מגישים :

1. רועי שוודרון – 200978294
2. סער אליעד - 204402598

שאלה 1

סעיף א:

התכנית תדפיס

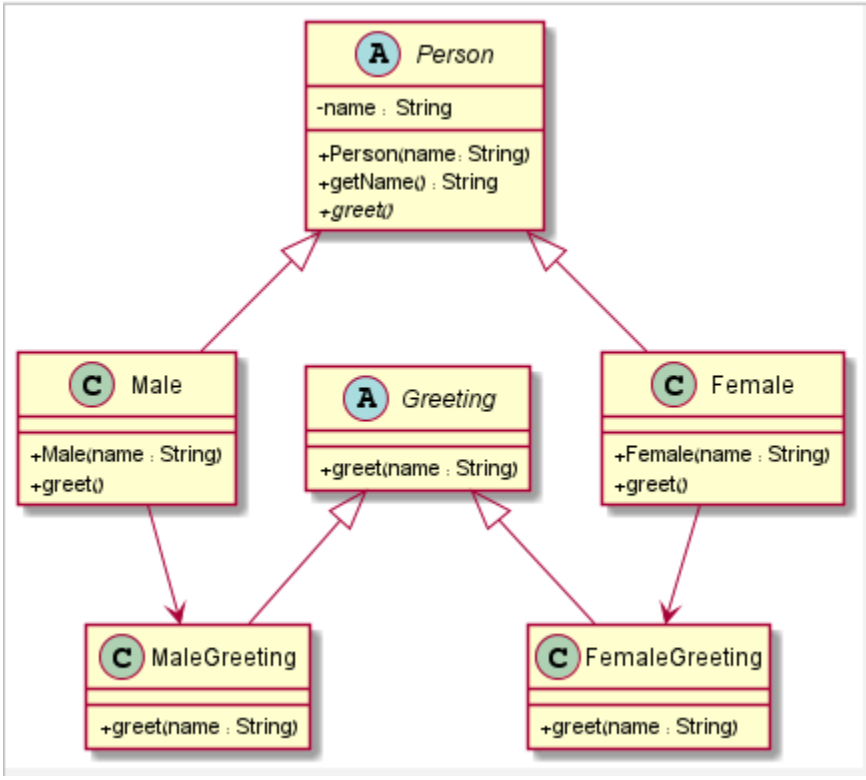
```
Hello Mr. Danny

Hello Ms. Danna
```

סעיף ב:

ממומש כאן design pattern של Factory Method. הוא פותר אותנו מהצורך לקשר מחלקות ספציפיות לקוד, יוצר נקודת הרחבה לתת המחלקות ומאפשר גמישות של כל מחלקה בנפרד.

כלומר הבעיה שהוא פותר היא שיש פעולות (כמו למשל ברכה לשלום) שאנו רוצים לבצע בצורה שונה עבור טיפוסים שונים (למשל נשים וגברים) אך במקום לפזר את ההחלטה לכל אורך הקוד, (כמו הוספת שדה מין, ובהתאם בחירת פעולה לפי מין בעזרת משפטי if) אנו עושים זאת במרוכז ובאופן מובנה.

סעיף ג':



אופן הפעולה של ה design pattern :

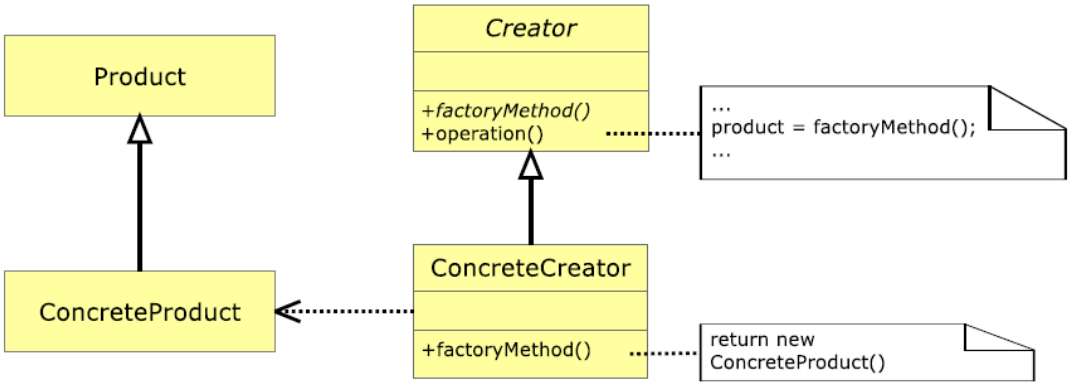
התוצר שאנו רוצים לקבל הוא ברכה. המחלקה האבסטרקטית Greeting מגדירה את הממשק לתוצר שאנו רוצים לקבל, ומכניסה מימוש חלקי לתוצר הזה.

המחלקות MaleGreeting,FemaleGreeting הן concrete products , כלומר ממשות את הממשק ש Greeting הגדיר עבור ברכה, אבל ברכה ספציפית לפי מין.

המחלקה האבסטרקטית Person היא בתפקיד Creator. היא מגדירה את הממשק של ה Factory Method שהוא greet().

המחלקות Male ו- Female הן בתפקיד ConcreteCreator, ממשות את הממשק ש Person ל- greet(), בעזרת יצירה ושימוש ב Concrete Product המתאים בהתאם למין.

הערה – התבססנו על המבנה הכללי של Factory Method מתוך התרגול:



אך כאן אצלנו המקווקו התחלף בחץ רגיל כי יש גם שימוש ולא רק יצירה כמו בדוגמאות מהתרגול.

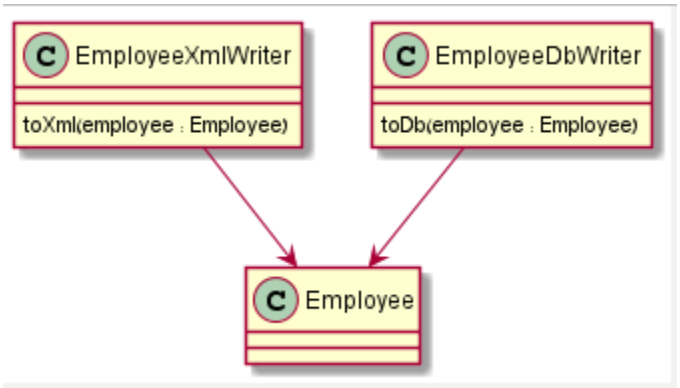
שאלה 2

סעיף א:

העקרון המופר כאן הוא עקרון האחריות היחידה. כעת, תהיה יותר מסיבה אחת לשנות את מחלקת העובד –

- סיבות שקשורות לתכונות של עובד
- סיבות שקשורות לפורמט ה XML
- סיבולת שקשורות למבנה\סוג של מבנה הנתונים

במקום זאת, אנו מציעים את המבנה הבא.



כעת, המחלקה Employee תהיה אחראית רק לתכונות שקשורות לעובד עצמו, והמחלקות EmployeeXmlWriter

EmployeeDbWriter יהיו אחראיות לרשום את התכונות של העובד לפורמטים הנכונים במבנה הנתונים\XML.

לדוגמה - כעת אם נשנה את מבנה הנתונים כך שימומש בעזרת סט במקום רשימה – אין הבדל מבחינת העובד, אלא רק השפעה על המחלקה EmployeeDbWriter שאחראית להכניס עובדים למבנה הנתונים.

סעיף ב:

מופר עקרון ההחלפה של ליסקוב.

למשל, אם יש למלבן פונקציה של setWidth, אנו נצפה שעבור מלבן היא תשנה רק את הרוחב, אך עבור ריבוע ההתנהגות לא מוגדרת היטב. אם למשל נבחר בריבוע לשנות גם את הגובה כתוצאה מ- setWidth - אנו עלולים להפתיע ולהביא להתנהגות של המשתמש לא מצפה לה.

סעיף ג:

עקרון הפתיחות\סגירות בא לידי ביטוי ב Factory Method בכך שניתן להוסיף עוד ConcreteCreators מסוגים שונים ו ConcreteProducts מסוגים שונים שירשו מהמחלקות Creator ו- Product בהתאמה, וכך למעשה להרחיב את הפונקצינליות של הקוד בעזרת הוספת מחלקות חדשות – מבלי צורך לשנות קוד קיים.

(וזאת מכיוון שקוד חיצוני מכיר בד"כ רק את ה- Creator (ולפעמים גם את ה – Product) ולכן יכול לפעול על כל מחלקה שיורשת ממנו – והקוד הפנימי – נמצא בעיקר ב ConcreteProducts ו ConcreteCreators ולכן נפרד מהמחלקות החדשות שיתווספו – ולכן גם כאן לא ידרשו שינויים).

עקרון "היוצר" מ GRASP בא לידי ביטוי ב Factory Method בכך שהבעיה ש Factory Method פותר היא שמחלקת האב, ה Creator, יודעת מתי ליצור אובייקט, ואיזה שימוש לעשות באובייקט, אבל **אין לה את המידע המספיק על מנת להחליט איזה אובייקט ליצור**. לשם כך, היא משתמש בממשק של ה- Product אך **דוחה את ההחלטה** ליצור אובייקט למחלקה היורשת, ה – ConcreteCreator, שדורסת את ה factoryMethod, ויוצרת את האובייקט הדרוש, ConcreteProduct.

כלומר, עקרון היוצר בא לידי ביטויי בכך שמי שיוצר את האובייקט (במקרה שלנו ConcreteCreator) **הוא מי שיש לו את המידע הדרוש לשם יצירת האובייקט.**

סעיף ד:

דוגמה לעקרון ה Creator מ- GRASP:

בטיפול שלנו בפעולות של מנהל המערכת, תמכנו באפשרות שמנהל המערכת יעדכן אפשרויות עבודה של מסעדה (כגון – זמן מראש שבו שומרים שולחן), יוסיף עובדי VIP, ויוסיף חברות חדשות למערכת.

אופן הפעולה שתכננו לפעולה זו: המנהל פותח חלון, מכניס פרטים בטיפוסים פשוטים (כמו למשל string) ולאחר מכן הפרטים עוברים המרה למבנים שהמסעדה עובדת איתם.

- (ה AdminHandler משמש סוג של adapter שאחראי לקריאה לפונקציות של המסעדה בעזרת הקלט הבסיסי מהמשתמש.)

הוא הראשון שיש לו את המידע הדרוש ליצירת מבנים בסיסיים, ולכן הוא יוצר אותם.

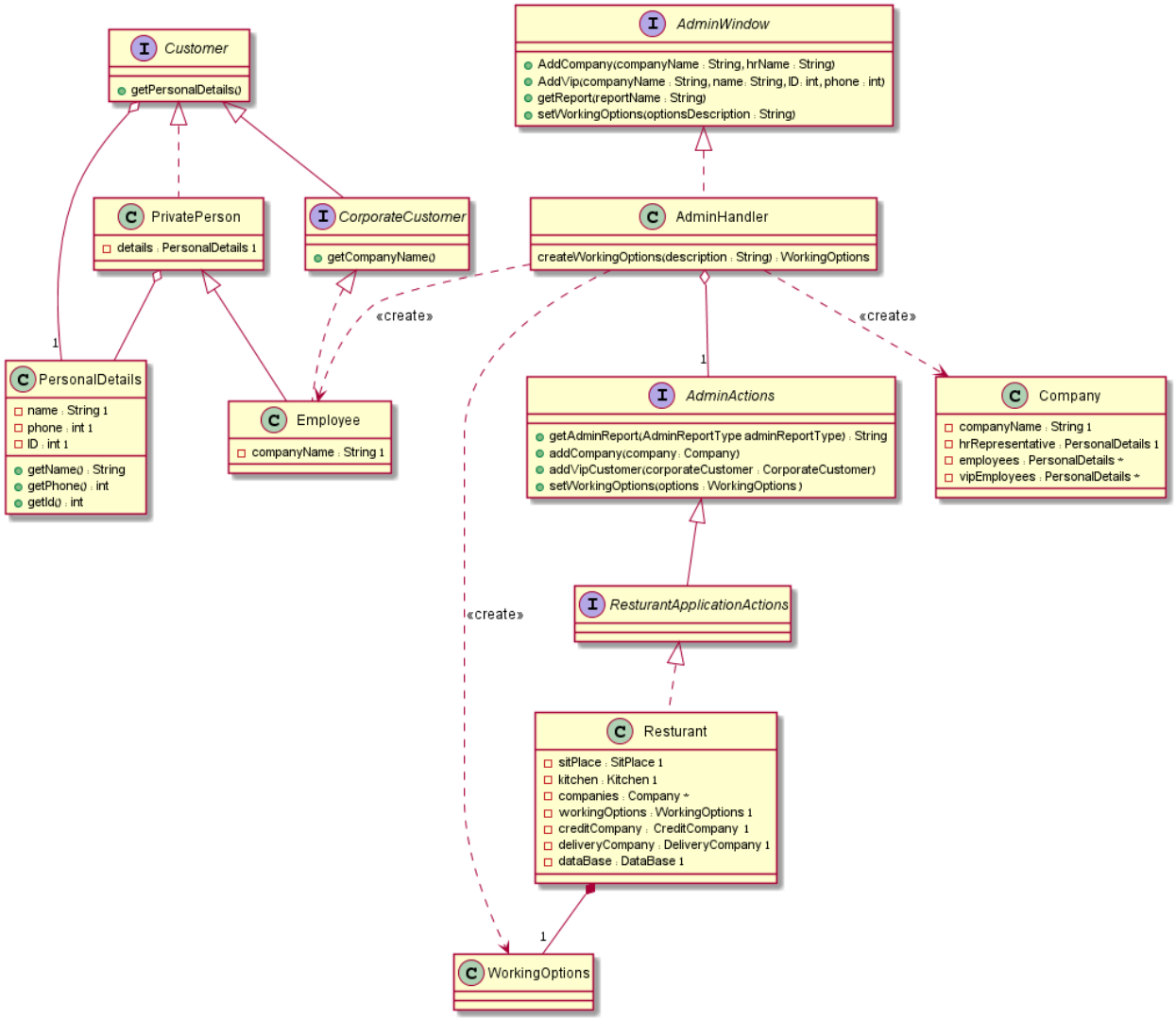
1. במקרה של חברה חדשה, הטיפוס הוא Company
2. במקרה של אפשרויות עבודה, הטיפוס הוא מחלקה WorkingOptions.
3. במקרה של עובדי VIP , הטיפוס הוא מחלקה Employee, שמממשת את הממשק CorporatePerson שבו המסעדה משתמשת.

בכל מקרים:

- מי שיוצר את האובייקטים הוא AdminHandler שהוא הראשון שיש בו את כל המידע על מנת אותם.
- מי שעושה שימוש באובייקטים היא המסעדה עצמה.

ראה דיאגרמה מצורפת מתוך תרגיל 3:

(הדיאגרמה חלקית, ומכילה רק מידע רלוונטי לסעיף זה, על מנת שתכנס כאן.הדיאגרמה המקורית קשה יותר לקריאה).

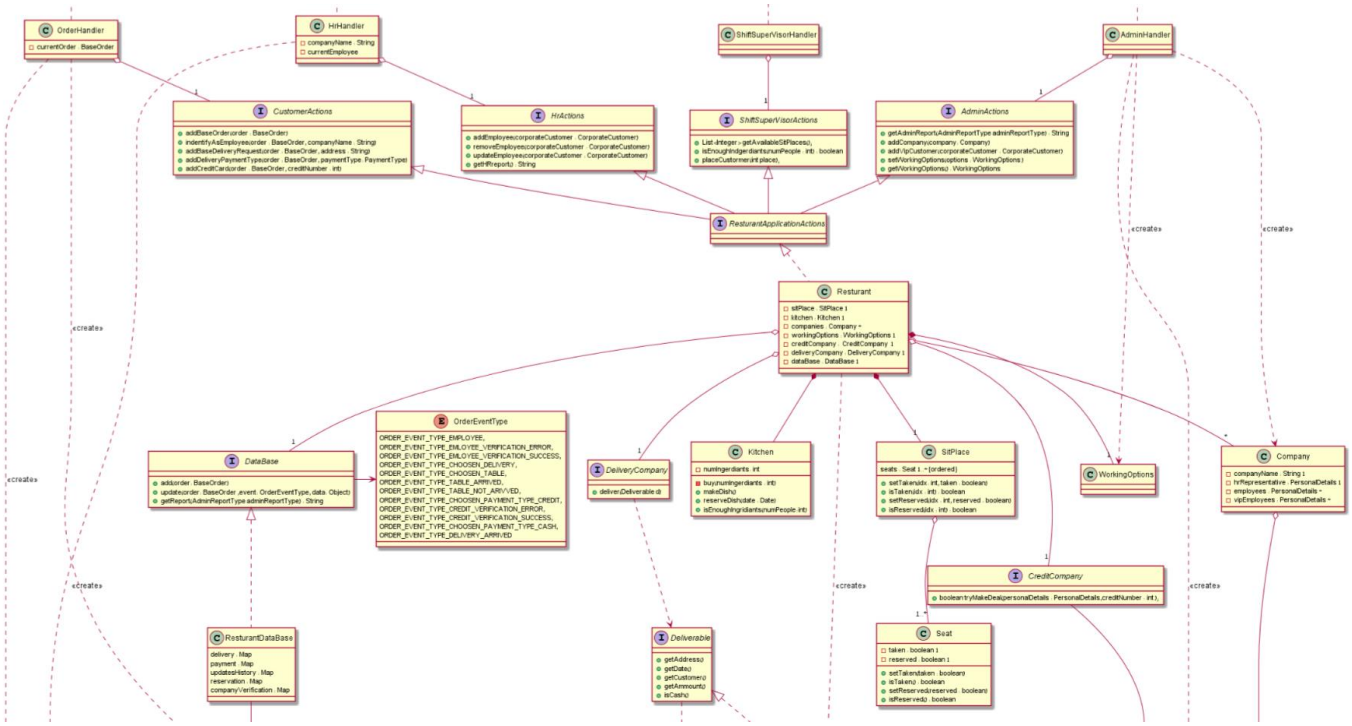


דוגמה לעקרון ה – Information Expert מ- GRASP:

עבור מרבית הפעולות במערכת יש צורך במידע מסוגים שונים:

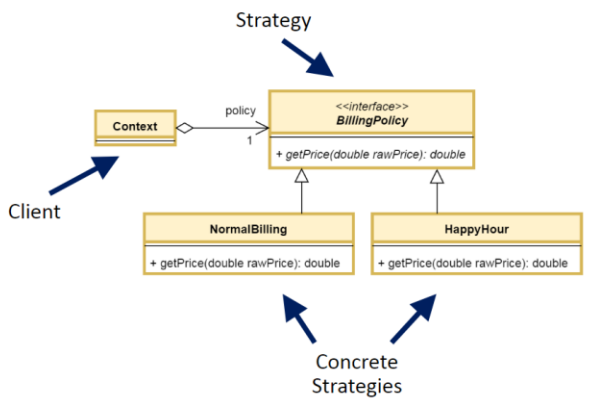
לכן, ניתן לראות שהמחלקה Resturant מממשת ממשק שמממש הרבה ממשקים שכוללים ביצוע פעולות מורכבות שדורשות המון מידע, ביניהן, למשל, הודספת אדם לחברה, או בדיקה האם ניתן להושיב אנשים.

- למשל, עבור הוספת אדם לחברה (נקרא ע"י ה HR) יש צורך בלבדוק האם החברה בכלל קיימת. המידע הזה נמצא אצל המחלקה Resturant שמחזיקה את רשימת החברות.
- דוגמה נוספת ניתן למשל לאמר לגבי המארח (ShiftSuperVisor) שכדי להושיב אדם צריך לבדוק אם יש מקום הושבה וגם אם יש מוצרים במטבח. מי שיש לו את כל המידע לשם כך היא המסעדה Resturant, שפונה למחלקות Kitchen ו- SeatPlace ולכן הלוגיקה של הפעולה תמצא בה.



סעיף ה:

לדעתנו ממומש כאן design pattern של strategy. ניתן לראות שהמבנה תואם את המבנה שנלמד בהרצאה:



כלומר:
ה-Client הוא JButton והוא תומך בהרבה סוגים של מסגרות. לכן, הוא מגדיר ממשק למסגרת ומשתמש בה.

הStrategy היא הממשק Border.

ה ConcreteStrategy בקוד הנתון היא מה שיוצרים בשורה 2 בעזרת BorderFactory, מדובר על LineBorder בצבע שחור.

בעזרת המתודה setBorder נותנים ל - JButton מסגרת.

סעיף ו:

לדעתנו ממומשת כאן ואריאציה של סינגלטון:

קודם כל ניתן להויכח מהקוד שקיים לכל היותר מופע אחד של SecurityManager.

הפקודה getSecurityManager() היא ואריאציה של getInstance() של סינגלטון, רק שכאן בניגוד ל getInstance() שלמדנו, לא מאתחלים את המופע אם הוא לא קיים, ולכן מחזירים במקרה זה null.

אפשר להניח שקיימת מתודה נפרדת שאחראית לאתחול של האובייקט.

שאלה 3 :

במימוש הלוח האלקטרוני השתמשנו ב-design patterns הבאים :

- Singelton – השתמשנו בדפוס זה עבור המחלקה ColorGenerator ממנה התבקשנו ליצור עצם אחד בלבד. בדומה לנלמד בתרגול, בחרנו להשתמש בדפוס זה במימוש lazy initialization מכיוון שהעדפנו שאובייקט ייווצר רק לאחר הקריאה הראשונה לבנאי ולא באופן אוטומטי בתחילת התוכנית. בנוסף מכיוון שאין צורך לאפשר הורשה הגדרנו את הבנאי כפרטי.
- Observer – בשאלה אנו נדרשים לשנות בסדר מסוים (ארבע אופציות נתונות) את צבעם של הלוחות בהתאם לשינוי בצבעו של האובייקט ColoeGenerator. החלטנו להשתמש בדפוס מכיוון שזהו מקרה קלאסי בו ישנם סובייקט (ColorGenerator) המשנה את מצבו באופן תדיר, וישנם אובייקטים משקיפים (Panels) המגיבים לעדכון במצבו של הסובייקט.
  לכן המחלקה ColorGenerator יורשת מהמחלקה הנתונה ע"י Java – Observable , והמחלקה Panel מממשת את הממשק הנתון ע"י Java – Observer .
  המימוש שלנו עבור ColorGenerator כלל מימוש רשימה של אובייקטים צופים, ודריסה של המתודות addObserver(Observer) , deleteObserver(Observer) , notifyObservers() .
  המימוש עבור Panel כלל את מימוש הפונקציה update(Observable,Object) .
- Strategy – בשאלה נתונות לנו 4 אופציות בהן ניתן לאפשר למשתמש בלוח המודעות לבחור את סדר עדכון צבעם של הלוחות. השתמשנו בדפוס זה בשביל לבצע אנקפסולציה לארבעת האלגוריתמים הנ"ל שדרכם אנו קובעים את סדר העדכון ומיד מעדכנים את הלוחות ואת צבעם. המימוש בו בחרנו זהה למימוש שהוצג בתרגול : ארבעת האלגוריתמים יורשים מהממשק שהגדרנו NotifiyObserversAlgorithm המגדיר פעולה אחת בודדת notify .
  האלגוריתמים מומשו בתור מחלקות נפרדות היורשות מממשק זה ולמחלקה ColorGenerator קיים שדה שמתעדכן בהתאם להחלטת המשתמש והוא מכיל את האלגוריתם המתאים.