

1. מערכת \ מודל \ הפשטה:

- **מערכת –** אוסף ישויות ממשיות או מופשטות, המקיימות קשרי גומלין או תלות הדדית ויחדיו יוצרות שלם המשרת מטרה משותפת. למערכת תמיד יש מטרה (אחת או יותר)!
- **מודל –** מודל הוא הבסיס לפיתוח תוכנה. ייצוג מופשט של תופעה או רעיון, של אילוצים, חישובים ותהליכים.
- **הפשטה –** עיקר, תמצית. ניתן להגיד שכל מערכת בנויה מהפשטה, בעלת גבולות, מבנה (OOP למשל), יחסים פונקציונליים – מקבלת קלט ומחזירה פלט (יש לכל מערכת התנהגות), ובעלת מטרה.
- **מערכת סגורה –** מערכת שאין לה קשר עם הסביבה החיצונית, היא רק בתיאוריה.

2. הנדסה לעומת אמנות כתיב יצירה:

הנדסה ואומנות עוסקות ביצירה, אז מה למעשה ההבדל?
הנדסה מתבססת על דיסציפלינות "ידועות" (כמו חוקי הפיזיקה לדוג'), בעוד אומנות אינה בהכרח מתבססת על דיסציפלינות כאלה ואחרות.
בתהליך היצירה ההנדסי אנו מנסים לצמצם את השונות- זאת אומרת לעשות את הדברים כמה שיותר כללים ע"פ כללי אצבע כאלה ואחרים, לדוג- חברת פייסבוק מפתחת הרבה מוצרים (הנדסיים) פייסבוק שואפת שהשונות בין תהליכי הפיתוח יהיו מינימליים כדי שיוכלו בסופו של דבר לסנכרן את כל התהליכים.
לעומת זאת באומנות ישנה שאיפה למגוון רב ולמעשה יש עידוד של שונות.

3. מאפייני הנדסת מוצרי תוכנה ואתגרים אופייניים:

- ראשית נפרט לגבי מוצר תוכנה, מוצר תוכנה מסופק ב-3 תצורות:
- **מוצר עצמאי:** מגיע בדיסק קשיח או בהורדה מהאינטרנט, מיועד לשימוש על מחשב המשתמש ומספק מערכת הפעלה ותוכנה תשתיתית. דוגמא: מערכת Office.
 - **מוצר משובץ:** (embedded) מגיע ביחד עם החומרה המתאימה לו ספציפית, יש בו מערכת הפעלה ותוכנה תשתיתית כמוצר שלם. דוגמא: טלפון סלולרי.
 - **תוכנה כשירות (SaaS = Software as a Service):** התוכנה עצמה אינה מגיעה ללקוח ואינה נשארת ברשותו, אלא הוא מקבל שירותים הדרושים באמצעותה. דוגמא: Gmail, Dropbox.

מאפיינים:

- מורכבות הדורשת עבודת צוות ממספר דיסציפלינות (ידע הנלמד/נחקר באקדמיה).
- אינטראקציה ותקשורת בין מגוון בעלי מקצוע.
- דינמיות וחדשנות בכל אספקט אפשרי.

אתגרים אופייניים:

- **גורם אנושי –** חוסר תקשורת, אינטרסים מנוגדים, אי שימוש בסטנדרטים הנקבעים מראש, חוסר מקצועיות.
- **טבעו של עולם –** לחצי זמנים ותקציב, דרישות משתנות ואי יציבות בהגדרת הצרכים.

פתרונות:

- קבלת המציאות והתאמה פרקטית אליה.
- שילוב שיטות תכנות אדפטיביות וכלים תומכים.

4. סוגי פעילויות בהנדסת תוכנה

- **ייזום –** זיהוי בעיה והזדמנות (תכלס לחשוב על רעיון חדש).
- **הגדרת דרישות –** תיאור מדויק של הנדרש (למימוש הרעיון).
- **ניתוח –** כיצד ניתן לפתור את הבעיה (התעמקות).
- **עיצוב –** בחירה ותכנון הפתרון המתאים.
- **מימוש –** תרגום התכנית למציאות.
- **בדיקות –** בדיקות התוצאות מול התכנון.
- **הטמעה –** התקנת המערכת והטמעה אצל המשתמשים.
- **תחזוקה –** תהליך מתמשך של ניפוי שגיאות והרחבות (תחזוקה שוטפת של המוצר).
- **גם:** דיונים, הערכות סיכונים, תיעוד, תיעוד, ארכיטקטורה ועוד.. (דברים שפשוט עושים בתחזוקה, שזה בתכלס מה שחברות עושות ביום יום כנראה).

5. סוגי תפקידים בהנדסת תוכנה

בעלי עניין:

- אדם בעל אינטרס חיובי או שלילי היכול להשפיע על התפתחות הפרויקט.
- גורם שמעורב "מאחורי הקלעים" בצורה אקטיבית לדוגמא: לקוח, משתמש, משקיע, ספק, רגולטור.

מנתח מערכות:

- מנתח את המצב הקיים (מרחב הבעיה) ומבין את הבעיות הדרושות.
- מגדיר אפיון למרחב הבעיה וקווי מתאר לפתרון.
- תוחם ומחדד את מטרות המערכת ומגדיר את דרישותיה.
- אוסף ומסווג דרישות מלקוחות ומשתמשים.
- תוצרים פורמליים - מסמך ייזום, ניתוח מצב קיים, הגדרת דרישות, אפיון פתרון, דיאגרמות.

מנהל פרויקט:

- אחראי לעמידה בביצועים הנדרשים, בתקציב ובלו"ז.
- בקיא בתהליכי ניהול פרויקט מקובלים, מתאים את התהליכים לפרויקט הספציפי ודואג ליישומם.
- נדרש ליכולות ניהוליות – יודע לארגן, לחשוב ולהניע אנשים.

מנהל מוצר (PO):

- דואג לאינטרסים של הלקוח.
- קולו של המשתמש בצוות הפיתוח – מסתכל מהצד של המשתמש.
- מביא את המוצר הנכון למשתמשים – מעיין רפרנס בין הלקוח למפתח.
- קובע חזון לאחר חקר שוק, הבנת הלקוח וצרכיו והגדרת אופי הבעיה.
- מגדיר תוכנית פעולה ומפת דרכים לפיתוח איטרטיבי לטובת מימוש מלא של החזון.
- עובד בצמוד לצוות הפיתוח - מפקח על התכנון, על הקצב ועל פתרון הבעיות.
- כאשר הגרסה יוצאת תפקידו לעקוב ולקבל פידבקים מהמשתמשים.
- בעל תפקיד אסטרטגי – קובע את הכיוון, לאן החברה הולכת.

מהנדס מערכת:

- אחראי טכנולוגית על הפיתוח.
- קובע את מיפוי הטכנולוגיות שבהם המערכת תשתמש.
- בעל ראייה מערכתית – בקיא ומכיר את כל התהליכים והקשרים ביניהם.
- יודע להסתכל על המערכת גם מבחינת חומרה וגם מבחינת תכנות.
- משולב בפרויקטים הנדסיים רחבי היקף בכל שלביהם.

ארכיטקט תוכנה:

- מי שמעצב ומייצר את המערכת.
- אחראי על High level design.
- בוחר את תשתיות התוכנה.
- אחראי על חלוקת יישום מורכב למודולים.
- מגדיר את כללי העיצוב והפיתוח (איך מוסיפים רכיב חדש? איך מאפיינים רכיב חדש?) ומעביר אותם למפתחים.
- פונקציונאלי – מבין מה כל חלק עושה ואת הקשרים והתלויות בין כל חלקי המערכת.

ראש צוות:

- אחראי על ה Low level design.
- בונה את הצוות – בוחר את האנשים.
- מנהל מס' אנשים בעלי תפקיד זהה.
- תיאום עבודת הצוות.
- קביעת נהלים, ביצוע code review.
- מתואם עם גורמי ניהול הפרויקט.

מטמיע מערכת:

- אחראי על כתיבת מדריכים טכניים למשתמש.
- מתקין את המערכות במחשבים השונים בחברה (משך זמן ההטמעה תלוי בגודל המערכת).
- נמצא בבקרה מתמדת על פעילות המערכות.
- מעניק שירותי תמיכה וליווי שוטפים לעובדים בתפעול התוכנות השונות.

מומחה יישום:

- נציג מטעם הלקוח אשר מהווה משתמש עיקרי של המערכת אשר נבחר כבר בשלב הייזום.
- אחראי ללוות את המערכת לאורך כל השלבים, לארגן את דרישות המערכת וביניהן בדיקות קבלה ומסירה.
- מאשר את קבלת המערכת.

6. סוגי ארטיפקטים בהנדסת תוכנה

ארטיפקטים, אלו תוצרי הלוואי של פיתוח התוכנה.
לדוגמה: דיאגרמות UML, הערכת סיכונים, Use Case, קוד המערכת.

7. Stakeholders ופוטנציאל מעורבותם:

Stakeholders - אדם בעל עניין או אינטרס שנמצא בתוך המערכת והרבה פעמים משנה את כל קבלת ההחלטות בשלב תכנון ויצירת התוכנה, מה שיכול להשפיע על ביצוע הפרויקט והשלמתו.

8. סוגי תהליכים ומאפייניהם (לינארי, אינקרמנטלי, ניהול סיכונים):

1. ללא תהליך:

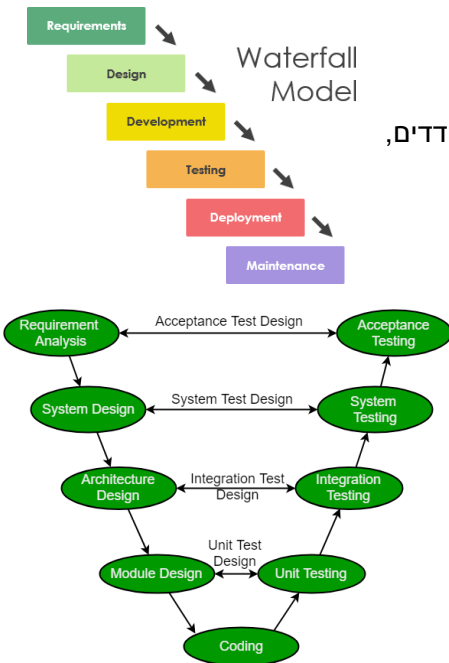
- **קודד ותקן (code & fix)** - מתודולוגיה ששמה דגש על מהירות, מציאת הבעיות תוך כדי ניסיון התיקון בדרך של ניסיון וטעייה. יתרון: פיתוח מהיר על פי דרישות הלקוח. חסרון: תהליך בלתי מתוכנן וקשה לתחזוקה.

2. תהליך מתוכנן:

- **מפל המים** - מתודולוגיה פיתוח שדוגלת במחזור פיתוח אחד. לפני שמתחילים יש ליצור פורמליזציה נרחבת של כל הדרישות. שמם דגש רב על עיצוב מוקדם של התוכנה לפני שמקודדים, ורק בסוף יוצרים בדיקות. יתרון: תהליך מתועד היטב, תחזוקה קלה, מסודר. חסרון: צריך להיצמד לתכנון הראשוני ואין מקום לשינויים.

- **מודל V** - שם דגש על קשר הדוק בין רמת ההגדרה והפירוט לבין רמת המימוש והבדיקה. מאפייני התהליך: התהליך מגדיר 4 רמות פיתוח, ולכל אחת בדיקות מתאימות כגון:
 1. בדיקת יחידה - כדי להיפטר מבאגים בקוד.
 2. בדיקות אינטגרציה - לוודא כי היחידות שנוצרו באופן עצמאי מתקשרות.
 3. בדיקות מערכת - מבטיחה כי הציפיות של הלקוח מהיישום מתקיימות.
 4. בדיקות קבלת משתמשים - מוודא שמערכת שהועברה עומדת בדרישת המשתמש.

יתרונות: (1) חסכון בכסף ובמשאבים עקב תחילת תהליך הבדיקות בשלב מוקדם של הפרויקט. (2) כיסוי מקסימלי של בדיקות. חסרונות: (1) התארכות זמן הפיתוח. (2) חוסר ודאות. (3) סיכון גבוה. (4) אין גמישות לשינויים במהלך הסבב.



- **אב טיפוס** - דגם של המוצר. משמש להדגמה בלבד, הוא לא המוצר הסופי!

המפתחים בונים אב טיפוס בשלב ניתוח הדרישות. משתמשי קצה מספקים משוב לתיקונים ולפי זה המפתחים משפרים את אב הטיפוס.

כאשר משתמשי הקצה מרוצים, משלימים את השאר כדי להפוך אותו למוצר סופי.

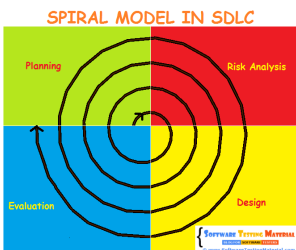
שימושי במיוחד כאשר דרישות המשתמש לא מלאות ונושאים טכניים לא לגמרי ברורים.

יתרונות: (1) הלקוחות יכולים לראות תרגום הדרישות למשהו מוחשי עוד בשלב ניתוח הדרישות. (2) המפתחים לומדים ממש

המשתמשים. (3) איתור דרישות ופונקציונאליות שלא היו צפויות. (4) מאפשר עיצוב גמיש. (5) קל לראות התקדמות.

חסרונות: (1) ביצועים לא יעילים. (2) קיימת נטייה לאובדן מבנה התהליך. (3) ארכיטקטורה וראייה מערכתית אינם זוכים למקום מרכזי.

(4) מתמקדים בריצוי מיידי ודרישות תחזוקה נדחקות לשוליים. (5) קיימת סכנה לתהליך שלא נגמר (Creep Scope).



3. תהליך איטרטיבי / אינקרמנטלי:

- **ספירלי** - פיתוח המערכת בגישה הספירלית נעשה בכמה סבבים (כל סבב יכול להימשך חודשים רבים בין שישה חודשים לשנתיים). הפיתוח מחולק לארבעה שלבים:
 - הגדרת מטרות, חלופות ואילוצים.
 - ניתוח חלופות, זיהוי סיכונים ומציאת פתרונות.
 - פיתוח והטמעה (שחרור גרסה).
 - תכנון הסבב הבא.

יתרונות: (1) ניתוח הסיכונים שקיימים נעשים תוך כדי הפיתוח. (2) הפיתוח מתחיל מוקדם בתהליך.

חסרונות: (1) מתאים לפרויקטים גדולים כיוון שעלות הניתוח גבוהה. (2) הצלחת הפרויקט תלויה בעיקר בשלב ניתוח הסיכונים.

(3) לא מתאים לפרויקטים קטנים.

4. תהליך מסתגל:



- **Scrum** - מפורט למטה
- **Agile** - מפורט למטה
- **Kanban** - מודל מבוסס על שיטת agile הנותן דגש לצד הוויזואלי של התהליך- מה לייצר, מתי לייצר וכמה לייצר. השיטה מעודדת ביצוע שינויים קטנים והדרגתיים במערכת. התהליך מגביר את הגמישות לשינויים, מצמצם את בזבזי זמן, קל להבנה ומקצר את זמני המסירה של המשימות, אך כל הזמן צריך לעדכן את הלוח ואין בלוח מידע לגבי זמני המסירה.
- **XP** - מודל המבוסס על שיטת agile ומאפשר מעורבות גבוהה של הלקוח בתהליך הפיתוח. כאשר התוכנה תענה על צרכי הלקוח בצורה מיטבית. XP משתמשת בפרקטיקות הבאות:
 1. **Programming Pair**: זוג מתכנתים על אותו מסך מחשב, כותבים יחד את הקוד.
 2. **Design Simple**: קוד פשוט. ללא קוד כפול, עם קוד מובן למתכנתים, מכילה את המינימום הנדרש של מחלקות ושיטות.
 3. **Refactoring**: שיפור קוד קיים על ידי שימוש בטכניקות שנועדו לשפר את המבנה הפנימי של הקוד מבלי לשנות את ההתנהגות החיצונית שלו.
 4. **TDD**: בדיקת יחידה הנכתבת בטרם נכתב הקוד אותו היא בודקת.

9. הנדסת דרישות (סיווג דרישות, תעדוף)

דרישה איכותית צריכה להיות בדידה, ברורה (מנוסחת בשפת הלקוח), חד משמעית, שלמה, נצרכת, לא סותרת דרישות אחרות, ניתנת לבדיקה באמצעות טסטים, ניתנת למעקב, מתועדת.

דרישות משתמש - משפטים בשפה טבעית המלווים בדיאגרמות, השירותים תספק ומסגרת אילוצים – התיאור הוא עבור משתמשים. דוג: על המערכת לספק דרך לקרוא מסמכים שנוצרו במערכות אחרות.

דרישות מערכת - מסמך מובנה הפורס תיאור מפורט של פונקציות המערכת, שירותיה ואילוצי התפעול, מתאר מה יש לממש באופן המאפשר להשתמש בו כבסיס לחוזה \ פיתוח \ בקרה \ בדיקות. דוג: המערכת תאפשר למשתמש לבחור ולקשר בין סוג מסמך לעורך ספציפי. לכל סוג מסמך יופיע סמל אחר והמשתמש יוכל לקבוע מהו הסמל.

דרישות פונקציונאליות - משפטים אודות השירותים שהמערכת תספק, כיצד המערכת תגיב לסוגי קלטים ולמצבים נתונים ומהם הפלטים. בשורה התחתונה, מה המערכת עושה ולא איך היא עושה זאת. דוג: על המערכת לספק דרך להזמין פיצה טעימה במיוחד.

דרישות לא פונקציונאליות - אילוצים על השירותים או הפונקציות שהמערכת מספקת כגון חלונות זמן לתגובה, נפחים מקסימליים, מספר משתמשים בו זמני. אילוצים כמו אבטחה וכו'. דוג: המערכת תתבסס על מחשב מסוג

דרישה תפעולית (OR = Operational Requirement) - דרישה המתייחסת לתפעול, לאינטראקציה או להתנהגות של המוצר. דוג: המערכת תציג תמונה של המשתמש.

דרישת מידע (DR = Data Requirement)

- דרישה המתייחסת לישויות המידע ולנתונים בהן נדרשת התוכנה לטפל (לקלוט, לאחסן, לאחזר, לעבד, להפיק כפלט).
- לדוגמא: נתונים ומבני נתונים, מאגרי מידע/ בסיסי נתונים, דרישות קלט/פלט.

תעדוף דרישות:

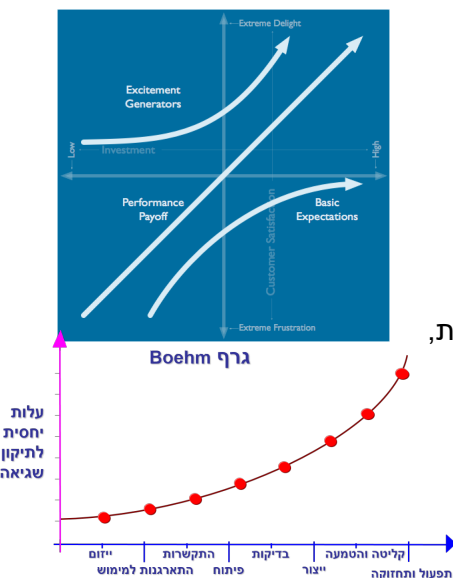
מודל Kano: הרעיון העיקרי הוא שהדרישות לאו דווקא מתועדפות ע"פ שביעות רצון של הלקוחות. מערכת צירים של מודל Kano:

10. Separation Of Concerns/Coupling/Cohesion

עקרון עיצובי שבו כל יחידה (לדוג' יחידת קוד) היא בעלת תפקיד מוגדר ויחודי. תוכנית המגלמת היטב את עקרון ה SoC נקראת תוכנית מודולרית. **ניקפדה**: עיקרון עיצובי להפרדה של תוכנית מחשב למקטעים נפרדים. כל סעיף מתייחס לדאגה נפרדת, אוסף מידע המשפיע על הקוד של תוכנית מחשב. דאגה יכולה להיות כללית כמו "פרטי החומרה של אפליקציה", או ספציפית כמו "השם של איזו מחלקה להפעיל". מודולריות מושגת על ידי קיבולת מידע בתוך קטע קוד שיש לו ממשק מוגדר היטב.

11. חוקי עקומת בוהם, חשיבות שלבים (ניתוח, עיצוב, בדיקות, תחזוקה)

עקרון בוהם - איתור שגיאה בשלב מאוחר תגרוור עלות תיקון גדולה יותר.



Unified Modeling Language או בקיצור UML היא שפת מפרט תקנית לעיצוב מונחה-עצמים. התיווי בשפה הוא גרפי ומאפשר תיאור מופשט של מפרטי המערכת, בדרגות שונות של דיוק. בשל אופייה הוויזואלי, UML היא שפה קלה-יחסית ללימוד, בהשוואה לשפות מפרט אחרות. לרוב, כל בעלי התפקידים בצוות פיתוח תוכנה מכירים את הדיאגרמות העיקריות בשפה, והדבר מסייע לקשר את מפרטי התוכנה ביניהם. ב-UML קיימים מספר סוגי תרשימים (דיאגרמות), המשמשים למטרות שונות, ומתחלקים לסוגים שונים. התרשימים השונים מורכבים משלוש אבני בניין עיקריות, גורמי יסוד, יחסים בין גורמי היסוד ודיאגרמות - התרשימים שמאגדים את גורמי היסוד והיחסים לגורמים בעלי משמעות.

1. גורמי יסוד - גורמי היסוד (elements) מתחלקים לארבע קבוצות עיקריות:
 - גורמים מבניים (structural): אלה הם "שמות עצם" המרכיבים כל מודל ואשר בדרך כלל מתארים גורמים פיזיים או קונספטואליים. לדוגמה שחקן (Actor), שמסומן על ידי דמות אדם.
 - גורמים התנהגותיים (behavioral): אלה הם החלקים הדינמיים של מודל UML אשר מוגדרים כ"פעלים" של המודל. גורמים אלו מגדירים את ההתנהגות של גורם מסוים בזמן, או ביחס לגורמים אחרים במודל.
 - גורמים מקבצים (grouping): אלה הם החלקים שמאפשרים ארגון טוב יותר של המודל. הגורם הנפוץ ביותר השייך לקבוצה הוא package. גורם זה הוא מעין ריבוע שמאפשר לארגן בתוכו גורמים מסוגים שונים לקבוצה אחת שאיתה ניתן לקבץ גורמים התנהגותיים, מבניים ואף גורמים מקבצים נוספים. בין הגורמים השייכים לקבוצה זו: מודל, מערכת (framework), תת-מערכת (sub-system).
 - גורמים מפרשים (annotational): גורמים אשר בעזרתם ניתן להסביר ולפרש גורמים במודל או מודל UML. הגורם העיקרי בקבוצה זו הוא ההערה (note). בהערה המשויכת לגורם, ניתן לרשום הערות בטקסט חופשי או אילוצים (constraints) המתייחסים לגורם ומגדירים עליו אי אלו הגבלות.

2. יחסים - היחסים (relationships) מגדירים את היחסים והקשרים בין האלמנטים. אף הם מתחלקים ל-4 סוגים:
 - תלות (dependency): יחס סמנטי בין שני גורמים. משמעותו היא ששינוי כלשהו בעצם הבלתי תלוי, עשוי להשפיע על העצם התלוי בו. תלות מסומנת באמצעות חץ מקווקו.
 - חיבור (association): מתורגם לקשר בין שני גורמים. מסומן באמצעות חץ פשוט.
 - הכללה (generalization): באופן אינטואיטיבי ניתן לחשוב על יחס זה כעל "הורשה" של מחלקות, כאשר האב הוא הגורם המכליל (generalized), והבן הוא הגורם "מיוחד" (specialized). מסומן באמצעות חץ עם משולש בקצהו.
 - מימוש (realization): זהו קשר סמנטי, שמשמעותו גורם אשר פעולתו מבוצעת על ידי גורם אחר. בדרך כלל משתמשים ביחס זה בין ממשק (interface) לבין המחלקה המממשת אותו. מסומן באמצעות חץ מקווקו עם משולש בקצהו.
 - Composition: יחס הכלה חזק - אובייקט שלא יכול להתקיים בלי אובייקט אחר. יש ביניהם יחס הכלה חזק. מסומן עם מעוין מלא.
 - Aggregation: יחס הכלה חלש - כמו מקודם רק שכן יכול להתקיים גם לבד. מסומן עם מעוין ריק (לדוגמה, איש וקבוצה של אנשים).

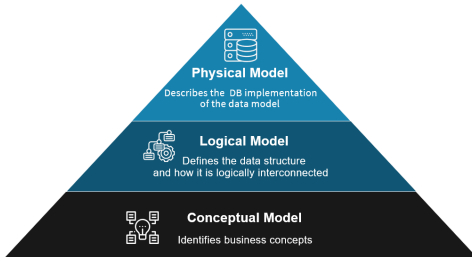
3. דיאגרמות - הדיאגרמות מאגדות את גורמי היסוד והיחסים לגורמים בעלי משמעות.

- דיאגרמות מבניות:
 - דיאגרמת מחלקה (Class diagram) - מתארת את מחלקות התוכנה ואת היחסים ביניהן.
 - דיאגרמת רכיבים (Component diagram) - מתארת את רכיבי המערכת כדוגמת טבלאות וקבצים.
 - Composite structure diagram
 - דיאגרמת פריסה (Deployment diagram) - מתארת את פריסת המערכת באופן פיזי אצל הלקוח.
 - דיאגרמת אובייקט (Object diagram) - תרשים המציג תצוגה מלאה או חלקית של המבנה של מערכת ממודלת בזמן מסוים.
 - דיאגרמת חבילה (Package diagram)
 - Profile diagram
- דיאגרמות התנהגותיות:
 - דיאגרמת פעילות (Activity diagram) - מתארת תהליכים המתרחשים בתוכנה בתגובה לפעולה של המשתמש.
 - דיאגרמת מצב (State diagram) - מתארת את העצם במצבים שונים.
 - דיאגרמת אופן שימוש (Use case diagram) - מתארת את השימוש במערכת.
- דיאגרמות אינטראקציה:
 - דיאגרמת שיתוף פעולה (Collaboration diagram) - דיאגרמה זו חלופית לדיאגרמת רצף.
 - Communication diagram
 - דיאגרמת סקירת אינטראקציה (Interaction overview diagram)
 - דיאגרמת רצף (Sequence diagram) - מתארת קשר בין עצמי המערכת על ציר הזמן.
 - דיאגרמת תזמון (Timing diagram)

13. Scope Creep

ההגדרה של Scope Creep היא שכאשר היקף הפרויקט משתנה, עבודת הפרויקט מתחילה להתרחב, או "לזחול", מעבר למה שהוסכם במקור. זה יכול להתרחש כאשר היקף הפרויקט אינו מוגדר, מתועד או מבוקר כהלכה. זה נחשב בדרך כלל כמזיק. בנוסף Scope Creep גורמת לרוב לחריגת עלויות. זה דומה אך שונה מ-feature creep, מכיוון ש-feature creep מתייחס לפיצ'רים, ו-scope creep מתייחס לכל הפרויקט. הסיבות השכיחות ביותר ל-Scope creep בפרויקטים הן: היקף פרויקט המוגדר בצורה גרועה, חוסר בשיטות לניהול פרויקטים, הוספה של פיצ'רים מיותרים, פערי תקשורת בין בעלי העניין בפרויקט.

14. שלושת רמות מידול נתונים ומה מייצג תרשים ER?



מודל קונספטואלי -

- תיאור סמנטי "טהור" של תחום הבעיה (משמעות).
- תיאור יישויות (Entities) תכונות (Attributes) וקשרים (Relationships).

מודל לוגי -

- ייצוג תחום הבעיה במושגי טכנולוגיות, ניהול הנתונים (לדוג: טבלאות, רשומות ERD).

מודל פיסי:

- כיצד הנתונים נשמרים, במסגרת מאפייני ניהול RDBMS (מסד נתונים רלציוני כמו SQL).

הרעיון בכללי הוא שהמודל הקונספטואלי מהווה בסיס למודל שעליו חשבנו, לאחר מכן אנו מארגנים את הרעיון בצורה מסודרת בטבלאות או רשומות (מודל לוגי) לבסוף שומרים את כל הנתונים במודל פיזי כלשהו כמו SQL.

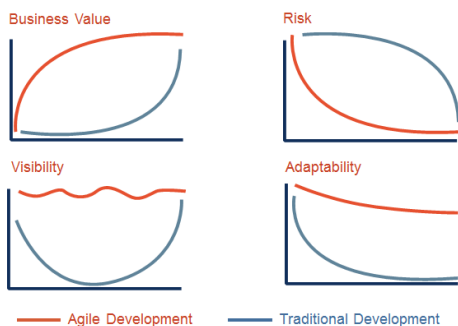
מודל ישות קשרים (Entity Relationship – ER):

מדובר על הגדרת הקשרים בין ישויות מסוימות במערכת. זאת אומרת לאחר שהגדרנו את הישויות במערכת שלנו נרצה להגדיר את הקשרים ביניהם. אפשר לעשות זאת באמצעות תרשימים כמו ERD.

ויקיפדיה: דרך להציג מידע. הצגה כזאת מאפשרת תכנון מלמעלה-למטה של מערכות מסדי נתונים רלציוניות. המודל אינו מתחשב בארכיטקטורת המחשב עליו יורץ מסד הנתונים אלא רק במבנה הלוגי הרצוי של מסד הנתונים באופן שיאפשר נוחות ויעילות בגישה למידע.

סוגי קשרים: (רק דוגמאות)

1. יחיד ליחיד: כיתה -> מורה מחנכת, מורה מחנכת -> כיתה (לכל כיתה יש מורה מחנכת אחת ולכל מורה מחנכת יש כיתה אחת שהיא המורה המחנכת שלה).
2. יחיד לרבים: כיתה -> תלמידים, בכיתה (יחיד) יש הרבה תלמידים (רבים).
3. רבים לרבים: מורה -> כיתות בהן הוא מלמד, כיתה -> המורים שמלמדים את הכיתה. לכל מורה יש מספר כיתות שבהן הוא מלמד לכל כיתה יש מספר מורים שמלמדים אותה.



15. אגיליות: פילוסופיה, חשיבות, סוגים, כלים, תפקידים, פרקטיקות

מהי שיטת ה-agile?

שיטה ששמה דגש מיוחד על ניהול פרויקט באופן שיאפשר יכולת תגובה מתמדת לשינוי, יעילות, זריזות ואיכות התוכנה הן מבחינת המענה שניתן לצרכי הלקוח והן מבחינת העדר תקלות ובאגים. הגישה מאמינה כי אי אפשר להגדיר או לחזות מראש את כל הצרכים של תוכנה לפני תהליך הפיתוח בפועל. הגישה מתאימה בעיקר לפרויקטים בהם ישנה חוסר ודאות לגבי המוצר הסופי, מאחר והיא משלבת תהליכי למידה ושיפור מתמידים, תוך כדי התקדמות הפרויקט.

Agility - שילוב של גמישות וזריזות – השארת הקוד פשוט, עבודה על חלקים קטנים שתמיד עוברים תיקוף של הלקוח ובדיקות – מצמצמים אי ודאות ולהבטיח איכות.

Agility מתבטא גם באיכות הצוות להיות גמיש זריז ובעל תקשורת בריאה וטובה – מעיין צוות רב זרועי, כלי אחד המכיל הכל כמו אולר שוויצרי וזו הסביבה והצוות שנרצה לייצר. נצטרך פתרון מבוסס agility כיוון שהוא פותר ומונע את הסיבות לכישלון.

עקרונות אגיליים:

- הפצה ממושכת ומוקדמת ככל האפשר לתוכנה בעלת ערך.
- הפצת גרסאות תכופה.
- תוכנה עובדת זה העקרון הבסיסי ביותר.
- תקשורת ישירה ללא מחסומים.
- צוות המורכב מאנשים מוכשרים ובעלי מוטיבציה.
- סומכים על צוות הפיתוח.

עיקרי מושגים ונושאים

- עבודה בקצב סביר, לא להגיע למצב של לחץ והתפוצצות.
- מתן סמכויות לצוות ולהתארגנות עצמאית.
- לא לפחד שהצוות נכשל – מהכישלונות נבנים ומתפתחים (בהנחה שהכישלון לא נובע מטיפשות ורשלנות).
- פשטות.
- חתירה למצוינות.

תהליך אג'ילי:

- תהליך איטרטיבי – איסוף אינפורמציה ע"י סקר ולמידת השוק, הערכת סיכונים, איסוף וארגון דרישות - כל הדברים שצריך לעשות, המרת הדרישות לפיצ'רים (לכל פיצ'ר נצימד USE CASE) ותיעדופם, יצירת ארכיטקטורה ב-High level ופיצול צוותים.
- תהליך איטרטיבי ואינקרמנטלי – פיצול הפיצ'רים ל stories (יחידה פונקציונאלית שפיתוחה ובדיקתה לוקח כשבועיים), הערכת זמני פיתוח, דו שיח עם הלקוח באופן שוטף ותיעדוף הפיתוח ע"י הלקוח, פיתוח מבוסס TDD, הערכת הסטאטוס שלי על בסיס מצב הקוד ושיקפונותו ללקוח.

פרקטיקות ופרוצדורות אג'יליות:

- מטאפורה – הלבשת רעיון חדש על ידע קיים – בסיס לחשיבה ופיתוח דיון אודות המוצר.
- User story – מייצג פיצ'ר אחד או יותר במערכת, קטן וממוקד.
- Sustainable Pace – דאגה לצוות רענן ואפקטיבי במשך כל זמן הפרויקט – הימנעות משחיקה.
- Customer Team Member – ראה מנהל מוצר בהרצאה 3.
- Planning Game – תהליך ששותפים לו גם הלקוח וגם צוות הפיתוח כאשר המתכנתים מעריכים את משקל הפיצ'רים והלקוח מתעדף את הפיתוח שלהם.
- Simple Design – השארת העיצוב כמה שיותר פשוט ונח לשינויים ותוספות.
- Small Releases – שחרור גרסאות קטנות ולא גרסה אחת גדולה בבת אחת.
- Test Driven Development – לכל יחידת קוד נגדיר סט בדיקות לפני הכתיבה בפועל (כתיבה מונחת בבדיקות TDD).
- Collective Ownership – הקוד באחריות כל הצוות, כולם אמורים לשלוט בו.
- Coding Standards – סטנדרטים בריאים לכתיבת קוד – אחידות וסגנון.
- Refactoring – המרת מבנה קוד קיים מבלי לשנות את הפונקציונאליות, התהליך שומר על העיצוב המקורי.

16. *Burn Down Charts* - (תרשים שריפה):

כלי המשמש את צוותי ה-agile כדי לאסוף מידע על העבודה שהתבצעה עד עכשיו ועל העבודה העתידית שצריכה להתבצע בפרק זמן הנתון (צוותי ה-scrum מכנים אותה כ-sprint). לעתים קרובות, צוותים יכולים להשתמש בו ככלי חיזוי המאפשר להם לדמיין מתי הפרויקט שלהם יסתיים. הכלי מספק גם תובנות לגבי בריאות הספרינט שלהם. לאחר השתקפות של תרשים השריפה, הצוותים מקבלים תובנות לגבי צווארי בקבוק בתהליך ולאחר מכן הם מסוגלים לקבוע פתרונות למכשולים, המובילים לתוצאות משמעותיות. דוג': JIRA.

17. *Scrum*:

מהי שיטת scrum?

טכניקה לתהליך פיתוח זריז המבוסס על עקרונות ה-agile. בטכניקה זו יש דגש על פיתוח איטרטיבי- עבודה בספרינטים, עבודה מרוכזת עם שקיפות ושיפור.

משתמשים בשיטה זו בכל מקום החל מארגונים גדולים ועד סטארטאפים קטנים.

שיטה זו מקדמת את הערכים הבאים: מוטיבציה, עבודת צוות ותקשורת, העצמת הפרט, תרבות אירגונית והגברת קצב הפיתוח.

- sprint: תקופה מסוימת בה מחליטים על משימות ויעדים שבסיומה נגיע אליהם. לאחר כל סיום sprint יגיע sprint חדש.
- sprint planning: הכנה לתקופה הקרובה והגדרת המשימות בהם הצוות יעסוק.
- מה ניתן לספק בסיום התקופה? כיצד נספק את היעדים עד לסיום התקופה?
- daily scrum: פגישה יומית של חברי הצוות בה מתכננים את העבודה ליום הקרוב.
- sprint review: פגישה המתקיימת בסיום כל sprint בה כל אחד מראה ומסביר על התוספות שהכניס למוצר בתקופה שהסתיימה.
- sprint retrospective: פגישה המתרחשת לפני ה-planning ואחרי ה-review ומטרתה לתת לצוות אפשרות לבדוק את עצמו ואת המוצר שלו ולתת פרספקטיבה לגבי התכנון העתידי.

תסרוכות:

- לא ידוע מה הולך להיות התוצר הסופי.
- תפקיד ה-scrum master קשה.
- הטמעת השיטה מצריכה שינויים ברמת הארגון.

במודל זה יש מספר תפקידים:

- Scrum master – מעיין מנטור שמנתב את הצוות להצלחה אך לא בהכרח בעל הכח לקבל החלטות.
- Product owner, Developers, Testers
- הנהלה – כיוון שבמודל זה אין מנהל פרויקט והצוות מנהל את פעילותו אזי תפקידה של הנהלה הוא ליצור מודל עסקי, לכבד את החוקים והשיטה, לסייע בסילוק מכשולים, לאתגר את הצוות למצוינות.

18. דרכים להתמודדות עם מורכבות

- אבסטרקציה: התעלמות מפרטים והתמקדות "בתמונה הגדולה".
- דקומפוזיציה: פירוק הבעיה לאוסף תת בעיות בלתי תלויות (האם אי תלות באמת קיימת?).
- מודולציה: הגדרת מבנים יציבים לאורך זמן.
- פרויקציה: התמקדות בזווית ראייה אחת כל פעם (View) בהתייחס לתת הבעיה הנבחרת.

19. גישות Top Down ו-Bottom Up לעיצוב

Bottom Up ו-Top Down הן אסטרטגיות לפיתוח תוכנה.

במודל **Top Down** מנוסחת תחילה סקירת מערכת בקווים כלליים בלי להיכנס לפרטים של שום חלק ממנה. לאחר מכן מנוסח כל חלק של המערכת לפרטיו, ואז כל פרט חדש עשוי להיות מנוסח שוב תוך שמגדירים אותו לפרטים נוספים עד שכל המפרט מפורט דיו כדי לתקף את המודל. מודל מעלה-מטה מעוצב לעיתים קרובות תוך הסתייעות ב"קופסאות שחורות" שמקלות את מימוש סקירת המערכת, אך באופן שאין בו די להבנת המנגנון שביסוד המערכת.

בניגוד לזאת, במודל **Bottom Up** מוגדרים תחילה החלקים הפרטיים של המערכת לפרטיהם, ואז מצורפים החלקים יחדיו לרכיבים גדולים יותר, שבתורם מצורפים גם הם עד להשגתה של המערכת השלמה. אסטרטגיות שמבוססות על זרימת מידע מטה-מעלה עשויות להראות כחיוניות ומספקות עקב היותן מבוססות על ידיעת כל הגורמים שעשויים להשפיע על חלקיה היסודיים של המערכת.

20. עיצוב תוכנה ומדוע הוא נדרש (שינויים יגיעו, יש להתכונן בעיקר לשלב התחזוקה)

עיצוב (תכן) מבנה תוכנה :

ארכיטקטורת מערכת vs ארכיטקטורת תוכנה:

- **ארכיטקטורת מערכת** – מציגה מודל שמכיל קונספטים שמתארים מבנה והתנהגות הכוללים תתי מערכות. בנוסף מודל זה מציג את החומרה ואת התקשורת.
 - **ארכיטקטורת תוכנה** – מציגה מבנה high level (מבט מלמעלה בזום אאוט) לתוכנה הרצויה המכילה סעיפי אב כך שלכל מימוש או תתי מערכות יש אב (מעין עץ).
- בנוסף, הארכיטקטורה מכילה את הגדרות מטרות העיצוב, מודלים מרכזיים, תוכנות מרכזיות והקשרים ביניהם.

הסיבות לחשיבות שלב עיצוב התוכנה:

- **התמודדות עם מורכבות**: תהליך הפיתוח הוא דינאמי ומשתנה כל הזמן ולכן חשוב שיהיה לנו מבט מלמעלה שמרכז ומפקס את הכל ובכך יוצר יציבות, בנוסף שלב העיצוב מפרק לנו את הבעיות הגדולות לתתי בעיות קטנות ובלתי תלויות, בנוסף נבין כי המערכת מורכבת מהמון ישויות שנוכל ללכת לאיבוד מרוב שהם רבות, שלב העיצוב נותן לנו את היכולת להבין את הקשרים ביניהם.
- **התמודדות עם מחזור חיי תוכנה ותנאי סביבת פיתוח**: כאמור לעיל יש שינויים תכופים כל הזמן אך בנוסף נרצה שאורך שלב התחזוקה שהוא אחד המרכיבים החשובים ביותר לא יהיה ארוך ומרכזי מעבר לתפקידו (לא נרצה לפתח מוצר שנה ולהשתמש בו חצי שנה) – מבנה ועיצוב נכון ימנעו את זה ויאפשרו אדפטציה מהירה לשינויים.

21. Design Goals (הכיוונים המרכזיים אליהם חותרים בעיצוב המערכת)

- **High Level Design** - הגדרת מודלים מרכזיים, גרנולציה (רמת פרטים) נמוכה. (דוג: MVC) מכונה גם Architectural Design.
- **Low Level Design** - הגדרת מודלים ברמת גרנולציה גבוהה. (חלוקה למחלקות בקוד). מכונה גם Detailed Design.

High Level Design:

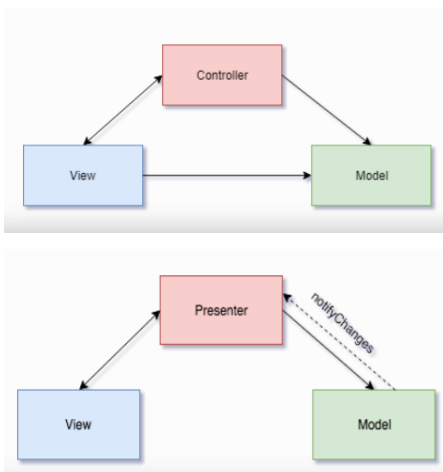
• MVC - Model-View-Controller:

הפרדה בין הלוגיקה (Model) לתצוגה (View) המודל אחראי רק על הלוגיקה של הנתונים (לדוג' כל המחלקות הרגילות של JAVA שמייצגות משתמשים כמו פרופיל מורה ועוד) והתצוגה אחראית על ההצגה של הנתונים.

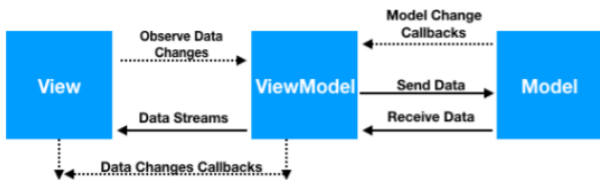
התפקיד של הבקר (Controller) הוא לקשר ביניהן, איך זה נעשה בפועל? התצוגה מקבלת קלט מהמשתמש (לדוג' לחיצה על כפתור באפליקציה) התצוגה מידע את הבקר על מה שאירע הבקר מעביר את הנתונים למודל והוא מעבד אותם (לדוג' מוסיף סטודנט לרשימת התלמידים של מורה מסוים).

• MVP - Model-View-Presenter:

הרעיון הוא למעשה למנות איזשהו נציג בצורה של ממשק שיקשר בין כל חלקי המערכת. ה-Presenter יחזיק את הממשקים של ה-View ושל ה-Model וכך יתקשר איתם. המודל יחזיק מצביע לממשק של ה-Presenter וכאשר יהיו שינויים, הוא יעדכן אותו באמצעות המצביע (callback). חשוב לזכור שזה כמו MVC רק שיש ממשק מקשר.



עיקרי מושגים ונושאים



• MVVM - Model-View-ViewModel :

מאוד דומה ל-MVC, עקרונות התבנית בנויות מ:

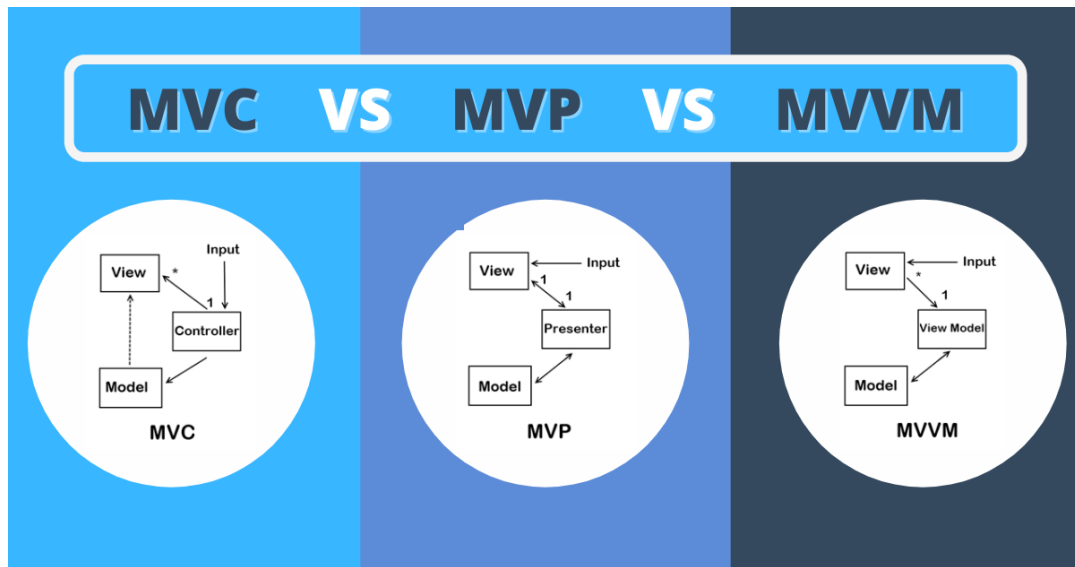
Model - מתאר את האובייקטים שמהווים את ה-Data של התוכנית (לוגיקה).

View - החלק האחראי על הגרפיקה ועל ממשק המשתמש (UI).

כמו כן הוא מחזיק מופע של ה-ViewModel.

ViewModel - החלק אשר מהווה רכיב תקשורת בין החלקים ומחזיק רכיבי לוגיקה של UI ומופע של Model.

הרעיון הוא הפרדה מוחלטת בין ה-View לבין ה-Model כשהתקשורת ביניהם היא באמצעות מנגנוני Commands להעברת פקודות/מסרים, ו-Data Binding בין ה-View ל-ViewModel כדי להציג נתונים.



: Low Level Design

- OOA - ניתוח מונחה עצמים: איתור הדרישות הפונקציונליות והלא פונקציונליות של המערכת: סט הקלטים, ההתנהגויות והפלטות הנדרשים למימוש לצד החלטות ואילוצים נלווים. בגדול: ניתוח המערכת בצורה של אובייקטים ואירגונה בעזרת תרשימים כמו ERD וכו'.
- OOD - עיצוב מונחה עצמים: עיצוב אפליקציה ועיצוב מערכת (לא מבחינת נראות) שימוש בתבניות ארכיטקטוניות ותבניות תכן, שילוב עקרונות תכן. בגדול: תכנון של המחלקות בצורה של מונחה עצמים (מחלקות, ממשקים וכו) תוך שימוש בתבניות עיצוב.
- OOP - תכנות מונחה עצמים: התכלס, פרקטיקות מימוש המעודדות יצירת קוד מונחה עצמים "טוב": שימוש בממשקים (יצירת גרסאות קוד אבסטרקטיות), TDD וכו'...

22. Scalability (מדרגיות):

מדרגיות, היא פיתוח תוכנה שמסוגלת לגדול לפי הצורך, היכולת של מערכת להתמודד עם כמות גדלה של עבודה על ידי הוספת משאבים למערכת. דוגמה: מערכת משלוחי חבילות היא מדרגית כיוון שהיא ניתנת להרחבה, הוספת משאבים (רכבי משלוח) תאפשר להתמודד עם כמות חבילות גדולה יותר.

23. סימפטומים למערכת הנמצאת בדעיכת הגדרות עיצוב (שבירות, צמיגות, נוקשות, אי-מוביליות):

תכונות ומאפיינים למערכת הנמצאת בדעיכת הגדרות עיצוב:

- נוקשות – אין את היכולת לשנות בקלות, המערכת נוקשה ומגיבה רע לשינויים. יכול להתבטא גם בהשקעה גדולה בשביל שינויים קטנים.
- שבירות - שינוי קטן יכול לפרק את כל המערכת.
- אי-מוביליות - חוסר יכולת לעשות שימוש חוזר (reuse), כלומר המערכת לא מופרדת כמו שצריך ולכן לא תוכל לקחת משהו ולהשתמש בו במקום אחר.
- צמיגות – חוסר גמישות, קשה מאוד לזוז – כמות השלבים שתצטרך לעבור בשביל לעשות שינויים היא קטסטרופלית ומלווה בבירוקרטיה.

24. Dependency Management, Dependency Firewalls:

ניהול תלות (Dependency Management) הוא התהליך של ניהול כל המשימות והמשאבים הקשורים זה בזה כדי להבטיח שהפרויקט הכולל יסתיים בהצלחה, בזמן ובתקציב. כאשר יש תלות שצריך לנהל בין פרויקטים, זה מכונה ניהול תלות הדדית בפרויקט.

הרעיון בתלות בחומות (Dependency Firewalls) הוא ליצור הפרדה בין מודלים שונים בקוד כך שלא יהיו תלויים אחד בשני, לדוגמה אם תיגרם שגיאה במודל אחד לא נרצה שזה יפיל את המודל השני. ממשק הוא דוגמה מצויינת ל-Dependency Firewalls כיוון שהוא יכול להוות גדר וחציצה בין מודלים ובכך למנוע משינויים להיכנס פנימה ולהקריס את המערכת.

25. חשיבות ממשקים בכתיבת קוד

ממשק יכול להוות dependency firewall כיוון שהוא מעיין גדר וחציצה בין מודלים ובכך יכול למנוע משינויים לבעבע פנימה ולהקריס מערכת. בנוסף, תכנון מערכת על בסיס Interfaces היוצרים בצורה אבסטרקטית חיבורים ביניהם, מאפשרים לקבל מידול וחלוקה נכונה.

26. ארכיטקטורה ועיצוב תוכנה

ארכיטקטורה היא התחום העוסק בתכנון מערכות תוכנה. המונח ארכיטקטורה בהנדסת תוכנה פירושו ייצוג היבטים שונים של התוכנה באופן מופשט. ארכיטקטורה של מערכות תוכנה היא לפיכך, תכנון מופשט של ההיבטים השונים של התוכנה, היחסים בין המרכיבים השונים של התוכנה והחוקים החלים עליהם.

27. תבניות ארכיטקטוניות

MVC, MVP, MVVM כפי שמפורטים בסעיף 21 ובנוסף:

:Event Bus

דומה ל-bus ממערכות הפעלה, סוג של אזור המשמש להעברת מידע בין תהליכים. בתמונה ניתן לראות שישנם שני תהליכים שמעדכנים שני ערוצים (1,2) bus ושתי תהליכים שמאזינים לאותו מידע והכל קורה דרך bus.

:Multi Layer

ארכיטקטורת שלוש השכבות, התבנית מורכבת מהשכבות הבאות:

• שכבת תצוגה (UI):

זוהי השכבה העליונה של היישום. שכבת התצוגה מציגה מידע הקשור לשירותים כדוגמת דפדוף בין מוצרים, מידע אודות הקנייה ותוכן עגלת הקניות. השכבה מתקשרת עם שכבות אחרות באמצעות העברת פלט התוצאות לדפדפן או לתוכנת לקוח אחרת, ולכל יתר השכבות ברשת.

• שכבת היישום (Business Logic):

נקראת גם "שכבת הלוגיקה העסקית", "שכבת הגישה לנתונים" או "השכבה האמצעית". שכבה זו שולטת בפונקציונליות של היישום על ידי ביצוע עיבוד הנתונים הפרטני.

• שכבת הנתונים (Data Access):

שכבה זו מורכבת משרת בסיס נתונים אחד או יותר. מכאן המידע נשלף ומאוחסן. שכבה זו שומרת על הנתונים עצמאיים ובלתי תלויים בשרתי היישומים ובלוגיקה העסקית. אחסון הנתונים בשכבה נפרדת משפר גם את הסילומיות והביצועים של היישום.

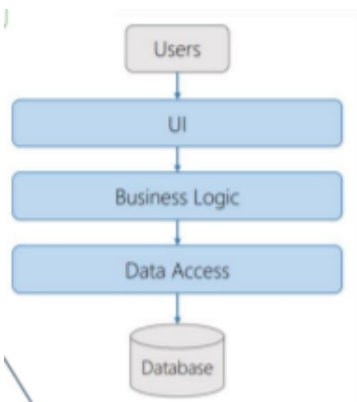
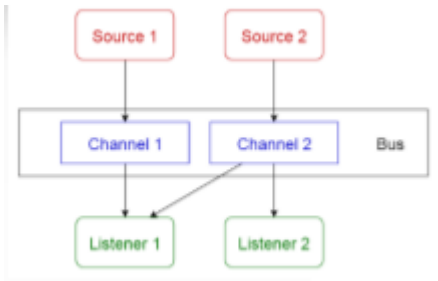
השוואה בין Multi Layer לבין מודל ה-MVC:

מבט ראשון, מודל שלוש השכבות נראה דומה לתבנית העיצוב Model-View-Controller.

עם זאת, מבחינה טופולוגית הם שונים. כלל יסוד בארכיטקטורת שלוש השכבות הוא שהלקוח לעולם לא מתקשר ישירות עם שכבת הנתונים. במודל שלוש השכבות כל התקשורת חייבת לעבור דרך השכבה האמצעית, ולכן מבחינה רעיונית, ארכיטקטורת שלוש השכבות היא ליניארית. לעומת זאת, ארכיטקטורת MVC היא משולשת: המבט (view) שולח עדכונים לבקר (controller), הבקר מעדכן את המודל (model), וה-View מתעדכן ישירות מהמודל.

:Ports and Adapters

יצירת האפליקציה כך שתהיה מסוגלת לרוץ בלי UI ובלי דאטה בייס מקושר, זאת על מנת שנוכל להריץ טסטים אוטומטים על מערכת (ללא קישור לדאטה בייס) ממומש כאשר ה-DB אינו זמין מסיבות כאלה ולמרות זאת אנו רוצים לבדוק את האפליקציה.

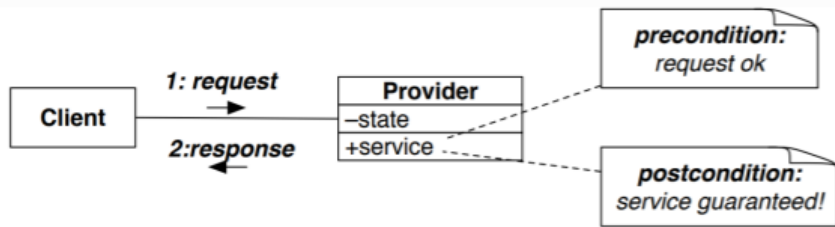


28. DBC (צפייה לקיום תנאים מוקדמים, הבטחה למילוי תנאים ביציאה, שימור אינווריאנטים)

Design By Contract: עיצוב ע"פ חוזה.

הגדרת ממשקים פורמליים עם תנאים מדויקים אשר יאפיינו את חלקי המערכת בצורה מדויקת. ממשק הוא למעשה סוג של חוזה, כי כל מי שממש אותו חייב לממש את כל הפונקציות שהוגדרו בממשק (חוזה).

- preconditions: תנאים שחייבים להתקיים לפני שמפעילים פונקציה (או תהליך).
- Postconditions: תנאים שחייבים להתקיים לאחר ביצוע פונקציה או תהליך.
- Invariants: תנאים שחייבים להתקיים לפני ואחרי ביצוע פונקציה או תהליך.
- Side effects: צריך להימנע משינוי של משתנה במערכת מחוץ לתחום שבו הוא מוגדר.



תנאים מקדימים מחייבים את המשתמשים (שולחים קלט), תנאים מאוחרים מחייבים את המחלקות (שמעבדות את הקלט).

החוזה למעשה מניח שכל התנאים המוקדמים והמאוחרים מתקיימים (רק כך המערכת עובדת כמו שצריך).

החוזה מחייב את הלקוח לבקש בקשות תקפות (מוגדר ע"פ החוזה) ומחייב את הספק לספק את השירות בצורה נאותה (גם מוגדר ע"פ החוזה) אם אחד מהצדדים הפר את החוזה יהיה כשל במערכת ותיזרק שגיאה.

29. Unit Testing vs TDD

TDD - עיצוב מונחה בדיקות: בדיקות יחידה הנכתבת בטרם נכתב הקוד אותו היא בודקת.

Unit Testing - כתיבת טסטים קטנים ורבים כאשר כל טסט בודק פונקציה פשוטה מאוד או התנהגות אובייקט מסוים. ההבדל המרכזי בין TDD לבין UT הוא שTDD הוא תהליך חשיבה שמביא לבדיקות יחידה, ו"thinking in tests" נוטה להוביל לבדיקות עדינות ומקיפות יותר ובנוסף הוא קל יותר להרחבה.

30. Unit Tests and Mock Objects

Unit Test, אובייקטים מדומים (mock objects) יכולים לדמות התנהגות של אובייקטים מורכבים ואמיתיים ולכן הם שימושיים כאשר אובייקט אמיתי אינו מעשי או שלא אפשרי לשלב אותו בUnit Test.

אם לאובייקט יש אחד מהמאפיינים הבאים, ייתכן שיהיה שימושי להשתמש באובייקט מדומה במקומו:

- האובייקט מספק תוצאות לא דטרמיניסטיות (למשל השעה הנוכחית או הטמפרטורה הנוכחית).
- יש לו מצבים שקשה ליצור או לשחזר (למשל שגיאת רשת).
- הוא איטי (למשל מסד נתונים שלם, שיהיה צורך לאתחל לפני הבדיקה).
- הוא עדיין לא קיים או עשוי לשנות התנהגות.
- הוא יצטרך לכלול מידע ושיטות אך ורק למטרות בדיקה (ולא למשימתו בפועל).

דוגמה: תוכנית שעון מעורר שגורמת לפעמון לצלצל בשעה מסוימת עשויה לקבל את השעה הנוכחית משירות זמן.

כדי לבדוק זאת, על הבדיקה להמתין עד למועד האזעקה כדי לדעת אם היא צלצלה נכון בפעמון.

אם נעשה שימוש בשירות זמן מדומה במקום שירות בזמן אמת, ניתן לתכנת אותו לספק את זמן צלצול הפעמון (או כל זמן אחר) ללא קשר לזמן האמיתי, כך שניתן יהיה לבדוק את תוכנית השעון המעורר במנותק.

31. מהו קידוד ויזואלי (בהקשר הקורס):

קידוד ויזואלי/חזותי מתייחס לתהליך שבו אנו, כבני אדם זוכרים תמונות חזותיות.

דוגמה: הלוגו של מאסטרקארד. כשאתה רואה את עיגול כתום ואדום משתלבים, אתה יודע באיזו חברה מדובר. אתה לא מעבד עיגול כתום כאן, עיגול אדום כאן, הטקסט שם, ואז חושב על חיבורו. אתה רואה את הסכום, לא את החלקים (תיאוריית הגשטאלט).

היתרונות של קידוד חזותי: הפירושים של התמונות השלמות הללו הן מה שנאגר בזיכרון כמייצג של האובייקט או הגירוי שנצפה. קידוד חזותי יכול להגביר את הזיכרון ולעזור לקודד מידע עמוק יותר לתוך הזיכרון. בהקשר לקורס נרצה להיטמע במאגרי הזיכרון של המשתמש.

32. User Profile

פרופיל משתמש (User profile) הוא אוסף הנתונים האישיים הנשמרים עבור משתמש קצה מסוים, לרוב בפלטפורמה דיגיטלית מקוונת. חשוב להפריד בין מערכות תגובות לבין מערכות פרופילים. פרופיל המשתמש נשמר על ידי כל מערכת המעוניינת בכך, שאיתה המשתמש בא במגע, החל מיישומים המעוניינים לשמור את הגדרות ממשק המשתמש, דרך מערכת הפעלה המעוניינת לשמור את פעולותיו האחרונות של המשתמש לצורך נוחות בהפעלה נוספת, וכלה באתר אינטרנט המכיל מידע אישי שאותו הוא מציג למשתמש בכל פעם שזה גולש אליו. פרופיל משתמש אם כן הוא הייצוג הדיגיטלי של זהותו של אדם מסוים בעיני תוכנה מסוימת ובעיני שאר המשתמשים באותו מימד.

33. מודל חמשת ה-S

חמשת השכבות לחווית משתמש (UX) טובה: Strategy, Scope, Structure, Skeleton & Surface.

Strategy (אסטרטגיה) - מצא איזון בין היעדים העסקיים לצורכי המשתמשים. Booking, למשל, שואפת למצוינות בתחום הזמנת הנסיעות ומכירה בכך שהצרכנים רוצים גמישות ושירות איכותי. כדי להשיג את האיזון הזה, Booking מציעה מידע וכלים המאפשרים למשתמשים לקבל החלטות ולהזמין בזמן הנכון ובמחיר הנכון.

Scope (היקף/תחום) - הפיכת אסטרטגיה לדרישות, הגדרת מאפייני המוצר או השירות או כל מידע בעל ערך מוסף אחר שיוצג באתר או באפליקציה. פשטות היא המטרה הסופית. ערכו רשימה של המידע, הכלים והאפשרויות הרצויות. עבור Booking, זה כולל נתונים על לינה, טיסות, השילוב של תעריף טיסה ומלון, השכרת רכב, דירוג משתמשים וירידות מחירים.

Structure (מבנה) - כאן נכנס לתמונה עיצוב הפלטפורמה וממשק המשתמש (UI). מבנה של אתר או אפליקציה קובע כיצד המערכות יגיבו לפעולות המשתמש ומורכב משני היבטים:

1. עיצוב אינטראקציה. המטרה היא להגדיר התקדמות כדי להדריך את המשתמשים לאן אתה רוצה שהם יגיעו.

ב-Booking, הצעד הראשון הוא לגרום למשתמשים להזין את היעדים, התאריכים ומספר הנוסעים שלהם.

משתמשים ממלאים טופס כדי לראות רשימה של אפשרויות זמינות, והם יכולים לראות פרטים אודות המלון לפני אישור ההזמנה שלהם.

2. ארכיטקטורת מידע. זה מפרט את התוכן, איך הוא יהיה בנוי והיכן הוא יוצג על המסך. Booking בוחר להציג את המאפיינים המבדילים בין המלונות כבר בתוצאות החיפוש, וכך המשתמש לא צריך ללחוץ על הקישור של המלון כדי לראות את מיקומו. לפיכך, זה עוזר לנוסעים לבחור (או לבטל) כל אפשרות מהר יותר, בהתבסס על מה שהם מחפשים.

Skeleton (שלד) - ברמה זו המבנה המדויק של האפליקציה/אתר נחתם, כולל היכן ממוקמים רכיבי הממשק על המסך וכיצד הם מקיימים אינטראקציה זה עם זה. מה שחיוני כאן הוא עיצוב הממשק והניווט (אלמנטים המאפשרים למשתמש לנווט בצורה אינטואיטיבית, למצוא מידע או אינטראקציה).

ההפצה של Booking מבוססת על משולש הזהב (שלפיו האזור הנצפה ביותר באתר הוא החלק השמאלי העליון). בחלק העליון של העמוד יש כפתורים המקשרים לשירותים העיקריים המוצעים, בעוד שחלונית משמאל מכילה אפשרויות חיפוש ולאחר מכן מסננים. תוצאות החיפוש מופיעות באמצע העמוד.

Surface (משטח) - עיצוב חזותי אמור לעזור למשתמשים להבין את המידע המופיע באתר ולהנחות אותם בשימוש בו באופן אינטואיטיבי, תוך שהוא מתאים מבחינה חברתית ותרבותית.

Booking תמיד מציבה את האפשרויות והמידע הרלוונטיים באותם מיקומים כדי לאפשר ניווט קל. תחת שם המלון, למשל, הוא מציג תמיד את הכתובת וקישור למפה, ומתחת הוא מציג תמונות, דירוגים ממוצעים והערות משתמשים.

34. צימוד ולכידות

בהנדסת תוכנה, מדד **צימוד** (Coupling) מייצג את רמת התלות בין מודולים שונים באותה מערכת. מונח נוסף אשר הומצא במסגרת שיטה זו הוא מדד **הלכידות** (Cohesion), המתייחס לחוזק הקשר הפונקציונלי בין פעולות שונות תחת אותו מודול. רמת לכידות גבוהה, במקרים רבים, היא סימן לרמת צימוד נמוכה ולהפך.

בהערכת איכותו של קוד תוכנה, יועדף קוד בעל צמידות נמוכה, המהווה סימן לכך שהקוד תוכנן כהלכה ובנוי היטב. על מנת להשיג צמידות נמוכה, יש לתכנן כל מודול כיחידה עצמאית ככל הניתן, אשר תלויה כמה שפחות במודולים נוספים במערכת. במצב זה, במידה שיידרשו שינויים בקוד, אלו יבוצעו ביתר קלות כיוון ששינוי במודול מסוים לא יאלץ שינויים בכל המערכת.

שילוב של צמידות נמוכה ולכידות גבוהה הוא סימן לכך שהקוד בעל יכולת תחזוקה גבוהה, מצביע על חוסן, אמינות, יכולת שימוש מחדש ויכולת הבנה גבוהה של הקוד.

35. Solid

עיקרי מושגים ונושאים

עקרונות SOLID הם חמישה עקרונות בסיסיים בעיצוב מונחה-עצמים. המחשבה מאחורי העקרונות היא שכאשר הם מיושמים יחדיו בפיתוח של מערכת תכנה, סביר שהיא תהיה יותר קלה לתחזוק והרחבה לאורך הזמן.

- Single responsibility - לכל מחלקה צריכה להיות אחריות אחת ויחידה.
- Open for Extension, Closed for Modification - מחלקה צריכה להיות סגורה לשינויים ופתוחה להרחבה (למנוע שינויים ולאפשר ירושות).
- Liskov substitution principle - אובייקטים בתוכנה יכולים להיות מוחלפים על ידי מחלקות ירושות ללא שינוי תפקוד התוכנה בכללותה.
- Interface segregation principle - למחלקות יהיו ממשקים שונים אשר יותאמו לפי צרכי המשתמשים.
- Dependency inversion - מימושים יהיו תלויים בממשקים ואבסטרקציות ולא במימושים פנימיים.

36. SMART

מודל לקביעת יעדים:

- Specific - יעדים ממוקדים ככל הניתן (ולא כלליים).
- Measurable - יעדים הניתנים למדידה.
- Attainable - יעדים ברי השגה, ריאליים והגיוניים.
- Relevant - יעדים המשרתים את הייעוד ואת האסטרטגיה הארגונית.
- Time-Bound - יעדים התחומים בזמן.