**תרגול 8 – OOP**

* **תרגיל 4:**
  + עוברים על הפורום לשאלות כל יום חוץ מרביעי ב-10 בבוקר
  + יש בפורום פוסט נעוץ עם עדכונים
* **קריאת תרשימי UML:**
  + UML היא שפה שמתכנתים משתמשים בה כדי לתכנן עיצובים ולהסביר אותן
  + מחלקה –
    - בחלק העליון שם
    - בחלק השני התכונות, data members
    - בחלק האחרון מתודות (לא תמיד עם חתימות מלאות)
  + אם מופיע + ליד תכונה או מתודה אז היא פומבית, - אומר שהיא פרטית ו# אומר שהיא protected
  + מחלקה אבסטרקטית תופיע בכתב נטוי.
  + יחסים:
    - קו ללא חץ – association
    - חץ עם משולש ריק – ירושה
      * יחס של is-a
    - חץ עם קו מקווקוו – realization (לפעמים)
      * מערכת יחסים בין ממשק לבין מה שמממש אותו
        + כשיש ממשק יהיה כתוב שהוא ממשק בחלק של השם
      * החץ יהיה בצד של הממשק
    - חץ עם מעוין במקום משולש – aggregation
      * יחסים של many-to-one
      * כלומר אני מחזיק מופע אחד אבל הרבה אובייקטים אחרים יכולים להיות מקושרים אליו
      * דוגמא – דקורטור – לרכיב אחד יכולים להיות הרבה דקורטורים (דקורטור אבסטרקטי שיכול להיות ממשק)

**תבניות עיצוב:**

* **תבנית עיצוב Observer:**
  + נועדה לתווך בין כמה מכשירים
    - למשל וואטסאפ שניתן לפתוח במחשב אנו רוצים שיהיה סנכרון בין הטלפון למחשב.
  + מבנה:
    - יש לנו ממשק או מחלקה אבסטרקטית עם פונקציית update שמתחייבת אם המשתמש משנה משהו היא תעדכן בכל אחד מהמכשירים
    - Subject – הממשק שמוצג ללקוח שהוא מחזיק אוסף של כל הobservers ואחראי לקרוא ל-update של כל אחד מהם (כלומר תהיה לו פונקציה שאחראית לכך)
      * מכיל פונקציות שמוסיפות ומסירות Observers
    - בגלל שהobserver הוא אבסטרקטי זה מבטיח שלכל observer קונקרטי תהיה פונקציית העדכון
* **תבנית עיצוב State:**
  + דוגמא – למשל בשעון – לכל ספרה יש לנו 8 קווים שנראים אותו דבר ולפי איזה מהם שדולקים נקבל ספרות שונות. כלומר יש לנו כמה אובייקטים שהם אותו דבר וכל מצב שלהם מוגדר אחרת לפי איזה מהם דולקים ואיזה מהם לא
    - יכולנו לעשות את זה עם ENUM (כי יש לנו 10 ספרות שזה מספר סופי)
  + תבנית העיצוב state נוחה כאשר יש מעבר מובנה בין מצב למצב (כמו שעון, אני יודעת תמיד מה המצב הבא), זה נוח שבתוך האובייקט שלי יש משהו שמצריך אובייקט נוסף
  + מבנה:
    - ממשק או מחלקה אבסטרקטית state שמחזיק את הפונקציה handle שאומרת מה לעשות לפי המצב – מדליק ומכבה את מה שצריך
    - ממנו ירשו מחלקות קונקרטיות לפי כל המצבים השונים
    - יהיה לנו משהו ראשי (מחלקה) שיחזיק את כל המצבים הקונקרטיים והוא ידע לקרוא כל פעם למצב המתאים עם הפונקציה request
  + צריך לבחור להשתמש בו בזהירות – לפעמים זה סתם יסבך את המצב
  + תבנית עיצוב שימושית כאשר הוא לא מסבך מידי, נלך אליו כשיהיו לנו ENUMים מסובכים
  + יכול להיות דומה לאסטרטגיה
    - אבל אסטרטגיה משנה התנהגות לפי מה שנקבל, לעומת state שתמיד יתנהג אותו דבר
    - כלומר במהות שלהן הן שונות
  + כלומר אם יש לנו קבוצה סגורה של מצבים נלך או לENUM או ל-state, ואם יש לנו קבוצה לא סגורה (שיכולה תמיד להתרחב) נלך לאסטרטגיה
    - עבור התנהגויות
* **תבנית עיצוב Singleton:**
  + סינגלטון הוא מקרה מאוד נישתי.
  + נניח שיש לנו אובייקט שכל מה שאנו צריכים ממנו הוא מספר מופעים ספציפי – לא נרצה לאפשר למשתמש לייצר עוד מופעים שלו.

כמו כן, אנחנו רוצים שהשדות שלו יהיו כמו משתנים גלובליים

* + מבנה:
    - יהיה לו קונסטרקטור פרטי – אף אחד לא יכול ליצור מופע מחלקה שלו
    - הוא למעשה ייצר לעצמו מופע מחלקה וישמור אותו בשדה פרטי סופי וסטטי –

Private static final Singleton singleObj = new Singleton()

* + - הוא יחזיק מתודה פומבית getInstance() ורק כך נוכל לגשת אליו
      * תחזיר את המשתנה הסטטי הקבוע של המופע ש"יצרנו"
    - נגדיר את השדות שלו כסטטיים ופרטיים כדי שלא יהיו מקושרים למופע ספציפי
      * יהיו לנו מתודות שמחזירות את השדה
  + מה חייב להיות בסינגלטון?
    - קונסטרקטור מוחבא – פרטי או מוגן
    - מתודה סטטית פומבית שמחזירה את המופע
  + למה לא אוהבים סינגלטון?
    - אותה סיבה שלא אוהבים משתנים גלובליים
    - עושה בלאגן בקוד
    - קשה יותר לדבג אותו
    - סינגלטון מייצר תלות במחלקה שלו, כלומר אם הוא יחזיק מופע של משהו נוסף יכול להיות שנייצר תלות בלי לדעת לעקוב אחריה
    - לא כל כך אופי, יותר תכנות פרוצדורלי (כי תמיד נצטרך לקרוא לפונקציה)