

הנדסת תוכנה - סיכום חומר

מה זה תוכנה?

הוראות כדי שהמשתמש ייצור אינטראקציה עם המחשב לביצוע מטלות.

מורכבות מקרית:

המורכבות הנוצרת עקב כתיבת תוכנה לא יעילה או עם טעויות, אם נוכל להפטר מהמורכבות נוכל לשפר את הפרודוקטיביות.

דרכים לצמצום:

- שימוש בשפה high level משפר עד פי 5 בפרודוקטיביות
- שימוש במערכות מומחה- לדוגמא יצירת בדיקות אוטומטיות
- תכנות אוטומטי
- מציאת באגים בתחילת התהליך
- סביבות וכלים
- שימוש בשיטות עבודה מתקדמות

מורכבות מהותית:

לב התוכנה- הנתונים האלגוריתם ויחסי הגומלין ביניהם, למורכבות זו אין פתרון קסם היא טבועה בבעיה, הבעיה נפתרת שכותבים את התוכנה.

תכונות מהותיות- סיבוכיות(שלא יהיה מסובך מידי לביצוע),תאימות(שיפור מוצר קיים),ניתן לשינוי,בלתי נראה(מאחורי הקלעים).

דרכים לצמצום:

- קניית תוכנת מדף
- חידוד הדרישות ובניית אב טיפוס
- למצוא מהנדסי תוכנה טובים יותר
- מתחילים בקטן ומתפתחים.

שלב הייזום:

פעולות בשלב היזום:

- הגדרת הלקוח/ות.
- מציאת מומחה יישום
- ניתוח מצב קיים
- בדיקות האם מערכות דומות קיימות
- הגדרת מטרות ויעדים
- הגדרת אילוצים
- מגבלות וסיכונים
- הגדרת תועלות וחסכונות
- בדיקת ישימות ועלות/ תועלת
- גיבוש צוות התכנון של המערכת (ועדת היגוי)
- קביעת לו"ז ומשאבים

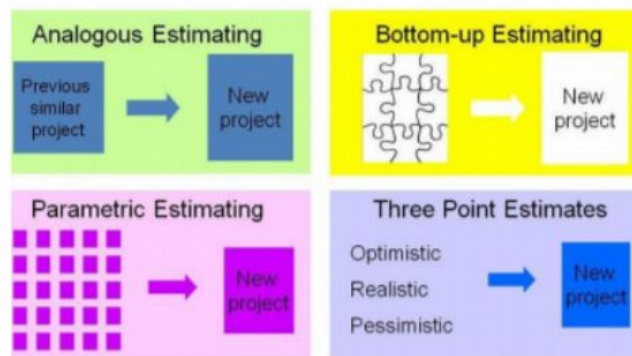
מומחה יישום:

בשלב הייזום הלקוח ממנה נציג מטעמו אשר ילווה את המערכת מנהלית ומקצועית, הוא ירכז אצלו את כל הדרישות וילוה את המערכת משלב הייזום ועד הטמעת המערכת בארגון, ויהיה גורם מקשר בין הגורמים השונים במקרה של התנגשות בדרישות.

במערכות מידע המומחה בדר"כ מחוץ ליחידת המחשב ובמערכות תשתית המומחה מתוך יחידת המחשב. גם המערכת ללא ארגון יש למצוא מומחה יישום שיהיה אחראי על שמירת האינטרסים של הלקוחות.

שיטות לשערוך עלויות:

שיטות לשערוך עלויות



Analogous - הערכה על בסיס פרויקט קוד

Parametric - חלוקת הפרויקט לחלקים וניתן הערכה בהסתמך על כל חלק

Bottom-up - חלוקה לחתיכות הכי קטנות ונותנת הערכה מאוד מדויקת - מאוד מסובכת

Three-Point : בודקים כל משימה על בסיס שלושה פרמטים: אופטימית, ראלית, פסימית

1. *Analogous* - לוקחים פרויקט דומה ומערכים על סמך העלות שלו, זה דומה אבל זה גם מאוד לא דומה ולכן זה בעייתי להגיע להערכה טובה, אבל זה יהיה מהיר.
2. *Parametric* - חלוקה לחלקים וכל חלק מערכים לעומת פרויקט דומה, גם לא נותן הערכה הכי טובה אבל מהיר.
3. *Three point estimates* - עושים הערכה לפי צפי אופטימי מציאותי ופסימי (יש לזה נוסחה)
4. *Bottom-up* - נפרק לחתיכות קטנות ונבדוק כל אחת בנפרד, זה הכי ארוך אבל גם נותן הערכה הכי טובה.

מודל SMART -

- *Specific* - יעדים צריכים להיות ממוקדים ככל שניתן. (לדוגמה לא לרשום "הגדלת רווחים" בכללי)
- *Measurable* - ניתנים למדידה כדי לדעת האם היעדים הללו הושגו.
- *Attainable* - יעדים הגיוניים וברי השגה.

- Relevant - יעדים המשרתים את היעוד והאסטרטגיה הארגונית.
- Time bound - יעדים התחומים בזמן.

שיטות לבדיקת כדאיות כלכלית:

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

כאשר:

- R_t - הכנסות פחות הוצאות באותה תקופת זמן.
- N - מספר השנים/חודשים לחישוב.
- i - שיעור ההיוון.
- t - מתי תגיע ההכנסה.

$$ROI = \% \text{ החזר ההשקעה} = \frac{benefits - costs}{costs} \cdot 100 \quad \bullet$$

כאשר:

- ה benefit זה רק ההכנסות אשר מחושבות בעזרת NPV (ב R שמים רק הכנסות בלי הוצאות).
- ה costs זה רק הוצאות אשר מחושבת בעזרת NPV (ב R שמים רק הוצאות בלי הכנסות).

ועדת היגוי:

כל הנציגים מתחומי ידע שונים (טכנולוגיה תשתיות DB HR כספים ולקוחות).
תפקידים:

1. קביעת מדיניות פיתוח.
2. מעקב אחר הלו"ז ועלויות המערכת.
3. פתרון בעיות במהלך הפיתוח.
4. שינויים בשלב הפיתוח.
5. קבלת החלטות בשלבים השונים של הפיתוח.
6. יצירת מחויבות אצל הלקוחות ואנשי הפיתוח.
7. קביעת לוז ומשאבים.

שלב הדרישות:

בעיה אופיינית בשלב זה:

משתמשים חושבים שהם יודעים מה הם רוצים, עד שהם רואים את התוצאה בעיניים.
המנתחים מניחים מה המשתמשים רוצים.
המפתחים חושבים שהם יודעים מה המשתמשים רוצים.
אין אמון בין בעלי העניין.
המשתמשים לא יודעים להסביר מה הם רוצים.

סוגי דרישות:

דרישה פונקציונאלית: מה המערכת אמורה לעשות מנקודת המבט של המשתמש.

סוגים:

1. דרישה תפעולית - דרישה המתייחסת לתפעול, לאינטרקציה או התנהגות של המוצר.
2. דרישת מידע - מתייחסת לישויות המידע.

דרישה לא פונקציונאלית:

סוגים:

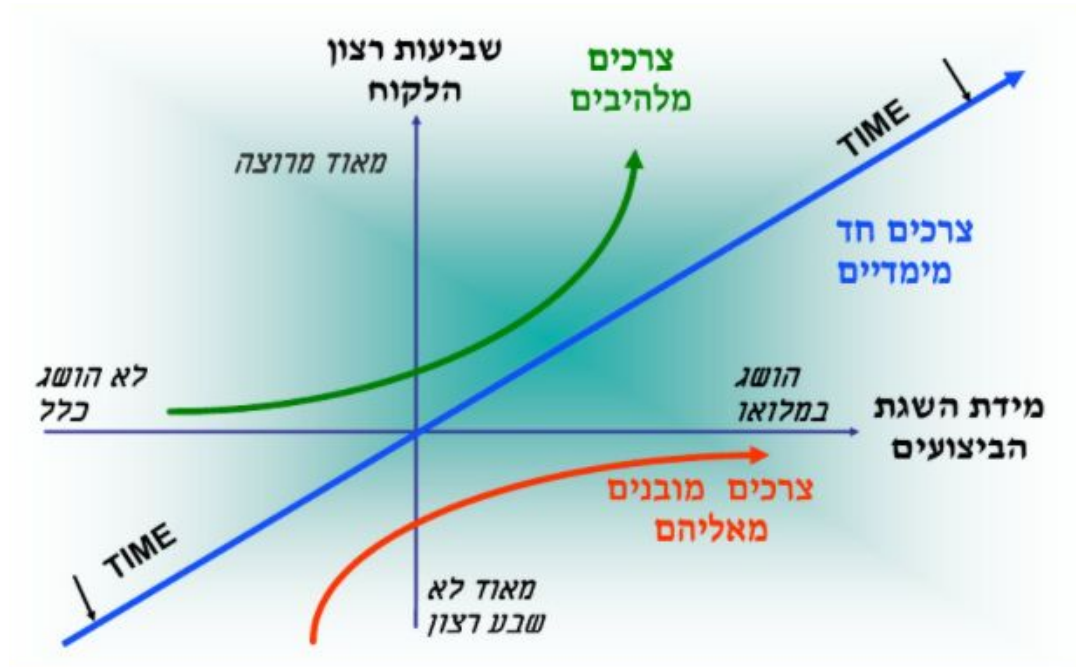
1. אילוצים:
 - a. אילוץ ניהולי - תקציב לזז זמינות משאבים וכו'...
 - b. אילוץ מימוש - אלגוריתם מבני נתונים וכו'...
 - c. אילוץ חומרה - רכיבים ארכיטקטורה וכו'...
2. איכות פתרון:
 - a. דרישות ביצועים - זמן תגובה נפח אחסון ניצול מעבד וכו'.
 - b. מאפייני איכות:
 - i. התאמת פונקציונאליות (functional suitability) - פיצ'רים הכרחיים ועד כמה המערכת מספקת תוצאות נכונות.
 - ii. יעילות ביצועים (performance efficiency) - אם המוצר רץ מהר וכמות המשאבים שאנחנו צורכים.
 - iii. תאימות (compatibility) - יכול להחליף מידע ולהתממשק עם מערכות נוספות.
 - iv. שימושיות (usability) - תרומת המוצר למשתמש וביצוע משימותיו, נוח ומובן.
 - v. אמינות (reliability) - פעולה ללא תקלות לאורך זמן.
 - vi. דרישת ביצועים (performance req) - פרמטרים ניתנים למדידה לגבי ביצועי התוכנה.
 - vii. זמינות (availability) - שירות רצוף התאוששות מהירה מתקלות ותפקוד על אף הבעיות.
 - viii. בטיחות (safety) - שמירה על חייהם ובריאותם של המשתמשים.
 - ix. בטחון (security) - המידע מוגן.
 - x. אחזקתיות (maintainability) - מתמודדת עם שינויים וקלה לבדיקה.
 - xi. נידוד (portability) - מידת האפקטיביות שבה ניתן להעביר מערכת סביבה תפעולית אחת לאחרת.

קריטריונים לדרישה איכותית:

1. Identified - שייכות ברורה.
2. Understandable - מנוסחת בשפת הלקוח.
3. Necessary - הכרחית
4. Complete - שלמה
5. Consistent - עקבית כלומר לא סותרת דרישות אחרות.
6. Unambiguous - חד משמעית
7. Verifiable - ניתנת לבדיקה באמצעות מבחני קבלה
8. Traceable - עקיבה
9. Prioritized - מתועדפת

מודל KANO קאנו

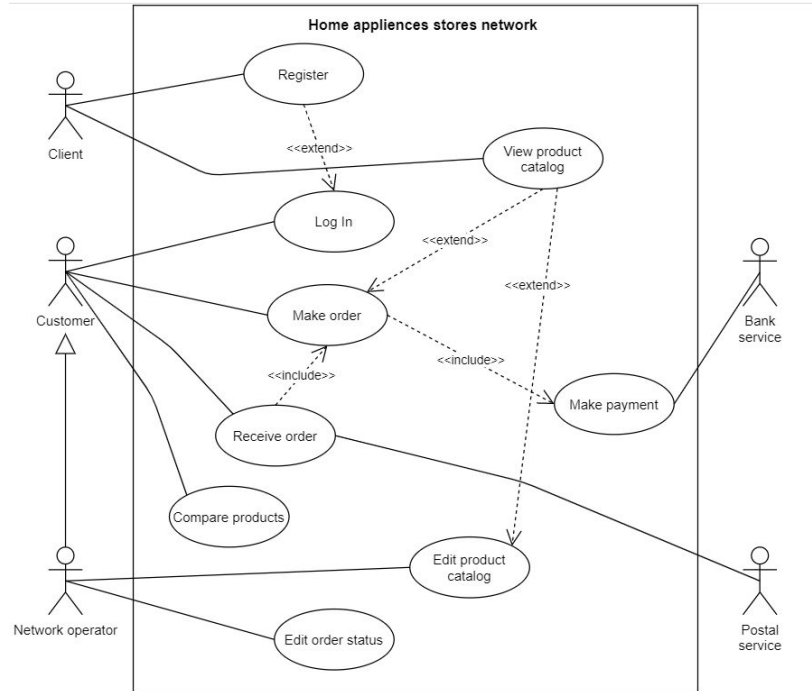
עושה השוואה בין רמת ההשקעה (ציר X) לבין רמת שביעות רצון הלקוח (ציר Y)



שלב הניתוח

דיאגרמות סטטיות:

- Use case -

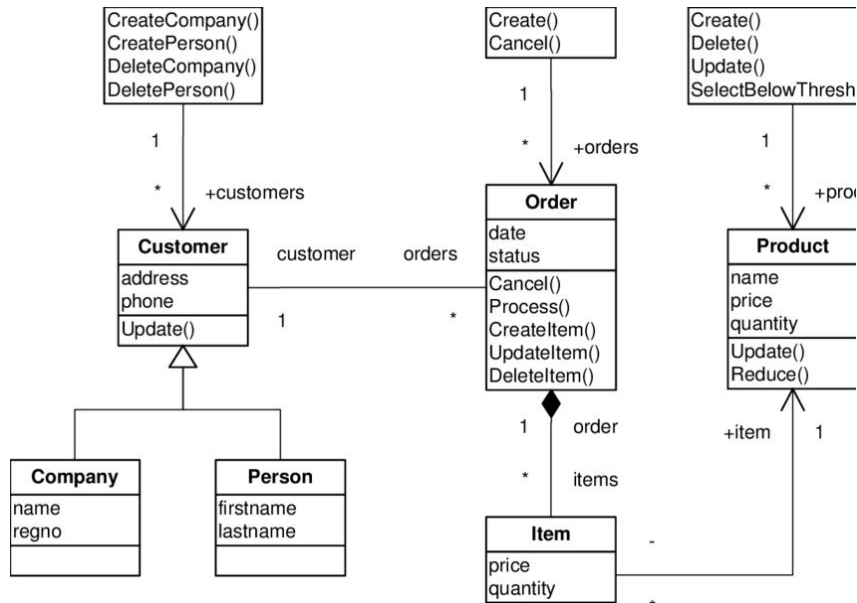


מונחים:

Extend - אופציה ללכת למחלקה לא מחייב

Include - חובה ללכת למחלקה

:Class diagram



סימונים:



.private : "-" ❖

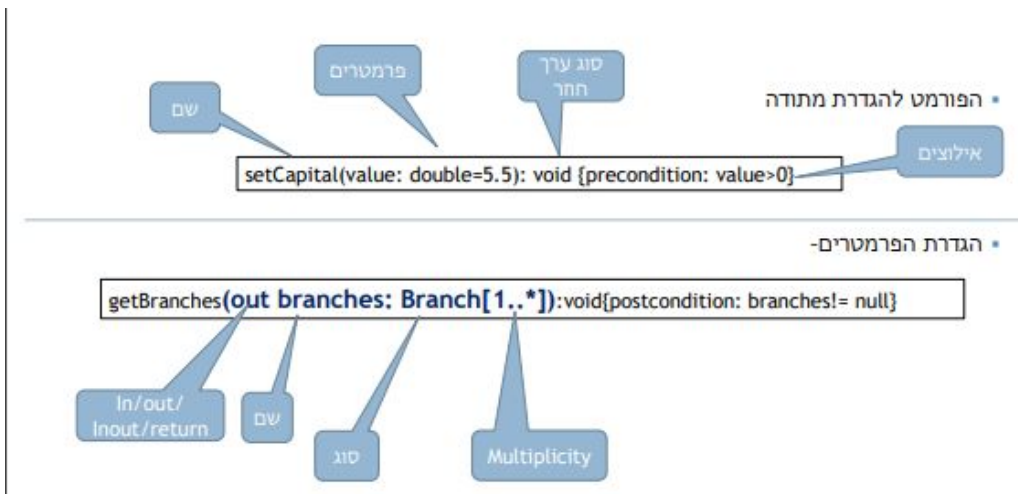
.public : "+" ❖

.protected : "#" ❖

- ❖ ~" : packedge.
- ❖ "\" : שדה שמחושב לפי שדות אחרים.
- ❖ קו תחתון : סטאטיק.
- ❖ כתב גולש : abstract.

קשרים בין מחלקות:

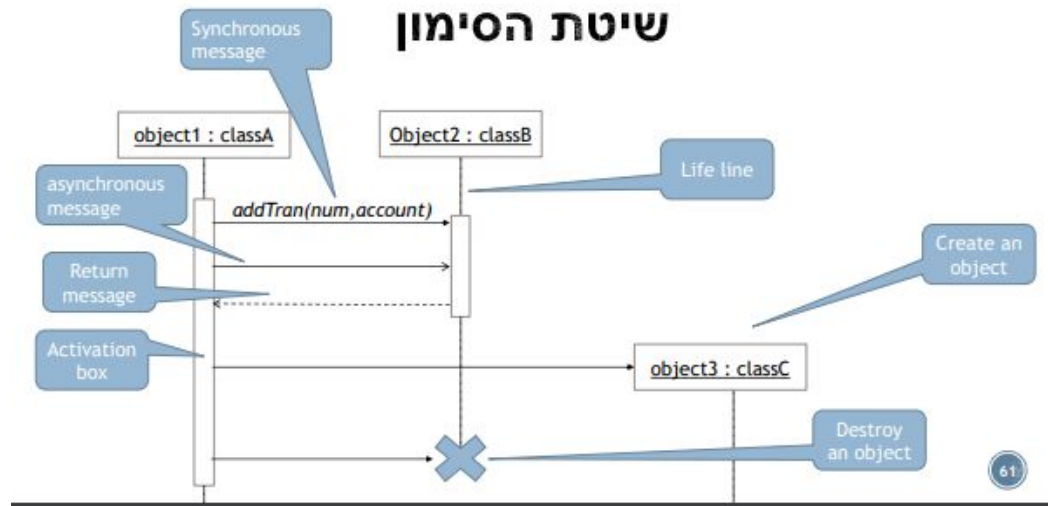
- ★ תלות - חץ מקווקו "-----<"
- ★ שייכות - חץ מקווקו עם מספר בסוף "1<----" אשר מציין את מספר האובייקטים שיכולים להיות שייכים. מאובייקט מוכל לאובייקט מכיל.
- ★ הרכב -  זה אומר מחייב שיהיה קיים.
- ★ צבירה -  זה אומר שיכול להיות קיים אבל לא מחייב.
- ★ הורשה - החץ מופנה כלפי האבא מהבן היורש בעזרת קו מלא.
- ★ ממשק - החץ פונה לכיוון הממשק ממי שממש אותו בעזרת קו מקווקו.



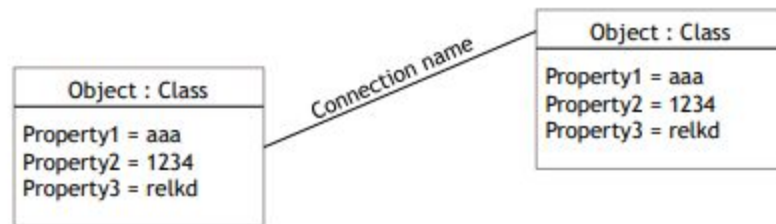
• Sequence Diagram

- ציר X האובייקטים ציר Y ההודעות שהאובייקטים שולחים.
- ❖ שליחת הודעה - מסומן בקו מלא.
- ❖ החזרת הודעה - מסומן בקו מקווקו. (return של פונקציה).

שיטת הסימון



Object Diagram •



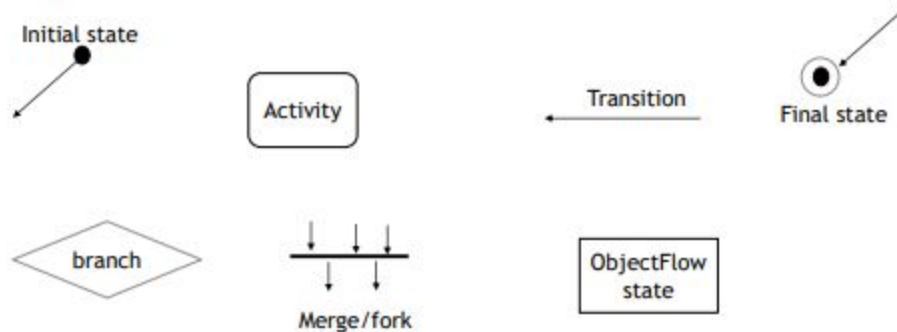
State machine Diagram •

שם את האובייקט במרכז



Activity diagram •

תרשים פעילויות



• ERD

דיאגרמה של הDB.

- מפתח חד חד ערכי.

* - מסמל שדה עם ערך.

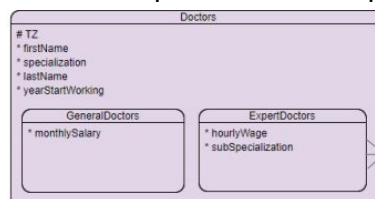
סוגי קשרים:

- ❖ קשר 1:1 - חח"ע מסומן בקו מקווקו.
- ❖ קשר M:1 - חד רב ערכי מסומן עם קו מקווקו שמתפצל.
- ❖ קשר M:N - רב רב ערכי מתפצל משתי הישויות.

קרדינליות הקשר - מספר מינימלי ומקסימלי של ישויות מסומן כ (x,y) מעל הקו.



תלות קיומית - כאשר קיום ישות מותנה בקיום ישות אחרת, מסומן



ישות על ותת ישות -



יחס בחירה בין מספר קשרים -

נרמול -

- NF 1 - כל תכונה יכולה לקבל ערך אחד בלבד ללא קבוצות.
- NF 2 - תכונה שאינה מופיעה באף אחד מהמפתחות לא יכולה להיות תלויה בתת קבוצה ממש של מפתח.
- NF 3 - תכונה שלא מופיעה באף אחד מהמפתחות לא יכולה להיות תלויה בקבוצה שאינה סופר מפתח.
- NF3.5 \ BCNF - תכונה לא יכולה להיות תלויה בקבוצה שאינה סופר מפתח.
- NF 4 - כל ישות צריכה להחזיק רעיון אחד בלבד.

טוּט

- User interface - ממשק המשתמש התצוגה שהוא רואה בזמן ביצוע פעולה.
- User experience - חווית המשתמש מודדת שביעות רצון.
 - מרכיבי חווית המשתמש:
 - שימושיות - פונקציות קלות לתפעול.
 - שגיאות - כמה שגיאות המשתמשים עושים, חומרתם והקלות לתקנם.
 - סיפוק - כמה נהנה מהמערכת.
 - יעילות - לאחר הכרה של הממשק, מהירות התפעול.
 - זכירות - לאחר חוסר שימוש, כמה קל יהיה לחזור להשתמש.
 - לימודיות - קלות לבצע משימוש החל מהשימוש הראשון.
 - אסתטיות - ..
 - תנועה - אנימציה וזרימה טבעית.
 - אמינות - נתינת מידע נכון ללקוח.
 - נגישות - ממשק נגיש לחיפוש והגעה להיכן שאתה רוצה בתוכנה.

שלב המימוש

עיקרון KISS - שאומר: Keep it simple stupid.

SOLID - עקרונות לתוכן יציב.

1. Single responsibility principle - כל מחלקה אחראי על פונקציונאליות אחת.
2. Open close principle - כל מחלקה סגורה לשינויים ופתוחה להרחבות. (ירושה). (כשיש שאלה שצריך לבדוק איזה מחלקה כדי לעשות פעולה)
3. Liskov substitution principle - אם S היא תת מחלקה יורשת של T, אז ניתן להחליף כל מופע של T במופע של S בלי לשנות את ההתנהגות.
4. Interface segregation principle - לעשות ממשקים קטנים כדי לא ליצור מצב שעושים שימוש בממשק שאין צורך בחלק מהפונקציות שלו.
5. Dependency inversion principle - אסור שבמחלקה יהיה משתנה מטיפוס מחלקה אחרת, צריך להחליף במשתנה מטיפוס הממשק. (כשיש שאלה שאחד ממשתני המחלקה הוא מסוג מחלקה אחרת ולא מסוג ממשק)

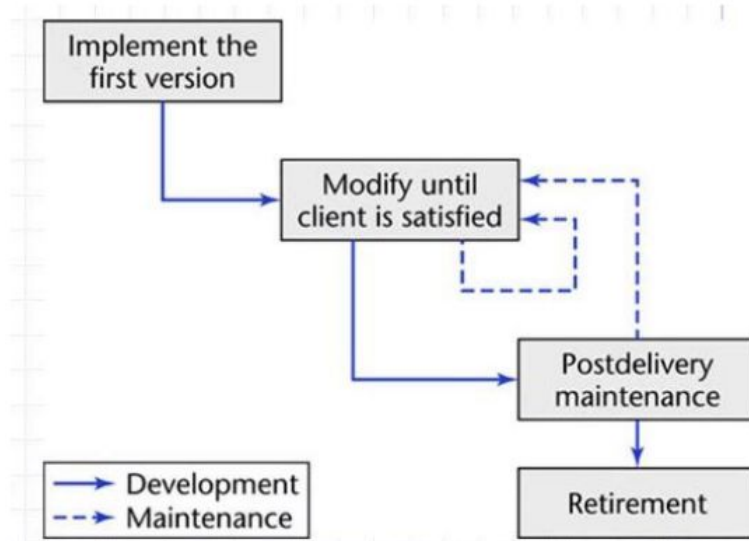
Code review - נעשית על ידי עמיתים, לא על ידי המתכנת עצמו, על הקוד עצמו במטרה למצוא קוד מת, דליפת זכרון או מידע, שימוש חוזר בקוד, קוד לא יעיל.

שיטות עבודה:

מודלים לינאריים:

- Code and Fix

Code and Fix



- ❖ עבודה: מתחילים ישר בכתיבת קוד ללא תכנון או מכינים מוצר בסיס שעליו עובדים לפי ביקוש.
- ❖ תאימות: פרויקטים קטנים.
- ❖ מבנה: אין מבנה אין התאמות מיוחדות ואין עיצוב.
- ❖ בכליות: מפתחים משחררים ללקוח מתקנים משחררים וכן הלאה..

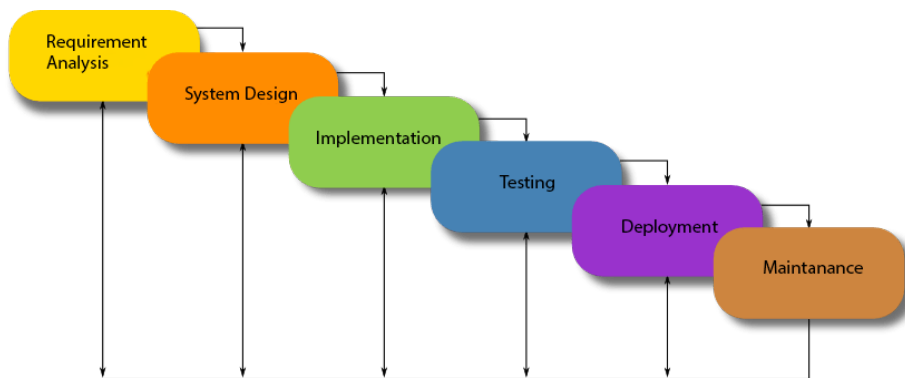
חסרונות:

1. סיכון מאוד גבוה.
2. גרוע למערכות מורכבות.
3. לא מתאים בפרויקטים מתמשכים.
4. אין לו"ז.

יתרונות:

1. קל לפיתוח
2. מהיר
3. קל לניהול
4. גמיש

• מפל המים \ WaterFall

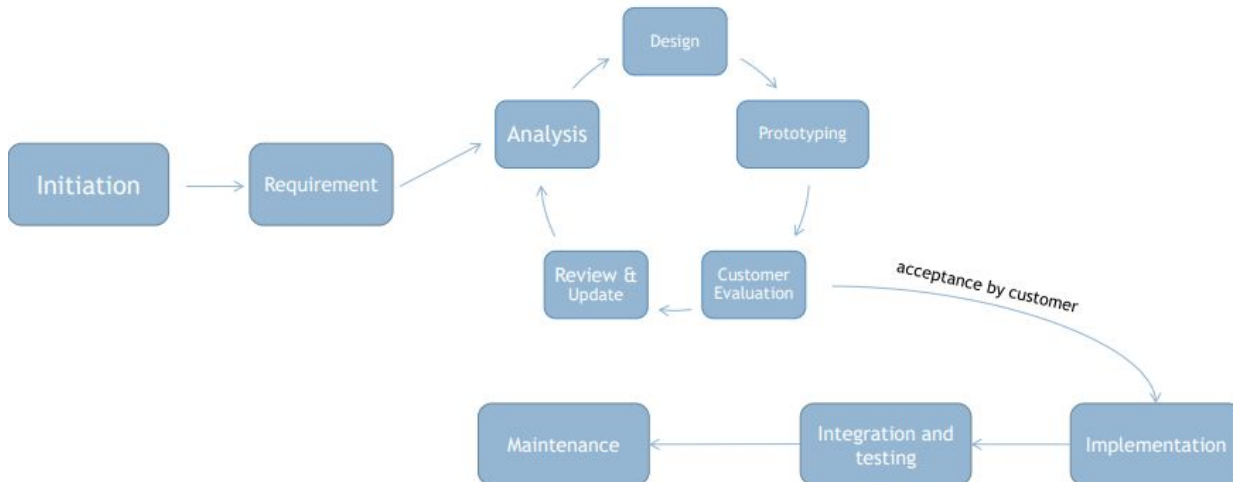


- ❖ עבודה: מחזור פיתוח אחד יזרום -> דרישות -> ניתוח -> עיצוב -> מימוש -> אינטגרציה -> תחזוק.

❖ בכלליות: שמה דגש על השלבים הראשונים ועיצוב מוקדם, יכול לקחת יותר זמן מאשר פיתוח.

יתרונות:	חסרונות:
1. תהליך מתועד היטב	1. המון זמן עד כתיבת קוד, לכן לא ניתן להראות אב טיפוס מוקדם.
2. החזקה קלה.	2. קשה למדוד התקדמות.
3. ...	3. לא טוב לפרויקט דינאמי.
4. ...	4. אין גמישות לשינויים.

• מודל אב טיפוס



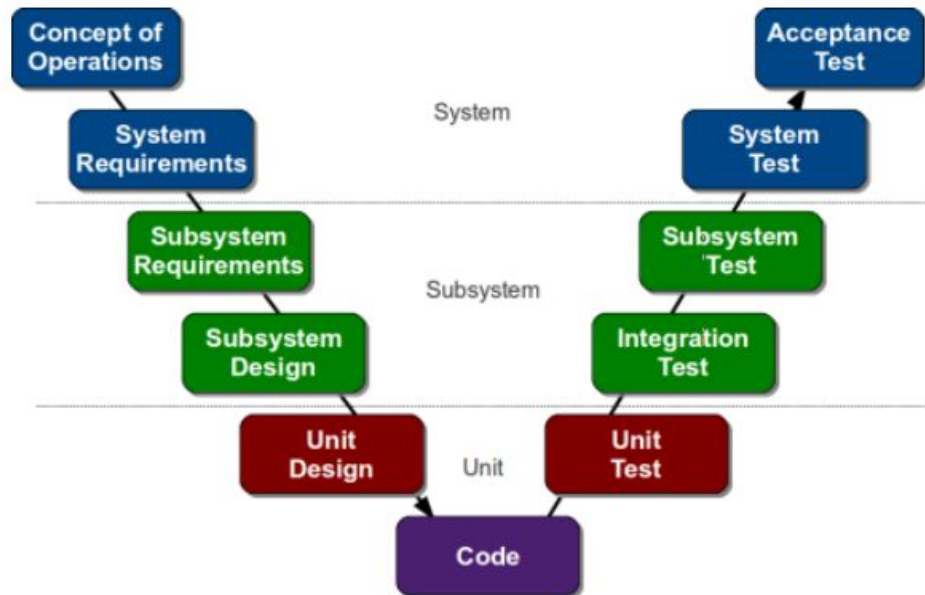
במפל המים הדרישות של הלקוח לא ברורות ולא סופיות ולכן נוציא גרסת BETA כדי להגדיר את הדרישות למוצר הסופי.

יתרונות:	חסרונות:
1. מבהיר את דרישות הלקוח.	1. מגדיל את עלות המערכת.
2. מקל במקרים של קשיים טכניים.	2. לא מקל על סיכונים שמתגלים בתהליך הפיתוח.

• V model

דומה למפל המים רק שבכל שלב יש בדיקות ונועד לפשט ולשפר את תהליך הפיתוח.

V model



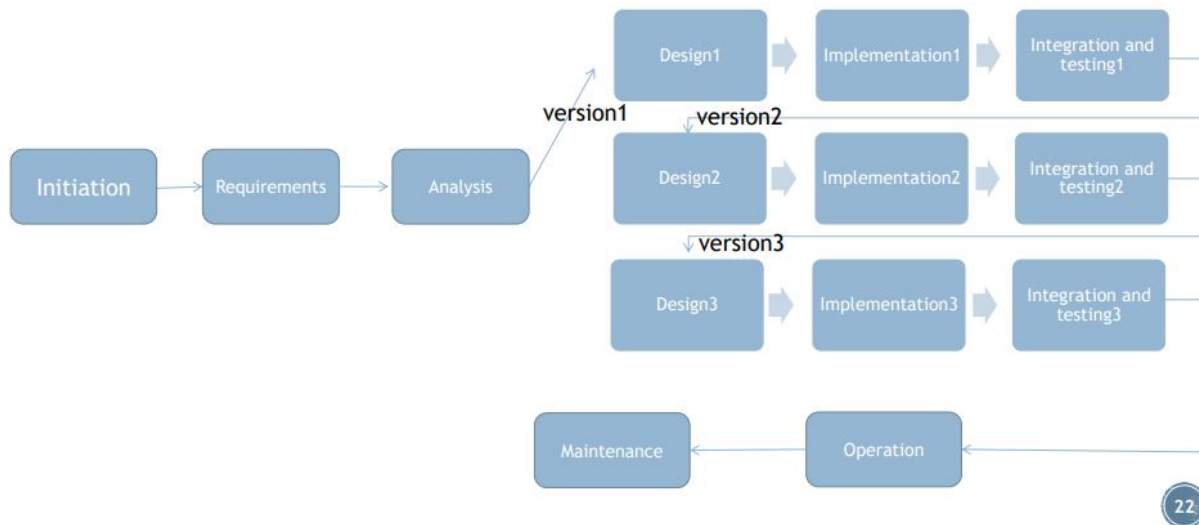
- ❖ שלב ראשוני: ניתוח ברמה הכללים ובשלים יורדים ברזולוציה והניתוח נהיה מפורט יותר עד לרמת יחידה.
- ❖ שלב פיתוח: מפתחים מתחילים מהיחידות הכי קטנות עד לקבלת תמונה מלאה.
- ❖ שלב שני: בדיקות בסדר הפוך מהשלב הראשוני - מבדיקת יחידה עד מערכת ובדיקות קבלה סופיות.

- | יטרונות: | חסרונות: |
|------------------------------------|-----------------------------|
| 1. קל להבנה ושימוש. | 1. פיתוח מתחיל בשלב מאוחר. |
| 2. קל לניהול בגלל התיעוד והסדר. | 2. חוסר וודאות. |
| 3. השלבים ואבני הדרך מוגדרים היטב. | 3. אי התאמה לפרויקט דינאמי. |
| 4. תהליך ותוצאות מתועדות היטב. | 4. חוסר גמישות לשינויים. |

מודלים איטרטיביים:

מתאימים לשינוי הדרישות (פרויקט דינאמי).

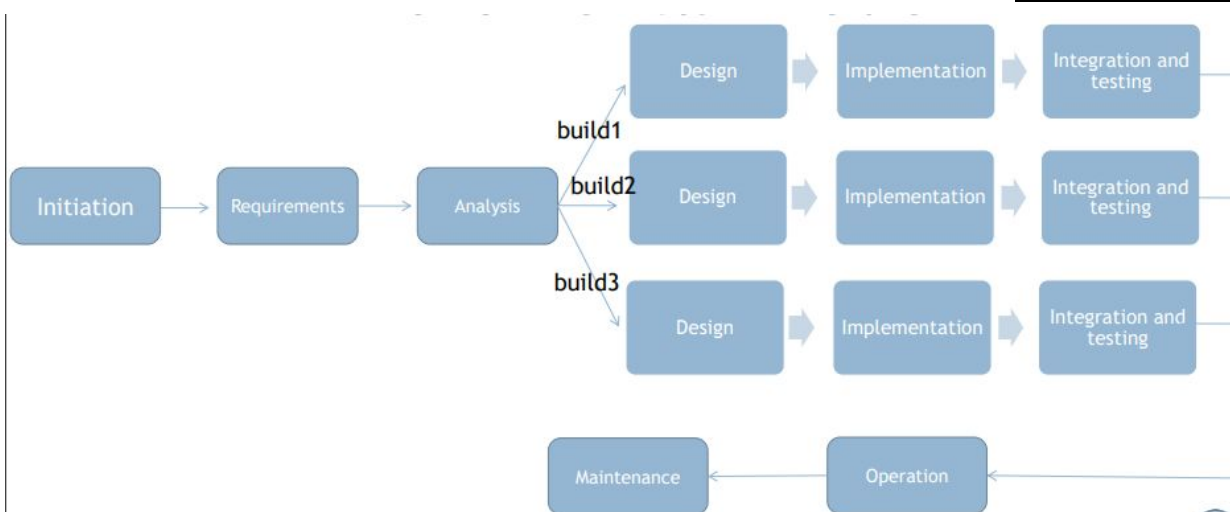
- Iterative model \ מודל איטרטיבי



- ❖ עבודה: ייזום -> דרישות -> אנליזה -> *3 (עיצוב -> מימוש -> אינטגרציה -> טסט) -> תחזוק.
- ❖ בכליות: בונים משהו ועושים סיבובי שיפור עד הגרסה הסופית. יש להגדיר דרישות עיקריות ולפתח אותן עם הזמן.

- | | |
|--|---|
| <p>יתרונות:</p> <ol style="list-style-type: none"> 1. ניתן לראות תוצאות בשלב מוקדם. 2. ניתן למדוד התקדמות בפרויקט. 3. אפשר לפתח במקביל. 4. שינויים בעלויות נמוכות. 5. יותר קל לבדיקה. | <p>חסרונות:</p> <ol style="list-style-type: none"> 1. המון משאבים. 2. לא מתאים לשינויים בדרישות. 3. דורש ניהול צמוד. 4. אין תמונה מלאה לפני תחילת התהליך. |
|--|---|

Incremental model •

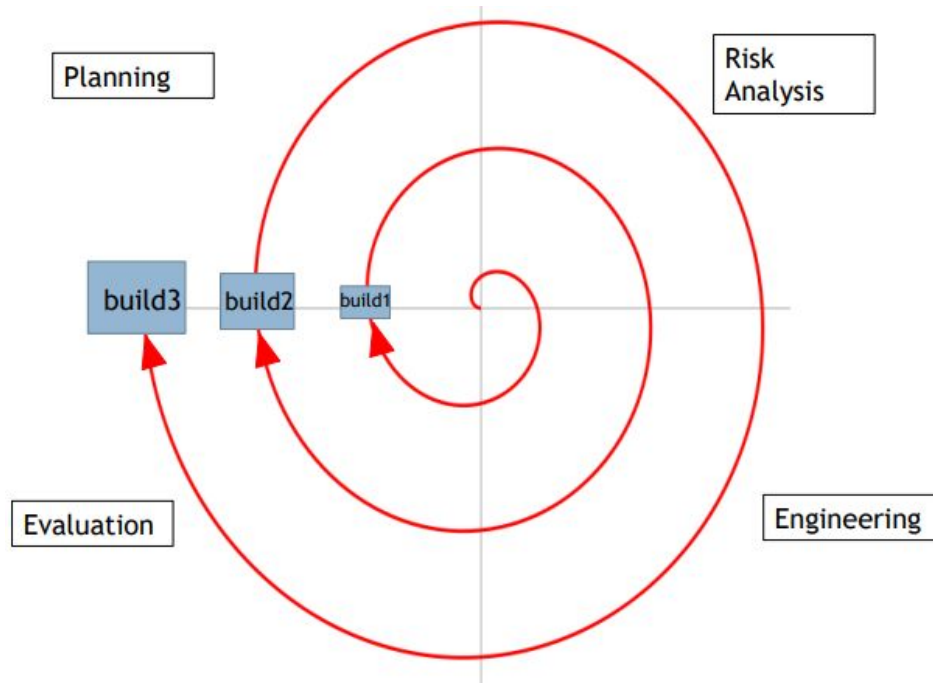


- ❖ עבודה: ייזום -> דרישות -> אנליזה -> במקביל: (עיצוב -> מימוש -> אינטגרציה וטסט) -> תחזוק.

- | | |
|--|---|
| <p>יתרונות:</p> <ol style="list-style-type: none"> 1. ניתן לראות מוצר מהר ומוקדם. | <p>חסרונות:</p> <ol style="list-style-type: none"> 1. דרוש עיצוב ותכנון קפדניים. |
|--|---|

2. קל יותר לנהל סיכונים.
2. יש צורך להבהיר ולהגדיר את כל (חלקים מסוכנים מוגדרים ומטופלים ביטריה שלהם) המערכת לפני החילוק.
3. גמיש וזול לשינויים.
3. העלות הכוללת גבוהה ממפל המים.
4. קל יותר לבדוק איטרציות קטנות.

• המודל הספיראלי



- ❖ עבודה: תכנות - ניתוח סיכונים - מימוש - הערכה.
- ❖ כלליות: עובדים באיטרציות על ארבעת השלבים אשר כל איטרציה מבוססת על הקודמת.

- | | |
|---|--|
| <p>יתרונות:</p> <ol style="list-style-type: none"> 1. ניהול סיכונים ברמה גבוהה. 2. מאפשר להתחיל פרויקט פשוט ולהוסיף סיבוכיות. 3. הפיתוח מתחיל מוקדם. | <p>חסרונות:</p> <ol style="list-style-type: none"> 1. עלויות גבוהות. 2. הצלחת הפרויקט תלויה בניתוח סיכונים. 3. לא עובד בפרויקטים קטנים. |
|---|--|

מודלים אגיליים Agile:

פיתוח זריז ויעיל בצוותים קטנים. מניח שלא ניתן להגדיר תוכנה במלואה לפני פיתוחה ומתמקד במקום זאת בשיפור היכולת של הצוות לספק תוצרים.

➤ עקרונות הגישה:

- עדיפות עליונה - לשחרר גרסה עובדת ללקוח כמה שיותר מהר ולעיתים קרובות.
- שינויים בדרישות משפרים את התוכנה ולכן נקדם בברכה.
- שת"פ בין מומחי היישום למפתחים.
- תוכנה עובדת היא המדד העיקרי להתקדמות.
- תחזוקת המערכת היא חלק בלתי נפרד מהפיתוח וכל בעלי העניין והמפתחים צריכים לעשות זאת.
- יש לפתח רק את מה שהכרחי.

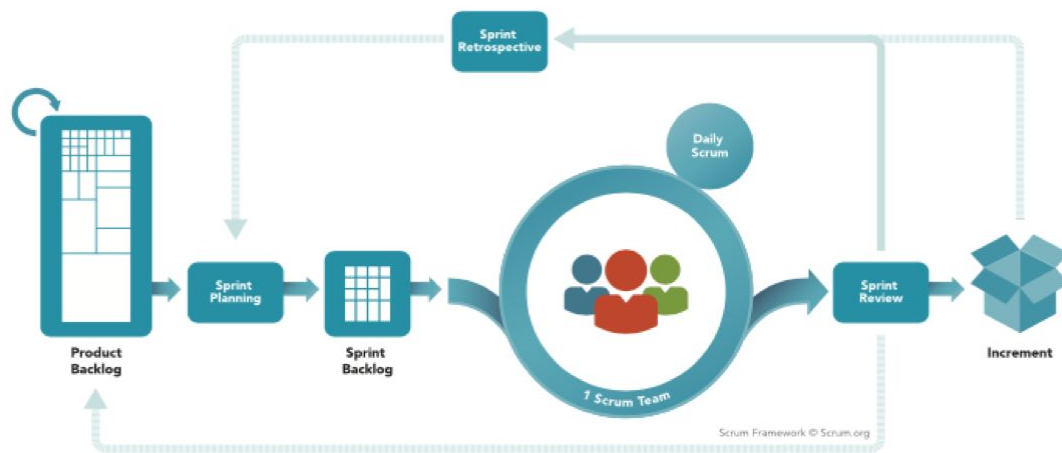
- נשאר לצוותים את האחריות לארכי' ולעיצוב.

השוואת בין מודלים ליניאריים למודלים אג'ילים:

Agile	ליניארי	
מעט ידועות מראש הרוב נבנה בהמשך התהליך	ידועות מראש	הדרישות בפרויקט
גמיש	קשה	שינויים בתכולת הפרויקט
נמוכה	גבוהה	רמת סיכון
גבוהה תמיד	גבוהה בהתחלה נמוכה בהמשך	מעורבות הלקוח
באיטרציות	פעם אחת	מסירה ללקוח
לאנשים	לתהליך	אוריינטציה של המודל
צוותים קטנים	גדול	גודל הצוות
הערך העסקי ללקוח	בוצעו כל הדרישות	מדדים להצלחה

SCRUM •

SCRUM FRAMEWORK



פיתוח תוכנה היא בעיה אמפירית שנפתרת בשיטה של ניסוי וטעייה.
 בכליות: נתחיל בהדרגה נלמד מטעויות ונשפר את המערכת כל הזמן.

❖ מחזור פעילות:

- SCRUM EVENT SPRINT - תקופה של חודש בו נוצר מוצר וכל המשימות בוצעו. ספרינט חדש מתחיל מיד לאחר סיום ספרינט קודם.
- SPRINT PLANNING - העבודה שיש לבצע בספרינט ואיך לבצע נוצרת על ידי כל הצוות.
- DAILY SCRUM - פגישה קצרה של חברי הצוות בכל יום במהלך הספרינט בה צוות הפיתוח מתכנן את העבודה למשך היום.
- SPRINT REVIEW - פגישה המתקיימת בסיום הספרינט על מנת לבדוק את התוספת.
- SPRINT RETROSPECTIVE - פגישה המתרחשת לאחר הספרינט ריוויזיוני לפני הספרינט פלנינג הבאה ומטרתה לתת הזדמנות לצוות לבדוק את עצמו וליצור תוכנית שיפורים שיכולו על הספרינט הבא.

❖ בעיות:

- חוסר בהירות בנוגע לתכולה הסופית של הפרויקט - הלקוח יכול להמשיך לבקש עוד פונקציות.
- צוות שמנהל את עצמו.
- עבודה ביחידות קטנות.
- צוותים קטנים.
- מצריך שינויים מאוד גדולים בארגון.
- תפקיד המאסטר קשה לביצוע וקשה לאיש.

• XP \ Extreme programming

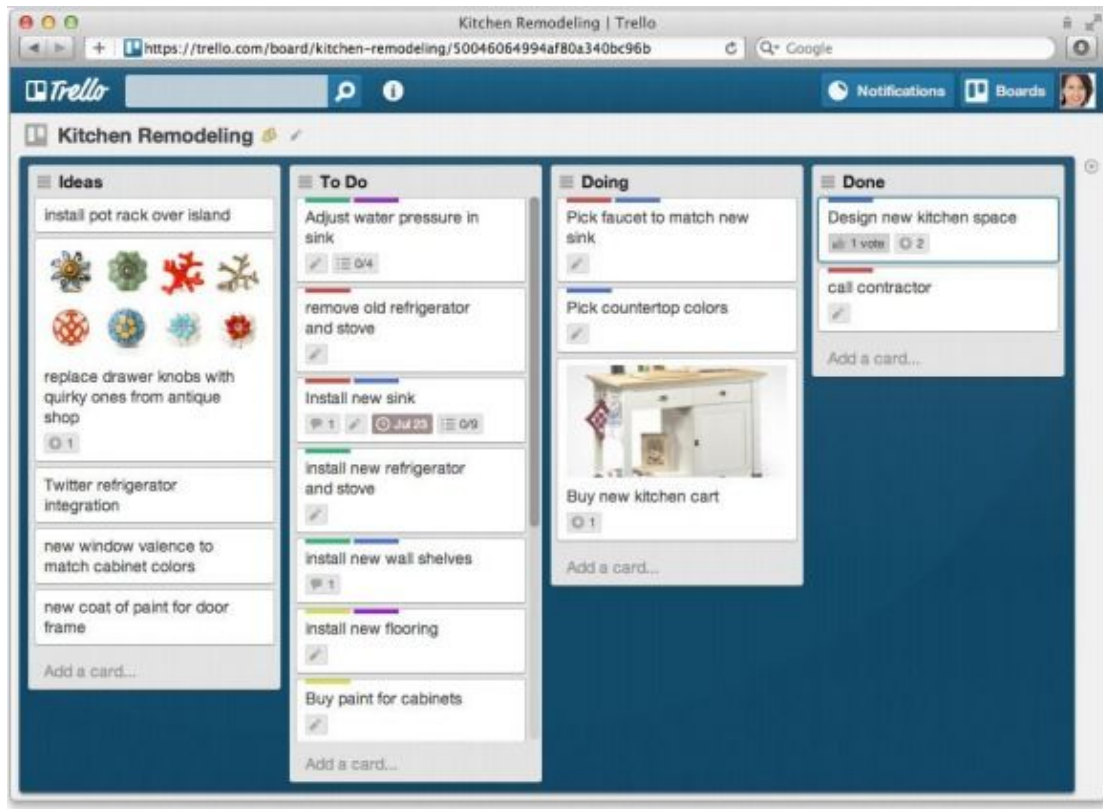
נותנת דגש למעורבות הלקוח, תקשורת טובה בצוותים ועבודה באיטרציות פיתוח.
❖ עבודה: תכנון -> עדיפות -> טסטים -> מחזור של (קוד -> אינטגרציה וטסט) -> שחרור.

יתרונות:	חסרונות:
1. מעורבות הלקוח - מגדילה את הסיכוי שהתוכנה תענה על צרכי המשתמשים.	1. מיועד לפרויקט אחד מתוחזק על ידי צוות אחד.
2. מקטין את הסיכון.	2. פגיע למפתחים שעובדים לבד.
3. בדיקות ואינטגרציה מגבירות את איכות העבודה.	3. לא יעבוד בפרויקטים שצריכים מפרט מלא לפני התכנות.
4. זמני פיתוח קצרים.	4. לא יעבוד בסביבה בה המתכנתים מופרדים גיאוגרפית.
5. הכנת הבדיקות בשלב ההגדרות.	

• Pair programming

שני מפתחים על אותו מחשב זוגות דינמיים.
הרעיון: שני אנשים בוודאות יכתבו קוד יותר טוב ויעיל ממישהו אחד.
שני המתכנתים הם בעלי רמות שונות של ניסיון אבל זה לא נועד לחניכה.

• KanBan - סימן ויזואלי



נותנת דגש לצד הויזואלי - מה מתי וכמה לייצר.
מעודדת שינויים קטנים והדרגתיים.

יתרונות:

1. מגביר את הגמישות לשינויים.
2. מצמצם בזבז זמן - תמיד ממתינה למתכנת עוד משימה.
3. קלה להבנה והטמעה - ניתן להשתמש על גבי מתודולוגיה אחרת.
4. מקצר את זמני מסירת המשימות.

חסרונות:

1. כל הזמן צריך לעדכן את הלוח.
2. אין בלוח מידע לגבי זמני המסירה.

Scrum	Kanban	
מוגדר מראש: מאסטר משמש כמאמן אג'יל של הצוות. פרודקט owner מגדיר מטרות ויעדים. חברי הצוות מבצעים את העבודה.	לא מוגדרים מראש, כולם צריכים לעזור לכולם לבצע את מה שצריך.	תפקידים ואחריות:
נקבע באמצעות הגדרת הספרינט.	באופן רציף על בסיס צורך.	מועדי אספקה של תוכנה עובדת:
לפי הספרינט.	כל פעם עושים משימה חדשה מה-todo.	טיפול במשימות:

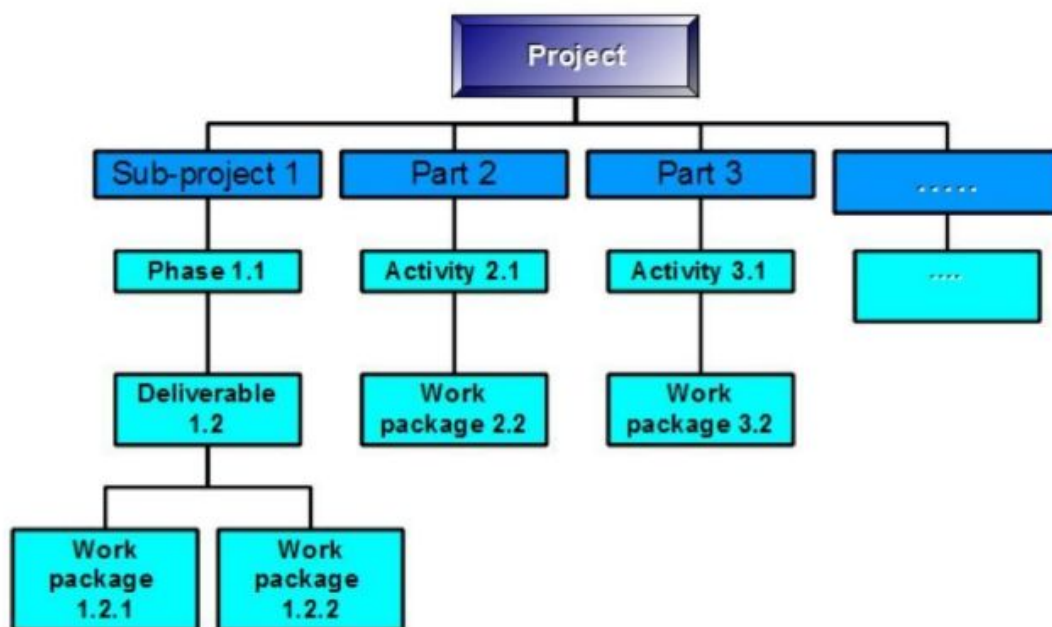
מדידת תפוקה:	לפי זמן מחזור - הזמן העובר מתחילת הטיפול במשימה ועד סיום המשימה.	ההספק הממוצע בספרינטים הקודמים. נמדד על ידי מושג הנקרא: Story Points, או על ידי מספר המשימות שהסתיימו בספרינט.
שינויים:	אפשר בכל נקודה בזמן	אסור לבצע בזמן ספרינט.

Project management

כלים בשלב תכנון הפרויקט:

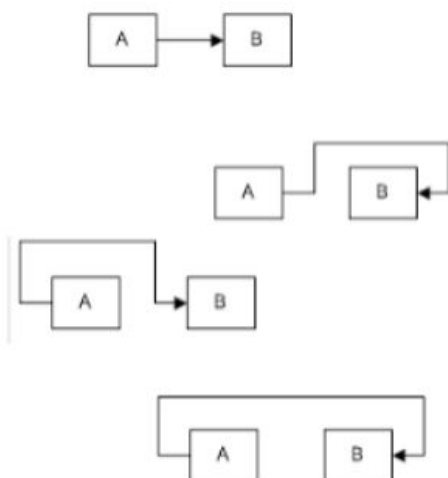
• WBS - work breakdown structure

חלוקת הפרויקט לתת משימות, הרמה העליונה מייצגת את המוצר הסופי והרמה התחתונה נקרא חבילת עבודה וניתן להקצות אותה לאדם מהצוות.



• תרשים רשת

מראה תלות בין מטלות.



Finish to Start (FS) ▪

- B יכול להתחיל רק כאשר A מסתיים
- זו בדרך כלל ברירת המחדל
- התלות הנפוצה ביותר

Finish to Finish (FF) ▪

- B יכול להסתיים יחד עם A או אחריו (A חייב לסיים קודם)
- או- B לא יכול להסתיים לפני ש-A מסתיים

Start to Start (SS) ▪

- B יכול להתחיל בו זמנית עם A או אחריו
- B לא יכול להתחיל לפני A

Start to Finish (SF) ▪

- B מסתיים כאשר A מתחיל
- או- חייבים להתחיל את A בשביל ש-B יוכל להסתיים
- זהו סוג יחסים נדיר

• Pert

נוסחה לחישוב זמנים.

$$\frac{\text{shortest time} + 4 * \text{likely time} + \text{longest time}}{6}$$

6

יתרונות:

1. מאפשר לנתח ולהעריך את הזמן העלות והמשאבים הדרושים.
2. נותנת תמונה מלאה למנהל ולשאר בעלי העניין.
3. מאפשר מעקב אחרי ביצוע הפרויקט.

חסרונות:

1. לא נותן עלות מדויקת אלה משוערכת ולא אובייקטיבית.
2. ממוקד בהערכות זמנים ופחות בדברים אחרים.
3. שיטה מורכבת וקשה לביצוע.

נתיב קריטי: שרשרת פעילויות הקשורות זו בזו אשר כל שינוי בזמנים שלהן ישפיע על מועד סיום הפרויקט.

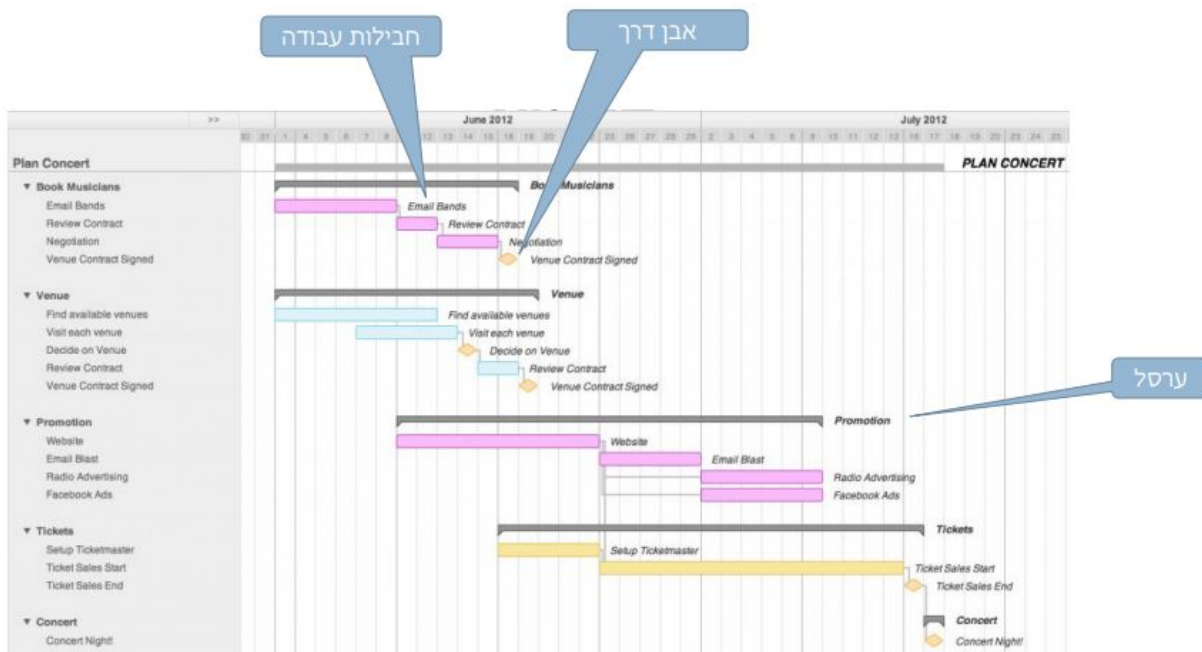
Activity

Early Start	Duration	Early Finish
Activity Name		
Late Start	Slack	Late finish

- Duration – משך הזמן בו המשימה תתבצע (בימים/חודשים).
- Early start – התאריך המוקדם ביותר בו משימה יכולה להתחיל ביחס לתלויות.
- Early finish – התאריך המוקדם ביותר בו משימה יכולה להסתיים בהתייחס לתלויות.
- Late start – התאריך המאוחר ביותר בו משימה יכולה להתחיל בלי לדחות את מועד סיום הפרויקט.
- Late finish – התאריך המאוחר ביותר בו משימה יכולה להסתיים בלי לדחות את מועד סיום הפרויקט.

Gantt •

הציר האופקי מייצג את הזמן והציר האנכי מייצג את חבילות העבודה.



דרכים לצמצום לוז :

- בדיקה מחודשת של הנחות ודברים שלא היו ידועים בזמנו.
- הוספת משאבים לנתיב הקריטי.
- העברת תת משימה שהזמן שלה קטן מה SLACK מהנתיב הקריטי.

ערך מזוכה:

ניתוח מגמת זמן ומשאבי הפרויקט במטרת חיזוי עלות הפרויקט.

BAC - העלות המתוכננת.

AC - העלות בפועל עד נקודת הבקרה.

נוסחאות:

סימון	הסבר	נוסחה
PV	כמה היינו אמורים לשלם בזמן הזה	אחוז זמן הביצוע בתכנון כפול BAC. (בהינתן תקופת זמן - עלות כל שלב כפול האחוז עשייה שהיה אמור לעשות בתקופת הזמן הנתונה)
EV ערך מזוכה	כמה היינו אמורים לשלם כדי להגיע לכמות העבודה שעכשיו אנחנו נמצאים בה.	אחוז הביצוע בפועל כפול BAC. (עלות כל שלב כפול האחוז עשייה בפועל שנעשה)
CV	שונות העלות.	EV - AC

EV - PV	שונות הזמנים.	SV
$\frac{EV}{AC}$	אינדקס שאומר לי האם אנחנו במצב חיובי מבחינת עלויות.	CPI
$\frac{EV}{PV}$	אינדקס על הזמנים.	SPI
$\frac{BAC}{CPI}$	כמה באמת יעלה לי שאסיים	EAC
EAC - AC	כמה נשאר לי כדי לסיים	ETC
BAC - EAC	שונות עלות הסיום	VAC

בדיקות

קופסא לבנה	קופסא אפורה	קופסא שחורה
מתוך הכרה של המערכת נבדקים כל התרחישים האפשריים לקבלת פלט. (מכיר את כל התוכנה והמימוש ובודק את כל השלבים)	הכרות המבנה הפנימי אבל עושות בדיקת קופסא שחורה. (מכיר את המימוש אבל בודק את הפלט)	לא מכירות את המערכת מודקות מה הפלט לקלט מסוים האם זה נכון. (לא מכיר את המימוש ובודק רק את הפלט)

בדיקות מסירה: בדיקות תרם מסירת המוצר ללקוח. באחריות צוות הפיתוח וצוות הבדיקות.
בדיקות קבלה: בדיקה לפני הפעלה. וידוא עמידה בדרישות, התאמת המערכת לצרכי הלקוח. באחריות הלקוח ובסביבת הלקוח.

בדיקות יחידה: על ידי תוכניתן כל פונקציה נבדקת בנפרד כל פעם שיחידה משתנה. Code review.



בדיקות אינטגרציה: בדיקה של שילוב של המודילים בדיקת תהליכים END TO END על ידי צוות הבדיקות בסביבת הבדיקות.



בדיקת ממשקים: בדיקה של תתי מערכת שונים ושילובם.



בדיקת מערכת: אבטחה, התאוששות, DATA, בדיקה סופית לפני העברה ללקוח.

תוצרים של בדיקות:

1. STP - Software test plan - סדר הבדיקות מנהל כותב.
2. STD - Software test description - אופן הבדיקות, מתכנן הבדיקות כותב.
3. STR - Software test report - דוח הבדיקות, כל התקלות שנמצאו ברמת מוכנות המערכת. נכתב על ידי ראש צוות הבדיקות.

STR	STD	STP	
מה זה?	התכנון של הבדיקות	תיאור של כל הבדיקות שנעשה	זה הדו"ח עצמו מתארים את הבאגים, האם המערכת מוכנה וכו..
מי כותב?	מנהל הבדיקות	המתכננים של הבדיקות - הבודקים. (נכתב בשיטות של כמה בודקים)	מנהל הבדיקות \ ראש צוות.
מתי כותבים?	במקביל לזמן שבו אנחנו כותבים את האפיון. חייב להסתיים לפני הבדיקות בפועל וSTD.	נכתב אחרי STP. את התכנון והפירוט של הבדיקות אמורים לכתוב לפני הבדיקות ובמקביל לפיתוח.	כותבים אותו בסוף תהליך הבדיקות, יכול להיות גם בין סבבי בדיקות.
למי זה מיועד?	לכולם.. מתכנתים לקוחות וכו'..	בודקי התוכנה.	מנהלי המערכת, נציגי הלקוחות.

שיטה לזכור: Plan - Description - Report.

7 עקרונות הבדיקה

1. מה זה בדיקה - לעשות הכל כדי לגרום למערכת להיכשל בלמצוא באגים.
2. בדיקה לעומת מפרטים - מפרט = מה בודקים ואיך. בדיקה לפי המפרט.
3. בדיקות גרסיה - כל ביצוע שנכשל מניב בדיקה קבוע שתמיד נבדוק.
4. תוצאות הבדיקה הן test oracles - ההחלטה האם הצלחנו או נכשלו תהיה אוטומטית ומבוססת.
5. בדיקות ידניות ואוטומטיות - לעשות את שניהם.
6. הערכה אמפירית של אסטרטגיות הבדיקה - נבדוק האם השיטה יותר עובדת מאשר נכשלת.
7. קריטריוני הערכה - מספר השגיאות שהיא מצליחה לגלות כפונקציה של זמן.

Testability - בדיקתיות

עמידה בה מערכת או רכיב מאפשרים את ההגדרה של קריטריוני הבדיקה ואת הביצוע של הבדיקות אשר יקבעו האם הקריטריונים הללו הושגו.

כלים להבטחת בדיקתיות

1. Controllability - יכולת בקרה - קיים ממשק בו ניתן להפעיל בדיקות. \ניתן לשלוט על המשתנים באופן ישיר ובלתי תלוי. יש נגישות.
2. Observability - שקיפות - ניתן לראות את כל מה שמתרחש במערכת כל הגורמים הרלוונטיים גלויים. שגיאות פנימיות מתגלות באופן אוטומטי ומדווחות על ידי מנגנוני בדיקה עצמית.
3. Availability - זמינות - בתוכנה קיימים כמה באגים אך באגים אינם מפריעים להרצת בדיקות.
4. Simplicity - פשטות - קוד עקבי וקריא, מחלקות קטנות, לכל דבר יש אחריות אחת.
5. Stability - יציבות - שינויים לתוכנה לא קורים הרבה, הם מבוקרים ומפרסמים ולא הופכים בדיקות אוטומטיות לבלתי תקפות.
6. Information - כל מבנה המערכת ידוע ומתועד.

סוגי בדיקות המערכת:

1. בדיקות פונקציונאליות - בדיקות נקודתיות על כל המערכת תוך ניסוי קומבינציות שונות שיכולות להוות בעיה. תכנון התרחיש ובדיקת התוצאות שלו. ביצוע סבב בדיקות שלם. בדיקת התנהגות המערכת מול התוצאה הצפויה.
2. בדיקות תצוגה - נוודא שהתצוגה היא כמו שהגדרנו בדרישות.
3. בדיקות נתונים והסבות - נבדוק לאחר שינוי תשתית או עיצוב החל מרמת השדה הבודד ועד הצגה של טבלאות ומה שחוזר משאילתות.
4. בדיקת שימושיות - נבדוק את הממשק משתמש את התפקוד המערכת תחת עומס. בבדיקה ננסה כמה שיותר לדמות את הפעולות השכיחות משתמש עושה.
5. בדיקת ביצועים - נעמיס את המערכת בהמון נתונים ושימוש ונבדוק שהיא עומדת בהן.
6. בדיקת רגרסיה - בדיקות רוחביות בעזרת תרחישי בדיקה שנכתבו בעבר עבור בדיקה חדשה כדי לראות שהגרסה החדשה לא פוגעת בישנה.

אופי בדיקות

1. ידניות - בודק בודק מקרי קצה.
יתרונות:
 - עלות נמוכה ובקלות ניתן לשנות את התסריט.
 - הכי קרוב למציאות כי הוא מדמה משתמש.חסרונות:
 - לא כל דבר אפשר לבדוק ידנית.
 - יכול להיות שצריך לעבור על משהו הרבה פעמים.
 - כל שינוי מצריך בדיקה מחודשת.
2. אוטומטיות - מינימום התערבות אנושית.
יתרונות:
 - מהיר ויעיל.
 - תפקיד הבודק הופך מעניין שדורש חשיבה.
 - תוצאות מופקות בדוח שכולם רואים.חסרונות:
 - עלויות.
 - אי אפשר לבדוק הכל אוטומטית.

אפקטיביות של בדיקת תוכנה

אפקטיביות = כמות השגיאות שהתגו בסבב הבדיקות (חלקי) סה"כ שגיאות שהתגלו במחזור החיים שמקורם בתוצר הנבדק.

אנדרואיד

מבנה:

- Linux kernel
- Hardware Abstraction Layer
- Native C/C++ Libs
- Android runtime

- JAVA API Framework
- System Apps

הגדרות:

- Activity - כל מסך באפליקציה מורכב מ2 קבצים:
 - קובץ JAVA - אחראי על הלוגיקה הפעולות שקורות במסך
 - קובץ XML - אחראי על ה Layout - UI.
- בנוסף יש לה מתודות שמורות משלה בקשר למחזור החיים שלה (...onCreate Onresume)
- הינה תיקיה המכילה את מבנה המסכים (קבצי XML) - סידור הרכיבים המוצגים במסך.

שיטות להגדרת פריסת הרכיבים השונים על המסך:

- ה Linear layout - יש לנו אפשרות בין אנכי לאופקי, בשניהם הרכיבים יופיעו אחד אחרי השני והולכים לאותו כיוון כמו הסדר שבו הם מופיעים בקובץ הXML.
- ה Table layout - המסך מוצג בצורה של טבלה ניתן למקם פריטים בשורות. ניתן גם לחלק את המסך לשורות ועמודות ומה גודל של כל תא.
- ה Constraint layout - השיטה הרגילה בה כל רכיב צריך ליצור לו "קווי גבולות" ויחסים עם שאר הפריטים במסך.
- ה Frame layout - שיטת הצבה בה הרכיבים מתמקמים אחד על גבי השני.

הערה: ניתן לערבב בין כל ה layouts אחד בתוך הקוד של השני ללא שום הגבלה, זה פתרון עיצובי שקל למימוש אבל לא תמיד יעיל.

- ה Scroll view - אנו נראה כי בתוך הליניארי ניתן להגדיר scroll, ז"א רשימה שניתן לגלול בה.

תכנות מונחה אירועים: מתודות שיפעלו בתגובה ל"אירוע" שקורה. דוגמאות לכך:

- ה onClick() - מתוך view.onclicklistener פונקציה שנקראת כאשר לוחצים על פריט מסוים שזה מוגדר לו. אין לזה ערך חזרה.
- ה onLongClick() - מתוך view.onLongclicklistener . נקראת כאשר עושים לחציה ארוכה. מחזיר בוליאני.
- ה onFocusChange() - כאשר המשתמש מנווט רחוק מהפריט.
- ה onKeyDown() - כאשר משתמש מתמקד על פריט לוחץ על כפתור פיזי במכשיר.
- ה onTouch() - כאשר משתמש נוגע במסך מקבלת כפרמטר MotionEvent. מחזירה ערך בוליאני.
- ה onOptionsItemSelected() - כאשר משתמש ילחץ על פריט בתפריט. מחזיר בוליאני.
- ה onCreateContextMenu() - נקראת כאשר תפריט ההקשר עבור תצוגה נבנה.

דרכים שונות להגדיר EVENT:

1. מחלקה של Activity תיישם את הממשק של Listener.
2. שימוש ב class פנימי אנונימי.
3. שימוש ב layout בקובץ activity_main.xml כדי לציין ישירות את מטפל האירוע.

:Context

כאשר יוצרים אובייקט חדש אנדרואיד רוצים לתת לו את תמונת העולם שהייתה לפני היצירה שלו.
השימוש של context:

- לתת מידע על חלק אחר של האפליקציה כמו על אקטיביטי הקודם
- נקודת גישה למשאבי המערכת כמו תמונות DB וכו'.

סוגי context:

- Activity context
- Application context

פונקציות של context:

- `view.getContext()` - לקבל context בתוך activity קיים.
- פחות בשימוש:
- `contextwrapper.getBaseContext()` - לקבל context מתוך context אחר.
- `Activity.getApplicationContext()` - לקבלת context של כל האפליקציה.

תכנות דינאמי:

- לעצב דרך ה XML.

Shared Preference

מאפשר שמירת מידע מצומצם בקובץ במנה של key value (קובץ XML)

- ניתן לשמור נתונים הסוגים הבסיסיים.
- לכל Activity יש גישה לקובץ.
- לכל המשתמש אותו קובץ.

מחזור החיים של Activity - אילו מתודות יופעלו:

- `onCreate` - שנוגרת האפליקציה בעזרת מערכת ההפעלה ונכנס שוב אליה.
- `onStart` - כאשר סוגרים לבד ופותחים שוב.
- `onPause` - כאשר יש פרסומת שעוצרת ולא מאפשר להמשיך עד שלא מגיבים לפרסומת.
- `onDestroy` - מוחק את כל הנתונים החשובים בנוסף גם ניתן לחשב סטטיסטיקה בזמן הזה.

Back Stack

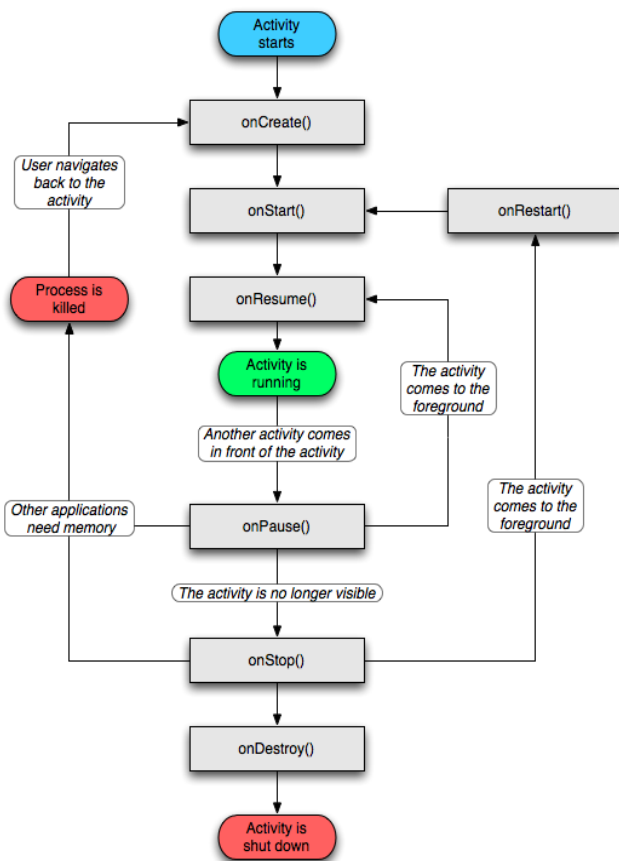
כל המסכים מסודרים במחסנית, המסך הנוכחי נמצא בראש. נועד כדי לחזור אחורה או להוציא אם רוצים לא לאפשר חזרה לאחור.

:Toasts

הודעה פשוטה שקופצת למשתמש.
המסך נשאר אינטראקטיבי היא לא מפריעה.
נעלמת לאחר זמן קצר.
ניתן ליצור תגית משל עצמו בנפרד ולא חייב שיהיה לו קובץ JAVA.

Dialogs

חלון חדש שמנחה משתמש לקבל החלטה או להזין מידע.
בדרך כלל קטן לא ממלא את כל המסך.



סוגי שונים:

- Alert
- Progress
- Date
- Time
- Custome

Manifest

המקום שבו האפליקציה מגדירה את הצרכים שלה.
לכל אפליקציה יש קובץ אחד כזה.
מכיל הצהרות:

- Activities
- Intents
- Intent filter
- Permissions
- ועוד

בקובץ יש גם את אבי הבניה של אנדרואיד:

- Content provider
- Broadcast
- Service
- Intent
- Activity

INTENT

שימושים:

- מעבר בין מסכים באפליקציה
- לאתחל service
- שידור של הודעה להרבה רכיבים broadcast
- לפנות לאפליקציה אחרת בהתקן

סוגים של intent:

- Explicit intent - ה intent יעבור לרכיב שצוין במפורש. יכול להיות מסך אחר או רכיב אחר.
 - Implicit intent - מערכת ההפעלה תעבור לרכיב המתאים או התאפשר למשתמש לבחור. אנחנו לא מעבירים את הפרמטר של ה component data לכן בשביל שהמערכת תדע את מה לפתוח צריך לתת לה רמזים.
- רמזים אלו הם intent filter והם יכולים להיות מסוג : action,category,data.

Permissions

הרשאת גישה של האפליקציה לרכיבים אחרים במכשיר.
סוגים:

- הרשאות רגילות - אינטרנט בלוטוס WIFI
- הרשאות מסוכנות - מצלמה שטח אחסון אנשים קשר מיקום...

Sensors

חיישנים שיכול למדוד דברים או לתת מידע.

סוגים לדוגמה:

- תנועה
- סביבה
- מיקום

מה ניתן לעשות עם סנסורים?

- למצוא זמינות סנסורים במכשיר
- למצוא יכולת של חיישן ספציפי
- להשיג נתונים ממנו
- שימוש ב event listener לצורך מעקב אחרי החיישן.

On Sensor Changed:

- זה Event שקורה כאשר יש קריאה חדשה יותר מהחיישן.

ניהול חיישנים לפי התקן:

בכל התקן יכולים להיות חיישנים שונים ולא נוכל לדעת בדיוק איזה חיישנים קיימים במכשיר, לכן נפלט לפי google play.

User Feature

בקובץ המניפסט נגדיר את החיישנים את החיישנים שמשתמשים בהם באפליקציה.
זה יהיה הפילטר שלנו בgoogle play.
במניפסט נגדיר את רמת הצורך בחיישן (חיוני או אופציונאלי).

Notification

רכיב המאפשר להציג הודעה מחוץ לאפליקציה.

Broadcast Receiver

מחלקה שיכולה לקבל הודעות מאפליקציות אחרות או ממערכת הפעלה. ניתן לקבל דרך הקוד או דרך קובץ המניפסט אבל צריכה להיות הרשאה בשביל זה בקובץ המניפסט. צריך לכתוב מחלקה שממשת broadcast ולתפעל דרכה.

Gradle

מערכת קוד פתוח לקימפול קוד
Flavor יש אפשרות לתת מאפליציה אחת כמה אפליקציות עם גרסא אחרת. לדוגמא אפליקציה לכל מדינה, גרסא חנימית וגרסא בתשלום וכו..
Type זה נתונים על הדיאבג ועל הקבצי הרצה והכיוון שלהם.
Variants נוצר משילוב של האפשרויות של טייפ ופלוור. פלוור אומר לי כל גרסא וטייפ יוצר לי רליס ודיבאג
לדוגמא freeHebrawDebug
variantsFilter אפשר לסנן שילובים מסוימים.