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

* StatusSearch (מומש בתרגיל הקודם במחלקה StatusSearch.cs)

חיפוש משפט או מילה מסוימת בכל הסטטוסים של החברים של המשתמש.

הפיצ'ר ממומש באופן הבא: כתיבת מחרוזת לחיפוש בתיבת החיפוש בראש הלשונית ולאחר מכן לחיצה על כפתור ה - Search להפעלת החיפוש.

במידה ונמצא סטטוס המכיל את המחורזת, הוא מפרסם את הפרטים הבאים:

* תאריך כתיבת הסטטוס
* שמו של החבר אשר רשם את הסטטוס
* מלל הסטטוס
* כמות הלייקים
* כמות התגובות

בנוסף, במידה וקיימים מספר סטטוסים המכילים את המחרוזת, כולם יודפסו.

* StatusAnalyzer (מומש בתרגיל הקודם במחלקה StatusAnalyzer.cs)

ניתוח של כל הסטטוסים של המשתמש באופן הבא:

* יוצג הסטטוס אשר קיבל את המספר הגבוה ביותר של לייקים. אם קיימים מספר סטטוסים עם מספר לייקים זה (הגבוה ביותר), כל הסטטוסים האלה יוצגו.
* יוצג הסטטוס אשר קיבל את המספר הגבוה ביותר של תגובות. אם קיימים מספר סטטוסים עם מספר תגובות זה (הגבוה ביותר), כל הסטטוסים האלו יוצגו.
* יוצג החבר אשר עשה הכי הרבה לייקים בסך הכל למשתמש. (ייתכן וזה המשתמש עצמו). במידה וישנם מספר חברים אשר עשו את אותה הכמות של לייקים (הכמות הכי גבוה מבין כולם), אז אחד מהם יבחר רנדומלית והוא יוצג.
* במידה ואכן האדם שעשה הכי הרבה לייקים הוא אכן חבר ולא המשתמש עצמו, תפתח אפשרות ללחוץ על כפתור נוסף, אשר יפרסם סטטוס ובו המשתמש מודה לאותו חבר ויתייג אותו בסטטוס.

### תבנית מס' 1 – [Facade]

* סיבת הבחירה / שימוש בתבנית:

החלטנו להשתמש בפאטרן Facade כדי ליצור ממשק משתמש נוח יותר ופשוט להבנה. הFacade מבצע קישור בין הלוגיקה המורכבת לUI באופן בו יהיה פשוט וקל יותר לUI לעבוד עם הלוגיקה. באמצעות פאטרן זה נוצרה הפרדה בין החלק הלוגי לחלק של ממשק המשתמש במובן המאפשר שימוש פשוט יותר שכן מחלקה מנהלת את משימות החלק הלוגי והUI שהוא הקליינט שלה, לא צריך ליצור אינטרקציות מרובות עם הרובד הלוגי שכן הFacade עושה זאת עבורו בצורה נוחה על ידי כך שה-UI "מדבר" אך ורק עם ה-Facade אשר הוא אחראי על כל מחלקות הלוגיקה.

בנוסף בעתיד, אם היה ונצטרך להשתמש ב-UI שונה, לא נצטרך לפנות לחלקים הלוגים מחדש שכן הכל מתבצע דרך הFacade ולכן נוכל להחליף UI באופן פשוט ונוח שכן מחלקה זו היא שימושית לכל UI שנרצה.

* אופן המימוש:

ה-Client הוא FormMainApp. הוא למעשה הUI והוא עושה שימוש ב-Facade. בכל פעולה של המשתמש הדורשת איזהשהו רובד לוגי, ה UI קורא למטודה הרלוונטית ומשם ה-Facade קורא לרובד הלוגי הרלוונטי ומחזיר תשובה לUI בצורה פשוטה שעובדה כבר מראש בתוכו.

יצרנו Facade אטום שכן כל מחלקות שאחראיות על לוגיקה נמצאות בDLL נפרד יחד איתו, הוא המחלקה public היחידה בDLL- זה ויתר המחלקות הם internal. למעשה רק הוא יכול לפנות לרובד הלוגי של המערכת.

ה"שחקנים" בתבנית:

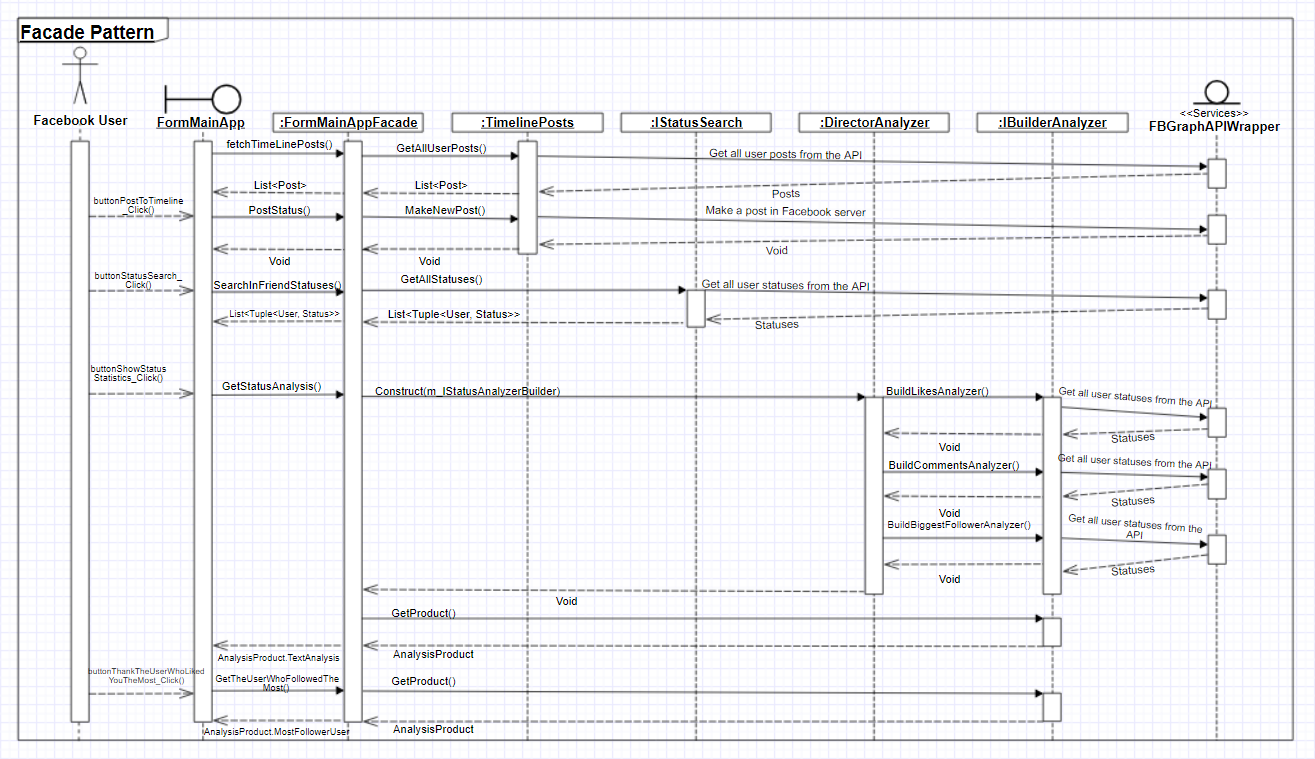
**Client**: FormMainApp.cs

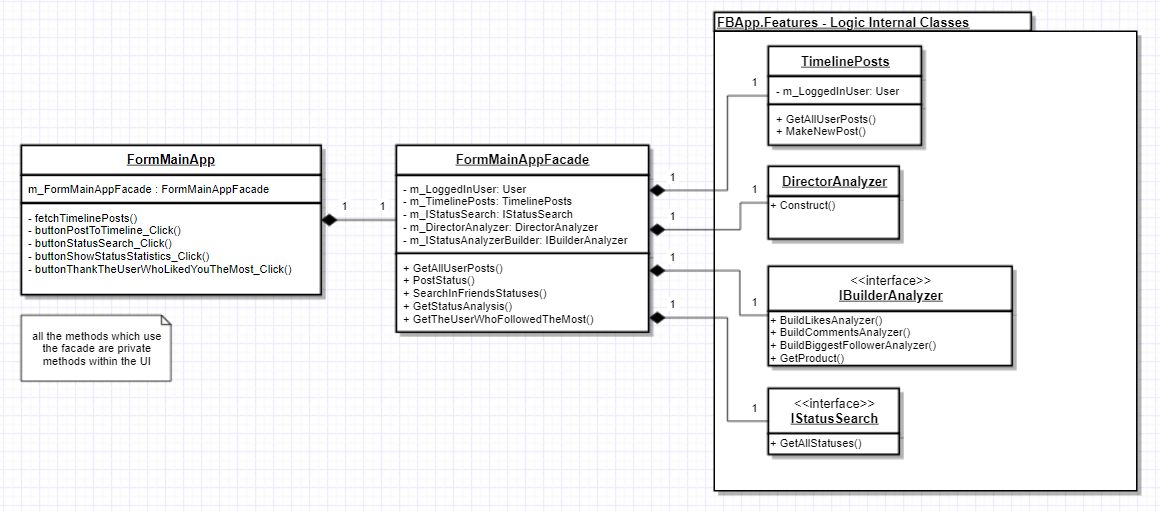
**Facade**: FormMainAppFacade.cs

**SubsystemA**: TimelinePosts.cs

**SubsystemB**: IStatusSearch.cs

**SubsystemC**: IBuilderAnalyzer.cs ,DirectorAnalyzer.cs

* Sequence Diagram
* Class Diagram



### תבנית מס' 2 – [Proxy]

* סיבת הבחירה / שימוש בתבנית:

מימשנוproxy המשמש כ- cache עבור הפיצ'ר של Status Search אשר בהינתן מחרוזת על ידי המשתמש מוצא את כל הסטטוסים בהם נמצאת מחרוזת זו אצל כל החברים שלו. תהליך מציאת כל הסטטוסים המכילים מחרוזת מסוימת עלול להימשך זמן רב מאחר שיש לעבור על מספר רב של אירועים ולעבדם. כתוצאה מכך, על מנת שלא נבצע כל פעם מחדש את תהליך המעבר על כל הסטטוסים של החברים של המשתמש בנינו מחלקת proxy המכילה בתוכה את המחלקה Status Search ובעצם מתחזה אליה. שומרת לאורך כל חייה של התוכנית את שלושת החיפושים האחרונים של מחרוזות שבוצעו ובכך אם המשתמש רוצה לחזור למחרוזת שביצע לאחרונה, אשר לא משנה כמה סטטוסים תכיל, הזמן שייקח לכל הסטטוסים להופיע יהיה קצר בהרבה.

יתרה מכך, אם המשתמש יכניס מחרוזת אשר אינה מופיעה בשום סטטוס יחזיר הודעה מתאימה ולא יכניסה אל מבנה הנתונים אשר משמש מחרוזות אחרונות שחופשו. בנוסף, החלפת המחרוזות האחרונות שחופשו יהיה מסונכרן באופן הבא: המחרוזת הראשונה שהמשתמש חיפש מבין כל המחרוזות שנשמרו היא זאת שגם תוחלף ראשונה.

* אופן המימוש:

המחלקה IStatusSearch אשר ממומשת כ – interface ומכילה בתוכה את ההכרזה של הפונקציה המרכזית והיחידה של StatusSearch ומשמשת מעין בסיס ל-proxy שלנו אשר יורש ממנה ונקרא .StatusSearchCacheProxy

הפרוקסי מכיל בתוכו member מסוג המחלקה StatusSearch. בכל פעם בו נשתמש בפיצ'ר של Status Search ב-UI במקום להגיע למחלקה הייעודית שלה StatusSearch נגיע בעצם אל StatusSearchCacheProxy שלנו אשר יבצע את כל מה שהמחלקה StatusSearch מבצעת ודברים נוספים באופן הבא:

* בדיקת המחרוזת הנתונה האם המשתמש הקליד אותה לאחרונה על ידי בדיקת המערך סטטוסים שלנו. במידה וכן, יחזיר את רשימת הסטטוסים התואמים למחרוזת. במידה ולא, יפנה אל המחלקה StatusSearch במטרה לקבל רשימת סטטוסים חדשה התואמים למחרוזת.
* לאחר מכן, יבדוק את הרשימה שקיבלנו ובמידה ואינה ריקה וקיימים סטטוסים המכילים אותה יכניס אותה אל מערך הסטטוסים שלנו תוך הוצאת המחרוזת הראשונה אשר נכנסה לרשימה מיתר המחרוזות.
* יחזיר את רשימת הסטטוסים הדרושה.

ה"שחקנים" בתבנית:

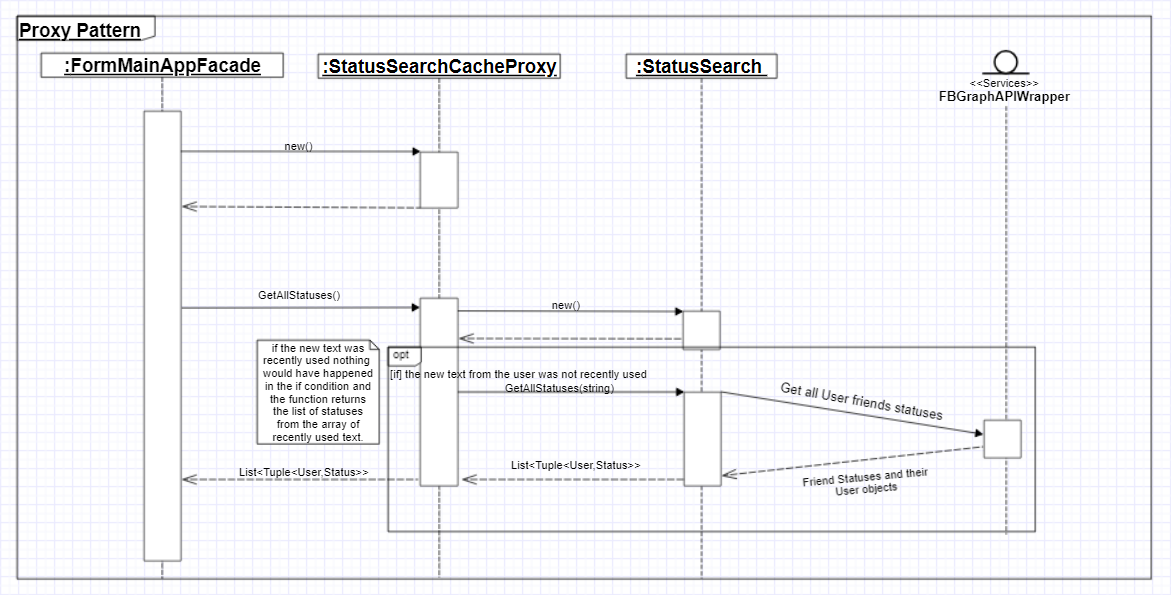
**Client**: FormMainAppFacade.cs

**ISubject**: IStatusSearch.cs

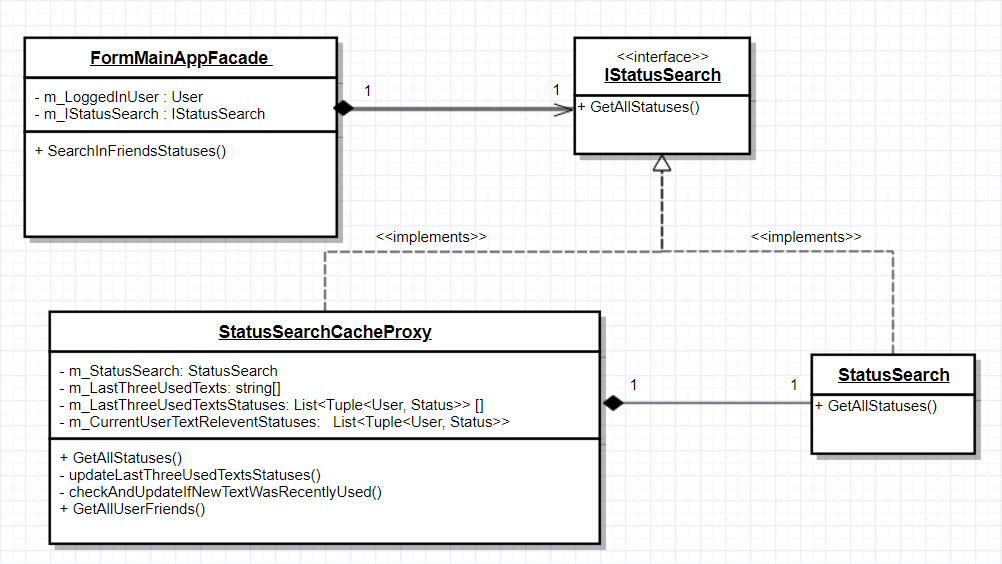
**Proxy**: StatusSearchCacheProxy.cs

**RealSubject**: StatusSearch.cs

* Sequence Diagram



* Class Diagram



### תבנית מס' 3 – [Builder]

* סיבת הבחירה / שימוש בתבנית:

במימוש הפיצ'ר StatusAnalyzer אשר מימשנו באפליקציית הפייסבוק שלנו ניתן לראות כי אפשר לחלק את התוצאה אשר נקבל עבור חיפוש אודות הסטטוסים של המשתמש ל-3 חלקים:

תוצאה עבור מספר הלייקים של הסטטוסים של משתמש,

תוצאה עבור מספר התגובות של הסטטוסים של משתמש

ותוצאה עבור מי הגיב ועשה לייק הכי הרבה לסטטוסים של המשתמש.

מכאן נקבל כי יש לנו תוצאה מורכבת שמכילה את כל המידע והוא מיוצג על ידיstring וכן רפרנס לUser הרלוונטי. את המחרוזת הזאת אנו רוצים להדפיס בסופו של דבר לתיבה המתאימה באפליקציה ולכן במקום לבצע את הבנייה שלה בפעם אחת אשר תהיה מורכבת ומסורבלת ולעיתים גם לא ברורה תלוי בגודל התוכנית ניתן לחלק את הבנייה שלו ל-3 חלקים שונים אשר כל אחד מוסיף עוד חלק בלתי תלוי לחלקים האחרים של המחרוזת. לאחר ביצוע 3 החלקים נקבל את ה-string הסופי אותו יש להדפיס לתיבה באפליקציה, הוא יוחזר באמצעות אוביקט מורכב שיצרנו שנקרא AnalysisProduct יחד עם ה-User הרלוונטי.

* אופן המימוש:

יצרנו מחלקה DirectorAnalyzer אשר מכילה בתוכה את שלבי הבנייה של האובייקט באמצעות אובייקט מסוגBuilder אשר יורש ממחלקה IBuilderAnalyzer אשר מהווה interface ומכיל הכרזות לכל השלבים שעל הBuilder לבצע והכרזה על החזרת המוצר הסופי.

IBuilderAnalyzer משמשת לבסיס פולימורפי במידה ונרצה לממש בעתיד עוד צורות של Analyzer ולא רק של סטטוסים ובכך מאפשרת לנו מגוון רחב של אפשרויות לשיפור והרחבה של הפיצ'ר בעתיד.

בנוסף, יש לנו את המחלקה שהיא הבילדר בפועל אשר יוצרת את ה-AnalysisProduct שהוא המוצר הסופי הרצוי. מחלקה זו נקראת StatusAnalyzerBuilder וממשמת את ה-interface IBuilderAnalyzer. המחלקה מייצרת שלב אחר שלב בעזרת המתודות המתאימות את הדרוש ולמעשה בונה את ה-AnalysisProduct בהתאם להגדרת הפיצ'ר.

ה"שחקנים" בתבנית:

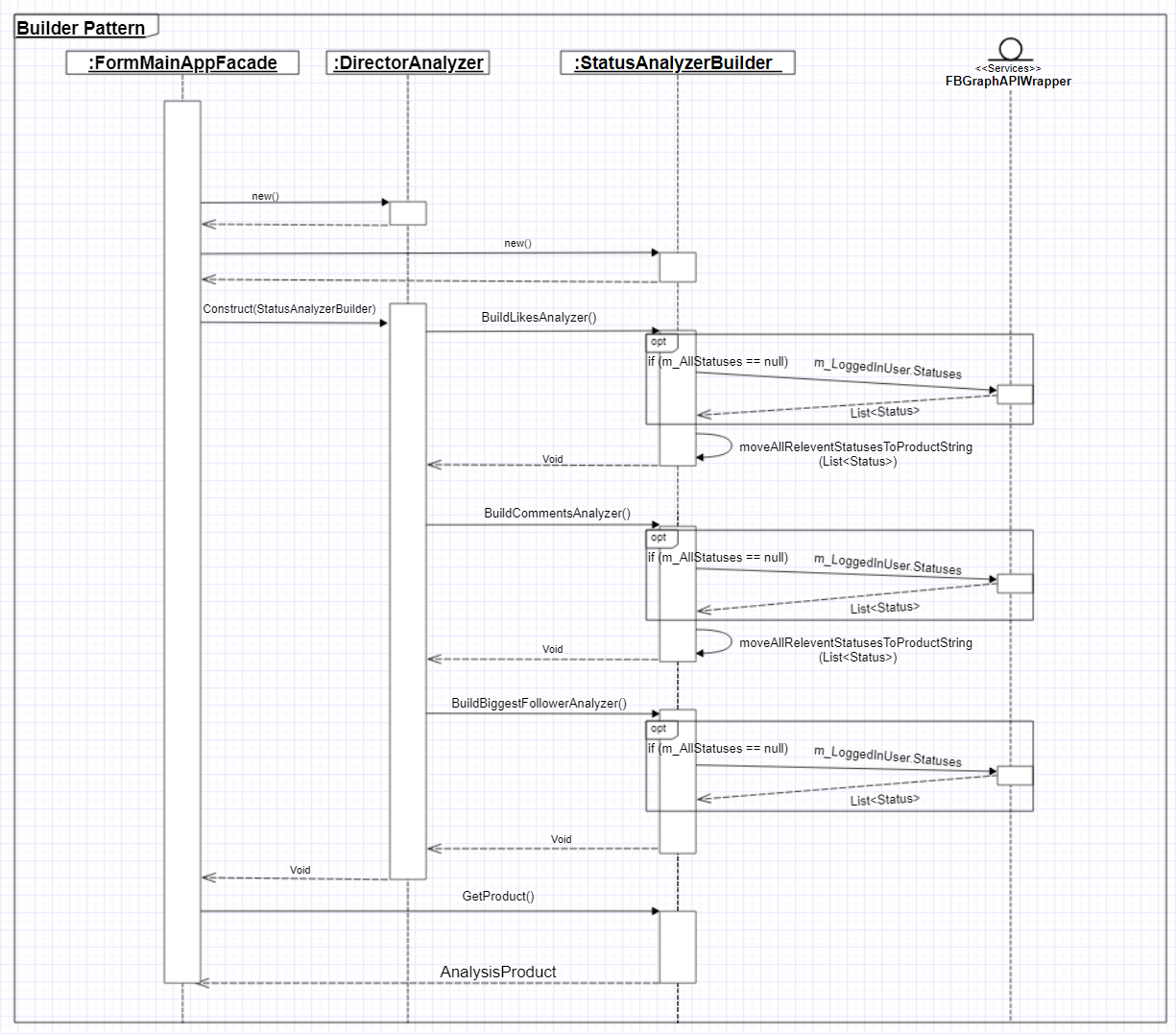
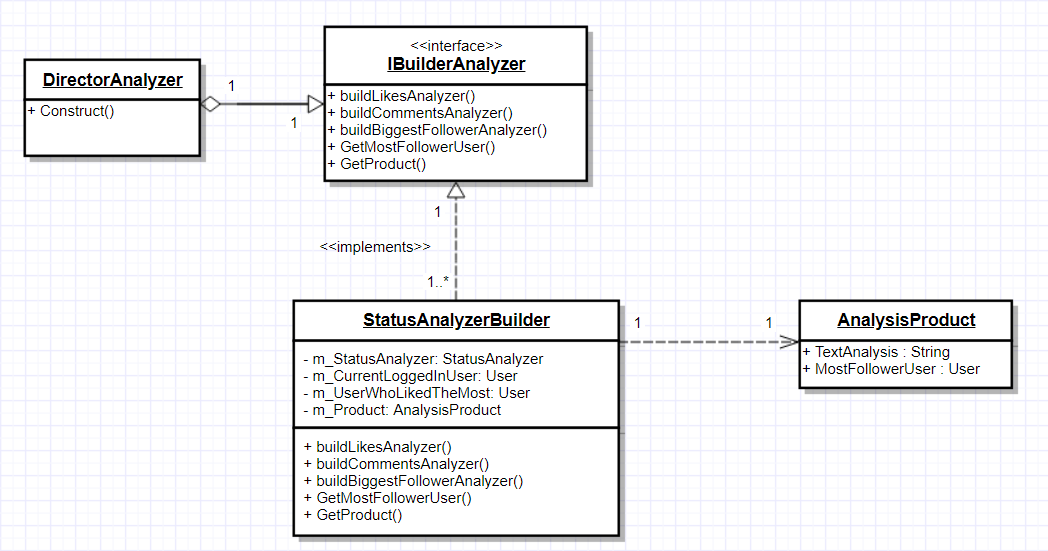
**Client**: FormMainAppFacade.cs

**Director**: DirectorAnalyzer.cs

**Builder**: IBuilderAnalyzer.cs

**ConcreteBuilder**: StatusAnalyzerBuilder.cs

**Product**: AnalysisProduct.cs

* Sequence Diagram
* Class Diagram

### עבודה אסינכרונית

בתרגיל השתמשנו מס' פעמים בעבודה אסינכרונית:

1. בהתחברות למשתמש אשר סימן שברצונו להיות זכור בכניסה הבאה למערכת.
2. בטעינת מידע בסיסי אודות המשתמש.
3. בטעינת האלבומים של המשתמש.
4. בטעינת הפוסטים של המשתמש.

סיבות לעבודה אסינכורנית:

הסיבה העיקרית לעבודה אסינכורנית במקרים הנ"ל היא קיצור זמן הטעינה של האפליקציה. מתודות אלו עובדות מול שרתי Facebook דבר אשר מהווה גורם מרכזי להאטה בקצב עליית האפליקציה באתחולה. שימוש במספר Threads במקביל, מנצל את יכולת השרת לקבל מספר פניות בו זמנית וככה מקצר את סך כל הזמן שיש לחכות לשליפת המידע.

כמו כן באמצעות שימוש זה, ה-UI של המערכת לא נתקע והמשתמש לא נאלץ להמתין עד אשר המידע נטען באופן מלא, יש למשתמש את היכולת להפעיל טאבים וכפתורים במערכת כפי שהוא רוצה מבלי להמתין להשלמת הטעינות הנ"ל. זאת מאחר והMain Thread- לא עסוק ואחראי על פעולות אלו.

אופן השימוש הוא באמצעות הפעלת Threads נפרדים עבור המטודות הרלוונטיות בתחילת הפעלת האפליקציה.

את השימושים ניתן למצוא בקוד במקומות הבאים:

* בקובץ FormMainApp.cs במתודות:

1. FormMainApp\_Load - קריאה למתודה connectToRememberdUser ב-Thread נפרד.
2. loadMainAppFeatures – קריאה למתודות:

fetchBasicUserInfoAndFillLabels

fetchTimelinePosts

fetchUserAlbums

כל אחת ב-Thread נפרד.

### עבודה עם Data Binding

בלשונית האלבומים מימשנו Data Binding בצורת One – Way פעמיים באופן הבא:

Data Binding ראשון מומש באופן הבא: בלשונית אשר מראה לנו את האלבומים שיש למשתמש הנוכחי בחשבון המשתמש שלו, ה – Data Bindig בא ליד ביטוי בכך שעל ידי לחיצת המשתמש על אלבום מסוים מבין רשימת האלבומים יקבל פרטים כללים על אותו אלבום (מספר תמונות, תאריך יצירה, תיאור כללי ומיקום).

Data Binding שני מומש בכך שאנו בוחרים תמונות מתוך האלבום הנבחר ותוכן התמונות ופרטים ספציפים לכל תמונה מתעדכנים ב UI בהתאם לבחירת המשתמש.

ה-Data Bindig בא ליד ביטוי בכך שעל ידי לחיצת המשתמש על אלבום מסוים מבין רשימת האלבומים יקבל בחלון נפרד את רשימת התמונות ובלחיצה על כל תמונה יקבל בחלון נפרד נוסף פרטים כלליים אודות התמונה(תאריך יצירה ותיאור כללי).

נשים לב כי מימוש ה-Data Binding שביצענו הינו Data Binding המוכל בתוך Data Binding.

את המימוש של שני Data Binding האלו ניתן למצוא:

**Project:** FBApp.UI

**Namespace:** FBApp.UI

**Class:** FormMainApp

**File:** FormMainApp.cs

**Methods:** fetchUserAlbums

listBoxAlbums\_SelectedIndexChanged