Dan Inon – 314966482

Valery Geber - 208417568

New Features Explained

Game of Life

**John Conway’s Game of Life:**

We will separate the explanation to the feature itself and it’s implementation:

**General:**

The Game of life is not your typical computer game. It is a cellular automation, and was invented by Cambridge mathematician John Conway.  
This game became widely known when it was mentioned in an article published by Scientific American in 1970. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

**The Controls**:

Choose a pattern by clicking on the cells, initializing the initial state of the cells to living\dying. The ‘Start’ button advances the game by several generations (each new generation corresponding to one iteration of the rules).

**Classes\Code**

The game consists out of three classes:

1. GameOfLifeForm: the visual container of the game, contains a visual table with a random Facebook friend picture as the background, and buttons as nodes which changes appearance by the logical game engine’s board which we’ll talk about later. The visual table node’s can either be transparent or anti-transparent(black), the more transparent cells – the easier it will be to identify the mystery friend.  
   living cells will be portrayed as transparent while the dead cells are displayed as the other. When the app is open all the cells are dead, and by pressing them you can turn them alive.  
   The form also contains ‘start\stop’, ’reset’ and ‘next’ buttons which apply the logic stated above. The form also contains a timer which acts at this version of the game every second to bring the next generation from the engine by the game’s algorithm and then the form updates it visually.
2. GameEngine: The game engine contains the logical part of the game and the GameBoard(class) logical data.
3. GameBoard: This class contains a boolean matrix which for a true value the node at the GameOfLifeForm will display as transparent and for a false one will be displayed as the other. It also contains some elementary game board actions such as clear board.

Log Manager

This is a feature that tracks the user’s activity in our app.

It provides a documentation of most actions made by the user, sorted by the time the actions were made.

Some of the action types that are monitored by the app are: logging into the app, watching profile of a friend, playing “Game of life” and adding a new post.

In addition, this feature includes a graph-like representation of the actions, which are filtered by the action type.

In order to watch this information, click the “Settings” button in the user’s profile toolbar.

The classes that are associated with this feature are: **SettingsForm** - this is the form that show the data to the user, **MyFacebookService** which holds an in instance of **LogManager** that saves the actions made by the user in a collection, and provides the statistics numbers.

Pattern number 1 - Builder

**Reason of choice and Usage**

**Choice:**

We’ve chosen to implement the builder design pattern on our GameOfLifeForm.

1. The GameOfLifeForm is a complex object which aggerigates sub-systems such as GameBoard, GameEngine, UI and more aggerigated information, the builder pattern simplifies the construction of the game-form.
2. The game can be constructed with different sizes of game-board or with different background pictures, inviting the usage of different builders for our game composer in the upcoming future.
3. The Compositor was also added, so if in the future we would like to implement the game to a different platform, the compositor will still be usable. And just a builder will be required for the implementation.

**Usage:**

\*For references, see implementations below.

The client creates the composer via a static method Create.

The client fills the composer with data.

The client can seek missing information from a method in the interface.

Once the user wants the build a game-form, he will use the composer’s method – Compose to try and compose the game-form, which will throw an exception in-case some property – fields are missing.

An instance of a compositor aggerigates an instance of a builder, thus, the client may only work with the composer.

**Classes\Interfaces implementing the pattern:**

Client – UserForm

Composer – IgameComposer

ConcreteComposer – MyGameComposer : IGameComposer

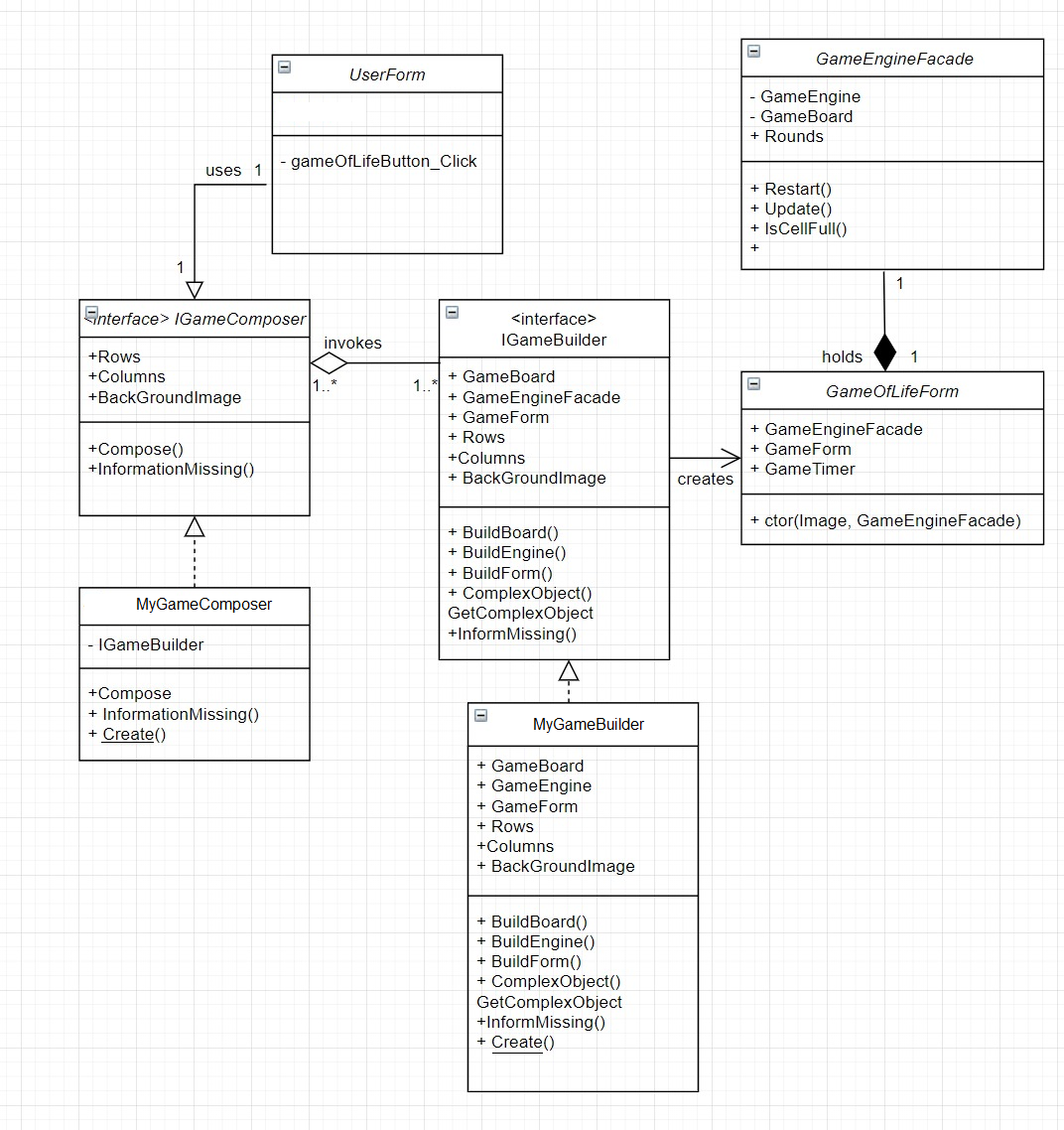
Builder – IgameBuilder

ConcreteBuilder – MyGameBuilder : IGameBuilder

ComplexObject - GameOfLifeForm

The code can be found in the GameOfLife directory.

Class Diagram:



Sequence Diagram:

Diagram

Description automatically generated

Pattern number 2 - Façade

**Reason of choice and Usage**

**Choice:**

We chose this pattern in order to simplify the usage of GameEngine class.

Game Engine is a complex class that holds the logic of the game. In the future, if we will choose to add more functionality to our game, (for example- game speakers, similarly to GameBoard) the interface of Game Engine will be hard to work with for the client.

The Game Engine Façade we created provides a simpler interface for the client, with more understandable function names. For example, the function UpdateToNextGeneration() is now called Update().

Moreover, the facade hides the complexity of the original Game Engine.

**Usage:**

The GameEngineFacade is created by the GameBuilder, and it is a class member of GameOfLifeForm.

In the constructor of GameOfLifeForm, we pass an instance of GameEngineFacade.

The Game Engine Façade holds a member of type GameEngine and we use it in all the functions that we expose to the client. In addition, the façade holds a Board property in order to provide for the client the board properties and functions, and a property called “Rounds” that holds the GameEngine’s property under the same name (because the GameEngine is a private member, the client cannot access its property directly).

**Classes\Interfaces implementing the pattern:**

Client – GameOfLifeForm

Façade – GameEngineFacade

SubSystem1 – GameEngine

Class Diagram:

Diagram

Description automatically generated with low confidence

Sequence Diagram:

Diagram, schematic

Description automatically generated

Pattern number 3 - Singleton

**Reason of choice and Usage**

**Choice:**

We’ve chosen to implement the Singleton design pattern on our LogManager feature.

Logged actions are common in many of our Facebook actions, somewhat global.  
The user’s actions are unique on the timeline, in a way no two logged actions can be gotten at once, thus, only one unique instance is required. For example, the logger can log a post, editing settings, viewing another user or playing a game.

The Singleton accesses a single list of Facebook actions which are required to be safe with threads since working with a list of data, thus, a safe implementation was chosen.

**Usage:**

\*For references, see implementations below.

Every method of every class which uses the LogManager Singelton asks for an instance by calling a static method named Instance.  
The client then is able to create a new facebook action and add it to the actionlist of the LogManager.

**Classes\Interfaces implementing the pattern:**

Client – Classes\Interfaces which support FaceBook actions listed on the LogManager.

Singleton – LogManager

Class Diagram:

Diagram

Description automatically generated

Sequence Diagram:

Diagram

Description automatically generated

**Asynchronous programming:**

At the moment the user chooses an album from the albums list, an AlbumForm is opened.

If the user chose an album with many pictures, it would take a while for the form to open and until it loads all the pictures – the system is “frozen”.

Therefore, we chose to load the albums photos in a different thread, in order to cancel the “frozen” effect of the system and the feeling that it is stuck.

We did this by creating a new thread and passing the MyInitializeComponent() method, that gets the album as a parameter and creates picture boxes for its photos and places them in the form.

In order to do this, we had to move this thread to the Album\_OnLoad event, because leaving it in the constructor is bad practice.

**Data Binding:**

We chose to do the data binding in the User Form, on the listbox of liked pages.

We define the pageBindingSource DataSource in the loadLikedPages() method.