Object Oriented
Programming and Design

Homework 4

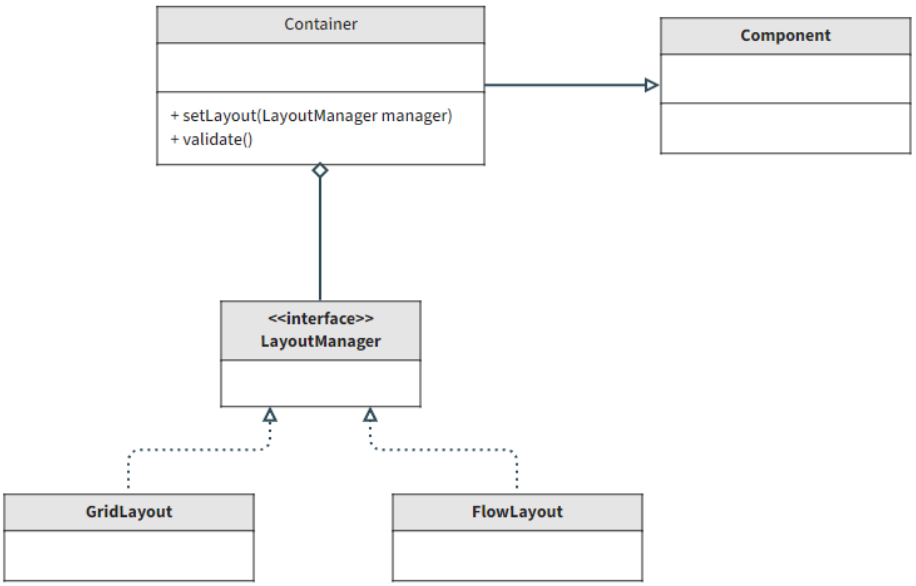
דניאל לוי 315668129

לידור אלפסי 307854430

שאלה 1

א. הוריאציה שממומשת כאן היא Strategy Design Pattern.
וריאציה זו באה לפתור את הצורך בהפרדה בין האובייקט לאלגוריתמים שונים שלו. האלגוריתם עצמו משמש כאובייקט של האובייקט הכללי, ומאפשר החלפה בין האלגוריתמים השונים בזמן ריצה.
בדוגמה הנתונה, ה-Container ישתמש בLayoutManager המתאים בהתאם למה שייקבע על ידי המתודה-setLayout.

ב. תרשים מחלקות של המחלקות והממשק הנתונים-



אנחנו רואים מחלקת Container שיורשת מ-Component, ו-LayoutManager שמשמש כ-interface שאותו מממשות המחלקות-GridLayout, FlowLayout.
שתי מחלקות אלה מממשות אלגוריתמים שונים שמשפיעים על אופן פעולת ה-Container. על ידי השימוש ב-Strategy Design Pattern, מנענו שכפול קוד שיצריך מימוש דומה של המחלקה עבור התנהגות שונה. ובנוסף לכך, על ידי מימושן בנפרד מה-Container, אפשרנו את היכולת לשנות את אלגוריתם ה-layout על המחלקה בזמן ריצה.
מהתרשים ניתן לראות ש-Container מנהל מופע של LayoutManager, ומאפשר שינוי של האלגוריתם בו הוא משתמש. זאת בעזרת המתודה-setLayout. בנוסף לכך מסופקת מתודה בשם validate, שעל פי הדוקומנטציה שלה ניתן להסיק שבין היתר, תפקידה לוודא את אופן עדכון ה-layout(זאת כנגזרת מכך שהוא subcomponent של המחלקה Container).

שאלה 2

על מנת לממש את המוטל עלינו בשאלה, נעזרנו ב-Observer design pattern. בפתרון שלנו השתמשנו בגרסת ה-push model. העדפנו להשתמש באפשרות זו משום שכך אנחנו שומרים את המערכת מעודכנת, ומודיעים ל-observers על מזג האוויר בכל רגע נתון. דרך זו נראתה לנו יותר רלוונטית בשביל מערכת לדיווחי מזג אוויר, שאמורה לעבוד בזמן אמת. כך לדוגמה, במצב תיאורטי בו שני אתרי חדשות מקימים observer עצמאי שמקבל נתוני מזג אוויר, אנחנו לא מחכים שהם ימלאו לנו את זמן הריצה בבקשות עדכון כדי להרוויח קדימות על הדיווח. אנחנו מאפשרים להם להמשיך לעבוד בשאר המטלות שלהם, מאפשרים לעצמנו להמשיך בפעילות התקינה של המערכת שלנו, ומבצעים תקשורת רק כאשר יש על מה לעדכן.

במימוש שלנו כתבנו 4 מחלקות שמהוות את הבסיס למערכת-

- **WeatherData – מדובר במחלקה אבסטרקטית אשר מייצגת את נתוני מזג האוויר כאובייקט, תוך ניהול ה-Observers אשר עוקבים אחר נתוני מזג האוויר הללו. על רישום והסרת ה-Observers ניתן לשלוט בעזרת מתודות ייעודיות של המחלקה.**
  בכדי לאפשר את מימוש ה-push model, כתבנו את מתודת ה-notifyObserver. כאשר בסוף השימוש במתודה setWeather, אשר אחראית על עדכון נתוני מזג האוויר באובייקט, המתודה notifyObserver נקראת ומודיעה לכל ה-Observers הרשומים על נתוני מזג האוויר החדשים.

- **LocationWeatherData – מדובר בתת-מחלקה מוחשית של מחלקת ה-WeatherData. מתודה זו מרחיבה את הפונקציונליות של WeatherData על ידי הוספת מחרוזת שתפקידה הוא ייצוג מיקום מסוים(המיקום שאת מזג האוויר בו אנחנו רוצים לייצג).**

- **WeatherObserver – מדובר בממשק אשר מגדיר את המתודה update. מתודה זו נדרשת משום שזו המתודה שנקראת על ידי WeatherData על מנת לעדכן את ה-Observers בעדכוני מזג האוויר. כל Observer שירצה לעקוב אחר נתוני מזג אוויר של אובייקט מסוים, חייב לעמוד בדרישה הזו למען העבודה התקינה של המערכת.**

- **WeatherDataFactory – מדובר במימוש של שיטת ה-Factory Method. הפרדנו את תהליך היצירה של אובייקטים מסוג LocationWeatherData וצאצאיה אל מתודה נפרדת, על מנת להסיר את הצימוד בין יצירת המופע, לבין ההחלטה על הטיפוס שלו.**
  בתרגיל הזה ובמערכת המינימלית שבחרנו, מטרה זו הייתה פחות רלוונטית, אך השיטה עזרה לנו לשלוט ברשימת האובייקטים שיצרנו. כך למשל, אם כבר קיים מופע של העיר תל אביב, אין סיבה ליצור עוד אחד כזה. ריבוי מופעים שכזה יגרום לחוסר סנכרון בין Observers שונים למרות שהם מסתכלים על אותה העיר. לכן שמרנו את כל האובייקטים שיצרנו בתוך ה-Factory, ובמידה ויצירת אובייקט עבור עיר מסויימת כבר קרה לפני, אז פשוט שלחנו את אותו מופע שקיים בחזרה למשתמש(במקום ליצור מופע חדש).

יצרנו דוגמת הרצה של הקורס, אשר ממחישה יצירה של 4 Observers שונים על 3 ערים שונות(תל אביב, חיפה וטבריה). מהפלט שמופיע ניתן להסיק כי נוצרו באמת רק 3 LocationWeatherData כמו שציפינו, וש- 2 Observers אשר מסתכלים על אותה עיר מתעדכנים ביחד עבור כל שינוי שמעודכן עבור מזג האוויר בעיר. כאן ניתן לראות את הבקשות ליצירת LocationWeatherData עבור 3 ערים שונות. מהמימוש שלנו, נצפה שיווצרו רק 3 אובייקטים שונים, כאשר המופע הראשון והאחרון אמורים להיות זהים.

```
System.out.println("Initializes the LocationWeatherData");
LocationWeatherData tiberias_data = WeatherDataFactory.createLocationWeatherData("Tiberias");
LocationWeatherData tel_aviv_data = WeatherDataFactory.createLocationWeatherData("Tel Aviv ya habibi Tel Aviv");
LocationWeatherData haifa_data = WeatherDataFactory.createLocationWeatherData("Haifa");
LocationWeatherData second_tiberias_data = WeatherDataFactory.createLocationWeatherData("Tiberias");
```

לאחר הגדרת ה-Observers בהתאם לממשק שהגדרנו, ביצענו סדרת בדיקות אשר אמורה להראות לנו את התנהגות העדכון של מזג האוויר עבור ה-Observers השונים-

```
System.out.println("\nUpdate 1- " + tiberias_data.getLocation());
tiberias_data.setWeather(28.5, 70.0, 50.0);
System.out.println("\nUpdate 2- " + tel_aviv_data.getLocation());
tel_aviv_data.setWeather(27.0, 65.0, 70.0);
System.out.println("\nUpdate 3- " + haifa_data.getLocation());
haifa_data.setWeather(23.5, 50.0, 100.0);
System.out.println("\nUpdate 4- " + second_tiberias_data.getLocation());
second_tiberias_data.setWeather(29.0, 72.0, 25.0);
```

```
Starts the test
Initializes the LocationWeatherData
Starts the test

Update 1- Tiberias
Got an update
Tiberias: Temperature - 28.5, Humidity - 70.0, Air pressuere - 50.0
Got an update
Tiberias again: Temperature - 28.5, Humidity - 70.0, Air pressuere - 50.0

Update 2- Tel Aviv ya habibi Tel Aviv
Got an update
Tel Aviv Temperature - 27.0, Humidity - 65.0, Air pressuere - 70.0

Update 3- Haifa
Got an update
Haifa: Temperature - 23.5, Humidity - 50.0, Air pressuere - 100.0

Update 4- Tiberias
Got an update
Tiberias: Temperature - 29.0, Humidity - 72.0, Air pressuere - 25.0
Got an update
Tiberias again: Temperature - 29.0, Humidity - 72.0, Air pressuere - 25.0
```

ואכן ניתן לראות שכאשר אנחנו מעדכנים את העיר טבריה, שני Observers מקבלים את העדכון. בפעם הראשונה זה קורה כאשר אנחנו מעדכנים את המופע של tiberias_data, ובפעם האחרונה זה קורה כאשר אנחנו מעדכנים את המופע של second_tiberias_data. שני המופעים הם בעצם אותו מופע שמעדכן את ה-Observers שנרשמו אליו, כפי שרצינו לכפות על ידי ה-Factory.

שאלה 3

במימוש תרגיל זה נעזרנו ב-Composite Design Pattern. שיטה זו עוזרת לנו להסתכל על האובייקטים כמעין עץ, שמבטא היררכיה אשר יורדת דרך הפעולות עד למספרי ה-Double וה-Integer בקצה. מטבע אופי הביטויים המתמטיים, זהו ה-design pattern שהיה נראה לנו הכי מתאים למימוש.
כך, אוסף הפעולות-Addition, Subtraction, Multiplication, Division, UnaryMinus שימשו כאובייקטים אשר ממומשים כהרכבה של אובייקטים אחרים מאותו סוג(אובייקטים מורכבים). ומאידך, המחלקות- IntegerValue, DoubleValue משמשות כאובייקטים פשוטים, כך שהם יהיו העלים בעץ ויוכלו להכניס את התוכן המספרי לביטויים. זאת על ידי שמירת הערך המספרי שהן מייצגות, לעומת אחסון אובייקטי Expression כפי שקורה במחלקות המורכבות.

במימוש שלנו כל האובייקטים, גם הפשוטים וגם המורכבים שציינו, מימשו ממשק של Expression. מתוך הביטוי המתמטי הכולל, ביצענו מעין רדוקציה לביטויים מתמטיים(Expressions) קטנים יותר. זה הקל עלינו ביכולת לממש ממשק אחיד, ללא מתודות מיותרות ועם אחידות באופן הפעלת כל מחלקה.

הממשק כלל שתי מתודות בלבד-

- **Eval- מתודה זו מחזירה לנו את הערך שה-Expression מבטא. בביטויים המורכבים מדובר בתוצאה של הפעולה המיוצגת, ובביטויים הפשוטים מדובר בערך מספרי שאחסנו בתוכו. במימוש עצמו, כאשר נקרא למתודה הזו של אחד מהביטויים המורכבים, אז מתודה זו תקרא למתודות המקבילות של ה-Expressions שהיא מאחסנת. קריאות אלו ממשיכות עד אשר נגיע לקצוות בהם יש ביטויים פשוטים-IntegerValue, DoubleValue, אשר מחזירות פשוט ערך מספרי ששמור בתוכן. לשם אחידות הממשק והכללה של כל הביטויים האפשריים, ערך זה הוא מסוג double.**
- **toString- המתודה מחזירה את הביטוי המיוצג ב-Expression כ-String לשם ויזואליזציה של הביטוי. גם מתודה זו קוראת מאחורי הקלעים למתודה המקבילה ב-Expressions ששמורים בתוכה, עד אשר מגיעים לעלים בעץ שמציגים ערך מספרי ידוע.**

דוגמת הרצה של 3 ביטויים שונים(כולל הדוגמה שפורסמה בגיליון)-

```
Expression staff_example = new Multiplication(
    new Addition(
        new DoubleValue(2.5),
        new DoubleValue(3.5)
    ),
    new UnaryMinus(
        new IntegerValue(5)
    )
);

Expression example2 =  new Subtraction(
    new Division(
        new DoubleValue(5),
        new DoubleValue(2)),
    new UnaryMinus(
        new Addition(
            new IntegerValue(5),
            new Multiplication(
                new IntegerValue(3),
                new DoubleValue(2.5)
            )
        )
    )
);
```

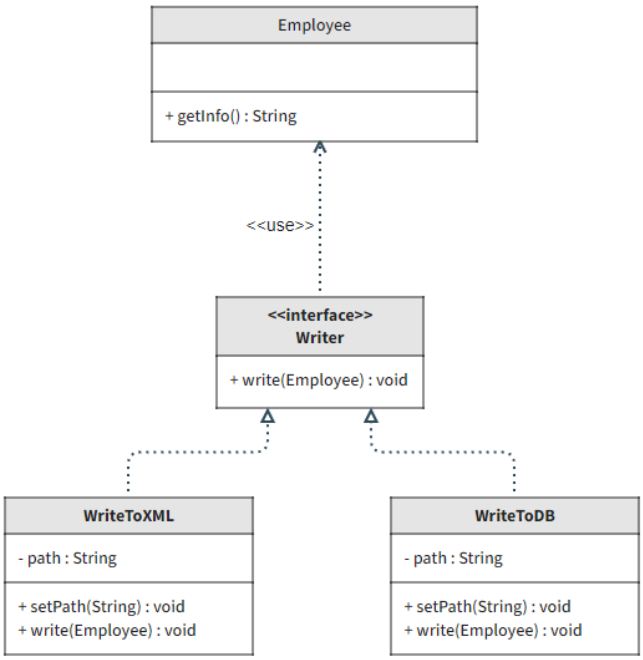
```
Expression example3 =  new Multiplication(
    new Division(
        new DoubleValue(5),
        new Addition(
            new IntegerValue(2),
            new DoubleValue(2)
        )
    ),
    new Addition(
        new Addition(
            new IntegerValue(5),
            new Multiplication(
                new IntegerValue(3),
                new DoubleValue(2.5)
            )
        ),
        new Multiplication(
            new Addition(
                new DoubleValue(2.5),
                new IntegerValue(1)
            ),
            new DoubleValue(1.25)
        )
    )
);
```

```
System.out.println(staff_example.toString() + " = " + staff_example.eval());
System.out.println(example2.toString() + " = " + example2.eval());
System.out.println(example3.toString() + " = " + example3.eval());
```

```
Starts the test
(((2.5) + (3.5)) * (-(5))) = -30.0
(((5.0) / (2.0)) - (-((5) + ((3) * (2.5))))) = 15.0
(((5.0) / ((2) + (2.0))) * (((5) + ((3) * (2.5))) + (((2.5) + (1)) * (1.25)))) = 21.09375
```

שאלה 4

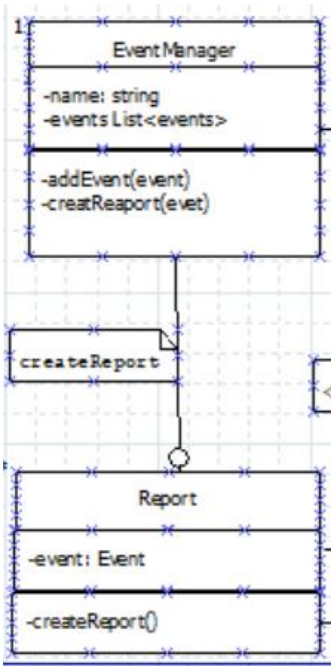
א. עקרון ה-SOLID שמופר כאן הוא עקרון הפתיחות-סגירות. הסטודנט המדובר הוסיף למחלקת Employee את המתודות-toDB, toXML, אבל אם נרצה בעתיד להוסיף תמיכה לפורמט כתיבה אחר אז ניאלץ להוסיף קוד למחלקה הקיימת. נרצה להימנע מכך על ידי הפרדת הפעולה מהמחלקה עצמה.
ניצור ממשק חדש אשר ייקרא Writer. לממשק נגדיר מתודה בשם writeEmployee, אשר מקבלת מופע של Employee כפרמטר. כמובן שנצטרך תמיכה של מתודה מתוך ה-Employee שתיתן לנו גישה לנתונים הנדרשים מאיתנו.
כעת עבור כל פורמט שאליו נרצה לכתוב, נוכל להגדיר מחלקה שמממשת ממשק זה עם הפונקציונליות הנדרשת עבור הפורמט הספציפי. במקום לערוך קוד קיים, נוכל פשוט לכתוב מחלקה נוספת ולקיים את עקרון הפתיחות-סגירות.



ב. עקרון האחריות היחידה עוזר לנו רבות לפשטות ב-Chain of Responsibility design pattern.
ב-design pattern שכזה אנחנו מגדירים שרשרת של אובייקטים אשר יודעים לבצע פעולות כלשהן.
כל אובייקט שכזה לאורך הדרך יודע לטפל בבקשה שהועברה אליו, או להעבירה הלאה בשרשרת.
אנחנו מממשים זאת עם אובייקטים אשר יש להם אחריות יחידה, פעולה יחידה שהם אמורים לבצע, ובכך אנחנו שומרים על העיקרון המדובר תוך מימוש ה-design pattern הנתון.
השיתוף של שני המושגים מעניק לנו גמישות ביצירת ובתחזוקת הקוד, תוך מתן אפשרות רחבה יותר לזיהוי תקלות לאורך הדרך ולהרחבת הפונקציונליות מבלי לפגוע בעקרונות אחרים כמו פתיחות-סגירות.

ג. עקרון ה-Creator מגדיר מי יוצר מופעים של האובייקט X. בתרגיל 3 ניתן לקחת את אובייקט ה-Report, ועל ידי השאלה-למי יש מספיק מידע לאתחול המופעים של Report, הגענו למסקנה שה-Creator המתאים יהיה EventManager.

ניתן לראות בתרשים את המתודה createReport בה השתמשנו. בתור המחלקה שמנהלת את כל ה-Events ברשימה, יצירת ה-Reports דרך ה-EventManager הייתה מתבקשת.



עקרון ה-Information Expert מגדיר את העקרון לחלוקת תחומי האחריות. למי יש את המידע הנחוץ למילוי האחריות. בתרגיל 3 אפשר להסתכל על ה-Event כמומחה מידע על ניהול הכרטיסים.

כרטיסים נמכרים לפי כל Event בנפרד, ולכן היה מבחינתנו טבעי ליצור את התכן כך שהאחריות על ניהול הכרטיסים תתבצע בכל Event.

לכן ה-Event הוא זה שמחזיק בכל הנתונים אודות הכרטיסים. בין היתר, הוא מנהל רשימת Tickets, סופר את רשימת הכרטיסים שנמכרו ואת רשימת כרטיסי ה-VIP.