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

* **"שמירה במועדפים"** – פיצ'ר זה מאפשר לשמור פוסטים שהמשתמש יצר.   
  הפוסט נכתב לקובץ JSON ורשימת הפוסטים השמורים נטענת ברגע שהאפליקציה עולה שוב.
* **סטטיסטיקות אישיות** – פיצ'ר זה מאפשר למשתמש באפליקציה להציג עליו פרטים וסטטיסטיקות כלפי השימוש שלו בפייסבוק, למשל הצגת מספר חברים, כמות פוסטים וכו במקום מרוכז אחד.

### תבנית מס' 1 – Observer

* סיבת הבחירה / שימוש בתבנית:  
  במערכת שלנו כיום, כאשר משתמש מתנתק מהמערכת ומשתמש חדש מבצע התחברות, התהליך אפשרי רק כאשר המשתמש הישן מתנתק, סוגרים את התוכנה, מפעילים את התוכנה מחדש ורק אז המשתמש החדש מתחבר.  
  אנו מעוניינים ביכולת שכאשר משתמש מתנתק מהמערכת ומשתמש אחר מתחבר במקומו, לא תצטרך להתבצע הפעלה מחדש של התוכנית כולה, על מנת לשפר את חווית המשתמש ולהקל עליו.  
  לפיכך, נעזרנו בpattern observer, בעזרתו כאשר משתמש מתנתק מהמערכת, המערכת "מודיעה" לרכיבים הפנימיים המאזינים לאירוע של סגירת המערכת (כגון ה-FacebookApiFacade וה-Forms השונים) על ההתנתקות ובכך המאזינים יבצעו פעולות ש"יכינו" את המערכת לקראת השימוש בה של המשתמש הבא.  
  כלומר, כאשר משתמש מתנתק מהמערכת, ה-FacebookApiFacade "מנקה" את כל המידע הטמון בו (בתרגיל 2 הגדרנו את ה-facade להיות caching proxy, כלומר הוא מבצע טעינה פעם אחת של המידע ולאחר מכן אוגר אותו מבלי לבצע טעינה מחודשת) וכן המסכים השונים בתוכנה מנקים את התוכן מהם. לאחר מחיקת המידע של המשתמש הקודם, המשתמש החדש יכול להתחבר לאפליקציה מבלי צורך לפתוח אותה מחדש, תוך דאגה למחיקה של התוכן של המשתמש הקודם.
* אופן המימוש:  
  יצרנו מחלקה חדשה בשם Connection, בה אנו מרכזים את כל הפעולות של התחברות, התנתקות, ובנוסף אנו מחזיקים שם Action event בשם LogoutDetected, אליו נרשמים הרכיבים המעוניינים להגיב במצב בו משתמש מתנתק מהאפליקציה. בנוסף, הרכיב FacebookApiFacade מעוניין למחוק את המידע השמור בו כאשר משתמש מתנתק, לכן הוא נרשם ל-event כמאזין, וכאשר המשתמש מתנתק אזי ה-facade מבצע פעולת resetFacade. בנוסף, כל ה-Forms בתוכנה נרשמים כמאזינים לאירוע התנתקות, ומבצעים פעולת resetContent כאשר מתבצעת ההתנתקות.
* Diagram, schematic

  Description automatically generatedSequence Diagram
* Class Diagram

Diagram

Description automatically generated

### תבנית מס' 2 – Iterator

* סיבת הבחירה / שימוש בתבנית:  
  באפליקציה שלנו כיום, אנו מקבצים את כל הפוסטים של המשתמש תחת מבנה הנתונים רשימה של פוסטים, ואנו עוברים על רשימה זו בעת הצגת הפוסטים למשתמש.   
  אולם, נוצרו מספר בעיות עם מימוש זה. הבעיה הראשונה היא שאם בעתיד נרצה לשנות את מבנה הנתונים מרשימה לגרף למשל, ולעבור על הפוסטים בצורה שונה, אין באפשרותנו לעשות זאת מבלי לשנות המון קוד קיים. בנוסף, אין באפשרותנו לרוץ על רשימת הפוסטים עם פילטור כלשהו על הפוסטים עצמם ולהציג רק את הפוסטים שעומדים בתנאים מסויימים, מבלי להוסיף המון לוגיקות. לפיכך, אנו מעוניינים לא לחשוף את המבנה הפנימי של מי שמחזיק את הפוסטים של המשתמש, כדי שאם נשנה אותו בעתיד אז הקוד הקיים לא ייפגע. בנוסף, נרצה להוסיף יכולת לרוץ על מקבץ הנתונים בכמה דרכים שונות מבלי לפגוע במבנה הקיים. לכן עלה הצורך בשימוש ב-Iterator Pattern, שכן בעזרתו אנו לא חושפים את מבנה הנתונים שבו אנו מקבצים את הפוסטים, ושניתן בעזרתו ליצור כמה Iterators שונים שירוצו על מקבץ הפוסטים, כאשר כל איטרטור ממומש באופן שונה עם פילטורים שונים.
* אופן המימוש:  
  יצרנו תיקייה חדשה בשם Iterators ובתוכה את הקובץ PostsList. בתוך קובץ זה יצרנו את ה-interface-ים השונים, גם עבור מקבץ הנתונים וגם עבור ה-iterator עצמו   
  (IAggregate, IIterator). בנוסף יצרנו את המחלקה שמחזיקה את הפוסטים, PostsList, ובתוכה את ה-iterator שלה, PostsIterator. בנוסף עידכנו את ה-facade שיחזיק אובייקט m\_PostsList מסוג PostsList, ובמקום הפעולה(ב-facade) שמחזירה רשימה של פוסטים fetchPosts, שינינו אותה ל-GetPostsIterator ובהתאם החזרנו את PostsIterator.  
  לסיום, ב-FormPosts הפעלנו את הפונקציה GetPostsIterator, רצנו על האיטרטור שקיבלנו ובכך מילאנו את רשימת הפוסטים של המשתמש.
* Diagram

  Description automatically generatedSequence Diagram
* Class Diagram

Diagram

Description automatically generated

### תבנית מס' 3 – Visitor

* סיבת הבחירה / שימוש בתבנית:  
  אנו מעוניינים לבצע סגירת טפסים במערכת באמצעות כפתור "חזור" בכל טופס, אבל במימוש הנוכחי של המערכת לא ניתן לבצע זאת מבלי לשכפל את קוד סגירת הטופס בכל הטפסים אותם אנו רוצים לסגור. אנו מעוניינים לבצע פעולה זאת על ידי מחלקה חיצונית שתנהל את תהליך סגירת הטופס אצלה, כדי שאם בעתיד נרצה לשנות את האופן בו אנו סוגרים את הטופס נוכל לבצע זאת במקום אחד ולהימנע משכפול קוד בכל הטפסים.   
  לכן, מימשנו את הפטרן Visitor, שמקבל טופס ובעל יכולת לבצע על הטופס פעולת סגירה עם יכולת של הפעלת לוגיקות נוספות. ב-Visitor שמימשנו, הלוגיקה שביצענו היא שכאשר סוגרים טופס, יתבצע חילוף צבעים מהיר ברקע הטופס הנוכחי כדי לתת אינדיקציה יותר טובה למשתמש שהטופס נסגר. ה-Visitor אינו מכיר את הטפסים שמשתמשים בו, וכתוצאה מכך אם בעתיד נרצה לשנות את האופן בו טופס נסגר, למשל להוסיף אפקטים נוספים או להקפיץ הודעה למסך, נוכל לבצע זאת במקום ייעודי אחד, והוא ב-visitor עצמו.
* אופן המימוש:  
  המחלקה BackToLandingVisitor מכילה את הפונקציה ShowEffectAndGoToInitPage,  
  המקבלת טופס ועליו מבצעת אפקט כלשהו(במקרה שלנו, החלפת הרקע של הטופס), סוגרת אותו ועוברת למסך הראשי של האפליקציה. בנוסף, כל אחד מהטפסים באפליקציה מחזיקים מופע של visitor זה, וכאשר לוחצים על הכפתור close בכל טופס, הם מפעילים את הפעולה ShowEffectAndGoToInitPage ב-visitor, כאשר הטופס אותו הם מעבירים לפונקציה הוא הטופס הנוכחי, ובכך הטופס הנוכחי נסגר עם האפקט הרצוי מבלי לממש לגיקת סגירה שחוזרת על עצמה בכל הטפסים.
* Diagram

  Description automatically generatedSequence Diagram  
  \*הערה: דיאגרמה זו הינה ספציפית ל-FormPosts. הראנו דוגמה על טופס ספציפי אבל כך מתבצע התהליך עבור כל טופס. בכל טופס לוחצים על כפתור Close והתהליך יתבצע בדיוק באותה דרך.
* Diagram

  Description automatically generatedClass Diagram   
  \*ה-Form Class הוא ה-Form של System.Windows.Forms