### **ארכיטקטורת האפליקצייה**

נתאר להלן את 2 הפיצרים שבחרנו לממש,

כולל **UseCase Diagrams**.

ואת 6 התבניות שהשתמשנו בהן – design patterns.

(כולל **Sequence and Calss Diagram**).

### **תיאור קצר של הפיצ'רים שבחרנו:**

* **The Twelve Zodiac :**

פיצ'ר זה מאפשר למשתמש למצוא "מאצים" המתבססים על שלוש קטגוריות: מזל (אסטורולוגי), מין וטווח גילאים.

לשם מימוש הפיצ'ר, יצרנו בדף הבית (FormMainApp) של האפליקציה TAB ובו האופציה ללחוץ על כפתור START ולהתחיל במשחק. לאחר הלחיצה - ייפתח טופס חדש (FormTheTwelveZodiac) ובו יוצגו למשתמש השאלות השונות ואופציות לבחירה המועדפת עליו.

בסוף המשחק, תוצג למשתמש האופציה ללחוץ על כפתור - "הצג את התוצאות" ולאחר הלחיצה יוצגו החברים המתאימים, לכל שלוש הקטגרויות, על פי בחירתו של המשתמש. למשתמש יופיע כפתור למעבר ל"מאצ" הבא.

בסוף המשחק תוצג הודעה מתאימה בהתאם להאם היו התאמות/ לא היו כלל/ נגמרו ההתאמות.

ניתן למצוא את השכבה הלוגית של הפיצ'ר תחת הAppLogic.dll במחלקה LogicTheTwelveZodiac.

* **Friends Trivia Game :**

פיצ'ר זה מאפשר למשתמש לשחק משחק טריוויה הבוחן את ההיכרות שלו עם רשימת החברים שלו בפייסבוק.

לשם מימוש הפיצ'ר, יצרנו TAB בדף הבית של האפליקציה ובו האפשרות ללחוץ על כפתור "התחל" ולהתחיל את המשחק. לאחר הלחיצה על הכפתור יפתח חלון המשחק (FormTriviaGame), יופיעו למשתמש 5 שאלות, אחת אחרי השנייה, על אחד מחבריו.

למשתמש תוצג תמונה של החבר הנשאל, החבר ייבחר באופן אקראי מתוך רשימת החברים.

כמו כן, תבחר שאלה אקראית מתוך מאגר השאלות ותשובות אקראיות מתוך מאגר התשובות. כאשר תמיד תשובה אחת תהיה התשובה הנכונה.

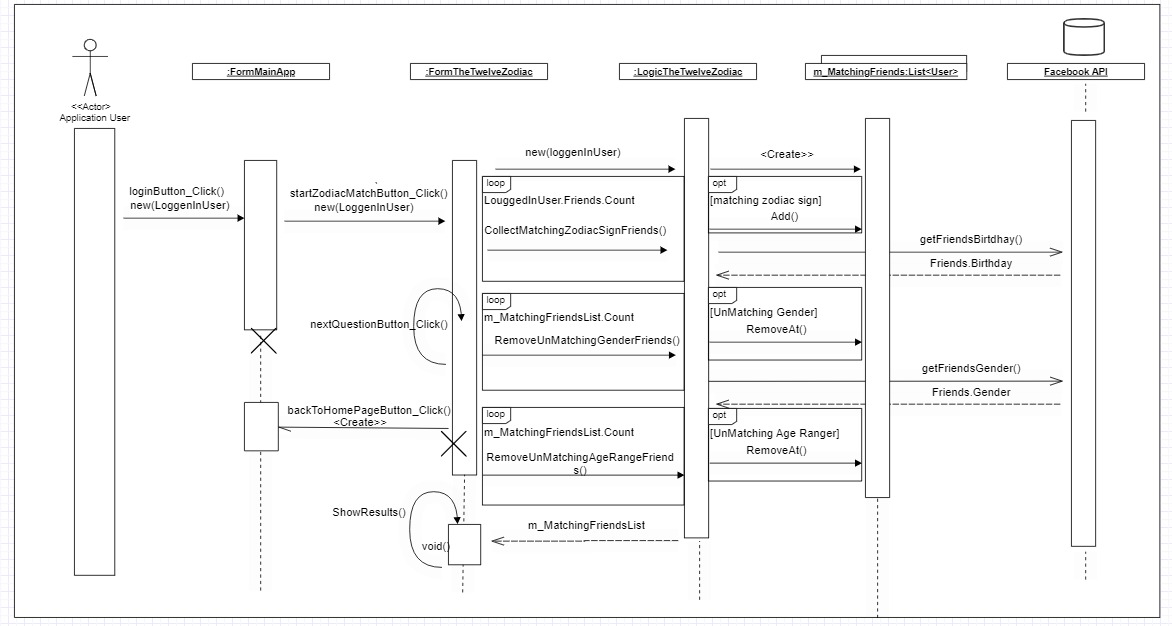
לאורך המשחק יופיע על המסך הניקוד הנוכחי. בסופו של המשחק תוצג הודעה המציגה את תוצאות המשתמש והניקוד שלו.

ניתן למצוא את השכבה הלוגית של הפיצ'ר תחת הAppLogic.dll במחלקה LogicTriviaGame.

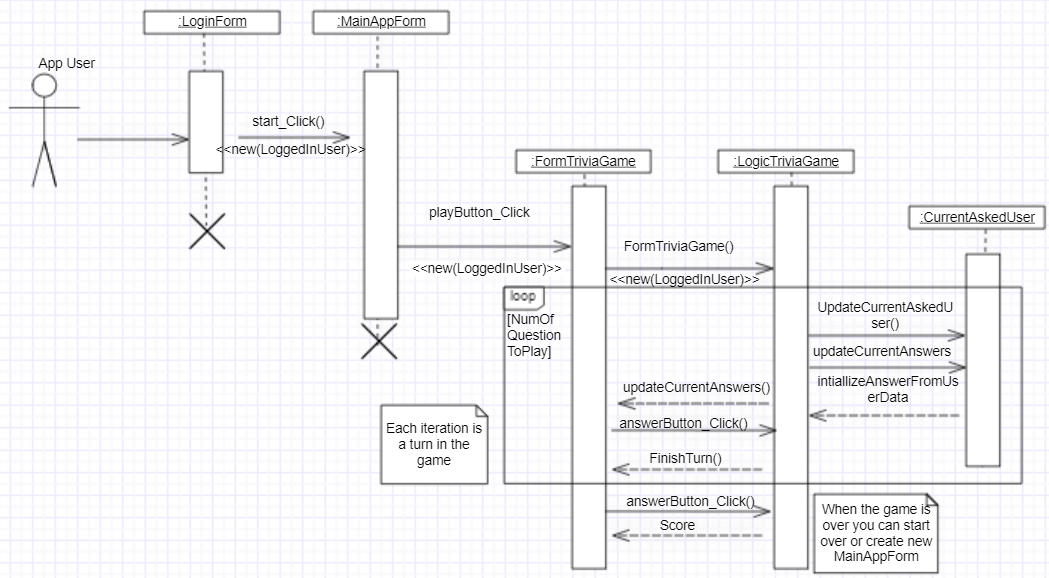
\*בכל שלב, בשני הפיצ'רים, יוכל המשתמש לחזור לדף הבית ולצאת מן המשחק..

* **תמונה שמכילה טקסט, מפה

  התיאור נוצר באופן אוטומטיUseCase Diagrams:**
* **Sequence Diagrams:**

**The Twelve Zodiac**

**Friends Trivia game**



### **תבנית מס' 1 – Singelton**

* סיבת הבחירה / שימוש בתבנית:

בחרנו לממש את LoggedInUser, כסינגלטון ממספר סיבות:

אנו מעוניינים לאפשר בכל זמן LoggedInUser יחיד ולא יותר.

אם בעתיד נרצה להוסיף אובייקט למערכת ונרצה שיקבל מופע של ה- LoggedInUser האובייקט יוכל לקבל את המופע היחיד שנוצר ממנו בלי צורך בשינוי של קוד המחלקה.

לכן בעזרת שימוש בתבנית סינגלטון נוכל לקבל פתרון שמאפשר LoggedInUser אחד עבור כל המופעים שלו, ובנוסף אפשרות לקבל מופע ללא שינוי בקוד המחלקה.

אופן המימוש:

מימשנו את המחלקה Singleton<T> בתצורה גנרית ומיקמנו אותה בקובץ dll נפרד, כדי לאפשר שימוש חוזר בתבנית אם נהיה מעוניינים בה עבור מחלקות נוספות.

מימשנו בנוסף את המחלקה SingletonLoggedInUser תחת הקובץ המאגד את כל המחלקות הלוגיות של האפליקציה (AppLogic) אשר 'אורזת' תחתיה את פונקציות החיבור והניתוק מהמערכת ובנוסף את ה- LoggedInUser.

השתמשנו בה מהמחלקה SingletonLoggedInUser על מנת שתהיה סנגלטון ותאפשר מופע יחיד של LoggedInUser לכל אובייקט שיירצה לקבל כזה.

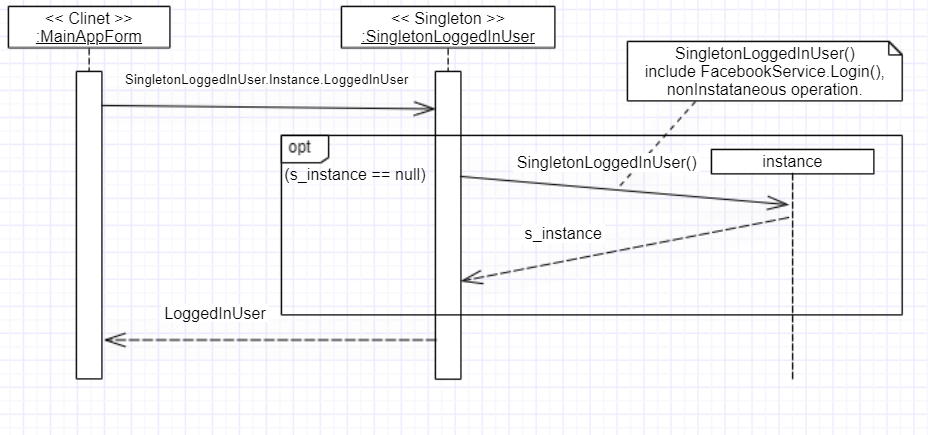
ובכך כל אחד מהאובייקטים שצריכים LoggedInUser פונים אל SingletonLoggedInUser באופן הבא : AppLogic.SingletonLoggedInUser.Instance.LoggedInUser

כאשר בפנייה הראשונה יופעל הקונסטרקטור של SingletonLoggedInUser ותתבצענה הפעולהlogin() כדי לקבל את ה LoggedInUserמכן כל פנייה תחזיר מופע שלו.

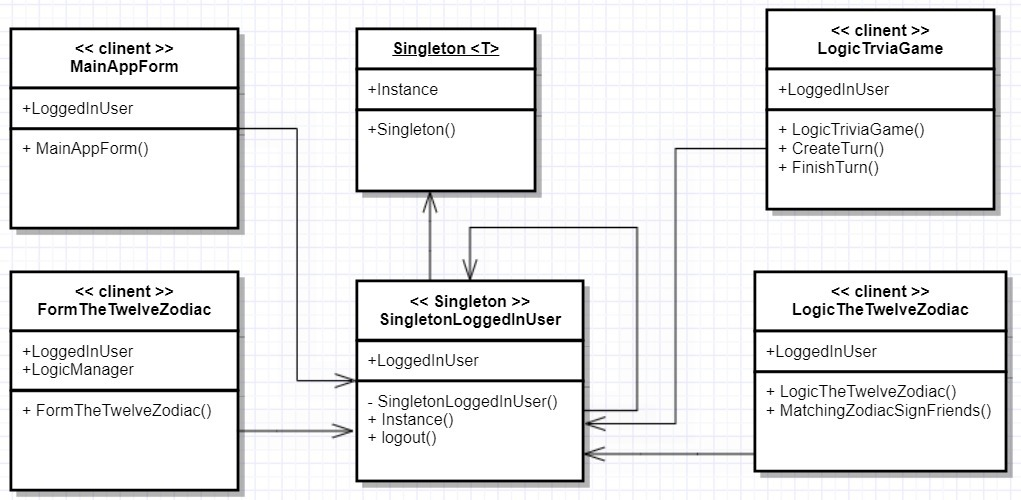
**השחקנים בתבנית:**

1. SingletonLoggedInUser **:Singleton**
2. FormTheTwelveZodiac, LogicTheTwelveZodiac, LogicTriviaGame, **:Client** MainAppForm

* Sequence Diagram



* Class Diagram



### **תבנית מס' 2 – facade**

* סיבת הבחירה / שימוש בתבנית:

פיצ'ר מספר 1 הוא פיצ'ר המורכב מטופס ראשי וממספר תתי לוגיקות, כאשר הלוגיקות, כל אחת בהתאם לצריכה, אוספות נתונים מהטופס (על פי המשתמש) ומפעילות את עצמה בהתאם. לכן, בחרנו להשתמש בתבנית זו וליצור רכיב אטום לכל המשתמש בתתי הלוגיקות.  
הרכיב מקל ומפשט את העבודה מול תתי הלוגיקות, כך שמאחורי הקלעים נמצאים שירותים מסובכים ומורכבים יותר, ואילו המשתמש (טופס המשחק במקרה זה) חשוף אך ורק לרכיב אחד פשוט וקל לתפעול.  
בנוסף, במידה ובעתיד נרצה למסור את הלוגיקה של הפיצ'ר לUI אחר, הרכיב מאפשר לנו חיבור נוח ומיידי בין השניים. כמו כן, אם נרצה להוסיף בעתיד פיצ'רים נוספים, ו/או שכלול הקיימים, הרכיב יוכל לשרת גם את הצורך הזה בצורה הטובה ביותר.

אופן המימוש:

* לצורך מימוש התבנית ביצענו את הדברים הבאים:  
  FormTheTwelveZodiac : **client** - נמצא בקוד בתיקיית הUI בשם C20\_Ex02 משמש כUI של הפיצ'ר, אוסף את הנתונים מהמשתמש ומוסר אותם לFacade . LogicManager **: Facade** - נמצא בקוד בתיקייה .AppLogic מחלקה זו יוצרת הפרדה מלאה בין הממשק ובין החלקים הלוגים לצורך מימוש הפיצ'ר. המחלקה מחזיקה בתוכה מופע של המחלקה הלוגית, מכילה מתודה אחת הנגישה לממשק, מתודה זו אחראית להפעלת כל התהליכים הלוגים הנדרשים.  
   LogicTheTwelveZodiac **: subSystemA** - נמצא בקוד בתיקייה AppLogic . מחלקה internal החשופה ל Facade בלבד. מחלקה זו מממשת את החלקים הלוגים השונים וחושפת מתודות internal לFacade על מנת שיוכל לדאוג להפעלתן ולהעביר להן את המידע הדרוש להן מהממשק.  
    
   במהלך הרצת האפליקציה, לאחר שהמשתמש מסיים לענות על השאלות הרלוונטיות, הממשק פונה לfacade , מוסר לו את הנתונים שהתקבלו מהמשתמש ומפעיל את המתודה FindMatchingFriendsList(). המתודה נמצאת בFacade ומפעילה את שלושת תתי הלוגיקות אל מול המחלקה הלוגית. בסיום, המחלקה הלוגית מוסרת לFacade את רשימת ה"מאצים" הסופית, וזו מוסרת אותה לממשק לצורך הצגת החברים.
* תמונה שמכילה טקסט, מפה

  התיאור נוצר באופן אוטומטיSequence Diagram
* Class Diagram

תמונה שמכילה צילום מסך

התיאור נוצר באופן אוטומטי

### **תבנית מס' 3 – Proxy**

* סיבת הבחירה / שימוש בתבנית:

זיהינו שלעיתים כאשר משתמש מקבל את תמונות המאצ'ים שהפיצ'ר מצא לו, תמונת הפרופיל המוצגת מעט מרוחקת ולפעמים אפילו איננה במנח ברור (כלומר לא כל התמונות יהיו 'מיושרות כלפי מעלה' ). רצינו לאפשר ליוזר המשתמש באפליקצייה לבצע הגדלה או סיבוב של התמונה בצורה קלה ומהירה.

כוון שהממשק הקיים בו הוצגו התמונות (PictureBox) לא איפשר זאת בחרנו להשתמש בתבנית ובכך להרחיב ולשפר את היכולות הניתנות למשתמש באפליקצייה בכל הנוגע למופעים בהם מוצגת תמונה בטופס.

לכן מימשנו עבור המשתמש את הפעולות הבאות:

1. הגדלת התמונה – בעזרת העברת האכבר על התמונה.
2. סיבוב התמונה ב90 מעלות בעזרת לחיצה על התמונה.

בכך שיפרנו את חווית המשתמש ואיפשרנו מענה על הצורך שזיהינו.

* אופן המימוש:

באפליקציה שני רכיבים (ויותר) שהם "leagacy components" - רכיב מסוג form ורכיב נוסף מסוג picture box המקיימים בינהם ממשק תקין שאותו ברצוננו לשכלל.

יצרנו מחלקה בשם PicturesBoxAdditionalsProxy היורשת מPictureBox. הראשונה בעצם "מתחזה" לשנייה ומממשת בתוכה את הלוגיקה הנדרשת לשיפור הממשק תוך דריסה של המתודות הבאות:

- OnMouseEnter - הגדלת התמונה.

- OnMouseLeave - החזרת התמונה לגודלה המקורי.

- OnMouseClick - כל לחיצה מאפשרת סיבוב של 90 מעלות נגד כיוון השעון.

הטפסים FormTheTwelveZodiac, FormMainApp משתמשים ברכיב הProxy שנוצר ובכך מאפשרים למשתמש ממשק משופר.

**בתבנית משתתפים:**

FormMainApp, FormTheTweleveZodiac **- clients**

IControl - **ISubject**

PictureBox **- real subject**

PicturesBoxAdditionalsProxy **- proxy**

* תמונה שמכילה טקסט, מפה

  התיאור נוצר באופן אוטומטיSequence Diagram
* Class Diagram

תמונה שמכילה טקסט, מפה

התיאור נוצר באופן אוטומטי

### **תבנית מס' 4 – Visitor**

* סיבת הבחירה / שימוש בתבנית:

רצינו לממש תגובה לתשובה נכונה במהלך משחק הטרוויה, על מנת לשפר את חווית המשתמש.

התגובה אותה רצינו לממש היא רעד של תמונת החבר עליו נשאלת השאלה, אשר מופיע במרכז הטופס בעת הצגת השאלה.

בנוסף רצינו לממש תגובה דומה עבור לחיצה של המשתמש על הודעת ה"welcome" במסך הבית של האפליקצייה, כך בכל פעם שהמשתמש יילחץ עלייה היא תרעד.

חשבנו שהדבר יכול להוסיף עניין למשתמש בזמן שהוא נמצא במסך הבית ולשפר את חווית השימוש באפליקצייה.

זיהינו שמדובר בשיכפול קוד ותחזוקה של פעולת הרעד של האובייקט בשתי המחלקות שאנו מעוניינים לאפשר להן לבצע את הפעולה. בנוסף ידוע לנו שאם אנו רוצים להוציא את הקוד של הפעולה למחלקה ניפרדת, הדבר עלול לגרום להיכרות ציקלית כוון שהפעולה משנה את מצב האובייקט הנוכחי – במקרה שלנו את מיקומו. לכן בחרנו לממש זאת באמצעות תבנית visitor ובכך יתאפשר לנו להוציא את הקוד שאחראי לפעולה למחלקה נפרדת ולמנוע שכפול קוד ותחזוקה בין המחלקות ובנוסף כתוצאה מהשימוש בתבנית אנו נמנעים גם מהיכרות ציקלית בין המחלקה שאחראית על הפעולה למחלקות שעליהם הפעולה מתבצעת.

אופן המימוש:

תחילה זיהינו שלשני האובייקטים יש אב משותף Control ועל כן ניתן לעבוד איתם בצורה פולימורפית ללא מימוש ממשק משותף.

מימשנו את המחלקה ControlShaker בקובץ dll נפרד על מנת לאפשר בו שימוש חוזר.

למחלקה מימשנו מתודה Shake(Control i\_ControlToStretch) אשר מקבלת אובייקט מסוג Control ו"מנערת" אותו כפי שרצינו.

עבור המחלקות PictureBox, Lable ממשנו את המחלקות ActivePictureBox, ActiveLable בהתאמה כך שעבור כל אחת מהם, מדובר במחלקה שיורשת ממחלקת המקור ובנוסף מחזיקה בקומפוזיציה את הויזיטור ControlShaker.

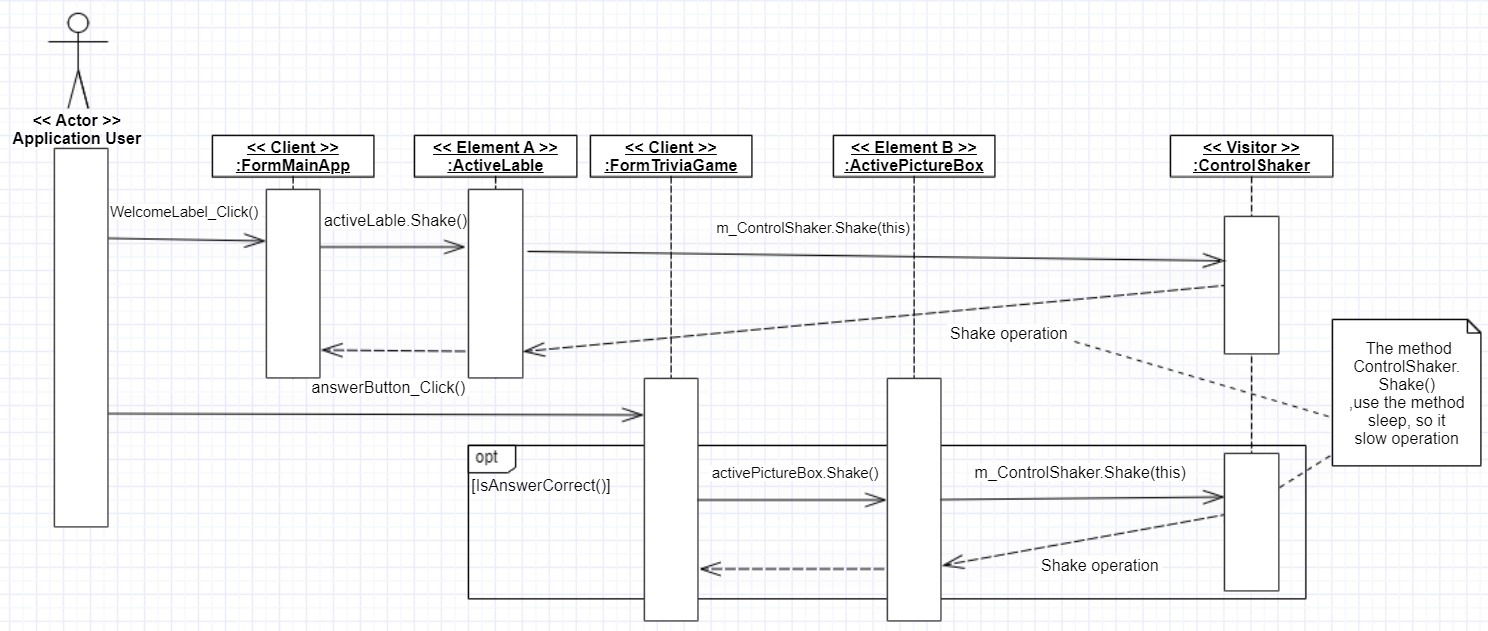
וכך במחלקות ActivePictureBox, ActiveLabl מימשנו מתודת Shake() אשר לא מקבלת פרמטרים ומפעילה את פונקציית ה"ניעור" מה"משקשק" שהיא מחזיקה בקומפוזיציה,

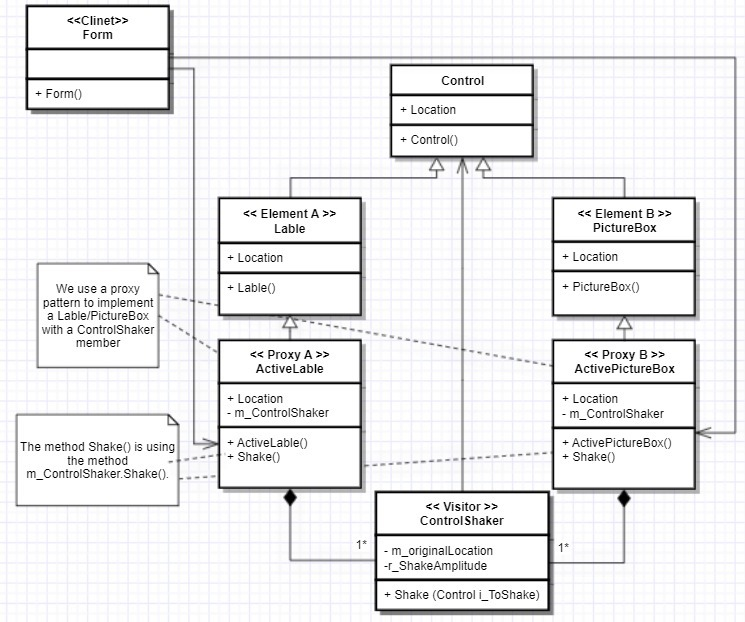
כלומר את הפונקציה m\_ControlShaker.Shake(this) .

כך שיפרנו גם את הקריאות, ואיפשרנו להפעיל את פונקציית הרעד באופן שקוף למשתמש.

**השחקנים בתבנית:**

1. ControlShaker **:Visitor** נמצא תחת – ControlShaker.dll
2. **:Element A** ActiveLable נמצא תחת – AppUI.dll
3. **:Element B** ActivePictureBox נמצא תחת – AppUI.dll
4. **:Clinet** FormMainApp,FormTriviaGame נמצא תחת – AppUI.dll

* Sequence Diagram
* Class Diagram



### **תבנית מס' 5 – Strategy**

* סיבת הבחירה / שימוש בתבנית:

ייחודו של פיצ'ר מספר אחד, הוא היכולת למצוא התאמות, לפי רצון המשתמש המתבססות ונוגעות בנושא האסטרולוגיה. הפיצ'ר איך שהוא ממומש כעת, מאפשר למשתמש לבחור התאמה על בסיס המזל בו הוא מעוניין. בחרנו להשתמש בתבנית זו, מכיוון שהיא מאפשרת לנו לשים משפחה של אלגוריתמים, הנוגעים בפיצ'ר מסויים (במקרה זה - מזל מועדף) תחת קורת גג אחת ונותנת לנו את היכולת לשנות את אופי הפיצ'ר, כלומר את האסטרטגיה לפיה נבחרת ההתאמה, באופן המאפשר לנו לעשות זאת מבלי לשכפל קוד, לאפשר תחזוקה קלה ויעילה יותר, להקטין אופציה ליצירת באגים ומבלי לשנות קוד במחלקות העובדות עם הפיצ'ר.

בגלל שרצינו לאפשר שינוי עתידי של אופציית סינון ההתאמות, בעולם האסטרולוגיה שהוא עולם רחב מאוד, ולאפשר למשל מנגנון הבוחר התאמה לפי המזל "האידאלי" ביותר עבור המזל של המשתמש/מזל נפוץ ועוד, נוכל לעשות זאת בקלות באמצעות התבנית ושינוי האסטרטגיה בתוך הפיצ'ר.

* אופן המימוש:

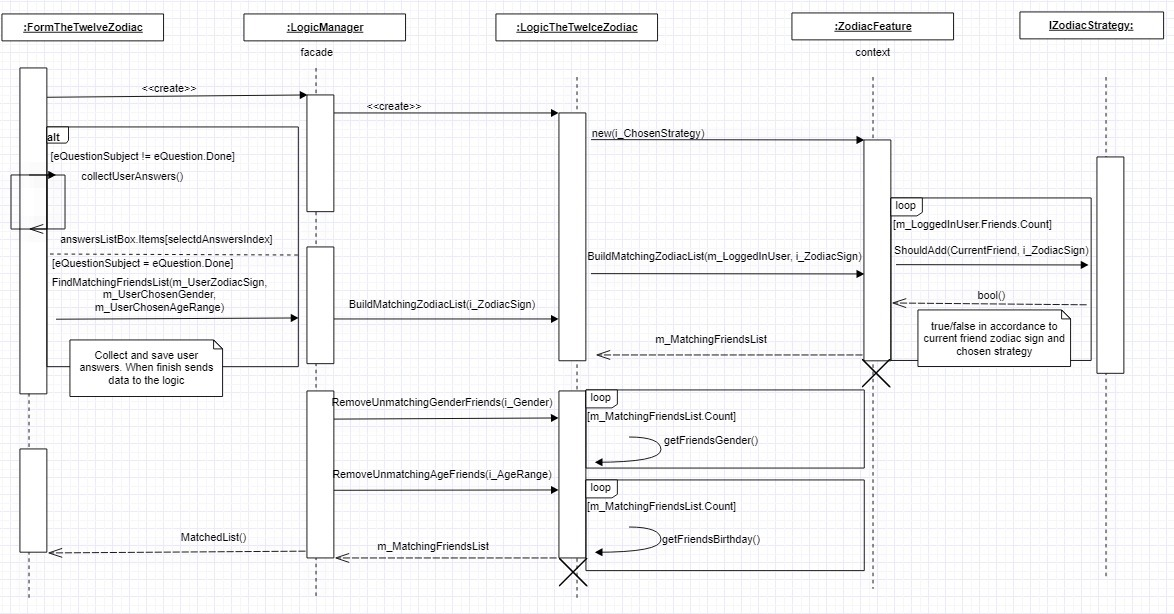
ראשית, יצרנו אינטרפייס בשם IZodiacStrategy ובו מתודה בוליאנית בשם ()ShouldAdd. המחלקה ZodiacFeture היא המחלקה שבעצם משמשת כ"מכניזם" והיא מכילה את כל הקשור במימוש הפיצ'ר. כמו כן, היא מחזיקה בתוכה property מסוג IZodiacStrategy - דבר המאפשר לה לממש את הלוגיקה העוטפת הדרושה לה, ולפנות לאסטרטגיה הנוכחית, באופן פולימורפי, ולקבל תשובה מהאסטרטגיה האם עליה להוסיף את החבר הנוכחי לרשימה שלה או שלא.

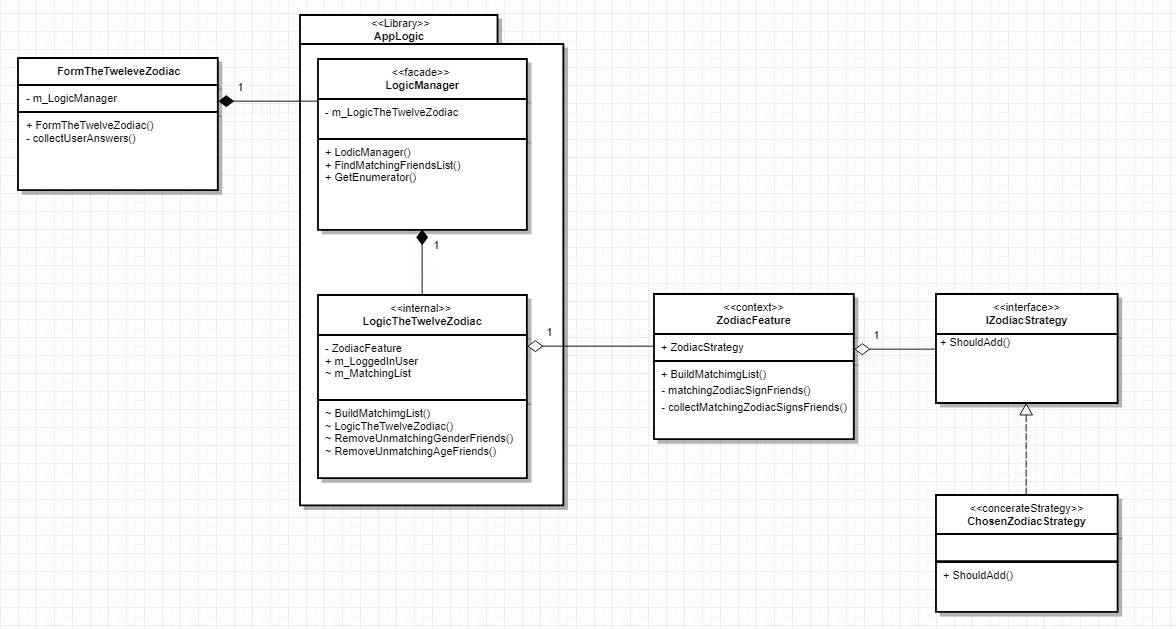
כמו כן, יצרנו את המחלקה ChosenZodiacStrategy, מחלקה זו ממשת את האינטרפייס ואת המתודה בו בהתאם לנושא הפיצ'ר הנוכחי - התאמה לפי מזל נבחר.

המחלקה הלוגית של האפליקציה כולה, מחזיקה בתוכה property מסוג ZodiacFeture ומעבירה לו את האסטרטגיה הנבחרת במהלך הרצת התכנית. כמו כן, יש לה את היכולת למסור ל ZodiacFeture בעתיד אתחולים של אסטרטגיות שונות, על מנת שיפעל על פיהן.

**השחקנים בתבנית:**

1. ZodiacFeature **: Context** נמצא תחת – AppLogic.dll
2. **: Strategy** IZodiacStrategy נמצא תחת – AppLogic.dll
3. **: ConcreateStrategy** ChosenZodiacStrategy נמצא תחת – AppLogic.dll

* Sequence Diagram
* Class Diagram



### **תבנית מס' 6 – Iterator**

* סיבת הבחירה / שימוש בתבנית:

פיצ'ר מספר 1, יוצר במהלך הרצת התכנית, רשימה של חברים המהווים "התאמות" לפי רצונותיו השונים של המשתמש. רשימה זו נוצרת על ידי הרכיב הלוגי של האפליקציה ומועברת לטופס הראשי של האפליקציה, הטופס עובר על הרשימה ומציג למשתמש את התוצאות (תמונה, שם וכו).

זיהינו צורך להשתמש בתבנית זו מכיוון שרצינו לאפשר לטופס עבודה זהה, על כל מבנה נתונים באשר הוא, מבלי שהטופס יידע באיזה מבנה נתונים הוא משתמש. זאת מכיוון, שייתכן שבעתיד נרצה לשנות את מבנה הנתונים מרשימה לכל מבנה נתונים אחר מצורכי יעילות/ נוחות/ משום שהעבודה עם הAPI השתנתה ועוד.

התבנית מאפשרת לנו תחזוקה נוחה יותר, משום שלא נציטרך לשנות את הקוד בצד הUI, כמו כן מאפשרת שימוש חוזר בלוגיקת המשחק גם עבור UI אחרים וכן בעת שינוי מבנה הנתונים הפנימי של הלוגיקה.

* אופן המימוש:

המחלקה LogicManage המשמשת facade בתכנית יצרנו מימוש של האינטרפייס

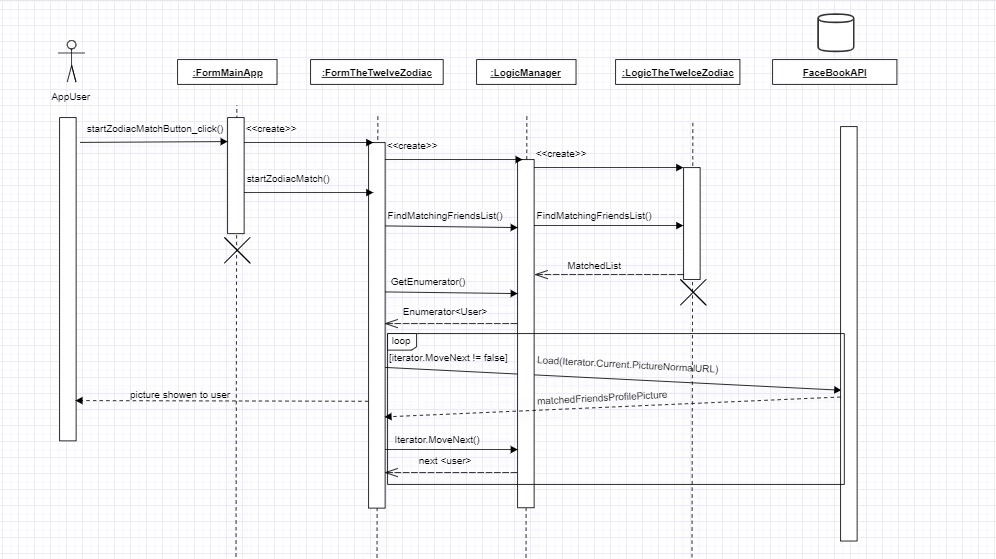
IEnumerable <User>. בתוך מחלקה זו יצרנו שתי פונקציות ()GetEnumerator. האחת מפורשת, כלומר מהצורה: ()IEnumerable.GetEnumerator המחזירה IEnumerator.

פונקציה נוספת המחזירה IEnumerator<User>, עוברת בforeach על כל איברי הרשימה ומשתמשת במגנון של השפה "yield return" המאפשר לנו מימוש נוח וקל יותר של המנגנון.

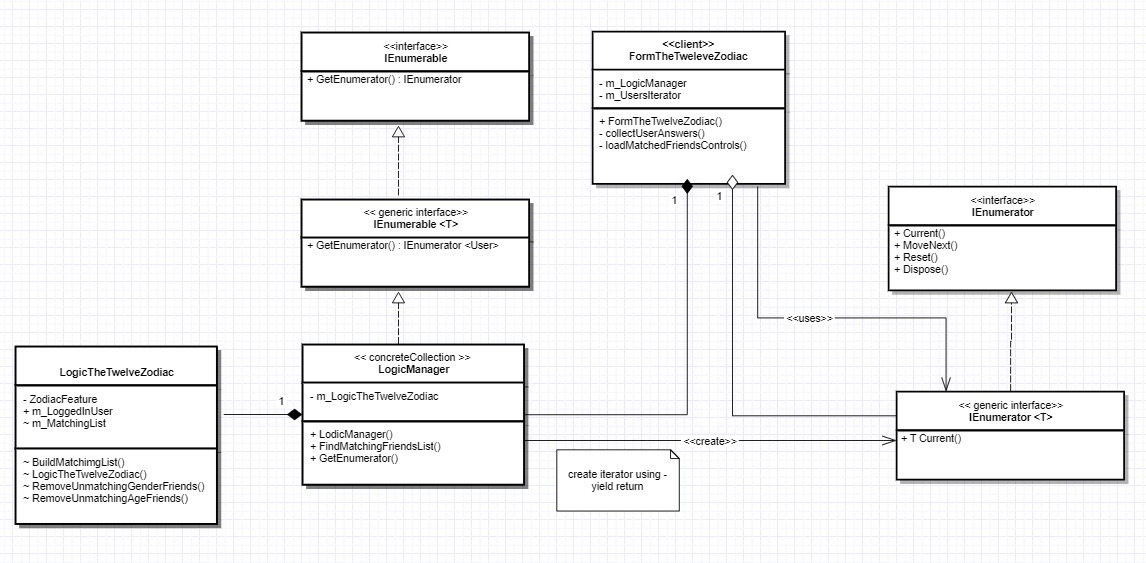
המחלקה ממנה מסתירים את מבנה הנתונים זו המחלקה FormTheTwelveZodiac והיא מחזיקה בה כproperty אובייקט מסוג IEnumerator<User>.

במהלך הרצת התכנית אובייקט זה מאותחל על ידי GetEnumerator של המחלקה LogicManager ובעצם כך ניתנת לטופס היכולת לרוץ על מבנה הנתונים מבלי להכיר אותו על ידי המתודות: ()MoveNext(), Currnt.

* Sequence Diagram



* Class Diagram



### **עבודה אסינכרונית:**

בחרנו להשתמש בעבודה אסינכרונית לאחר שנתקלנו בתקורת זמן גבוהה מאוד לטובת מימוש הפקודות אשר מזינות את ה- MainAppForm בנתונים. פקודות אלא אוספות נתונים מה LoggedInUser- ומציגות אותן למשתמש.

כוון שפקודות אלא אינן נסמכות אחת על השנייה ניתן לחלק אותן ל- Thread-ים נפרדים וכתוצאה מכך עבודתם תתבצע בצורה אסינכרונית ותאפשר פעולות אחרות במקביל.

לכן השתמשנו בתכנות אסינכרוני כדי להזין כל אחד מאוספי הנתונים באמצעות Thread נפרד. בכך אנו מאפשרים ממשק משתמש "נעים" יותר, שכן המשתמש יכול לבצע פעולות במקביל לשלב אכלוס הנתונים ובנוסף לאפשר הבאת מספר אוספים במקביל.

המתודות המתבצעות בThread-ים נפרדים הן:

1. fetchCheckinsListBox()

במחלקה MainAppForm שורה 362.

1. fetchFriendsListBox()

במחלקה MainAppForm שורה 368.

1. fetchPostsListBox()

במחלקה MainAppForm שורה 374.

### **Data Binding**

במערכת ביצענו שימוש ב Two Way Data Binding לטובת הצגה ועדכון של פרטיו האישיים של היוזר המחובר למערכת.

בעת העלאת החלון הראשי (MainAppForm), מתחת לתמונת הפרופיל של היוזר, הוספנו פאנל שבו מוצגים פרטיו האישיים של היוזר (שם מלא, תאריך לידה ומייל).

בנוסף מתאפשר ליוזר לעדכן את הנתונים המוצגים (כולל בדיקת תקינות לקלט לדוגמא לא יתאפשר להזין תאריך לידה לא תקין), וכך הנתונים יישמרו גם כאשר המשתמש יעבור לחלון אחר (לדוגמא של אחד הפיצ'רים של האפליקצייה) וכאשר יחזור יופיעו הנתונים המעודכנים.

האובייקט ששימש אותנו כ- Data Binding Sourceהוא ה-LoggedInUser והשדות שבחנו להציג ולאפשר את עדכונם בפאנל הם:

1. שם מלא.
2. תאריך לידה.
3. מייל.