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

* **Random Fact Generator** – בהתאם למשתמש המחובר, בלחיצה על כפתור ה- Fact Generator, מקפיץ למשתמש הודעה עם עובדה רנדומלית עליו.
* **Application Usage Limiter** – בהתאם לגיל של המשתמש המחובר, מגביל את זמן השימוש באפליקציה כאשר הגבלת הזמן היא בדקות לפי גיל המשתמש (גיל 25 – 25 דקות)

### תבנית מס' 1 – Adapter

* **סיבת הבחירה / שימוש בתבנית:**

בחרנו להשתמש בתבנית Adapter עם המחלקה SimplifiedUser מכיוון שבמהלך פיתוח האפליקציה הבנו שנדרש להתאים את המחלקה User של פייסבוק לצרכים של האפליקציה שלנו. כמעט בכל פעם שנדרשנו לגשת ל- User על מנת לקבל אובייקט מסוים (לדוגמא תמונת הקאבר של המשתמש המחובר), נדרשנו לבצע מספר פעולות שונות. מעבר לכך, המחלקה User מכילה הרבה תכונות שאיננו בהכרח זקוקים להן, וחלקן גם אינן פעילות.

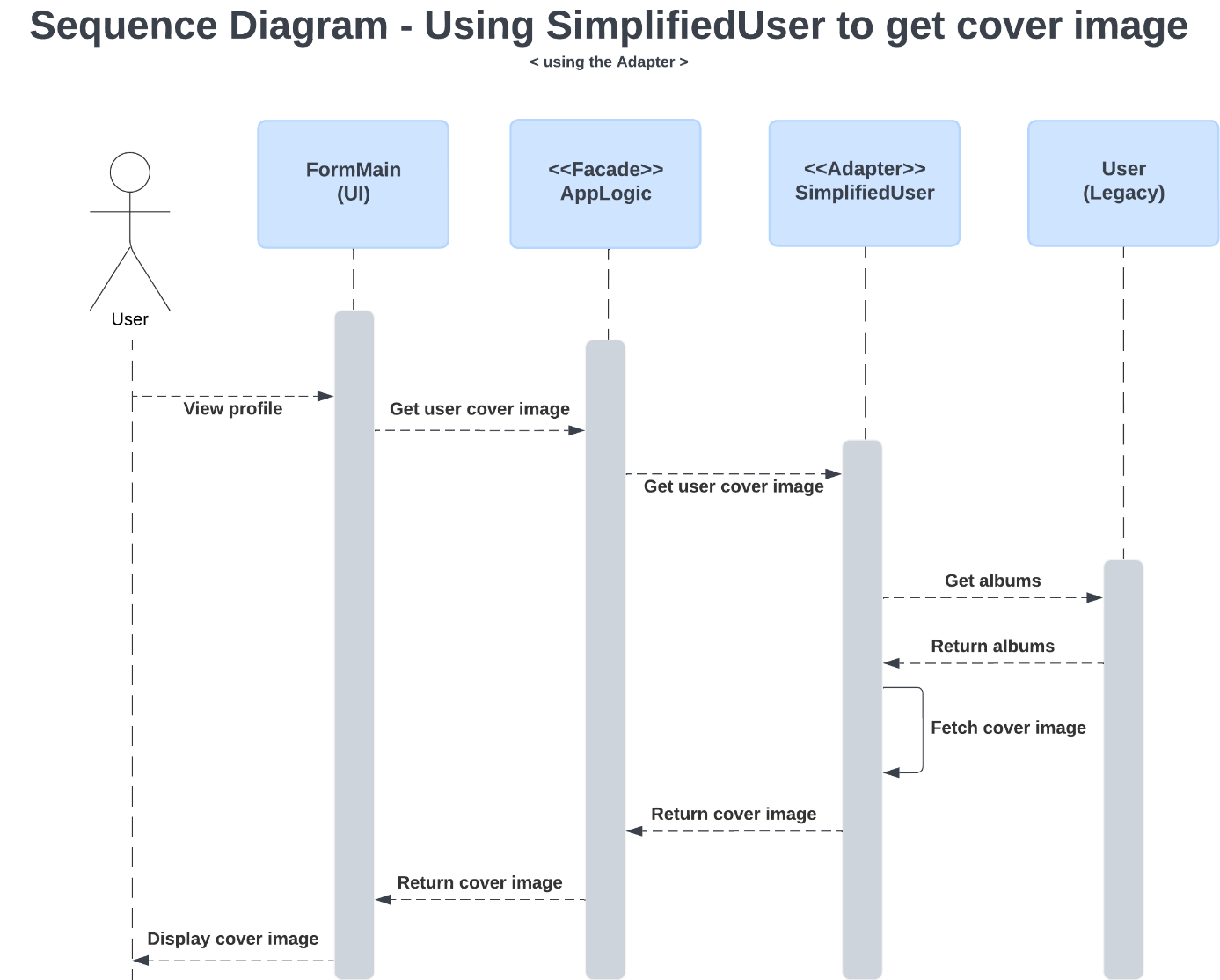
* **אופן המימוש:**

באמצעות המחלקה SimplifiedUser אנחנו יכולים להציג ולהשתמש רק במידע הרלוונטי עבורנו, ובכך לפשט את הקוד מבלי לשנות את המחלקה המקורית של פייסבוק (ה-Legacy Component). SimplifiedUser משמשת כמתווך ומספקת גישה פשוטה ונוחה למידע הדרוש.

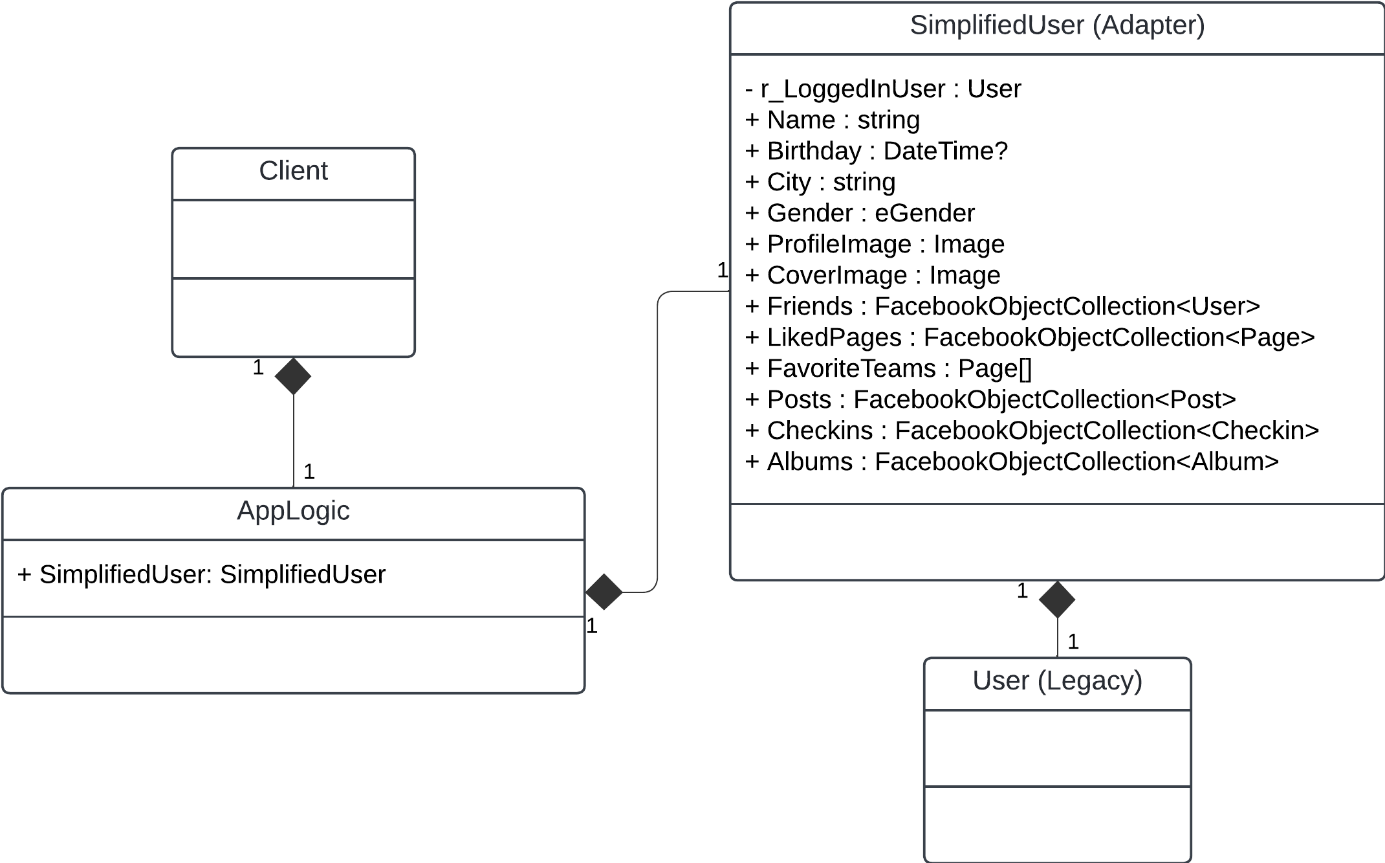
במחלקה AppLogic, קיים משתנה בשם m\_SimplifiedUser, אותו אנחנו מאתחלים עם אובייקט מהמחלקה SimplifiedUser. משם אנחנו משתמשים בו לביצוע פעולות שונות באפליקציה, כמו שליפת מידע להצגה בממשק המשתמש או ביצוע שינויים מקומיים בפרופיל המשתמש.

ניתן למצוא את המימוש של התבנית (והמחלקות) תחת תיקיית ה- Logic בפרויקט.

* **Sequence Diagram**



* **Class Diagram**

**עבור כל מחלקה שלכם, כיתבו מי המקבילה שלה (אם יש כזו) בתבנית (Pattern) שבחרתם**

### תבנית מס' 2 – Facade

* **סיבת הבחירה / שימוש בתבנית:**

בחרנו להשתמש בתבנית העיצוב Facade עם המחלקה AppLogic כדי לפשט את הגישה לרכיבים שונים באפליקציה ולרכז את כל הלוגיקה המרכזית במקום אחד. תבנית ה-Facade מאפשרת לנו להסתיר את המורכבות הפנימית של המערכת ולספק ממשק פשוט ואחיד לשימוש בממשק המשתמש, מבלי שהמתכנת יצטרך לדעת כיצד כל רכיב עובד מאחורי הקלעים.

* **אופן המימוש:**

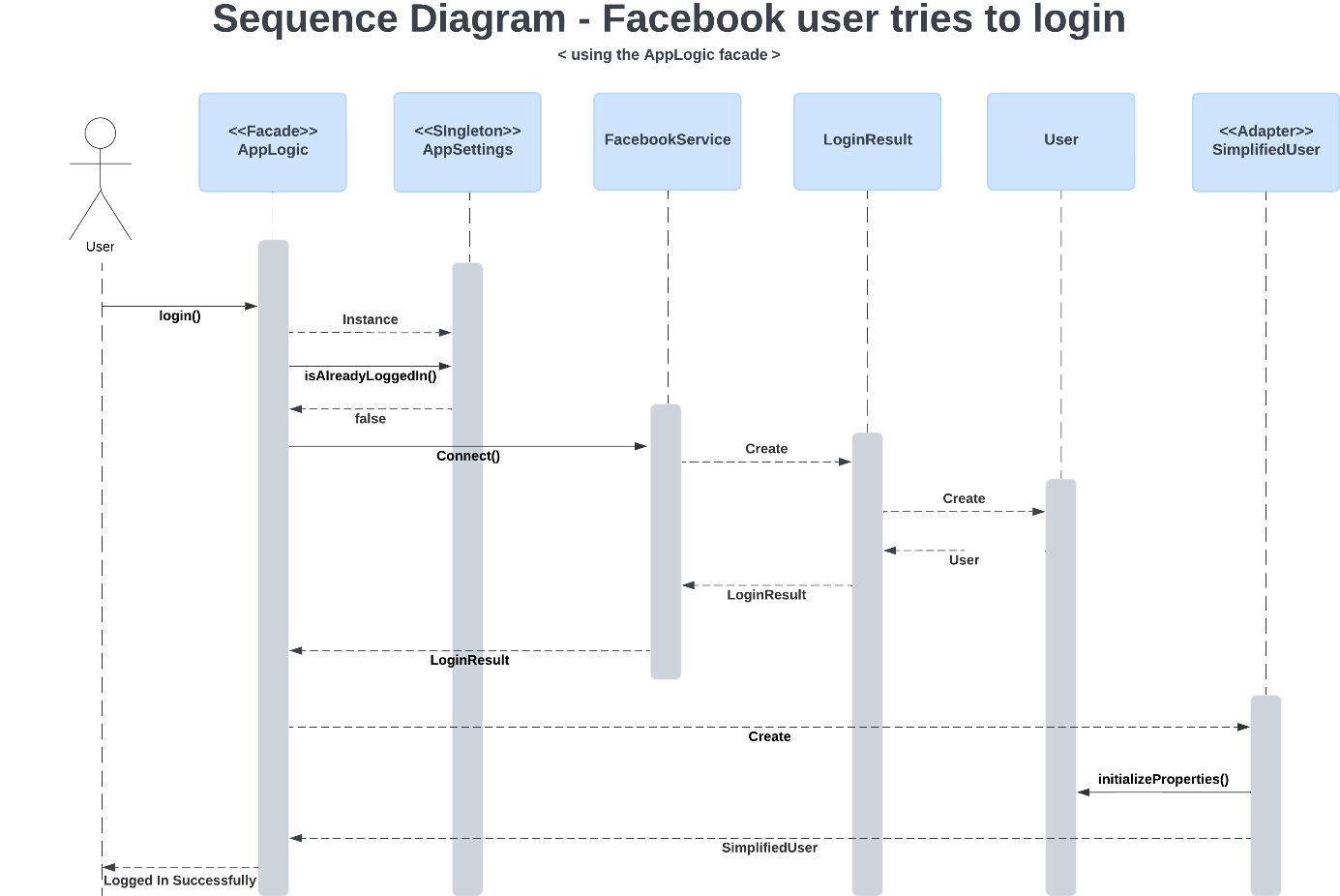
המחלקה AppLogic משמשת כנקודת גישה אחת לכל הלוגיקה באפליקציה, והיא מרכזת בתוכה פעולות כגון שליפת נתוני המשתמש, עדכון מידע בפרופיל, וטיפול באינטראקציות עם הפייסבוק. היא פועלת כמעין "חזית" (Facade) שמתקשרת עם מחלקות אחרות וטפסים שונים.

באמצעות AppLogic, רכיבים שונים באפליקציה כמו ממשק המשתמש יכולים לגשת בצורה פשוטה לפונקציות השונות מבלי להתעמק במורכבות הפנימית. כך אנחנו משפרים את הקריאות של הקוד ומאפשרים תחזוקה קלה יותר.

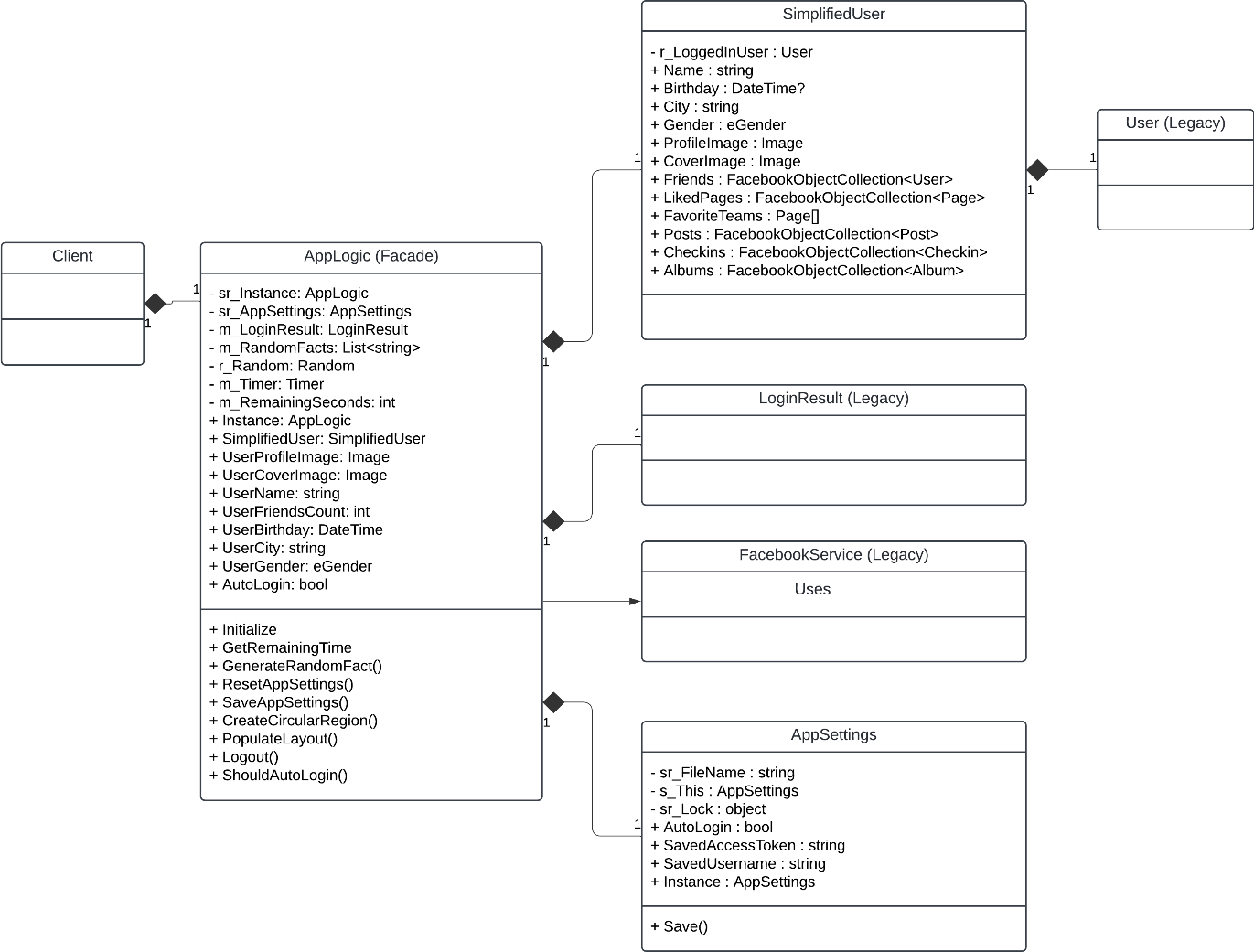
כל פעולה שמבוצעת באפליקציה, כמו שליפת פרטי המשתמש או עדכון פרופיל, מתבצעת דרך המחלקה הזו. היא מבצעת את האינטראקציות מאחורי הקלעים עם הרכיבים הנדרשים ומחזירה תשובות בצורה אחידה לממשק המשתמש.

ניתן למצוא את המימוש של התבנית (והמחלקה) תחת תיקיית ה- Logic בפרויקט.

* **Sequence Diagram**



* **Class Diagram**



**עבור כל מחלקה שלכם, כיתבו מי המקבילה שלה (אם יש כזו) בתבנית (Pattern) שבחרתם**

### תבנית מס' 3 – Template Method and Strategy

* **סיבת הבחירה / שימוש בתבנית:**

בחרנו להשתמש בשילוב של התבניות Template Method ו-Strategy כדי לטפל במילוי טאבים שונים באפליקציה בצורה גמישה ומודולרית. תבנית ה- Strategy מאפשרת לנו להגדיר אלגוריתמים שונים עבור כל טאב בממשק המשתמש (כמו טאב חברים, תמונות, קבוצות אהובות), כאשר כל אחד מהם ממומש כקבוצת לוגיקה נפרדת. בעזרת תבנית Template Method, כל אסטרטגיה יכולה לממש שלבים ספציפיים משלה בתוך תבנית פעולה כללית המוגדרת במחלקת האב, מה שמאפשר שמירה על מבנה כללי משותף, אך עם אפשרות להתאמה אישית של פרטי המימוש.

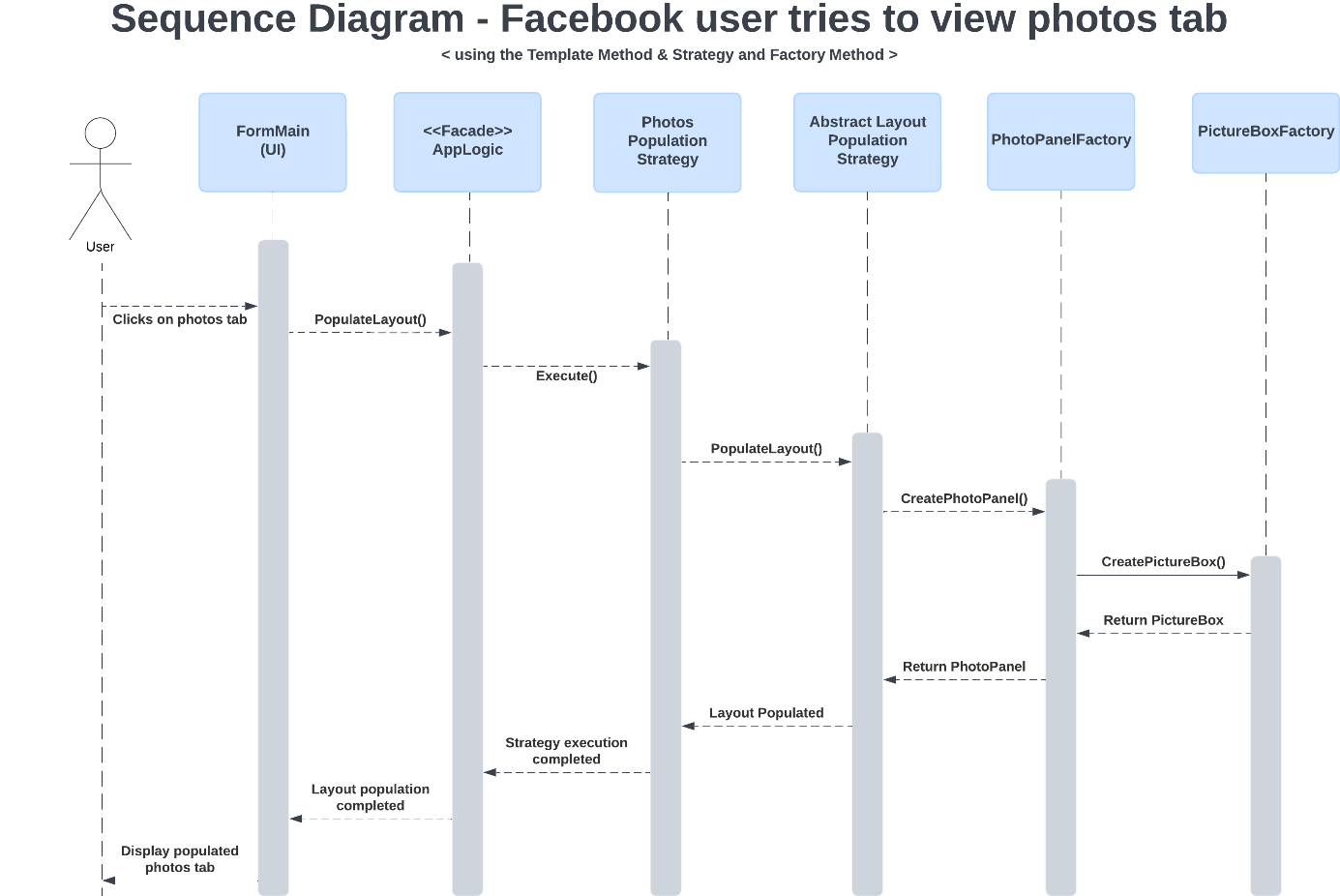
* **אופן המימוש:**

Strategy: כל טאב בממשק המשתמש מקבל אסטרטגיה שונה שמתאימה לו, כמו FriendsPopulationStrategy עבור טאב חברים, ו-PhotosPopulationStrategy עבור טאב תמונות. כל אסטרטגיה אחראית על קבלת הנתונים הרלוונטיים לאותו טאב ויצירת הפאנל המתאים להצגת הנתונים.

Template Method: כל אסטרטגיה מממשת את השיטה PopulateLayout, שבה נבנה תהליך כללי למילוי הטאב. כך שנוצר מבנה משותף לכל הטאבים, תוך מתן אפשרות לכל טאב ליישם פרטי פעולה ספציפיים בהתאם לצרכים שלו. זה מאפשר שליטה על התהליך הכולל, אך עם גמישות פנימית.

ניתן למצוא את המימוש של התבניות (והמחלקות) תחת תיקיית ה- Strategies בפרויקט.

* **Sequence Diagram**



* **Class Diagram**

A diagram of a company

Description automatically generated

עבור כל מחלקה שלכם, כיתבו מי המקבילה שלה (אם יש כזו) בתבנית (Pattern) שבחרתם

### תבנית מס' 4 – Factory Method

* **סיבת הבחירה / שימוש בתבנית:**

בחרנו להשתמש בתבנית Factory Method כדי להקל על יצירת רכיבי ממשק משתמש שונים (כמו פאנלים עבור צ'ק-אין, תמונות ופוסטים) מבלי לחשוף את המתכנת למורכבות של תהליך הבנייה שלהם. כל פאנל מיוצר באמצעות מתודולוגיה ייחודית המותאמת לסוג התוכן, כך שניתן להבטיח שהפאנל ייבנה בצורה נכונה עם כל האלמנטים הנדרשים.

* **אופן המימוש:**

כל אחת מהמחלקות, כגון CheckInPanelFactory, PhotoPanelFactory וכו', אחראיות על יצירת פאנל ספציפי.

המחלקות מספקות מתודה סטטית בשם Create<PanelType>Panel, שמקבלת פרמטרים כמו התוכן שצריך להציג (למשל, צ'ק-אין, תמונה או פוסט) וגודל הפאנל.

בתהליך הבנייה של הפאנל, כל פקטורי בונה את הפאנל עם התכנים המתאימים לו, כגון תוויות ותמונות, ומחזיר את הפאנל המוכן לשימוש.

ניתן למצוא את המימוש של התבנית (והמחלקות) תחת תיקיית ה- ComponentFactories בפרויקט.

**מכיוון ש- Template Method & Strategy עובד אצלנו יחד עם Factory Method, הדיאגרמות הן אותן הדיאגרמות**

* **Sequence Diagram**

A diagram of a diagram

Description automatically generated

* **Class Diagram**

A diagram of a company

Description automatically generated

עבור כל מחלקה שלכם, כיתבו מי המקבילה שלה (אם יש כזו) בתבנית (Pattern) שבחרתם

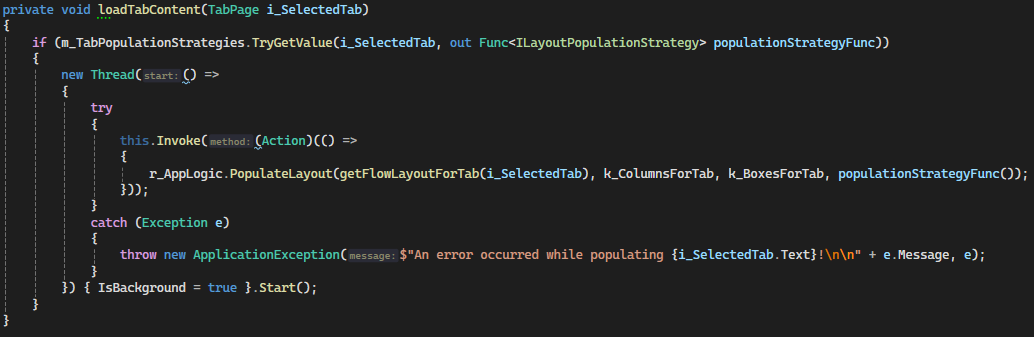
### עבודה אסינכרונית עם ממשק משתמש

* **סיבת הבחירה:**

השימוש ב-Thread מאפשר לבצע פעולות שמביאות נתונים לרכיבי ה-UI ברקע, מבלי לחסום את הממשק. זה מונע הקפאת ה-UI ומספק חוויית משתמש חלקה יותר.

* **אופן המימוש:**

ניתן לראות דוגמא למימוש מסוג זה בקוד בטופס ה- FormMain במתודה loadTabContent כדלהלן:



כמובן קיים גם שימוש ב- Invoke המשמש להפעלת קוד על Thread ה-UI. זה הכרחי כי לא ניתן לגשת לרכיבי UI מתוך Thread אחר. כך אנו מבטיחים שהשינויים המבוצעים על רכיבי ה-UI מתבצעים בצורה בטוחה.

### עבודה עם Data Binding

* **אופן המימוש:**

בטופס "EditProfileForm", אנו מממשים Data Binding על ידי חיבור רכיבי UI כמו TextBox, ו- DateTimePicker לשדות פרופיל המשתמש, כך שכאשר המשתמש משנה את המידע ברכיבים, הוא מתעדכן אוטומטית במשתנה המחזיק את המשתמש (m\_SimplifiedUser ב- AppLogic).