סיכומון למבחן שיטות בהנדסת תוכנה

דרישות פונקציונליות(Functional Requirement)

מתארות את הפעולות השונות שבהן המערכת תומכת מנקודת מבט חיצונית של המשתמש(מה הן מצפות לקבל, מה הן מחזירות וכיצד הן משנות את מצב המערכת).

דרישות לא פונקציונליות(Non-Functional Requirement)

מתארות תכונות כגון מהירות ביצוע, כמות נתונים מרבית, אמינות, תמיכה בסטנדרטיים פלטפורמות חומרה וכו'.

1.הבהרת דרישות(Requirements Elicitation)

המטרה של פעילות זו היא להגדיר את מטרת המערכת, זוהי פעילות משותפת ללקוחות ולמפתחים.

ביחד הם מזהים את תחום הבעיה ומגדירים מערכת אשר תפתור את הבעיה.

זיהוי השחקנים

המטרה היא לברר מיהם סוגי המשתמשים במערכת? הסיבה לזיהוי השחקנים היא לעצור ולחשוב מי צריך את המערכת בכדי לא ללכת רחוק מדי בפרטים הטכניים.

זיהוי תסריטים

ממחישים כיצד המערכת פותרת את הבעיה למשתמש.

פיתוח תרחישים קונקרטיים המתארים כיצד ישתמשו במערכת וכיצד המערכת תשיג את מטרות השחקנים.

זיהוי use-cases

על בסיס התרחישים הקונקרטיים יוצרים המפתחים את קבוצת הU.C שמתארת באופן מלא את התנהגות המערכת ואת גבולות המערכת.

ה-use cases לא מתייחסים כלל לממשק המשתמש או לביצועים, ולכן יש לתת ייחוס לדברים אלו בנפרד.

זיהוי דרישות לא פונקציונליות

בתת פעילות זו מזהים את מגבלות המערכת, שלא קשורות באופן ישיר לאפיון השירותים שהמערכת מספקת יעילות, פלטפורמות(לדוגמא סביבת הפיתוח, מחשבים שעליהם נרוץ וכו'. נדבר על כך בהמשך), בטיחות, אמינות, וכו'.

Glossary

מילון מונחי שבו נתעד את הישויות שבהן המערכת מטפלת ואת משמעותן.

2.פעילות הניתוח המערכת(System analysis)

זוהי הפעילות הבאה אחרי פעילות הבהרת הדרישות. פעילות זו, מקבלת כקלט את מסמך אפיון הדרישות ומייצרת כפלט את מודל המערכת(analysis system model).

מודל המערכת מכיל את החלקים הבאים:

Analysis object model(static model)

מודל זה מתאר את המושגים שבהם המערכת מטפלת, התכונות שלהם והקשרים ביניהם. המודל מתואר ע"י Class Diagram. זהו ייצוג ויזואלי ופורמלי של מילון המושגים(Glossary) הנראים למשתמש.

Dynamic model

המודל הדינמי מתמקד בתיאור התנהגות המערכת, המודל הדינמי מתואר ע"י דיאגרמות מצבים (State Diagram) וע"י Sequence Diagrams.

דיאגרמות מצבים מתארות את ההתנהגות המלאה של אובייקט יחד. לעומת זאת Sequence Diagrams, מתארות את התקשורת בין קבוצה של אובייקטים המשתתפת במהלך Use Case ספציפי. המודל הדינמי עוזר לנו להגדיר תפקידים למחלקות ובמהלך הבנייה שלו הוא עוזר לזהות מחלקות חדשות.

נחלק את מודל העצמים לשלושה סוגים:

Entity objects – מייצגים את המידע אשר המערכת מנהלת ואשר אחריו היא עוקבת.

Boundary objects – מייצגים את התקשורת והממשק בין המערכת לשחקנים.

Control objects – אחראים על ביצוע ה use case.

פעילות מס' 1: זיהוי וסיווג אובייקטים

זיהוי אובייקטי Entity.

מכיוון שאפיון המערכת מתואר ע"י טקסט(Use Cases), נוכל לנצל את המבנה הדקדוקי(פירוק לפעלים/שמות עצם) של הטקסט כדי לקבל רשימה התחלתית של מועמדים שאותם נסנן אחר כך כדי לקבל את ישויות המערכת. שיטה זו מיוחסת לחוקר ששמו Abbott ולכן נקרא לה "השיטה של **Abbott**".השיטה מתבסמת על הטבלה הבאה:

Part of Speech	Model Component	Example
Proper noun	Instance	**Alice**
Common noun	Class	**Car, Report**
Doing verb	Operation	**Submits, Creates, Selects**
Being verb	Inheritance	**Is a kind of, is one of**
Having verb	Aggregation	**Has, consists of..., includes..**
Modal verb	Constraints	**Must be,**
Adjective	Attribute	**Incident time, Wine label, Car license plate**

בשלב הראשון של פעילות הזיהוי נייצר רשימה שבה כל אובייקט מתואר באופן "חופשי" בעזרת שם ופסקה קצרה. הכוללת את התכונות ותחומי האחריות שלהם.

זיהוי אובייקטי Boundary.

אובייקטים אלה מתארים את ממשק המערכת. הם אוספים מידע מהשחקנים ומתרגמים אותו לטובת אובייקטי ה- Entity וה-Control. ולא יותר(הדבר מאוד חשוב, שכן אנו רוצים שאובייקטי ה-Boundary שלנו יהיו כמה שיותר חופשיים מהתחום הספציפי שבו אנחנו מנסים לבנות את המערכת, אנו נרצה שהם יהיו מין רכיבים של קלט/פלט אשר נוכל לעשות בהם שימוש כמעט בכל מערכת).

היוריסטיקות לזיהוי אובייקטי Boundary

זהה בקרי ממשק אשר נדרשים להתחלת ה-Use Case(לדוגמא, כפתור לכיוון השעון/בחירה מתפריט/טפסים וכו').

זהה טפסים שעל המשתמש למלא כדי להזין מידע למערכת.

זהה הודעות שהמערכת מייצרת כדי להגיב למשתמש.

כאשר ה-U.C מערב כמה שחקנים, זהה נקודות קצה(terminals) עבור השחקנים השונים.

זיהוי אובייקטי Control

אובייקטי Control, משמשים לתיאום הפעולות של אובייקטי Entity ו-boundary.

לצורך ביצוע Use-Case. לרוב אובייקט Control נוצר בתחילת ה-Use Case, ונעלם בסיומו. תפקיד האובייקט – לאסוף את המידע מאובייקטי ה-Boundary ולפזרו בין אובייקטי ה-Entity.

כאשר יש Use Case של מספר שחקנים, אנו ניתן אובייקטי Control עבור כל שחקן.

היתרון בשיטה הזאת: נובע מהיציבות של האובייקטים, אובייקטי Entity – הם כמעט ולא משתנים באותו עולם הבעיה(הם נשארים יציבים ללא קשר לפרטים המדויקים של המערכות). לדוגמא במערכות חניה, ה-entity "מקום חנייה" יהיה מקום חניה בכל עולם בעיה, אבל ההגדרות של ה-אובייקט Control שאחראי על איפה מותר לחנות הוא ספציפי לעולם הבעיה הנ"ל.

פעילות מס' 2: מיפוי U.C`s לאובייקטים בעזרת sequence diagrams

דיאגרמת sequence קושרת בין אובייקטים ל-U.C הדיאגרמה מראה כיצד ההתנהגות של U.C יחיד, מחולקת בין האובייקטים שמשתתפים בו.
קו החיים של האובייקטים אשר קיימים בתחילת ה-u.c מתחיל מראש הדיאגרמה.
קו החיים של אובייקטים שנוצרים במהלך התסריט (בעזרת «create») מתחיל בנקודה שבה הם נוצרו.
קו החיים מסתיים בהודעת «destroy»
העמודה השמאלית ביותר מתארת את השחקן שיזם את ה-u.c
העמודה השנייה תתאר את אובייקט הboundary אשר אתו השחקן מתקשר.
העמודה השלישית תתאר על פי רוב את אובייקטי אשר אחרי על תזמון ה-u.c.
אובייקטי control נוצרים ע"י אובייקטי boundary.
אובייקטי boundary נוצרים ע"י אובייקטי control.
אובייקטי control ניגשים לאובייקטי entity אך לעולם לא להיפך.
הערה חשובה!
לא תהייה גישה של אובייקט Entity לאובייקט Control לעולם, כי אז נאבד את המודולריות, כלומר, אם נרצה לפתח את אותה מערכת תחת ממשק אחר לדוגמא, ונרצה לקחת את ה-Entities מעולם הבעיה הקיים(כמו שהם), הם יגררו איתם Controllers שהם תלויים בהם, ובכך בעצם לא נוכל להפרידם.
לעומת זאת, Control יכול לגשת ל-Entity מכיוון שה-Entity נשאר ללא שינוי בעולם הבעיה.

פעילות מס' 3: זיהוי קשרים(associations)

במקרים רבים המידע שהמערכת שומרת ועוקבת אחריו נמצא בקשרים.
תיאור הקשרים והמגבלות שלהם עוזר להכתיב מגבלות על התנהגות המערכת, פעולות המערכת, חייבות לשמור על מבנה הקשרים כפי שהוא מתואר במודל האובייקטים.
לדוגמא: מערכת של מכירת כרטיסים לעולם קולנוע, דורשת לתאר קשר בין מי שהולך לצפות בסרט לבין המקום בו הוא יושב, והקשר הנ"ל יהיה בעל יחסי של 0:1, והדבר מציב מגבלה על פעולת המכירה(קרי, לא למכור את אותו מקום ל-2 אנשים שונים).
כדי לזהות את הקשרים נעזר ביוריסטיקה של Abbott, בכך שנבחן ביטויי פועל ופעלים המזהים מצב:
Has, Is part of, manages, reports to, is contained in, includes

פעילות מס' 4: זיהוי תכונות(attributes)

ההבדל בין תכונות לקשרים: תכונות הינם יחסים בין אובייקטים לערכים, לעומת קשרים שהינם ייצוג של יחס בין מספר אובייקטים. נקודה חשובה היא, שאם הגדרנו משהו כתכונה, המשהו הזה לא יכול להיות אובייקט. תכונה היא יחס בין אובייקטים לערכים, לדוגמא, השם של עובד, מס' לוחית הזיהוי של הרכב, הקואורדינטות של בית קפה במערכת לאיתור מסעדות וכו'.
מכיוון שתכונות הן החלק הכי פחות יציב באובייקט לא נתמקד בתיאור מלא של התכונות אלא נחפש רק את התכונות שמהותיות אל המודל, בראש ובראשונה תכונות אשר משמשות לזיהוי האובייקטים. נוכל להיעזר בהיוריסטיקה של Abbott כדי לזהות תכונות ע"י כך שנחפש ב-U.C ביטויי תואר

פעילות מס' 5: מידול התנהגות תלוית מצב של אובייקטים

כאשר אובייקט מופיע בכמה Seq. diagrams, עלינו לוודא שההתנהגות שלו עקבית. דרך אפקטיבית לעשות זאת, היא לבנות **State diagram** עבור האובייקט ולוודא שאפשר "להריץ" את ה-Seq. diags על דיאגרמת המצבים.
בנוסף, מכיוון שדיאגרמת מצבים **מתארת את כל ההתנהגות של אובייקט יחיד**, הדיאגרמה מהווה מקור השראה לתסריטים שעדיין לא חשבנו עליהם.

תכון מערכת(תכנה) – Software system design

פעילות התכון מקבלת כקלט את מודל המערכת. מטרתה למפות את האובייקטים שזוהו בשלב הקודם לתתי מערכות, ולמפות את תתי המערכות לרכיבי חומרה(Nodes).

יעדי התכון (Design goals):
מתארים את האיכויות שעל המפתחים למטב(Optimize).
אנו מקבלים בעצם מהלקוח את עולם הבעיה בשפתו, כלומר אם הוא אומר שהוא רוצה מערכת זולה, אמינה וזמינה, הוא לא מודע לכך שישנם דברים סותרים ושיש צורך לתעדף אותם בכדי שהדבר יהיה ניתן ליישום.

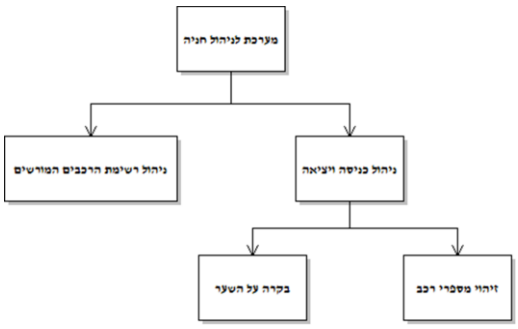
ארכיטקטורת התכנה(Software architecture):
מתארת את החלוקה של המערכת לתתי מערכות, ואת התלויות בין תתי המערכות ואת המיפוי של תתי המערכות לרכיבי החומרה.
בקרת גישה(access control) – לדוגמא, במרפאה האחיות לא יכולות "לראות" את היסטוריית הטיפולים של הלקוח, ואילו הרופא יכול(גם כן בצורה מוגבלת לחולים שלו בלבד).
זרימת הבקרה הגלובלית(global control flow) – מדבר על כיצד המערכת רצה, מה רץ באיזה זמן, לדוגמא: מערכות Thread של צ'אטים, יצטרכו להתחשב בעניין של האם לפתוח Thread עבור כל Client, או שהדבר ייעשה באותו ה-Thread וכו'.

Use Cases – עבור פעולות טכניות כמו: קונפיגורציה, אתחול, כיבוי ומקרים חריגים.

תתי הפעילויות של תהליך התכון
זיהוי יעדי התכון: המפתחים מזהים ומתעדפים את איכויות המערכת שיש למטב(Optimize).
פירוק ראשוני לתתי מערכות: המפתחים מחלקים את מהערכת לחלקים קטנים יותר בהתבסס על הu.c ועל מודל הניתוח. המפתחים משתמשים בפתרונות מקובלים הנקראים "סגנונות ארכיטקטוניים" כנקודת התחלה לפעילות זו. עידון הפירוק כדי להשיג את יעדי התכון לרוב הפירוק הראשוני לא עונה על יעדי התכון. בפעילות זו מעדנים את הפירוק עד אשר כל יעדי התכון מושגים.

דרישות המערכת מתארות בעיה מורכבת אשר איננו יכולים לפתור מיד. ולכן נפרק את הדרישות לתתי בעיות. כל תת בעיה תפורק לתתי בעיות עד שנגיע לתתי בעיות שאנחנו יודעים לפתור. הפתרון של כל תת בעיה הוא תת ממערכת. כלומר נקבל שני עצים מקבילים: עץ הבעיות(הדרישות) שמחלק את דרישות המערכת לדרישות מתתי המערכות, ועץ המערכת שמתאר את תתי המערכות שעונות על הדרישות.
ניתן להסתכל על הדוגמא הבאה:

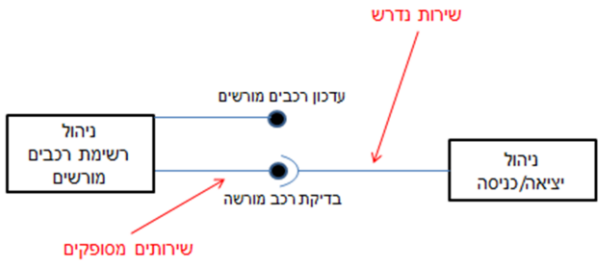
פירוק דרישות המערכת לניהול חניה



למשל זיהוי מספרי רכב, יכולה להיות בעיה פשוטה שכן קיימת "ספרייה" שתספק לנו מידע זה, מצד שני, יכול להיות שהיא בעיה יחסית מסובכת ולכן נפרק גם אותה לתתי בעיות.

שירותים וממשקים של תתי מערכות

תת מערכת מספקת שירותים(Services) לתתי מערכות אחרת. שירות הוא קבוצה של פעולות קשורות ע"י מטרה משותפת. לדוגמא מערכת תקשורת, שיכולה לספק שירותים של העברת הודעות. למשל המשתמש נרשם לשירות, ניתן לשלוח/לקבל הודעות, לבטל את השירות. כלל הפעולות האלו מגדירות את השירות של העברת ההודעות. וקיים מצב בו תת מערכת נותנת כמה שירותים שונים(לאו דווקא אחד). מערכות מספקות שירותים(Services) לתתי מערכות אחרות. שירות הוא קבוצה של פעילות קשורות ע"י מטרה משותפת.



:

צימוד(Coupling)

צימוד מודד את מידת התלויות בין שתי תתי מערכות. דרגת הצימוד בין שתי תתי מערכות A ו-B, מתאימה לסיכוי שיהיה צורך לשנות את תת מערכת A, בהינתן שתת מערכת B השתנתה.

גורמים המשפיעים על הצימוד

סוג החיבור בין תת המערכות: חיבור בעל צימוד **נמוך** הוא כזה שבו תת מערכת א' מבקשת שירות מתת מערכת ב'. חיבור בעל צימוד **גבוה** הוא כזה שבו תת מערכת א' יכולה לגשת ולשנות ישירות אובייקט בתת מערכת ב'.

זמן הקישור(Binding time) של החיבור – ככל שהזמן מאוחר יותר, הצימוד יורד. ניתן להסתכל על נושא זה בצורה של דוגמא של קבועים ב-C. אם אנו נשתמש במספר עצמו בעת כתיבת הקוד, למעשה ה-Binding time שלנו יהיה מאוד "קרוב", והתלותיות תהייה גדולה יותר. מצד שני, אם נכתוב את אותו המספר כ-קבוע, אנו למעשה "נדחה" את זמן ה-binding עד לריצת התוכנית, ובכך נבטיח צימוד חלש יותר.

כמות החיבורים בין תתי המערכות – הרבה חיבורים ← צימוד חזק.

טכניקות להורדת הצימוד

צמצום המידע המשותף לשתי תתי המערכות – ככל שנצמצם יותר את המידע, בכך נקטין יותר את הצימוד. בדיוק לפי העיקרון השני שהוזכר מעלה בסעיף א'. לדוגמא נסתכל על החניון, אם ניתן גישה למחלקה של פתיחת השער גישה לכלל הרשימה של הרכבים המורשים, הדבר לא ייעל לנו את הכול, ורק ייתן מידע למחלקה שאין לה צורך בו. לאומת זאת אם נעשה תקשורת בין המחלקה של פתיחת השער לבין המחלקה של אישורי הכניסה, נוכל להקטין את המידע ל-"האם יש אישור לרכב שמספרו _____". ובכך להקטין את הצימוד.

הכנסת חוצץ(Buffer) בין תתי המערכות – ניתן להסתכל עליו כמשהו שסופג את השינויים במקום המערכת שאליה הוא מקשר.
לדוגמא, אם קיים לנו database כלשהו, שיש צורך לעבוד מולו. אנו נוכל להיעזר בחוצץ שכזה בכדי "להמיר" פקודות רגילות לפקודות SQL לדוגמא, ובכך, במידה ונרצה לשנות דברים גדולים במבנה של ה-database, לא נצטרך לעשות זאת דרכו, אלא דרך פרט קטן שהוא אותו החוצץ, ובכך נקל על העבודה.
ניתן להסתכל גם על תורים כחוצצים, לדוגמא, אם קיימים 2 תהליכים שפועלים, ונוצר עליהם עומס של בקשות שונות, הדבר יכול להפריע לתפקודם.
מצד שני, אם נבנה תור עבור התהליכים האלו, התור הוא זה ש"יספוג" את העומס, ולא התהליכים.

הפשטה – יוצרת ניתוק בין האובייקטים הקונקרטיים והפרטים שלהם, לבין תת המערכת שמשתמשת בהפשטה ובכך מורידה את הצימוד, בנוסף, הפשטה עוזרת לנו להוריד את כמות הפרטים של תת מערכת מסוימת.

לכידות(Cohesion)

מודדת את דרגת הקירבה הפונקציונלית בין החלקים השונים(מחלקות ותתי מערכות) אשר מרכיבים תת מערכת. קרי, עד כמה באמת הקישורים בין המערכות השונות "צריכות" אחד את השני. כעקרון מנחה נשאף לפרק את המערכת לתתי מערכות בעלות לכידות גבוהה שהצימוד ביניהן נמוך.

סגנונות ארכיטקטוניים

סגנון ה-Repository

בסגנון זה קיימת תת מערכת אחת(ה-Repository) שמחזיקה את המידע, ותתי מערכות אחרות, ניגשות ומשנות את המידע אך ורק דרך ה-Repository. קיימות שתי וריאציות עיקריות לסגנון זה בהתאם לאופי זרימת הבקרה: בסגנון ה-Database תתי המערכות קובעות באופן עצמאי מתי הן ניגשות לנתונים. בסגנון ה-Black Board ה-Repository מפעיל את תתי המערכות בהתאם למידע שהוא מחזיק ולשינויים בו.
הצימוד בין תתי המערכות ל-Repository הוא גבוה. לעומת זאת בין ה-Repository לתתי המערכות הצימוד חלש(במקרה של Database חלש מאוד). הצימוד בין תתי המערכות לבין עצמן נמוך מכיוון שהן מתקשרות רק דרך ה-Repository.

סגנון ה-Model View Controller (MVC)

בסגנון זה נחלק את תתי המערכות לשלושה סוגים:
תתי מערכות מסוג **Model** מתחזקות את המידע שמייצג את עולם הבעיה.
תתי מערכות מסוג **View** מציגות את המידע.
תתי מערכות מסוג **Control** מנהלות את האינטראקציה(קלט) מול המשתמש.
תתי מערכות ה-**Model** מפותחות כך שאינן תלויות בתתי המערכות האחרות(View או Control). ה-**Control** מקבל קלט(עכבר, בחירה מתפריט, לחיצה על כפתור וכו'), פונה ל-**Model** בכדי להעביר ולבצע את הפעולה, וה-**Model**, מודיע לכל ה-**Views** שהוא השתנה. כל **View** שמקבל הודעת שינוי, פונה למודל בכדי לעדכן את התצוגה. מבנה ה-MVC, יוצר צימוד חלש בין ה-Views השונים, וצימוד חלש מאוד בין ה-Model ל-Views ול-Controller.לעומת זאת, יש צימוד חזק בין ה-Views ל-Model, ובין ה-Controller ל-Model ובין ה-Controller וה-Views.

סגנון ה-Client/Server

זוהי וריאציה של Repository שבה תתי המערכות נמצאים בתהליכים נפרדים ובמקרים רבים בקדקודי חומרה נפרדים ובמקרים רבים בקדקודי חומרה נפרדים. ה-Repository נקרא Server ותתי המערכות שניגשות אליו נקראות Clients ה-Clients מתקשרים עם המשתמשים אוספים מהם מידע ומעבירים אותם ל-Server. ה-Clients לא מתקשרים זה עם זה ישירות אלא רק באופן עקיף דרך ה-Server. הצימוד הוא כך שה-Client בעל קשירות חזקה ל-Server מכיוון שהם תלויים בו, ואילו ההפך אינו נכון, שכן ה-Server אינו תלוי ב-Clients בכלל.

סגנון ה-Peer to Peer

תתי המערכות מתפקדים גם כ-Server וגם כ-Client. זרימת הבקרה של כל תת מערכת היא עצמאית, למעט נקודות סנכרון ביניהם. מכיוון שהמידע מבוזר בין ה-peers, יש למערכת יתירות(ניתן לגשת למידע גם כשחלק מה-peers לא זמינים). מצד שני, למערכת כזו יש 2 סכנות עיקריות:
- מצב של Lock Dead בין ה-Peers השונים, כאשר כמה peers זקוקים למידע בו זמנית מכמה peers אחרים יתכן מצב שבו אף peer לא יכול להתקדם כי הוא מחכה ל-peer אחר.
- מצב בו המידע לא קונסיסטנטי, שינויים ב-peer אחד צריכים לעבור לכל ה-peers, אחרת אותו מידע ב-peers שונים לא יהיה עקבי.

סגנון ה-Pipes and Filters

בסגנון זה תתי המערכות מתחלקות לשני סוגים:
filters מעבדים את המידע ו-pipes מחברים בין ה-filters.
יתרונות:
ניתן לראות שאין צימוד בין הפילטרים(מאוד מאוד חלש ביניהם), שכן אם נוציא את Filter 2, וה-Pipe`s הרלוונטיים עדיין יוכלו להעביר את המידע בין Filter 1 ל-Filter 3, למעשה המערכת תמשיך לעבוד. ניתן לבנות מערכת מקבילית לגמרי, ללא צורך לבצע את הסנכרון. שכן ה-pipe כבר מבצע את הסנכרון הנ"ל, ובכך מאפשר לכתוב/לקרוא לפי "תור" מסוים.
חסרונות:
במצב כזה המערכת מאוד מקובעת, לא נצרכות פעולות מהמשתמש ולא ניתן להשפיע על אופן הפעולה של המערכת.