**שאלה 1:**

1. תארו את תהליך הניתוח שביצעתם למרכיב: ניהול ביקור לא מתוכנן (מזדמן).
קוים מנחים לתשובה: פרטו מה הם השאלות/הפרטים שהתייחסתם אליהם בתהליך הניתוח והחשיבה, וכן התייחסו לקשרים ולמעברים 1. ממודל Use-case למודל תהליכי המיוצג בעזרת Activity Diagram, 2. ממודל התהליכים לקראת מימושו בתוכנה.

**תשובה:**

תהליך ניתוח מרכיב הביקור הלא מתוכנן התחלק לשלושה חלקים מרכזיים:

- חלק ה-"מי?".
- חלקה-"מה?"
- חלק ה-"איך".

חלק ה-"מי?":

בחלק זה ניסינו להבין **מי** לוקח חלק במרכיב הביקור הלא מתוכנן, ומי אחראי על מרכיב זה.

לפי ה- מודל UseCase שלנו ראינו כי גם המטייל וגם העובד כניסה לוקחים חלק ברכיב זה:

- המטייל מגיע לפארק ומבקש להיכנס.
- העובד בודק שיש מקום בפארק ובמידה וכן מעדכן את הביקור במערכת ומכניס את המטייל.

לאחר שהבנו מי לוקח חלק במרכיב זה, השאלה שנותרה היא מה צריך לממש במרכיב זה.

חלק ה"מה?":

**חלק זה הוא המעבר ממודל ה-UseCase למודל ה-Activity.**

בשלב זה ניסינו להבין **מה** צריך להתרחש במרכיב הביקור הלא מתוכנן.

ניסנו לשאוב רעיונות ממקומות שכבר ממשיים "ביקור לא מתוכנן", לדוגמה, לונה פארקים, בתי קולנוע ועוד.

מה שצריך להתרחש:

1) מבקר ללא הזמנה מגיע לפארק ומבקש להיכנס.
2) עובד הכניסה בודק במערכת האם הפארק מלא
   a) אם הפארק מלא, העובד מודיע למטייל כי אין מקום ולא מכניס אותו
   b) אם הפארק לא מלא, העובד מבקש מהמטייל פרטים.
      i) העובד מעדכן במערכת את הביקור ומכניס את המבקר.

חלק ה-"איך?":

**שלב זה מתאר את המעבר משלב המודל תהליכים למימוש בפועל** – **איך** נממש זאת בתוכנה שלנו.

בשלב זה ניסינו לחשוב איך כל מה נממש את כל מה שתואר בחלק ה"מה?".

השתדלנו לפרק זאת לתהליכים "אטומיים" ככל האפשר:

- כדי לבדוק האם הפארק מלא – עלינו לשלוף מהמערכת את מספר המבקרים הנוכחי ומספר המבקרים המקסימלי:
  - העובד פארק ילחץ על כפתור לראות את כמות האנשים בפארק.
  - הקליינט ישלח בקשה לסרבר לקבל את המידע הזה.
  - בעזרת שאילתות מה- database נשלוף את המידע.
  - הסרבר יחזיר את המידע לקליינט.
  - הקליינט יעדכן זאת במסך של העובד פארק.

- אם הפארק לא מלא על המערכת לקחת את נתוני המטייל שהוזנו לה ולהעביר לסרבר שיעדכן את הביקור ב-database:
  - העובד פארק ילחץ על כפתור הוספת ביקור (לאחר שהזין את נתוני המטייל)
  - המערכת תבדוק שלא חסרים פרטי מידע
  - הקליינט ישלח בקשה לסרבר להוסיף את הביקור ל-database.
  - בעזרת שאילתת הכנסה הסרבר יוסיף את המידע ל-database.
  - הקליינט יעדכן במסך של העובד שהפעולה הצליחה.

## שאלה 2:

2. בהרצאה הוגדרה ***Reusability*** כתכונה של תוצר של תהליך הפיתוח אשר משקפת את היכולת לבצע reuse בהקשר לתוצר זה. בהתאם להגדרה זו, תארו יישום של 3 התכונות המאפשרות לכם לשלב במערכת GoNature שאתם מפתחים קטעי קוד ומרכיבים אחרים שלא אתם כתבתם או תכננתם.
תארו בדיוק (ובהתייחסות ספציפית) ובפירוט את התכונות המאפשרות Reuse של אותם מרכיבים אשר בחרתם לשלב במערכת שלכם, תוך התייחסות בדוגמאות ספציפיות (לא 'עקרוניות' או 'כלליות') לדרישות הפונקציונליות של המערכת שתכננתם (התייחסות ספציפית בהקשר זה = התייחסות למרכיבים ספציפיים מתוך התיאור המילולי הראשוני של פעולת המערכת ששאתם מפתחים מהתחלת הסמסטר. לא כולל תהליך זיהוי משתמש).
אם יש מי מ-3 התכונות הנ"ל אשר לא באה לידי ביטוי ב-reuse שביצעתם - הסבירו את הסיבה לכך.

## תשובה:

בהרצאה הוגדרו שלוש תכונות של Reusability:

- Availability
- Understandability
- Flexibility

נתאר כיצד תכונות אלו באות לידי ביטוי בפרויקט שלנו:

**Availability:**

בפרויקט שלנו אנו עושים שימוש בשלושה כלים מרכזיים שקיימים כבר ומוכחים שעובדים באופן תקין:

- **OCSF** – framework המממש לנו את ארכיטקטורת לקוח – שרת.
- **JDBC** – Api שמנהל לנו את ההתעסקות עם ה-database (חיבור, שאילתות, שגיאות ועוד)
- **JavaFx** – ספריות קיימות שבעזרתן אנו בונים את ה-Gui של המערכת שלנו.

שימוש בכלים אלו "פותר" אותנו מלכתוב מאפס את הפונקציונליות שכלים אלו מעניקים לנו ולכן חוסך לנו בהרבה זמן. בנוסף אלו כלים שנמצאים בשוק זמן רב, שמתוחזקים, עם כמות גדולה של משתמשים ולכן גם אמינים.
אם היינו מממשים לבד את ה-כל ייתכן שהיינו נתקלים בבאגים רבים, או שאפילו לא היינו עולים על מרבית הבאגים.

**Understandability**:

כמו שנאמר לעיל, אלו כלים שמתוחזקים ושכתובים לפי כללי ה-best practice ולכן הקוד קריא וקל להבנה.

בנוסף, לכל הכלים האלו יש דוקומנטציה רחבה שניתן זניתן להיעזר בה:

**JavaFx**: https://docs.oracle.com/javafx/2/

**JDBC**: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

**OCSF**: http://www.site.uottawa.ca/school/research/lloseng/supportMaterial/source/

**Flexibility**:

גם JavaFx וגם OCSF כתובים בצורה כזאת שאנו יכולים לבצע extends למחלקות או לבצע Override לשיטות.

כלומר אנו יכולים להשתמש בקוד הקיים, אך גם לשנות אותו במידה ואנו צריכים.

**דוגמאות ספציפיות:**

- כל פעם שסוגרים את המערכת (סוגרים קליינט), מופעלת השיטה quit (תכונת ה-Availability ) שמטרתה לטפל במה קורה כאשר הקליינט נסגר. לנו יש את האפשרות לכתוב אותה מחדש (תכונת ה-Flexibility) כדי שתעשה מה שאנו צריכים.
כנ"ל לגבי הפעלה וסגירה של הסרבר עם השיטות serverStarted ו-ServerStopped.

- לאורך כל השימוש במערכת שלנו, הקליינט מדבר עם הסרבר והסרבר עונה לקליינט.
לדוגמה, כאשר אנו מוסיפים הזמנה חדשה למערכת הקליינט מבקש מהסרבר להכניס את ההזמנה החדשה ל-database.
כלומר בעזרת השיטה handleMessageFromClient שמספק לנו ה-OCSF, הסרבר מזהה כי יש פנייה חדשה. (תכונת ה-Availability )
בנוסף נשים לב כי handleMessageFromClient זוהי שיטה אבסטרקטית, כלומר אין לה מימוש ואנו חייבים לממש אותה כרצוננו. (תכונת ה-Flexibility).
אז הסרבר מזהה שיש לו בקשה מקליינט, במקרה זה הזמנה חדשה, הוא מפעיל את השיטה המתאימה שמוסיפה הזמנה חדשה ל-database.
לאחר ההוספה, הסרבר מחזיר תשובה לקליינט, במקרה זה שההוספה הצליחה.
ההודעה מגיעה לשיטה handleMessageFromServer שנמצאת בקליינט (תכונת ה-Availability ) וגם אותה אנו צריכים לכתוב כרצוננו שתתאים למערכת שלנו. (תכונת ה-Flexibility).

**שאלה 3:**

3. א. הערכה כללית:
1. מהם היתרונות של מודל UML כעזר לתהליך התכנון?
(i) הסבירו איך מתקבלים (מתממשים) היתרונות שציינתם.
(ii) ציינו דוגמה אחת קונקרטית ממוקדת (לא כללית ולא Login) מתוך תהליך הניתוח והתכן שאתם בצעתם לשימוש מועיל ב-UML תוך תיאור והתייחסות ספציפית למרכיבים של מערכת "GoNature" שתכננתם ומידלתם.
2. ציינו קשיים הנובעים מחסרונות של UML שנתקלתם בהם. גם כאן התייחסו ספציפית לתהליך שבצעתם לפיתוח מערכת זו.

ב. ניתוח ודיון:
בהתאם לניסיון שרכשתם במהלך העבודה על מטלה זו, תארו אפשרויות לשינויים ושיפורים במתודולוגית UML אשר נותנים מענה לחסרונות שנתקלתם בהם במהלך ה-**design** שביצעתם בפרויקט שלכם. הסבירו את תשובתכם תוך תיאור דוגמה ספציפית (כולל שמות של רכיבים, לא כולל Login) מתוך עבודתכם.

**תשובה:**

א.

היתרונות של מודל ה-UML לתהליך התכנון:

- מעניק תצוגה ויזואלית:
UML מציג בצורה ויזואלית את הפונקציונליות של המערכת (UC), מציג את הקשרים שבין המחלקות לישויות במערכת שלנו (Class), מציג בצורה דינאמית מה קורה בכל פונקציה של המערכת (Activity) ומציג איך נממש את הפונקציונליות במערכת (Sequence).
ע"י הצגת המידע בצורה ויזואלית, יותר קל למתכנת שבא לכתוב את המערכת להבין ולדמיין איך המערכת צריכה להיות.

- קל להבנה:
UML אינה כתובה בשפת תכנות מסוימת, והיא קלה להבנה לכל סוג של מתכנת.

- מאפשרת זיהוי של REUSE:
מתכנת שמסתכל על ה-UML יכול לזהות קטעים שחוזרים על עצמם מספר פעמים במקומות שונים במערכת או לזהות שיש כבר קוד מוכן שמישהו כתב שמבצע בדיוק את מה שצריך.

- מעבר קל בין התכנון לכתיבת הקוד:
בעזרת Activity ו –Sequence , המתכנת מבין איך הפונקציונליות צריכה להתבצע ואיך לכתוב את הקוד בהתאם.

i) היתרונות שציינו באים לידי ביטוי בכך בעזרת ה- UML של הפרויקט שלנו ניתן להבין ביתר קלות מה הפונקציונליות של המערכת שלנו, אילו מחלקות יהיו במערכת, ואיך נממש את הפונקציונליות של המערכת.
בנוסף בעזרת התרשימים של ה-Activity וה-Sequence שעשינו, יש לנו מושג איך לממש את הפונקציונליות בקוד עצמו.

ii) דוגמה לשימוש מועיל של ה-UML הוא בניתוח תהליך הזמנת ביקור בפארק.
בעזרת ה-UML אנו למדנו אילו ישויות לוקחות חלק בתהליך זה.
ב- Activity תארנו הפעולות שצריכות להתבצע ואת הסדר שלהם בתהליך זה: הכנסת פרטים של מטייל, בדיקה האם התאריך פנוי, חישוב מחיר בהתאם לסוג מנוי, הכנסה ל-database, שליחת הודעה במייל/בפלאפון, ותזכורת במייל ובפלאפון.
ב-Sequence תארנו איך אנו מבצעים את הפעולות שהגדרנו ששייכות להזמנת ביקור.
כעת כשנרצה לבוא לכתוב בפועל את הקוד שיממש את הפונקציונליות של הזמנת ביקור נוכל להיעזר ב-UML ולהבין בדיוק מה אנחנו צריכים לכתוב.

.2

החסרונות שנתקלו בהם בעת השימוש ב-UML:

- יצירת התרשימים זו משימה לא פשוטה שלוקחת זמן רב. אין בעיה כאשר מדובר בחברות גדולות שיש אנשים שזה תפקידם העיקרי אך אצלנו בפרויקט, בתור מתכנתים, יצירת ה-UML לוקח זמן רב וזהו זמן שבו היינו יכולים בפועל לכתוב בפועל את המערכת.
כל הוספה או שינוי של פונקציונליות למערכת מצריכה שינוי של מספר דיאגרמות – הרבה זמן שמושקע בדיאגרמות.

- תרשים ה-UML יכול להגיע למצב שהוא מכיל מלא מידע ודיאגרמות עד כדי כך שקשה לעקוב אחרי התרשים.
אצלנו בפרויקט, ה-class diagram הגיע למצב שמבט ראשון בו מכניס אותך ל"פחד" ובלבול בגלל הכמות הרבה של המידע שנמצא שם.

- יש דברים שקשה ל-מדל ולכן ה-UML לא מעניק לנו מספיק מידע עבור מקרים אלו.
אצלנו בפרויקט, אין מידול מספק (יש בסיסי) למערכות החיצוניות: card reader ומערכת המידע של העובדים.
בנוסף פעולות שקורות אוטומטית ע"י המערכת יותר קשה להביא לידיי ביטוי ב-UML.

ב. הצעות לשיפורים UML:

- אחד החסרונות שהצגנו הוא ש-UML יכול להגיע למצב שהוא עמוס בגלל כל הרכיבים הגרפיים ולכן אפשר לקבל "שוק" כאשר מביטים בדיאגרמה עמוסה.
אפשרות לשיפור, היא אולי הוספה של "תרגום מילולי" לדיאגרמה. כלומר שניתן בלחיצת כפתור להציג את הדיאגרמה כתיאור מילולי במילים ואז יהיה ניתן להביט בצורה גרפית או מילולית.

- שינוי אפשרי במתודולוגיית UML היא לקחת את ה-UML בגישה יותר בסיסית וכקו מנחה.
כלומר לא להשקיע כל כך הרבה זמן ביצירת ה-UML ובפירוט רב, אלא לייצר את אותם תרשימים אך בצורה בסיסית יותר תוך כדי הסבר מילולי.
זה יביא לכך שנבזבז פחות זמן ביצירת ה-UML ונוכל להשקיע את הזמן הזה בכתיבת קוד בפועל.

- בהמשך לכך שיצירת ה- UML זו משימה שלוקחת זמן רב, לדעתנו ניתן להוסיף אפשרויות כדי להפחית את זמן היצירה של ה-UML.
לדוגמה, ב-class diagram יצירה של templates כלליים שמשתמשים בהם הרבה ( לדוגמה, template עבור person ) ואז יצירה מהירה של class עבור person.
והוספת אפשרות ליצירת templates מותאמים אישית.

- בנוסף אחד החסרונות שהצגנו ב- UseCase (במטלה הראשונה) היא שאי אפשר לדעת את רצף האירועים, ומה צריך להתרחש לפני מה ע"י הסתכלות רק על הדיאגרמה.
אז אולי אפשר להוסיף רכיב גרפי שמראה אם יש תלות במשהו אחר יש פעולה אחרת ש"מפעילה" את הפעולה הזאת.
לדוגמה אצלנו בתרשים UseCase לא ידוע מתי נשלחות הודעות ה-email/sms אז פתרון לכך זה הוספת ייצוג גרפי שמראה שההודעות מתרחשות בעת אישור הזמנה/תזכורת יום לפני הביקור/ביטול ביקור וכו'.