### תיאור קצר של הפיצ'רים שבחרנו לממש בתרגיל הקודם:

1. **PostFinder**  
   חיפוש פוסטים עפ"י דרישה של המשתמש.  
   הפיצ'ר ממומש באופן הבא,  
   בטאב **PostFinder** יופיע:  
   בחירת טווח תאריכים ספציפיים אשר פורסמו הפוסטים של המשתמש.  
   כתיבת משפט שאותו המשתמש מעוניין לחפש בפוסטים שמוצגים למשתמש בתצוגה השמאלית (My Original posts).  
   אופציה לבחור את הפוסטים עם הכי הרבה תגובות.

הוספנו אופציה לבחור מיון הפוסטים שנבחרו ע"י:

* כמות לייקים
* כמות תגובות
* תאריך יצירת הפוסטים.

לאחר מכן, ע"י לחיצה על כפתור ה – Search מתבצעת הפעלת החיפוש למציאת הפוסטים הרלוונטים למשתמש.

אם נמצא פוסטים כאלו, אז יוצג בתצוגת הימנית ( My filtered posts) את הפרטים הבאים:

* תאריך יצירת הפוסט.
* מי כתב את הפוסט.
* תוכן הפוסט.

1. **StarSignFinder**  
   הצגת מזלות (מגלגל המזלות/הזודיאק) של החברים של המשתמש עפ"י בחירתו.  
   ישנם 2 רשימות תצוגה עבור החברים של המשתמש:

* רשימת שמות של חברים עם אותו מזל כמו של המשתמש.
* רשימת שמות של חברים עם מזל שונה מהמשתמש.

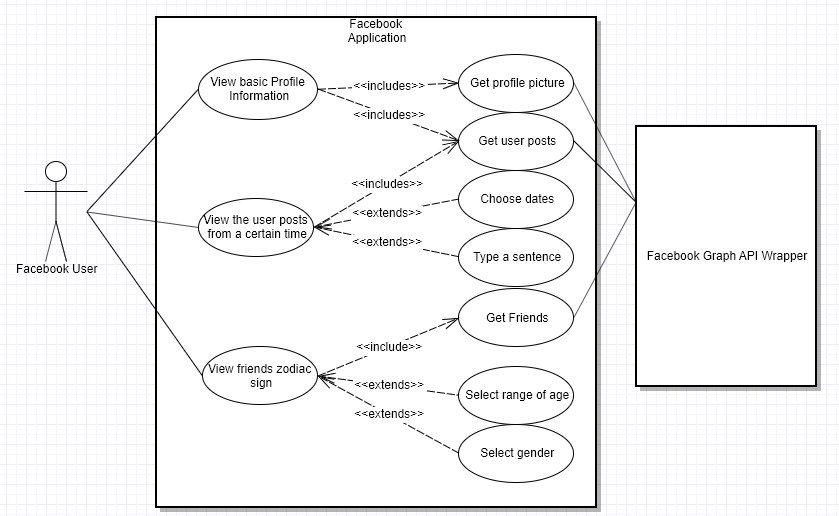
הפיצ'ר ממומש באופן הבא,  
בטאב **StarSignFinder** יופיע:  
בחירת מגדר של החברים שיוצגו ברשימות בטופס.  
בחירת טווח הגילאים של החברים שיוצגו ברשימות בטופס.  
  
ולאחר מכן לחיצה על שם של אחד החברים שמופיעים ברשימת חברים (לחיצה על פריט ב listBox).

יפורסם המידע הבא לגבי החבר שנלחץ:

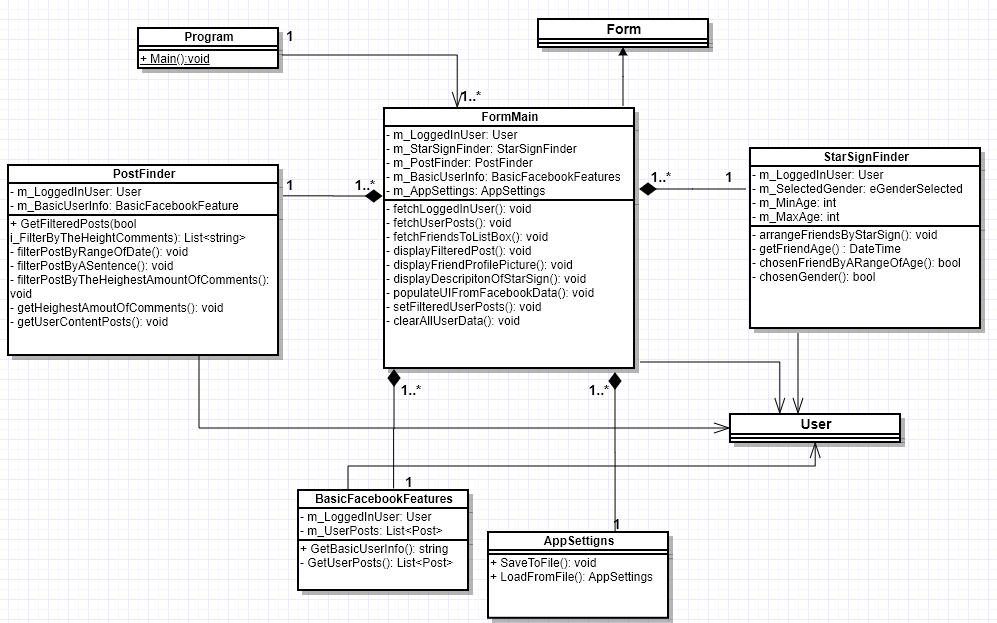
* שם המזל של החבר.
* מידע על המזל של החבר.
* תמונת הפרופיל של החבר.

במידה והמשתמש החליט לשנות את הבחירות, יש ללחוץ על כפתור "Find" להצגת מידע מעודכן.

**Use Case Diagram:**



**Class diagram:**



### תבנית מס' 1 – [Opaque Facade]

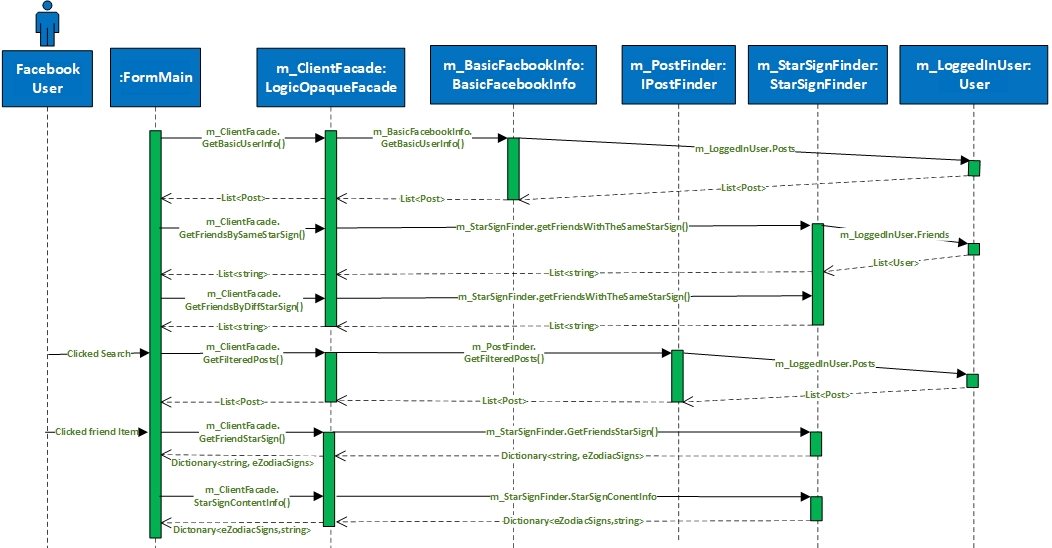
1. **סיבת הבחירה / שימוש בתבנית:**

החלטנו להשתמש ב Façade pattern על מנת שנוכל ליצור ממשק משתמש פשוט ונוח למשתמש. ה Facade מקשר בין ה client במקרה שלנו ה FormMain לבין השכבה הלוגית שמכילה תתי מערכות מורכבות (contain serveral complex processes).  
ה Facade הוא wrapper component שמסתיר את התתי המערכות המורכבות בשכבה הלוגית שלנו. ובכך מאפשרים לנו ממשק משתמש פשוט והוא מסתיר הרבה פעולות מורכבות שמתרכשות מאחורי הקלעים (בשכבה הלוגית).  
התקשורת בין השכבה הלוגית לבין ה Client יתבצע דרך ה Facade בלבד.  
בנוסף התבנית עיצוב Reduce coupling מכיוון שאנחנו פונים לאותן תתי מערכות אך ורק דרך ה Facade (כאשר נרצה להחליף Client לא נצטרך לפנות לאותן תתי מערכות מחדש בשכבה הלוגית, נוכל לעשות לעשות זאת דרך ה Facade שלנו שאחראי על כך).

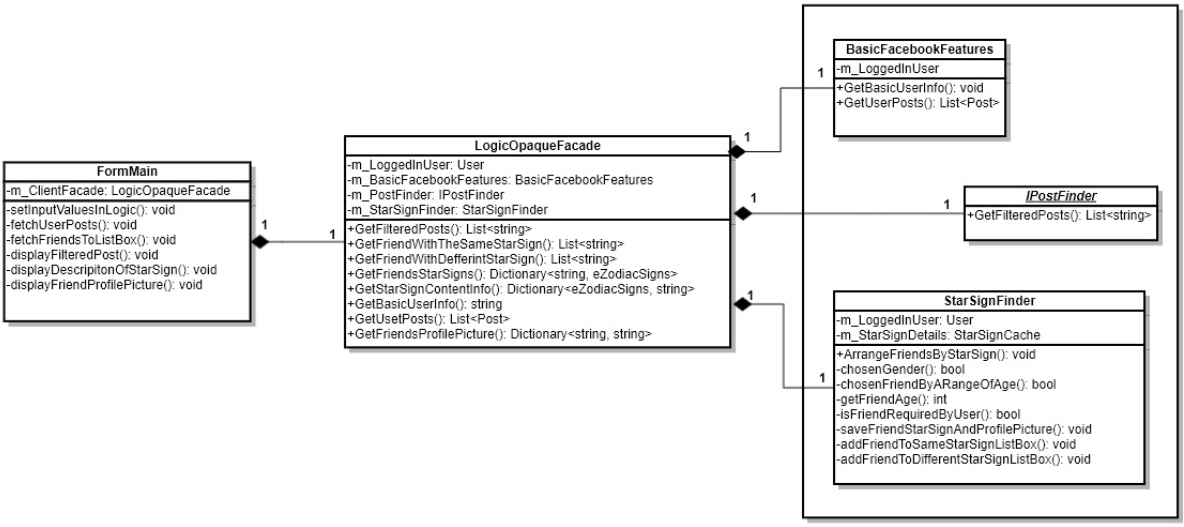
1. **אופן המימוש:**  
   ה – Client הוא ה FormMain כלומר ה User Interface, הוא משתמש ב Facade (במחלקה LogicOpaqueFacade).   
   אשר המשתמש נדרש לקבל מידע כלשהו מהשכבה הלוגית (בתוך dll: UserFeaturesLogic)   
   אז ה Facade מקשר בין המשתמש לבין השכבה הלוגית.  
   המשתמש בוחר בפעולה מסויימת באפליקצייה לרוב יידרש להשתמש בשכבה הלוגית על מנת לקבל מידע מהשרת פייסבוק ואז שנוכל לעבד את התוצאות בהתאם לדרישת המשתמש.  
     
   אנחנו בפרויקט מימשנו Facade אטום כלומר כל המחלקות בשכבה הלוגית הן internal פרט למחלקה Facade שהיא public שרק דרך אפשר ליצור תקשורת עם התתי המערכות בשכבה הלוגית.  
     
   **ניתן למצוא את המחלקות בקוד:**  
     
   פרויקט: UserFeaturesLogic  
   תיקייה: LogicOpaqueFacade  
   מחלקה: LogicOpaqueFacade  
     
   **תפקיד כל מחלקה מהתבנית עיצוב בקוד שלנו:**

**Client:** FormMain  
**Façade:** LogicOpaqueFacade  
**SubsystemA:** BasicFacebookFeatures   
**SubSystemB:** IPostFinder  
**SubsystemC:** StarSignFinder

**Façade Sequence Diagram:**



**Façade Class Diagram:**



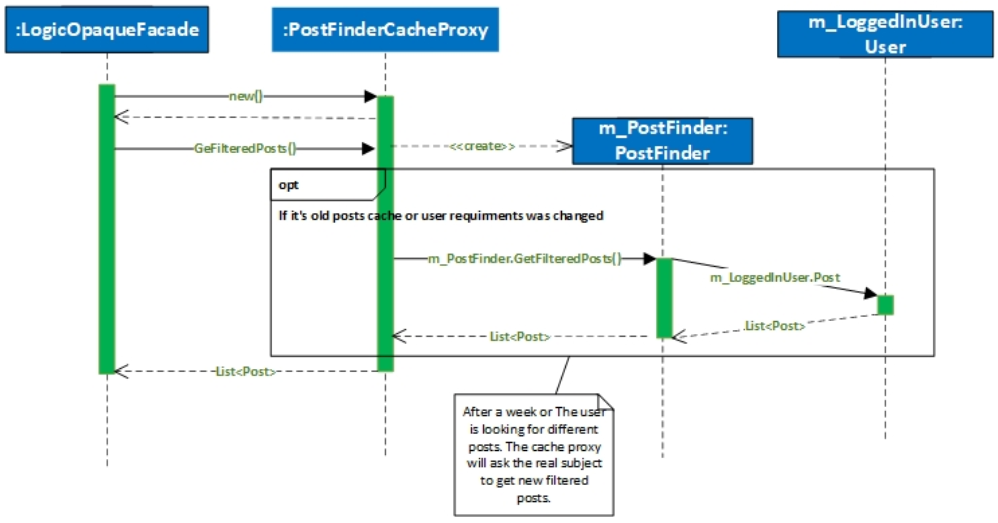
### תבנית מס' 2 – [Proxy]

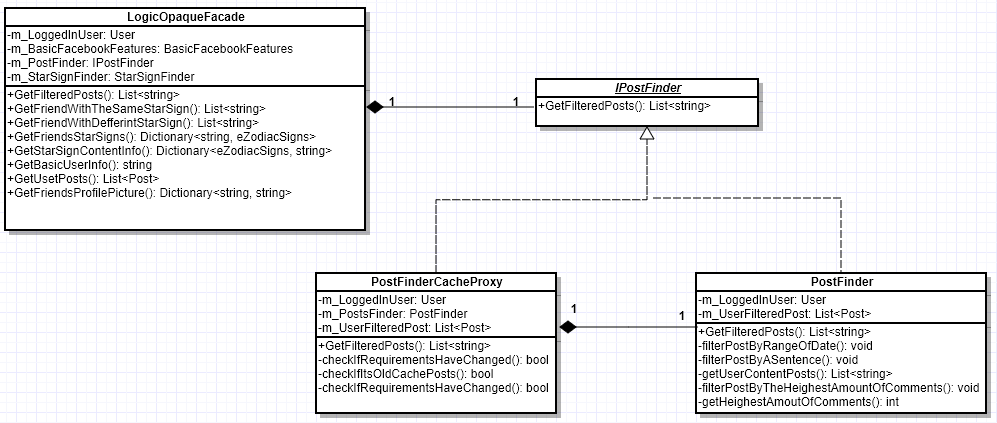
1. סיבת הבחירה / שימוש בתבנית:  
   מימשנו proxy שמתחזה למחלקה ל PostFinder.  
   ה proxy משתמש ב Cache על מנת לשמור את התוצאות של הפוסטים שקיבלנו משרת הפייסבוק לאחר עיבודם בשכבה הלוגית.  
   בקשת הפוסטים משרת הפייסבוק נחשבת לפעולה "יקרה" (גם במשאבים וגם עלול לקחת זמן רב מכיוון שיש שמתרכש מספר רב של תהליכים מאחורי הקלעים עד לקבלת הפוסטים.)  
   מכיוון שהמשתמש יכול לקבל את התוצאות תוצאות של חיפוש הפייסבוק, אין צורך כל פעם מחדש לפנות לשרת הפייסבוק ולעבד את המידע מחדש לקבלת התוצאות הדרשות.  
     
   מימשנו את ה cache ב proxy כך שישמור את התוצאות הפוסטים האחרונות של המשתמש.  
   ברגע שהמשתמש מחפש מחדש את אותן פוסטים בדיוק (כלומר הוא לא משנה את טווח התאריכים שחיפש, את המשפט שחיפש וכד') ובנוסף לא עבר 7 ימים מהחיפוש האחרון.  
   אז הוא יקבל את ה Cache במיידי (שמכיל את התוצאות חיפוש הפוסטים האחרון שלו), מבלי הצורך לחזור על כל התהליך הארוך, אין צורך לפנות לשרת פייסבוק ולעבד את המידע שוב פעם.
2. אופן המימוש:

המחלקה IPostFinder היא בעצם ה Interface ויש בה את המתודה GetFilteredPosts שאחראית למצוא את הפוסטים שדרש המשתמש.  
המחלקה המקורית היא PostFinder שממשת את הממשק, בצורה כזאת שהיא ניגשת לשרת הפייסבוק ומעבדת את המידע לקבל תוצאת הפוסטים הרלוונטים שהמשתמש דרש.  
המחלקה PostFinderCacheProxy היא המתחזה למחלקה PostFinder.  
היא מחזיקה רפרנס לאובייקט מהטיפוס של המחלקה המקורית.  
והמחלקה המתחזה גם היא ממשת את הממשק, אך הפעם ברגע שה Client (במקרה שלנו ה Facade) יבקש מהממשק דרך המתודה לקבלת הפוסטים הנדרשים למשתמש.  
אז המחלקה המתחזה תעשה שימוש במחלקה המקורית ותשמור את התוצאות האחרונות כ Cache שחיפש המשתמש את הפוסטים.  
ובפעם הבא שיבקש שוב את אותן פוסטים, בעצם יעמוד בתנאים הבאים:  
1. יחפש בדיוק באותן טווח תאריכים, את אותו משפט שחיפש.  
 ואת אותו אופציה אם לחפש או שלא, את הפוסטים עם הכי הרבה תגובות.  
2. לא עבר 7 ימים מהחיפוש האחרון שלו.  
  
אז המחלקה המתחזה תחזיר לו באופן מיידי את תוצאות הפוסטים עבור המשתמש מבלי הצורך לגשת שוב לשרת הפייסבוק ולעבד מחדש את התוצאות.  
  
**ניתן למצוא את המחלקות בקוד:**פרויקט: UserFeaturesLogic  
תיקייה: PostFinder  
מחלקות: IPostFinder, PostFinderCacheProxy, PostFinder.  
  
**תפקיד כל מחלקה מהתבנית עיצוב בקוד שלנו:**

**Client:** LogicOpaqueFacade  
**ISubject:** IPostFinder  
**Proxy:** PostFinderCacheProxy  
**RealSubject:** PostFinder

**Proxy Sequence Diagram:**



**Proxy Class Diagram:**  


### תבנית מס' 3 – [Singleton]

1. סיבת הבחירה / שימוש בתבנית:

החלטנו לממש Singleton, מכיוון שאפשר להשתמש ב Cache-memory כאובייקט סינגלטוני. במהלך כל חייו של הסינגלטון קיים לנו רק מופע אחד שאפשר דרכו לגשת לאובייקט הסינגלטון.  
השתמשנו בסינגלטון מכיוון שאנחנו צריכים להשתמש ב Cache שמספק נתונים לגבי המזלות (Zodiac signs) והוא קיים מתחילת ריצת האפליקציה ועד הסיום שלה.

אנחנו רוצים שיהיה לנו מופע אחד של Cache וגם שנוכל לקרוא לאותו אובייקט שיצרנו (StarSignCache) בכל מני מקומות בתכנית רק למופע שלו, בכדי שלא נצטרך ליצור מופעים חדשים על מנת לקבל את אותו מידע של אובייקט שיצרנו כבר.

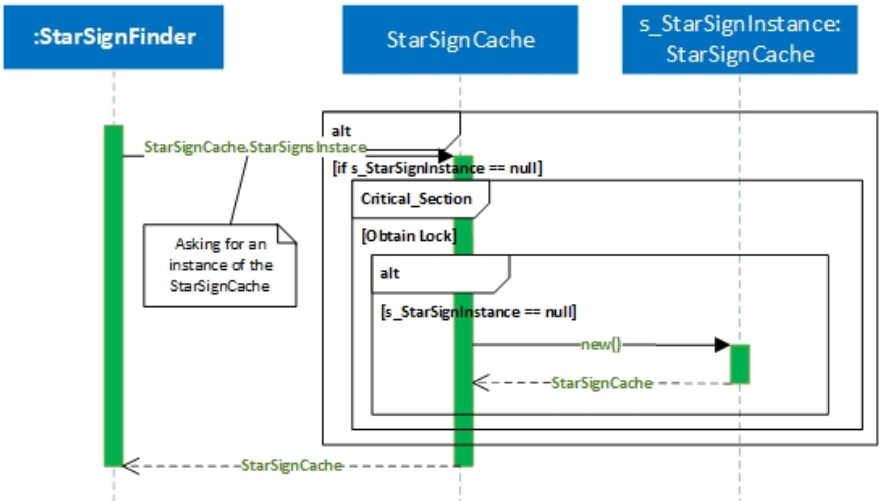
1. אופן המימוש:

המחלקה StarSignCache היא בעצם ה Singleton שתייצר לנו מופע יחיד ברגע ברגע שישתמשו ב property הסטטי StarSignInstance של המחלקה.  
ברגע שה Client במקרה שלנו הStarSignFinder פונה אל המחלקה StarSignCache באמצעות אותו property – StarSignInstance אז הוא יקבל את המופע היחיד שנוצר.  
אנחנו ממשנו את הסינגלטון באמצעות double check lock.  
הקצאת הזיכרון עבור הסינגלטון מתרחשת בתוך הקטע הקריטי.  
על מנת שאם clientים אחרים ירצו גם לפנות לאותו property במחלקה באמצעות Threadים שונים אז שנוכל לאפשר אותם להיות Thread safe.  
מכיוון שפעולת ה Lock היא פעולה "יקרה" במשאבים, אז השתמשנו ב double check כלומר במקום לבדוק כל פעם אם קיים לנו מופע סינגלטון בתוך אזור הקטע הקריטי, בשביל לייעל את התכנית רצוי לפני ה Lock לבדוק אם יצרנו מופע סינגלטון מבלי להיכנס לקטע הקריטי ולבדוק בפנים אם הוקצה כבר סינגלטון.

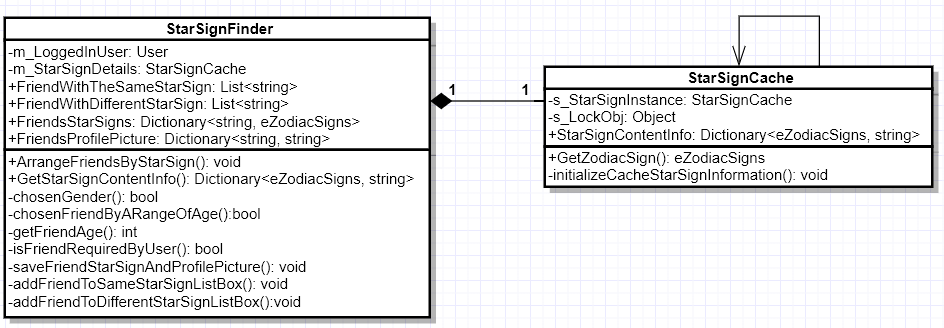
**ניתן למצוא את המחלקות בקוד:**פרויקט: UserFeaturesLogicתיקייה: StarSignFinder  
מחלקות: StarSignCache, StarSignFinder.  
**תפקיד כל מחלקה מהתבנית עיצוב בקוד שלנו:**

**Client:** StarSignFinder  
**Singleton:** StarSignCache

**Singleton Sequence Diagram:**



**Singleton Class Diagram:**



### עבודה אסינכרונית:

השתמשנו בעבודה אסינכרונית במקרים הבאים:  
1. בהבאת תמונת הפרופיל של המשתמש.  
2. בהבאת המידע הבסיסי לגבי המשתמש.  
3. בהבאת הפוסטים של המשתמש.  
4. בהבאת רשימות החברים של המשתמש.  
5. בהבאת רשימות אלבומים של המשתמש.  
  
אחת הסיבות העיקריות שהשתמנו בעבודה אסינכרונית באותן מקרים היא קיצור זמן המידע אל הממשק משתמש.  
מכיוון שהמתודות עובדות מול שרת הפייסבוק, מה שלוקח זמן רב לקבלת המידע והבאתן לתצוגת המשתמש.  
באמצעות שימוש ב Threadים במקביל, אפשר לנצל את היכולת השרת לקבל מספר פניות במקביל ובכך לקצר את הזמן לקבלת הנתונים.  
  
בנוסף ה MainForm שלנו לא נתקע עד שהוא מקבל את כל המידע בתצוגה, ויש אפשרות למשתמש להזיז את חלון התצוגה בזמן שאנחנו מקבלים את הנתונים במקביל.  
  
**את השימושים ניתן למצוא בקוד:**  
  
פרויקט: UserInterface  
בקובץ FormMain.cs במתודות:  
1. populateUIFromFacebookData – שקוראת למתודות:

* fetchUserProfilePicture
* fetchBasicUserInfo
* fetchUserPosts
* fetchFriendsToListBox
* fetchAllUserAlbums

2. loginSetUpForm - שקוראת למתודה:

* fetchLoggedInUser

### עבודה עם Data Binding:

את שני המימושים ניתן לראות בתצוגה של האפליקצייה בטאב ViewDetailedPostsAndAlbums:  
  
Data binding הראשון ממומש בצורה הבאה, הצגת פרטים אודות הפוסט שבחר המשתמש:  
ברגע שהמשתמש לוחץ על אחד מה itemים ברשימת ה postים ב listbox אז הוא יקבל את הפרטים הבאים אודות הפוסט שלחץ.

1. תאריך יצירת הפוסט.
2. תאריך עדכן אחרון של הפוסט.
3. הודעת תוכן הפוסט.

Data binding השני ממומש בצורה הבאה, הצגת פרטים הודעות האלבומים שבחר המשתמש:  
ברגע שהמשתמש לוחץ על אחד מה itemים ברשימת הalbumים ב listbox אז הוא יקבל את הפרטים הבאים אודות האלבום שלחץ.

1. תאריך יצירת האלבום.
2. תאריך עדכון אחרון של האלבום.
3. תוכן האלבום.

**את המימושים ניתן למצוא בקוד:**  
  
פרויקט: UserInterface  
מחלקה: FormMain  
מתודה: fetchAllUserPosts  
מתודה: fetchAllUserAlbum

### תבנית מס' 4 – [Strategy]

* סיבת הבחירה / שימוש בתבנית:

השתמשנו בתבנית עיצוב Strategy Pattern על מנת ליצור אפשרות של מיון הפוסטים שחיפש המשתמש בפיצ'ר PostFinder.  
המיון נעשה לפי כמות הלייקים, כמות התגובות ותאריך יצירת הפוסטים.  
מכיוון שאפשר להשתמש בכל מני אסטרטגיות למיון פוסטים, הגדרנו משפחה של מתודות למיון. ואם נרצה בעתיד למיין בצורה אחרת ולהוסיף עוד אופציות לאסטרגיות מיון מבלי לשנות את את החלק של הלוגיקה במימוש הקוד.

* אופן המימוש:

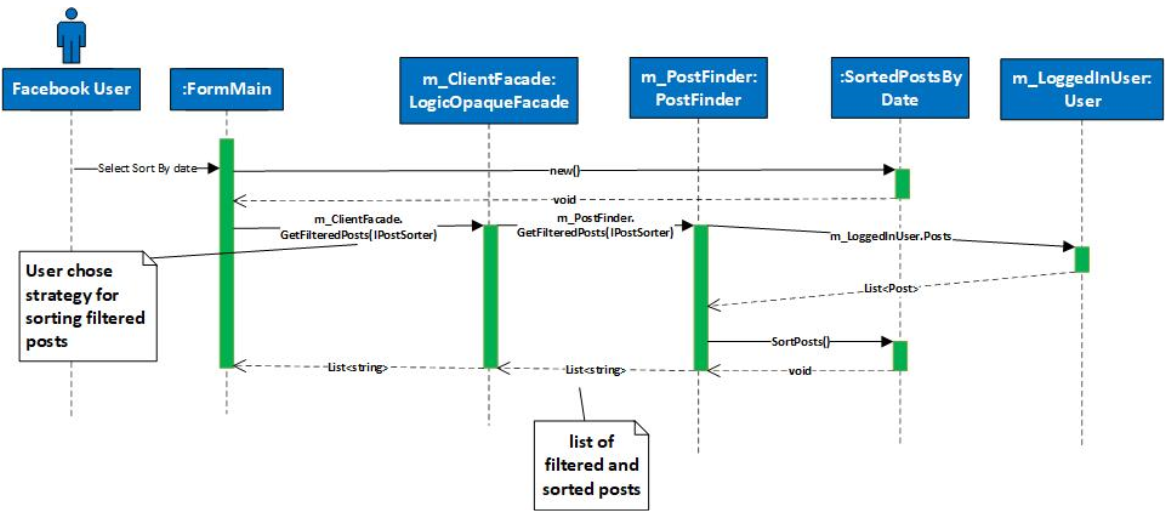
בנינו ממשק בשם IPostComparer שמחזיק רק את המתודה Sort.  
מימשנו 3 מחלקות שממשות את הממשק IPostSorter:

1. SortPostsByDate.
2. SortPostsByComments.
3. SortPostsByLikes.

במחלקה PostFinder יש את המתודה "GetFilteredPosts" שמקבלת את האובייקט IPostSorter שמכילה את האסטרטגיה למיון הפוסטים אשר בחר המשתמש.  
וע"י הפעלת המתודה SortPosts נוכל לבצע את המיון הרצוי עבור הפוסטים.

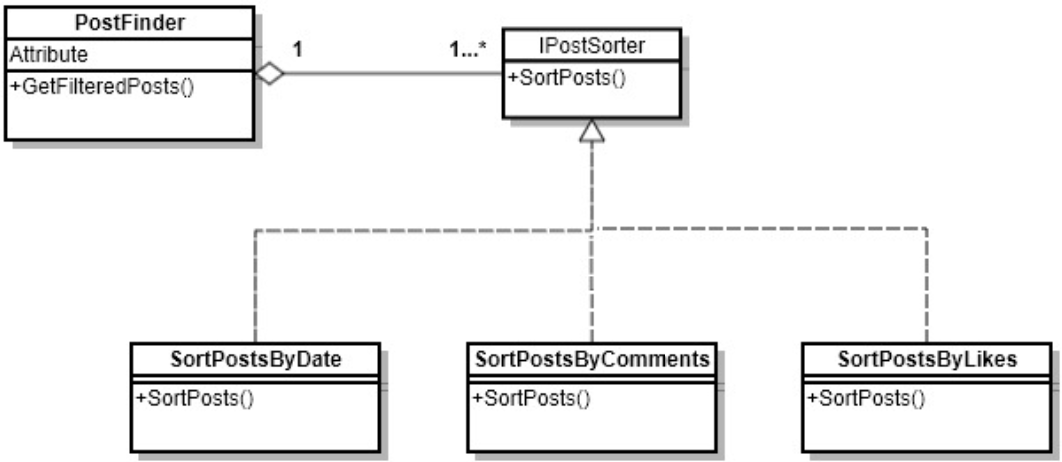
**ניתן למצוא את המימוש בקוד:**  
בפרויקט: UserFeaturesLogic  
בתיקייה: StrategyPostFinder וגם PostFinder  
המחלקות: IPostFinderSorter, SortPostsByComments, SortPostsByDate, PostFinder SortPostsByLikes.

**Sequence Diagram**



מוצג בתרשים הרצף אסטרטגיה למיון אחת לדוגמא SortedPostsByDate, שאר סוגי המיון מתוארים באופן דומה.

**Class Diagram**



**"השחקנים" בתבנית עיצוב:**

**Context:** PostFinder **Strategy:** IPostSorter **ConcreteStrategyA:** SortPostsByDate **ConcreteStrategyB:** SortPostsByComments **ConcreteStrategyC:** SortPostsByLikes

### תבנית מס' 5 – [Iterator]

1. סיבת הבחירה / שימוש בתבנית:

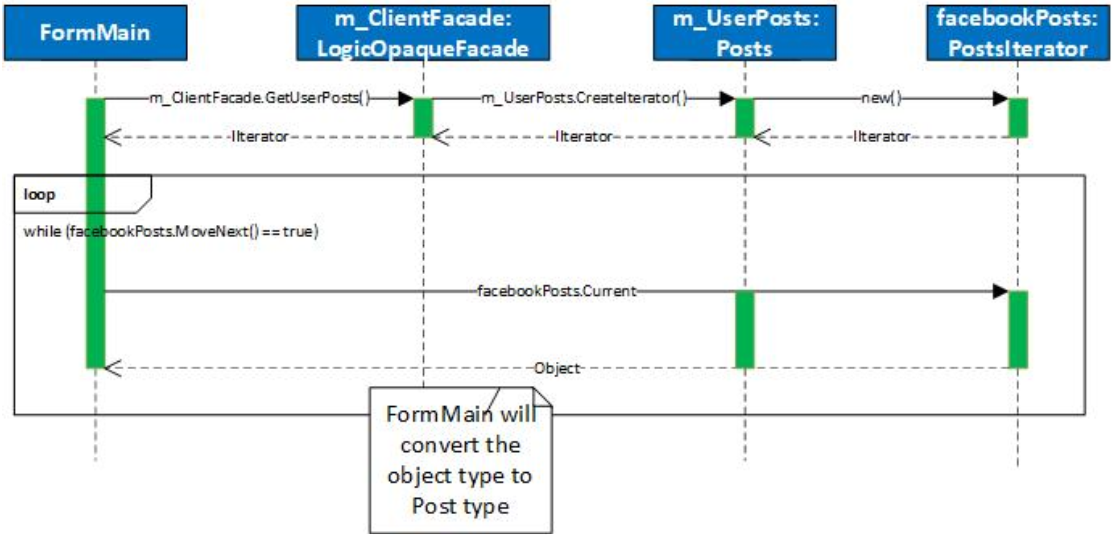
בחרנו לממש את התבנית Iterator,  
בנינו מחלקה בשם Posts שיש בתוכה אוסף פוסטים של המשתמש.  
נרצה לתת ל Client איטראטור כדי להסתיר ממנו את המבנה הנתונים ששומר את אוסף הפוסטים.  
מכיוון שאם נרצה לשנות בעתיד מבנה נתונים אחר נוכל לשנות מבלי לעדכן את ה Client ובכך לא נצטרך לשנות את הקוד אצל ה Client.  
בנוסף, גרמנו לקוד של ה Client להיות נקי יותר כשהבאנו לו iterator לעבור על אוסף הפוסטים מבלי הצורך שהוא יממש את האלגוריתם סריקת האוסף.  
ה Client גם לא יכול להוריד או להוסיף נתונים חדשים לאוסף הפוסטים.

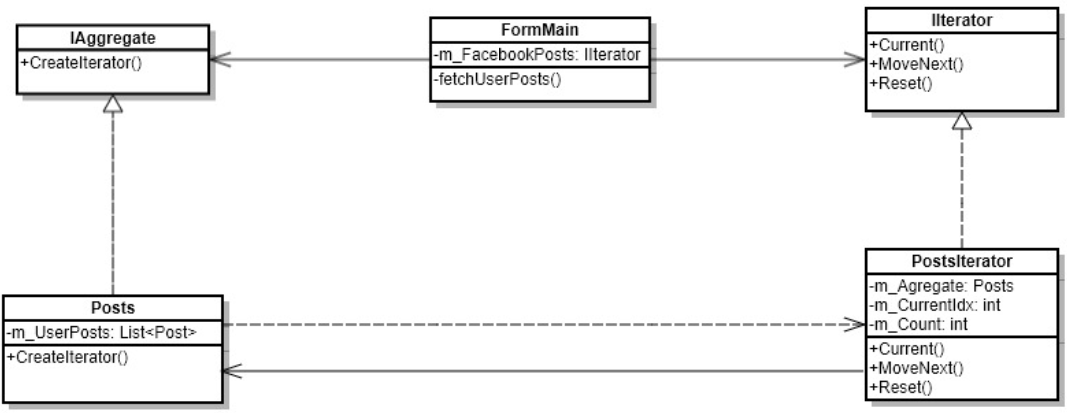
1. אופן המימוש:  
   בנינו ממשק בשם IIterator שמכיל את המתודות וה properties הבאים:
2. MoveNext
3. Current
4. Reset

וממשק IAggregate שמכיל את המתודה CreateIterator. שתפקידה להחזיר ל Client iterator, (במקרה שלנו להחזיר בסוף למחלקה FormMain) .  
  
במחלקה Posts שמחזיקה את האוסף של הפוסטים, אנחנו יורשים את הממשק IAggregate וממשמשים את המתודה CreateIterator שמחזירה איטראטור על מנת שה Client יסורק את כל האיברים באוסף של הפוסטים.  
  
בנוסף, בתוך המחלקה מימשנו private nested class בשם PostsIterator שיורש מהממשק IIterator וממש את המתודות לסריקה המבנה הנתונים.  
במקום לשלוח ל Client מבנה נתונים קרונקרטי שמכיל את האוסף הפוסטים.  
אנחנו יכול לשלוח לו איטראטור מבלי לחשוף לו את המבנה הנתונים של האוסף.

**ניתן למצוא את המימוש בקוד:**  
בפרויקט: UserFeaturesLogic  
בתיקייה: IteratorPosts  
במחלקות: IAggregate, IIterator, Posts

**Sequence Diagram**



**Class Diagram**  


**"השחקנים" בתבנית עיצוב:**

**Client**: FormMain  
**Aggregate:** IAggregate  
**ConcreteAggregate**: Posts  
**ConcereteIterator:** PostIterator  
**Iterator**: IIterator

### תבנית מס' 6 – Observer]]

1. סיבת הבחירה / שימוש בתבנית:

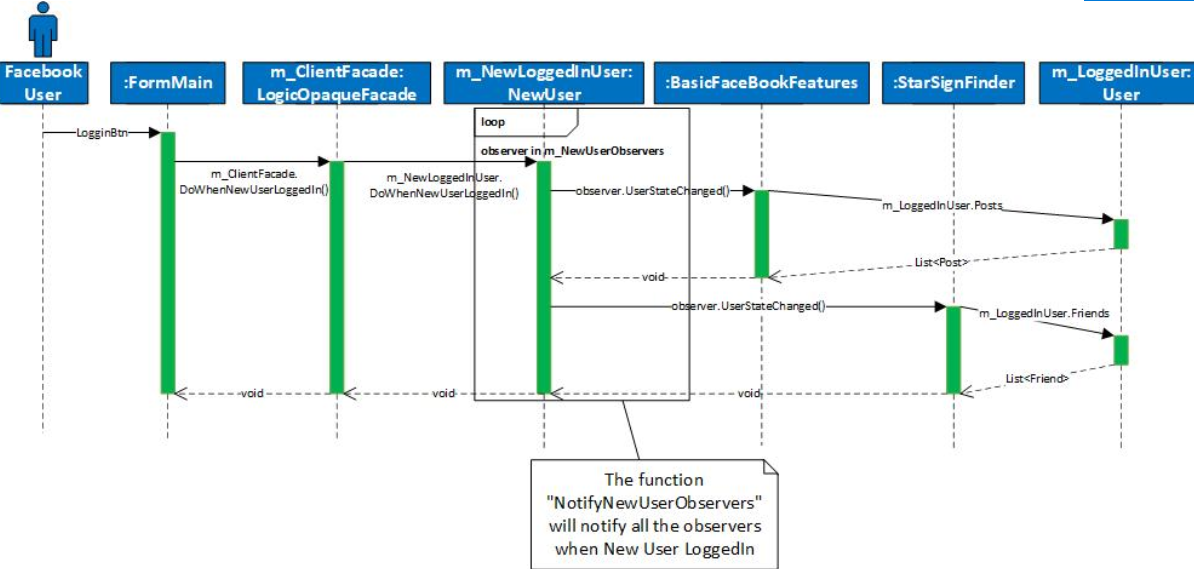
בחרנו בתבנית Observer Pattern מכיוון שאנחנו רוצים ליידע מחלקות אחרות בעת שינוי באפליקציה שלנו.  
ברגע שמשתמש נכנס לאפליקציה, אנחנו רוצים לעדכן את כל המחלוקות הרלוונטיות ולהודיע להן שיש משתמש חדש במערכת על מנת לקנפג (לאתחל את כל המידע שנדרש לאפליקציה עבור אותו משתמש חדש, כמו שליפת כל הפוסטים של המשתמש משרת הפייסבוק והצגתן בממשק משתמש או אתחול רשימת החברים בזיכרון זמני בשביל שנוכל להשתמש מהר יותר כאשר נרצה להשתמש בהם).  
נוכל לעשות זאת בצורה דינמית ( configured dynamically) .  
  
המחלקה NewUser היא מחלקה שמקבלת חיווי מהFormMain בעת התחברות יוזר חדש למערכת (כאשר המשתמש לוחץ על הכפתור Login ומתחבר לאפליקצייה באמצעות הפייסבוק).  
המחלקה NewUser גם יודעת להודיע לאובייקטים שמחזיקים אותה שהתחבר לאפליקציה משתמש חדש.  
לכן כל המחלקות (ה observers) במערכת אשר מאזינות לשינויים, יכולות לעדכן בזמן השינוי, את הנתונים שלהן ובכך לקנפג את האפליקציה עבור המשתמש החדש.  
  
לא נצטרך לשנות כלום בקוד ב NewUser אם נרצה בעתיד להוסיף עוד מאזינים (מחלקות שיקבלו הודעה כאשר נכנס יוזר חדש לאפליקציה).

1. אופן המימוש:

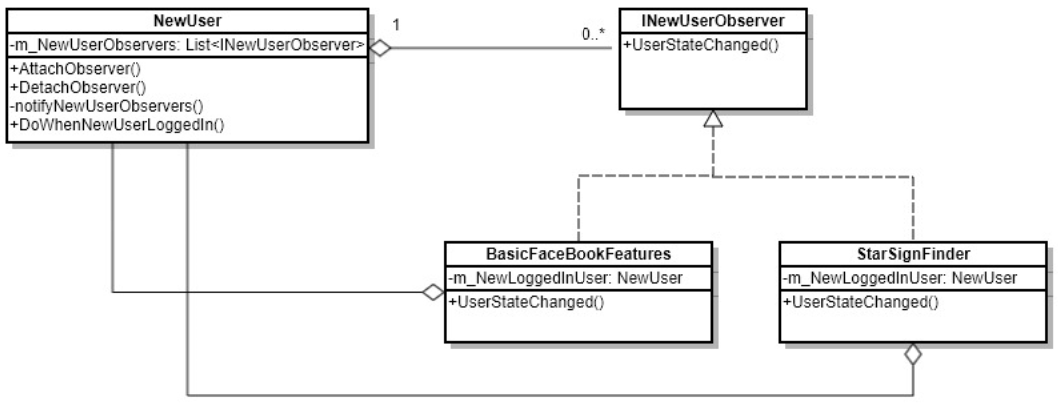
בנינו ממשק בשם INewUserObserver שמכילה את המתודה UserStateChanged   
ה Observable(Subject) נמצא במחלקה NewUser ושאר המחלקות Observers/listeners הן:BasicFacebookFeatures, StarSignFinder .  
  
כל המחלקות המאזינות מחזיקות רפרנס ל NewUser  
ואז יכולות להירשם אל המחלקה ולהאזין כאשר יש שינוי ב State (במקרה שלנו ב User).  
כלומר כאשר יש יוזר חדש באפליקציה, המחלקה NewUser תודיע לכל המאזינים שלה ותפעיל את המתודה UserStateChanged.  
והמחלקות המאזינות יוכלו לעדכן את הנתונים בהתאם למשתמש האפליקציה.

כלומר בעזרת המחלקה BasicFacebookFeatures נוכל לייבא משרת הפייסבוק את הפוסטים של המשתמש החדש. ובעזרת המחלקה StarSignFinder נוכל לייבא משרת הפייסבוק את רשימת החברים לצורך אתחול רשימת החברים הרלוונטית של המשתמש החדש.

**ניתן למצוא את המימוש בקוד:**  
בפרויקט: UserFeaturesLogic  
בתיקייה: NewUserObservable, BasicFacebookFeatures, StarSignFinder  
במחלקות: INewUserObserver, NewUser, BasicFacebookFeatures, StarSignFinder

**Sequence Diagram**

**ClassDiagram**



**"השחקנים" בתבנית עיצוב:**

**Subject:** NewUser  
**IObserver:** INewUserObserver  
**Observers:  
ObserverA**: BasicFaceBookFeatures  
**ObserverB**: StarSignFinder