### תיאור קצר של הפיצ'רים:

* **פיצ'ר ראשון – מציאת רשימת 5 החברים הכי טובים שלך על פי נתונים מהפייסבוק:**

פיצר זה מנתח את כל התגובות והלייקים שחברים של אותו משתמש רשמו או עשו לו

המערכת תשלוף את כל האלבומים,תמונות,פוסטים,ואירועים של המשתמש שמופיעים בפייסבוק.

לאחר מכן מנתחת כל אחד מהתונים שהיא שלפה ובודקת מי מהחברים הביא לייק וכתב תגובה או היה נוכח באירוע (הן לפוסט,תמונות,אלבומים,אירועים)

לאחר שקיבלנו את כל האנשים וסכמנו על ידי אלגוריתם חישוב (כל תגובה או לייק מחושב שונה על פי הקטגוריה)

הרשימה עוברת למיון ומחזירה רק את ה5 החברים שדורגו הכי גבוה (בתור הרשימה כמובן יש את פרטי המשתמש + הדירוג שלו שהוא קיבל)

לאחר שהכל בוצע הפרטים שהוחזרו ברשימה מופיעים בטבלה שמציגה את שם + שם משפחה של החבר + תמונת פרופיל שלו + כמה הציון שהוא קיבל.

הפעולות שמומוש בפיצר הזה:

\* מציאת כל החברים והצגתם בטבלה

\* מציאת כל הלייקים שחברים שלך עשו על אלבומים,תמונות,פוסטים

\* מציאת כל התגובות שחברים שלך עשו על אלבומים,תמונות,פוסטים

\* מציאת כל האירועים שאתה הייתה בהם

\* מציאת כל האירועים שחברים שלך היו בהם

\* מציאת כל האירועים שאתה וחברים שלך הייתם נוכחים באותו מקום

\* חישוב וסכימה של הציון שכל חבר קיבל

מיקום בפרויקט: קובץ UI - נקרא בשם BestFiveFriends.cs הוא נמצא בתוך תקייה Panels

קובץ שדרכו המערכת עובדת(Façade) – קובץ הלוגי היחידי שהפיצר מכיר שאיתו הוא עובד הוא נקרא FacebookApplication.cs והוא נמצא בתוך תקייה בשם Façade.

קובצי לוגיקה שמאחורי הFaçade -

\* FetchInformations - קובץ הנמצא בתוך תקייה BestFriendsFolder אחראי על שליפת המידע מהשרת ושמירתו בתוך ServerInfomartion

\* UserRate - קובץ הנמצא בתוך תקייה BestFriendsFolder אחראי על יצירת אובייקט ששומר את המידע אודות המשתמש והניקוד של אותו משתמש

קבצי מאגר מידע -

\* ServerInformation - קובץ הנמצא בתוך תקייה ServerDetails אחראי על שמירה של כל המידע שנשלף מהשרת לשימוש חוזר (למנוע בקשות מיותרות לשרת ולשיפור ביצועי מערכת)

\* LoginInformation - קובץ הנמצא בתוך תקייה ServerDetails אחראי על שמירה של כל המידע אודות ההתחברות לשרת כמו הרשאות וקוד האפליקציה , ובנוסף שומר את היוזר שהתחבר אחרון.

* **פיצ'ר שני – ניתוח לפי תאריך שהמשתמש בוחר והצגת מידע מהפייסבוק בצורה טבלאית:**

פיצ'ר זה מאפשר למשתמש לבחור כל תאריך שהוא רוצה

והמערכת תעבור על כל חבריו של המשתמש ותחפש אם אחד מחבריו של המשתמש עשה  
פעולה מסוימת (העלה תמונה/פוסט/חוגג יומהולדת) בתאריך הנבחר לפי יום וחודש (לא לפי אותה שנה).  
  
הפעולות שמומשו והפיצ'ר מזהה אותם והם מוצגות בצורה טבלאית הם:

* מציג את כל החברים של המשתמש אשר חוגגים יום הולדת בתאריך שהמשתמש בחר
* מציג את כל החברים של המשתמש אשר בתוך אחד מהאלבומים שלהם נמצאת תמונה שהועלתה בתאריך שהמשתמש בחר
* מציג את כל החברים של המשתמש אשר העלו פוסט בתאריך שהמשתמש בחר

מיקום בקוד: קובץ SmartCalendar.cs שנמצא בתוך התיקייה Panels

החלק הלוגי נמצא בעיקר תחת תיקיית Façade בקובץ FacebookApplication.cs)

### תבנית מס' 1 – Command

* סיבת הבחירה / שימוש בתבנית:

הסיבה בגללה בחרנו לממש Pattern Command במערכת שלנו נבעה   
מכך שנתקלנו בבעיה הנדסית שהיא אומרת שאם בעתיד נרצה לבצע פעולות נוספות שיופיעו אצלנו בCombobox אנחנו נצטרך לשנות בכמה מקומות שונים בקוד דבר שיקשה עלינו מבחינת תחזוקתיות, קריאות, ופוטנציאל לבאגים, כלומר אנחנו מחפשים פיתרון הנדסי אשר תפתור לנו את הבעיה, לכן בחרנו ב command שזה מה שהוא בדיוק פותר , הוא מציע אפשרות לייצירת תפריט פעולות שאם בעתיד נרצה להוסיף או להסיר פעולה כל מה שנצטרך זה לשנות במקום אחד בלבד והוא יעשה זאת לבד אוטומטית.

בPanel SmartCalendar יש לנו תפריט בחירה המוצג למשתמש ב ComboBox בשם comboBoxCalendarSelection רצינו לשפר את התפריט.

הבנו שיש לנו פה בעיה הנדסית שבמידה ובעתיד נרצה לעדכן את התפריט נצטרך לבצע כמה שינויים בקוד ורצינו לייצר מצב שכדי לעדכן את התפריט נצטרך להוסיף שורת קוד אחת ופונקציה שעושה את הפעולה של ה MenuItem החדש שהוספנו.  
כך כשנרצה לשנות את התפריט שלנו נוכל למנוע באגים שיכולים לקרות עקב כך כי שכחנו להוסיף שורה בעוד מקום שהיינו צריכים לשנות אותה כדי שהפונקציה שרצינו שתפעל תעבוד.  
התבנית מצמצמת משמעותית את הזמן שייקח למתכנת עתידי לעדכן את התפריט, האופציה לבאג שנובעת מעדכון התפריט בכמה מקומות, תחזוקתיות של המערכת,   
הקוד יותר קריא ומסודר.   
\* עשינו שימוש ב Databinding כדי לקשר בין התפריט שייצרנו לבין ה comboBoxCalendarSelection שנמצא בתיקיית Panel ב SmartCalendar

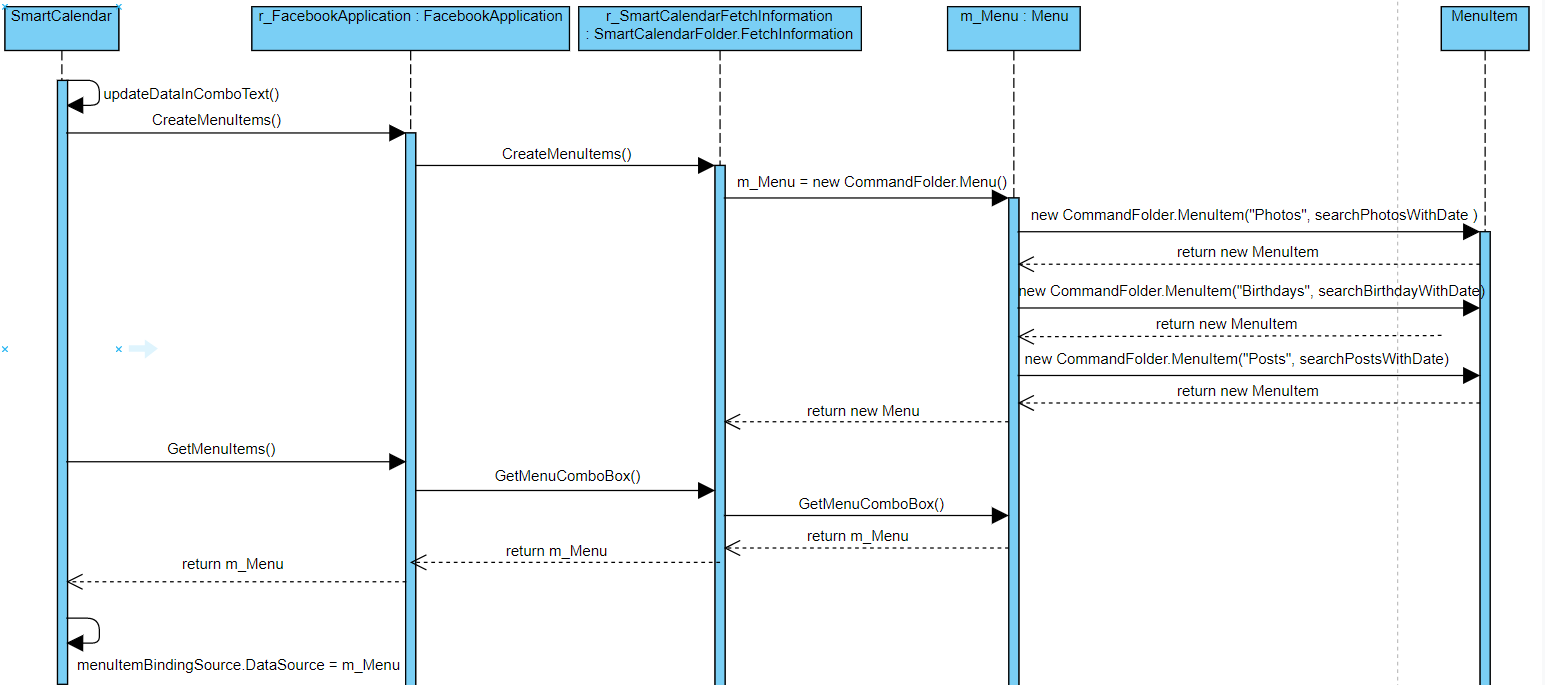
* אופן המימוש:

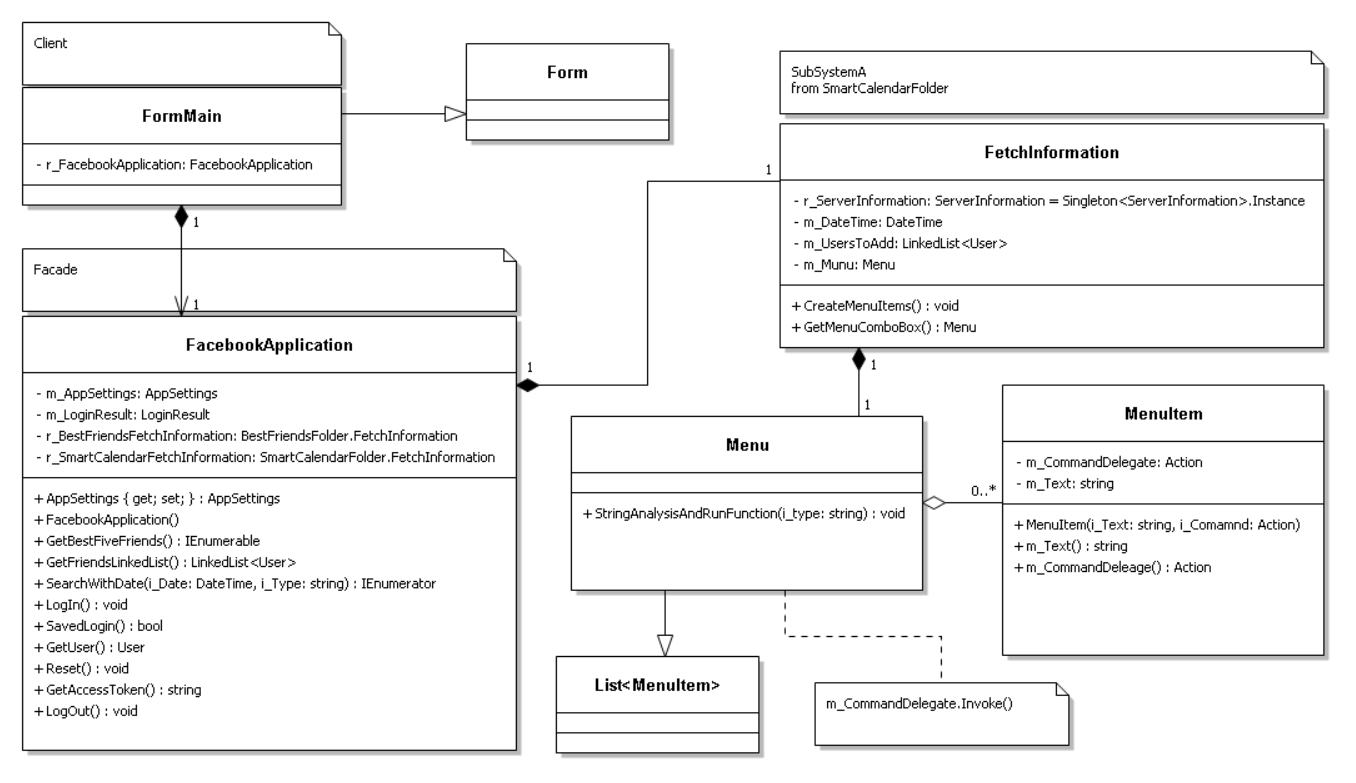
הוספנו תיקייה ששמה CommandFolder ובתוכה שני מחלקות :  
מחלקה ראשונה "MenuItem" אחראית על המבנה של האובייקטים שנרצה להוסיף לMenu,   
שיש בכל אחד מהם string - השם שאנחנו נרצה להציג ב comboBoxCalendarSelection למשתמש.

ו reference - של מתודה שתופעל ברגע שהמשתמש ייבחר בה.

מחלקה שנייה "Menu" מחלקה שיורשת מ List<MenuItem> אשר כל האובייקטים של המחלקה MenuItem נמצאים בה ברשימה/תפריט.  
ב Menu ישנה מתודה בשם StringAnalysisAndRunFunction(string i\_Type)   
אשר עוברת על כל הMenuItem שקיימים בתפריט ומפעילה את המתודה המתאימה לstring שהיא קיבלה כשאר היא מצאה את המתודה שהיא צריכה להפעיל היא בודקת אם המתודה לא null ובמידה והיא לא null היא מפעילה את המתודה המתאימה.  
בפעם הראשונה שמוצג החלון של SmartCalendar נקראת המתודה updateDataInComboText() אשר נמצאת ב SmartCalendar בתוך תיקיית Panel   
ועושה שתי פעולות :

יוצרת את הMenu על ידי שימוש בFacade שלנו.  
ומעדכנת את הDataSource של menuItemBindingSource אשר מציגה למשתמש את הפעולות שבהם הוא יכול להשתמש במערכת שלנו שמופיעות בתפריט מסודר שמי שאחראי להציג אותו הוא : comboBoxCalendarSelection שנמצא בתיקיית Panel ב SmartCalendar

* Sequence Diagram
* Class Diagram



### תבנית מס' 2 – [Iterator]

* סיבת הבחירה / שימוש בתבנית:

סיבות לבחירת Iterator :  
גישה למבנה נתונים של אובייקטים מבלי לחשוף את סוג מבנה הנתונים.

חשיפה חלקית של properties של אובייקט.

הגבלה של כמות האובייקטים ממבנה הנתונים.

ובמערכת שלנו בשני הפיצרים רצינו לתת גישה חלקית שנותנת מידע על המשתמש (לדוגמה שם פרטי ומשפחה , תמונה ובפיצר הראשון גם ניקוד), בנוסף רצינו אפשרות עתידית שאם נרצה להגדיל את כמות החברים הכי טובים נוכל להגדיל זאת על ידי שינוי במקום אחד בקוד, בלי לשנות את כל הקוד, רצינו שאם בעתיד נירצה להחליף מבנה נתונים שבו נשמר המידע אודות האובייקטים שלנו לא נצטרך לשנות בצד הקליינט בכלל, כלומר אנחנו רצינו תבנית עיצוב שתאפשר לנו לרוץ על מבנה נתונים בלי לדעת את מבנה הנתונים, ובנוסף לרוץ על כמות ספציפית שאנחנו נרצה להגדיר לו בצד הקליינט, שהקליינט לא יהיה גלוי לכל המידע שקיים באובקייט ששמור אלא מידע חלקי ספציפי, ואם בעתיד יבוצעו שינויים בצד הלוגי כמו שינוי מבנה נתונים או הדרך שהוא ממומש זה לא יפגע ויצטרך מאיתנו לשנות בצד הקליינט שום דבר.

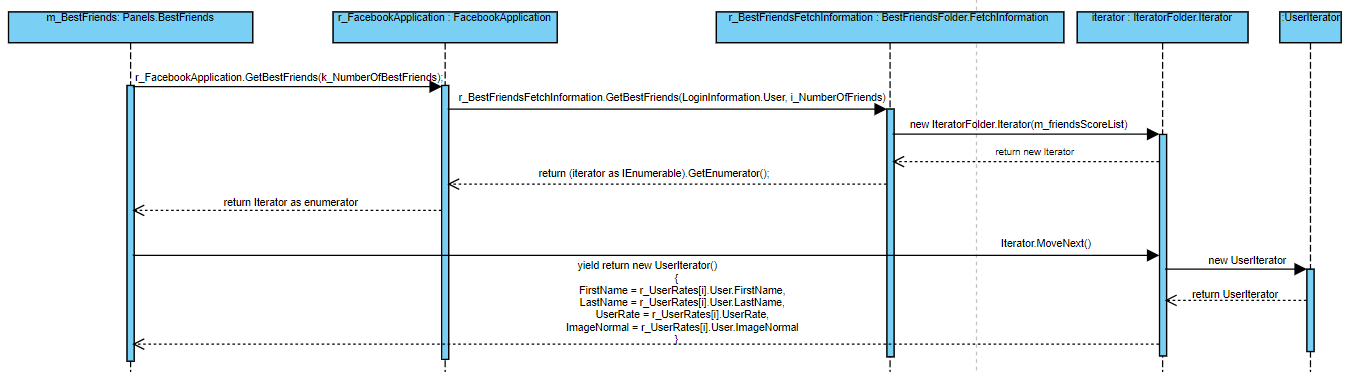
לכן תבנית עיצוב Iterator פותר לנו את הבעיות שבהם נתקלנו בכך שהקליינט אינו מכיר את מבנה הנתונים שהלוגיקה עובדת איתם , הקליינט יכול להגדיר כמה להגביל את כמות האוביקייטים כלומר כמה אובייקטים מתוך האוסף אנחנו רוצים , ובנוסף המידע שמגיע מוגבל כלומר אנחנו לא מקבלים את User עצמו שבתוכו יש מלא אלמנטים אלא מקבלים UserIterator שהוא מביא לנו בדיוק את המידע שאנחנו צריכים וככה מונע מאיתנו גישה לדברים שאיננו צריכים ומונע פוטנציאל לבאגים.

* אופן המימוש:

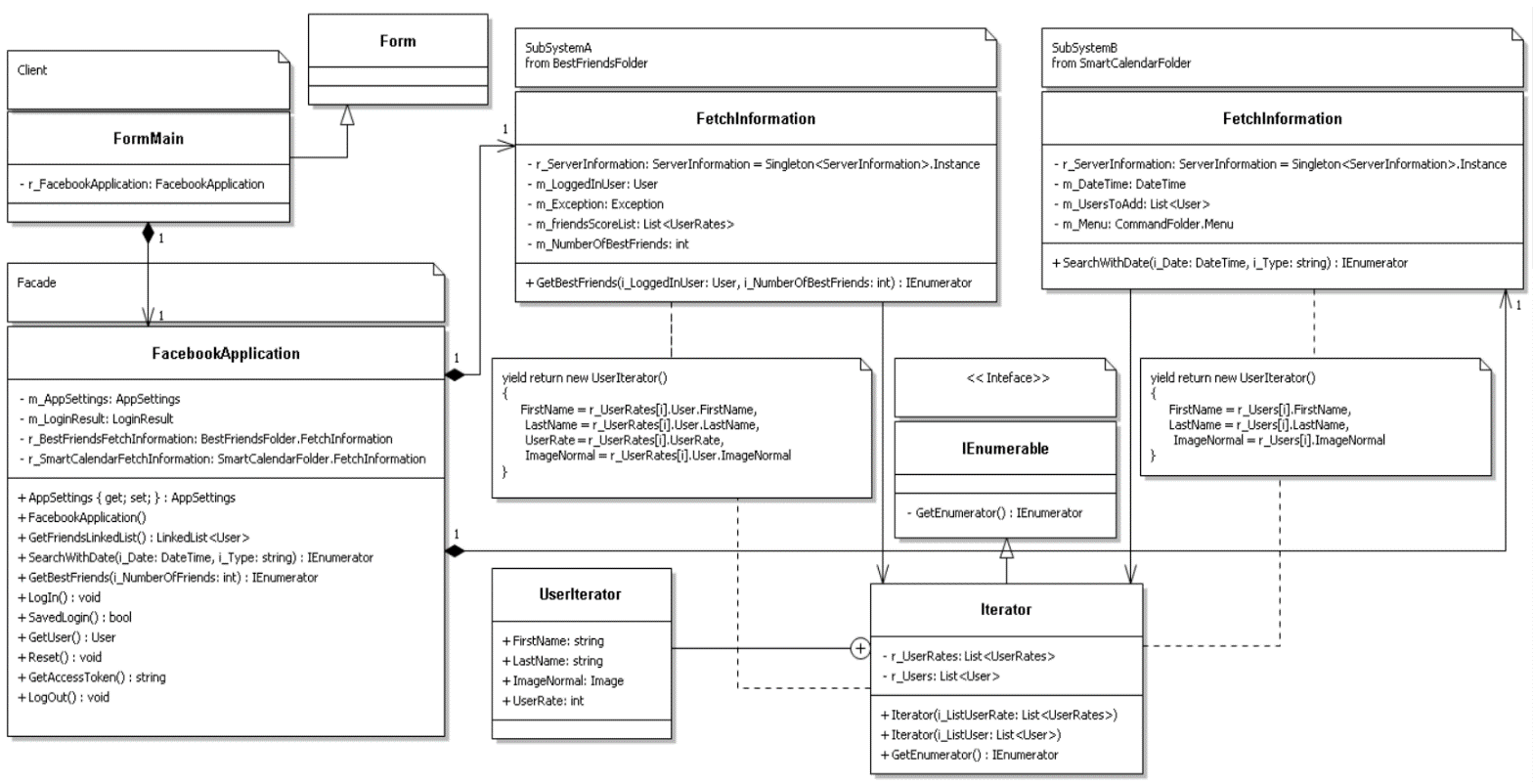
את המימוש שלנו ביצענו במחלקות -מחלקה בשם Iterator אשר נמצאת בתוך ,IteratorFolder FacebookApplication שהוא הFACADE שלנו, ובשני מחלקות הלוגיקה של שני הפיצרים BestFriendsFolder.FetchInformation

ו-SmartCalendarFolder.FetchInformation, דוגמת מימוש של הפיצר BestFriends אשר מחזיר את ה5 החברים הכי טובים, במחלקה FetchInformation של הפיצר הראשון (BestFriends) קיימת מטודה אשר עוברת בלולאה על הקולקציה הנ"ל (רשימת החברים הכי טובים) וממשת את ממשק הIEnumerable. במטודה GetEnumratorשנמצאת במחלקה Iterator אנו יוצרים אובייקט UserIterator אשר מממש את הממשק IEnumartor והוא האיטרטור המוחזר לFacebookApplication אשר הוא הFacade שמוחזר אל הClient.

* Sequence Diagram



* Class Diagram



### תבנית מס' 3 – [Strategy]

* סיבת הבחירה / שימוש בתבנית:

הסיבה שבחרנו להשתמש ב strategy היא מכיוון שהפיצר BestFriends בתוכו ממומשים אלגוריתמים שמבניהם דומה והשוני בהם הוא בתנאי הבדיקה ולכן אנחנו צריכים פתרון לבעיה הנדסית על מנת למנוע שיכפול קוד, תחזוקתיות, ופוטנציאל לבאגים, וכאן strategy בא לעזרתנו מכיוון שהוא מאפשר לנו לבנות אלגוריתם כללי שאפשר להבדיל אותו על ידי תנאי לדוגמה אצלנו בקוד אם אחנו רוצים לשלוף מידע אודות פוסטים ולייקים , פוסטים ותגובות, תמונות ולייקים, תמונות ותגובות, אלבומים ולייקים, אלבומים ותמונות, ובנוסף חישוב המידע אודות לייקים או תגובות, כלומר אלגוריתם שמובדל על ידי אסטרטגיה שונה על פי בחירת הקליינט.

לכן בחרנו בתבנית עיצוב Strategy וככה במקום שיהיה אלגוריתם לכל אחד מהדברים שתיארנו קודם, יש אלגוריתם אחד ויחיד אשר מה שאנו שולחים אליו זה את התנאים שמבדילים בין אלגוריתם לאלגוריתם כלומר אסטרטגיה בדיקה שונה, ככה מנענו שיכפול קוד, שיפרנו תחזוקתיות עתידים שאם בעתיד נרצה להוסיף גם אירועים ולייקים לא נצטרך לבנות אלגוריתם נוסף אלא להוסיף שורת קוד אחת שמשתמשת באלגוריתם הקיים ובנוסף אם נגלה באג בקוד לא נצטרך לשנות בכל האלגוריתמים אלא במקום אחד בלבד, ובנוסף מנענו פוטנצאל לבאגים בכך שאין צורך שוב פעם לבנות את הקוד מחדש.

* אופן המימוש:

את המימוש ביצענו בצורה הבאה:

יצרנו מחלקות בשמות UserRateBy ו-CalculatorUserRate.

המחלקה CalculatorUserRate היא מחלקה גנרית אשר מקבלת T אשר הוא יכול להיות סוג הטיפוס בהתאם למה שהקלייט מבקש.(לדוגמה Like).

המחלקה מחזיקה בתוכה member מסוג delegate Func אליו נשלח כל פעם האסטרטגיה הרצויה, כאשר התבנית היא <T, string, bool>.

בתוך המחלקה קיימת מטודה CalculateUserRate אשר מחשבת על פי האסטרטגיה שאותה קיבלנו ועל פי הסוג T שקיבלנו את הניקוד, אותה אסטרגטיה אנחנו מפעילים באמצעות פעולת Invoke, יחד עם שני פרמרטים T וstring אותם נשלח להשוואה.

באופן דומה יצרנו מחלקה UserRateBy אשר גם היא מחלקה גנרית אשר מקבלת T1, T2

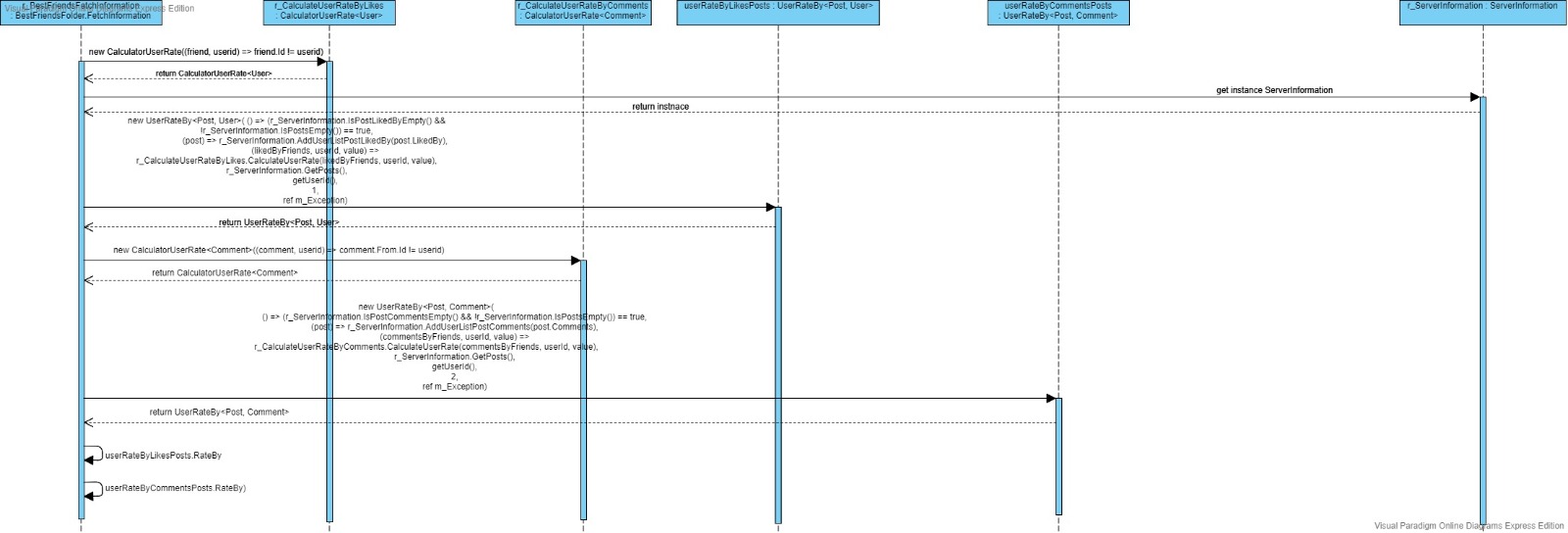
ובתוכה יש כמה אסטרטגיות שנצטרך לבדוק ולכן קיימים members שנקראים isEmpty, MethodOne וMethodTwo, אשר IsEmpty הוא delegate Func בעל תבנית <bool> וMethodOne הוא delegate Func בעל תבנית <T1, FacebookObjectCollection<T2>>

וMethodTwo הוא delegate Action בעל תבנית <FacebookObjectCollection<T2>

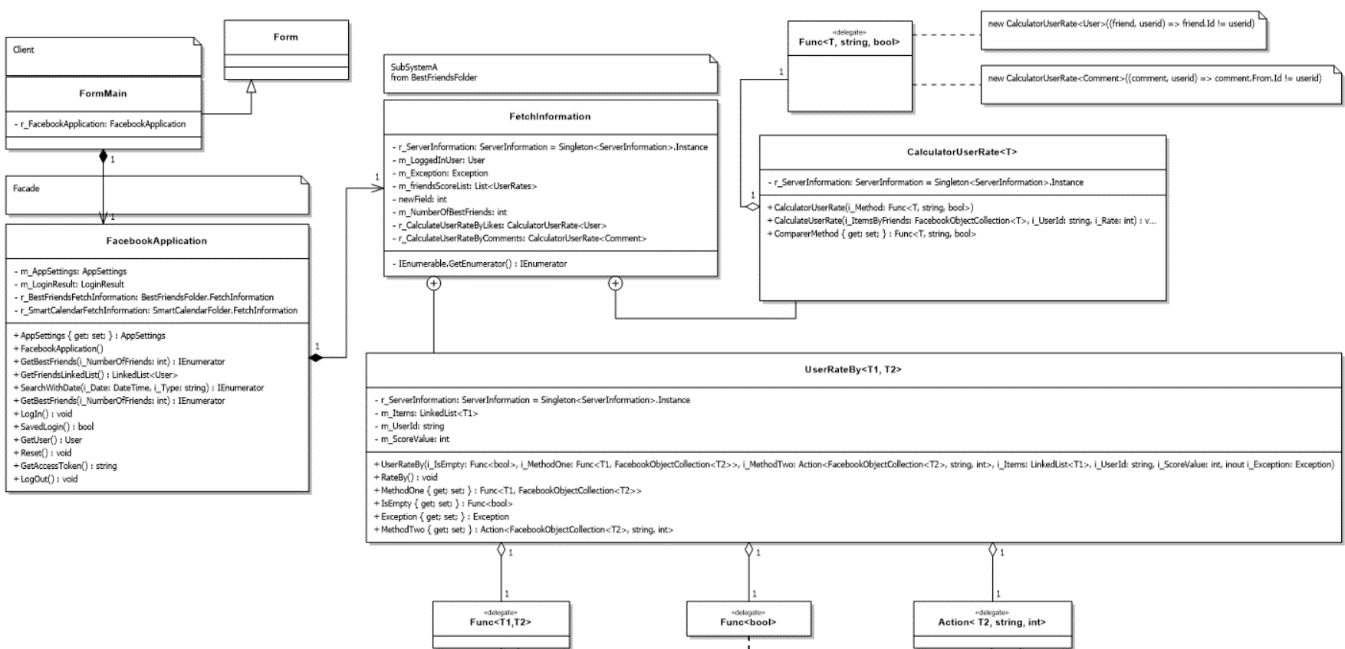
בתוך המחלקה קיימת מטודה RateBy אשר מריצה את האסטרטגיה הראשונה IsEmpty ובודקת אם היא מתקיימת אם כן היא רצה בלולאה על items מסוג T1 ועליהם את האסרטגיה בשם MethodOne אשר מחזירה איבר מסוג T2 שעליו מריצים את אסטרטגיה בשם MethodTwo.

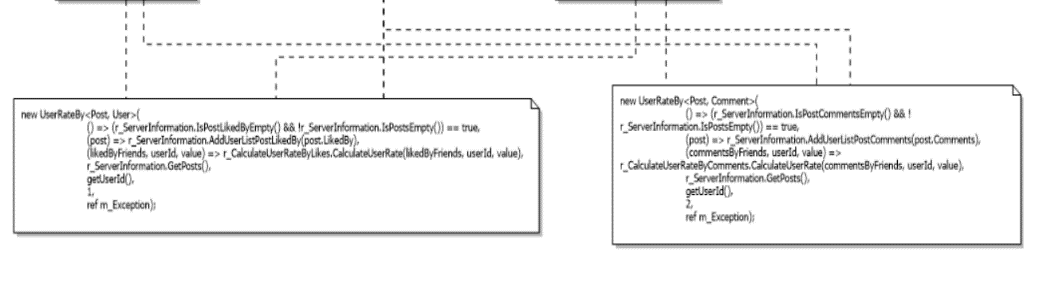
מיקום שני המחלקות הנ"ל נמצא במחלקה BestFriends.FetchInformation מכיוון ששני המחלקות הנ"ל קשורות בקשר ישיר אחד לשני ואינם יכולות להתקיים מחוץ למחלקה ולכן גם שני המחלקות הנ"ל Private.

* Sequence Diagram



* Class Diagram





### תבנית מס' 4 – Façade

* סיבת הבחירה / שימוש בתבנית:

במערכת כולה, אנו משתמשים בתתי מערכות וברכיבים שונים, לדוגמה שירותי המחלקה FacebookService ואף בפיצרים (BestFriends and SmartCalendar) והמחלקות ServerInformation and LoginInformation

בחרנו לספק ממשק שטוח לשירותים הנ"ל, באמצעות המחלקה FacebookApplictaion , אשר משמשת כFacade שקוף, ודרכה ה-Client יכול לגשת לכל השירותים באופן פשוט ומבלי לגשת לתת הרכיבים באופן ישיר ומסובך, דבר המונע מידיעה של תתי המערכת השונים ושיפור התחזוקתיות ומניעת באגים ואם בעתיד נרצה להחליף את אחד מתתי המערכת השונים נוכל לשנות זאת ללא שינוי כלשהו אצל הClient.

* אופן המימוש:

מימשנו את תבנית עיצוב באופן הבא:

המחלקה FacebookApplication מחזיקה בתוכה References לתתי המערכת השונים כמו :

BestFriendsFolder.FetchInformation

SmartCalendarFolder.FetchInformation

AppSettings

LoginResult

המחלקה חושפת לקליינט את השירותים הרלוונטים למערכת בלבד, באופן נוח ופשוט ע"י מטודות אשר מבצעות את התהליכים עצמם מול הרכיבים הלוגים השונים, ולמעשה מחברת בין ממשק המשתמש לבין השיכבה הלוגית.

לדוגמה : LogIn, LogOut, GetUser, GetBestFiveFriends

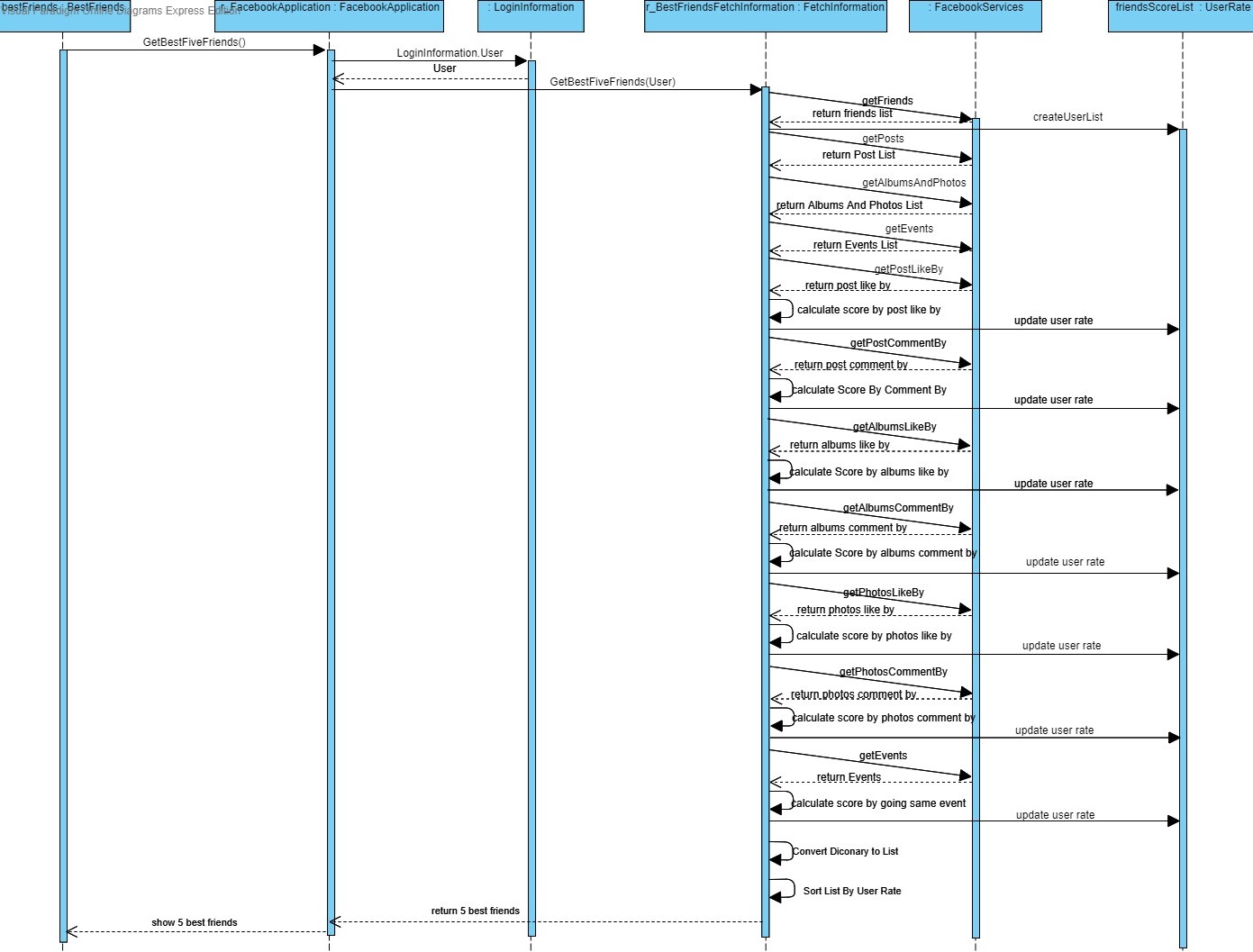
ועוד..

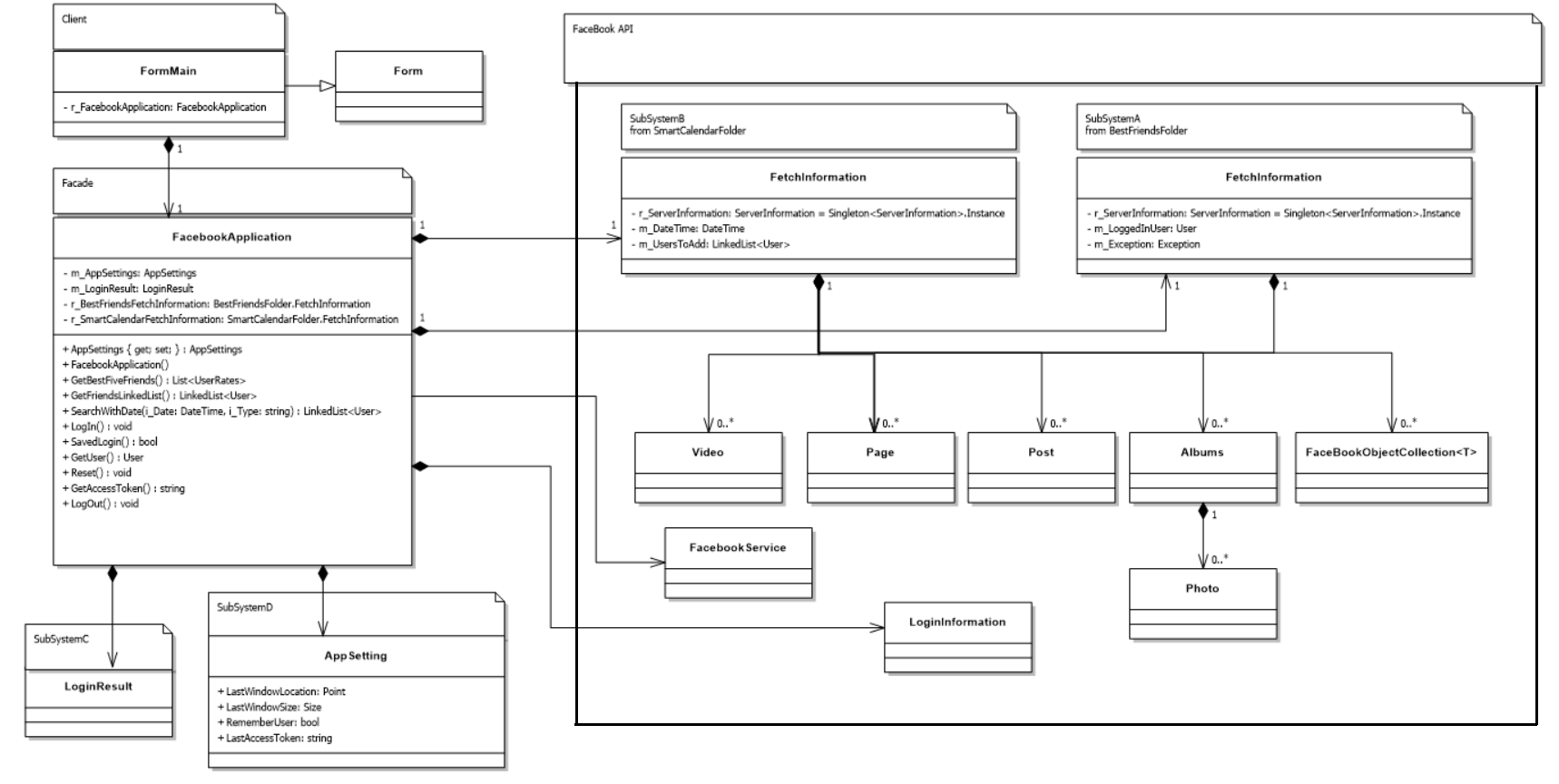
מיקום בקוד:

בתקייה: Façade

קובץ : FacebookApplication.cs

* Sequence Diagram



* Class Diagram

### תבנית מס' 5 – Singleton

* סיבת הבחירה / שימוש בתבנית:

במערכת כולה אנחנו משתמשים בFacade אשר נוצר לצורת הבאת מידע למערכת כולה, בעת היצירה של הFacade נוצרים עוד כמה תתי מערכת כמו למשל ServerInformation אשר היא מחלקה שמנהלת את המידע שמוצג מפייסבוק ולכן אנו צריכים מופע אחד ויחיד של המחלקה הנ"ל, כי אם ניצור כל פעם מחדש את המחלקה ServerInformation כל המידע שנשמר ימחק , לכן אנו צריכים מופע אחד ויחיד שייווצר בפעם הראשונה וכל פעם שנצטרך את האוביקט אם הוא כבר קיים הוא לא יווצר שוב אלא ישלח הקיים, ופה התבנית עיצוב Singleton פותרת לנו את הבעיה ההנדסית שנקלתנו בה.

* אופן המימוש:

מימוש התבנית חייב לעמוד בשני עקרונות: מופע יחיד של המחלקה, וגישה גלובלית. נדרשת האפשרות לגשת לאיבר במחלקה singleton מבלי ליצור אובייקט של המחלקה, וכן מנגנון אשר ישמור את ערכי האיברים של המופע שנוצר. התבנית מושגת באמצעות מחלקה הכוללת מטודה, אשר בקריאה הראשונה יוצרת מופע חדש, ובקריאות הבאות, מחזירה מצביע לאותו מופע שיצרה קודם. כדי להבטיח שאכן יהיה מופע בודד של המחלקה ולא ניתן יהיה ליצור אותה בצורה אחרת, הגדרת הגישה לפונקציית הבנאי (Constructor) מוגדרת כprivate.

ביישומים מרובי תהליכונים (multi-threaded applications) יש להיזהר במימוש התבנית. אם שני תהליכונים ניגשים לפונקציית הבנייה באותו הזמן, ייתכן מצב בו שניהם ייצרו מופע – מצב אותו אנו מנסים למנוע.

לכן המימוש שאנו יצרנו הוא Lazy Protective Generics , כלומר מופע שנוצר רק כאשר פונים אליו + הוא מוגן מפני שימוש בתהליכונים + הוא מתאים לכל מחלקה שנרצה לההשתמש בצורה של Singleton.

מיקום בקוד:

תיקיות:

ServerDetails

BestFriendsFolder

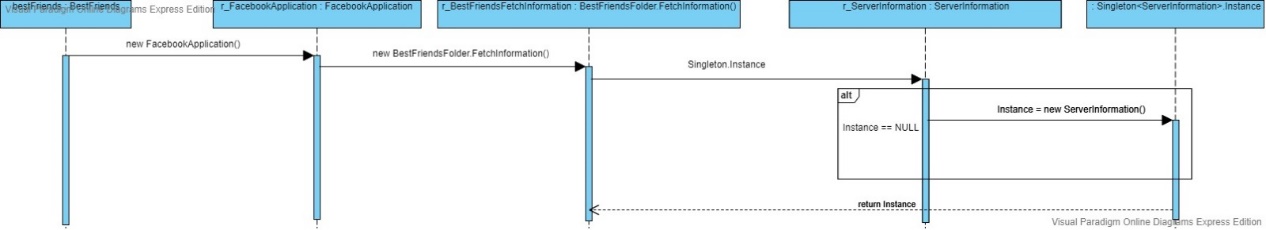
SmartCalendarFolder

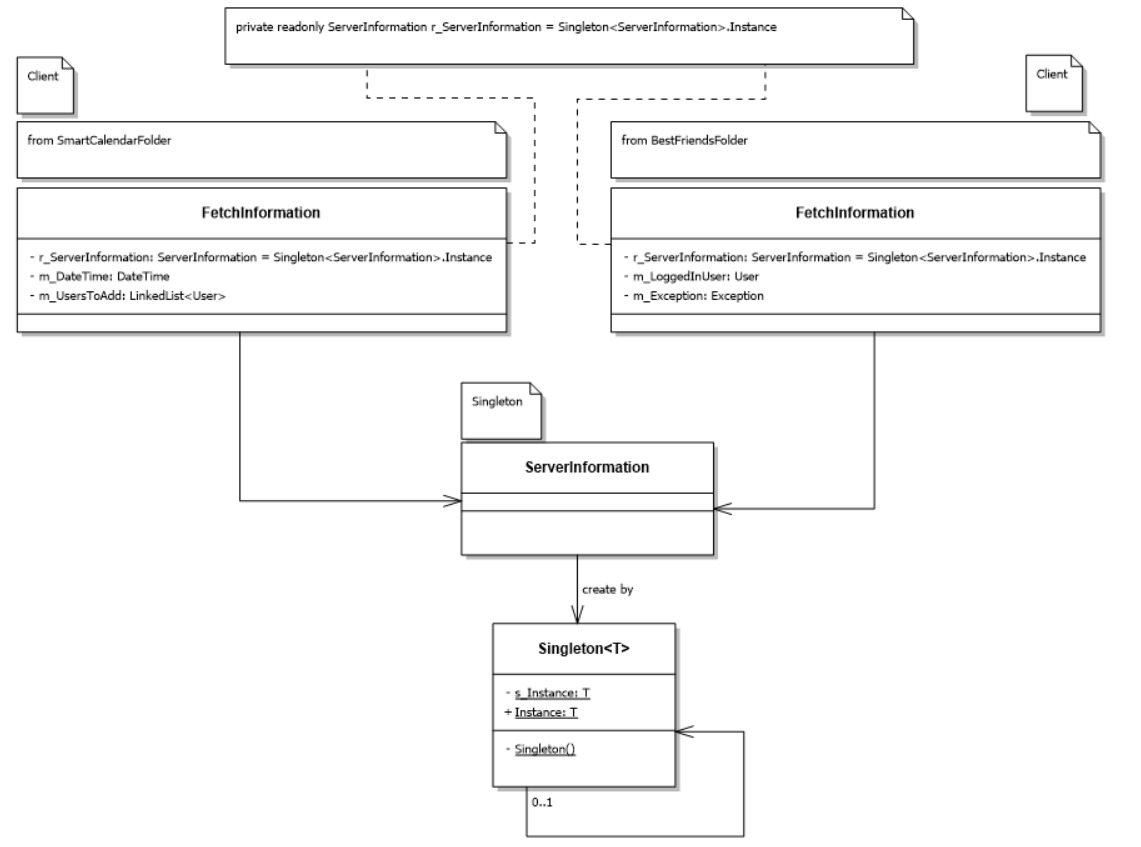
ובקבצים:

ServerInformation.cs

FetchInformation.cs(BestFriends)

FetchInformation.cs(SmartCalendar)

* Sequence Diagram
* Class Diagram



### תבנית מס' 6 - Proxy

* סיבת הבחירה / שימוש בתבנית:

כחלק מפונקציונליות המערכת שלנו רצינו להוסיף PictureBox and Lable (legacy component) פונקציונאליות חדשה.

המטרות שלנו היו לייצר PictureBox אשר הוא עגול וכאשר העכבר שלנו עובר מעליו הוא הופך לריבוע (ומציג את כל התמונה) וישנתנה חזרה כאשר העכבר יזוז ממנו, ורצינו לייצר גם Lable שכאשר העכבר שלנו עליו הוא משנה את צבעו לצבע רנדומלי (אקראי), וכאשר העכבר יזוז ממנו הוא יחזור לצבעו המקורי.

מכיוון שרצינו להשתמש בPictureBox והLable החדשים באותו אופן שמתמשים עם המקוריים, אז בחרנו לעשות שימוש בהורשה ולא בקומפוזיציה על מנת שהclient יוכל להשתמש בהם כאילו הם המקוריים, ללא חשיפה של השינויים הלוגיים לclient ובנוסף לכך הם הפכו להיות בעלי יכולת לשימוש חוזר, ולכן בחרנו להשתמש בתבנית עיצוב Class Proxy.

* אופן המימוש:

בכדי לממש את CustomPictureBox ו CustomLable ירשנו מPictureBox ו-Lable בהתאמה , והוספנו מימושים ל2 מטודות וירטואליות שקיימות בשניהם שנקראות

OnMouseHover/ OnMouseLeave, ובCustomPictureBox הוספנו מימוש פונקציה נוספת שנקרא OnResize.

כל השינויים שבוצעו בפקדים אינם משנים את האופן שהClient משתמש או עובד איתם, כאילו הוא משתמש בפקד המקורי.

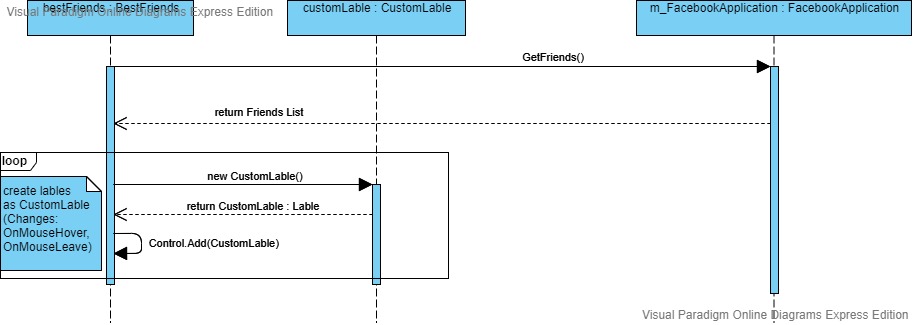
מיקום בקוד:

תיקייה: Proxy

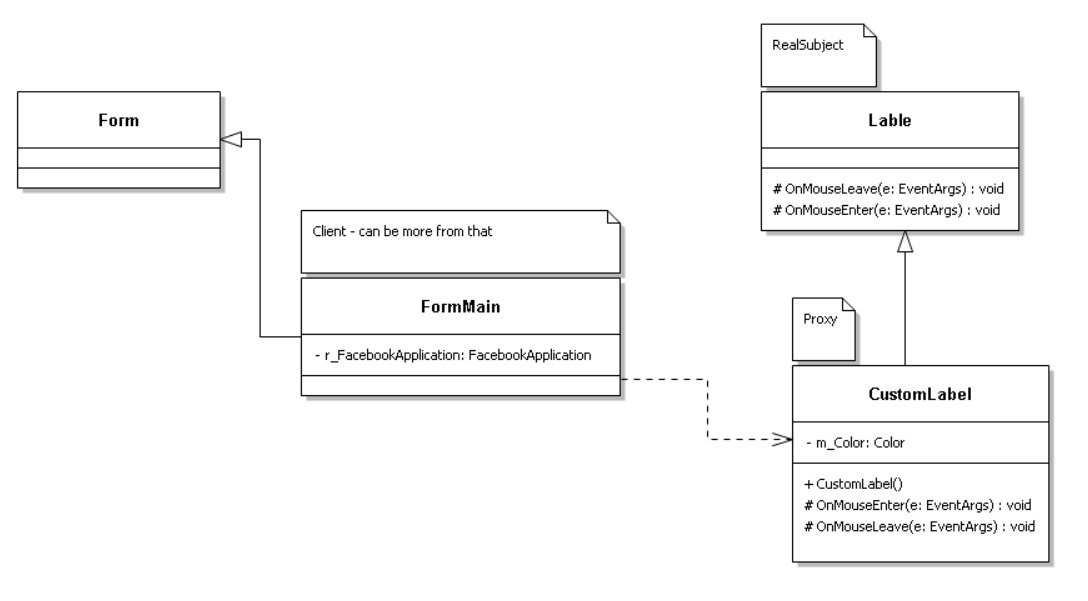
קבצים: CustomLable.cs

CustomPictureBox.cs

* Sequence Diagram

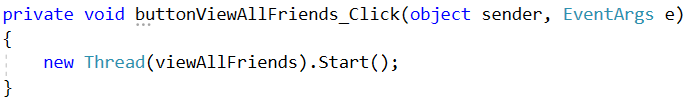


* Class Diagram

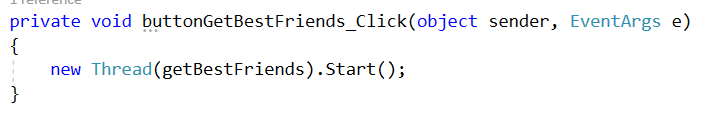


**עבודה אסינכרונית**  
השתמשנו ב Threads במספר מקומות בקוד בהן היינו צריכים לבצע פעולות ארוכות,  
ורצינו לאפשר ליתר המערכת לפעול באותו זמן.  
מנגנון הריצה באמצעות Threads מאפשר לספק למשתמש במערכת ההפעלה מהירות תגובה ורציפות פעולה כשאר התהליך (יישום) מבצע כמה משימות במקביל.  
  
בכל מקום שבו השתמשנו ב Threads השתמשנו ב Invoke על הפקדים כדי לא להגיע לשגיאה  
הנקראת cross-thread operation אשר נגרמת כי רוב הפקדים לא מאפשרים לעבוד איתם מתהליכון שונה מהתהליכון שבו הם נוצרו.

במספר מקומות השתמשנו בפקודה thread1.Join() אשר לא ממשיכה ל thread הבא עד אשר thread1סיים את פעולתו ללא הפקודה במקומות הנכונים התכנית לא הייתה רצה כמו שאנו המפתחים שלה רצינו וזה כנראה היה יוצר לנו בתוכנית באגים.  
  
בתיקייה בשם Panels ישנה מחלקה בשם BestFriends במחלקה הזאת בחרנו להשתמש במנגנון Threads בשתי פונקציות viewAllFriends , getBestFriends :

1) כאשר המשתמש לוחץ על כפתור View All Friends מתחילה פעולה של יצירת רשימה שיש בה את כל החברים של המשתמש שביצע Login ובמידה שרשימת החברים של המשתמש לא ריקה  
אנו מקשרים בין החברים של המשתמש ל ListBox בשם friendsListBox באמצעות DataBinding  
ולכן בגלל מספר הפעולות הרב השתמשנו ב Thread כדי להמשיך את רציפות המערכת.

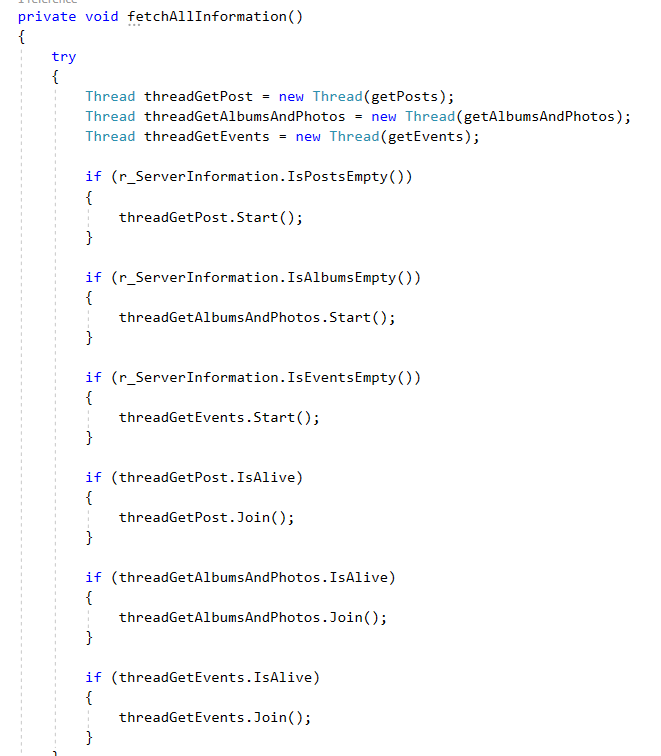
2) כאשר המשתמש לוחץ על כפתור Five of my best friends מתחילה פעולה של יצירת רשימה שיש בה את חמשת החברים הכי טובים של המשתמש שביצע Login .  
פעולה זאת מורכבת מכמה תתי פעולות  
לאחר שיצרנו את רשימת חמשת החברים הכי טובים של המשתמש לפי האלגוריתם שכתבנו  
אנו מכניסים את רשימת החברים לDataGrid בשם bestFriendsTable  
ולכן בגלל מספר הפעולות הרב השתמשנו ב Thread כדי להמשיך את רציפות המערכת.



בתיקייה בשם BestFriendsFolder ישנה מחלקה בשם FetchInformation במחלקה הזאת בחרנו להשתמש במנגנון Threads בפונקציה בשם fetchAllInformation

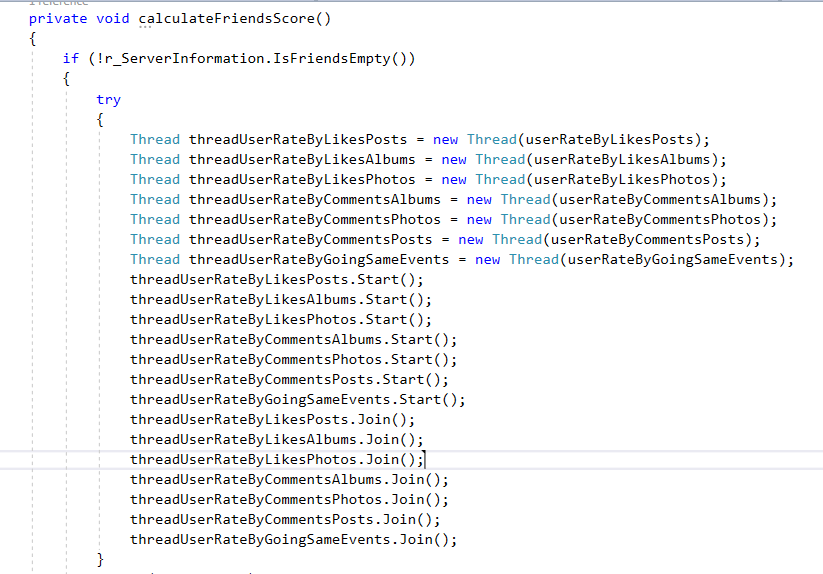
הפונקציה fetchAllInformation אחראית לתפוס את כל  
הנתונים (Posts , Albums and Photos , Events)  
תפיסה של כל אחד מהנתונים לוקחת זמן ממושך כשאר היא  
מפעילה פונקציה אשר לוקחת זמן ממושך.  
ולכן בגלל מספר הפעולות הרב השתמשנו ב Thread בכל תפיסה של נתון כדי לספק למשתמש במערכת ההפעלה מהירות תגובה ולהמשיך את רציפות המערכת.

והשתמשנו בJoin על מנת שלא ימשיך למטודה הבאה לפני שהוא מקבל את כל המידע הנחוץ לשלב הבא

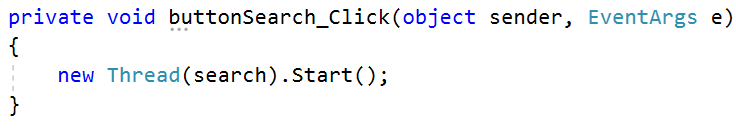


בתיקייה בשם BestFriendsFolder ישנה מחלקה בשם FetchInformation במחלקה הזאת בחרנו להשתמש במנגנון Threads בפונקציה בשם calculateFriendsScore

הפונקציה calculateFriendsScore אחראית לחשב את ה Score של כל אחד מהחברים של המשתמש שעשה Login.  
החישוב מתבצע במספר פעולות שזמן של כל פעולה אשר מחשבת Score לפי likes ו- comments  
על פי נתון ספציפי (Posts , Albums , Photos , Events) לוקח זמן ממושך.  
ולכן בגלל מספר הפעולות הרב בכל חישוב של נתון השתמשנו ב Thread כדי לספק למשתמש במערכת ההפעלה מהירות תגובה ולהמשיך את רציפות המערכת. גם כאן אנו רואים שימוש בפקודה join אשר ממתינה לסיום הטרד על מנת לחכות שכל הטרדים יסיימו את פעולת כדי לעבור לשלב הבא שהוא תלוי בסיום השלב הנוכחי ואם לא נחכה דבר שיגרור באג חמקמק (כזה שאפילו לא נשים לב אליו).



בתיקייה בשם Panels ישנה מחלקה בשם SmartCalendar במחלקה הזאת בחרנו להשתמש במנגנון Threads בפונקציה בשם search

כשאר המשתמש לוחץ על כפתור Search במערכת, מתחילה פעולה של חיפוש חברים (Users) ויצירת רשימה של אותם חברים אשר תואמים לחיפוש הספציפי שהמשתמש רצה לבדוק,  
ולאחר שיצרנו את רשימת החברים  
אנו מכניסים את רשימת החברים ל DataGrid בשם dataGridViewFriendsBirthdayInTheDate  
ולכן בגלל מספר הפעולות הרב השתמשנו ב Thread כדי להמשיך את רציפות המערכת.

**עבודה עם Data Binding**:  
  
בחרנו להשתמש בעבודה עם Data Binding עבור הפרטים האישיים של כל החברים של המשתמש שביצע Login (First Name , Last Name , Profile Picture, Birthday, Email, Current City, About) , About עובד בצורה של 2-way databinding כלומר ניתן לערוך את המידע והוא ישמר (עובד לשני הכיוונים)  
  
ממומש בתרגיל במקום אחד בלבד :  
בתיקייה בשם Panel ישנה מחלקה בשם BestFriend  
ובמחלקה ישנה פונקציה בשם viewAllFriends  
  
בחרנו להשתמש כאן ב Data Binding מכיוון שאנו רוצים להציג למשתמש את הפרטים האישיים של החברים שלו, אשר כולם נלקחים מאובייקט יחיד User, ולכן בשורת קוד אחת אנו יכולים לבצע קישור אל מול כל הפרטים האישיים של חבריו של המשתמש.  
קישור זה מביא לכך שכאשר אחד מהפרטים של חבריו ישתנה, השינוי יתבצע באופן אוטומטי  
ואילו ללא השימוש בData Binding המידע שיוצג יהיה המידע שנלקח בתחילת ריצת האפליקציה אשר מהווה מידע לא עדכני במקרים מסוימים.