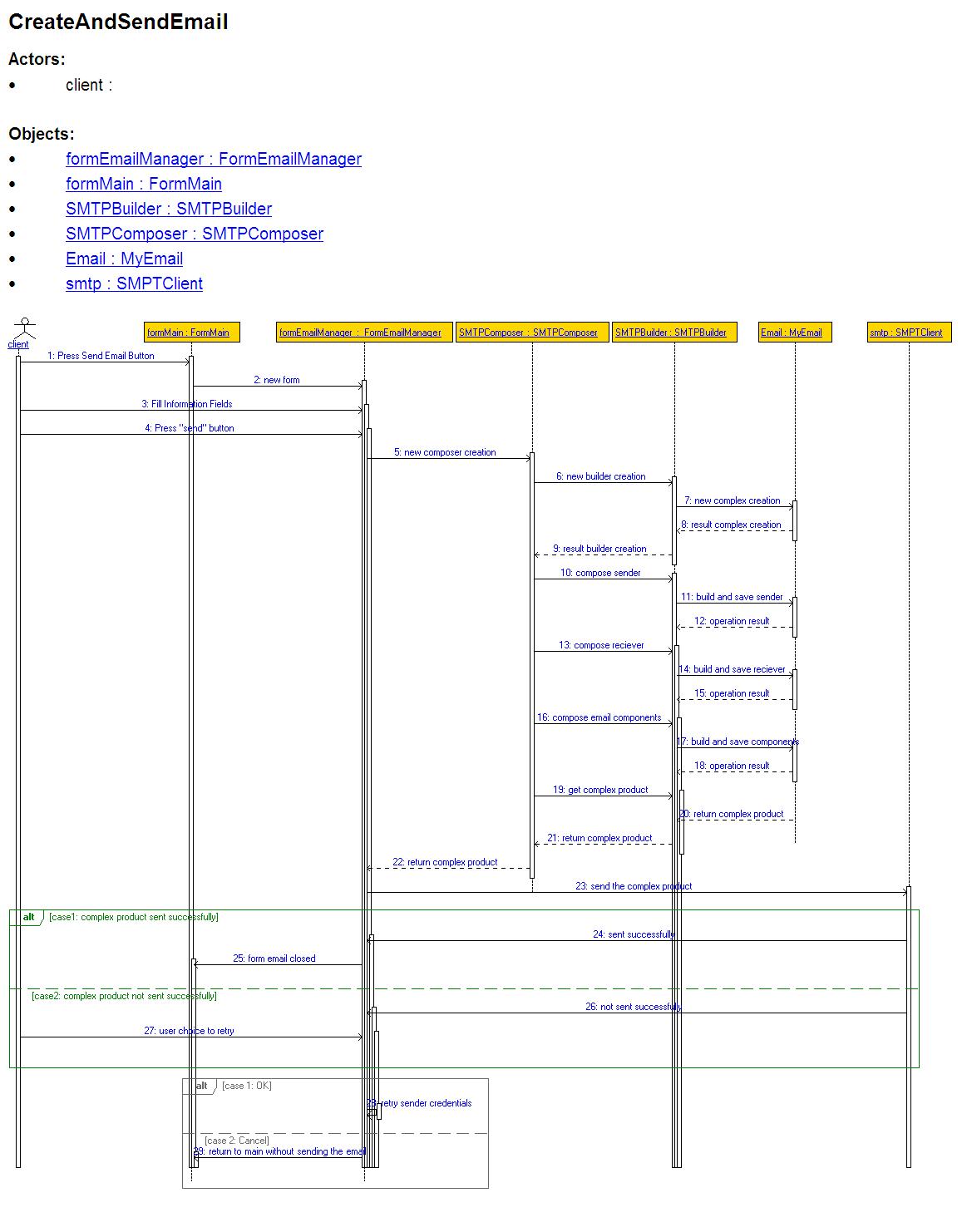


General view of our implementation:

**Builder Class Diagram**



**Builder Sequence Diagram**

****

We know that operations that takes time should be shown as a slanted arrow.

The EasyCRC does not able this action.

Therefore we will detail the needed operations :

1. 4: Press "send" button
2. 23: send the complex product

**Factory Method**

The application user has the option to watch his albums, each album has photo(s) inside of it.

We chose this pattern because we have a polymorphic family (PictureBox) and we wanted to separate the creation logic to be more maintainable.

The factory method pattern implements the concept of factories and deals with the problem of creating objects without specifying the exact class of object that will be created.

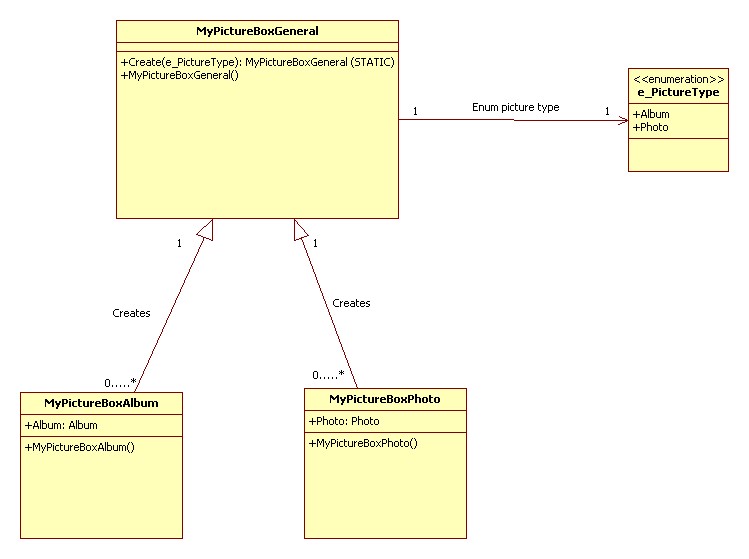
Using factory method in our program:

Of all implementations of this pattern we chose the Static Factory Method (Base Class). Base class determines the concrete to create by the context (enum) received.

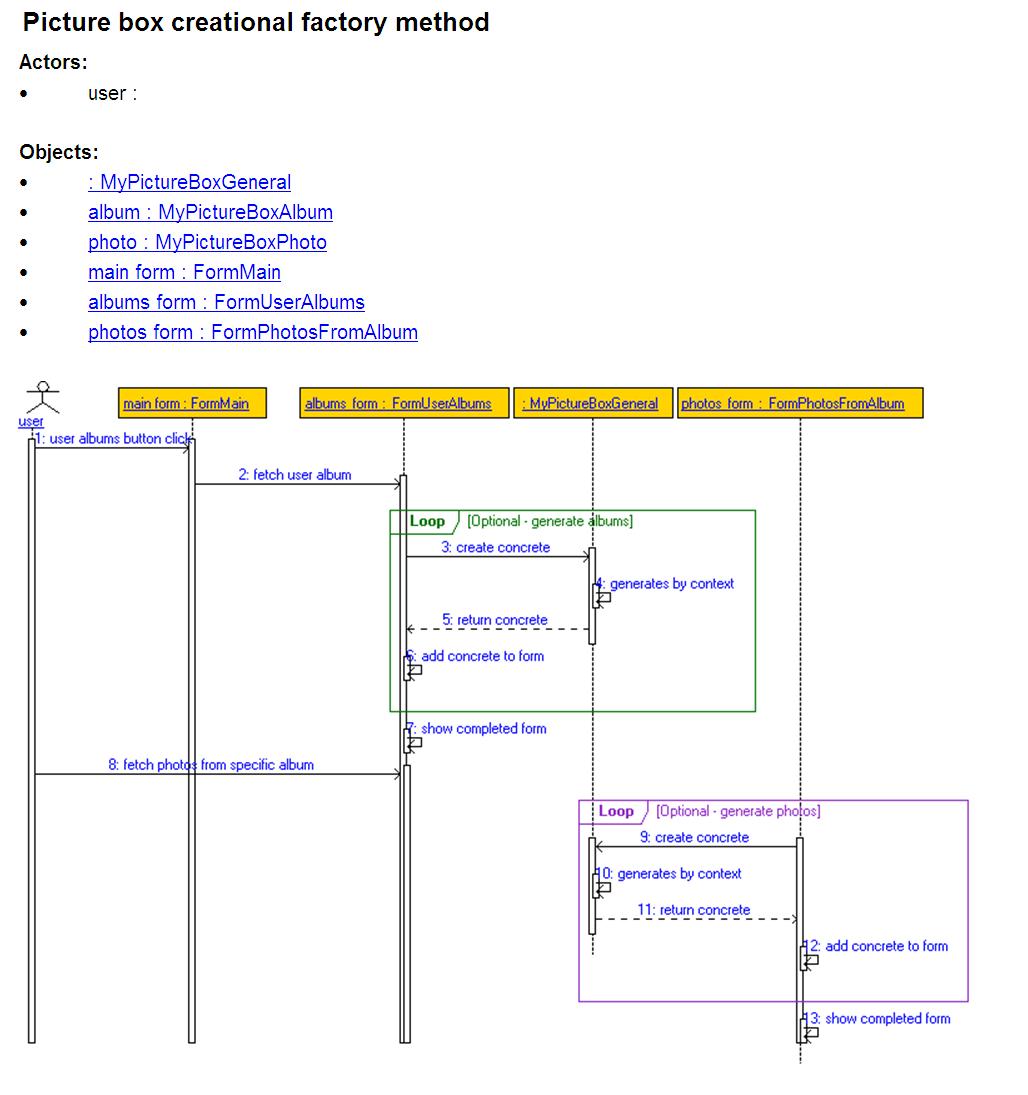
We know this implementation generates a cycle between the tightly couple: the base class and the concretes.

In our case, “MyPictureBoxPhoto” & “MyPictureBoxAlbum” are the concretes that inherit from “MyPictureBoxGeneral”, both cannot exist without the other.

**Factory Method Class Diagram**



**Factory Method Sequence Diagram**



We know that operations that takes time should be shown as a slanted arrow.

The EasyCRC does not able this action.

Therefore we will detail the needed operations :

1. 2: fetch user album
2. 8: fetch photos from specific album

**Prototype**

The application user has the option to choose a specific event he already attended to, and display it on screen in few different viewable controls.

We chose this pattern because we wanted to specify the kinds of objects to create, and create new objects by copying this prototype (use its reference). We wouldn’t want to have more than one viewable control, and that’s why we would like to use its reference.

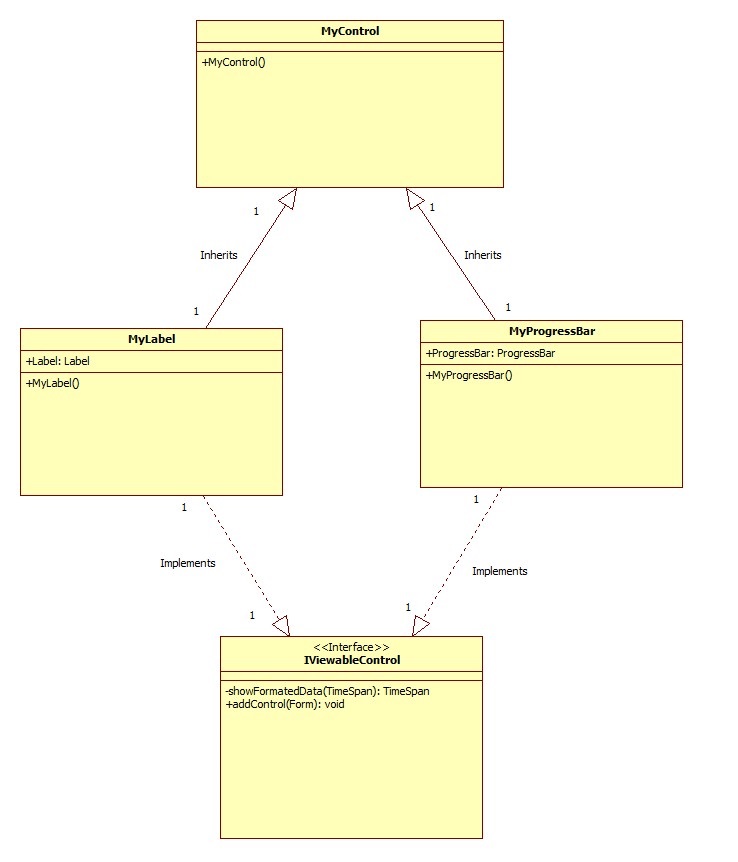
Using builder in our program:

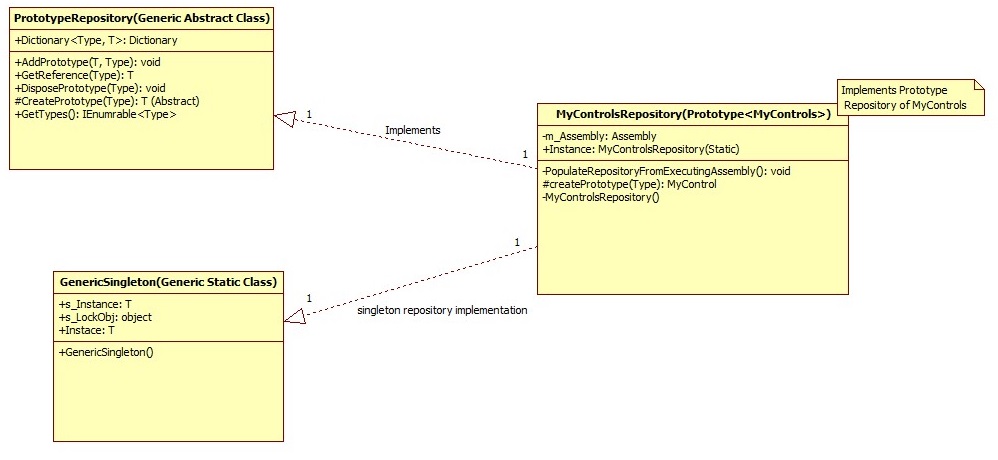
Our implementation of this pattern creates an acquaintance between the client and the composer.

In our case, a comboBox will show two different view components: Label and ProgressBar.

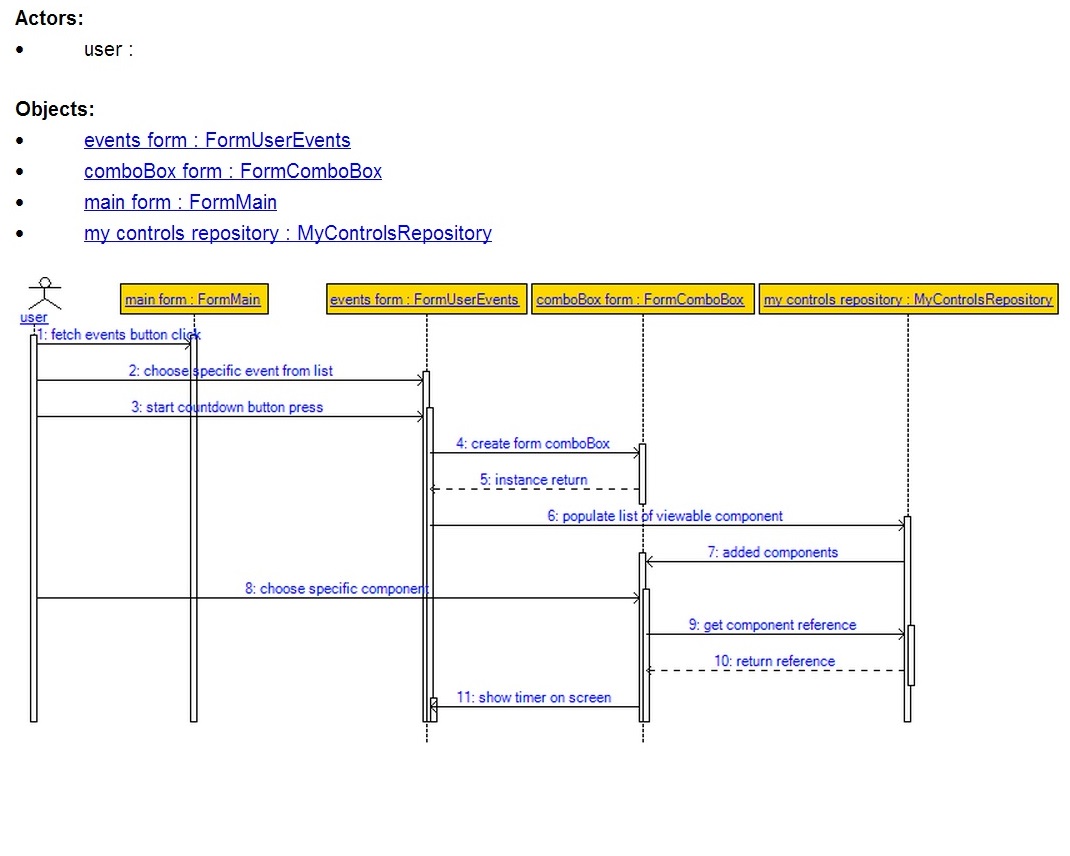
These viewable components are added to the repository during runtime through Reflection. And then from the Dictionary to the ComboBox to user’s display. Therefore we can easily add NEW viewable components that will implement the ShowFormattedData and AddControl methods.

**Prototype Class Diagram**





**Prototype Sequence Diagram**

****

We know that operations that takes time should be shown as a slanted arrow.

The EasyCRC does not able this action.

Therefore we will detail the needed operations :

1. 1: fetch events button click

**Features Explanation**

1. **Events count down.**

**(can be found in class FormUserEvents in method showCountDown row number 43)**

User can choose any of the events that were already RSVP (as attending or as maybe), once choosing an event and pressing on the start count down button a timer will be started presenting the remaining time to the beginning of the event (the timer will be updated on intervals of 1 sec).

User can switch at any time the event by another and the timer will immediately show the relevant event count down.

An additional button was added to stop the count down.

Once the event start time has arrived, the timer will set to 0:0:0:0, the text box will change to “event happening now”, a message box will open with the same text and a sound will be played for the user’s attention.

1. **Joining events that your friends have confirmed and being notified for overlapping events taking place at the same time (attending/ maybe. Open events only).**

**(can be found in class formFriendEvents in method “confirmRSVP” row number 59)**

Given the user the possibility to choose a friend from the friend list and fetch the events that friend has RSVP to, then choose the desired event and RSVP to it. If user is trying to confirm (attend/ maybe) an event that takes place at the same time as a former event a message will be shown saying that there is already an event happening to warn/ remind the user of those overlapping events.

1. **Google direct access through the facebook application.**

**(can be found in class FormMain in method “pictureBoxGoogle\_click” row number 405)**

User needs to enter his desired value to search and by clicking the Google Icon it will automatically open an explorer web page with the relevant results from the google’s search engine.

1. **Providing friend’s first name meaning.**

**(can ve found in class FormFriends in method “linkLabelFriendNameMeaning\_LinkClicked” row number 47)**

User can mark each and any one of his friends on his friends list box, once the chosen friend is marked a link will be enabled (beneath the friends list box) and by clicking that link user will be automatically referred to a web page that will present the meaning of the chosen friend.

**Observer**

The application user has the option to view the remaining time for an attended event .

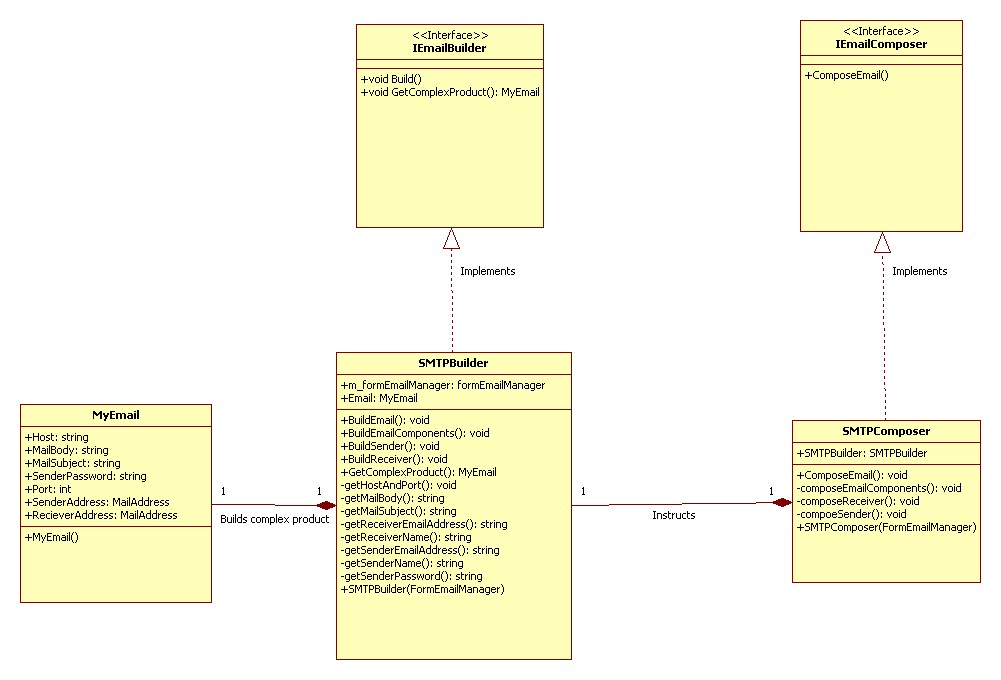
We chose this pattern to enable an online “listening” to the object that is responsible to the event of user event’s happening.

The observer pattern offers the possibility to one component observe another and wait for a specific event to happen and then excute a written method. This is all implemented without generating a cycle between the components.

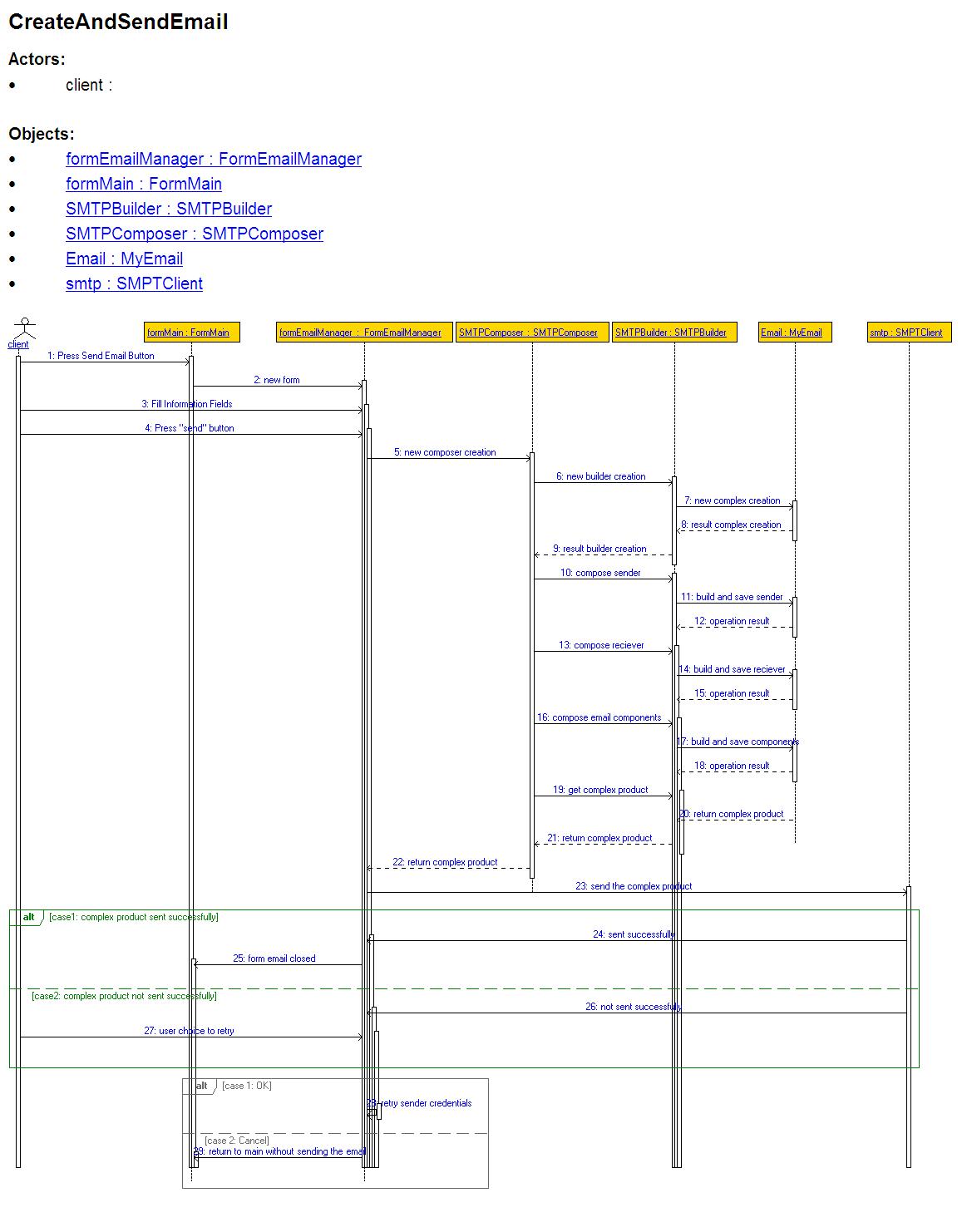
Using ovserver in our program:

Our implementation of this pattern determines the FormMain as the listener of FormUserEvents. When a facebook user chooses the count down button, we will check the TimeSpan instance to get to 0:0:0:0 – then the event will happen, and through the FormMain a message box will pop with a relevant message.

**Observer Class Diagram**



**Observer Sequence Diagram**

****

We know that operations that takes time should be shown as a slanted arrow.

The EasyCRC does not able this action.

Therefore we will detail the needed operations :

1. 4: Press "send" button
2. 23: send the complex product

**Chain of responsibility**

The application user has the option to view his albums and photos from each album

We chose this pattern to move the responsibility of showing pictures from the form to the photo itself to be more maintainable.

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both.

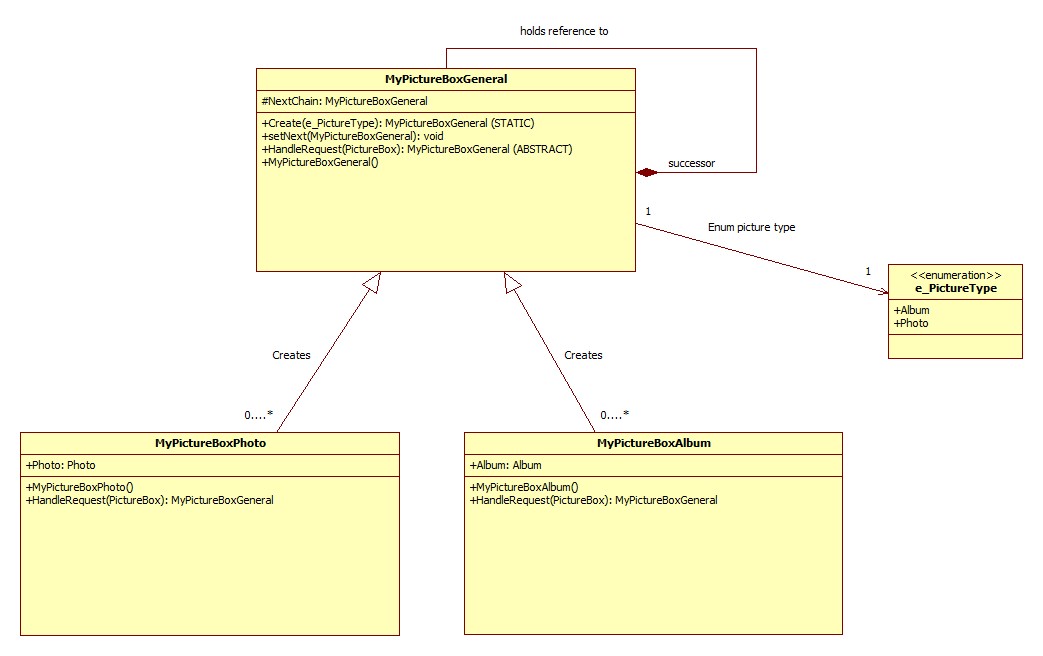
Using chain of responsibility in our program:

once the client chose which album's photos he wanted to show, we will create instances of MyPictureBoxPhote and saving those instances in a list.

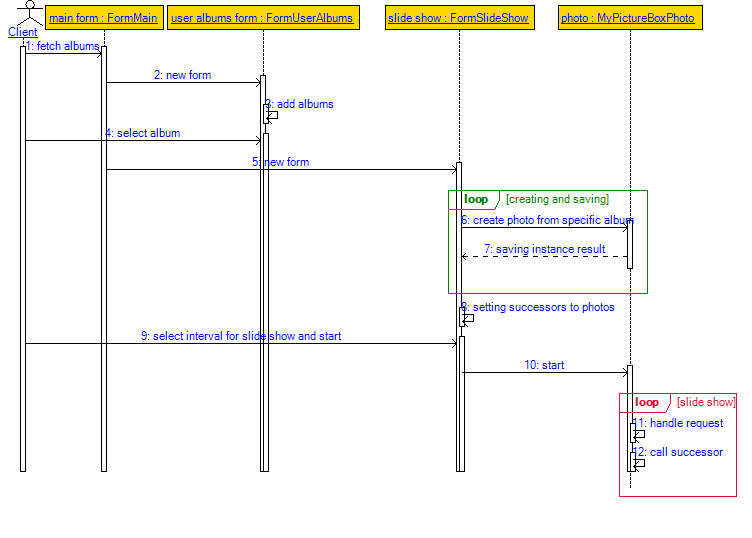
then, setting each photo to its successor.

After the client chooses an interval for the slide show the first photo will be called and doing itself execute meaning that it will show his photo on the picture box on the FormSlideShow. after that, if there is a successor in the chain it will be called until the last photo in the chain

**Chain of responsibility Class Diagram**



**Chain of responsibility Sequence Diagram**

****

We know that operations that takes time should be shown as a slanted arrow.

The EasyCRC does not able this action.

Therefore we will detail the needed operations :

1. 1: fetch album.

**Iterator**

The application user has the option to choose a specific event he already attended to, and display it on screen in few different viewable controls.

We chose this pattern because we wanted to give the option to see which viewable controls we have in our controls repository.

Using builder in our program:

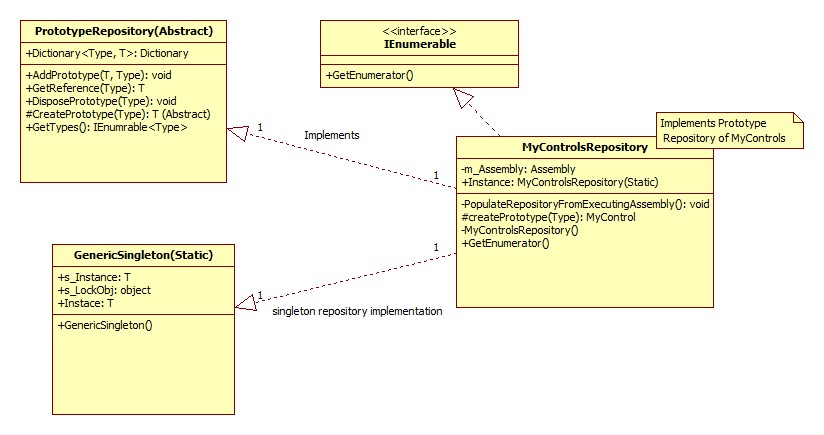
Our implementation of this pattern is to implement the interface IEnumerable and using yield statement to allow us to write iterators "inline" in a single method, with the compiler doing all the hard work of keeping track of state behind the scenes.

In our case, when the client chooses to see an event he already attended to, a comboBox will show two different viewable components: Label and ProgressBar.

These viewable components are added to a repository during runtime through Reflection.

the client now have the option to see which vieable controls are in the repository through the iterator

**Iterator Class Diagram**



**Iterator Sequence Diagram**

