סיכום קורס הנדסת תוכנה :

שיעורים 1+2 :

מושגי יסוד והגדרות :

הנדסה – תפקידה של ההנדסה זה ליצור דברים, אך לעומת אומנות לצורך העניין ההנדסה מבוססת ומסתמכת על העולם המדעי.

בהנדסה נרצה לצמצם את האי ודאות וננסה להיכנס לכמה שיותר תבניות בכדי שנוכל לייצר את המוצרים כמה שיותר דומים אחד לשני, באומנות לעומת זאת השונות היא לכתחילה וזה מה שמייחד אותה.

תוכנה – אוסף של שירותים דיגיטליים ,לדוגמא מערכת הפעלה זה אוסף של שירותים מסוימים, או שאפליקציה זה אוסף של שירותים ייעודיים (עיבוד תמונה וכו').

הנדסת תוכנה – עוסקת ביצירה של מוצרי תוכנה – אך בניגוד להנדסה אין שמירה על אחידות המוצרים והתוכנות שונות ומשונות בהתאם לדרישות הלקוח, אך בכל זאת אנו שואפים להיות הנדסה – אנחנו משתמשים בתבניות, תהליכים, שיטות עבודה ופרקטיקות כלליות.

מערכת – אוסף של חלקים שמקיפים את כל המציאות הקיימת (מערכת השמש וכו'), כלומר אנחנו לוקחים כמה יישויות מופשטות (העיקר שלו, התמצית שלו) שמקיימות קשרי גומלין או עם תלות הדדית מתמשכת ובעלת מטרה.

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.

Stakeholders – אדם בעל עניין או אינטרס שנמצא בתוך המערכת והרבה פעמים הוא משנה את כל קבלת ההחלטות בשלב תכנון ויצירת התוכנה.

UI/UX – חייבת להיות התאמה כמה שיותר שלמה בין חווית המשתמש (UX) למשתמש עצמו (UI), לכן קודם נבין לעומק מי המשתמש כדי שנוכל להתאים את המוצר כמה שיותר.

מודל – לקיחת מערכת וייצוגה בצורה מופשטת.

תיכנון תוכנה (software design) – עיצוב זו הגדרת פתרון כאשר אתה כפוף לאילוצים שאני חייב לעמוד בהם שוב בניגוד לאומנות שלא מוגבלת באילוצים מהנדס צריך לעמוד בשלל אילוצים והגבלות ובהתאם אליהם הוא צריך למקסם את הפתרון (התוכנה) שלו.

כאמור בתהליך הפיתוח התוכנה החכמה זה להתמודד עם דרישות (אילוצים) דינמיים ומשתנים תוך כדי הפיתוח, כלומר האילוצים שהיו לי בהתחלה והפתרון שהיה לי כבר לא רלוונטיים לפעמים בתהליך הפיתוח עצמו.

בתכנון התוכנה יש להתחשב גם בקלות השימוש וזמן הקימפול.

ארכיטקטורה – זה סוג של עיצוב וההבדל בין ארכיטקטורה לעיצוב זה עומק השינוי , כלומר אם אעשה שינוי בארכיטקטורה אני מבצע שינוי כבד ועמוק מאוד.

עיצוב מודולרי – עיצוב תוכנה = מודולציה – לקיחת השלם ופריסה לחלקים כאשר כל חלק יקרא מודול אם הוא מתחלק בצורה נכונה ושלמה.

וזה העיקר בעיצוב התוכנה.

מודולציה לוגית – חלוקת המערכת למרכיבים לפי תפקיד.

מודולציה פיזית – חלוקת המערכת לחלקים בלי התחשבות בתפקיד.

המינימום נדרש והבסיסי כדי שהעיצוב יהיה ראוי זה שהלוגיקה המרכזית יהיה חלק והטיפול בנתונים יהיה חלק שונה.

עקרון SOC – כל פעם שיש יחידה בקוד או יחידה כלשהיא שעושה יותר מתפקיד אחד זו בעיה , לכן בעיקרון זה כל יחידה היא בעלת תפקיד מוגדר ויחידי, על פי עקרון זה אנו פועלים.

הדרישות הנצרכות כדי ליצור מוצר תוכנה –

- אין מוצר תוכנה שאין לו צורך ובעיה הדורשת פתרון, זה הבסיס לכל מוצר.
- רעיון ,חזון , דרישות, התכנות קיימת בעולם.
- משאבים ושותפים לדרך.
- ידע – תמיד חסר בשוק ידע.
- תהליכי ניהול – יכולות שיווקיות פיננסיות והנדסיות.
- סביבות פיתוח – שלושה סביבות נדרשות כדי לפתח תוכנה – פיתוח בדיקות וייצור.
- תזמון נכון.

קשיים בפיתוח תוכנה – לו"ז ,תקציב ,התאמה לצרכים ,איכות ,שימוש חוזר ,תחזוקה.

הסיבות המרכזיות לכשלים בפיתוח תוכנה –

- הגורם האנושי – תקשורת לקויה (פתרון : גיבוש קבוצתי והכרת הסיבה והאנשים בעבודה) ,ניהול לוקה ,אינטרסים מנוגדים ,סטנדרטים ושיטות עבודה שלא מוגדרות היטב , חוסר בבקרת איכות , חוסר מקצועיות.
- טבעו של עולם – אי וודאות , מורכבות ,דרישות מוגזמות , לו"ז ותקציב.
הפתרון : קבלת המציאות והתאמה של המציאות למצב ,שילוב ושימוש בכלים שלך בצורה נכונה.

פעילויות בפיתוח מוצרי תוכנה –

- ייזום – היזם מזהה את הבעיה ואת ההזדמנות.
- הגדרת דרישות – הגדרת ותיאור הדרישות.
- ניתוח – ניתוח ושיפור הבעיות.
- עיצוב ומימוש.
- בדיקות.
- הטמעה והחזקה.

תהליך יזום יצירת תוכנה –

מסמך יזום – יצירת מסמך שנותן "כותרת" לפרויקט בגדול.

המסמך אינו מסמך טכני אך הוא מתאר את המצב הקיים ואת הבעיות והפתרונות.

המסמך ישמש כקלט ובסיס לתהליך הפיתוח.

המסמך מורכב מהמטרה וההצדקה לפרויקט ,יעדים ומדדים להשגתם ,דרישות ותיאור כללי, אילוצים ולו"ז משוער ,הערכות תקציב ורשימת בעלי עניין.

מודל SMART לקביעת יעדים –

- **Specific**- יעדים ממוקדים ככל הניתן, ולא כלליים מידי.
- **Measurable**- יעדים הניתנים למדידה
- **Attainable**- יעדים ברי השגה- ריאליים והגיוניים.
- **Relevant**- יעדים המשרתים את הייעוד והאסטרטגיה הארגונית.
- **Time-bound**- יעדים התחומים בזמן.

Use case – מי בעל התפקיד ומה הוא רוצה מהמערכת ומה השימושים שלו (כל דרישה).

- ההבדל בין לקוח למשתמש – לקוח זה אחד שמשלם ואילו משתמש זה מי שמפעיל את התוכנה בקצה , האינטרסים של שניהם שונים לגמרי וצריך להבין מה הלקוח רוצה כדי למקסם את השימוש של המשתמש בקצה.
- תהליך התמודדות עם קשיים בתכנון והנדוס תוכנה :
  - תהליכים אדפטיביים – היכולת להגיב לשינויים.
  - עקרונות – עיסוק בכוח אדם, שיתוף פעולה , תקשורת.
  - פרקטיות הנדסיות – התאמת הקוד לשינויים (החלפת אלגוריתמים ,שינוי בהתאם לבאגים),TDD - תכנון ויצירת בדיקות לפני כתיבת הקוד (כתיבה מונחת בדיקות).

<u>שיעור 3 :</u>

<u>בעלי תפקידים :</u>

בעלי עניין :

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

מנתח מערכות :

- מנתח את המצבים הקיימים(מרחב הבעיה) ומבין את הבעיות הדרושות.
- מצליח להגדיר את הבעיה ואת קווי המתאר לפתרון שלה.
- קובע את מטרות המערכת ומגדיר את דרשותיה.
- אוסף ומסווג את הדרישות מהלקוחות ומהמשתמשים.

מנהל פרויקט :

- האחראי שצריך לדאוג לעמידה בדרישות ,בתקציב ובלו"ז.
- מכיר את כל התהליכים ומתאימים אותם לצורכי הפרויקט.
- נדרש ליכולות ניהוליות – יודע לארגן ,לחשוב ולהניע אנשים.
- יצירת gant.

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.
- חוקר את השוק ומבין את הצרכים ולאחר מכן בונה חזון לפתרון הבעיה.
- פיקוח על התכולה, הקצב ופתרון בעיות.
- כאשר הגרסה יוצאת תפקידו לעקוב ולקבל פידבקים מהמשתמשים.
- תפקיד אסטרטגי – קובע את הכיוון – לאן החברה הולכת.

מהנדס מערכת :

- אחראי טכנולוגית על הפיתוח.
- קובע את מיפוי הטכנולוגיות שבהם המערכת תשתמש.
- ראיה מערכתית – בקיא ומכיר את כל התהליכים וצריך להבין את הקשרים בינהם.
- יודע להסתכל על המערכת גם מבחינה חומרתית וגם מבחינה תיכנותית.

ארכיטקט תוכנה :

- מי שמעצב ומייצר את המערכת.
- High level design
- בוחר את תשתיות התוכנה.
- מחלק למודולים פיזיים ולוגים
- כללי העיצוב – איך מוסיפים רכיב חדש ,איך מאפיינים רכיב חדש .
- פונקציונאליות – מבין מה עושה כל חלק.
- הבנת הקשרים והתלויות בין כל חלקי המערכת.

ראש צוות :

- בונה את הצוות – בוחר את האנשים.
- תיאום עבודת הצוות.
- קביעת נהלים.
- אחראי על הlow level design.

מטמיע מערכת :

- התקנות, כתיבת מדריכים ,התקנה בפועל במחשבים שונים.
- תמיכה וליווי שוטפים לעובדים שנתקעים בתפעול המערכות השונות.
- משך זמן ההטמעה תלוי בגודל המערכת.

<u>תוצרי ניתוח מערכת והנדסת דרישות :</u>

הנדסת דרישות (RE) – דרישות, הכוונה לשירותים ואילוצים.

מה עושים עם דרישות ?

- אוספים אותם – תשאול בעלי עניין.
- ניתוח הדרישה – האם היא רלוונטית ?
- הכנסת מסמך דרישות.
- בדיקת אימות – נכונות ביחס לצרכים.

כיצד אוספים או מאתרים דרישות ?

- סיעור מוחות – אנשים מביעים את דעתם.
- יצירת demo - יצירת אב טיפוס שמדליק את הדמיון אצל אנשים – דבר שמניע דרישות ופיתוחים.
- אינטרנט, ראיונות ושאלונים.
- קבוצות מיקוד.

אפיון וסיווג דרישות :

<u>מאפייני דרישות :</u>

- יש דרישה כללית ויש דרישה ספציפית.
- הדרישות הן בסיסי לבקשת הצעות ולכן עליהם להיות פתוחות לשינויים.
- בניגוד לאמור לעיל הדרישות הם בסיס לחוזה ולכן עליהם להיות מוגדרות היטב וסגורות לשינויים וזה סוג שני של דרישות.
- דרישה טובה זו דרישה שהיא ברורה לגמרי ,עקבית ומתועדפת.

<u>סיווג דרישות – ישנם שלושה סיווגים :</u>

1. דרישות משתמש – משפטים בשפה פשוטה המלווים בדיאגרמות השירותים שהמערכת תספק - תיאור הדרישה היא עבור המשתמשים, כלומר המשתמש רואה ורוצה להשתמש במערכת הזו.

דרישות מערכת – מה הפונקציות של המערכת ? המטרה היא שונה – הדרישות הם בסיס לחוזה, פיתוח וכו' – איך המערכת תעבוד.

2. דרישות פונקציונליות – מה המערכת מספקת ? איך היא תגיב לקלט ומה הפלט שלו - לא איך היא עושה אלא מה היא עושה.

דרישות לא פונקציונאליות – כל מיני אילוצים , דברים שצריך לעשות – לדוגמא, רגולציה, אבטחה וסטנדרטים ,חלונות זמן לתגובה – כמה פעולות היא יכולה לקלוט ולעבד בזמן מסוים.

3. דרישה תפעולית – איך מתנהגת התוכנה, איך אני מדבר איתה ואיך היא מגיבה.

דרישת מידע – באיזה נתונים אנו מטפלים ? – איזה קלט נדרש לתהליך מסוים ומה הלקט בסופו.

דוגמאות :

תיעדוף דרישות – קיימים מודלים רבים ונבחר מודל בהתאם לסיטואציה אבל המיקום האידיאלי בו אנו נרצה להיות זה מינימום מאמץ ומקסימום תועלת.

תיעוד דרישות – את הדרישות נתעד כדי ליצור תקשורת טובה יותר.

הפורמט שבו אנו נשתמש זה מסמך SRS שמהווה בסיס לדיאגרמות בתקן UML.

<u>סוגי תבניות לייצוג דרישות :</u>

PRD – ייצוג של הפיצ'רים או החידושים מנקודת מבטו של המשתמש.

FRD – פונקציונאלי - משמש לאיך אני מתמודד טכנית עם דרישות מסוימות – וזו הפעם הראשונה שבו נדבר על טכנולוגיות מקצועיות.

SRS – התוכנית שבה נשתמש כדי לפתח – הבסיס לפיתוח מה שיהיה שם זה מה שיהיה איזה פיצ'רים צריכים להיות ומה ההתנהגות המצופה , השלד המרכזי איתו אנו יוצרים לדרך.

בסוגי התבניות לעיל ניתן להשתמש בהדרגה PRD -> FRD -> SRS .

- **שאלת מבחן** – מה אומרת עקומת בוהם ?
ת. עקומת בוהם באה להראות לנו כמה חשוב להגדיר דרישות מלכתחילה, בעצם היא מראה לנו את היחס בין הזמן שאיתרת שצריך משהו (דרישה) לבין כמות ההשקעה הנדרשת שצריך כדי לממש אותו.

כלומר ניתן לראות מהגרף כי עדיף לי להשקיע בהתחלה עוד זמן בשלב הדרישות, העיצוב והמימוש דבר שיוביל למיעוט תחזוקה וטיפול בבאגים בשלב מאוחר יותר.

- **שאלת מבחן –** מה יכול להיחשב כדרישה ?
ת. דרישה יכולה להיחשב כשירות או אילוץ , שירות ופיתוח או שירות ועיצוב.

- **שאלת מבחן –** מה זה התנהגות של מערכת ?
ת. התנהגות של מערכת זה הטרנספורמציה שהיא עושה לקלט – מגיעות בקשות והיא עושה את הטרנספורמציה.

<u>שיעור 4 :</u>

מקרי שימוש (תרחישים) USE CASE - ישנם שני דרכים להציג תרחישים ויזואלית (דיאגרמה) וטקסטוריאלית (תיאור מילולי מפורט).

כמובן שתרחישים כאלו משמשים כמודל (מודל ויזואלי).

תרגום דרישות למודלים :

<u>גישות :</u>

- גישת התהליכים – מה בעצם קורה פה ? בתהליכי משתמש ובתהליכי המערכת.
- גישת ישויות הנתונים – פירוק המערכת לחבילות (ישויות) של נתונים.
- גישות משולבות – גישה שמשלבת בין השנים הנ"ל.
- גישה תהליכית - data flow – השבלים שעובר הדאטה מהמקורות החיצוניים אל הפונקציות שעושות שינויים מסוימים ועד לאחסון שלהם.

<u>תרשים DFD – תרשים זרימה :</u>
מורכב מחמישה סמלים מרכזיים :

עיגול – פונקציה.

מלבן – ישות חיצונית .

מלבן מאורך – מאגר מידע.

משולש – יחידת זמן.

חץ – זרימה.

<u>USE CASE – ראה לעיל.</u>

<u>ERD -Entity relation model:</u>

מודל גישה ריאלציוני המבוסס נתונים.

שימוש בתרשימי ERD.

שימוש מודרני בתרשימי UML – בדיאגרמות CLASS ו ACTIVITY.

סוגי מודלים :

1. מודל קונספטואלי (מחלקתי) – מתאר איזה ישויות (קונספטים, מחלקות) יש לנו ומה היחסים ביניהם.

2. מודל לוגי – מדבר על הגישה הטכנולוגית בלי המימוש המלא החיסרון הוא שהמשמעות והתכלס הולך לאיבוד.

3. מודל פיסי - מדבר על איך אני מאחסן ושומר את הנתונים ממש – הקצאת נפחים , אופטימיזציה.

לרוב נתחיל בפיתוח המודל הקונספטואלי ולאחר מכן נוסיף ונתרגם מודל לוגי ומודל נתונים לצורך המימוש. את היחסים והקשר ביניהם נבטא באמצעות דיאגרמת R-E מבוססת קשרים.

- **שאלת מבחן –** תיאור תהליכים.
ת. Long – תהליכים שרצים בזמן ארוך ומטפל בדברים שרצים ברקע
Short - תהליכים שמטפלים בדברים אינטראקטיביים עכשויים.
תהליכי משתמש – תהליכים שהמשתמש עובר כדי להשיג את יעדו.
תהליך ישים – תהליך שמבטיח את השלמת כל הפעולות עד קבלת התוצאה.
- **שאלת מבחן –** מה זה Domain ?
ת. Domain – זה תחום עיסוק (לדוגמא : רפואה ,מסעדנות וכו').

<u>שיעור 5 :</u>

תהליכים (המטרה של כל התהליכים נועדו להתמודד עם אי וודאות) SDLC :

SDLC – תהליך שמארגן את הפעילות והשלבים, פיתוח המערכת ייעשה בתהליך עם מחזור חיים שמוגדר היטב, שהוא חלק ממתודולוגיה (ראה שאלת מבחן), בכדי שנוכל לצמצם כמה שיותר את האי וודאות.

<u>מודלים :</u>

- מודל מפל המים – (non object - אובייקט שלא עבר מודולציה נכונה ושהוא עושה הכל) בתהליך זה אנו עוברים שלבים בצורה מסודרת משלב לשלב בצורה הדרגתית (הוא מנוגד לnon object).
  יתרונות : יש תהליך מובנה וקל להשתלב פנימה , מספק יציבות לדרישות ,מתמודד ומתרכז באיכות ופחות עם אילוצים.
  חסרונות : יש להכיר את כל הדרישות מראש (דרישות בפרויקט הם דבר דינאמי), כיוון שזה בשלבים ההעברה לייצור קורה בבת אחת דבר שיכול לגרום להצפה של בעיות וקריסה, בתהליך עצמו אין פידבק מהלקוח ,שלב התחזוקה הוא הארוך ביותר – לא נפתח תוכנה שנה כדי להשתמש בה חצי שנה.
  מודל זה לא אידאלי ונשתמש בו כאשר הדרישות ברור מראש וידועות, שהטכנולוגיה קיימת ומוכנה.
- Code & fix (cowboy coding) - כתיבת קוד כמו "קאובוי" – ללא שום בקרה, סדר והיררכיה, בצורת ניסוי וטעיה.
  מתאים לצוות תוכנה קטן או מתכנת בודד.
  חסרונות : טיפול בדברים זמניים וחוסר התקדמות , מתכון לאסון – אחוזי הצלחה נמוכים.
- מודל אב טיפוס – בניית אב טיפוס כדי לראות שזה עובד ,לראות שזה מה שאנחנו רוצים ולקבל פידבקים ממשתמשי הקצה.
  יתרונות : ניתן לראות משהו מוחשי ,פידבק ממשתמשי קצה ,חידוד דרישות ,מאפשר גמישות ורואים התקדמות.
  חסרונות : ריצוי מידי, מבנה התהליך נפגע ,מתקדמים רק כדי "לראות בעיניים" ,חוסר גבולות.
  נשתמש כאשר אנחנו בהתחלה והדרישות אינן ברורות עוד.
- מודל V – סוג של מפל מים , כאשר מול כל שלב בפיתוח נעשה בדיקות , לדוגמא, כאשר אנחנו מפתחים מוצר רפואי הרגולציה דורשת המון בדיקות ולכן בסיום כל שלב בפיתוח לפני שאנו עוברים לשלב הבא נבצע בדיקות.
  יתרונות : כל שלב שעברנו מאומת ומתוקף ,נוכל תמיד להבין איפה אנחנו בפיתוח.
  חסרונות : מקשה על טיפול באירועים בו זמנית ,מקשה על התמודדות עם דרישות דינאמיות.
- מודלים הדרגתיים – Spiral incremental – וארציות נוספות של מודל מפעל המים – למשל ספירלה וכדו'.

פתרונות זריזים וגמישים – אנו חוזרים על עצמנו בצורה מחזורית כל הזמן יחד עם גמישות מסוימת כאשר אנו משלבים את כל הכלים בצוותי הפיתוח בתקשורת בריאה.

יש לא מעט מודלים לפתרונות זריזים וגמישים וטוב לקחת מה שטוב מכל אחד כאשר הקו המנחה הוא שהלקוח יודע את הדרישות שלו – "מה הוא רוצה".

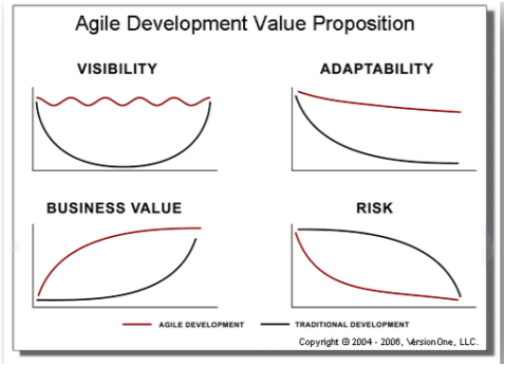
פתרונות זריזים וגמישים - ההבדלים

מסורתי	Agile
דגש על חיזוי והצגת תשובות מראש, קושי בהתמודדות עם שינויים	אדפטציה ויכולת התאמה לשינויים, מקבלים בברכה שינויים
מבוסס מסמכים	מבוסס אמון
מבוסס הגדרות תפקידים	מבוסס עבודת צוות
מונחה תכנון	מונחה בניית תוצרים
מבוסס תיאוריות	מבוסס פרקטיקה

Agility - שילב 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 ליצור.

נצטרך פתרון מבוסס agility כיוון שהוא פותר ומונע לא מעט את הסיבות לכישלון עליהם דיברנו בהרצאה 1 ,ניתן לראות זאת בגרפים הבאים :



<u>עקרונות אג'יליים :</u>

- הפצה ממושכת ומוקדמת ככל האפשר לתוכנה בעלת ערך.
- הפצת גרסאות תכופה.
- שוכנה שהיא עובדת זה הדבר הבסיסי.
- תקשורת – ישירה ללא מחסומים.
- אנשים מוכשרים בצוות ובעלי מוטיבציה.
- סומכים על צוות הפיתוח.
- עבודה בקצב סביר ולא 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.

<u>תהליך אג'ילי :</u>

- 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 נצמיד USE CASE) ותיעדופם , יצירת ארכיטקטורה בHigh 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 ושקיפותו ללקוח.

<u>פרקטיקות ופרוצדורות אג'יליות :</u>

- 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 מערכים 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 בדיקות 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.

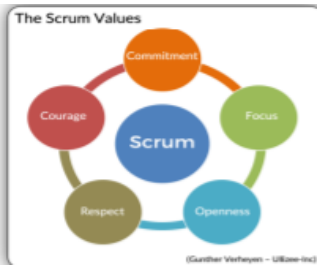
- **שאלת מבחן –** מה זה UML ?
ת. UML אלו סטנדרטים ליצירת תוצרים בשלבי הניתוח והעיצוב של מוצרי תוכנה.

- **שאלת מבחן –** מה ההבדל בין מתודה לגישה ?
ת. גישה בפיתוח מדבר על איזה שהוא שלב ללא יחס לשלבים הקודמים או הבאים, מתודולוגיה מדבר על כלל הפתרון והתהליך השלם.

- **שאלת מבחן -** מה הסיבה שאנו נותנים יחס מועדף והשקעה נוספת בשלב העיצוב של הקוד ?
ת. שלב התחזוקה – לא נרצה להגיע למצב בו לאחר שהתוכנה קיימת אנו תמיד מטפלים בבאגים ובתחזוקה (ראה עקומת בוהם).

- **שאלת מבחן –** מה זה Scope creep ?
ת. מין "חיידק" כזה שמשתלט על כל התהליך ובעצם דופק את כל הלוח זמנים בפיתוח – חוסר גבולות.

הרצאה 6 :



Scrum – גישת פיתוח המקדמת מיקוד בהירות ושקיפות לתכנון ומימוש פרויקטים.

מציע לנו מבנה של צוות ושל תהליך.

משומש בכל מקום החל מארגונים גדולים ועד סטרטאפים קטנים.

נקדם את הערכים הבאים : מוטיבציה ,עבודת צוות ותקשורת ,העצמת הפרט,

תרבות אירגונית והגברת קצב הפיתוח.

במודל זה יש את התפקידים הבאים :

- Scrum master– מעיין מנטור שמנתב את הצוות להצלחה אך לאו בהכרח בעל הכח לקבל החלטות.
- Product owner - ראה הרצאה 3.
- Developers
- Testers
- ההנהלה – כיוון שבמודל זה אין מנהל פרויקט והצוות מנהל את פעילותו אזי תפקידה של ההנהלה הוא ליצור מודל עסקי, לכבד את החוקים והשיטה ,לסייע בסילוק מכשולים , לאתגר את הצוות למצוינות.

scrum כולל בתוכו מחזורי תכנון , תכן, פיתוח ואינטגרציה.

עיצוב (תכן) מבנה תוכנה :

ארכיטקטורת מערכת מול ארכיטקטורת תוכנה -

ארכיטקטורת מערכת – מציגה מודל שמכיל קונספטים שמתאר גם מבנה וגם התנהגות הכוללות גם תתי מערכות ובנוסף 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 (מעיין עץ).

הארכיטקטורה מכילה בנוסף גם הגדרת 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 (שוב, לא נרצה לפתח מוצר שנה ולהשתמש בו חצי שנה) – מבנה ועיצוב נכון ימנעו את זה ויאפשרו אדפטציה מהירה לשינויים.

רמות עיצוב –

- High level design – מבט כולל והתעסקות נמוכה בפרטים (רמת גרנולציה נמוכה) .
- Low level design – התעסקות גבוהה בפרטים (רמת גרנולציה גבוהה).

אסטרטגיות עיצוב –

מהיכן נתחיל לעצב מהמבט הכולל ואז ירידה לפרטים או להפך ?

בדרך כלל נח יותר להתחיל מהמבט הכללי ומשם נפתח ונתעסק בפרטים.

מה קורה למערכת בה אין עיצוב טוב או שהעיצוב "מת" –

מבוסס על מאמרו של רוברט מרטין, קישור למאמר :

https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf

ארכיטקטורה ועיצוב טוב זה ניהול של תלויות – מי תלוי במי.

תכונות ומאפיינים למערכת לא בריאה :

- נוקשות – אין את היכולת להשתנות בקלות המערכת נוקשה ומגיבה רע לשינויים לאו דווקא באגים אלא יכול להתבטא גם בהשקעה גדולה בכדי לעשות שינויים קטנים.
- כאשר אני עושה שינוי קטן הכל מתפרק לי.
- חוסר יכולת לעשות שימוש חוזר (reuse) - 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 קטסטרופאלית ומלווה בבירוקרטיה – יצירת מורכבות מיותרת.

השינויים שגורמים לעיצוב שלי להתקלקל :

- יצירת תלות בין שני דברים שלא היו תלויים אחד בשני לפי כן.
- אי השמת מחיצות בין מודולים – כלומר נשים ממשק בכדי ליצור firewall שתהיה גדר שתמנע משינויים לבעבע פנימה.

- **שאלת מבחן –** מה תפקידו המרכזי של ה scrum master?

ת. לפנות את הדרך לכל מיני קשיים שמונעים מהצוות לעבוד ובעצם להוות מעיין מנטור ולא דווקא האדם שמקבל את ההחלטות.

ובהרחבה :

- **שאלת מבחן** - איך ניתן לדעת שמערכת דועכת ?

ת. בהקבלה לרפואה בה יש תסמינים לאדם דועך כגון דופק ,נשימות ולחץ דם גם כאן ישנם תסמינים למערכת דועכת – תקשורת לקויה ,אין תקשורת בין אישית ,קוד לא מסודר ויעיל ,שימוש בכלים לא יעילים כלומר עוד לפני שצוללים לפרטים ניתן לראות כי המערכת דועכת.

- **שאלת מבחן –** זכרו את המשפט שמי שאשם זה תמיד הלקוחות כי תמיד יהיה להם דרישות.
- **שאלת מבחן –** מה יכול להיחשב כ dependency firewall ?

ת. ממשק יכול להוות כ dependency firewall כיוון שהוא יכול להוות גדר וחציצה בין מודלים ובכך למנוע משינויים לבעבע פנימה ולהקריס מערכת.

שיעורים 7+8 :

עקרון SoC - (ראה הרצאה 1) , כאשר אנו מתבוננים באיזה שלם מסוים נעביר אותו דה קומפוזיציה (פירוק) וכאשר אנו מבצעים פירוק טוב זה נקרה מודלזציה.

מודול שהופרד בצורה נכונה מתחשב בגודל של המודול (רמת הרגולציה – פונקציה, מחלקה או חבילה וכו') ,מודול טוב מטפל בנושא אחד ובעל תקשורת טובה עם המודולים המקבילים.

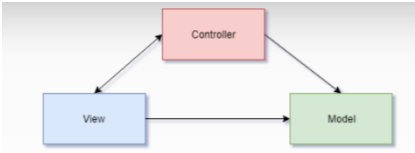
עיצוב תוכנה – הפרדת מרכיבי הפתרון למודולים וניהול התלות ביניהם.

Anti-pattern - אנטי-דפוס הוא תגובה נפוצה לבעיה חוזרת שבדרך כלל אינה יעילה.

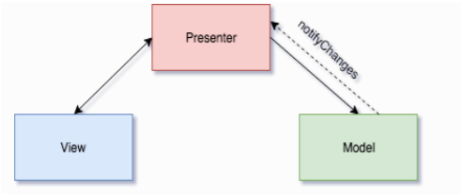
High level design :

תבניות ארכיטקטוניות נבחרות –

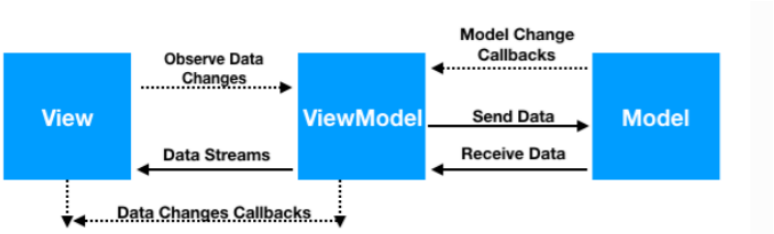
- **MVC** – Model- View -Controller – בנוי על תלויות כאשר ה controller תפקידו ל"האזין" ולקבל איזו התראה שצריך לעשות משהו, שינוי כלשהו, ה model הוא היחיד שיכול לבצע שינויים בנתונים, הview תלוי במודל ומציג את הפלט אך גם הוא יכול לעדכן את ה controller שקרה משהו.



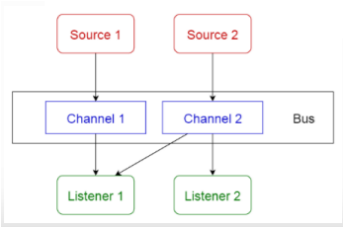
- **MVP** – Model-View-Presenter – הרעיון של מודל זה הוא לקחת את ה view ולהפוך אותו ל"קליפה ריקה" ונעביר את תקפידו לpresenter .
נסתכל על חלון מסוים ונעשה לו הפשטה ואת אותם הפשטות נגדיר כממשקים (ראה לעיל את תפקיד הממשקים).



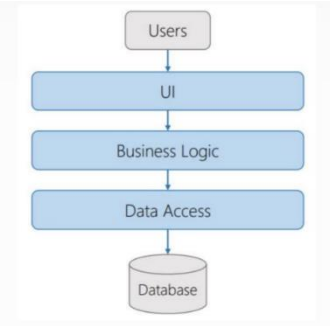
- **MVVM -** Model-View-ViewModel – מביאה לעולם את המושג ViewModel ,כאמור ה model אחראי על הטיפול בנתונים והview אחראי על הפלט אך לפעמים המודל לא מותאם לתצוגה ולכאן נכנס הViewModel שתפקידו לקחת דברים מתוך המודל ולהתאים אותם לתצוגה (כגון, תתי מחלקות או שדות מסויימים).
ViewModel - מודל נתונים שמותאם לview .



- **Event Bus** – אנו רגילים שאובייקטים מדברים ביניהם באמצעות signals דבר שיוצר בעיתיות במידה ואחד האובייקטים לא זמין לקבל signal לכן במקום שכולם ידברו אחד עם השני ניצור מכנה משותף הנקרא Bus.
כעת יהיו ערוצים שהם מה שמעניין אותי שיהוו כמקור ויהיו אובייקטים שיקבלו את הנוטפיקציה.

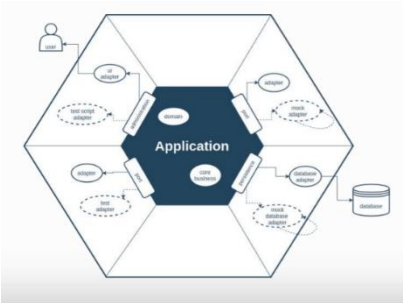


- **Multi Layer** – תבנית בעלת חלוקה לוגית, יש לנו מודול שמטפל בUI שהוא שכבה שהוא בעצמו מכיל MVP, MVC או MVVM כאשר בשכבה הבאה ישנה את כל הלוגיקה העסקית שלא תלויה בשכבת הUI והיא ליבת המערכת.
כל שכבה תלויה בשכבה שמעליה.
השכבה האחרונה הינה השכבה שמדברת עם העולם החיצון ולעיתים היא תכיל בתוכה SA – server agent שתפקידו להיות זה שמביא את הנתונים.



לעיתים במסגרת ה business logic ישנם גם business rules שבצירוף כמה תנאים מקבלים הטבה מסוימת לצורך העניין ,חוקים אלו דינמיים ומשתנים כל הזמן ולכן כמפתחים נשתמש ב business rules agent שנועד לאפשר לי לארגן חוקים בצורה שנכונה לתפעול.

- Port and Adapters –



- Pipes and filters –

IOC - Inversion of control - לא הקוד שלי מנהל את התהליך אלא אני מעביר את התהליך לקוד אחר שהוא מחליט מתי לקרוא לקוד שלי (קשור לMVP).

Microservices – לקיחת פונקציה אחת והפיכתו לmicroservices – מבצע פעולה אחת בצורה מאוד טובה.

יש לארכיטקטורה זו יש לא מעט יתרונות :

- משוחרר מכל אילוץ – ניתן לכתוב אותו בכל שפה שבא לי.
- הפיתוח לא דורש משאבים גבוהים (מפתח אחד או שניים).
- קל מאוד לשימוש ופיתוח.
- התאמה מאוד נוחה לארגון – בהתאם למה שארגון צריך אני מפתח.
- ממוקד – קל מאוד לתקן באגים.
- במצב של קריסה – לא הכל קורס וניתן להמשיך לתפקד.
- אי תלות מערכת מרכזית אחת – לדוגמא, במידה ואתה תלוי כמפתח בספריה אחד שהפסיקו לפתח אותה אתה בבעיה.

חסרונות :

- ביצוע כמות בדיקות גדולה יותר לכל סרוויס בו אתה משתמש.
- מעבר אינפורמציה חסום.
- שני צוותים יכולים לעשות אותו דבר כי לא כולם מודעים למה שכולם עושים.
- יותר מידי use cases .
- אין חוקים וכללים שתופסים לכל משך הפיתוח – דבר שיוצר עיצוב וחלוקה של קוד בצורה שגויה.

שיעור 9 :

<u>מושגים נוספים (לא בחומר) :</u>

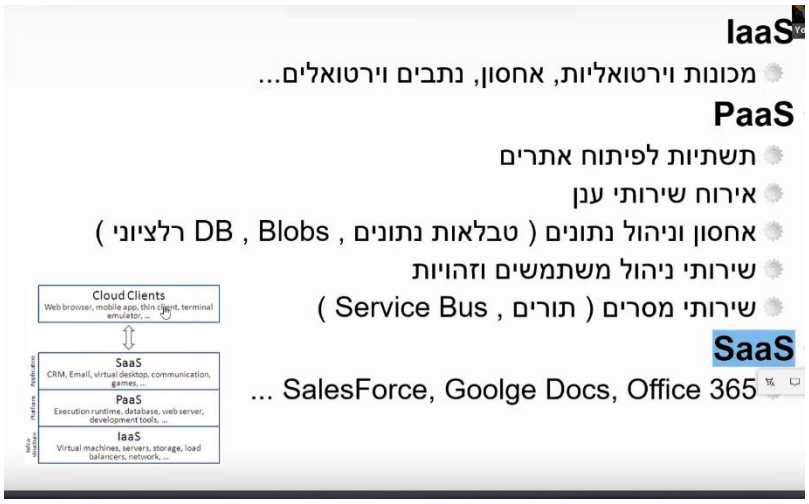
Software service – שירות זה קבלה של משהו כאשר אני לא הבעלים שלו זה השירות הקלאסי.

שירות תוכנה - יחידות הקצה שמהם אני בונה אפליקציה , אוסף של מחלקות (component) שנגישות למשתמשים בכל מקום, השירות כמובן לא חייב להיות בענן.

מתאפיינת בצימוד נמוך (כלומר, אני יכול להשתמש בפרט אחד בלי להביא את כל הקומפוננטה) כי אני לא יודע מי השתמש בה ומה הצרכים שלו.

Cloud Service – ענן – זה אוסף של שירותים בכל מיני רמות רגולציה שונות כאשר את השירות אני מקבל מהמאקרו או מהמיקרו.

המאקרו : SaaS, PaaS , IaaS. – שירות כתוכנה ,לדוגמא : PaaS– Platform as a service – מעיין ספריה המכילה מאות שירותים המוכנים לתת שירות בשלל נושאים כך שאין צורך שהשירות יהיה לוקאלי או פיזי בכתיבת הקוד שלי ויש לי גישה לשירות הזה מכל מקום.



מיקרו : web service . – קבלת שירות כחלקים מהתוכנה ,לדוגמא : AWS – AMAZOM WEB SERVICE שבו ניתן לקבל חלק מהענן כשירות ולא את כולו.

הבדלים בין המאקרו למיקרו (שאלת מבחן): במיקרו ב web service כל התחזוקה והאחריות היא עלי על המשתמש לעומת השירותים שהמאקרו נותן בו כל התחזוקה, העדכונים והתפעול הוא על נותן השירות.

SOA – Service oriented architecture - מחשוב ארגוני.

מהלך אופייני :

בשנות ה80 כאשר רצו לפתח אפליקציה השתמשו במונוליט – חבילת קוד שלמה שהפירוק שלה מהווה בעיה גדולה, לאחר מכן פיתחו שיטות עבודה בה הקוד מגיע בחלקים וכך יש יותר גמישות "לפרק ולהרכיב" בהתאם לצורכי הפיתוח בלי לשלם מחיר כבד מידי.

בשנות ה2000 היו נפוצות תבנית השכבות בו כל שכבה תלויה בשכבה שמעליה לדוגמא השכבה שמביאה את הנתונים מDB תלוי בשכבת הקוד שמעליה, בתבנית זו יש שלושה

אספקטים – פרזנטציה ,לוגיקה מרכזית וגישה לנתונים , שכבה זו הייתה נחשבת מאוד טובה.

עם השנים הפרידו את שכבת הפרזנטציה משני האספקטים האחרים שמשמים כעת כסרבר של האפליקציה, כעת יצרו קוד שיודע לדבר בHTTP (פרוטוקול תקשורת בשכבת האפליקציה) וכך יצרו ביזור של המערכת ממחשב אחד לשני מחשבים.

כעת לקחו את שני האספקטים (לוגיקה מרכזית , וגישה לנתונים) ופיצלו אותם – כך שיצא שישנם שני שרתים שאני יכול לנצל בצורה עצמאית בהתאם לצרכיי , כעת לסרברים האלו הרבה יותר קל לגשת והשירות שלהם יותר ממוקד.

לזה קראו SOA – עיצוב של מערכות מידע בתוך הארגון – לבסס את הפיתוחים שלו על סרוויסים וכל אפליקציה ספציפית לוקחת מה שהיא צריכה.

Microservices – לקיחת פונקציה אחת והפיכתו לmicroservices – מבצע פעולה אחת בצורה מאוד טובה.

יש לארכיטקטורה זו יש לא מעט יתרונות :

- משוחרר מכל אילוץ – ניתן לכתוב אותו בכל שפה שבא לי.
- הפיתוח לא דורש משאבים גבוהים (מפתח אחד או שניים).
- קל מאוד לשימוש ופיתוח.
- התאמה מאוד נוחה לארגון – בהתאם למה שארגון צריך אני מפתח.
- ממוקד – קל מאוד לתקן באגים.
- במצב של קריסה – לא הכל קורס וניתן להמשיך לתפקד.
- אי תלות מערכת מרכזית אחת – לדוגמא, במידה ואתה תלוי כמפתח בספריה אחד שהפסיקו לפתח אותה אתה בבעיה.

חסרונות :

- ביצוע כמות בדיקות גדולה יותר לכל סרוויס בו אתה משתמש.
- מעבר אינפורמציה חסום.
- שני צוותים יכולים לעשות אותו דבר כי לא כולם מודעים למה שכולם עושים.
- יותר מידי use cases .
- אין חוקים וכללים שתופסים לכל משך הפיתוח – דבר שיוצר עיצוב וחלוקה של קוד בצורה שגויה.

- מה ההבדל העקרוני בין XML , JSON ל CSV ?
ת. מה שדומה זה שכולם מעבירים נתונים בצורה טקסטוריאלית אך רמת הדחיסות שלהם שונה ורמת הביטחון שהכלים האלה נותנים בהעברת הנתונים שונה – כאשר XML מקסימלית וCSV מינימלית.
- מה ההבדלים בין web service ל Cloud service ?
ת. ב web service כל התחזוקה והאחריות היא עלי על מקבל השירות לעומת השירותים שCloud נותן בו כל התחזוקה, העדכונים והתפעול הוא על נותן השירות.