סיכום הקורס – שיטות בהנדסת תוכנה.

כתבו: אלכס איטקין ועידן מתתיהו.

תוכן

מושגי יסוד בהנדסת תוכנה

Project – מטרתו לפתח את המערכת, מורכב מקבוצת הפעילויות (Activities).
Activity – קבוצה של משימות המבוצעות להשגת מטרה ספציפית.
Task – משימה: יחידת עבודה אטומית מבחינה ניהולית משימה צורכת משאבים ומייצרת תוצרי עבודה.
Resource – משאב: נכס הנצרך בכדי לבצע עבודה(זמן, ציוד, כוח אדם).
Work Product – תוצר עבודה: מופק במהלך הפיתוח(מסמך תיעוד/מודל/מסמך דרישות/קוד).
System - מערכת קבוצת רכיבים המחוברים ביניהם ומשתפים פעולה לצורך יצירת שלם מורכב.
Model – הפשטה של חלק מסוים של המערכת.
Requirement – דרישות המתארות את תכונות המערכת, נחלק את הדרישות לשני סוגים:

1. **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).
2. **דרישות לא פונקציונליות(Non-Functional Requirement)**
מתארות תכונות כגון מהירות ביצוע, כמות נתונים מרבית, אמינות, תמיכה בסטנדרטיים פלטפורמות חומרה וכו'.

פעילויות עיקריות בהנדסת תוכנה

1. הבהרת דרישות(Requirement elicitations) – הלקוח והמפתחים מגדירים את תפקיד המערכת, תוצאה של פעילות זו היא תיאור המערכת במונחים של שחקנים(actors) ו- use cases.
Actor – שחקנים: מתארים ישויות חיצוניות המתקשרות עם המערכת.
Use Cases – הם תיאורים של סדרת אירועים, המתארים את פעולות המערכת. כל פעולה מתוארת ע"י Use Case.
2. ניתוח המערכת(System analysis) – תפקיד הניתוח הוא לבנות את מודל המערכת, במהלך פעילות זו, המפתחים מגלים דו משמעויות וסתירות ב-Use Cases, ופותרים אותן ביחד עם הלקוח. התוצאה של פעילות זו, היא מודל מערכת המכיל פעולות, עצמים(Objects) עם תכונות, וקשרים ביניהם.
3. תיכון המערכת(System Design) – תפקיד פעילות זו, היא לקבוע את תפקיד המערכת. חלוקה עיקרית לתת מערכות, והדרך בה המערכות יתקשרו זו עם זו. בנוסף, בשלב זה נבחר את הארגון הפיזי של המערכת(פלטפורמה, בסיסי נתונים, מערכת הפעלה), ואת המיפוי של תתי המערכות ליחידות המחשוב הפיזיות. התוצרים של פעילות זו, היא חלוקה ברורה של המערכת לתתי מערכות ודיאגרמת Deployment. המתארת את המיפוי בין רכיבי התוכנה והחומרה של המערכת.
4. תיכון העצמים/ממשקים(Object Design) – תפקיד פעילות זו, הוא להוסיף עצמים מעולם הפתרון בכדי להשלים את מימוש ארגון המערכת. בנוסף, בפעילות זו נגדיר באופן מדויק את הממשקים בין המערכות.
במקרים רבים, נשנה את מבנה המערכת בכדי להשיג מטרות, כמו יכולת הרחבה, פשטות ויעילות.
5. מימוש

6. <u>בדיקות מערכת</u>
בשני הנושאים האחרונים לא נתמקד בקורס בכלל.

UML – Unified Modeling Language

שפת ה-UML מאפשרת לנו לתאר מודלים במערכות תוכנה, אנו נתמקד בסוגי המודלים הבאים:

1. <u>המודל הפונקציונלי</u> - ב-UML, מודל זה מתואר ע"י Use-Case. מודל זה מייצג את התנהגות המערכת מנקודת המבט של המשתמשים.

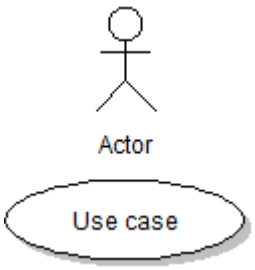
2. <u>מודל העצמים(Object Model)</u> - מתאר את מבנה המערכת במונחים של עצמים(Objects), תכונות(Attributes), וקשרים(Association's). קיימים שלושה סוגים של מודלים כאלו:
   א. מודל הניתוח שמתאר את המושגים והישויות הרלוונטיים לעולם הבעיה.
   ב. מודל העיצוב שמתאר את תתי המערכות והממשקים ביניהם.
   ג. מודל עיצוב עצמים(Object Design), שמתאר בפירוט את האובייקטים הנדרשים למימוש העיצוב.

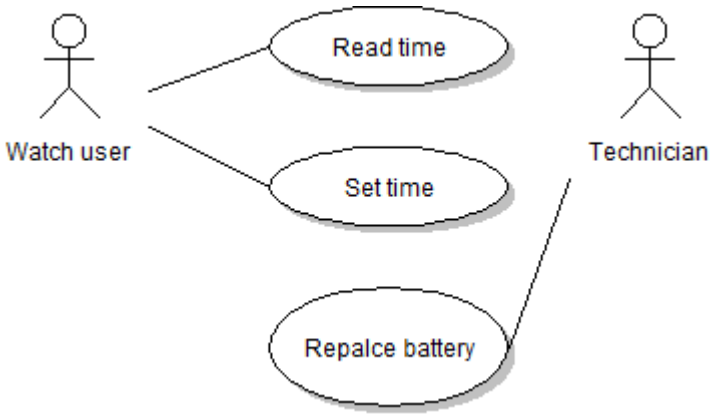
3. <u>המודל הדינאמי</u> – מודל זה מיוצג ע"י הדיאגרמות State diagram ו-Sequence diagram. והוא מתאר את ההתנהגות הפנימית של המערכת.

<u>ארבעת סוגי הדיאגרמות בהן נשתמש</u>

<u>Use Case/Use Case Diagram</u>

Use Case diagram – היא דיאגרמה שמתעדת את גבולות המערכת במונחים של השחקנים שמתקשרים עם המערכת והפעולות שהם מבצעים עליה.





דוגמא ל-USE CASE:

Use case name: Set Time
Participating Actors: Watch user
Flow of events:

1) The watch user presses button 1.
2) The watch starts blinking the hour display.
3) The user presses button 2.
4) …

Entry condition: ...
Exit condition: …
Quality Requirement: …

בתיאור הפשוט של Use-Case קיימות 3 בעיות הקשורות זו לזו:

- איך לתאר חלקי התנהגות שחוזרים שוב ושוב ב-Use Case רבים.
- איך לבצע שינויים בהתנהגות שנובעים ממצבים חריגים(תקלה בתקשורת/אין מקום אכסון/קלט לא תקין).
- איך לארגן Use Case ארוכים מאוד.

בכדי להתמודד עם בעיות אלו, מציע ה- UML את הכלים הבאים:

א. מושג ה-Include

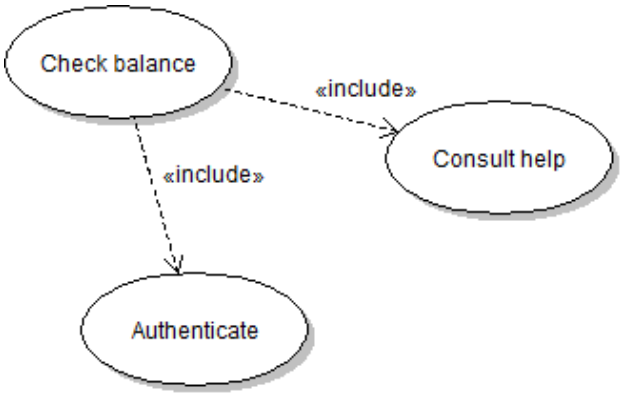
מתאר יחס הכלה, ומאפשר לתאר במקום אחד התנהגות שמופיע בכמה Use Cases.
לדוגמא:

Use case name: Check Balance
Participating Actors: Customer
Flow of events:

1) The customer authenticates with the system(Include the UseCase authenticate).
2) …

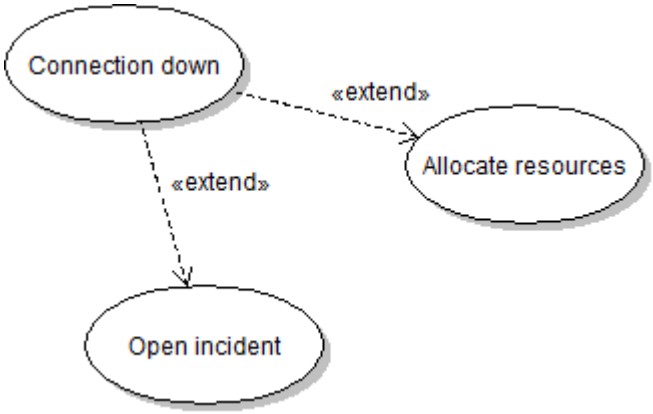
At any point, during the flow, the user may consult the help system(Include consult help – Use Case).

מושג ה-Include, בא לידי ביטוי בדיאגרמת ה-Use Case, בקשר בין ה-UC המכיל לזה המוכל.



ב. **<u>יחסי הרחבה(Extends)</u>**

משמשים לתיאור שינויים, בהתנהגות רגילה בעיקר בהקשר של מקרים חריגים.
למשל:

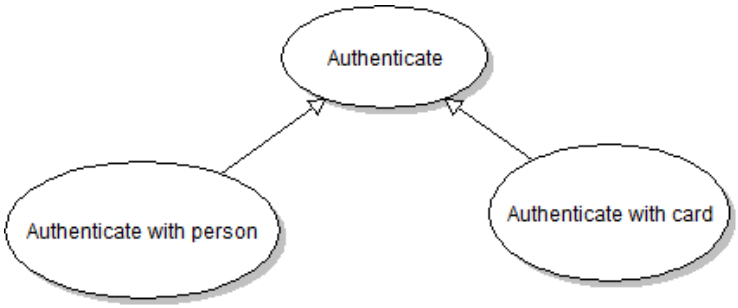


ה-Use Case שמרחיבים אותם, הינם ה-Open incident וה-Allocate resources. הם "לא מודעים" וכמובן אינם משתנים בעקבות ההרחבה. לעומת זאת ה-Use Case המרחיב, חייב לציין את מי הוא מרחיב.

Use case name: Connection down
Participating actors: Field officer, dispatcher
Entry condition: This Use case initiates whenever the communication link between the field officer and the dispatcher goes down. This use case extends open incident and allocate resources.
Exit condition: …
Flow of events: …

ג. **<u>יחסי ירושה</u>**

יחסי ירושה נועדו לתאר UC, ברמות שונות של הפשטה, ה-UC היורש מפורט יותר מה-UC שממנו הוא יורש. לדוגמא:



תנאי הכניסה והיציאה של ה-UC היורשים, **חייבים להיות זהים** לתנאי הכניסה והיציאה של ה-UC המורש. ה-UC היורש, מציין ממי הוא יורש והמקום שבו מגדירים את שמו.

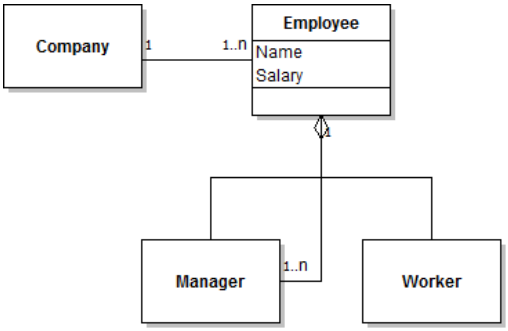
ד. תסריטים(Scenarios)

תסריט הינו מקרה פרטי של UC, שמתאר התנהגות אפשרית אחת שלו.
תסריט מכיל את הרכיבים הבאים: שם, שחקנים, משתתפים וזרימת אירועים.
אין תנאי כניסה ויציאה, ואין דרישות פונקציונליות.
לדוגמא, תסריט לארוחת ערב במסעדה:

Scenario Name: Dinner for two
Participating actors: Alice: Host, Bob: Weightier
Flow of events:

1) Alice welcomes the couple and asks the system to show the available tables.
2) The system indicates that tables 2,3 and 8 are available.
3) Alice reserves table 3, and asks the system to assign bob to that Table.
4) The system…

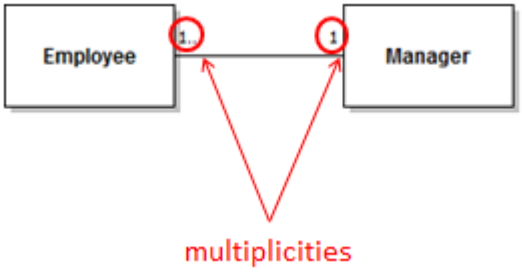
למעשה תסריט מהווה דיאלוג בין אחד המשתתפים למערכת.
ולכן תמיד נצפה לזרימת אירועים בסגנון פנייה/תשובה.



Class Diagram

מחלקה(Class) – היא הפשטה אשר מתארת תכונות משותפות לקבוצה של עצמים.
עצם(Object) – הוא ישות שיש לה מצב המתואר ע"י התכונות והקשרים שלה לעצמים אחרים.
נשתמש ב-Class Diagram, כדי לתאר את המערכת, במונחים של מחלקות, עצמים ותכונות.
וכמו כן קשרים בין המחלקות.
מחלקה המתוארת ע"י מלבן, שם המחלקה מופיע בראש המלבן, תכונות המחלקה בתוך המלבן, והקשרים מתוארים ע"י קווים בין מלבני המחלקות.

תפקידים וכפילויות(roles and multiplicities) - מוסיפים מידע על הקשר
כפילויות – מתארות מגבלות על כמות האובייקטים שיכולים להופיע בכל צד של הקשר, לדוגמא:



קוראים את הכפילויות כך:

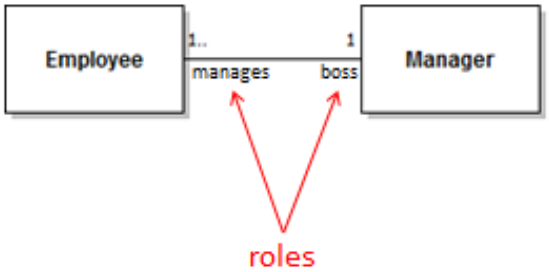
All employees, has exactly one manager.
All managers, has at least one employee.

הטבלה הבאה מתארת את סוגי הכפילויות השונים ואת משמעותם:

משמעות	סימון
בדיוק אחד(one)	1
אפס או יותר(many)	*
אחד או יותר	1..
אפס או אחד(optional)	0-1
בין n ל m	n..m

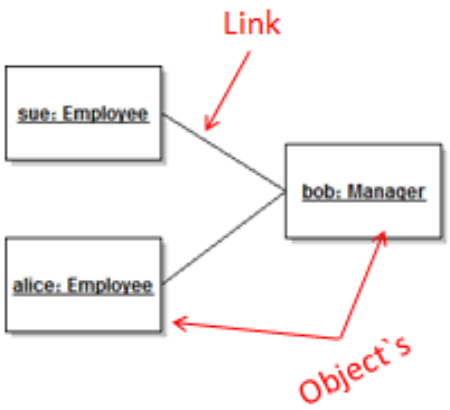
בכדי להבהיר טוב יותר את משמעות הקשר, נשתמש פעמים רבות ב-role(תפקידים).
הרעיון הוא, לתת שם לכל אחד מצדי הקשר בכדי להבהיר את תפקידי הקשר בכל מחלקה.

התפקיד של הקשר מבחינת ה-employee הוא boss.
(הכוונה היא, שה-boss של ה-employee הוא ה-manager),
וכמובן להפיך.



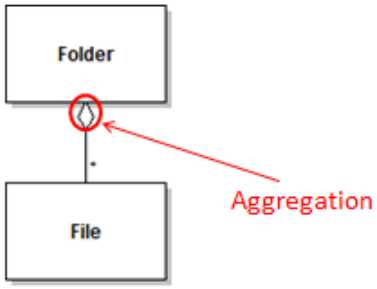
Object Diagram

Object diagram, מאפשר לנו לתאר מבנים קונקרטיים של אובייקטים.

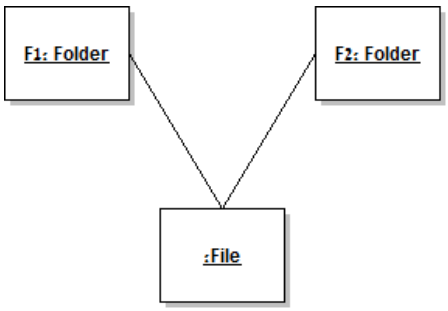


סוגים מיוחדים של קשרים:

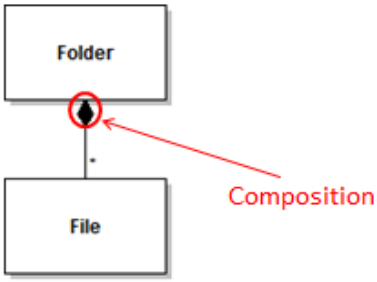
- קשרי הרכבה: קשר מסוג Aggregation, מתאר קשר היררכי בין מחלקות, אבל הקשר אינו אקסקלוסיבי.
  לדוגמא:



אולם, מודל זה מאפשר לאותו קובץ, להימצא בכמה תיקיות שונות:

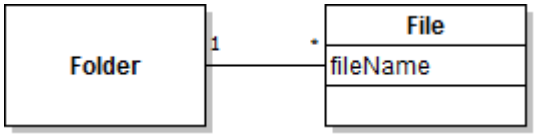


בכדי לאסור על כמה אובייקטים להכיל את אותו האובייקט, ניתן להשתמש ב-**Composition**,שהוא סוג של הכלה אקסקלוסיבית. המשמעות של Composition, היא שמשך החיים של האובייקטים המוכלים, כפוף למשך החיים של האובייקט המכיל, כמו כן בהקשר של הדוגמא שלנו, אותו קובץ לא יכול להיות מוכל ביותר מתיקייה אחת(כמובן קובץ בהכרח כפוף לתיקייה).

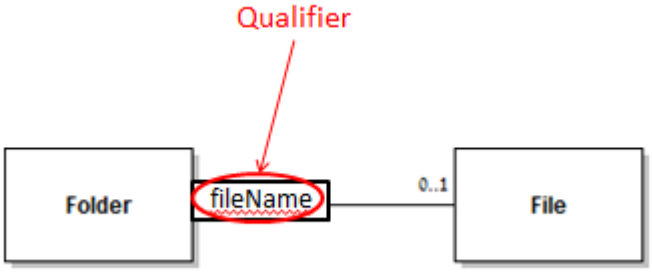


Qualification - כיצד מזהים אובייקט בתוך קשר של אחד לרבים?

לפנינו ניסיון ראשון לתאר קשר בין תיקייה לבין קובץ

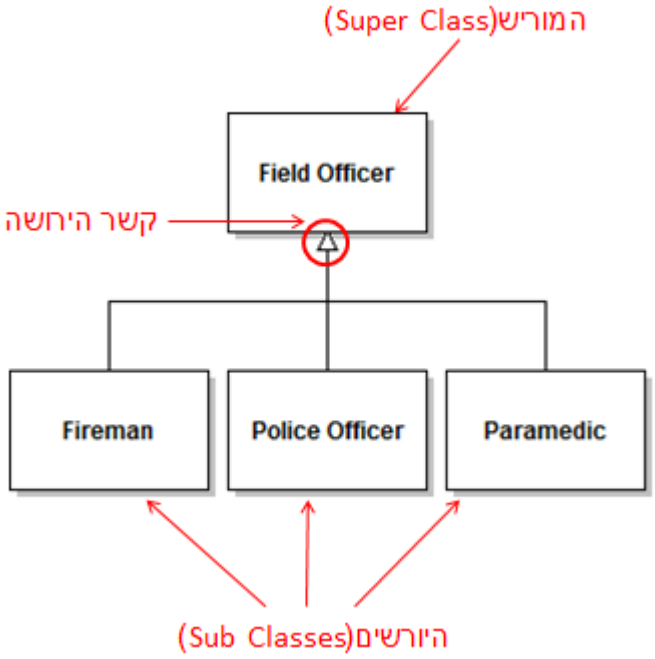


הבעייתיות במודל זה, הינה, שהוא אינו אומר של-filename, יש שם שמזהה אותו באופן יחיד בתוך התיקייה בה הוא נמצא(קרי, יכולים להיות מספר קבצים עם אותו השם באותה התיקייה לפי המודל הנ"ל).
בכדי למנוע, אי הבנה זו, ניתן לתאר את המודל באופן הבא:



ירושה – Inheritance

ירושה מתארת קשר בין מחלקות(Classes), יש לירושה סימון מיוחד ב-UML.

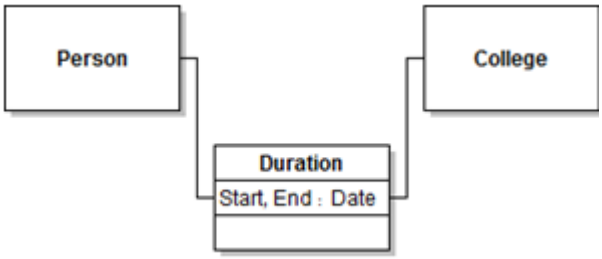


המחלקות היורשות(Sub Classes), מקבלות את כל התכונות(Attributes), והפעולות(Methods) של המחלקה המורישה(Super Class).
כל הופעה/מופע(Instance) של Sub Class, הוא גם instance של ה-Super Class.
לכן, קבוצת האובייקטים, המתוארת ע"י ה-Sub Classes, מוכלת בקבוצה המתוארת ע"י ה- Super Class.

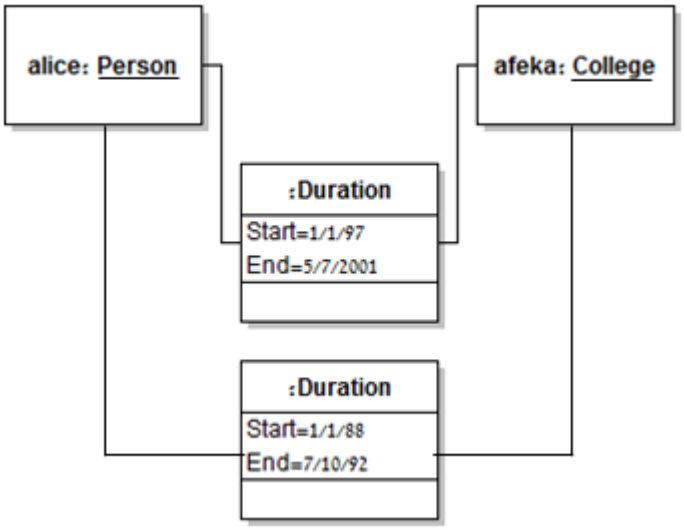
איך להוסיף מידע לקשרים בין מחלקות

נניח שנרצה לדעת באילו פרקי זמן למד כל אדם ב-college מסוים.
ה-UML נותן לנו 2 אפשרויות לתאר מידע זה:

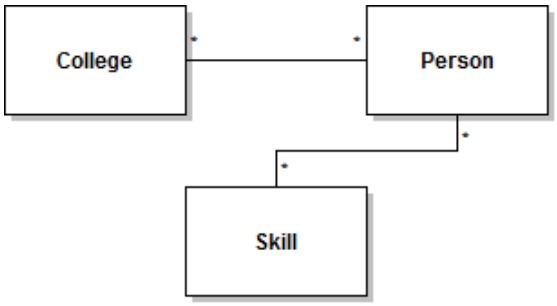
א. נגדיר מחלקה שמייצגת את המידע ונקשור אותה גם ל-Person וגם ל-College.



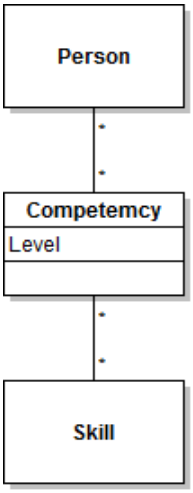
קיימת בעיה עם מודל זה, הוא מאפשר לאותו אדם ללמוד באותו college, בתקופות שונות. מה שכמובן לא מדויק.
נוכל "לתקן" את זה בשימוש במודל הבא:



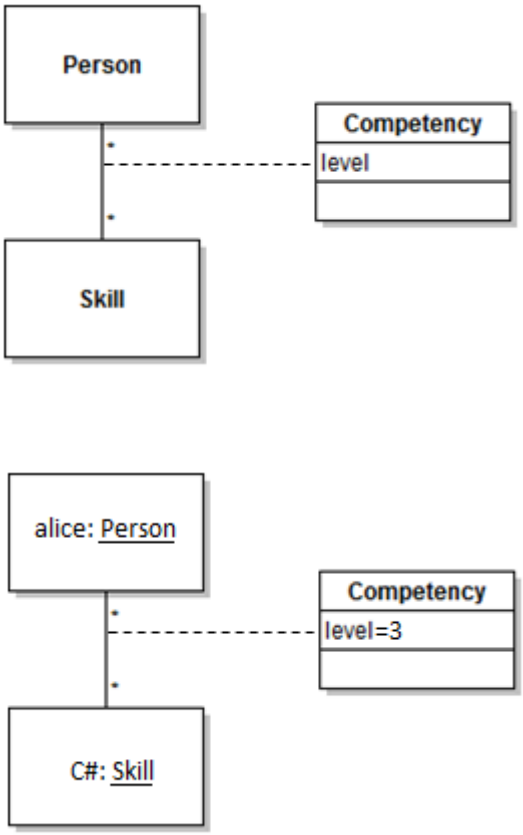
המודל שקיבלנו, שונה מהמודל המקורי, מכיוון שבמודל המקורי – אדם מסוים יכול להיות מקושר ל-college לכל היותר פעם אחת.
לא תמיד הגיוני לבצע שינוי כזה, למשל, נוסיף לכל אדם את המיומנות שלו:



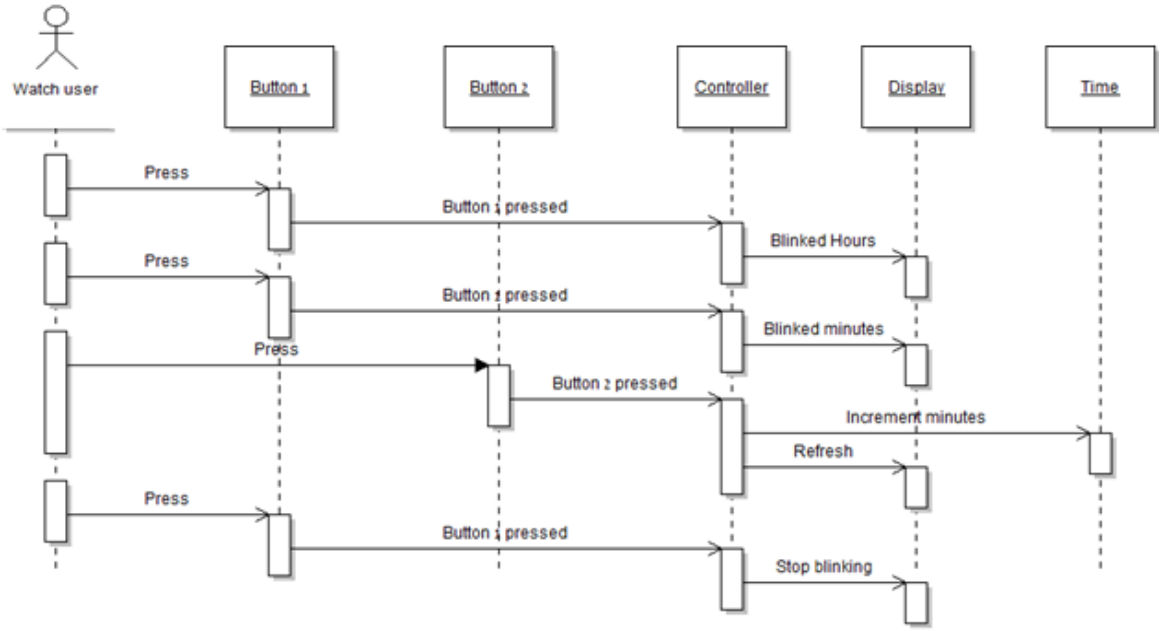
אולם, אנו רוצים לדעת, מהי רמת המיומנות של כל אדם. אם נוסיף מחלקה עבור רמה, נקבל מודל לא הגיוני. במצב כזה, יכול להיות לאותו אדם, שתי רמות מיומנות שונות עבור אותו ה-skill וזה לא הגיוני(לא יכול להיות שלאותו אדם, יהיה רמת מיומנות מתחיל בשפת C, וגם מקצוען בשפת C).



בכדי לפתור בעיה זו, ה-UML מאפשר לנו להצמיד מחלקה לקשר, כלומר, ניתן להוסיף מחלקה לקשרים, מבלי להוסיף/לשנות את התכונות בין שתי המחלקות בקשר הקיים.



Sequence Diagram

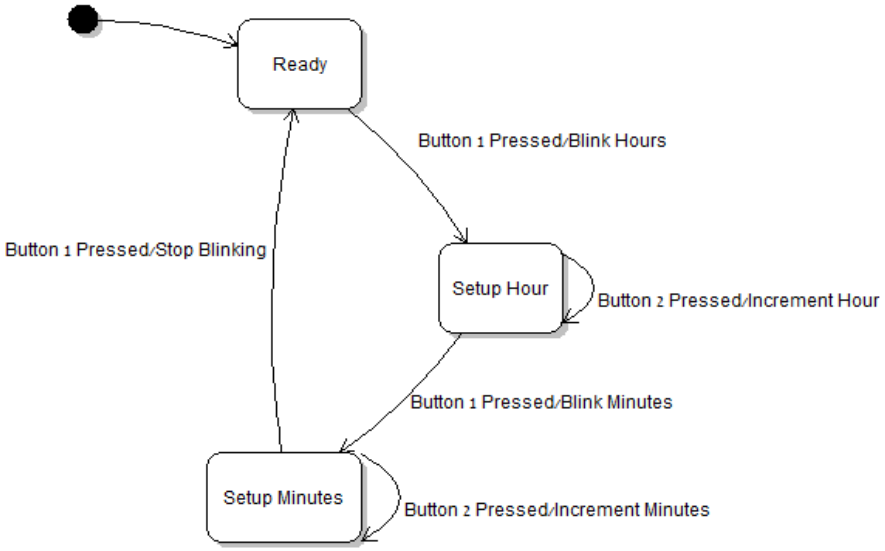


דיאגרמה זו מתארת את תבנית חילופי ההודעות בין קבוצת עצמים ושחקנים, במהלך תסריט ספציפי(אפשרות אחת של ביצוע Use Case).
כל עצם ושחקן, מתואר ע"י קו מקווקו אנכי(Life Line).
הזמן זורם מלמעלה למטה, חצים אופקיים בין הקווים, מתארים את ההודעות שהשחקנים והעצמים שולחים זה לזה.

פרטים נוספים – בעמ' 33.

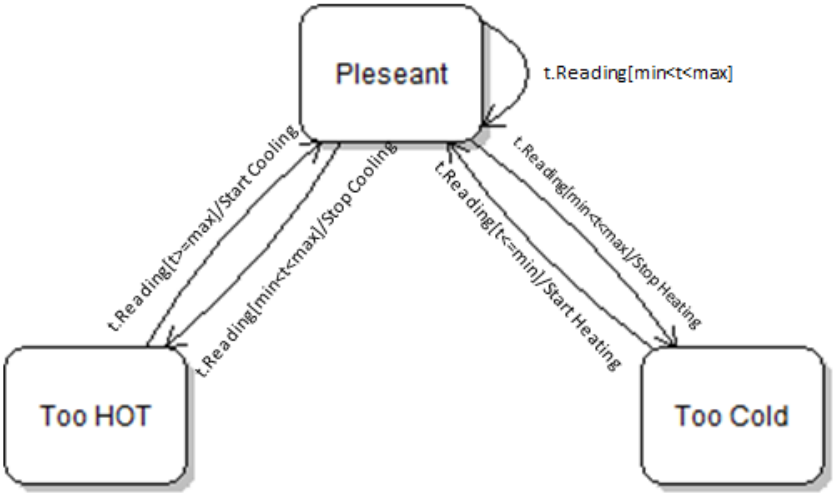
State Diagram

דיאגרמה המציגה התנהגות של אובייקט ספציפי, עבור כלל ה-Use Cases הקשורים אליו.
לדוגמא, דיאגרמת המצבים של ה-Controller של השעון:



דיאגרמת מצבים תשמש אותנו בכדי לתאר כיצד מצב של אובייקט מסוים משתנה בהתאם לפעולות המבוצעות עליו. במצב אנו מתכוונים לקבוצת תכונות של האובייקט, אשר אחריהן אנחנו מוכנים לעקוב.

ניתן לתאר מעברים בין מצבים, בעזרת תנאים מסוימים, אותם נסמן בסוגריים מרובעים([]).
למשל, דיאגרמת מצבים של בקר המזגן:



דיאגרמת מצבים - State Diagram

(היכרות עמוקה יותר)
פעולות ואירועים בתוך מצבים.

בתוך דיאגרמת המצבים יש לנו אירוע שהוא משהו חיצוני אשר מגיע למכונה, וכתוצאה מהאירוע הנ"ל אנחנו נבצע את הפעולה(או לעבור מצב כתוצאה מביצוע הפעולה הנ"ל).

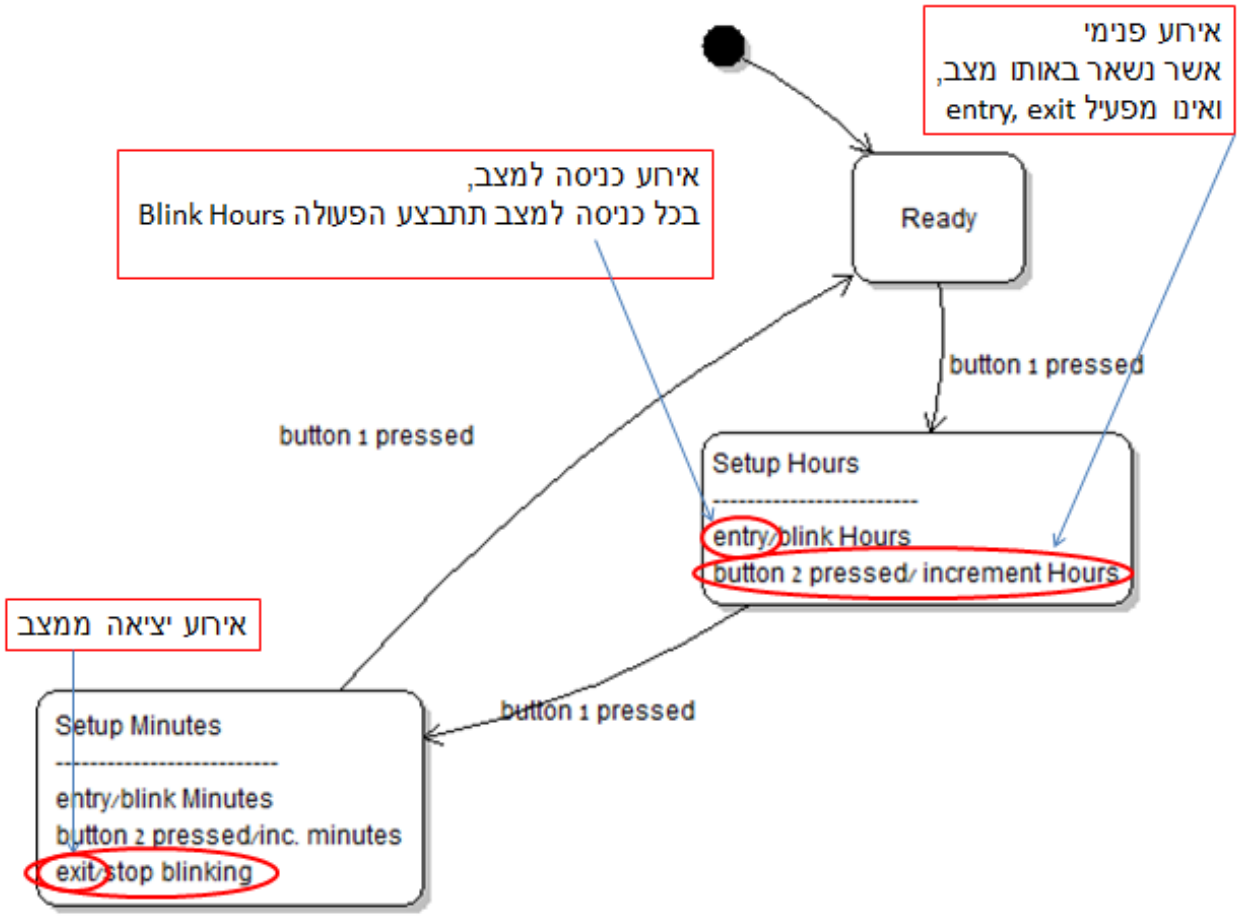
אפשר לתאר בתוך מצב פעולות ואירועים התלויים במצב.

התחביר:

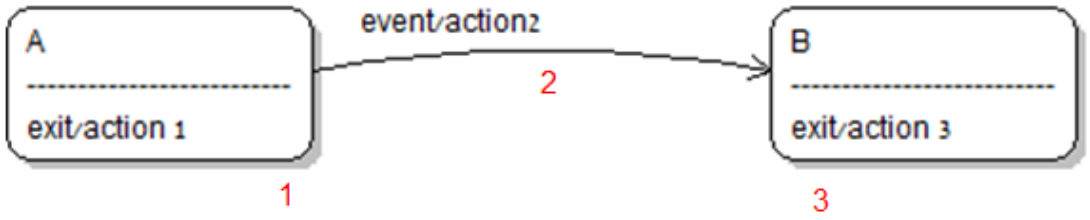
Entry/Action – אומר לבצע את action בכניסה למצב.
Exit/Action – אומר לבצע את action ביציאה מהמצב.

לדומגא

נשרטט דיאגרמת מצבים עבור בקר השעון הפעם ללא אירועים במעברים:



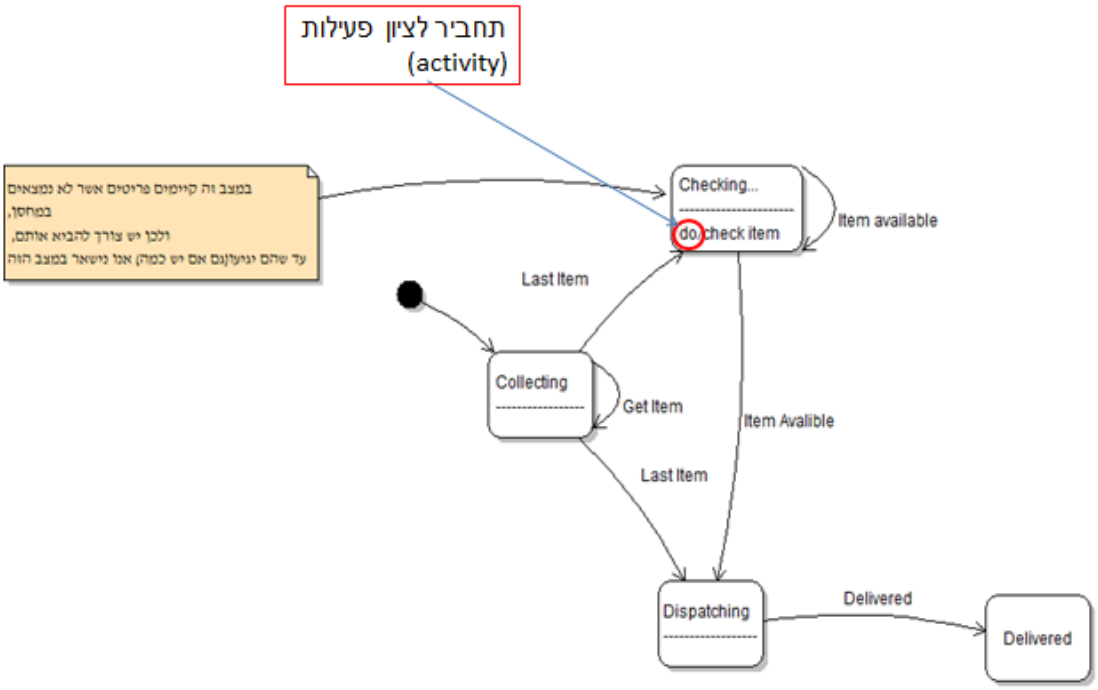
כללי הפעלת אירועים בדיאגרמת מצבים:
בהינתן מעבר בין שני מצבים:



קודם כל יתבצע ה-exit של A, אחר כך תתבצע הפעולה על המעבר ולבסוף ה-entry של B.

נתבונן בדוגמא מעט יותר מורכבת, דיאגרמת מצבים ב- UML היא <u>תמיד</u> עבור אובייקט <u>יחיד.</u>
לכן, שמות המצבים צריכים להתאים לתיאור ההתנהגות של האובייקט.

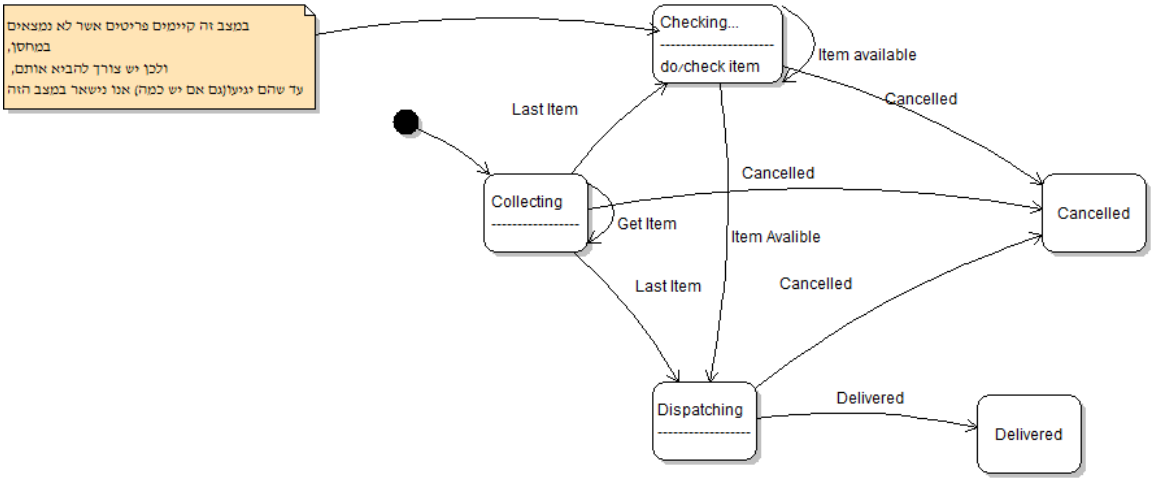
לדוגמא: דיאגרמת עבור אובייקט Order:



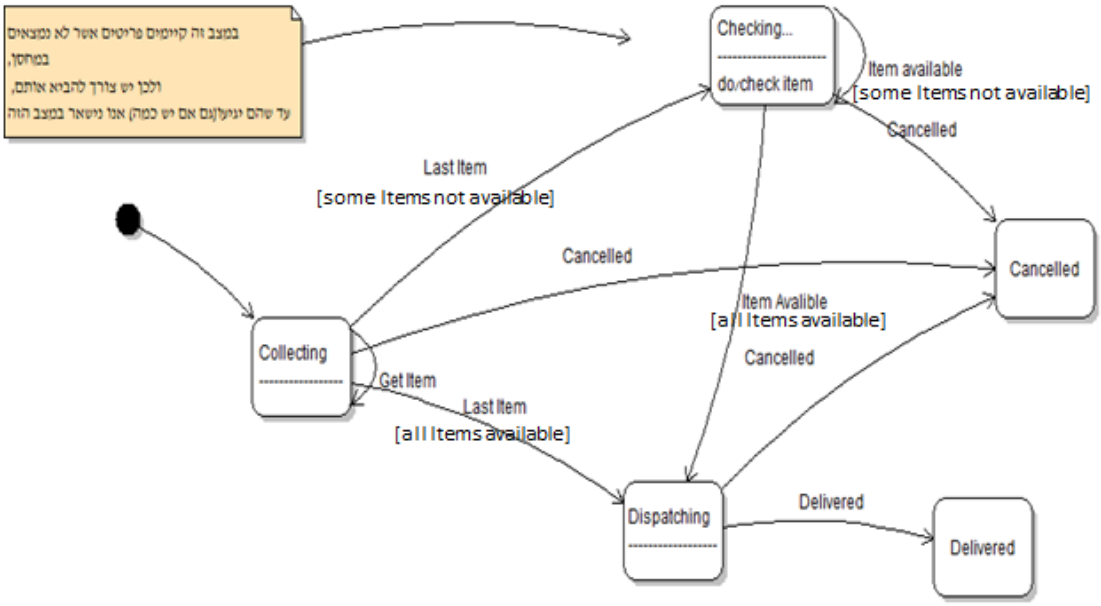
<u>פעילות</u> - פעילות מתארת התנהגות שמתרחשת בזמן שהאובייקט נמצא במצב מסוים,
פעילות לוקחת זמן וניתן להפסיק אותה באמצע בניגוד לכך הפעולה.

<u>פעולה</u>/<u>אירוע</u> – לא לוקחים זמן ואי אפשר להפסיק אותם.

כעת נניח וייתכן מצב בו הלקוח ביטל את ההזמנה שלו, אנו צריכים לבדוק האם ההזמנה בוטלה במהלך "חיפוש" הפריטים(...Checking), בנוסף, יש להבדיל בין המצבים השונים של ה-last item, שכן קיימים לנו 2 באותו השם, והדבר לא צריך להיות כך, ולכן הדיאגרמה תיראה כך:



כעת, נשאל את עצמינו איזה מכשיר UML(...Action, Event) יכול לעזור לנו להבדיל בין החצים השונים עם אותה המשמעות(לדוגמא last Item, item available...)
נוכל להשתמש בתנאי [] (Guard) שנלמד בכיתה(המתאר תנאי לביצוע ה-event הנ"ל).

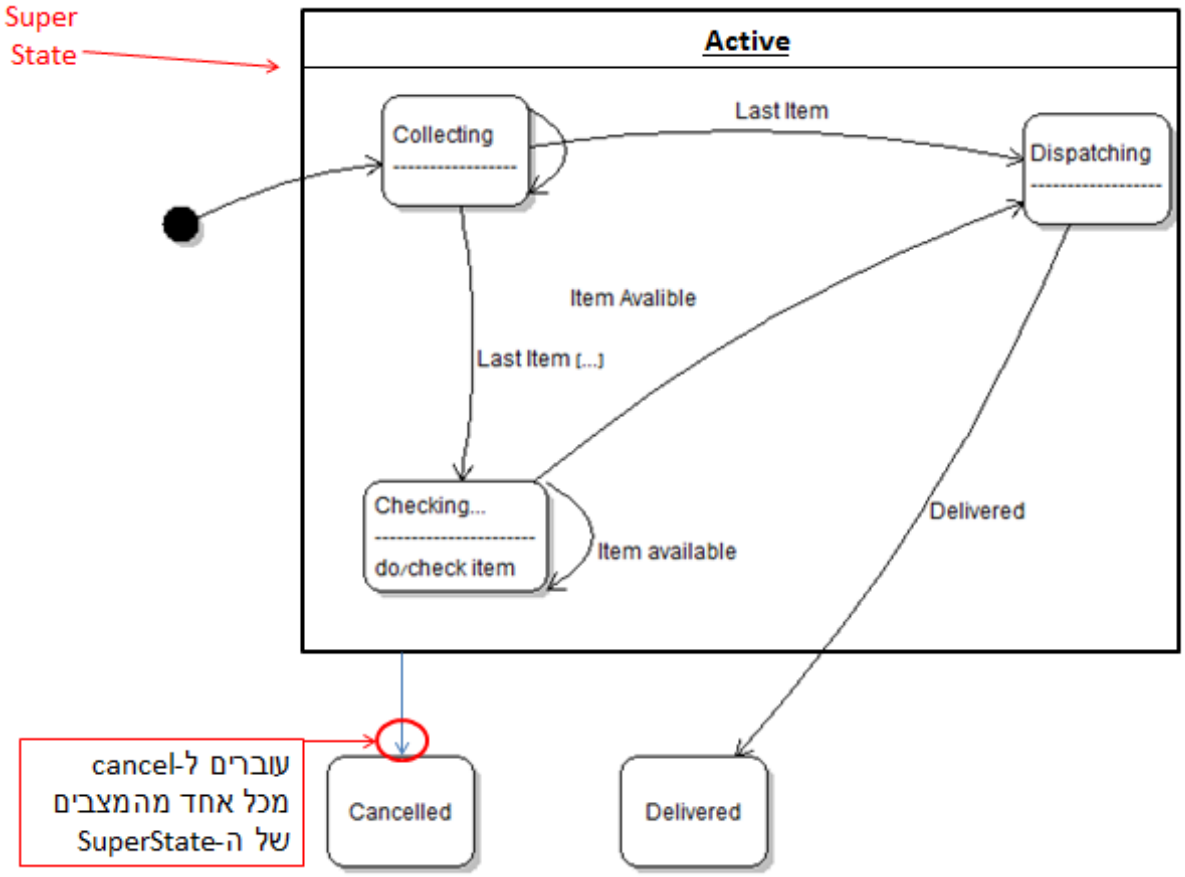


ניתן לראות שהדיאגרמה נהפכה להיות מסובכת לקריאה, ולכן ננסה לפשט אותה בעזרת כלי מאוד חזק הנקרא:

דיאגרמת מצבים היררכיות(Nested state machines)

נשתמש בדיאגרמות מצבים היררכיות כדי להתמודד עם דיאגרמות מצבים מורכבות. הרעיון הוא שמצב בודד יכול להכיל דיאגרמת מצבים שלמה.

דוגמא



ניתן להביט על השיטה הנ"ל כמו חלוקה לפונקציות(מטודות) בשפות תכנות. ניקח קבוצה של מצבים שיהיו מצב אחד אטומי, ונתייחס אליו כמצב של Active במקרה שלנו, כאשר הצוות שנניח יצטרך לעבוד על ה-Delivered ולפתח אותו, יביט על כלל המצבים כ"קופסא שחורה" שאין לו צורך להבין איך היא עובדת, אלא רק איך להתממשק מולה.

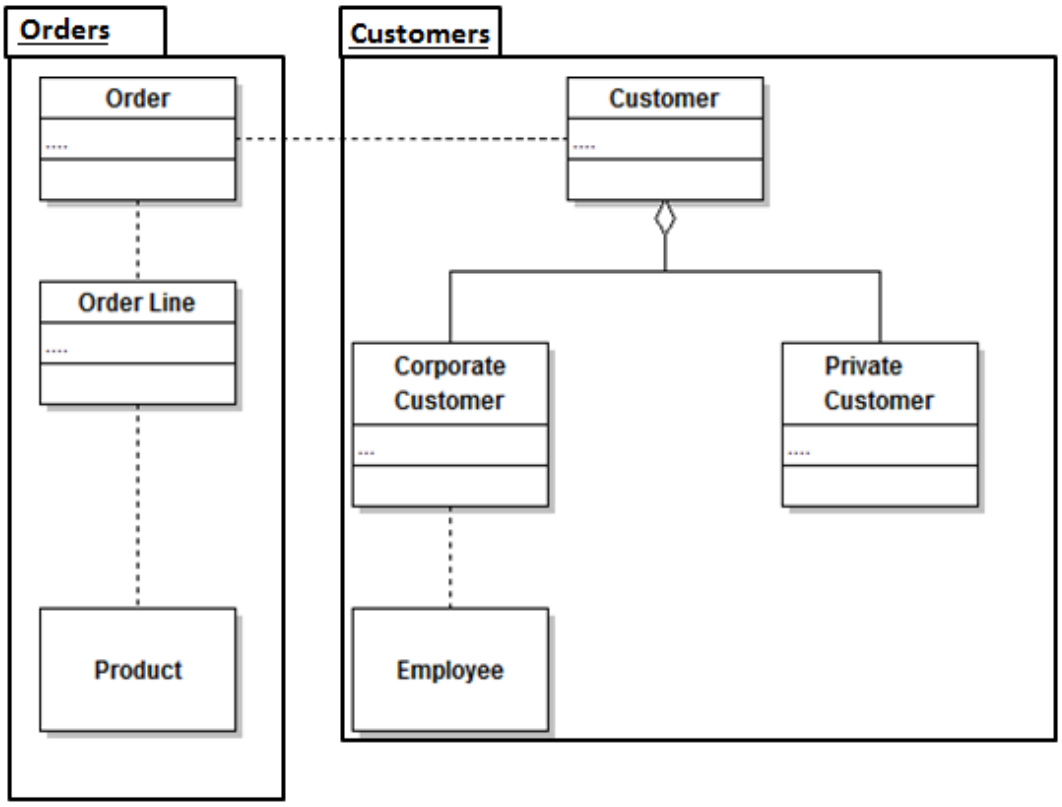
ארגון דיאגרמות UML - בעזרת Packages

כאשר בונים מודלים גדולים מתעורר הצורך לארגן אותם כדי שניתן יהיה להשתלט על כמות הפרטים הרבה.

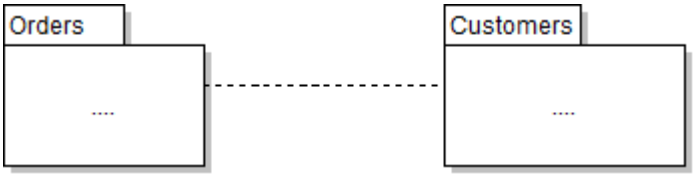
ב-UML, משיגים מטרה זו ע"י איסוף הפרטים שיש להם עניין משותף, ביחידות הנקראות חבילות(Packages).

ההבדל בין ה-superState ל-Package, הוא ש-superState הוא בעצמו State, ואילו ה-Package הוא אוסף של מספר אלמנטי מידול(אוסף Classes, UseCases..).

דוגמא(Class Model)

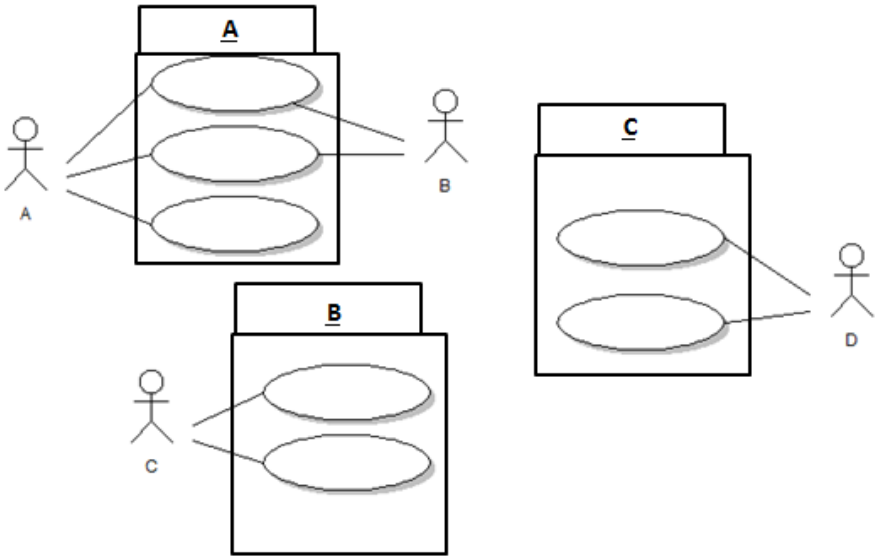


ניתן להציג את הדיאגרמה הזו כך:

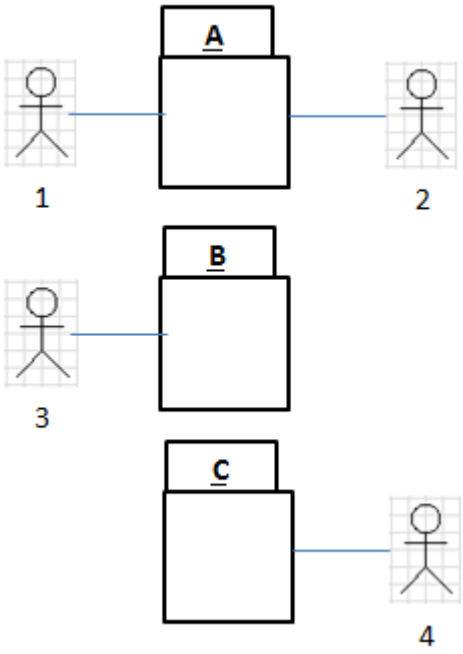


אפשר לתאר את הקשר בין החבילות כדי לקבל מבט "על" מופשט יותר על המחלקות בדיאגרמה.

דוגמא נוספת



ניתן להציג כך:



הבהרת דרישות(Requirements Elicitation)

זוהי הפעילות הראשונה בפיתוח מערכת מורכבת.
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.
הגדרה כזו נקראת אפיון דרישות(Requirements Specification), והיא מהווה חוזה בין הלקוח למפתחים.
* הערה: הדרישות הללו אינם נמצאים איפשהו, אנו צריכים להסיק ביחד עם הלקוח לטובת הבהרה של איזה מערכת הלקוח יצטרך, לא תמיד הלקוח ידע איזה מערכת הוא צריך כדי לפתור את הבעיה הנתונה שלו.
בדרישות שלנו נתייחס לפעולות שנצטרך לעשות, פעולות שלא נצטרך, כמה זמן לוקחת כל פעולה, כל זמן לוקח לבנות מערכת כזאת, או כמות הזמן ללמד את הקהל יעד לעבוד עם מערכת כזו.

פעילות הבהרת הדרישות מורכבת מתתי הפעילויות הבאות:

א. זיהוי השחקנים

המטרה היא לברר מיהם סוגי המשתמשים במערכת?
אנחנו(המגיעים מתוך העולם הטכני) נוטים להדבק לנושאים טכניים, ולכן ננסה להימנע מליפול למלכודות האלו.
הסיבה לזיהוי השחקנים היא לעצור ולחשוב מי צריך את המערכת בכדי לא ללכת רחוק מדי בפרטים הטכניים.

ב. זיהוי התסריטים

התסריטים כשמם כן הם, ממחישים כיצד המערכת פותרת את הבעיה למשתמש.
פיתוח תרחישים קונקרטיים המתארים כיצד ישתמשו במערכת וכיצד המערכת תשיג את מטרות השחקנים. הרמה שאנחנו נרד אליה בפרטי התסריטים השונים שאנו ניצור, תלויה כמובן בדרישת המערכת עצמה, למערכת של קוצב לב, יהיו תסריטים רבים יותר ממערכת של "רישום לאתר" לדוגמא.

לדוגמא

אם המערכת שלי צריכה לתאר מיקום של מטען בשדה תעופה, נוצר צורך ליצור תסריט, שבו בן אדם מגיע לשדה, שם את המזווה במסוע, עוברת בדיקה מסוימת ונכנסת לבטן המטוס, ולאחר מכן הבן אדם מקבל את המזוודות ביעד. דוגמא זו, מציגה בעיה ופתרון אליה ע"י המערכת הנתונה. כמובן שנצטרך להתייחס לבעיות כמו: מטען גדול מדי, או עיכוב בטיסה וכו'. יש המון מקרים כאלה, ואנחנו נצטרך לחקור אותם, רבים ככל שיהיו.

ג. זיהוי use-cases

על בסיס התרחישים הקונקרטיים יוצרים המפתחים את קבוצת הU.C שמתארת באופן מלא את התנהגות המערכת ואת גבולות המערכת.
רק אחרי זיהוי כל התסריטים ניתן לבנות את כלל ה-use cases שיתייחסו לכלל המקרים הללו.
נזכור שה-use cases לא מתייחסים כלל לממשק המשתמש או לביצועים, ולכן יש לתת ייחוס לדברים אלו בנפרד.

ד. **<u>זיהוי דרישות לא פונקציונליות</u>**

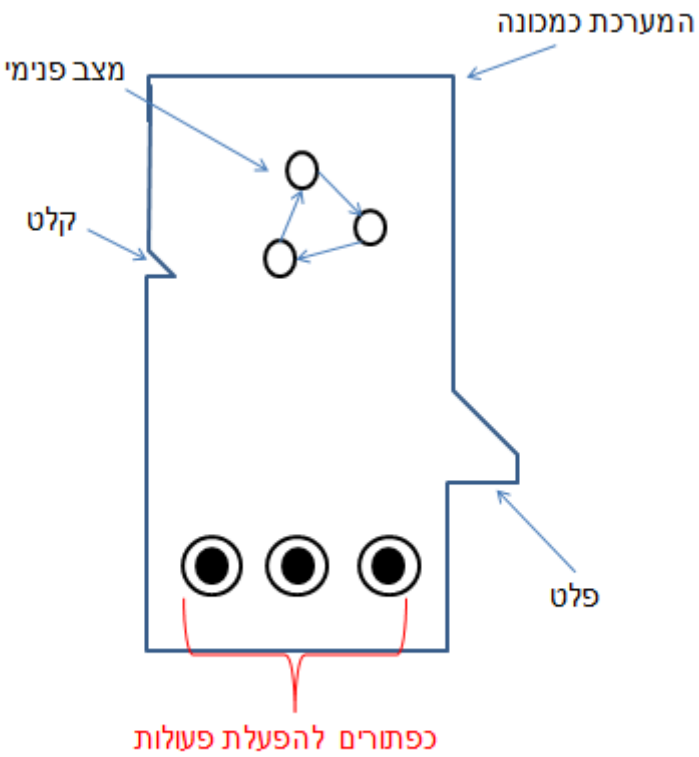
בתת פעילות זו מזהים את מגבלות המערכת, שלא קשורות באופן ישיר לאפיון השירותים שהמערכת מספקת יעילות, פלטפורמות(לדוגמא סביבת הפיתוח, מחשבים שעליהם נרוץ וכו'. נדבר על כך בהמשך), בטיחות, אמינות, וכו'.

<u>דוגמא</u>

קיימות מערכות שכל הזמן צריכות להיות "למעלה", לדוגמא, מרכזיית טלפון. ונצטרך למצוא זמן מסוים בו המערכת לא תהיה פעילה בכדי "לעדכן" אותה.
לעומת זאת, קיימות מערכות שלא צריכות להיות למעלה כל הזמן, לדוגמא רכב או מטוס, תחזוקה של מערכת כזאת יכולה להיות פשוטה וזמינה בכל עת, אך האמינות והבטיחות שמערכת זו צריכה לספק כשהיא כן עובדת, חייבת להיות גבוהה(בשונה ממרכזייה שיכולה "לפשל" לפעמים).

<u>מושגים(עבור הבהרת דרישות)</u>

1. **<u>דרישות פונקציונליות</u>**



בדוגמא זו מתוארת מכונה לדוגמא של מכירת פחיות, ומבט על הדרישות הפונקציונליות שלה.

דרישה פונקציונלית מתארת את התנהגות המערכת כקופסא שחורה המכילה נתונים(מצב המערכת) ואפשר לבצע עליה פעולות לכל פעולה אפשר לתת קלט והיא מייצרת פלט ומשנה את מצב המכונה.

אם ניקח כדוגמא את **מערכת החנייה** עליה דיברנו בשיעור הקודם, ניתן להביט על הדרישות הפונקציונליות שלה כך:

- כשרכב נעמד מול העין האלקטרונית ומספרו נמצא ברשימת הרכבים המותרים לכניסה, המערכת תפתח את השער ותוסיף את הרכב לרשימת הרכבים שנמצאים בחניון.
  ננתח את הדוגמא:
  כשרכב נעמד מול העין האלקטרונית – המצב מתאר קלט של המערכת
  מספרו נמצא ברשימת הרכבים המותרים לכניסה – המצב הנ"ל מתאר תנאי על מצב המערכת.
  המערכת תפתח את השער – מתאר את פלט המערכת.
  תוסיף את הרכב לרשימת הרכבים שנמצאים בחניון – מתאר שינוי מצב של המערכת.

2. **<u>דרישות לא פונקציונליות</u>**

כל מגבלה שנופלת מחוץ למודל הפונקציונלי: שימושיות, אמינות, ביצועים, קלות תחזוקה, וכו'.

לדוגמא דרישות לא פונקציונליות עבור החניון:

- מרגע שרכב נעמד מול העין ועד שהוא נכנס לחניה לא יעברו יותר מ-5 שניות.
- ממשק המערכת יהיה ידידותי למשתמש.
- זמינות, מערכת החניה תהיה פעילה לפחות 99% מהזמן שבו החניון פתוח.
- כמות השגיאות בזיהוי רכבים לא תעלה על 0.1%

2.1 **<u>דרישות ביצועים</u>**

דרישות <u>ביצועים</u> הן דרישות לא פונקציונליות <u>שניתן למדוד</u>, בדוגמא שלנו, ניתן לראות דרישות ביצועים בסעיפים הבאים:

- מרגע שרכב נעמד מול העין ועד שהוא נכנס לחניה לא יעברו יותר מ-5 שניות.
- זמינות, מערכת החניה תהיה פעילה לפחות 99% מהזמן שבו החניון פתוח.
- כמות השגיאות בזיהוי רכבים לא תעלה על 0.1%

אלו לדוגמא מדגימים לנו את ה<u>דרישות ביצועים</u> של המערכת.

לעומת זאת, דרישה "בעייתית" היא:

- ממשק המערכת יהיה ידידותי למשתמש

שכן קיימת בעיה להראות שהדרישה לא מתקיימת.

<u>נשים לב</u>

כשנבנה אפיון, ונצטרך לפרט על דרישות המערכת, אנו נצטרך לפרט עד למצב שבו נוכל להגדיר קריטריון כלשהו עבורו הדרישה הנ"ל לא תתקיים. שכן עבור הדרישה "ממשק ידידותי למשתמש", קשה לקבוע מתי היא לא תתקיים, לדוגמא בגלל שזה

תלוי במשתמש עצמו. לעומת זאת, 5 שניות שבהן הרכב עומד מול השער הינן זמן מדיד ולכן הדרישה היא דרישת ביצועים תקינה.

המסקנה בכלל, היא לבנות מערכות **ברות בדיקה**.
כאלו שנוכל לבדוק ולראות קיום/אי קיום של תנאים כאלה ואחרים שהגדרנו כהכרחיים.

2.2 **מגבלות(Constraints)**

סוג נוסף של דרישות לא פונקציונליות הן מגבלות(constraints) על מימוש המערכת: איזה מערכת הפעלה, איזה שפת תכנות, איזה מערכת הפעלה, נגישות למוגבלויות, רגולציה, אריזה.

קריטריונים עבור דרישות(מגבלות/דרישות)

כדי לוודא שלדרישות שאנחנו רושמים יהיה ערך עליהן לענות על הקריטריונים הבאים:

- **שלמות(Completeness)**
  כל ההתנהגות הרצויה חייבת להיות מתוארת על ידי הדרישות.
  כל המגבלות הרצויות חייבות להיות מתוארות ע"י הדרישות.

- **חד משמעיות (Unambiguous)**
  אסור שיהיו דרישות שאפשר להבין ביותר ממשמעות אחת.

- **עקביות (Consistency)**
  אסור שיהיו דרישות סותרות.

פעילות הבהרת הדרישות

1. זיהוי שחקנים
זוהי הפעילות הראשונה בתהליך הבהרת הדרישות, לחלק זה יש 2 מטרות חשובות:
   א. היא מגדירה את גבולות המערכת.
   ב. היא עוזרת להביא בחשבון את כל נקודות המבט השונות של משתמשי המערכת.

שאלות מנחות לזיהוי השחקנים:

- מיהם סוגי המשתמשים אשר זקוקים למערכת לצורך עבודתם?
- מיהם סוגי המשתמשים אשר עבורם פותחה המערכת?
- מיהם סוגי המשתמשים אשר יתחזקו וינהלו את המערכת?
- אילו מערכות חיצוניות יתקשרו עם המערכת?

לדוגמא בכדי לראות את ההבדל בין 2 השאלות הראשונות, נשאל שאלה פשוטה לצורך דוגמא, עבור חברת אל-על יש צורך לפתח מערכת לרכישת כרטיסים.
בדוגמא הנ"ל, התשובה לשאלה הראשונה מיהם המשתמשים ש**זקוקים** למערכת,

תהייה אל-על.
ואילו התשובה לשאלה השנייה, מי **ישתמש** במערכת, תהייה לקוחות אל-על(הנוסעים).
וכך בעצם ניתן לראות בבירור את ההבדל בין 2 השאלות.

<u>נפעיל לצורך דוגמא את השאלות הללו על **מערכת החנייה** שנלמדה קודם:</u>

<u>לדוגמא נזהה את השחקנים במערכת החניה:</u>

ש: איזה סוגי משתמשים זקוקים למערכת החניה לצורך עבודתם?
ת: הנהלת החניון והטכנאים.

ש: לאיזה סוג משתמשים פותחה המערכת?
ת: לנהגים חברי הסגל

ש: מיהם המשתמשים שיתחזקו/ינהלו את המערכת?
ת: הטכנאים יתחזקו, והנהלת החניון תנהל.

ש: אילו מערכות חיצונית יתקשרו עם המערכת?
ת: אין.

מרגע שזיהינו את השחקנים השלב הבא הוא לזהות איזה פעולות יבצע כל שחקן.
את המידע הזה, אנו נפיק מהתסריטים:

2. <u>זיהוי תסריטים</u>

שאלות מנחות לטובת זיהוי התסריט:

- מהם המשימות(מטרות), שהשחקן רוצה שהמערכת תבצע?
לכל שחקן במערכת יש מטרות שהוא צריך מהמערכת, ומה שאנחנו צריכים לעשות זה כמובן למצוא אותם(להבין מה הם), ולעשות תסריטים שידגימו כיצד השחקן הזה ישיג את המטרות שלו, וכמובן מה(איזה בעיות) ימנעו ממנו להשיג את המטרות האלו.

- לאיזה מידע השחקן זקוק? מי יוצר את המידע? האם אפשר לשנות או למחוק את המידע? מי יכול לעשות זאת?
הרבה פעמים מערכות שמכילות נתונים שהשחקן צריך, השאלה היא איך הם נוצרים, וכיצד הם מועברים לשחקן והאם הנתונים הללו ניתנים לשינוי, אם כן ע"י מי וכו'.

- איזה שינויים חיצוניים השחקן צריך להודיע למערכת? באיזו תדירות? מתי?

- על איזה אירועים המערכת צריכה להודיע לשחקן?

נזהה תסריטים עבור החניון, נזהה תסריטים עבור הנהלת החניון.
<u>לדוגמא(שוב חניון באפקה)</u>
ש: מהן המשימות שהנהלת החניון רוצה שהמערכת תבצע?
ת: לנהל את רשימת הרכבים המורשים.

ש: לאיזה מידע הנהלת החניון זקוקה?

ת: קיבולת החניון, כמות הרכבים בחניה, רשימת הרכבים המורשים.

ש: איזה שינוים חיצוניים הנהלת החניון מעדכנת את המערכת?
ת: החלפת רכב של חבר סגל, הוספה או הורדה של רכבים מורשים.

ש: על איזה אירועים על המערכת להודיע להנהלת החניון?
ת: תקלה מכאנית, כניסה או יציאה של רכבים לחניון(כמובן שנשים לב על התדירות שיכולה לשחק תפקיד במצב זה, באיזה תדירות לדוגמא להודיע להנהלה על הרכבים).

איזה תסריטים נפיק מניתוח זה?

- רכב נכנס בהצלחה לחניון.
- איש סגל החליף רכב.
- רכב מנסה להיכנס כשהחניון מלא
- רכב מנסה להיכנס אך יש תקלה במנגנון הכניסה
- חבר סגל עוזב את המכללה

3. <u>זיהוי use cases</u>

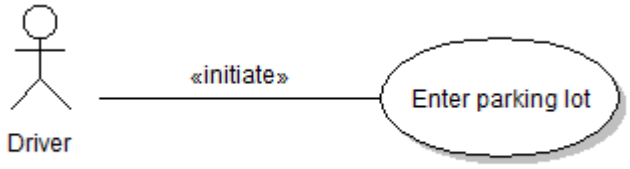
Use Case, מתאר את כל התסריטים האפשריים עבור פונקציונליות נתונה.
קיימות 2 מטרות:
- לקחת את התסריטים ולהפוך אותם לתיאור כללי של ההתנהגות.
- לוודא שהתיאור הזה מספיק מדויק וחד משמעי בכדי שנוכל להשתמש בו לאחר מכן בניתוח מערכת.

את המטרות הללו נחלק ל-2 שלבים:

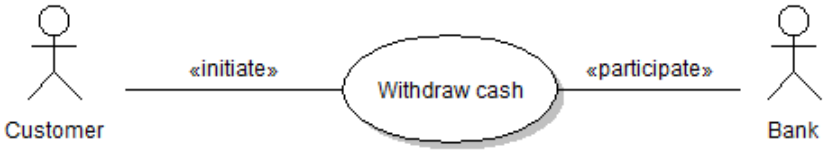
1) בשלב הראשון נתמקד בהסקת ההתנהגות הכללית מהתסריטים.
2) בשלב השני נעבור על ה-UC שהתקבלו ונוודא שהם שלמים ועקביים. במיוחד נתמקד בנושאים הבאים:
   - באיזה ישויות(אובייקטים) המערכת מטפלת?
   - מהן זכויות הגישה(Access Rights), כלומר <u>לאיזה שחקן מותר להתחיל איזה UC</u>?
   - זיהוי וטיפול במקרים חריגים.
   - זיהוי התנהגות משותפת בין כמה UC וחילוצה.

בנוסף, נצמיד לכל UC, את הדרישות הלא פונקציונליות שנוגעות אליו, וזאת בסעיף "Quality Requirements".

4. זיהוי קשרים בין UC לשחקנים.



דוגמא נוספת במערכת ATM:



5. זיהוי ראשוני של האובייקטים בעולם הבעיה.

נבנה מילון מונחי(Glossary) שבו נתעד את הישויות שבהן המערכת מטפלת ואת משמעותן.

כדי לעשות זאת נוכל להיעזר בהיוריסטיקות הבאות:

יוריסטיקה – הוא בן דוד של אלגוריתם, אלגוריתם תמיד מוגדר היטב(לדוגמא למיין, למצוא וכו'). יוריסטיקה לעומת זאת מדברת על טכניקה כלשהי, ולכן לא תמיד היא תעבוד, ואין בטחון לכך. לרוב היא תעבוד. אבל הדבר לא מובטח.

דוגמא ליוריסטיקה: מה לשים קודם בעגלה כשאתה נמצא בסופר?

יוריסטיקה מסיימת תעמוד על: לשים את הדברים הכבדים למטה ולאחר מכן את הקלים יותר.

יוריסטיקה אחרת היא לבחור מוצר מהמדף בלי מחשבה מקדימה, מה שכמובן מקל על העומס הקוגניטיבי שלנו כאשר אנו בוחרים מוצר.

היוריסטיקות שבהן ניעזר:

- מושגים שעל הלקוח להסביר כדי שניתן יהיה להבין את ה-UC. (בייחוד בתחומים ספציפיים, לדוגמא בתרחיש של לקיחת משכנתא, יהיו מושגים רבים שלא נכיר, ולכן נסתכל עליהם)
- שמות עצם שחוזרים על עצמם ב-UC. (בגישה אחרת אפשר פשוט להסתכל על מושגים שחוזרים על עצמם הרבה ב-UC. וכך נזהה את האובייקטים).
- ישויות פיזיות שהמערכת עוקבת אחריהן(בדוגמא של החניון לדוגמא, שמות עצם וישויות פיזיות הן בעצם אותו אחד. אבל במקרה של ה-ATM זה יכול להיות שטרות/כרטיס וכו').
- Use Cases – למעשה זה לא רגיל להסתכל על ה- UC כאובייקט, אנו עובדים בגישת ה-Object Oriented, ולכן נתייחס גם ל-Use Caseים הראשיים והגדולים כאובייקט. בדרך זו, נוכל להשתמש בבקרים מסוימים ש"יבקרו" על פעולת ה-UC הספציפי, כמו כן לבצע "הורשה" של UC ולהשתמש בכלל תכונות האובייקטים בעת העבודה.
- מקורות ויעדי מידע(לדוגמא, מדפסת – הדפסה ממנה תהווה בעצם "גישה לאובייקט" מסוג מדפסת).

<u>חשוב מאוד:</u> המונחים ב-Glossary חייבים תמיד להיות בשפת הבעיה(שפת הלקוח). הדבר אומר שאסור לנו ליפול ל"מלכודת" של דיבור "טכני מדי" ובשפה גבוהה וכו'.

<u>דוגמא למילון מונחים של מערכת החניה</u>

<u>Glossary</u>

Electronic Eye(Eye) – A device that detects license plate numbers of <u>cars</u>.

Car – Any vehicle that occupies a <u>parking space</u> and has a <u>license plate</u>.

Parking space – …

License plate - ….

סיכום פעילות הבהרת הדרישות(בניית מסמך אפיון הדרישות)

- זוהי הפעילות הראשונה בתהליך הפיתוח
- מטרת הפעילות היא לבנות את **מסמך אפיון הדרישות**(Requirements Spec.) אשר מתאר את התנהגות המערכת מנקודת המבט של המשתמש.
- **מסמך אפיון הדרישות** הוא חוזה בין הלקוח למפתחים.
- תהליך זה הוא היחידי בו דרושה השתתפות "אקטיבית" של הלקוח.

- **מסמך אפיון הדרישות מורכב מהחלקים הבאים**:
  1. סדרה של תסריטים ו-Use Cases.
  2. דיאגרמה Use Cases אשר מתארת את המבנה והקשר בין ה-Use Cases השונים ובין השחקנים(actors).
  3. רשימת דרישות לא פונקציונליות.
  4. מילון מונחים(Glossary) אשר מתאר ומגדיר את הישויות המופיעות בעולם הבעיה.

לאחר סיום שלב זה, נשאל את עצמינו מה יהיה השלב הבא בפיתוח המוצר?

נזכור שהמסמך הנ"ל כתוב בשפה טבעית, שפה הנתונה לפרשנויות רבות ודברים כאלה ואחרים שניתן להבין במספר מובנים. ולכן המסמך הנ"ל לא מוגדר היטב.
ולכן תפקידינו הוא לקחת את המסמך הנ"ל, ולהמיר אותו למונחים של עולם התוכנה(אובייקטים/תכונות/ירושה), זאת עדיין מבלי להתייחס לאופן הביצוע עצמו.

למסמך אפיון הדרישות יש שתי בעיות מרכזיות שמונעות ממנו להוות נקודת התחלה לתכנון המערכת:

א. במסמך האפיון חסר תיאור מפורט של האובייקטים אשר מרכיבים את המערכת.
ב. מכיוון שהמסמך כתוב רובו ככולו בשפה טבעית, הוא אינו מספיק חד-משמעי ומדויק כדי שנוכל להשתמש בו כנקודת התחלה לתכנון המערכת.

תפקיד **פעילות הניתוח** הוא להתמודד עם בעיות אלו.

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:

- מודל פונקציונלי(Use Cases)
- מודל האובייקטים של עולם הבעיה(analysis object model)
- המודל הדינמי(מיוצג ע"י דיאגרמות מצבים ו-seq diagrams).

עיקר הפעילות בשלב הניתוח היא עבור עידון המודל הפונקציונלי, בגזירת האובייקטים מתוכו ובתיאור המודל הדינמי.
מודל המערכת יהיה הקלט של הפעילות הבאה – תכנון המערכת.

מושגי יסוד במידול

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.
חשוב מאוד לזכור תמיד שמהחלקות והאובייקטים אותם אנחנו מתארים בעת פעילות הניתוח מייצגים מושגים השייכים לעולם הבעיה.

Entity, boundary & control object

נחלק את מודל העצמים לשלושה סוגים:

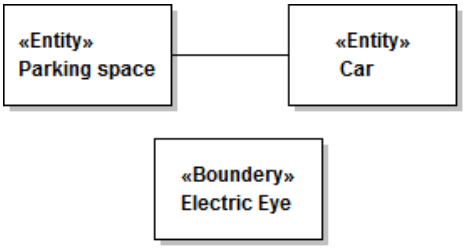
Entity objects – מייצגים את המידע אשר המערכת מנהלת ואשר אחריו היא עוקבת.
Boundary objects – מייצגים את התקשורת והממשק בין המערכת לשחקנים.
Control objects – אחראים על ביצוע ה use case.

היתרון בחלוקה זו הוא שהיא מפרידה את הישויות על פי העמידות שלהם לשינויים ובכך תורמת למודולריות של המערכת.

סוג הישות	מתי משתנה?
Boundary objects	קבועים בין עולמות הבעיה. משתנים בין ממשקים שונים.
Entity objects	משתנים בין עולמות הבעיה, אך קבועים בעולם בעיה ספציפי עבור ממשקים שונים.
Control objects	משתנים בין use cases בכל מערכת.

<u>הרחבה:</u>
החלוקה מבוצעת בצורה כזו כדי לקבל חלוקה מודולרית. אם נסתכל על אובייקטים המייצגים **ממשק**, אלו בד"כ אובייקטים שכלל לא משתנים. כלומר, ברוב המערכות שנפתח נקבל את האובייקטים האלו מהפלטפורמה (כפתורים, טפסים, סליידרים וכו). אלו למעשה אבני הבניין שיוצרים ממשקים.
לעומת זאת, **entity objects** אלו אובייקטים שלא תלויים בממשק אלא בעולם הבעיה עצמו. כלומר, משתנים בין תחומי עולם הבעיה אבל באותו עולם בעיה הם נשארים קבועים כך שאם נרצה לפתח בכלים שונים נשתמש באותם **entities** אך ב**boundary** שונים.
control הוא הכי ספציפי וישתנה לרוב עבור כל שינוי הן בעולם הבעיה והן בממשק.

נוכל להבחין בין סוגי הישויות בדיאגרמת מחלקות כך:



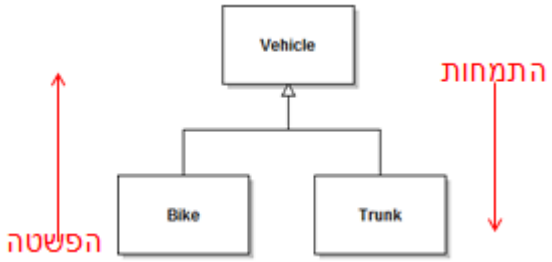
<u>המלצה</u>: לא לציין באותה דיאגרמה Boundary ו- Entity, אלא להשתמש בשני package נפרדים.

<u>הכללה והתמחות –</u>

- הכללה (Generalization)
  זוהי פעילות שמזהה מושגים מופשטים המשותפים לישויות ברמת הפשטה נמוכה יותר.
  לדוגמא: המשותף לאופניים קורקינט משאית ואופנוע הוא שהם <u>כלי רכב</u>.→ הכללה

- התמחות(Specialization)
  זוהי פעילות אשר מזהה מושגים ספציפיים מתוך מושגים מופשטים.
  לדוגמא: המושג "דלק" הוא תיאור כללי למושגים "סולר" ו-"בנזין".

שתי פעילויות אלו הן פעילויות שונות, אך מיוצגות באותו אופן בClass Diagram - כמבנה ירושה.



מ-Use Cases לאובייקטים

נעבור עתה על הפעילויות השונות שמפיקות מאפיון הדרישות את מודל הניתוח:

פעילות מס' 1: זיהוי וסיווג אובייקטים

א. 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 מתבסס 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.

ב. 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.

ג. 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.
לדוגמא: הקצין שנמצא בשטח צריך להעביר הודעה למטה, אז יש העברה כלשהי בצורת טופס שהקצין שולח למטה, הקצין במטה עובר על הטופס ושולח תגובה לשטח, שתי הקצינים מתממשקים מול המערכת, וקיימת להם התנהגות בקרה שונה מול המערכת, האחד שולח את הדוח והשני צריך "לאשר קבלה" לדוגמא, אז עבור כל אחד מהם יהיה אובייקט בקרה שייצג את ה"שחקן" בהעברת המידע ביניהם.

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.

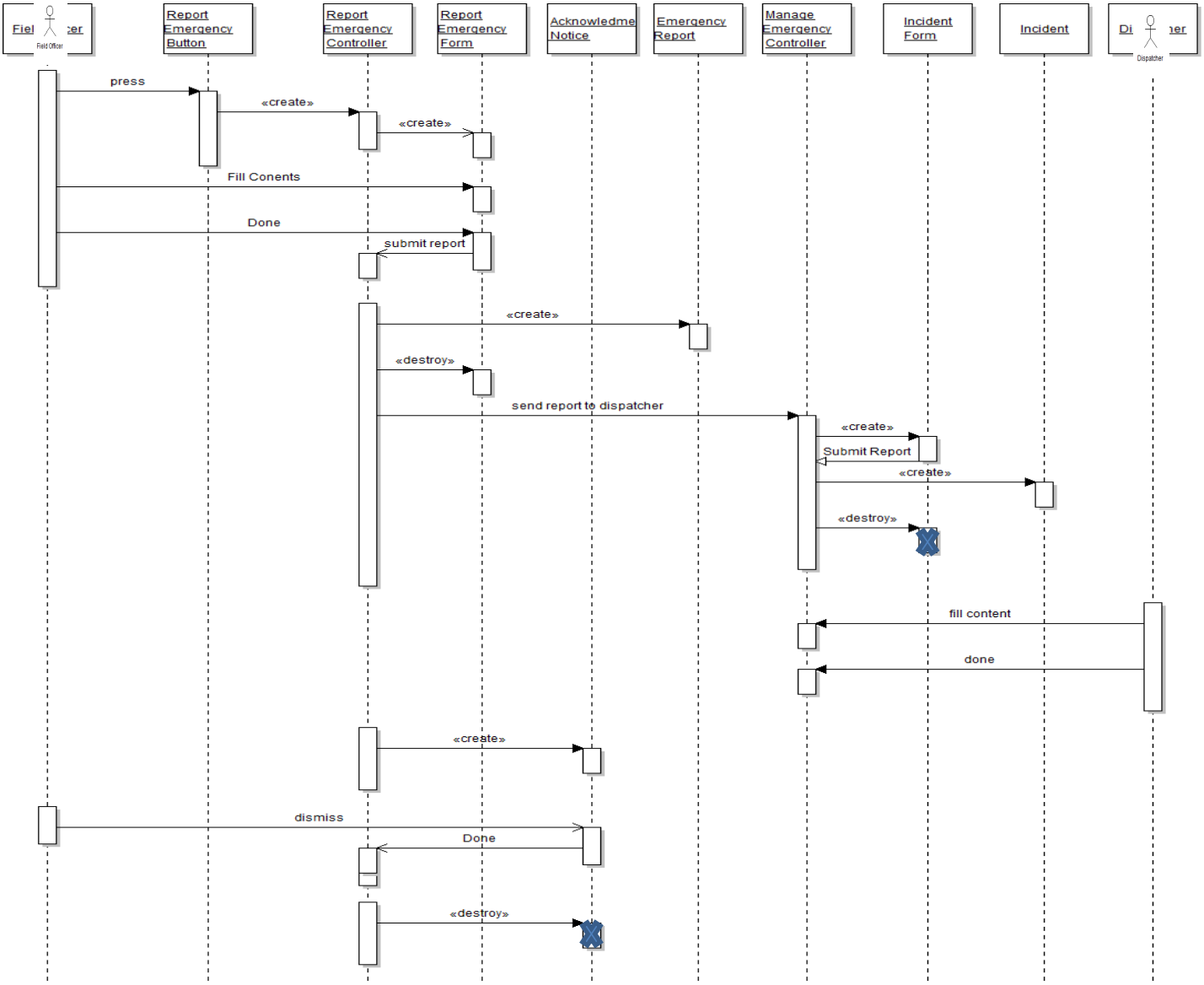
היוריסטיקות לזיהוי אובייקטי Control

- נזהה אובייקט Control אחד לכל U.C.
- נזהה אובייקט Control לכל שחקן ב-U.C.

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.

תיאור מצב: יש קצין בשטח שרוצה להתריע על מקרה חירום. המערכת מציגה לו טופס למלא פרטי מקרה. שולח למטה, במטרה מקבלים את התיאור והקצין שנמצא במטה אחראי על התגובה, פותח אירוע אצלו ביומן ושולחן אישור קבלה לקצין בשטח עם תיאור קצר של מה שעומד להתרחש. נתאר כאן sequence diagram על המצב הנ"ל.



עקרונות לבניית sequence diagram.

- 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**) ותכונות(**attributes**) של האובייקטים

למה בכלל צריך את הקשרים הללו?
בעיקר בגלל 2 סיבות:

1. במקרים רבים המידע שהמערכת שומרת ועוקבת אחריו נמצא בקשרים.
2. 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).
עוד דוגמא: איזה אנשים עובדים עבור איזה חברות, נגיד ונרצה פעולה שמעבירה עובד מסוים לחברה אחרת. הקשרים שיצרתי "מחייבים" אותי מה אני צריך לעשות בפעולה, כי לדוגמא אם העובד "עובר מקום עבודה", הדבר אומר שהוא כבר לא עובד בחברה הישנה שלו. כל זאת נעשה בכדי שהמודל יהיה עקבי.

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, ….

באיזה attributes לדוגמא נרצה להתמקד?
כאלה שיש להם שימוש במערכת, לדוגמא עבור רכב שנכנס לחניה, ה-attributes שיהיה רלוונטי לדבר עליו יהיה הלוחית זיהוי שכן היא בשימוש של העין האלקטרונית והמערכת כולה.

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.

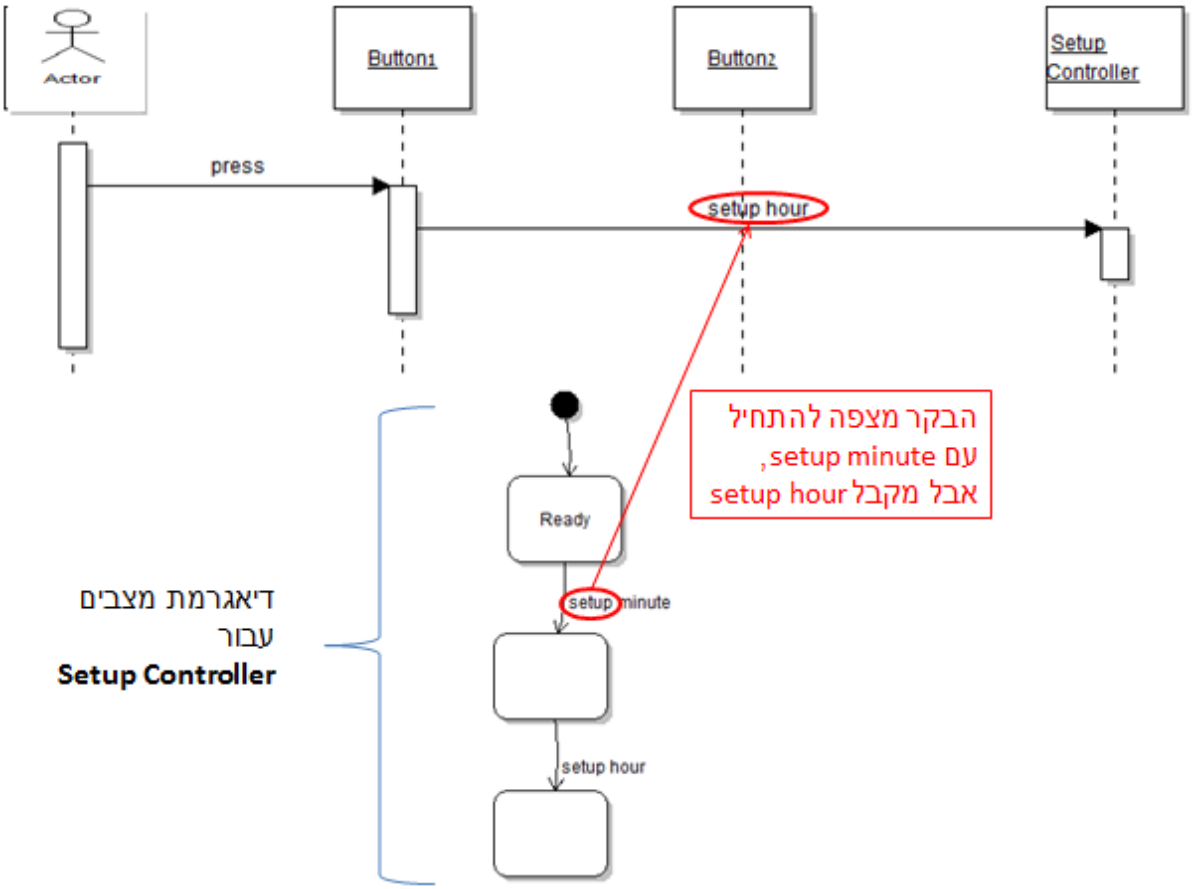
תכונה היא יחס בין אובייקטים לערכים, לדוגמא, השם של עובד, מס' לוחית הזיהוי של הרכב, הקואורדינטות של בית קפה במערכת לאיתור מסעדות וכו'.
מכיוון שתכונות הן החלק הכי פחות יציב באובייקט לא נתמקד בתיאור מלא של התכונות אלא נחפש רק את התכונות שמהותיות אל המודל, בראש ובראשונה תכונות אשר משמשות לזיהוי האובייקטים.
נוכל להיעזר בהיוריסטיקה של Abbott כדי לזהות תכונות ע"י כך שנחפש ב-U.C ביטויי תואר לדוגמא:

Emergency ←(attribute of..) description
Patient ←(attribute of..) identification number
Car ←(attribute of..) license plate

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 ולודא 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.

דוגמא להתנהגות לא עקבית:



הפעולה הראשונה כאן שמתבצעת על ה-Setup Controller היא ה-Setup Hour.
לעומת זאת ב-State diagram של ה-Controller הוא מצפה לקבל setup minute כלחיצה ראשונה.

סיכום פעילות הניתוח

פעילות הניתוח מקבלת כקלט את מסמך אפיון הדרישות, ומפיקה את מודל המערכת.
מודל המערכת, מורכב משלושה חלקים: ה-Use Cases, מודל הישויות(Classes) והמודל הדינמי.

תכון מערכת(תכנה) – Software system design

פעילות התכון מקבלת כקלט את מודל המערכת. מטרתה:

א. למפות את האובייקטים שזוהו בשלב הקודם לתתי מערכות.
ב. למפות את תתי המערכות לרכיבי חומרה(Nodes)

תוצר פעילות התכון

א) 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.

ב) ארכיטקטורת התכנה(Software architecture):
מתארת את החלוקה של המערכת לתתי מערכות, ואת התלויות בין תתי המערכות ואת המיפוי של תתי המערכות לרכיבי החומרה.

ג) החלטות אסטרטגיות כמו:
- אופן אחסון המידע
- 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'.

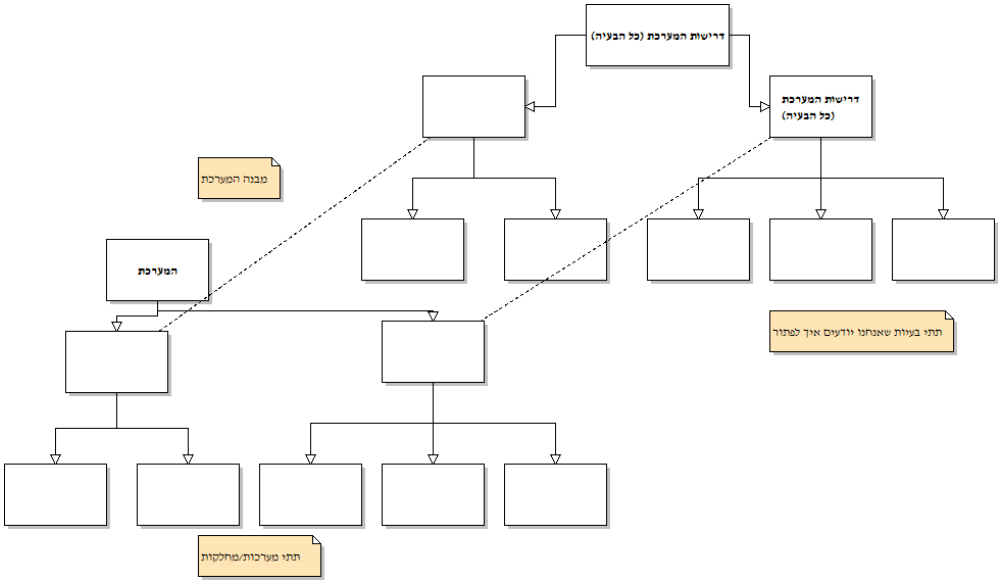
ד) Use Cases – עבור פעולות טכניות כמו: קונפיגורציה, אתחול, כיבוי ומקרים חריגים.

תתי הפעילויות של תהליך התכון

1. זיהוי יעדי התכון: המפתחים מזהים ומתעדפים את איכויות המערכת שיש למטב(Optimize).
2. 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.
3. 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.

מושגים בסיסיים בתיכון מערכות.

תתי מערכת ומחלקות



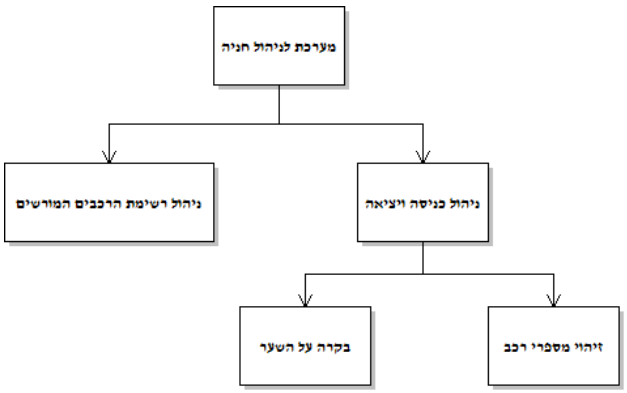
דרישות המערכת מתארים בעיה מורכבת אשר איננו יכולים לפתור מיד. ולכן נפרק את הדרישות לתתי בעיות. כל תת בעיה תפורק לתתי בעיות עד שנגיע לתתי בעיות שאנחנו יודעים לפתור. הפתרון של כל תת בעיה הוא תת ממערכת. כלומר נקבל שני עצים מקבילים: עץ הבעיות(הדרישות) שמחלק את דרישות המערכת לדרישות מתתי המערכות, ועץ המערכת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

ניקח לדוגמא את המערכת לניהול החניון, לאיזה תתי בעיות ניתן לפרק זאת?
הבעיה הראשית, היא תכנון החניון – הצורך לדעת להכניס רק רכבים מורשים לחניון, להיות מסוגלים לשנות את הרשימה הזו וכו'. איך ניתן להתחיל לפרק אותה לחלקים יותר קטנים?
לדוגמא:

- כיצד מזהים רכב מורשה
- ניהול בקרת רכבים מורשים
- כלל הניהול של כניסה ויציאה מהחניון.

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.

בין תתי המערכות קיימות לרוב תלויות הנובעות מכך שעליהן לתקשר זו עם זו.
למשל במערכת לניהול החנייה שהצגנו.
נניח ובנינו תת מערכת לכל אחת מהמערכות המוצגות למעלה(בקרה על השער, זיהוי מספרי רכב...).
נשאלת השאלה מי צריך לתקשר עם מי בתרשים שבנינו?
לדוגמא,
תת המערכת שאחראית על **כניסה ויציאה של רכבים לחניון** זקוקה למידע מתת המערכת ש**אחראית על ניהול הרכבים המורשים**:

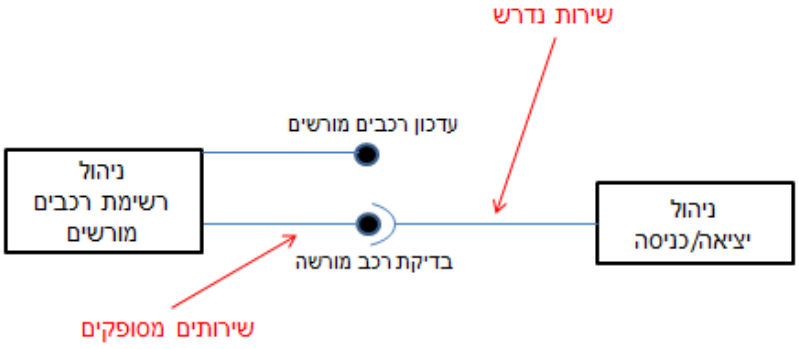
לסיכום, מחלקה היא הצורה הכי מינימלית איתה נעבוד, ומערכות יהיו מורכבות מתתי מערכות שמורכבות ממחלקות.

שירותים וממשקים של תתי מערכות
(Services and interfaces of subsystems)

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.

לדוגמא:



צימוד(Coupling) ולכידות(Cohesion)

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

א. 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'.

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.

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

טכניקות להורדת הצימוד

1. צמצום המידע המשותף לשתי תתי המערכות – ככל שנצמצם יותר את המידע, בכך נקטין יותר את הצימוד. בידיוק לפי העקרון 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 הכל, ורק יתן 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.
2. 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.
3. 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 מסויימת.

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.

לדוגמא, במסעדה.
קיים טיפול של הזמנת מקומות, וקיימת עוד אפשרות שהמערכת צריכה לעשות וזה ניהול תפריטים.
נשאלת השאלה עד כמה הנושא של ניהול תפריטים קשור להזמנת מקומות, התשובה לכך היא שאין קשר, ולכן הלכידות בין 2 המושגים האלה היא נמוכה. אם נבנה מערכות/תתי מערכות לשתי הפעולות האלו, לתת מערכת הזאת תהייה לכידות נמוכה שכן לגורמים בה אין ממש קשר.
נשאלת השאלה למה בכלל רע לעשות את זה? למה רע לשים אותם ביחד באותה תת מערכת?

התשובה לכך היא לדוגמא הזמן שייקח לנו להבין את המערכת כשנביט עליה, אבל מעבר לכך, ברגע שדברים יושבים ביחד, יש לנו נטייה לקשור ביניהם בקשרים שהם לא מין העניין. זאת אומרת שפתאום קיים לנו אותו הפתרון למספר בעיות שונות במערכות שונות. ובשפה המקצועית הקוד נקרא לספגטי, שכן שינוי כלשהו בקוד, משפיע על דברים שהוא לא היה אמור להשפיע עליהם.

לסיכום: תמיד נשאף לצימוד נמוך בין תתי המערכות, ולכידות גבוהה בכל תת מערכת.

מה הבעיה בתת מערכת שמכילה אובייקטים שיש להם תפקידים שונים(פונקציונליות שונה)?
(כלומר תת מערכת שיש לה לכידות נמוכה):

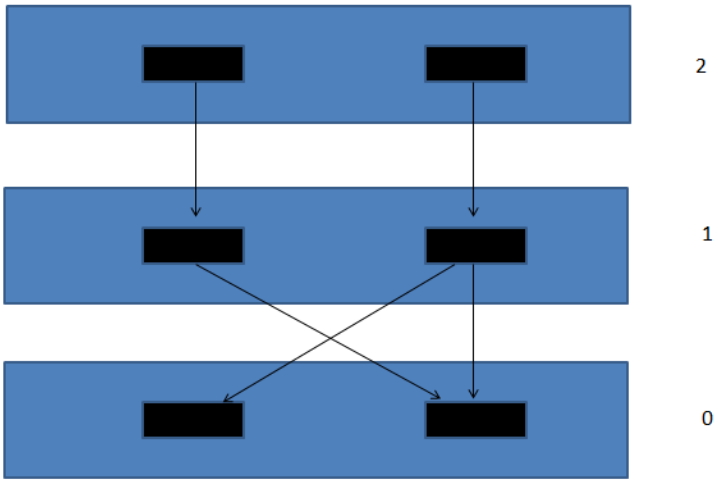
א. יש סכנה שיווצרו קשרים מקריים אשר יגרמו לכך ששינויים באובייקטים המייצגים פונקציונליות אחת יפריעו לאובייקטים שמייצגים פונקציונליות אחרת.
ב. אנחנו מפסידים הזדמנות לבנות תתי מערכות בעלות צימוד נמוך שכן חוסר קשר פונקציונלי בין אובייקטים משמעותו לרוב שניתן לפצל אותם בין תתי מערכות שונות.

כעקרון מנחה נשאף לפרק את המערכת לתתי מערכות בעלות לכידות גבוהה שהצימוד ביניהן נמוך.

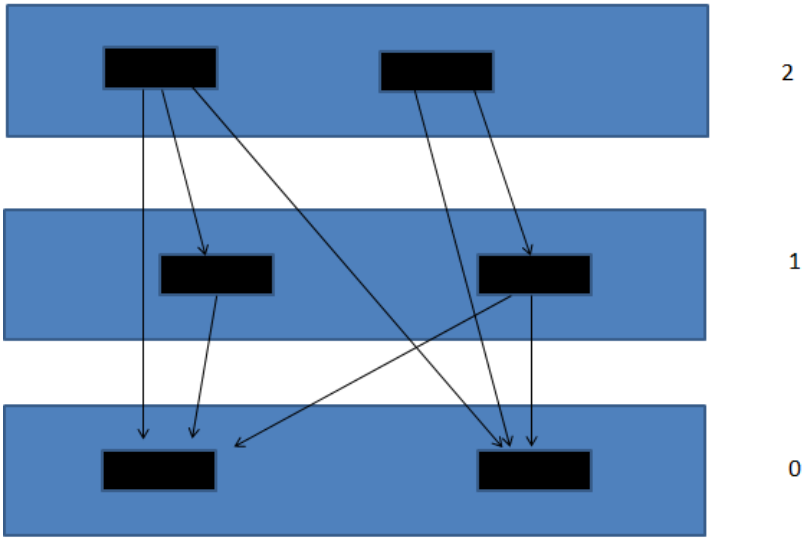
מודל השכבות

זהו מודל של קשרים בין תתי מערכות, שבו תתי המערכות מחולקות לשכבות.
השכבה 0 היא זו של תתי מערכות שלא תלויות באף תת מערכת אחרת. השכבה ה-i+1, מורכבות מתתי מערכות שתלויות בשכבה ה-i בלבד(עבור מודל סגור) ובשכבות 0-i (עבור מודל פתוח).

מבנה כללי של מודל שכבות סגור:



לעומת מודל שכבות פתוח:

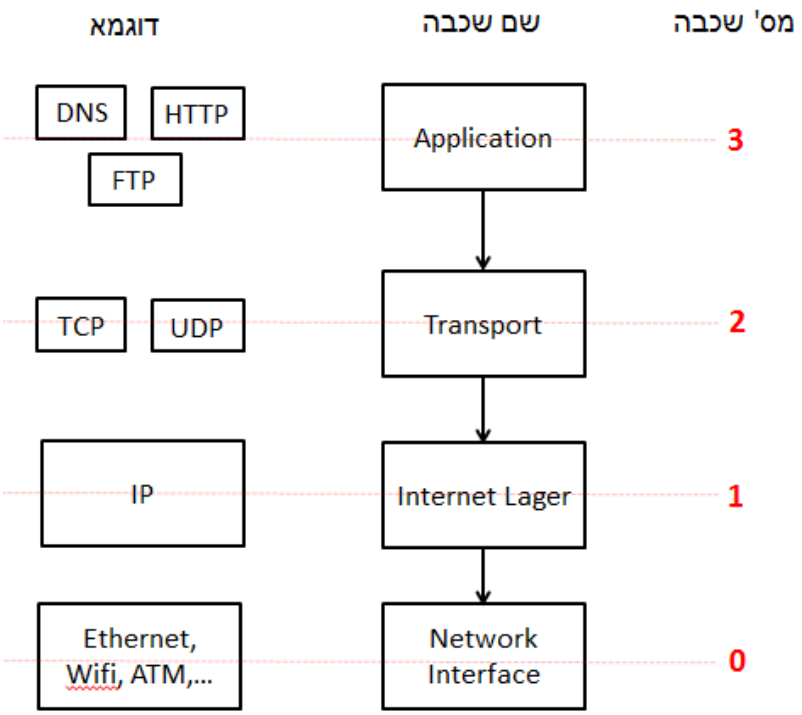


הבדלים בין מערכת סגורה/פתוחה במודל השכבת

במודל השכבות תתי המערכות מאורגנות במבנה היררכי ע"פ העיקרון הבא:

כל תתי המערכות שלא זקוקות לאף תת מערכת נמצאות בשכבה 0. תתי המערכות בשכבה i+1 יכולות לגשת(**במערכת סגורה**) אך ורק לשכבה ה-i.
ו**במערכת פתוחה**, לכל תתי המערכות בשכבות i,...,o

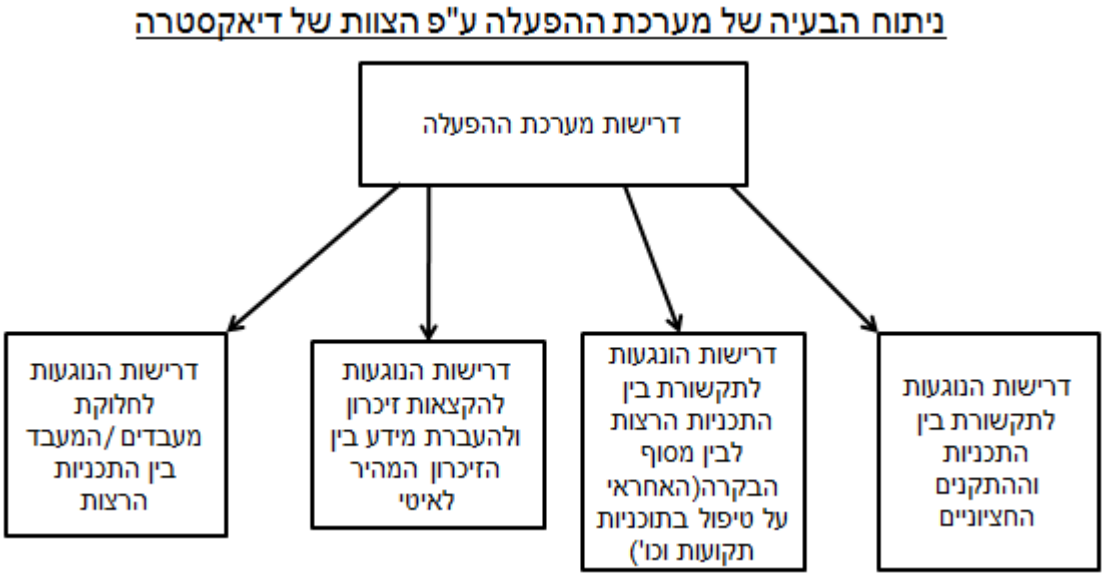
דוגמא למודל שכבות סגור:
פרוטוקול TCP/IP



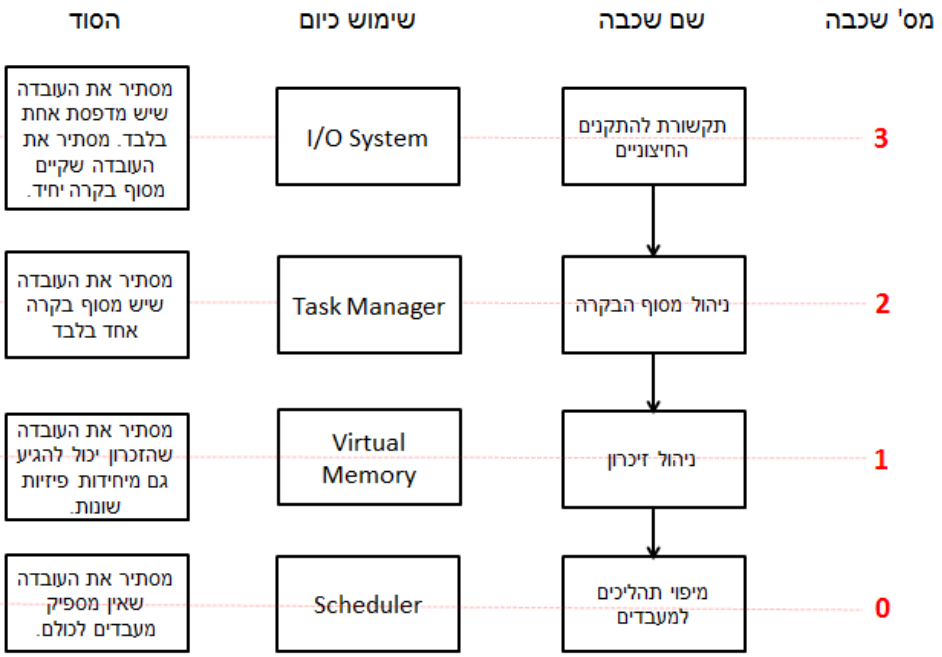
עקרון חשוב מאוד של תיכון:
כל שכבה מסתירה "סוד" מהשכבות שמעליה, ה-"סוד" מייצג משהו שיכול להשתנות.
בדוגמא הנ"ל, network interface – מסתיר את הגורם הפיזי, וכך גם שכבת ה-transport לדוגמא מסתירה את העניין של בקרה על שליחה תקינה.

דוגמא למודל שכבות פתוח

מערכת ההפעלה T.H.E, שפותחה ע"י צוות בראשותו של E. Dijkstra בסוף שנות השישים.



במהלך העבודה, הצוות גילה שע"י פתרון בעיות של חלקים מסויימים, הם יכולים לפשט את העבודה של התהליכים החלקים האחרים.
במידה ונצליח לפתור בעיה של "דרישות הנוגעות לחלוקת המעבד בין התוכניות", הפתרון הנ"ל(Process בדוגמא שלנו), יוכל לתת מענה לשאר הבעיות ולאופן פתרונן.
לאחר מכן, פותח הנושא הנ"ל למודל:



חשוב לראות שכלל השכבות עובדות בתצורה שבה הן צריכות שירותים מהשכבות שנמצאות מתחתיהן, הכלל הנ"ל משפיע לאורך כל תכנון המודל ולבסוף הגורם הישיר לאיך שהמודל נראה והשיקול העיקרי ל-איזה רכיב יהיה באיזה שכבה.

סגנונות ארכיטקטוניים

מתברר שהחלוקה של מערכות רבות נופלת לאחת ממספר לא רב של תבניות ללא קשר למטרה הספציפית של המערכת. תבניות כאלה נקראות תבניות ארכיטקטוניות או סגנונות ארכיטקטוניים. נסקור עתה כמה מהסגנונות החשובים ביותר.

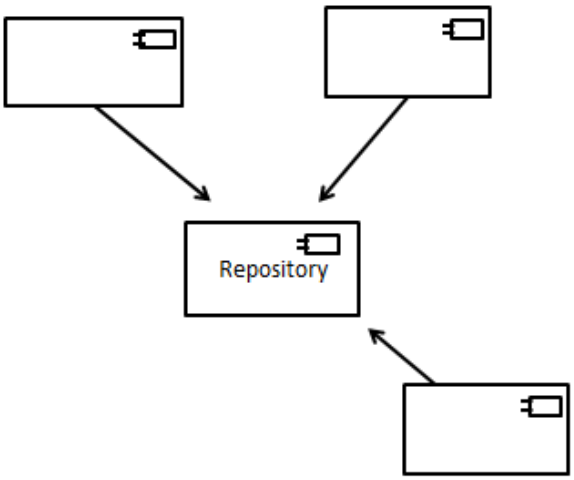
1. סגנון ה-Repository

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:

1.1 בסגנון ה-Database תתי המערכות קובעות באופן עצמאי מתי הן ניגשות לנתונים.

1.2 בסגנון ה-Black Board ה-Repository מפעיל את תתי המערכות בהתאם למידע שהוא מחזיק ולשינויים בו.

תיאור סכמטי של Repository



שאלה: מה אפשר לומר על הצימוד בסגנון זה?

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.

אם לקחת דוגמא, ניתן להסתכל על טבלאות של סטאטיסטיקה יומית/חודשית.
ויש לנו תת מערכת שלוקחת את השבועית והופכת אותה לחודשים, וכזאת שלוקחת מהיומית והופכת אותה לשבועית. אם זאת השבועית צריכה "לחכות" לסיום של המערכת שממירה את הטבלאות לשבועית, קיימת תלות לא רצויה בין המערכות.

2. סגנון ה-Model View Controller (MVC)

בסגנון זה נחלק את תתי המערכות לשלושה סוגים:

א. תתי מערכות מסוג **Model** מתחזקות את המידע שמייצג את עולם הבעיה.

ב. תתי מערכות מסוג **View** מציגות את המידע.

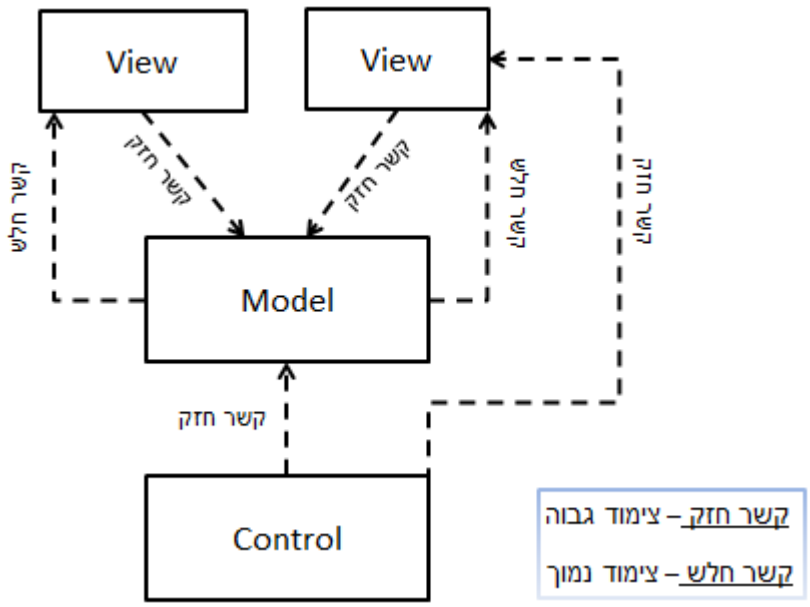
ג. תתי מערכות מסוג **Control** מנהלות את האינטרקציה(קלט) מול המשתמש.

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.

ניתן לראות זאת כך:



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.

ה- MVC הוא סוג של Repository שבו ה-Model הוא ה-Repository ומול ה-Views הבקרה היא בסגנון BlackBoard בעוד שמול ה-Controller הבקרה היא בסגנון Database.

3. 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**.

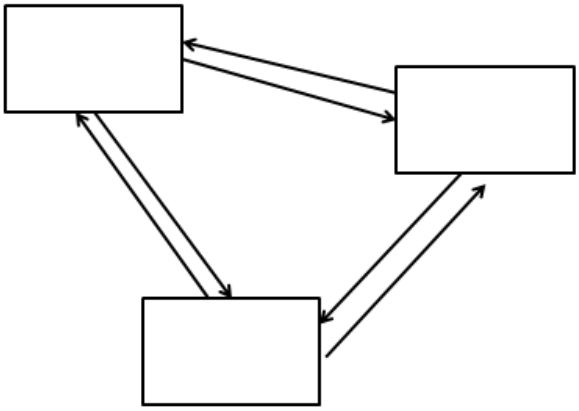
דוגמאות למערכות שעובדות בסגנון זה:

- Web Server
- מערכת הפעלה(בה ה-kernel הוא ה-Server, ואילו התהליכים הם ה-Clients).

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.

4. 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.



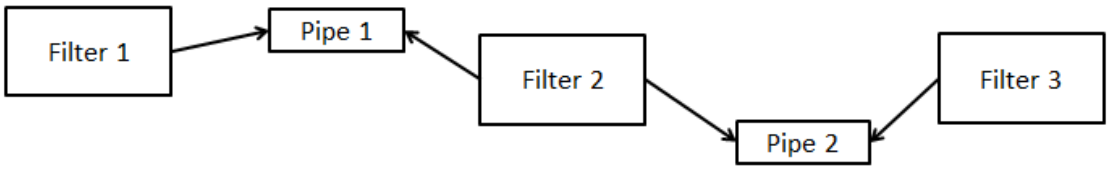
מכיוון שהמידע מבוזר בין ה-peers, יש למערכת יתירות(ניתן לגשת למידע גם כשחלק מה-peers לא זמינים). מצד שני, למערכת כזו יש 2 סכנות עיקריות:

<u>חסרונות אפשריים:</u>

- 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 קונסיסטנתי, שינויים ב-peer אחד צריכים לעבור לכל ה-peers, אחרת אותו מידע ב-peers שונים לא יהיה עקבי.

5. סגנון ה-Pipes and Filters

בסגנון זה תתי המערכות מתחלקות לשני סוגים:
filters מעבדים את המידע ו-pipes מחברים בין ה-filters.



נשים לב!

החצים בדוגמא זאת מייצגים **תלות של תתי מערכות**, ולא **זרימת מידע** ביניהן.

יתרונות:

צימוד נמוך
ניתן לראות שאין צימוד בין הפילטרים(מאוד מאוד חלש ביניהם), שכן אם נוציא את Filter 2, וה-Pipe`s הרלוונטים עדיין יוכלו להעביר את המידע בין Filter 1 ל-Filter 3, למעשה המערכת תמשיך לעבוד.

עבודה במקביל
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" מסויים.

חסרונות:

קיבוע המערכת – במצב כזה המערכת מאוד מקובעת, לא נצרכות פעולות מהמשתמש ולא ניתן להשפיע על אופן הפעולה של המערכת.

לסיכום:

בסגנון ה-pipes-and-filters אין צימוד בין ה-filters מכיוון שהם מבודדים ע"י ה-pipes.
כל filter מקבל מידע דרך ה-input pipes שלו ומייצר מידע לתוך ה-output pipes שלו. כמו filters מתחברים לשרשרת ע"י חיבור pipe ליציאה של filter אחד ולכניסה של ה-filter שבה אחריו. זרימת הבקרה מוכתב על ידי זרימת הנתונים.

כל filter עובד כאשר יש נתונים ב-input pipes שלו וכשאין נתונים או כשה-output pipes שלו מלאים הוא נח.
ניתן למקבל מערכת כזו בקלות מבלי להזדקק למנגנוני סנכרון מסובכים(כמו מנעולים) ע"י כך שנריץ כל filter ב-thread משלו.

פעילויות תיכון המערכת

1. זיהוי יעד התיכון

זהו הצעד הראשון בפעילות תיכון המערכת. מטרתו לזהות ולהתמקד באיכויות החשובות למערכת. יעדים רבים ניתן להסיק מהדרישות הלא פונקציונליות. אחרים יש להבין מהלקוח בכל מקרה יש לתדע את היעדים באופן מפורש כדי שהחלטות התיכון יתבצעו מתוך מודעות ליעדים.

נחלק את האיכויות(התכונות הרצויות) לחמש תתי קבוצות:

- ביצועים (performance)
- אמינות (dependability)
- עלות (cost)
- תחזוקה (maintenance)

מוכתב ע"י הלקוח

- שימושיות (usability)

כעת נפרט על האיכויות הנ"ל:

<u>קריטריונים של ביצועים(Performance criteria):</u>

Response time – How soon the system responds?

Name	Meaning
Response Time	How soon the system responds?
Throughput	How many requests the system processes in a given period of time
Memory usage	