סיכום החומר: הנדסת תוכנה

מבוא:

מורכבות מהותית לעומת מורכבות מקרית:

שתי סוגי מורכבויות שצריך לפתור בפיתוח תוכנה.

- **מורכבות מקרית:** נוצרת עקב כתיבת טובה לא יעילה או כתיבת טעויות (כמו באגים). אם נפטרים ממנה, אפשר לשפר את הפרודוקטיביות.
  דרכים לצמצם את המורכבות המקרית: שימוש ב-high-level languages, שימוש במערכות מומחה (לדוגמה יצירת בדיקות אוטומטיות), מציאת באגים בתחילת התהליך, סביבות וכלים ושימוש בשיטות עבודה מתקדמות, כמו agile.

- **מורכבות מהותית:** המורכבות טבועה בבעיה, אשר נפתרת כשכותבים את התוכנה. המורכבות היא בנתונים, באלגוריתמים וביחסי הגומלין ביניהם. אין לה פתרונות קסם.
  התכונות המהותיות בתוכנה כוללות: סיבוכיות, תאימות, יכולת שינוי ונסתרות.
  דרכים לצמצם את המורכבות המהותית: קניית תוכנת מדף, חידוד הדרישות ובניית אבטיפוס, למצוא מהנדסי תוכנה טובים יותר, שינוי הגישה של פיתוח תוכנה - להוסיף למערכת יכולות לאט לאט.

הנדסת תוכנה:

הנדסת תוכנה מתייחסת למחזור חיים של הפיתוח, החל משלב הרעיון, דרך התיכנון, הפיתוח, הבדיקות והתחזוקה של המוצר.
מטרותיה:
- הקטנת המורכבות שבפיתוח תוכנה
- לשפר את אמינות התוכנה (פחות באגים)
- להקטין את עלויות התחזוקה

מחזור החיים של מערכת תוכנה:

התחלה: דרישת לקוח
סיום: אין יותר שימוש בתוכנה

1. שלב הייזום
2. שלב הדרישות
3. שלב הניתוח
4. שלב התוכן
5. שלב המימוש
6. שלב השילוב
7. שלב התחזוקה
8. פרישה

<u>שלב הייזום:</u>

פעולות בשלב הייזום:

1. הגדרת הלקוח/ות - אם זו מערכת בתוך ארגון, לזהות מי הם הלקוחות (או ההרשאות) השונים. ואם המערכת מחוץ לארגון, לזהות את הלקוחות הפוטנציאלים שיתעניינו במערכת.
השוק בו המוצר שלנו יפעל יכול להיות:
B2B (business to business), או B2C (business to customer)

2. מציאת מומחה יישום - נציג מטעם הלקוח, אשר ילווה את המערכת מנהלית ומקצועית. במערכות מידע, הוא בד"כ גורם מחוץ ליחידת המחשב. גם במערכת ללא ארגון צריך למצוא מומחה יישום שיהיה אחראי על שמירת האינטרסים של הלקוחות הפוטנציאליים.
מומחה היישום מרכז את דרישות הארגון מהמערכת, מלווה את המערכת משלב הייזום ועד הטמעתה בארגון, מהווה גורם מקשר בין הלקוחות השונים במקרה של התנגשויות בדרישות מהמערכת.

3. ניתוח מצב קיים, בדיקות מערכות דומות קיימות - לבדוק האם יש מערכת דומה בארגון הזה, ואם אין לבדוק כיצד מתנהלת המערכת הידנית, האם היה בעבר ניסיון להקים מערכת דומה (מה היו התוצאות? מי היה מעורב?), האם יש מערכת דומה בארגון אחר?, מה ה"היסטוריה" של הארגון? במערכות ללא ארגון, גם כן צריך לבדוק האם קיימות מערכות דומות, ומה ניתן ללמוד מהן וממצבן?

4. הגדרת מטרות ויעדים.
מטרות = תיאור <u>כללי</u> של התכונות שרוצים שיהיו במערכת. דברים שהמערכת תדע/ תבצע.
יעדים = פירוט המטרות ליעדים שאותם אפשר לכמת.
מודל SMART: מודל שנועד לוודא שהיעד מוגדר כמו שצריך.
ספציפי (Specific) - יעדים ממוקדים ככל הניתן, ולא כלליים מדי.
ניתן לכימות (Measurable) - יעדים אשר ניתן לכמת אותם.
בר השגה (Attainable) - יעדים ריאליים והגיוניים.
רלוונטי (Relevant) - יעדים משמעותיים המשרתים את הייעוד והאסטרטגיה של הארגון.
מוגבל בזמן (Time bound) - -יעדים התחומים בזמן, שיאפשרו מעקב קבוע.
המודל הזה לא טוב כי הוא מסורבל, מתיש וקשה לעמוד בו וליישם על כל המטרות שלנו.
מודל MT: מכיל בתוכו את SMART, כי אם המטרה מתוזמנת ומדידה, היא ספציפית. בנוגע ליעד בר השגה: מה שבר השגה לאחד לא בהכרח בר השגה לאחר, וגם הרלוונטיות שונה אצל כל אדם.

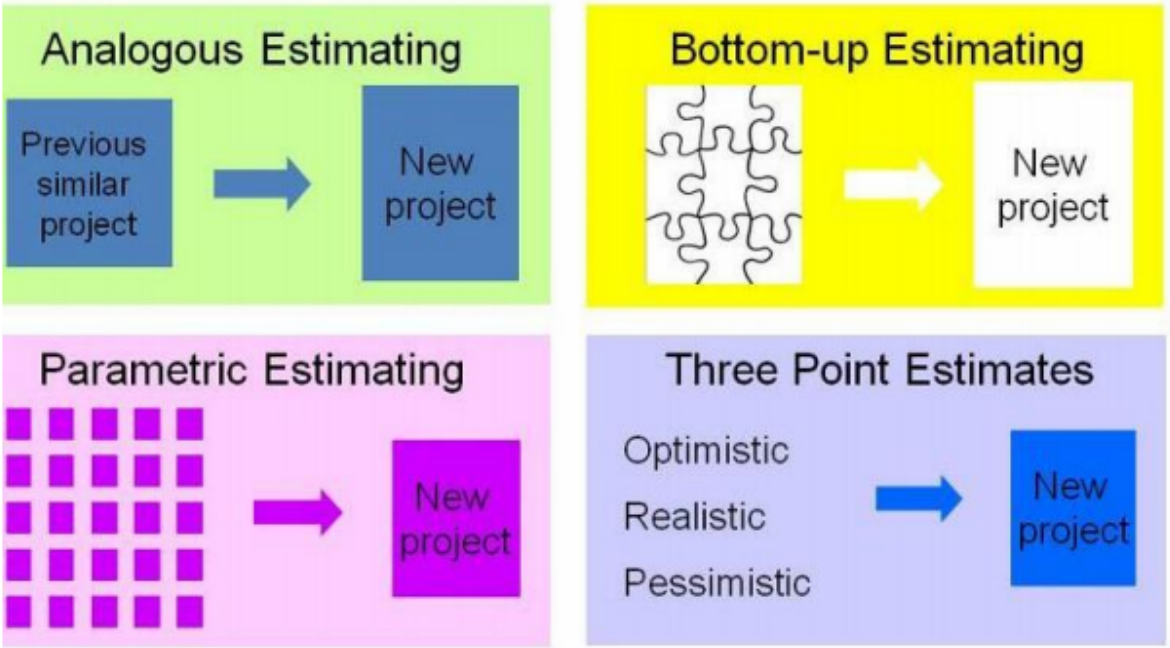
5. הגדרת אילוצים, מגבלות וסיכונים - ניתוח בעיות שעלולות לנבוע מאילוצים שונים (תפעוליים, טכנולוגיים, ארגוניים, זמן ומשאבים, תקציב).
סיכון הוא כל דבר שעלול להקטין את הסיכויים להצלחה בפרויקט.

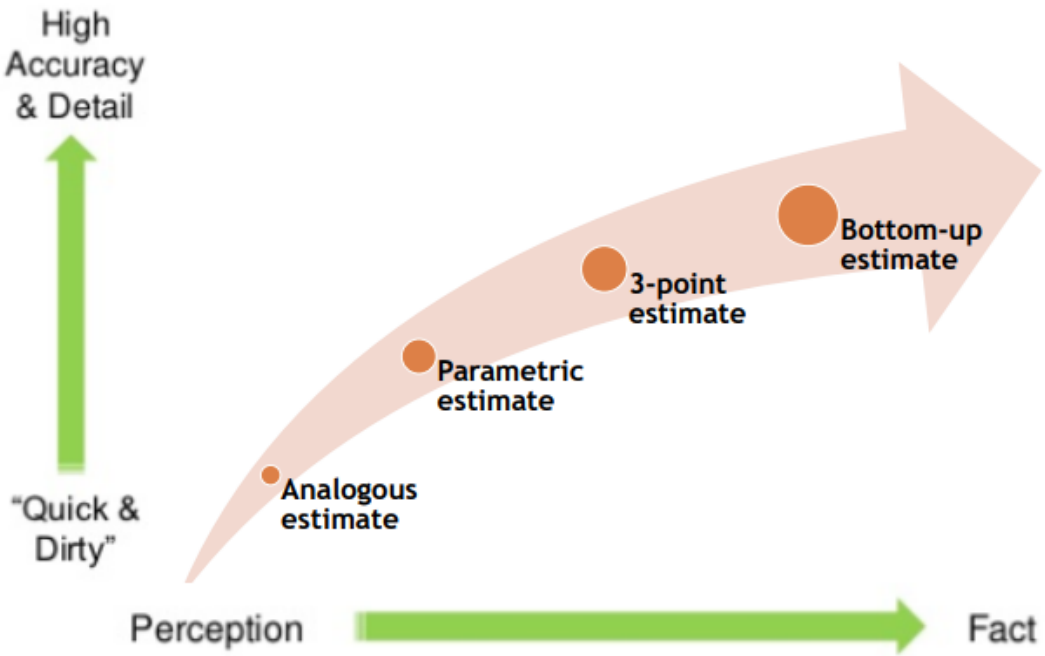
6. הגדרת תועלות וחסכונות.
- תועלות מוחשיות - ניתנות לכימות ומוגדרות במונחים כספיים.
- תועלות לא מוחשיות - לא ניתנות לכימות, מוגדרות במונחי תועלת.

תחומי התועלות יכולים להיות: בכ"א, בארגון ושיטות (ייעול תהליכים), בכלים וטכנולוגיה, ברמת המידע (מידע עדכני ותקין - שיפור האמינות).

7. בדיקת ישימות ועלות/ תועלת - שואלים את עצמנו האם צפויים קשיים/ בעיות/ מגבלות בשלב כלשהו.

שיטות לשערוך עלויות:





שיטות לבדיקת כדאיות כלכלית:

כלי המדידה הללו חשובים כי הם עוזרים למדוד את הכל בכסף, עוזרים בקבלת תקציב ותמיכה, הגדרת היקף ההשקעה במערכת, כלי לבחינת האלטרנטיבות וכו'.

ישנם אתגרים בשימוש בכלי מדידה שקשורים להגדרות הפרמטרים ולכימות.

לכסף יש ערך זמן - ככל שמקבלים אותו יותר מאוחר, הוא שווה פחות.

ענ"נ (NPV): ערך נוכחי נקי. משמש לניתוח הרווחיות של הפרויקט, כדי להחליט האם הוא כדאי.

מורכב מהשווי הנוכחי של סך כל ההכנסות העתידיות של השקעה מסוימת פחות ההוצאות הצפויות, ונקראית "היוון" (חישוב הערך הנוכחי של סכום שיתקבל בעתיד).

הNPV מהווה אינדיקציה האם כדאי להמשיך בפיתוח המערכת או לא.

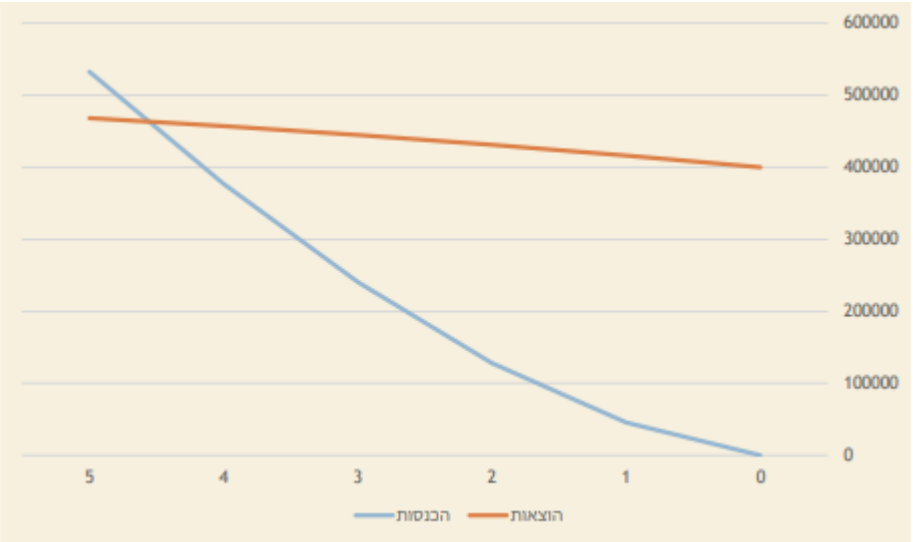
$$NPV(i,N) = \sum_{t=0}^{t=N} \frac{R_t}{(1+i)^t}$$

N - מס' השנים לחישוב
R_t- *הכנסות -הוצאות באותה שנה*
i - שיעור ההיוון
t - מתי תגיע ההכנסה

החזר ההשקעה (ROI): מציין את היחס בין הכסף והמשאבים שהושקעו בפיתוח המערכת לבין ההכנסות שנבעו מאותם מאמצים.

$$\text{\% החזר ההשקעה} = \frac{benefits-costs}{costs} * 100$$

ניתוח ההחזר (payback analysis): חשוב לקביעת משך הזמן שייקח למערכת להחזיר את השקעותיה. בד"כ מציגים את זה בתור גרף של העלויות והתועלות וכך אפשר לראות בבירור מתי התועלות עולות על העלויות.



8. גיבוש צוות התכנון של המערכת (ועדת היגוי) - ועדה מייעצת שהחברים בה הם בדרך כלל בעלי עניין או מומחים. הועדה הכוללת נציגים מתחומי ידע שונים.
הועדה מתגבשת בשלב הייזום וממשיכה בעבודתה עד העלאת המערכת לאוויר (לעיתים גם אחרי).
תפקידי ועדת ההיגוי:
   - קביעת מדיניות פיתוח.
   - מעקב ובקרה אחרי הלו"ז והעלויות של המערכת.
   - פתרון בעיות במהלך הפיתוח.
   - שינויים בשלב הפיתוח.
   - קבלת החלטות בשלבים השונים של פיתוח המערכת.
   - יצירת מעורבות ומחויבות אצל הלקוחות ואנשי הפיתוח.

9. קביעת לו"ז ומשאבים - קובעים לו"ז לשלבים השונים: סיום אפיון, סיום פיתוח, סיום בדיקות, מועד עלייה לאוויר. וכן קובעים משאבים: משאבי חומרה ותוכנה, משאבים אנושיים לאפיון, לפיתוח, לבדיקות, להרצה ולתחזוקה.

התוצר מכל הפעולות הללו הוא **מסמך ייזום**, אשר מועבר לאישור ועדת ההיגוי, שיכולה: לא לאשר אותו, להקפיא, להחזיר אותו לצורך ביצוע תיקונים או לתת אור ירוק ולאשר להמשיך לשלב הדרישות.

<u>שלב הדרישות:</u>

בעל עניין = כל גורם המושפע מפיתוח התוכנה או אחראי באופן כלשהו על פיתוחה. האינטרס שלהם יכול להיות לחיוב או לשלילה.
דוגמאות: משתמשים, לקוחות, מפתחים, מנהלים.

חשיבות מסמך הדרישות: חשוב להגדיר מסמך דרישות ברור, מכיוון שכל בעל עניין "רואה" את המערכת אחרת בעיני רוחו.

דרישה = תנאי או יכולת הדרושים על ידי בעלי העניין כדי לפתור בעיה או להשיג מטרה, או שיש למלא או למצוא פתרון כדי לספק התחייבות, תקן, מפרט או מסמכים רשמיים אחרים.
דרישות יכולות לתאר "מה" המערכת אמורה לעשות או "איך" המערכת אמורה לעבוד (רמת שירות).

שיטות למציאת הדרישות:
- ראיונות ושאלונים למשתמשים השונים.
- סיעור מוחות.
- יצירת demo.
- למידה ממערכת קיימת / למידה ממצב קיים.
- שימוש במודלים (UML ועוד).

בעיות אופייניות לשלב הגדרת הדרישות:

חשיבות הגדרת הדרישות:
- מקנה סיכוי גבוה יותר שתוצרים יענו על דרישות הלקוח.
- ממקדים את תהליכי האפיון והפיתוח של המערכת.
- משפר את היכולת לבצע שינויים במהירות תוך בקרה ובחינת השפעתם על דרישות אחרות.
- תיאום ציפיות בין ספק ללקוח על בסיס מנותח, מוסכם ומאושר.
- שיפור היכולת לבצע בקרה על התקדמות הפרויקט.
- שיפור היכולת לביצוע אמידת עלויות ע"י ניתוח של עלות תועלת עבור כל דרישה בנפרד או עבור מכלול דרישות.

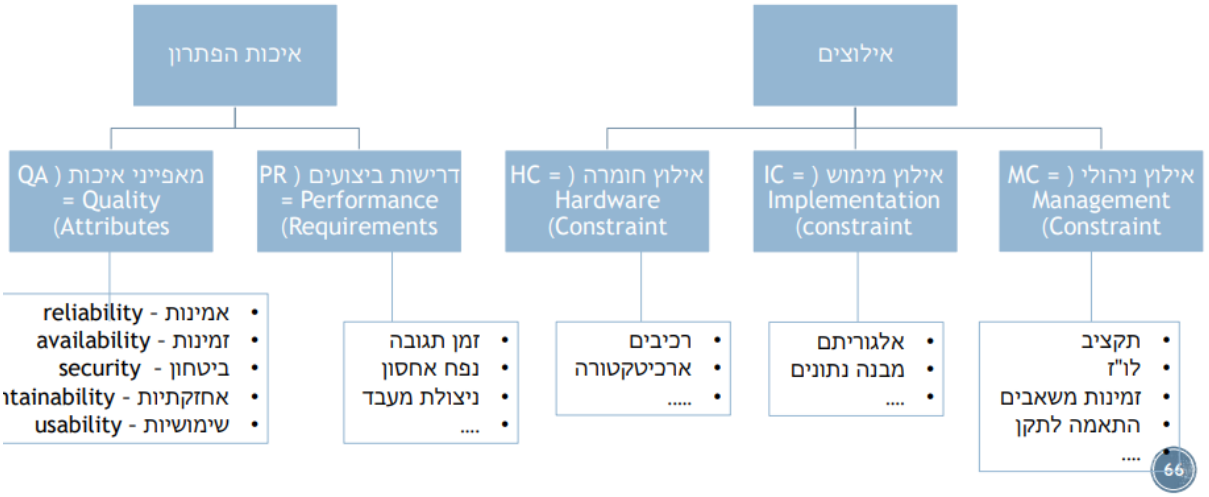
מוכח כי ככל שמוצאים שגיאות מוקדם יותר ומתקנים אותן, עלות התיקון נמוכה יותר (עקומת בוהם).

עודף דרישות:
זה לא טוב להגדיר יותר מידי דרישות, כי כל דרישה משמעה עלויות כספיות, סיבוכיות למערכת וניהול של הבאגים. לא נרצה לעשות את זה במקרה של דרישות איזוטריות או לא שימושיות.
לכן חשוב לשים לב להגדיר בשלב זה את מה <u>שצריך</u> ולא מה <u>שלא צריך</u>.

סוגי דרישות

- **דרישות פונקציונליות (עסקיות):**
מה המערכת אמורה לעשות/להגיב מנקודת המבט של המשתמש (פונקציות, שירותים).
מחולק לשני סוגים:
1. **דרישה תפעולית:** דרישה המתייחסת לתפעול, לאינטראקציה או להתנהגות של המוצר.
לדוגמא: פעולות, תרחישים, תגובות לאירועים וכו'
2. **דרישת מידע:** 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.

- **דרישות לא פונקציונליות (טכניות/ איכות הפתרון):**
דרישות המגדירות תכונות נוספות של הפתרון שצריכות להתמלא תוך כדי מילוי הדרישות הפונקציונליות או- דרישות ותנאים המגבילים את חופש בחירת כיווני הפתרון.
דוגמאות: זמני תגובה/ ביצוע, נפחי פעילות, אמינות, אבטחת מידע, אופני מימוש, תדירות ביצוע, עמידה בעומסים, שימושיות.
מתחלקות ל2:
1. **דרישות שקשורות לאילוצים: אילוץ ניהולי, אילוץ מימוש, אילוץ חומרה..**
2. **דרישות שקשורות לאיכות הפתרון: דרישות ביצועים, מאפייני איכות..**



דרישה היא איכותית אם היא:

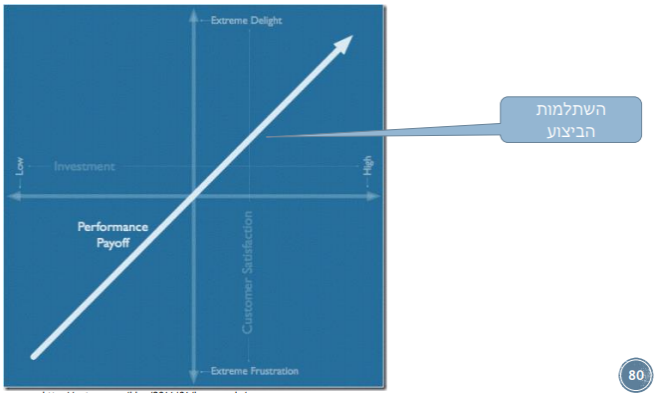
- Identified - בדידה , מזוהה חד ערכית , שייכות ברורה
- Understandable – מובנת (ברורה, מדויקת)- מנוסחת בשפת הלקוח
- Unambiguous - לא עמומה (חד משמעית)
- Complete – שלמה
- Necessary - הכרחית- בעלת תרומה משמעותית לשיפור תהליכי העבודה
- Consistent – עקבית (לא סותרת דרישות אחרות)
- Verifiable - ניתנת לבדיקה באמצעות מבחני קבלה
- Traceable - עקיבה (גם לדרישות ברמה גבוהה יותר וגם בהמשך האפיון)
- Prioritized - מתועדפת

מודל Kano:

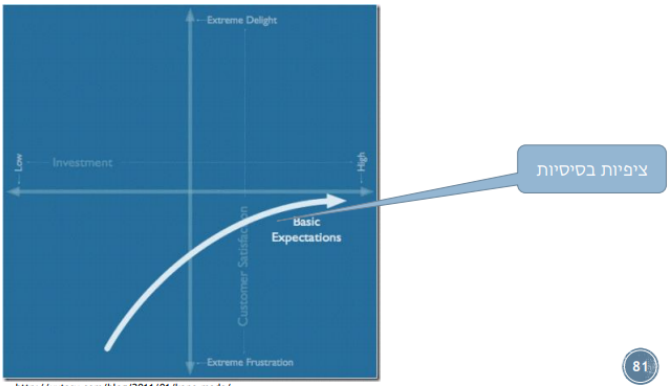
מודל המשמש לתעדוף דרישות במערכת.
מגדיר את הקשר בין רמת ההשקעה בדרישה לעומת שביעות רצון הלקוחות.
בעיני הלקוחות יש הבדלים מהותיים בין התכונות השונות של מוצר שתורמות לשביעות רצונם מהמערכת.

המודל מבדיל בין כמה דרישות:

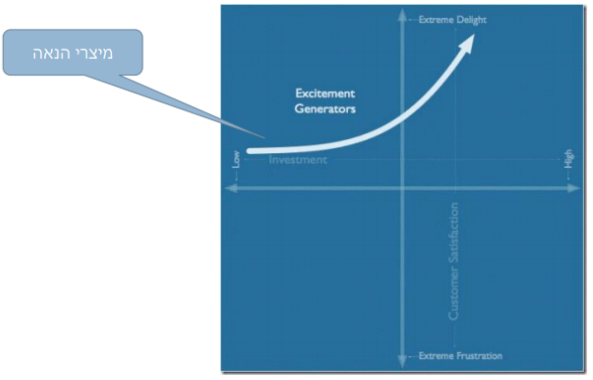
1. דרישות שמשתלם להשקיע בהן, כי יש קשר לינארי בין ההשקעה לבין שביעות הרצון של הלקוחות.



2. דרישות שהן ציפיות בסיסיות מהמערכת. אם נשקיע בהן הרבה, שביעות הרצון של הלקוחות לא תגדל משמעותית, כלומר הם לא יעריכו את זה יותר מידי, כי בעיניהם זוהי מהות המערכת.
לעומת זאת, אם לא נשקיע בהן (או נוותר עליהן), הלקוחות יהיו מאוד לא מרוצים.



3. דרישות שהן "מייצרי הנאה". כלומר כאלו שהלקוחות יכולים להיות מרוצים גם בלעדיהם, הם לא מצפים שהם יהיו חלק מהמערכת. אבל אם נשקיע בהן, שביעות רצון הלקוחות תגדל מאוד.
מסקנה: שווה להשקיע בהן אם יש זמן.



הצגת הדרישות:

מס' מזהה	תיאור	מקור	סוג	תת סוג	תיעדוף	הערות
1						
2	...	...	...	....	...	...
3						

- מזהה- ערך חד-חד ערכי. נשאר גם אם הדרישה מבוטלת
- תיאור- ברור, חד משמעי, נאמן למקור
- מקור- בעל העניין שהעלה את הדרישה או המסמך ממנו מוצתה הדרישה
- סוג- סוג הדרישה-פונקציונלית/ לא פונקציונלית
- תת סוג
- תיעדוף- ערך חד-חד ערכי המייצג את העדיפות
- הערות- רמת הקושי, עלות, דגשים לבדיקות

שלב הניתוח:

מטרת השלב: ניתוח הדרישות ומידולן.
- זיהוי הישויות הפועלות במערכת והכרת התכונות שלהן
- הכרת הקשרים בין הישויות

תוצר השלב: תרשימים שונים המתארים את המערכת.

דיאגרמות UML:

UML היא שפת סימולים גרפית וטקסטואלית לעיצוב מערכת מונחת עצמים. המודלים שכותבים בעזרתה מתאימים ללקוח, למשתמשים ולמפתחים. כל דיאגרמה מתארת היבט אחר של המערכת המתוכננת.

יתרונות:
- התמקדות בתחום הידע של המשתמש
- התאמה לדרך החשיבה של המשתמש
- תרשימים ידידותיים יותר למשתמש, למפתח ולמנהל הפרויקט
- חוסר תלות בטכנולוגיה

חסרונות:
- ריבוי מודלים וכלים עלולים לסבך את התהליך
- מורכבות הקשר בין המודלים
- קושי בשימוש במערכות מורכבות- כי קשה למדל אותן

חלוקות הדיאגרמות לסוגים:

חלוקה 1: דיאגרמות המתארות מבנה מול דיאגרמות המתארות התנהגות ואינטראקציה.

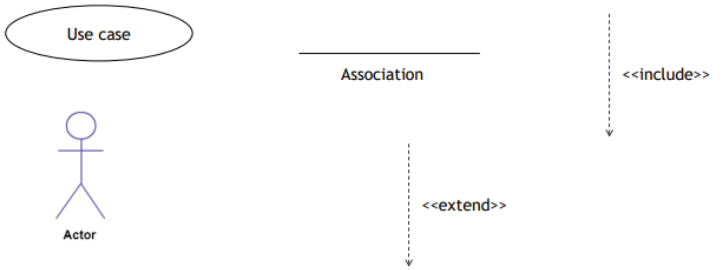
חלוקה 2: דיאגרמות סטטיות לעומת דיאגרמות דינאמיות.

דיאגרמות ספציפיות:

דיאגרמת Use-case
- מתארת את הפעולות שהמערכת מבצעת ויש להם תוצאות גלויות.
- מראה את האינטראקציה בין דברים מחוץ למערכת למערכת עצמה.
- מודל יכול להתייחס למערכת כולה או לחלקה.
- המודל לא מראה את סדר הדברים- לא סדר כרונולוגי ולא על ציר הזמן (דיאגרמה התנהגותית).
- לכל דיאגרמה מוסיפים תיעוד (documentation case use) הכולל את פירוט הפעולה, השחקנים, תדירות השימוש של הפונקציה, תנאים מקדימים להפעלת הפונקציה ודברים שקורים אחרי ביצוע הפונקציה.

השחקנים = ישות חיצונית/ מערכת אחרת/ כל אמצעי מחשוב (לא חלק מהמערכת) שיש לו קשר עם המערכת, ותפקיד כלשהו ביחס למערכת. מציירים אותם תמיד כבן אדם, גם אם הם לא.

שיטת הסימון:

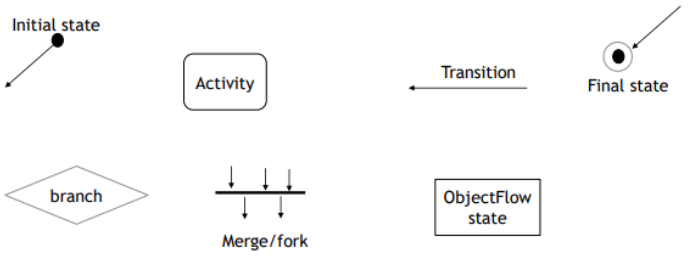


הקווים שמקשרים בין השחקנים ל-use case שהם יכולים לבצע, הם אינם חיצים, כלומר ללא כיוון. מדברים רק על פונקציות ולמי הן מקושרות, ולא על תהליך במערכת.
ירושה/ generalization מסומנת כקו עם משולש בקצה של מי שיורש ממישהו אחר.
קשרים בתוך המערכת:
חץ מקווקו עם משולש בקצה מסמן include. המשמעות היא שיש use-caseים בסיסיים שמשתמשים ב-use-case אחר.
חץ מקווקו עם חץ (>) בקצה מסמן extend. המשמעות היא שיש use-case שיצטרך להשתמש/ לבדוק משהו מ-use-case אחר במערכת. ה-use-case שעושים ממנו extend יכול להיות גם בשימוש ישיר משחקן חיצוני.
<u>הערה:</u> ה-use-caseים נבעו משלב הדרישות. דרישה יכולה גם להתפצל לשני use-cases שונים.

דיאגרמת Activity

- מתאר את הזרימה מ-activity ל-activity בתחום המערכת ובין השחקנים.
- כל פעילות ב-use case הופכת ל-Activity Diagram אחד.

<u>שיטת הסימון:</u>



דיאגרמת State-Machine

- האובייקט נמצא במרכז
- מתאר את זרימת האובייקט ממצב אחד למשנהו ואת המצבים השונים שבו האובייקט יכול להיות לאורך חיי המערכת.
- כל תרשים מתאר מחלקה אחת

<u>שיטת הסימון:</u>

דיאגרמת Class

- מתאר את מבנה המערכת ע"י מידול המחלקות, התכונות והפעולות.
- תכנית האב של המחלקות הדרושות לבניית התוכנה
- המתכנתים מפתחים את המערכת בהתבסס על מודל המחלקות שהוגדר
- זהו המודל הנפוץ ביותר ב-UML

<u>שיטת הסימון:</u>

כל מחלקה מסומנת בעזרת ריבוע שמחולק לשלושה חלקים: החלק העליון הוא שם המחלקה, החלק האמצעי מכיל את כל התכונות של המחלקה, והחלק התחתון מכיל את הפעולות שלה.

המשתנים:
בהתחלה כותבים את הרשאות הגישה שלהם

- Private -
- Public +
- Protected #
- Package ~

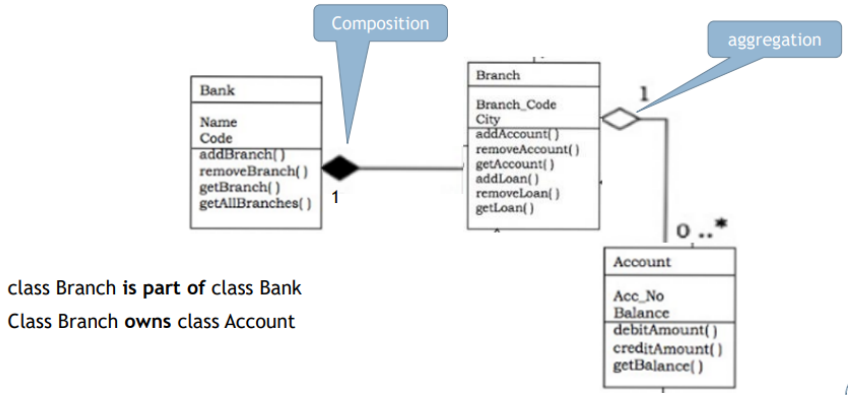
לאחר מכן אפשר להוסיף "/" שמתאר שהתכונה הזו נגזרת (מתכונה אחרת).
ואז כותבים את שמם, נקודותיים והסוג או הטיפוס שלהם, ולבסוף ניתן להוסיף "=" והערך הדיפולטיבי שלהם.
ניתן להוסיף לאחר טיפוס המשתנה את ערך ה-multiplicity שלו (באמצעות טווח ערכים אפשרי בסוגריים מרובעים). הערך הדיפולטיבי הוא 1, ושימוש ב-* משמעותה שהמשתנה ישמש לייצוג ערך אחד או יותר (עד אינסוף).

המתודות:
מתודה סטטית תסומן עם קו תחתון.
מתודה אבסטרקטית תסומן בכתב אלכסוני (*this way*), וכך גם המחלקה האבסטרקטית.
בהתחלה כותבים את שם המתודה, ואז בתוך סוגריים את הפרמטרים, לאחר מכן נקודותיים ואז סוג ערך חוזר, ואז סוגריים מסולסלים עם אילוצים.
הפרמטרים: תחילה מגדירים in/out/inout/return, אח"כ את שם הפרמטר, נקודותיים ואז הסוג שלו, וניתן להוסיף בסוגריים מרובעים את ה-multiplicity שלו.
האילוצים: מחולקים לשניים - preconditions = תנאים שצריכים להתקיים לפני שמפעילים את המתודה, ו-postconditions = תנאים שצריכים להתקיים לאחר שהמתודה מסיימת את פעולתה (בד"כ ביטויים לוגיים, עם ערך true או false).

תיאור תקלות:
כאשר בקריאה להפעלת מתודה יש סכנה שתתרחש תקלה (ייזרק exception) מקובל להוסיף note ובו תיאור ה- exception שעלול להיזרק ולחברו בקו מקווקו למתודה שבה מדובר.

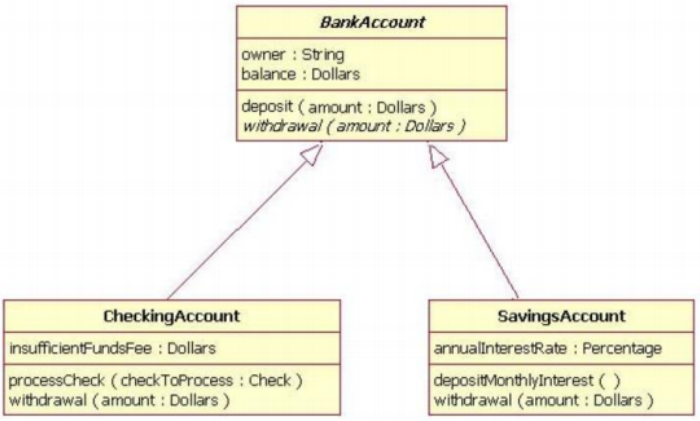
קשרים בין מחלקות:

1. תלות (Dependency) = המחלקה ממנה יוצא החץ עושה שימוש במחלקה עליה הוא מצביע. השימוש יכול להיות בכל מיני דרכים...
2. אסוציאטיבי (Association) = מסומן בקו. משמעותו שמחלקה אחת לפחות מחזיקה ברפרנס של האובייקט מהמחלקה השנייה (לפרק זמן קצר או לכל חייה). אפשר להוסיף חץ שאומר איפה נמצא המפתח של הקשר ביניהם, וגם multiplicity ליד הקו שמתאר את הכמות האפשרית והסבר קצר.
3. בעלות (Aggregation) = מסומן כמעוין לבן, ומשמעותו שמחלקה אחת היא בבעלות מחלקה אחרת.
4. הכלה (Composition) = מסומן כמעוין שחור, ומשמעותו שמחלקה אחת מוכלת במחלקה אחרת.

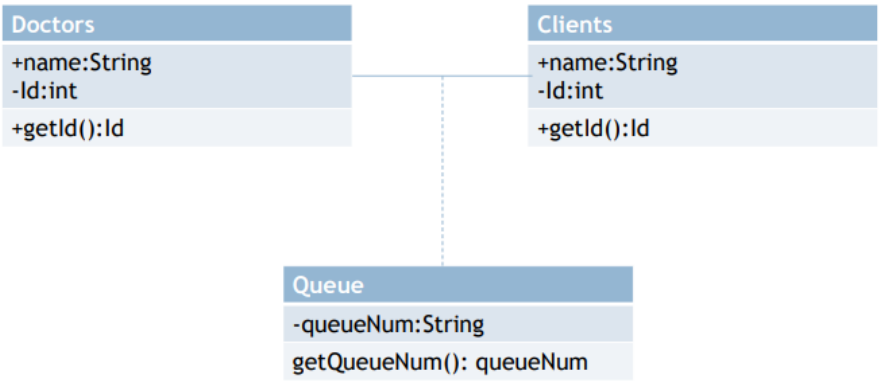


סימונים נוספים:

- הורשה מסומנת ע"י חץ עם משולש לבן בקצה המצביע על המחלקה היורשת.



- ניתן לתאר קשר בין מחלקות באמצעות class association, מתאר את מהות הקשר. קו מקווקו.

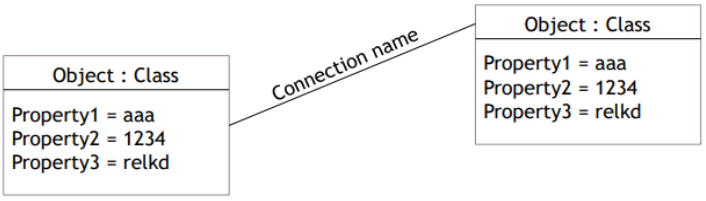


- ניתן לתאר ממשקים ומחלקות שמיישמות אותם, באמצעות קו מקווקו עם חץ משולש לבן בקצה של הממשק.

דיאגרמת Object

מתארת מופע של מודל המחלקות, בדומה לתרחישים אמיתיים שעל בסיסם בונים את המערכת.
היא דיאגרמה מבנית, וסטטית (מתארת את מבנה המערכת בזמן מסוים, ספציפי).
השימוש במודל האובייקטים דומה למודל המחלקות רק שהם מאפשרים בניית אב טיפוס של המערכת מנקודת מבט מעשית.
על הקו המחבר בין שתי מחלקות ניתן לכתוב את שם הקשר.

<u>שיטת הסימון:</u>



הבדלים בין Object Diagram לבין Class Diagram:

	Class diagram	Object diagram
מבנה	מכיל 3 חלקים: שם, רשימת תכונות ורשימת פעולות	מכיל 2 חלקים: שם ורשימת מאפיינים
שם המחלקה	שם המחלקה עומד בפני עצמו בחלק של שם המחלקה	הפורמט מורכב משם האובייקט + נקודותיים + שם המחלקה (Tom:Employee)
רשימת התכונות	מגדיר את התכונות של המחלקה	מגדיר את הערך הנוכחי של כל תכונה
רשימת הפעולות	כלולות	לא כלולות
קשרים	מוגדר הקשר בין מחלקות- שם הקשר וכמות הקשר.	מוגדר שם הקשר, אבל לא הכמות (לא רלוונטי כשמדברים על ישות בודדת)

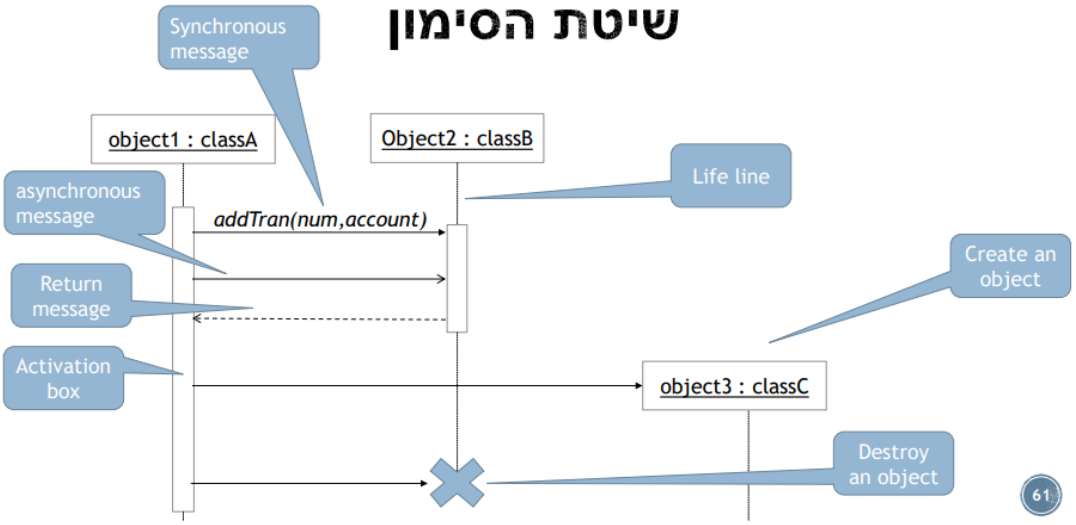
דיאגרמת Sequence (רצף)

ממחיש את סדר הפעילויות ברצף של הזמן.

נכתב בצורה של תרשים דו מימדי: על ציר ה-X נמצאים האובייקטים, ועל ציר ה-Y ממוקמות ההודעות שהאובייקטים האלה שולחים.

לכל פונקציונליות מכינים תרשים רצף נפרד.

<u>שיטת הסימון:</u>



הבדלים בין Activity Diagram לבין Sequence Diagram:

Sequence	Activity
דיאגרמת אינטראקציה	דיאגרמת התנהגות
מתמקד בהודעות שמועברות בין ישויות במערכת.	מתמקד בתהליך של האובייקטים ונותן דגש לרצף ולתנאים שיש בתהליך
נותן דגש לסדר ביצוע הפעולות	סדר ביצוע הפעולות לא מודגש
התהליך יותר מדויק ומפורט	כללי יותר ופחות מדויק

דילגתי: ERD, כל העניין של אנדרואיד ופיירבייס

שלב המימוש:

חלק א: שלב הפיתוח

עקרון ה-KISS:
Keep It Simple, Stupid
הרעיון הוא שמוטב להשאיר את הקוד כמה שיותר פשוט. מהנדסי תוכנה נוטים לסבך את הקוד, אז מנסים לא לסבך את הקוד בעת הכתיבה.

עקרונות ה-SOLID:
חמישה עקרונות בסיסיים בעיצוב מונחה עצמים, שכאשר הם מיושמים יחדיו בפיתוח של מערכת תוכנה, סביר שהיא תהיה יותר קלה לתחזוק והרחבה לאורך הזמן. העקרונות הם גנריים, ומתאימים לכל מערכת.

S	SRP: Single Responsibility Principle	עקרון האחריות היחידה
O	OCP: Open/Closed Principle	עקרון ה "פתוח/סגור"
L	LSP: Liskov substitution principle	עקרון ההחלפה של ליסקוב
I	ISP: Interface Segregation Principle	עקרון הפרדת הממשקים
D	DIP: Dependency Inversion Principle	עקרון היפוך התלות

חלק ב: שלב הבדיקות

הגדרה: בדיקות
מכלול תהליכים, שיטות, פעילויות וכלים על מנת לבדוק שהמוצר שלנו מתאים למה שרצינו: דרישות הלקוח, דרישות פונקציונליות ולא פונקציונליות, תקנים ודברים אחרים שמחייבים אותנו..
לבדיקות יש 2 פנים:

1. Validation- האם אנחנו מפתחים נכון את המוצר, כלומר ללא באגים?
2. Verification- האם אנחנו מפתחים את מה שנדרשנו לפתח? כלומר לא מספיק שהקוד שלנו ללא באגים, אלא צריך שהוא גם יעשה את מה שהוא אמור לעשות (לפי הדרישות).

למה צריך לבדוק?

- כדי לראות שכל הדרישות הפונקציונליות מתקיימות
- לבדוק שאין שגיאות בקוד (איכות)
- לבדוק שהכל מתבצע בצורה טובה, מבחינת עמידה בדרישות ביצועים, בעומסים ונפחים (דרישות לא פונקציונליות).

מטרה עקיפה: למנוע שגיאות עתידיות (מוצאים שגיאה אחרת "על הדרך", או שמתקנים משהו כללי שמתקן גם את השגיאה הספציפית הזו, אבל גם יעזור לגבי שגיאות עתידיות).

מי בודק?
כולם: גם מתכנתים שבודקים את הקוד של עצמם בסביבת הפיתוח, גם בודקי תוכנה בסביבת הבדיקות, וגם הלקוח בודק את התוכנה (שהיא עושה מה שהוא רוצה) בסביבת הpreprod.

שיטות לבדיקות תוכנה:

1. **בדיקות קופסה לבנה**
בבדיקת הקוד מכירים את הקוד שנמצא מולנו, יודעים מה כל פונקציה אמורה לעשות כיצד זה משתלב. לכן בד"כ עושים אותן המתכנתים.
דוגמה: בדיקות יחידה, בדיקת code review בערך.

הדרישה מהבודק: להכיר את הקוד ואת שפת התכנות. אבל לא צריך לדעת את דרישות המערכת.
כלומר רצוי שיידע, אך אפשר לעבור בהצלחה בדיקות קופסה לבנה גם מבלי להכיר את הדרישות, כי בודקים רק את נכונות הקוד עצמו, את מה שכתוב, ולא את הלוגיקה שעומדת מאחוריו.

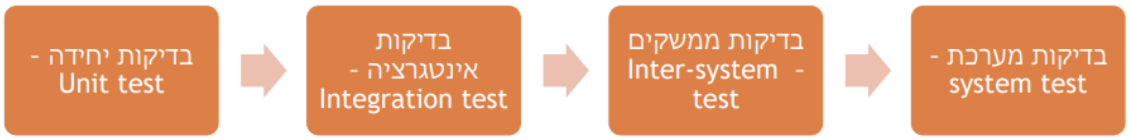
2. **בדיקות קופסה שחורה**
מכניסים קלט למערכת, והיא מוציאה פלט מסוים. לא יודעים מה קורה בפנים, ב"קופסה השחורה".
הדבר היחיד שנבדק כאן הוא שקיבלנו את התוצאה הנכונה בסוף.
דוגמה: בדיקות קבלה. צוות הבדיקות עושה זאת בד"כ, הוא לא צריך להכיר את הקוד עצמו, אלא רק את האפיון. מה צריך לקבל בתור תוצאה בכל פעם.

3. **בדיקות קופסה אפורה**
משהו בין קופסה שחורה לקופסה לבנה - שילוב של שניהם.
דוגמה: בדיקות אינטגרציה. לדוגמה להכניס נתונים ישנים לקוד חדש ולראות אם זה עובד כראוי, או בדיקה של שירות מסוים ששולחים לו נתונים והוא מפעיל את הפונקציונליות שלו.
עושים זאת בד"כ בודקים, אבל שיש להם ידע מסוים בתכנות (כמו html, db).
הבדיקה נעשית באמצעות "משחק" - שינוי משהן בעיצוב או בdb, ובדיקה שהכל עדיין עובד.
זו קופסה לבנה מכיוון שנוגעים במשהו בפנים, אבל הבדיקה עצמה נעשית מבלי להסתכל על הקוד.

סוגים עיקריים לבדיקות:

1. **בדיקות מסירה**
בדיקות הנעשות בתהליך הפיתוח, לפני שהלקוח קיבל את המוצר לבדיקה.
הבדיקות הללו נעשות באחריות צוות הפיתוח והבדיקות.

<u>שלבים:</u>



* יחידה: בודקים שבתוך המערכת שלנו זה עובד כמו שצריך. בודקים יחידה קטנה בקוד. הבדיקות נעשות בכל פעם שמסיימים לכתוב את היחידה הזו בקוד, או שמשהו השתנה בה.
* אינטגרציה: בודקים תת תהליך מסוים. לוקחים כמה units, ובודקים איך זה עובד ביחד - התהליך בצורה רציפה. מבצעים את הבדיקה בכל כניסה של תת תהליך/ מודולו חדש למערכת. הבדיקה נעשית מול מסמך האפיון, בו מוגדר איך צריך להיות כל תהליך, ומה הוא צריך לעשות.
* ממשקים: בודקים את הקשר בין ממשקים בתוך האפליקציה שלנו, או עם ממשקים חיצוניים. מבצעים את הבדיקה כשכבר יש אפליקציה מוכנה יחסית.
* מערכת: בודקים המון סוגים שונים של בדיקות שונות - עיצוב, עומסים, יעילות וכדומה. לא תמיד עושים את כולם, זה תלוי ברצון של הלקוח וכו'. מתבצעות לפני המסירה ללקוח - בדיקות סופיות.

<u>הערות:</u> בדיקות אינטגרציה נעשות ע"י הבודקים, ונעשות מול סביבת הפיתוח (בודקים שהקוד לא נופל), וכל השאר נעשות ע"י צוות הבודקים, ומול מסמך האפיון - בו מפורטים התהליכים, מה הם צריכים לעשות, מוגדרים עומסים וכו'.
בדיקות אינטגרציה וממשקים נעשות בסביבת הבדיקות, ובדיקות מערכת נעשות או בסביבת בדיקות, או בסביבת pre-prod שהיא יציבה יותר.

2. **בדיקות קבלה**
בדיקות כוללות המתבצעות בידי הלקוח/ המשתמש. בודק שכל מה שהוא רוצה נמצא במערכת.
קורות כמה שיותר קרוב לסביבת הפרודקשן (נקרא pre-prodsction).

- עושים את שניהם, בזה אחר זה.

<u>רמת חומרה של באגים:</u> עד כמה בדיקה היא חמורה, כך נדע במה יותר דחוף לטפל

חומרה	שם	תיאור
5	Highly Critical	-מונעת ביצוע של רכיבים מרכזיים במערכת -מסכנת את הבטיחות או הביטחון -תקלה מהותית בנושא שהוגדר בעל חשיבות גבוהה
4	Critical	-משפיעה לרעה על ביצוע של רכיבים מרכזיים במערכת -בעיית משאבים או ביצועים המשפיעה על תיפקוד המערכת -כל תקלה שברור שהמשתמש הסביר לא יסכים לקבל
3	Major	-תקלה בולטת במיוחד לעיני הלקוח (גם אם בפועל ההשפעה שלה פחותה) -בעיית משאבים או ביצועים חמורה (אך עדיין מאפשרת שימוש)
2	Medium	-תיפקוד של המערכת אינו פועל, אבל יש דרך סבירה לעקוף זאת. -סדר פעולות הנדרש מהמשתמש אינו טוב: לא אינטואיטיבי, מסורבל, מבלבל וכו' -תקלות בממשק הגראפי לא אינטואיטיבי, טקסט או נתונים קשים לקריאה או חתוכים, שגיאות כתיב וכו'
1	Minor	-תקלות קלות אחרות, בממשק הגראפי או בהיבטים אחרים

מסמכים בעולם הבדיקות:

STP- תכנון הבדיקות (כותב מנהל הבדיקות, ראש צוות הבודקים). מתארים מה נבדוק, מתי, ע"י מי ... נכתב במקביל לזמן בו כותבים את האפיון. חייב להסתיים לפני שמתחילות הבדיקות בפועל, ולפני הSTD.
מיועד לכולם - בודקים, מתכנתים ולקוחות.

STD- תיאור הבדיקות שנעשה (כותבים מתכנני הבדיקות). עבור תהליך מסוים מה הבדיקות שנעשה ... מה סדר הפעולות וכדומה. זה מסמך שנכתב בשיתוף של כל מיני מסמכים שכותבים כל מיני בודקים.
מיועד לבודקי התוכנה, כדי שיפעלו לפיו.
אמורים לכתוב לפני שמתחילים את הבדיקות (לפחות הSTD של תהליך מסוים, לפני שמתחילים לבדוק את התהליך הזה). בד"כ נכתב במקביל לפיתוח הקוד.

STR- הדוח בו מתארים את מצב הבדיקות שנעשו. כלומר: את הבאגים, מה נפתר, האם המערכת מוכנה לעלייה לאוויר או לא (כותב מנהל הבדיקות). נכתב בסוף תהליך הבדיקות (כולן או בתום סבב בדיקות מסוים).
מיועד למנהלי המערכת, לנציגי הלקוחות.

אופי הבדיקות:

ניתן לבצע את הבדיקות בפועל בשתי דרכים:

- בדיקות אוטומטיות: בתוך מערכת כלשהי. עושים דברים שבלוניים שחבל לבזבז עליהם את הזמן ידנית, אז עושים זאת במהירות ויעילות.
- בדיקות ידניות: כותבים בדיקות בצורה ידנית, בד"כ לדברים שאי אפשר לבדוק אוטומטית.

בכל מערכת עושים את שני סוגי הבדיקות הללו, וכך ממקסמים את היתרונות משני הסוגים.