# הנדסת תוכנה - הקדמה

## תוכנת מחשב

תוכנת מחשב היא אוסף של הוראות מחשב המאפשרות למשתמש לתקשר עם המחשב, עם החומרה שלו או לבצע משימות. תוכנה טובה היא תוכנה שמבצעת את מה שהיא אמורה לבצע, יש לה ממשק נוח ויפה עם המשתמש, וכתובה בצורה נקיה וברורה כך שקלה לבדיקה ותחזוקה.

מתברר שלא מעט פרויקטים ליצירת תוכנה נכשלים. הסיבות לכך הם:

* יעדי הפרויקט לא מציאותיים או שהדרישות מהתוכנה אינן מוגדרות היטב.
* אומדנים לא מדויקים של המשאבים הנדרשים ליצירת התוכנה.
* שימוש בטכנולוגיה לא בשלה או פרקטיקות פיתוח מרושלות.
* ניהול לקוי של הפרויקט.
* אי התמודדות עם בעיות שצצות ודיווח לקוי לגבי מצב הפרויקט.
* תקשורת לקויה בין הלקוח, המפתחים והמשתמשים.
* לחצים מסחריים להוציא מהר מוצר לשוק למרות שאינו טוב מספיק.

בפיתוח תוכנה יש 2 סוגי מורכבויות לפתור:

**מורכבות מקרית** - נוצרת עקב כתיבת תוכנה לא יעילה או כתיבת באגים. אם נצליח להיפטר מהמורכבות המקרית נוכל לשפר את הפרודוקטיביות של התוכנה. כלים שיכולים לעזור לנו לצמצם את המורכבות המקרית הם: סביבות עבודה מתקדמות, שפות עיליות, שיטות עבודה מתקדמות, בדיקות אוטומטיות וכו'.

**מורכבות מהותית** - הדרישות מהתוכנה הן לא פשוטות וקשה מאוד ליישם אותן. למורכבות זו אין פתרון קסם מפני שהמורכבות טבועה בבעיה עצמה. יש מספר דרכים כדי להתמודד עם מורכבות זו: למצוא מהנדסי תוכנה טובים, להשתמש כמה שניתן במוצרי מדף שהם בדוקים וידועים שעובדים, חידוד הדרישות, בניית אב טיפוס, פיתוח התוכנה בצורה מסודרת ומובנת לפי שלבים מוגדרים. השיטה האחרונה היא עיקר התחום של הנדסת תוכנה עליה נלמד בקורס זה.

## חלוקת רמות במערכת עתירת תוכנה

מערכת עתירת תוכנה היא מערכת אשר התוכנה מהווה בה חלק משמעותי מבחינת הפונקציונליות, עלות הפיתוח, סיכוני הפיתוח או משך הפיתוח. כל מערכת כזו נפרק לרמות הבאות:

* רמת הארגון - שאליו המערכת נותנת שירות. לדוגמה קופת חולים.
* רמת המערכת - זוהי המערכת שתפקידנו לבנות. בכל ארגון יש מספר מערכות. לדוגמה, מערכת ניהול שירותי הקופה.
* רמת פריט התוכנה - כל מערכת מורכבת מפריטי תוכנה ופריטי חומרה. פריט תוכנה זוהי תת-מערכת בתוך המערכת הכללית. לדוגמה, מערכת ניהול התורים בתוך מערכת ניהול שירותי הקופה.
* רמת רכיב התוכנה - כל פריט תוכנה מורכב ממספר רכיבי תוכנה. לדוגמה, ממשק המשתמש בתוך מערכת ניהול תורים.
* יחידת תוכנה - כל רכיב תוכנה מורכב ממספר יחידות תוכנה. כל יחידת תוכנה היא תהליך במערכת. לדוגמה, חלון ההזדהות בממשק המשתמש.

בכל אחד מהאלמנטים ברמות השונות לעיל מוגדרים שלושה מאפיינים:

1. תיחום - מה נכלל באלמנט ומה לא נכלל בו, ועם מי הוא מתממשק. הממשקים יכולים להיות למערכות מחוץ לארגון או למערכות אחרות פנימיות בתוך הארגון. בדרך כלל, ככל שרמת האלמנט עמוקה יותר כך הממשקים יהיו פנימיים לארגון ולא חיצוניים.
2. מבנה - מהם המרכיבים של כל אלמנט פיזיים ולוגיים.
3. תפקוד - מהם התהליכים שמבצע ומהי האינטראקציה שלו עם הסביבה.

דוגמא לכל המאפיינים בכל רמה של המערכת ניתן למצוא במצגת 1.2 שקופיות 4-10.

## הנדסת תוכנה

הנדסת תוכנה מתייחסת למחזור חיים של הפיתוח, החל משלב הרעיון, דרך התיכנון, הפיתוח, הבדיקות והתחזוקה של מערכת התוכנה. מטרות הנדסת תוכנה:

* הקטנת המורכבות שבפיתוח תוכנה.
* לשפר את אמינות התוכנה (פחות באגים).
* להקטין את עלויות התחזוקה.

הנדסת התוכנה מחלקת את תהליך פיתוח התוכנה לשלבים מובנים ומוגדרים כך שכל שלב הוא נדבך נוסף על השלב שקדם לו. כך התוכנה גדלה ומתפתחת באופן הדרגתי, כמו בן-אדם. השלבים בפיתוח התוכנה הם:

* **שלב הייזום**: הגדרת הצורך ליצירת התוכנה וכיצד היא מספקת צורך זה.
* **שלב הדרישות**: לפרט מהם הדרישות אותם המערכת צריכה לבצע.
* **שלב הניתוח**: ניתוח כל הדרישות ולראות במה הן כרוכות. יצירת תרשימים פשוטים אותם ניתן להציג ללקוח.
* **שלב התוכן/עיצוב**: תכנון כיצד המערכת צריכה להיבנות ולהיראות בכל הרמות שלה.
* **שלב המימוש**: מימוש כל רמות התוכנה בקוד.
* **שלב הבדיקות**: וידוא שהתוכנה עובדת ויעילה.
* **שלב השילוב**: של התוכנה במערכת הגדולה.
* **שלב התחזוקה**: ניהול גרסאות, פתירת באגים, ייעול ושיפור.
* **שלב הפרישה**.

# שלב הייזום

## הגדרה

זהו השלב הראשוני שבו מועלית הדרישה הבסיסית ליצירת מערכת תוכנה חדשה. המטרה של שלב זה היא לבדוק האם יש צורך במערכת התוכנה, האם היא משתלמת, לוודא שאנו אכן מסוגלים ליצור אותה, ולשרטט קווים כלליים כיצד לעשות זאת. הסעיפים הבאים מתארים את סדר השלבים לכתיבת "מסמך הייזום" ומה יש לעשות בכל שלב.

## הגדרת הלקוח

בשלב זה מנסים לזהות מיהם הלקוחות הפוטנציאליים שיש סיכוי שיהיו מעוניינים במערכת תוכנה ומהו המוטיבציה/צורך שהמערכת מיועדת לספק אצלם. כמו כן, יש לזהות מהו השוק בו המערכת שלנו פועלת.

## מציאת מומחה יישום

מומחה היישום הוא גורם מוסמך האחראי על שמירת האינטרסים של הלקוחות הפוטנציאליים. הוא יכול להיות ממונה על ידי הלקוח או מתוך הארגון שמפתח את המערכת. התפקידים של מומחה היישום הם:

* מרכז את דרישות הלקוח מהמערכת.
* מלווה את המערכת משלב הייזום ועד הטמעת המערכת אצל הלקוח.
* מקשר בין הלקוחות השונים ומכריע במקרה של התנגשויות בדרישות מהמערכת.

## ניתוח מצב קיים, בדיקת מערכות דומות קיימות

נרצה לבדוק האם קיימות מערכות דומות ומה אנו יכולים ללמוד מהן. במידה וקיימות מערכות דומות, יש ללמוד מה היתרונות וחסרונות שלהן, האם יש מקום למערכת חדשה, אילו בעיות היו בתהליך הפיתוח שלהן. במידה ואין מערכת דומה לברר האם היה ניסיון להקים מערכת כזו ומדוע הניסיון לא צלח.

עבור מערכת המיועדת עבור ארגון ספציפי, יש לברר בנוסף: מהם מאפייני הארגון, מה המצב הנוכחי בארגון ללא המערכת, האם יש מערכת דומה בארגון אחר.

## הגדרת מטרות ויעדים

יש להגדיר מהם המטרות והיעדים של המערכת תוכנה.

מטרות - פירוט המטרות הכלליות אותם אנו רוצים להגשים באמצעות המערכת.

יעדים - פירוק המטרות למשימות קטנות יותר אותם ניתן לכמת. בשלב הבא יעדים אלו יהפכו לדרישות מדויקות ומפורטות.

בשביל שנוכל לוודא שהיעד מוגדר כמו שצריך, ניתן להשתמש במודל S.M.A.R.T הכולל 5 תכונות שצריך לוודא שמתקיימות בכל יעד שהגדרנו.

Specific - יעדים ממוקדים ככל הניתן, ולא כלליים מדי.

Measurable - יעדים הניתנים למדידה.

Attainable - יעדים ברי השגה, ריאליים והגיוניים.

Relevant - יעדים המשרתים את הייעוד והאסטרטגיה הארגונית. יעדים משמעותיים לארגון.

Time bound - יעדים התחומים בזמן. כך אנחנו מוודאים שיש מסגרת זמנים שמאפשרת לנו מעקב קבוע.

בפועל, כיוון שמאוד קשה שכל יעד יעמוד בתכונות אלו, אפשר להסתפק רק בתכונות Measurable ו-Time bound, שבמידה מסוימת כוללות גם את שאר התכונות.

## הגדרת אילוצים וסיכונים

לנתח בעיות שעלולות לנבוע מסוגי האילוצים הבאים:

בתחום התפעול - כיצד יש לפתח את המערכת כדי שתתאים למי שאמור לתפעל אותה לבסוף.

טכנולוגיים - מהן ההשלכות של מערכות הפיתוח בה נבחר להשתמש.

ארגוניים - האם יכולות לעלות בעיות מדרישות הלקוח.

זמן ומשאבים - האם נוכל לעמוד ביעד הזמן ולא לחרוג מהמשאבים והתקציב שניתנו לנו.

ניהול סיכונים: לנתח כל מה שיכול לקרות שיקטין לנו את הסיכוי להצלחה ולציין כיצד נוכל למנוע מהסיכון להתממש.

## הגדרת תועלות וחסכונות

יש כמה רמות בהן ניתן לנתח את התועלת:

כוח אדם - העסקת פחות אנשים בארגון שאותם תחליף המערכת.

ארגון ושיטות - המערכת מייעלת תהליכים שננקטו עד כה.

כלים וטכנולוגיה - שיפור ברמת השירות והנגישות.

מידע - המערכת מספקת מידע חשוב או משפרת את אמינות ותקינות המידע.

כל אחת מרמות התועלת לעיל ניתן לחלק לתועלות מוחשיות, הניתנות לכימות ומוגדרות במונחים כספיים, ותועלות לא מוחשיות שלא ניתנות לכימות.

## בדיקת ישימות ועלות מול תועלת

לאחר שהגדרנו מהם האילוצים והסיכונים וכן מהם התועלות הצפויות מהמערכת, ננתח האם ניתן וכדאי לפתח אותה.

בדיקת ישימות - נברר האם אנו מסוגלים בכלל לפתח את המערכת. האם יש לנו את הכלים הטכנולוגיים לפיתוח המערכת והאם יש לנו מספיק זמן, תקציב ומשאבים לשם כך.

בדיקת עלות מול תועלת - גם במידה ויש לנו את כל הכלים הנצרכים לא בהכרח שכדאי לנו לפתח אותה. נלמד ארבעה שיטות כדי לשערך מהי העלות והזמן הצפויים של המערכת ולאחר מכן נפרט שלושה שיטות לבדוק כדאיות כלכלית.

### שיטות לשערוך עלויות וזמן :

1. Analogous Estimating - מחשבים עלויות בהשוואה לעלויות אמת של מערכות דומות שכבר פותחו.
2. Parametric Estimating - מפרקים את המערכת לחלקים וכל חלק משווים לעלות אמת של חלק שכבר פותח במערכות אחרות. בסוף מחברים הכל ביחד.
3. Three Point Estimates - כמו הקודם רק שלכל חלק נותנים 3 הערכות: אופטימי, ריאלי ופסימי.
4. Bottom-up Estimating - מפרקים את המערכת לחלקים מאוד קטנים שאותם אנו יודעים בוודאות את העלות שלהם ואז מחברים. שיטה זו היא בדרך כלל המדויקת ביותר.

ככלל, ככל שעובדים יותר קשה בשלב זה כך מגיעים לתוצאות מדויקות יותר.

### שיטות לבדיקת כדאיות כלכלית:

1. NPV (Net Present Value) - בשיטה זו מחשבים כדאיות כלכלית עם התחשבות בכך שההכנסות יגיעו בעתיד. נוסחת החישוב היא: . כאשר N הוא מספר השנים בחישוב (בדר"כ 5), הוא הרווח (הכנסות פחות הוצאות) בשנה t, ו-i הוא מקדם ההיוון, שהוא אחוז הריבית שניתן להרוויח ע"י השקעה של הכסף המיועד לפיתוח המערכת בבורסה או בכל מקום אחר.
2. ROI (Return on Investment) - מחשבת את אחוז הרווח מהמערכת ביחס להוצאה. הנוסחה: . אחוז זה נותן לנו תפיסה כמה הפרויקט משתלם. ככלל נהוג להגדיר פרויקט שבו מתקבל אחוז רווח של 50% כמשתלם.
3. Payback analysis - מציגים בגרף את פונקציית ההוצאות ופונקציית ההכנסות. נקודת המפגש, אם יש כזו, היא הנקודה בזמן שבה מתחילים באמת להרוויח מהמערכת.

## גיבוש צוות התכנון של המערכת (ועדת היגוי)

ועדת היגוי (committee steering) היא ועדה מייעצת שהחברים בה הם בדרך כלל בעלי עניין או מומחים. הועדה הכוללת נציגים מתחומי ידע שונים כמו: טכנולוגיה (מחשוב, תשתיות, DB), משאבי אנוש, כספים, ונציגי הלקוחות השונים. הועדה מתגבשת בשלב הייזום וממשיכה בעבודתה עד העלאת המערכת לאוויר. תפקידיה הם:

* קביעת מדיניות פיתוח.
* מעקב ובקרה אחרי לוח הזמנים והעלויות של המערכת.
* פתרון בעיות במהלך הפיתוח.
* שינויים בשלב הפיתוח.
* קבלת החלטות בשלבים השונים של פיתוח המערכת.
* יצירת מעורבות ומחויבות אצל הלקוחות ואנשי הפיתוח.

## קביעת לוח זמנים ומשאבים

מחלקים את פיתוח המערכת לשלבים מסודרים. לכל שלב נותנים זמן מוגדר שעד אליו צריך לבצע אותו וכן מהם המשאבים המוקצבים לכל שלב.

בסיום שלב הייזום מתקבל "מסמך הייזום" המועבר לוועדת ההיגוי והם מחליטים האם להמשיך בשלבי הפיתוח, לבצע תיקונים או להקפיא ואף לבטל את פיתוח המערכת.

# שלב התכנון

## הגדרה

שלב התכנון הוא השלב המהותי ביותר בפרויקט. הצלחה בתכנון תוביל לסיכוי רב הן לעמידה ביעדי הפרויקט, שביעות רצון לקוח, עמידה בלוחות הזמנים, עמידה באיכות ועוד.

שלב זה מתחיל במהלכו של שלב הייזום ומסתיים לקראת סיום הפרויקט. יש להקפיד להשלים את התכנון הנדרש ביסודיות (דילוג על שלב זה יוביל בפועל לכישלון בביצוע).

שלב זה כולל בניית תוכניות עבודה רבות, מומלץ שיהיו תוכניות לנושאים הבאים:

* תכנון תכולת העבודה, הן תכנון-על והן תכנון פרטני של חבילות העבודה בפרויקט, חלוקת אחריות וכיצד לעדכן על שינויים.
* תכנון לוחות זמנים, משימות לביצוע, סדר משימות, תזמון למשימות, תלויות, אילוצים ונתיב קריטי.
* תכנון עלות - הכנת תוכנית תקציב והערכות כספיות.
* אפיון איכות המוצר או השירות וכיצד לוודא שתתקבל האיכות הרצויה.
* תוכנית איוש כ"א - בניית הצוות המקצועי שיוביל את הפרויקט לסיומו המוצלח.
* תכנון תקשורת - קישור מוצלח של כלל הגורמים לעבודה משותפת.
* ניהול סיכונים - איתור סיכונים, הערכת סיכונים על סך סבירות והשפעה, החלטה במה לטפל וכיצד.
* תכנון רכש - האם לרכוש או לייצר? מי הקבלן שיופעל? חתימת חוזים ועוד.
* תוכנית אינטגרציה - כיצד ינוהל הפרויקט בפועל וכיצד כל מרכיביו משתלבים אלה באלה.

# שלב הדרישות

## הגדרה

בשלב זה מפרקים את פיתוח מערכת התוכנה שלנו לדרישות קטנות ומפורטות. "דרישה" היא תכונה או יכולת הנדרשת ממערכת התוכנה כדי לפתור בעיה, להשיג מטרה או כדי לעמוד באיזה תקן מחייב. דרישה גם יכולה לתאר מה המערכת אמורה לעשות או איך המערכת אמורה לעבוד.

הגדרת מסמך דרישות הוא שלב חשוב מאוד מכמה סיבות:

* בכל פיתוח מערכת יש מספר בעלי עניין (משתמשים, לקוחות, מפתחים, מנהלים וכו') במערכת התוכנה שכל אחד מהם עלול לראות את המערכת בצורה אחרת, לחיוב ולשלילה. הגדרת מסמך דרישות מונע מבעלי עניין למשוך כל אחד לכיוונו, שכן כבר בשלב ראשוני אנחנו מגדירים בדיוק למה אנו מצפים על בסיס מנותח, מוסכם ומאושר. כתוצאה מכך יש סיכוי גבוה יותר שהמערכת תענה על דרישת הלקוח.
* ממקד את תהליכי האפיון והפיתוח של המערכת.
* שיפור היכולת לבצע בקרה על התקדמות הפרויקט.
* משפר את היכולת לבצע שינויים במהירות תוך בקרה ובחינת השפעתם על דרישות אחרות.
* שיפור היכולת לביצוע אמידת עלויות ע"י ניתוח של עלות מול תועלת עבור כל דרישה בנפרד או עבור מכלול דרישות.
* פירוט הדרישות בהתחלת הפיתוח ימנע באגים וטעויות בהמשך. ככל שמאתרים באג בשלב מוקדם יותר כך עלות התיקון שלו קטנה יותר, כמו שניתן לראות בתוצאות מחקר של IBM.

## סוגי דרישות

### דרישות פונקציונאליות

אלו דרישות המפרטות מה המערכת אמורה לעשות/להגיב מנקודת המבט של המשתמש. ככלל, לדרישה פונקציונאלית ניתן לכתוב פונקציה שתענה עליה. דרישות אלו מקבלות מענה עם מימוש התוכנה. יש שני סוגי דרישות פונקציונאליות:

* דרישה תפעולית - דרישה המתייחסת לתפעול, לאינטראקציה או להתנהגות של המערכת (נפח אחסון, ניצול מעבד, וכו'). לדוגמא: פעולות, תרחישים, תגובות לאירועים וכו'.
* 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 פונקציונאליות

אלו דרישות המגדירות מאפיינים ואילוצים על הדרישות הפונקציונאליות. בדר"כ עוסקות באיכות המערכת. דרישות אלו מקבלות מענה כאשר המימוש עומד במאפיינים והאילוצים שהן מצריכות. יש מספר רב של סוגי דרישות לא פונקציונאליות:

* התאמה פונקציונאלית (Functional suitability) - המערכת צריכה לכלול את כל הרכיבים הנצרכים כדי לבצע את המטרה שפותחה בשבילה.
* יעילות ביצועים (Performance efficiency) - מערכת התוכנה רצה מהר, כמות המשאבים (זיכרון ומעבד) שאנו צורכים.
* תאימות (Compatibility) - כמה המערכת יכולה להתממשק ולעבוד בסנכרון עם מערכות נוספות.
* שימושיות (usability) - תרומת המערכת למשתמש בהשגת מטרותיו, ממשק משתמש נוח והבנה מהירה איך להשתמש.
* אמינות (reliability) - פועלת ללא תקלות לאורך זמן.
* דרישות ביצועים (Performance Requirements) - פרמטרים ניתנים למדידה לגבי ביצועי התוכנה.
* זמינות (availability) - שירות רצוף, התאוששות מהירה מתקלות, תפקוד על-אף בעיות.
* בטיחות (safety) - שמירה על חייהם ובריאותם (הפיזית והנפשית) של מפעילים ומשתמשים.
* בטחון (security) - הגנה על מרכיבי המערכת ועל המידע בה היא מטפלת.
* אחזקתיות (maintainability) - היכולת לערוך בקלות שינויים ותיקונים במוצר, עד כמה קל לבדוק את המערכת.
* ניוד (Portability) - מידת האפקטיביות שבה ניתן להתאים את המערכת מסביבה תפעולית אחת לאחרת.
* מאפייני איכות -
* אילוץ ניהולי – דרישה הנוגעת לתקציב, לו"ז, זמינות משאבים, התאמה לתקן וכו'.
* אילוץ חומרה – דרישות הנוגעות לרכיבי החומרה במערכת. תחזוקת רכיבים לדוגמה.
* אילוץ מימוש – באיזה אלגוריתם או מבנה נתונים להשתמש במערכת.

## שיטות למציאת דרישות

בעיות אופייניות במציאת דרישות הן:

* המשתמשים לא יודעים להסביר מה הם רוצים או חושבים שהם יודעים עד שהם רואים את התוצאה בעיניים.
* המפתחים חושבים שהם יודעים מה המשתמשים רוצים אך בפועל הם לא.

כדי להתגבר על בעיות אלו מומלץ לנקוט בשיטות הבאות למציאת דרישות:

* ראיונות ושאלונים לבעלי העניין השונים. מנסים לגלות ולתחקר מהם המניעים והמטרות שלהם.
* מומחים - הזמנת מומחים בתחום שיודעים לזהות את הדרישות הנצרכות.
* סיעור מוחות - צוות הפיתוח מעלה רעיונות לדרישות.
* למידה ממערכת קיימת או מערכות קודמות איזה דרישות היו טובות ואיזה יש לשפר.
* שימוש בתרשימי ניתוח כמו UML ועוד.
* יצירת demo והתנסות עליו.

## מהי דרישה איכותית

Identified - ממוספרת ונבדלת מדרישות אחרות.

Understandable - מובנת, ברורה ומדויקת. מנוסחת בשפת הלקוח.

Unambiguous - לא עמומה אלא חד משמעית.

Complete - לא חסר בה פרטים.

Necessary - בעלת תרומה משמעותית לשיפור תהליכי העבודה.

Consistent - לא סותרת דרישות אחרות.

Verifiable - ניתנת לבדיקה באמצעות מבחני קבלה.

Traceable - עקיבה גם לדרישות ברמה גבוהה יותר וגם בהמשך האפיון.

Prioritized - מתועדפת לפי רמת חשיבות למערכת.

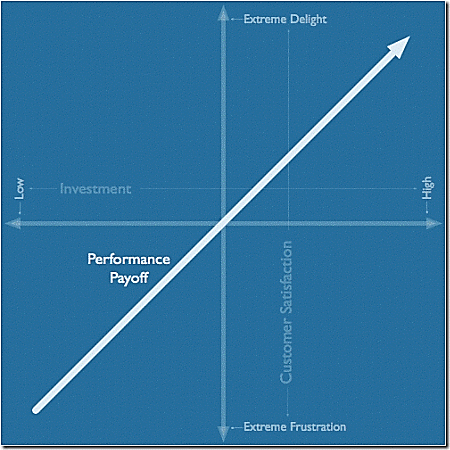
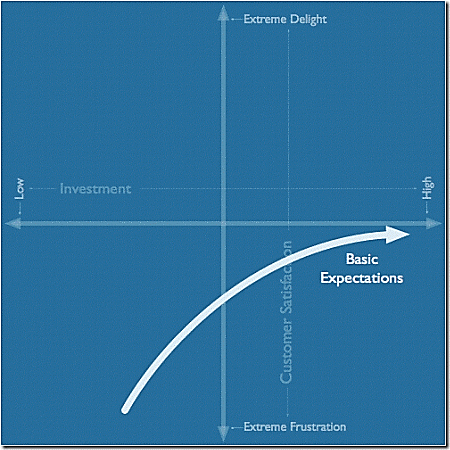
Short - אם מנוסחת ארוך מדי סימן שיש כאן כמה דרישות וצריך לפצל אותן.

## מודל Kano

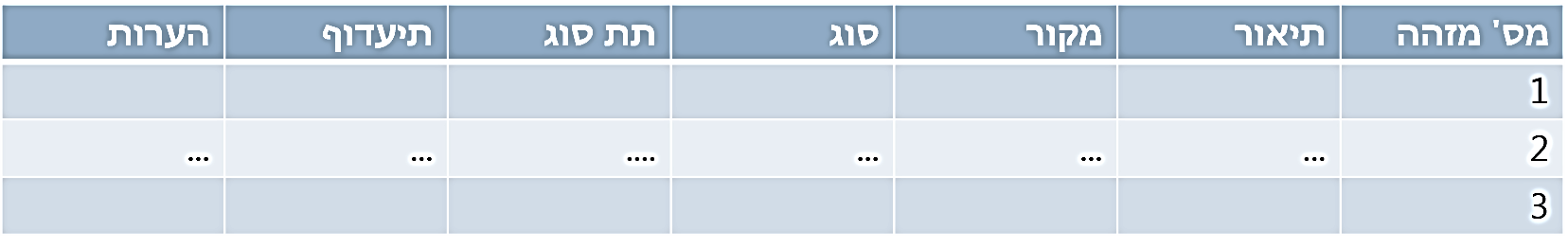
כל דרישה למערכת משמעה עלויות כספיות, סיבוכיות למערכת וניהול באגים, לכן לא נרצה להוסיף דרישות שאינן חשובות. כפתרון לכך יש לדרג את הדרישות לפי תיעדוף כדי לדעת אילו חשובות ואילו פחות. מודל קאנו עוזר לנו במשימה זו.

מודל קאנו זהו מודל שבאמצעותו ניתן להעריך כמה כדאי להשקיע בדרישה ביחס לתפוקה שנובעת ממנה המתבטאת בשביעות רצון הלקוחות. המודל מתאר גרף עם שני צירים. ציר ה-X מייצג את כמות ההשקעה בדרישה, ככל שהערך גדול יותר כך כמות ההשקעה גדלה. ציר ה-Y מייצג את רמת שביעות רצון הלקוחות. באופן כללי, ערך שלילי בציר ה-Y מייצג אי-שביעות רצון, ערך חיובי מייצג שביעות רצון, וערך 0 ניטרלי. ככל שהערך גדול יותר כך רמת שביעות הרצון גדלה.

יש שלושה סוגים של דרישות המתבטאות בצורה שונה במודל:

1. השתלמות הביצוע - דרישות שאם לא מסופקות מביאות לחוסר שביעות רצון וככל שמספקים אותן יותר כך שביעות הרצון עולה באופן ליניארי.
2. ציפיות בסיסיות - אלו דרישות שהלקוח מצפה להן. אם לא מסופקות מביאות לתסכול אצל הלקוח, אך אם כן מסופקות אינן מייצרות שביעות רצון גבוהה שכן זוהי דרישה בסיסית.
3. מייצרי הנאה - דרישות שהלקוח לא ציפה להן. אם לא מסופקות אינן יוצרות אי-שביעות רצון, אך אם מסופקות מייצרות שביעות רצון אצל הלקוח.

## מה מכיל מסמך הדרישות

לכל דרישה יש לתת מזהה ייחודי שנשאר גם אם הדרישה נמחקת, תיאור ברור, מי בעל העניין או מסמך שהעלה את הדרישה, סוג הדרישה (פונקציונאלית או לא), תת-סוג, תיעדוף על ידי מספר המייצג את העדיפות (לדוגמה מספר מ-1 עד 5, כאשר 1 גבוה ו-5 נמוך), והערות שונות (כמו רמת קושי, עלות, דגשים וכו').

# שלב הניתוח

## הגדרה

בשלב זה מנתחים את הדרישות משלב קודם ויוצרים מודלים (דיאגרמות) המתארים את המערכת מכמה זוויות שונות. מטרת דיאגרמות אלו היא לעזור ללקוחות ולמפתחים להבין את המערכת בקלות.

## UML

כדי ליצור את המודלים משתמשים בשפת UML (Unified Modeling Language) שהיא שפת סימולים גרפית וטקסטואלית לעיצוב מערכת מונחת עצמים. בגרסה הנוכחית (2.5) מספקת 13 סוגים שונים של דיאגרמות שכל אחת מתארת היבט אחר במערכת המתוכננת. יש מספר תוכנות ליצירת מודלים של UML כמו: Visio, SmartDraw, Rational rose ועוד.

היתרונות של UML שהיא שפה מאוד קלה ואינה מצריכה ידע מוקדם גדול כדי להבין אותה, ובנוסף היא מתארת את המערכת מנקודת המבט של המשתמש. מסיבות אלו היא מאוד נוחה כדי להסביר הן ללקוחות והן למפתחים את המערכת בצורה ברורה וקלה. החסרונות של UML שריבוי מודלים מהיבטים שונים לאותה פעולה עלול לסבך את ההסבר שלה. בנוסף, עלולה ליצור קושי במידול מערכות מורכבות.

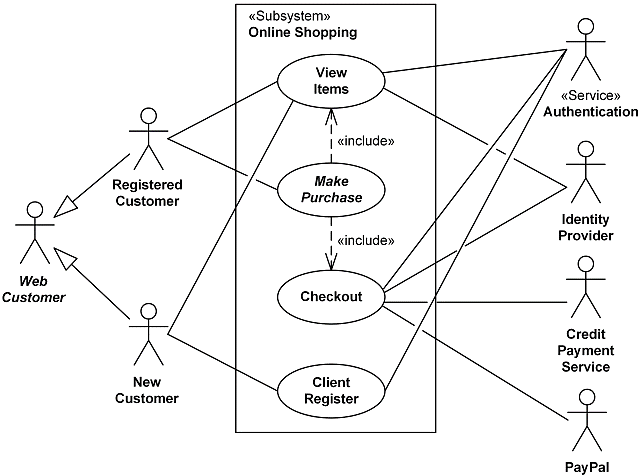
### סוגי מודלים

ניתן לחלק את המודלים הקיימים ב-UML לשני סוגים: דיאגרמות המתארות מבנה של המערכת ודיאגרמות המתארות כיצד המערכת מתנהגת. ניתן גם לחלק אותן לדיאגרמות המתארות מצב סטטי לדיאגרמות המתארות את המערכת לאורך זמן. אנו נתמקד רק בשש דיאגרמות עיקריות, השאר הן פחות חשובות או לא שימושיות במיוחד.

## Use-case Diagram

דיאגרמה המתארת את כל הפעולות שיש להן תוצאות גלויות שהמערכת מבצעת ומה הקשר בין פעולות אלו לישויות שהמערכת מתממשקת איתן. מטרת דיאגרמה זו היא להראות את כל הפונקציונאליות של המערכת וכיצד מתקשרת עם ישויות חיצוניות לה. הפעולות שהמערכת מבצעת נקראות "use-case" והישויות החיצוניות נקראים "שחקנים". זו הדיאגרמה הכללית ביותר של המערכת ולכן בדרך כלל ניצור אותה ראשונה. היא צריכה להיות פשוטה וברורה. אם המערכת מבצעת המון פעולות יש ליצור דיאגרמת use-case לכל קבוצת פעולות שיש קשר ביניהן.

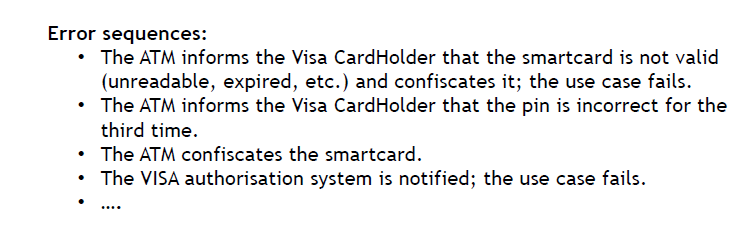
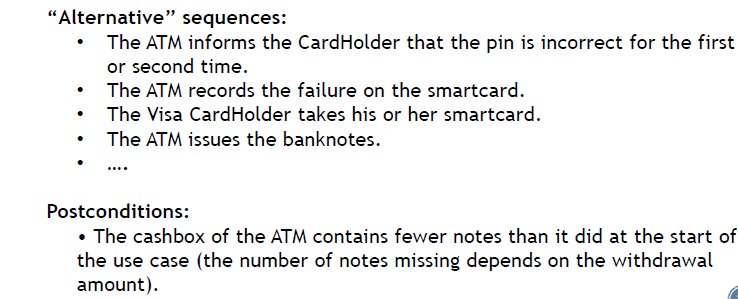
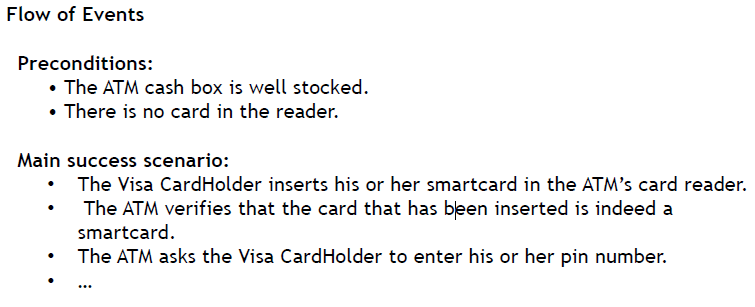
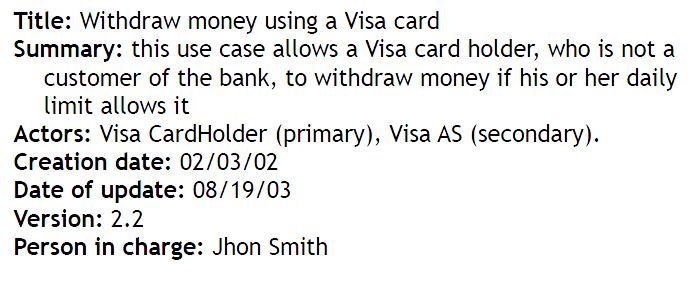
### שלבים ביצירת הדיאגרמה:

1. נעשה רשימה של כל השחקנים שמתממשקים עם המערכת שלנו. שחקנים הם ישות חיצונית כלשהי שיש לה קשר עם המערכת. נבדיל בין שחקנים ראשיים למשניים. שחקן ראשי הוא שחקן שמשתמש במערכת שלנו באופן עצמאי ואילו שחקן משני הוא כזה שבדרך כלל לא משתמש במערכת אלא במקרים מסוימים או שהמערכת משתמשת בו.
2. נבחן, מתוך מסמך הדרישות, מהם כל הפעולות של המערכת שלשחקנים שפירטנו יש גישה אליהן.
3. ניצור את הדיאגרמה בדרך הבאה: הפעולות (use-case) יהיו באמצע בתוך מלבן, שחקנים ראשיים משמאל ושחקנים משניים מימין. נמתח קו המייצג קשר בין השחקנים אל הפעולות שיש להם גישה אליהן. אם אין לשחקן גישה חופשית לפעילות נסמן קו בלי חץ. אפשר לציין גם multiplicity על הקווים, כלומר כמה שחקנים מסוג זה יכולים לגשת לפעולה.
4. נוסיף קשרים בין שחקנים כמו ירושה המתבטאת בחץ עם משולש ריק.
5. נוסיף קשרים בין פעולות. יש שני סוגים של קשרים: include - כאשר פעולה A תמיד משתמשת בפעולה B, נמתח קו מקווקו עם חץ מ-A ל-B. extend - כאשר פעולה A יכולה להשתמש בפעולה B אך לא תמיד, נמתח קו זהה מ-B ל-A.

### תיעוד ל-Use-case

לכל use-case צריך ליצור תיעוד. התיעוד מורכב מ-4 רכיבים כאשר השניים הראשונים הם חובה והשניים האחרים רשות.

1. זיהוי - נציין כותרת, סיכום קצר מה מבצעת, מי השחקנים שמשתמשים, מתי נוצר ועודכן, ומי בחברת הפיתוח אחראי לפתח אותו.
2. תרחישי זרימה - נפרט מהם התנאים המוקדמים הנצרכים כדי להשתמש בפעולה שאנו מנתחים, מהם כל השלבים בשימוש, ומה ההשלכות/תוצאות של השימוש בפעולה. בנוסף, יש לתאר תרחישים בעייתיים.
3. נפרט מהם מנגנוני הקלט והפלט איתם השחקנים המשתמשים בפעולה מתממשקים.
4. אילוצים לא פונקציונאליים על ה-use-case, כמו זמן תגובה, זמינות, אמינות, בטיחות וכו'. נלקח מתוך הדרישות הלא-פונקציונאליות ממסמך הדרישות.

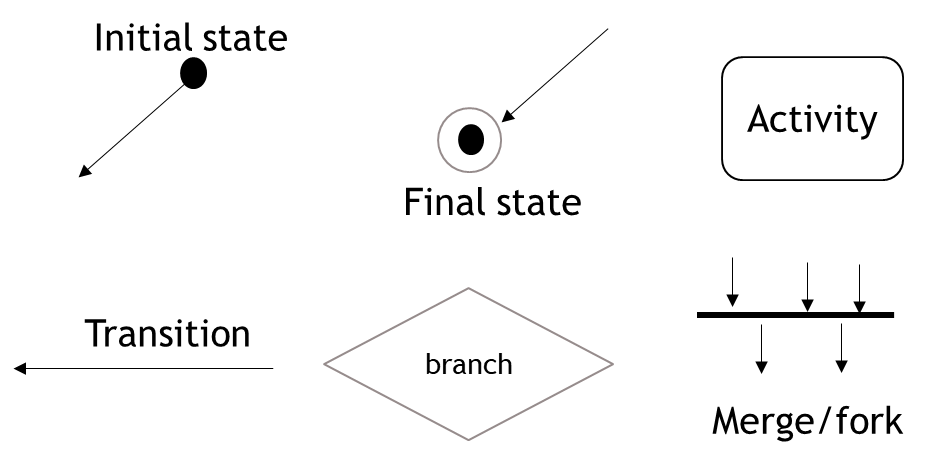


## Activity Diagram

בדיאגרמה זו הופכים כל פעולה מדיאגרמת use-case לתרשים זרימה המתאר את כל התתי-פעולות עבור כל מצב אפשרי.

### בניית התרשים

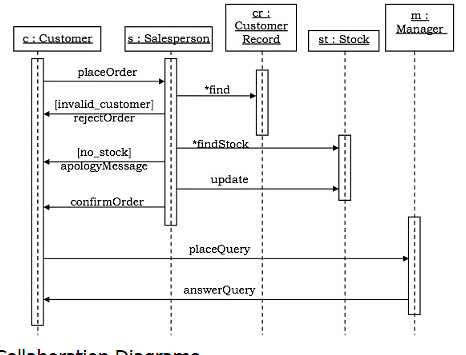
מתחילים ממצב התחלתי ובעזרת חיצים עוברים ממצב למצב. כאשר יש מספר אפשרויות נפצל את הזרימה ונסמן זאת באמצעות מעוין שמשני צדדיו יצאו שתי האפשרויות. כאשר שתי זרימות מתפצלות או מתמזגות נסמן זאת באמצעות קו.



## Sequence Diagram

בדיאגרמה זו הופכים כל פעולה מדיאגרמת use-case לתרשים רשת דו-מימדי המתאר את סדר כל התתי-פעולות ברצף של זמן.

### בניית התרשים

בציר ה-x נציין בקווים אנכיים את כל האובייקטים המשתתפים בעשיית הפעולה בסדר כרונולוגי, ובציר ה-Y נציין באמצעות חיצים את ההודעות שהאובייקטים שולחים זה לזה. מעל כל חץ נרשום את נושא ההודעה. יש שני סוגים של חיצים: חץ מלא מציין הודעה סינכרונית, כלומר מחכה לתשובה מהאובייקט ועד שלא מקבל לא עובר לשלב הבא. חץ ריק מציין הודעה לא סינכרונית, כלומר לא מחכה לתשובה וכבר עובר לשלב הבא. כאשר אובייקט מסיים את פעולתו נוכל לסמן זאת ב-x על הקו שלו.

### הבדלים בין Sequence Diagram ל-Activity Diagram

* Sequence Diagram מתמקד בהודעות שמועברות בין ישויות במערכת ואילו Activity Diagram מתמקד בתהליך של הפעולה ונותן דגש לכל המצבים האפשריים.
* Sequence Diagram מתאר את סדר הפעולות לפי זמן.
* Sequence Diagram מתאר תהליך מדויק ואילו Activity Diagram מתאר תהליך כללי.

## Class Diagram

מתאר את כל המחלקות במערכת ואת היחסים ביניהם. זהו המודל המרכזי והכי נפוץ ב-UML. מפתחים משתמשים במודל זה כדי לפתח את המערכת.

### ייצוג מחלקה

כל מחלקה במערכת תיוצג במלבן המחולק לשלושה חלקים: בחלק העליון שם המחלקה, באמצעי תכונות המחלקה ובתחתון שיטות המחלקה.

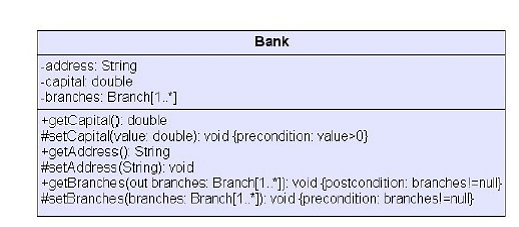
תכונות:

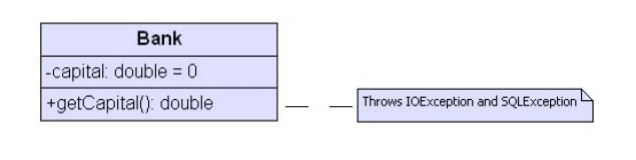
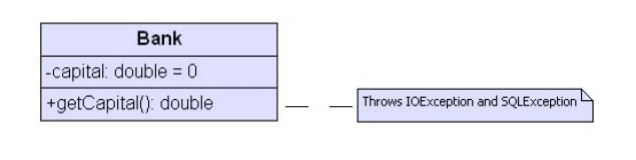
* לכל תכונה נציין קודם את סוג הרשאת הגישה שלה: private -, public +, protected #, package friendly ~.
* ניתן גם להוסיף סימן ‘/’ המציין שהערך של תכונה זו נקבע באמצעות חישוב על תכונות אחרות.
* לאחר סוג ההרשאה נרשום את שם התכונה, נקודותיים ולאחריהם את סוג הנתונים שמקבלת.

|  |
| --- |
| **Bank** |
| + + owner: Client[1..3]  - - number: int = 0  - - creditCard: CreditCard [\*] |
| *[functions]* |

* Multiplicity - לאחר סוג הנתונים אפשר לציין כמה ערכים מתכונה זו האובייקט יכול להכיל. בתוך סוגריים מרובעות נרשום את טווח הערכים. \* מציין ערך לא מוגבל. כאשר לא מציינים multiplicity ברירת המחדל היא 1.
* ניתן להוסיף ערך ברירת מחדל של התכונה באמצעות '=' שלאחריו ערך ברירת מחדל.

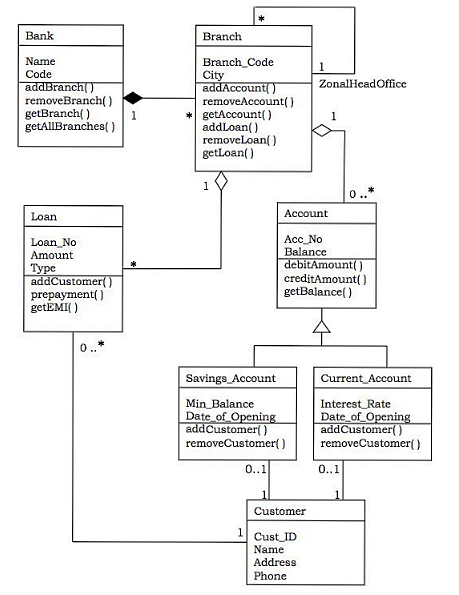
שיטות:

* לכל שיטה נציין את סוג הרשאת הגישה שלה כמו שפירטנו לעיל.
* שם השיטה.
* בסוגריים את כל הפרמטרים שהשיטה מקבלת. לכל פרמטר נציין את שמו, נקודותיים, סוג הנתונים ואפשר גם multiplicity וערך ברירת מחדל. בנוסף, בתחילת הפרמטר אפשר לציין את תפקידו: in מציין שהפרמטר משמש להעברת נתונים לשיטה, out משמש כדי שהפונקציה תכניס לתוכו ערכים, ו-return אותו גם הפונקציה מחזירה.
* לאחר הפרמטרים נרשום נקודותיים ואז את סוג ערך החזרה של הפונקציה.
* אילוצים - לאחר ערך החזרה נוכל בתוך סוגריים מסולסלות לרשום אילוצים על הפעלת השיטה. כל אילוץ הוא ביטוי לוגי על הפרמטרים של השיטה. יש שני סוגי אילוצים: precondition - אלו תנאים מקדימים שחייבים להתקיים כדי להפעיל את השיטה, ו-postcondition - אלו תנאים שצריכים להתקיים לאחר הפעלת השיטה.
* כאשר בקריאה להפעלת שיטה יש סכנה שייזרק exception, מקובל להוסיף note ובו תיאור ה-exception שעלול להיזרק ולחברו בקו מקווקו לשיטה שבה מדובר.
* שיטות סטטיות מסמנים עם קו תחתון ושיטות אבסטרקטיות רושמים בצורה נטויה (*italic*). אם גם המחלקה אבסטרקטית גם שם המחלקה יהיה נטוי.



### קשרים בין מחלקות

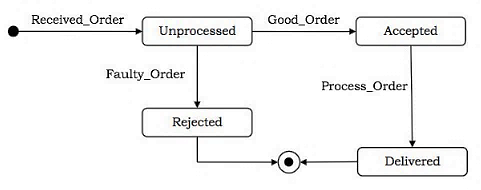
כדי לסמן קשרים בין מחלקות נמתח קווים מיוחדים בין מחלקות. על קווים אלו צריך להוסיף multiplicity. יש מספר סוגי קשרים:

* Dependency - כאשר מחלקה A משתמשת בשיטות של מחלקה B נמתח חץ מקווקו מ-A ל-B.
* Association - כאשר מחלקה A מחזיקה רפרנס של מחלקה B נמתח חץ רגיל מ-A ל-B. אם גם B מחזיקה רפרנס של A יהיה קו רגיל ללא חיצים.
* Aggregation - כאשר מחלקה A מכילה אובייקט של מחלקה B, אך B עומדת בפני עצמה ויש לה קיום גם אם A נמחקת. הקשר במקרה כזה יבוטא בקו שיוצא מ-B אל A ומסתיים במעוין ריק.
* Composition - כמו Aggregation אלא ש-B אינה עומדת בפני עצמה, כלומר ללא A האובייקט של B אינו יכול להתקיים. קשר כזה מבוטא בקו שיוצא מ-B ומסתיים במעוין מלא.
* ירושה - כאשר מחלקה A יורשת ממחלקה B נמתח קו מ-A ל-B שבסופו משולש ריק.
* ממשק - כאשר מחלקה A מממשת את הממשק B נמתח קו מקווקו מ-A ל-B שבסופו משולש ריק.

## Object Diagram

זוהי דיאגרמה המתארת מבנה של אובייקטים הדומה לתרחישים אמיתיים שעל בסיסם בונים את המערכת. המטרה במודל זה היא לראות דוגמה אמיתית כיצד ייראו האובייקטים במערכת.

נגדיר כל אובייקט באמצעות מלבן המחולק לשניים: בחלק העליון שם האובייקט והמחלקה שהוא מופע שלה, ובחלק התחתון כל הערכים שקיבל לכל תכונות המחלקה (יש להמציא ערכים שיהיו קרובים למצב אמיתי). בין האובייקטים נמתח קווים כמו שהגדרנו ב-Class Diagram. אין צורך לציין multiplicity על הקשרים שכן תמיד מדובר על ישות בודדת.



## State Machine Diagram

זהו תרשים זרימה של מצבים שונים שבהם אובייקט מרכזי במערכת יכול להיות. מייצגים מצב במלבן וכל חץ בין מצבים מייצג את הפעולה (trigger) שהביאה לשינוי במצב.

## ERD

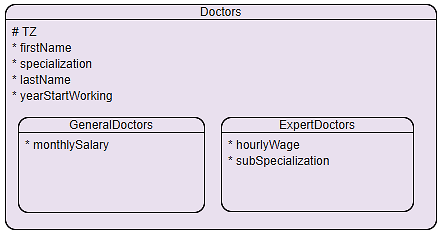
עד כה עסקנו במודלים של UML המתארות את המערכת. בסעיף זה נעסוק במודל של מסד הנתונים בו המערכת משתמשת. המודל שבו נשתמש כדי לבנות ולתאר את מסד הנתונים הוא ERD (Entity Relation Diagram). זהו מודל גרפי המייצג את מסד הנתונים כאוסף של ישויות עם תכונות וקשרים בין הישויות. המודל הוא פשוט וקל להבנה אך בו בזמן מדויק. מזכיר את תרשים Class Diagram של UML.

### ישות

כל ישות במודל מייצגת אובייקט ממשי או מופשט שיש לו קיום במציאות והוא בעל משמעות למערכת. דוגמאות לישויות מסוגים שונים: מכונה, פריט, סטודנט, קורס, דיווח נוכחות, תנועה בחשבון בנק, כניסת פריט למלאי. לכל ישות נוכל ליצור מופעים שונים.

לכל ישות יש תכונות. תכונה היא מאפיין כלשהו של הישות שיש לה משמעות ועניין במערכת. יש מספר סוגים של תכונות:

* תכונה פשוטה - זוהי תכונה המכילה מרכיב אחד בלבד ואינה ניתנת לחלוקה נוספת.
* תכונה מורכבת - תכונה המכילה מספר רכיבים וניתן לחלק אותה. אלו תכונות המכילות אובייקט.
* תכונה מחושבת - תכונה שערכה הוא תוצאה של חישוב נתונים אחרים במערכת.
* תכונה עם ערך בודד - מכילה רק ערך אחד בכל נקודת זמן.
* תכונה מרובת ערכים - יכולה להכיל מספר ערכים בכל נקודת זמן. הערכים נשמרים במערך או מבנה נתונים אחר.
* תכונת חובה - חייבות לקבל ערך. אין להן ערך ברירת מחדל.
* תכונת רשות - לא חייב לתת להן ערך ויכולות להכיל ערכי ברירת מחדל כמו 0 או null.
* מפתח ראשי (primary key) - אלו הן תכונה אחת או יותר המזהות באופן חד-חד ערכי מופע מסוים של ישות. מפתח פשוט הוא מפתח המכיל תכונה אחת בלבד (כמו ת.ז), ומפתח מורכב זהו מפתח המכיל מספר תכונות.
* מפתח זר - זוהי תכונה המופיעה גם בישות אחרת ומהווה המפתח הראשי שלה.



נייצג ישות באמצעות מלבן עם קצוות עגולים המחולק לשני חלקים. בחלק הראשון שם הישות ובחלק השני התכונות שלו. עבור כל תכונה יש לציין קודם את סוג התכונה: \* תכונת חובה, 0 תכונת רשות, pk (או #) מפתח ראשי, fk מפתח זר, ולאחר מכן את שם התכונה.

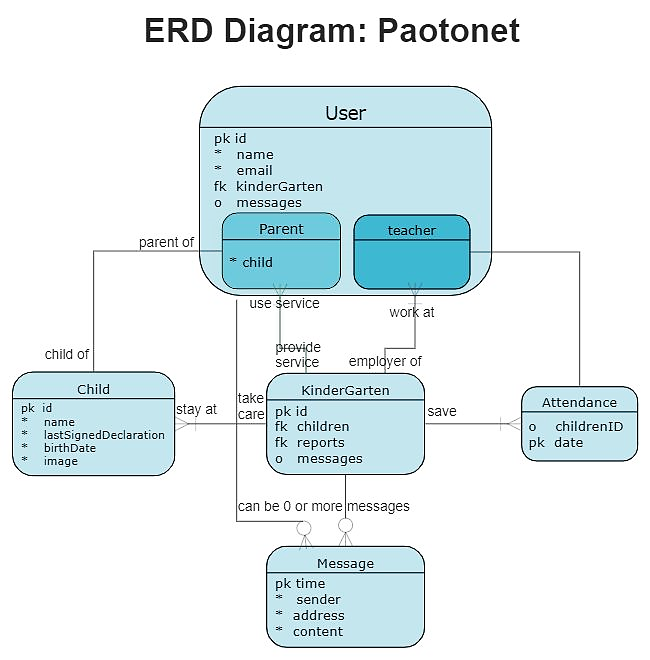
כאשר יש שני ישויות A ו-B כך ש-A מכיל את B, אזי נאמר ש-A היא "ישות על" ו-B היא "תת ישות". נייצג אותן כך שהמלבן של ישות B נמצאת בתוך המלבן של ישות A.

### קשרים בין ישויות

קשר מוגדר כיחס בעל משמעות בין שני ישויות. נייצג את הקשר באמצעות קו שנמתח בין הישויות. על קו זה נציין במילים בודדות את סוג הקשר וכן את קרדינליות הקשר. קרדינליות הקשר מוגדרת כמספר המופעים המינימלי והמקסימלי שניתן לקשר בין שני הישויות. יש מספר קשרים מיוחדים:

* קשר חד-חד ערכי 1:1 – מופע אחד של ישות A מתאים למופע של ישות B בלבד. אין סימון מיוחד לקשר זה.
* קשר חד-רב ערכי 1:M - מופע אחד בישות A מתאים לכמה מופעים בישות B. הקו שנמתח ביניהם מהצורה 

אם לא חייב להיות קשר, כלומר M יכול להיות 0, אזי נסמן זאת על ידי קו מקווקו ולא רציף.

* קשר רב-רב ערכי M:N - מספר מופעים מישות A מתאימים למספר מופעים מישות B. הקו שנמתח ביניהם 
* תלות קיומית - כאשר קיום ישות A מותנה בקיום ישות B. כלומר אם B לא קיים אזי ל-A אין שום משמעות. לדוגמה ישות מרשמים ללקוח יש תלות קיומית בלקוח. נסמן זאת בתרשים על ידי קו אנכי החוצה את הקו המייצג את הקשר בין הישויות.
* קשר רקורסיבי - כאשר ישות מסוימת קשורה לעצמה נמתח קו מהישות לעצמה.
* קשר של ירושה - קשר של ישות על הינו קשר גם של תת הישות, אמנם קשר של תת ישות הינו קשר אל עצמה בלבד.
* יחס בחירה - כאשר A יכול לבחור רק קשר אחד מתוך מספר ישויות. לדוגמה, לקוח בקופ"ח יכול להיות קשור אל רופא משפחה או רופא ילדים, אך לא שניהם. נסמן קשר של יחס בחירה על ידי חצי עיגול על כל הקשרים.

## נרמול מסד הנתונים

נורמליזציה של מסד נתונים היא שיטה המתארת תהליך לבניית מסד נתונים בהתאם לסדרה של צורות נורמליות, המתארות רמות של נרמול. לכל צורת נרמול ישנם כללים כיצד ליישם אותה במסד נתונים והיא כוללת גם את הרמות הקודמות. כללים אלו ניתן ליישם או כאשר בונים מסד נתונים חדש או כאשר משפרים מסד נתונים קיים. המטרות של נרמול הן:

1. להפחית נתונים מיותרים.
2. לארגן את מסד הנתונים כך שיחזיק מידע על נושא אחד בלבד.
3. לפשט את פעולות השליפה והעדכון במסד נתונים.

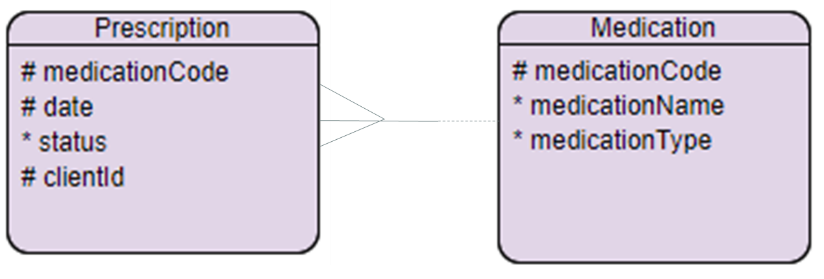
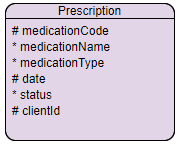
ככל שהמסד נתונים יותר מנורמל כך הוא מפורט יותר אך יש לכך גם חסרונות בהתאם לסוג המסד נתונים בו בחרנו להשתמש במערכת שלנו. בתחילת עיצוב המסד נתונים תמיד נרצה להגיע לדרגת הנרמול הטובה ביותר. בשלבים הבאים, כאשר נבחר באיזה סוג מסד נתונים להשתמש, נרד בדרגות הנרמול בהתאם לאילוצים.

### 1-NF

ברמת נרמול זו אין תכונות מורכבות אלא רק תכונות פשוטות, כלומר אין תכונה שמקבלת קבוצה של ערכים. כדי להגיע לרמה זו ניקח כל תכונה מורכבת וניצור ממנה ישות נפרד שהמופעים שלה הם כל הערכים השונים.

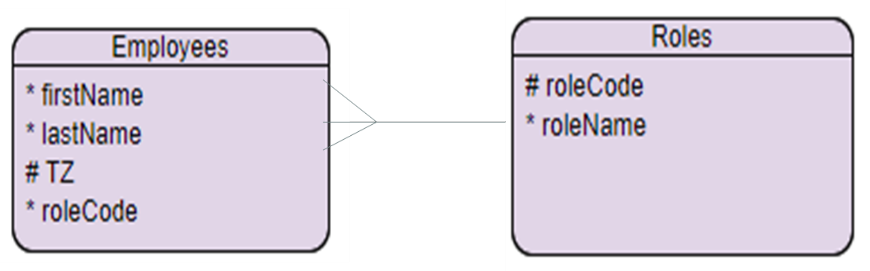
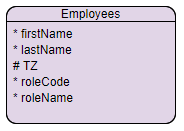
### 2-NF

מוסיפה שכל תכונה שאינה מפתח לא יכולה להיות תלויה בתת-קבוצה של מפתח. כדי להגיע לרמת נרמול זו נפצל כל ישות שהתנאי לא מתקיים לשני ישויות לפי שתי הקבוצות במפתח. בדוגמה בתמונה ניתן לראות ששם וסוג התרופה תלוי בתת קבוצה של המפתח שהוא קוד התרופה, לכן נפצל אותם משאר התכונות לישות חדשה.



### 3-NF

מוסיפה שכל תכונה שאינה מפתח חייבת להיות תלויה בקבוצת תכונות שמכילה את המפתח (סופר מפתח). כדי להגיע לרמה זו נפצל כל ישות שהתנאי לא מתקיים לשני ישויות. בישות אחת יהיו כל התכונות שתלויות בסופר-מפתח ובישות השנייה כל התכונות שאינן תלויות בסופר-מפתח. בדוגמה בתמונה ניתן לראות שמזהה התפקיד ושם התפקיד של העובד לא תלוי בקבוצת תכונות המכילה את הת"ז, לכן נפצל אותם משאר התכונות לישות חדשה.



### BCNF

מוסיפה שגם תכונה הנמצאת בקבוצת המפתח צריכה להיות תלויה בסופר מפתח. גם כאן במידה והישות לא מקיימת תנאי זה נפצל אותה לשניים כאשר התכונה במפתח שאינה תלויה במפתח תהיה בישות נפרדת עם כל התכונות התלויות בה.

### 4-NF

ברמה זה מקפידים שכל ישות תחזיק "רעיון" אחד בלבד, כלומר שכל התכונות רלוונטיות לישות אחת ולא למספר ישויות אפשריות. במידה ותנאי זה לא מתקיים, נפצל את הישות למספר ישויות שכל אחת מכילה רעיון אחד בלבד.

## שלבים ביצירת מודל ERD

בהינתן תיאור המערכת הרצויה/רשימת הדרישות:

1. זיהוי הישויות.
2. הגדרת הקשרים וסוגי הקשרים בין הישויות.
3. הגדרת קרדינליות הקשר.
4. זיהוי התכונות וסוג כל תכונה.
5. יצירת ERD ראשוני.
6. בדיקת רמת הנרמול של המודל ותיקונו עד קבלת מודל ברמת הנרמול המבוקשת.

# שלב העיצוב

## הגדרה

בשלב זה מתכננים את בניית המערכת הן מבחינת חומרה (שרת, מסד נתונים, תקשורת וכו') והן מבחינת תוכנה (אלגוריתמים, מבני נתונים, עיצוב הממשק). המטרה המרכזית בשלב זה היא עיצוב המערכת בצורה כזו שתיתן למשתמש חוויה טובה ושימושית כך שתענה על צרכיו וירצה להשתמש בה.

שלב זה בדרך כלל יבוצע במקביל עם שלב הניתוח בו מחליטים את מבנה המחלקות, מבנה המסד נתונים וכיצד עובד כל תהליך. התוצר של שלב העיצוב נקרא "מסמך אפיון".

ישנם מספר סיבות מדוע כדאי להשקיע בשלב העיצוב. הסיבה המובנת מאליה היא שעיצוב טוב יגרום לחוויית משתמש טובה יותר, יעלה את מידת שביעות הרצון של הלקוחות מהמערכת ויגרום ללקוחות נוספים לרצות להשתמש בה. אך בנוסף יש יתרון גדול גם למתכנתים. אם בשלב הטמעת המערכת בסביבת הלקוח חוויית המשתמש לא תהיה טובה זה ידרוש מצוות הפיתוח לעשות התאמות בשלב מאוד מתקדם של הפיתוח, וכבר [אמרנו לעיל](#_הגדרה) ששינויים בשלב זה הם יקרים ולוקחים זמן רב. לעומת זאת, אם חוויית המשתמש תהיה טובה נחסוך זמן וכסף על שינויים אלו.

## חווית המשתמש

חווית המשתמש (User Experience, בקיצור UX) עוסק באלמנט החווייתי של השימוש במוצר ומנסה למדוד את תחושותיו של המשתמש כתוצאה מהשימוש במוצר ואת שביעות רצונו ממנו. האלמנט החווייתי נולד מתוך ההבנה שהצרכנים זקוקים הן למענה פרקטי והן למענה רגשי. הצרכנים רוצים להרגיש שהמערכת מבינה אותם ואת הצרכים שלהם, והשימוש בה הוא נוח, ידידותי, פשוט ומהנה.

ישנם חמישה מרכיבים לחוויית המשתמש:

* **שימושיות:** הפונקציות הקיימות באתר וקלות התפעול שלהן (ביצוע פעולות, הזנת תוכן, קבלת משוב מתאים מן המערכת וכו'). זהו המרכיב החשוב ביותר ולכן נפרט עליו עוד בהמשך.
* **אסתטיות:** המראה הכללי של ממשק המשתמש, מידת היותו נעים לעין, שימוש באלמנטים גרפיים כגון, צבעים, צורות, פונטים וכו׳ שיהיו אסתטיים עבור המשתמש.
* **תנועה:** שימוש באנימציה מעניק למשתמש זרימה טבעית ואינדיקציות קוגניטיביות למה קרה במסך. משתמשים באנימציה גם במסכים עצמם וגם במעבר בין המסכים.
* **אמינות**: העברת מידע מהימן למשתמש והימנעות מטעויות או הטעיות.
* **נגישות**: יצירת ארכיטקטורת מידע המתאימה לדרך בה המשתמש חושב או מחפש מידע באתר.

### שימושיות

ישנם מספר מדדים בהן באמצעותם אנו מעריכים את יעילות המערכת:

* **לימודיות**: כמה קל למשתמשים לבצע מטלות פשוטות ברגע שנתקלו בהם בפעם הראשונה בממשק?
* **יעילות:** ברגע שמשתמשים למדו את הממשק כמה מהר הם מסוגלים לבצע מטלות?
* **זכירות:** כאשר משתמשים חוזרים למערכת אחרי פרק זמן ללא שימוש בה, כמה קל להם לרכוש מיומנות חוזרת?
* **שגיאות:** כמה שגיאות משתמשים מבצעים, כמה חמורות השגיאות וכמה קל לתקן את השגיאות?
* **סיפוק:** מהי מידת האהדה שהמשתמש רוחש למערכת?

## עקרונות עיצוב ממשק המשתמש (UI)

נפרט על מספר עקרונות מאוד חשובים שצריך לשים אליהם לב כאשר מעצבים את ממשק המערכת:

1. נראות - צריך להיות ברור, יפה ועקבי. עיצוב אסתטי ומינימליסטי.
2. התאמת המערכת לעולם האמיתי ולצורה שבה המשתמש חושב.
3. שליטה וחופש של המשתמש - המערכת צריכה לאפשר למשתמש לשלוט בה ולתקן עצמו גם אם טעה.
4. למנוע מהמשתמש לעשות טעויות - לדוגמה, אם יש אפשרות לא רלוונטית אזי שהמשתמש לא יוכל לבחור אותה.
5. זיהוי והתאוששות מטעויות - משתמש יבין באמצעות הודעה מתאימה כאשר מתרחשת שגיאה. צריך לאפשר לו לתקן.
6. זיהוי, לא זיכרון - כל משתמש יזהה אינטואיטיבית מה צריך לבצע ולא שיזכור מפעמים קודמות.
7. גמישות ויעילות בשימוש - צריך לאפשר מספר דרכים לבצע כל פעולה. הוספת קיצורי דרך למשתמשים מנוסים.
8. עזרה ותיעוד.
9. הכינו תרשים זרימה למערכת.
10. המלצות כלליות - לחצנים בגודל מתאים לאצבעות (7-10 מ"מ). תפריט ראשי ולחצנים מרכזיים במרכז המסך. לצמצם צורך בהקלדה. להקפיד על ניגודיות צבעים בין הרקע לטקסט.

## Material Design

Material Design (או Quantum Paper) היא שפת עיצוב שמפותחת על ידי גוגל. הוכרזה לראשונה ב-2014. כל האפליקציות של גוגל לאנדרואיד ורוב גרסאות הווב שלהן מיישמות את השפה. רוב אפליקציות האנדרואיד מיישמות את השפה, אך המפתחים אינם מחויבים להתאים את האפליקציות לשפה. השפה מתפתחת עם הזמן, והחל מחודש מאי 2018 כל חודש יוצאת גרסה חודשית של המערכת.

Material Design משלבת עקרונות קלאסיים של עיצוב טוב עם החדשנות של הטכנולוגיה והמדע. המטרות שלה הן:

1. יצירת שפה חזותית המשלבת עקרונות קלאסיים של עיצוב טוב עם החדשנות של הטכנולוגיה והמדע.
2. אחידות - לפתח תשתית המייצרת חוויית משתמש אחידה ללא תלות במערכת מסוימת, מכשיר מסוים או בשיטת הזנה מסוימת (הקלדה/תנועות עכבר/קול).
3. גמישות ויכולת התאמה - יצירת מנגנון הרחבה של השפה ליצירת תשתית גמישה עבור הכנסת שינויים של חדשנות אישית או סימני מיתוג.
4. תנועה מספקת משמעות - תנועה ממקדת את תשומת הלב ויוצרת חוויה של המשכיות דרך משוב עדין ומעברים עקביים. מהרגע שחלקי ממשק מופיעים על המסך, הם משתנים ומסדרים מחדש את כל הסביבה, בעדינות, כאשר אינטראקציות מייצרות שינויים חדשים.
5. גמישות - בנויה כך שהיא מאפשרת לבטא מיתוג וייחודיות של כל לקוח. ניתן למזג אותה עם קוד קיים של הלקוח, דבר שמאפשר יצירה חלקה של קומפוננטים, plugins ויחידות עיצוב קטנות.

כדאי להשתמש ב-Material Design מפני שהיא מעוצבת בצורה טובה ואחידה, מתעדכנת עם הזמן כך שתמיד נשארים בקדמת הטכנולוגיה, חוסכת בזמן עיצוב ונותנת למשתמש מראה מוכר ותחושת ביטחון.

אפשר לקרוא עוד בפירוט על Material Design [באתר שלהם](https://material.io/design/introduction), או במצגות הנדסת תוכנה משנה שעברה.

# שלב המימוש והבדיקות

## הגדרה

בשלב זה ממשים את המערכת ובודקים שאכן עובדת כנדרש ממספר בחינות שונות. פיתוח התוכנה כולל: התקנה של סביבת פיתוח, כתיבת הקוד, תיעוד ובדיקות.

## כתיבת קוד נכון

מאוד חשוב לכתוב קוד בצורה נכונה לפי כל העקרונות שנפרט בהמשך. יש לכך מספר סיבות:

* קוד יהיה יעיל ואיכותי יותר.
* יותר קל לפתור בעיות.
* קוד גמיש - שינויים או הוספות יתבצעו בצורה קלה ונוחה. יותר קל לתחזוקה.
* הקוד יהיה קריא וברור למתכנתים אחרים ולא תלוי רק במי שכתב אותו.

מתכנתים פעמים רבות נוטים לסבך את הקוד ולא לשמור על כל הכללים והעקרונות בכתיבת קוד נכון. שיטה זו יוצרת קוד מבולגן ולא יעיל. יש לבצע תכנון איכותי לפני שכותבים קוד ולא לכתוב את הקוד תוך כדי מחשבה. הרעיון המרכזי בכתיבת קוד הוא לשמור על פשטות כמה שיותר (עקרון ה-K.I.S.S). קוד צריך לספר סיפור, לכן קוד שקשה להסביר אותו הוא כנראה לא קוד טוב. לפני שכותבים קוד מומלץ לחשוב ולספר לעצמך אותו בראש בצורה הכי קלה ומובנת, ולאחר מכן לכתוב אותו בצורה זו.

### שמות משתנים ומחלקות

שמות משתנים צריכים להיות קצרים ובעלי משמעות, כך שפעולות השמה על המשתנים יהיו חלק "מהסיפור" של הקוד ולא פעולה הדורשת פיענוח. יש שתי שיטות נפוצות בנתינת שמות עם כמה מילים למשתנים, אחת עם אות גדולה בתחילת מילה numPending (נקראת camelCase), והשנייה עם מקף תחתון num\_pending. לא משנה באיזו שיטה נבחר העיקר שנהיה עקביים בשיטה אחת למשך הקוד.

### תיעוד

מטרת ההערות להעביר מידע לקורא שאינו ברור מאליו מקריאת הקוד, אין צורך לרשום בהערות מידע המובן מאליו. עקרונית הקוד אמור להסביר את עצמו, הערות נכתוב בשביל להסביר את ה-'למה'. לא להשאיר קוד ישן בגרסה הסופית של התוכנית.

חשוב מאוד להוסיף הערות Javadoc לקוד. מעבר לכך שמשמשים תיעוד יעיל למי שקורא את הקוד, אפשר ליצור מהם תיעוד בפורמט HTML מעוצב למתכנתים אחרים שישתמשו בקוד שלנו.

## עיצוב מחלקות

1. **סכמה קבועה ועקבית של הגדרת מחלקה** - הקפדה על תבנית ברורה בהגדרתה של מחלקה תורמת לבהירות הקוד. כך למשל, אם מחליטים כי המשתנים של המחלקה יופיעו לפני המתודות שלה, ומקפידים על כך בכל המחלקות שמגדירים אזי בהירות הקוד גדלה. באופן דומה, הקפדה על סגנון נאה, והערות עקביות ומסודרות באמצעות ה-Javadoc, כל זה גם תורם לבהירות הקוד.
2. **להמעיט במספרם של משתנים מטיפוס בסיסי בהגדרת המחלקה** - מחלקה שיש בה מספר רב מאוד של משתנים מטיפוס בסיסי מעלה את השאלה שמא ניתן היה להגדיר במקום מחלקה אחת מספר מחלקות, ובכך להקנות לתוכנית רמה גבוהה יותר של מודולאריות.
3. **לאתחל משתנים של המחלקה** - גם אם מאותחלים אוטומטית. אתחול מכוון מקנה בהירות רבה יותר לקוד.
4. **נתינת שמות** - לתת למחלקות ולמתודות שמוגדרות בתוכן שמות שמעידים על פעולתן.
5. **עקרון הכימוס** - לקבוע את הרשאת גישה כמה שיותר מצומצמת. מבטיחים את שלמות ונכונות נתוני האובייקט.

## S.O.L.I.D

עקרונות SOLID הם חמישה עקרונות בסיסיים בעיצוב מונחה עצמים. המחשבה מאחורי העקרונות היא שכאשר הם מיושמים יחדיו בפיתוח של מערכת תוכנה, היא תהיה יותר קלה לתחזוקה והרחבה לאורך הזמן.

באופן עקרוני, קוד גרוע הוא קוד שבו:

* כל שינוי משפיע על הרבה חלקים בקוד.
* שינוי בקוד משפיע על אזורים לא קשורים בקוד.
* קוד לא פריק. לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שלשמם נכתב הקוד במקור.

האופי המרכזי של הבעיות האלו הוא יותר מידי תלות בתוך הקוד. עקרונות SOLID באים לתת קווים מנחים שיגרמו לנו להימנע מלכתוב קוד עם הבעיות הנ"ל.

**Single responsibility**

לכל מחלקה צריך להיות תחום אחריות אחד עליו היא אחראית בצורה מלאה. טעות מאוד נפוצה היא שכאשר בונים מחלקה שמייצגת אובייקט כלשהו, אזי מוסיפים לה שיטות שלא בהכרח קשורות לאובייקט אך במבט ראשוני נראה שנוח אם האובייקט יוכל לבצע אותן. פעמים רבות שיטות אלו מאוד מגבילות ודורשות שינויים מהותיים כאשר מרחיבים או משנים משהו במחלקה. לדוגמא, עבור אובייקט שמייצג משתמש, לא נרצה שתהיה שיטה שמקבלת אימייל ובודקת אם הוא תקין. במקום זה נבנה מחלקה מיועדת שבודקת מידע.

באופן עקרוני, כדאי לפצל מחלקות למחלקות קטנות יותר כאשר זה אפשרי. מספר רב יותר של מחלקות מקנה לתוכנית רמה גבוהה יותר של מודולאריות. חשוב להשתדל לעבוד בכיוון זה.

**Open for Extension, Closed for Modification**

מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים. כלומר, שהוספת שיטות למחלקה לא ידרוש מאיתנו לשנות קוד שכבר כתבנו. גם הוספה של מחלקות אחרות לתוכנית לא אמור לשנות מחלקות שכבר כתבנו. נעשה זאת בעזרת תכנון נכון של המחלקה ושימוש נכון בממשקים.

**Liskov substitution principle**

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל. עצה לכך היא שכאשר כותבים מחלקה שיורשת ממחלקה אחרת או מממשת ממשק, אזי תמיד נסתכל על המחלקה שאנו כותבים כאל אובייקט מסוג מחלקת האב או הממשק עם תוספות קלות. באופן כללי, תמיד כדאי להשתמש בממשקים ולא בירושה.

**Interface segregation principle**

אין להכריח לקוח להיות תלוי בממשק שהוא אינו משתמש בו באופן מלא. יש לדאוג לממשקים מצומצמים, כך שיאלצו מחלקה שמממשת את הממשק לממש רק מה שרלוונטי לה. פתרון לבעיה זו הוא ממשקים שמממשים ממשקים.

**Dependency inversion**

מחלקות high level לא צריכות להשתמש באופן ישיר במחלקות low level, כלומר מחלקות המתארות מקרה כללי לא צריכות להשתמש במחלקות המתארות מקרה פרטי. הסיבה לכך היא שאם בשלב כלשהו נרצה להוסיף עוד מחלקה, המייצגת מקרה פרטי לתוכנית שלנו, נצטרך לשנות גם את המחלקה הכללית.

הפתרון לבעיה זו היא ליצור ממשק, המייצג גם כן מקרה כללי, שאותו יממשו המחלקות המתארות מקרה פרטי. כעת במחלקת ה-high level ממנה התחלנו, נוכל להשתמש בממשק זה במקום. כל הוספה של מחלקה המייצגת מקרה פרטי תממש את הממשק.

## Code Review

זוהי בדיקה שיטתית על קטע קוד בכדי למצוא ולתקן בעיות שנמצאות בקוד עוד בשלבי הפיתוח המוקדמים. הבדיקה נעשית על ידי עמיתים ולא על ידי המתכנת עצמו. מטרת הבדיקה היא למצוא בעיות ולשפר תהליכים מזווית של אדם אחר שלא כתב את הקוד. בדיקה כזו עוזרת למתכנת להעלות בעיות שלא ראה או להציע פתרונות שלא חשב עליהם. היא גם יעילה למצוא בעיות שקשה למצוא בבדיקות ממוחשבות, כמו: קוד מת, פעולות ומשתנים מיותרים, דליפת מידע. מטרה נוספת היא שמשפרת את רמת התחזוקה של הקוד, שכן לאחר הבדיקה יש שני אנשים שיודעים מה הולך בקוד ולא הכל תלוי באדם אחד.

## שלב הבדיקות - הגדרה

בדיקות תוכנה הן מכלול תהליכים, שיטות, פעילויות וכלים המלווים את פיתוח המערכות והמוצרים לאורך מחזור החיים, מתוך כוונה להעריך את איכות התוכנה והוכחת תקפות להתאמת התוצרים לדרישות הלקוח, למפרטים הטכניים, לתקנים ולנהלים מחייבים. בדיקות תוכנה מהוות חלק אינטגרלי מתהליכי הנדסת תוכנה, ומוסיפות נדבך בעל חשיבות מכרעת להבטיח את איכות המערכת.

הבדיקות מבוצעות בראשונה על ידי צוות הפיתוח עצמו בסביבת הפיתוח. לאחר מכן צוות הבדיקות מוסיף את הבדיקות שלו ומודיע לצוות הפיתוח על התוצאות. לבסוף, בשלבי הטמעת המערכת גם הלקוחות בודקים את המערכת. נעמוד על ההבדלים בין שלושת סוגי הבדיקות בהמשך. בכדי לייעל את תהליך הבדיקות, ניתן להשתמש בכלים שונים לניהול ולביצוע הבדיקות.

מטרות שלב הבדיקות הם:

* Validation - האם המערכת אכן עובדת כמו שרצינו?
* Verification - האם המערכת אכן תואמת לדרישות ולמפרטים שהציבו לנו? כולל בדיקות פונקציונאליות ובדיקות לא פונקציונאליות.
* לבדוק האם המערכת איכותית - מינימום שגיאות ועמידה בדרישות ביצועים ועומסים.
* מטרה נוספת של תהליך הבדיקות היא לצמצם ככל הניתן את עלויות אי האיכות. לפי העיקרון של בוהם, ככל שנאתר את השגיאות בשלבים מאוחרים יותר, תגדל עלות התיקון.

## Testability

ההגדרה של Testability היא עד כמה ניתן לבדוק מערכת/רכיב. הגדרה נוספת היא המידה בה מערכת או רכיב מאפשרים את ההגדרה של קריטריוני בדיקה ואת הביצוע של בדיקות אשר יקבעו האם קריטריונים אלה הושגו. ישנם כמה כללים המוודאים שמערכת היא אכן ניתנת לבדיקה איכותית:

* יכולת בקרה - קיים ממשק בו ניתן להפעיל את הבדיקות וניתן בו לשלוט על המשתנים באופן ישיר ובלתי תלוי.
* שקיפות - ניתן לראות מה מתרחש במערכת. כל הגורמים הרלוונטיים גלויים. פלט לא תקין מזוהה בקלות.
* זמינות - המערכת זמינה להרצה ולבדיקה. ניתן להריץ אותה ולבדוק כל חלק בנפרד וכן ניתן לגשת לקוד המקורי.
* פשטות - הקוד עקבי וקריא. המחלקות הן קטנות ולכל מחלקה יש אחריות אחת בלבד.
* יציבות - יש מעט שינויים בתוכנה. כל השינויים מתועדים ולא מפרים בדיקות קודמות.
* מידע - כל מבנה המערכת ידוע ומתועד.

## מה בודקים

### בדיקות מסירה

אלו הן מכלול הבדיקות הנעשות בתהליך הפיתוח טרם מסירת המוצר לבדיקות הקבלה של הלקוח. נמצאות באחריות צוות הפיתוח והבדיקות. אפשר לחלק לארבעה שלבים:

* בדיקות יחידה: בודקים כל פונקציה בקוד בנפרד.
* בדיקות אינטגרציה (שילוב): כיצד מחלקות או פונקציות עובדות ביחד.
* בדיקות ממשקים: בדיקות בין ממשקים במערכת או מחוצה לה.
* בדיקות מערכת: סוגים שונים של בדיקות למערכת כולה: פונקציונאליות, יעילות, עומסים, עיצוב וכו'.

### בדיקות קבלה

אלו הן בדיקות כוללות של המערכת לפני הפעלה, במטרה לוודא עמידה בדרישות והתאמת המערכת לצרכי הלקוח. נמצאות באחריות הלקוח/משתמש ומבוצעות בסביבת הלקוח או בסביבת pre-production.

## שיטות לבדיקה

כל הבדיקות מבוססות על הכנסת קלט, קבלת פלט ובדיקה על הפלט.

### בדיקות קופסה לבנה

סוג בדיקות מעמיק המתמקדות בזרימת המידע הפנימית בתוך המערכת בין רכיבי התוכנה/חומרה ובדיקת זרימת הנתונים בתוך התהליכים. בבדיקות אלו נדרש ידע אודות המימוש הפנימי של המערכת ולכן בדרך כלל מבוצעות על ידי המפתחים בלבד. לא צריך הסתכלות כוללת על המערכת אלא ספציפית לקטע קוד שאנו בודקים. דוגמה לבדיקות מסוג זה הן בדיקות יחידה ובדיקות אינטגרציה.

### בדיקות קופסה שחורה

שיטת הבדיקות הרלוונטית לרמות הבדיקה שלאחר בדיקות השילוב (בדיקות מערכת ומעלה). מוודאים שהמערכת מממשת את הדרישות שהציבו לה. אין עניין במבנה הפנימי של הקוד עצמו. בבדיקות קופסה שחורה יש חשיבות אך ורק ל-"מה" ולא ל-"איך", כלומר חשובה רק התוצאה ולא הדרך שבה המערכת הגיעה אליה. מבוצעים על ידי צוות הבדיקות והלקוחות.

### בדיקות קופסה אפורה

בדיקות ביניים בין קופסה לבנה לשחורה. הן כן ניגשות לקוד אך לא בצרה מעמיקה. מטרתן היא בעיקר לבדוק תהליכים גדולים ומקרי קצה. מבוצעים על ידי צוות הבדיקות. דוגמה לבדיקות מסוג זה הן בדיקות ממשקים.

## עקרונות הבדיקות

* מטרת הבדיקה היא להכשיל את המערכת.
* לא לבדוק רק מה שבמפרט אלא גם מקרי קצה.
* בדיקות רגרסיה - לוודא שהבדיקה לא פוגעת בדברים קודמים שעשינו.
* תוצאות הבדיקות צריכות להיות מושוות עם כלים מתאימים. לדוגמה, אם בדיקה צריכה להחזיר ערך מספרי נבדוק זאת עם כלי חישוב ולא ידנית.
* יש לשלב בין בדיקות ידניות ואוטומטיות.
* צריך לוודא בצורה אמפירית שאסטרטגיית הבדיקה היא נכונה לקוד שאנו בודקים.
* להגדיר קריטריונים כיצד להעריך את תוצאות הבדיקות. הקריטריון המרכזי הוא כמה שגיאות מתגלות כפונקציה של זמן.

## רמת חומרה של באגים

מאוד חשוב להגדיר רמות חומרה של שגיאות בבדיקות. דירוג השגיאות עוזר לכולם להבין מתי יש שגיאה קלה ומתי יש שגיאה חמורה, וכן עוזר למפתחים לתעדף שגיאות חמורות על פני קלות.

|  |  |  |
| --- | --- | --- |
| **תיאור** | **שם** | **חומרה** |
| - מונעת ביצוע של רכיבים מרכזיים במערכת.  - מסכנת את הבטיחות או הביטחון.  - תקלה מהותית בנושא שהוגדר בעל חשיבות גבוהה. | Highly Critical | 5 |
| - משפיעה לרעה על ביצוע של רכיבים מרכזיים במערכת.  - בעיית משאבים או ביצועים המשפיעה על תפקוד המערכת.  - כל תקלה שברור שהמשתמש הסביר לא יסכים לקבל. | Critical | 4 |
| - תקלה בולטת במיוחד לעיני הלקוח (גם אם בפועל ההשפעה שלה פחותה).  - בעיית משאבים או ביצועים חמורה אך עדיין מאפשרת שימוש. | Major | 3 |
| - תפקוד 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'. | Medium | 2 |
| - תקלות קלות אחרות בממשק הגראפי או בהיבטים אחרים. | Minor | 1 |

## סוגי בדיקות מערכת

* **בדיקות פונקציונליות** - לאימות פעילות המערכת. בדיקות אלו מבוססות על מסמך הדרישות ומסמך האפיון ומטרתן לבדוק כי המערכת עושה את מה שהיא צריכה ולא עושה את מה שאינה צריכה לעשות. יש לעשות בדיקות נקודתיות לפונקציות מקומיות ובדיקות כלליות של המערכת. הקלטים לבדיקות יהיו חוקיים ולא חוקיים לוודא שאכן המערכת מגיבה כצפוי בכל תרחיש.
* **בדיקות לא פונקציונליות** - בדיקות אלו בודקות איך פועלת המערכת וכוללות בדיקות עומסים, ביצועים, שימושיות וסוגי בדיקות רבים נוספים (שחלקם מפורטים למטה).
* **בדיקות תצוגה** - נוודא שממשק המשתמש עומד בדרישות שהוגדרו לו מבחינת פונקציונאליות, מבחינה ויזואלית ומבחינת סטנדרטים של עיצוב. בסיום הבדיקה נוכל לומר האם ממשק המשתמש מכיל את כל המרכיבים שהוא צריך להכיל והאם הם מתפקדים כיאות. יש למפות את כל מסכי המערכת ובכל מסך ממפים את כל השדות, פקדים, טקסטים ועוד. לאחר המיפוי ייבדק על רכיב בצורה פרטנית מול ההגדרות באפיון.
* **בדיקות שימושיות** - להעריך האם ממשק המשתמש יעיל, אסוציאטיבי, קל ונוח לשימוש, כך שיוכל להשתלב בקלות בתהליך עבודת המשתמשים ולא יכביד עליהם בשל בעיות עיצוב של ממשק לא נוח ולא ברור. בדיקת סובלנות של המערכת לטעויות המשתמש, הצגת הודעה במקרה של שגיאה ואפשרויות לתיקון. יש להתאים בדיקות מסוג זה לסביבת הלקוח כמה שניתן.
* **בדיקת ביצועים ועומסים** - עמידת המערכת בעומסים ובזמני תגובה הנדרשים. בדיקות אלו מאבחנות את יכולת המערכת להמשיך ולתפקד בצורה צפויה תחת עומסי עבודה שונים. יש להפעיל את המערכת תחת עומס ממוצע ועומס שיא ובחינת זמני הביצוע של תהליכים מרכזיים במערכת.
* **בדיקת רגרסיה** - בדיקות רגרסיה הן בדיקות רוחביות המבוצעות בכל סבב בדיקות, לקראת שחרור גרסת מערכת חדשה ועם סיום התקנת גרסה באתר הייצור, בכדי לאמת אי פגיעה בתקינות המערכת בעקבות תיקון / שינוי קוד. יש להריץ תרחישי בדיקות שנכתבו עבור תהליכים שפותחו ונבדקו בעבר.
* **בדיקות מסירה** (Factory Acceptance Test - FAT) - בדיקות מסירה של מערכת המפותחת ע"י גורם חיצוני. זוהי הבדיקה הסופית ואחרונה לפני בדיקות הקבלה אשר מבוצעות ע"י הלקוח.
* **בדיקות קבלה** (Acceptance Test Procedure - ATP) - בדיקות קבלה מבוצעות על ידי הלקוח ונציגיו, לאחר סיום מוצלח של בדיקות המערכת. מטרתן היא לוודא התאמת המערכת לצרכים ולדרישות שהוגדרו במשימת הפרויקט.

## אופי הבדיקות

### בדיקות אוטומטיות

בדיקות תוכנה הנעשות בצורה אוטומטית, רצוי עם מינימום התערבות אנושית. מגדירים את הבדיקות בכלי לבדיקות אוטומטיות והבדיקות רצות ומדווחות האם התוצאות בפועל מתאימות לתוצאות הצפויות.

יתרונות:

* ההגדרה הראשונית לוקחת זמן, אבל הריצות מהירות.
* הופך את תפקיד הבודק לפחות טכני.
* הגישה לתוצאות הבדיקות פתוחה לכל צוות הבדיקות והן מתועדות בצורה אוטומטית.

חסרונות:

* + עלויות רכישה.
  + לא מבטל הצורך בבדיקות ידניות, לדוגמה, בדיקות GUI.

### בדיקות תוכנה ידניות

בדיקות תוכנה הנעשות על ידי עובד שהוכשר לכך, בדרך כלל על פי תוכנית בדיקות מסודרת ומוסכמת. הבודק משתמש במערכת כפי שמשתמשי הקצה היו משתמשים בה ואז קובע אם התוכנית פועלת כראוי.

יתרונות:

* + עלות נמוכה בטווח הקצר.
  + הכי קרוב למציאות, לכן הסיכוי שיגלו בעיות אמיתיות יותר גבוה.
  + גמישות - שינוי תסריט הבדיקה קל ומיידי.

חסרונות:

* פעולות מסוימות קשה לבחון ידנית.
* בדיקות מסוימות יכולות להיות כל הזמן לחזור על עצמן וזה מקשה על העבודה.
* אחרי כל שינוי בקוד צריך לבצע את הבדיקות מחדש.

## אפקטיביות הבדיקות

מחשבים אפקטיביות של בדיקה ספציפית ביחס בין כמות הבדיקות שמצאה לכמות הבדיקות שהתגלו בסך הכל.

## תוצרים

יש שלושה מסמכים המופקים בשלב הבדיקות על ידי צוות הבדיקות:

1. Software Test Plan (STP) - מסמך המתאר מה אנו רוצים לבדוק, מה חשוב ומה פחות ומה סוג הבדיקות.
2. Software Test Description (STD) - מסמך המתאר איך הבדיקות יתבצעו בפועל. מתאר לכל בדיקה מה מטרתה, איך תתבצע ומה אמור להיות הפלט.
3. Software test Report (STR) - דיווח של תוצאות הבדיקות. מתאר איזה בדיקה הצליחה ואיזה נכשלה ומה סוג השגיאה. בכל סבב בדיקות מגישים STR מחדש.

# שיטות עבודה

## הגדרה

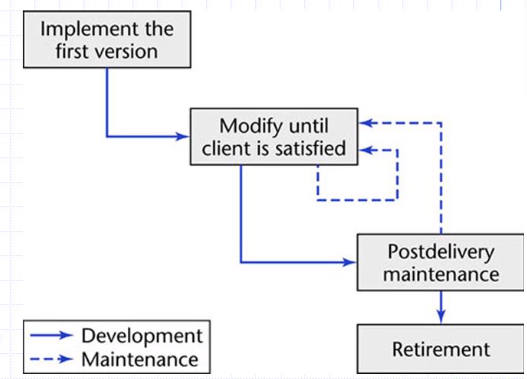
מחזור חיים של מערכת תוכנה שתיארנו הוא מודל תיאורטי, בפועל מחזור החיים משתנה בכל שיטת עבודה (נקרא גם "מתודולוגיית פיתוח"). בהנדסת תוכנה, שיטת עבודה היא סט מוסכם של עקרונות, תהליכים, פרקטיקות וכלים על פיהם מפותחות ומתוחזקות מערכות תוכנה.

בפרק זה נלמד על מספר סוגים שונים של שיטות עבודה, נפרט לכל אחת יתרונות וחסרונות במונחים של זמן שחרור הפיתוח, איכות וניהול סיכונים, ונערוך השוואה בין השיטות. שיטת העבודה שנבחר היא הבסיס לכל תהליך הפיתוח, החלטה טובה יכולה לשפר את איכות המערכת ולהקטין את הזמן שנדרש כדי לפתח אותה.

ניתן לחלק את שיטות העבודה לקבוצות הבאות:

* **מודל ליניארי**: מבצעים את כל השלבים ברצף כך שלא מתחילים שלב אחד עד שמסיימים את השלב הקודם. בפרט, שלב העיצוב והמימוש מבוצעים כיחידה אחת, כך שהתוצא הראשון הוא עצמו המערכת המלאה. מודל לא גמיש לשינויים. נחשב למיושן אך עדיין בשימוש במקומות רבים.
* **מודל איטרטיבי**: מטרתו להגיע לשלב המימוש מהר. העקרון שעובד לפיו הוא שלא ניתן לייצר מערכת מוכנה בבת אחת בגלל שהדרישות יכולות להשתנות תוך כדי הפיתוח. לכן יש לחלק את שלב המימוש לאיטרציות, כך שבכל איטרציה מוסיפים עוד נדבך למודל וכבר בהתחלה יש מערכת מאוד בסיסית מוכנה.
* **מודל Agile**: מודל מודרני שדומה לפיתוח איטרטיבי אך הותאם לפיתוח תוכנה בצוותים קטנים תוך שימת דגש של יעילות, זריזות ואיכות. מתמקדת בשיפור יכולתו של הצוות לספק תוצרים במהירות ולהגיב לדרישות העולות תוך כדי הפיתוח. נפרט על מודל זה בהרחבה בהמשך.

## Code & Fix

מתודולוגיה זו שמה דגש רב על המהירות שבה נעשים שינויים ותיקונים למערכת, תוך התעלמות מודעת מנושאי התחזוקתיות והאיכות הפנימית. מנסה להבין את הבעיה דרך בניית פתרון בגישה של ניסוי וטעיה. בשיטה זו מפתחים גרסה ראשונית, בלי לעשות שום תכנון לפני, ומשנים עד שהלקוח מרוצה. בשלב התחזוקה מתקנים כל פעם שיש בעיות.

יתרונות:

* קל לפיתוח מפני שלא דורש תכנון.
* מהיר.
* קל לניהול.
* גמיש לשינויים בדרישות לקוח ומאפשר הוספה מהירה של פונקציונאליות חדשה למערכת.

חסרונות:

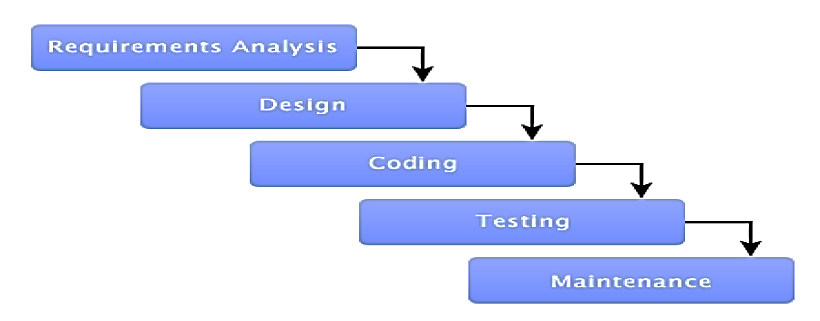
* סיכון מאד גבוה.
* גרוע למערכות מורכבות הדורשות תכנון.
* לא מתאים לפרויקטים מתמשכים.
* קשה לתחזוקה - עלות תיקונים גבוהה
* לא יודעים מתי יסתיים.

מסקנה: מתאים לפרויקטים מאוד קלים שיש להם תהליך בנייה ידוע.

## Waterfall

מודל ליניארי הדומה למחזור החיים שלמדנו. העקרון הוא לחלק את פיתוח המערכת לשלבים הגיוניים ולבצע כל שלב בפני עצמו. לא מתקדמים לשלב הבא עד שמסיימים את השלב הקודם, כל שלב הוא כמעין "קלט" לשלב הבא אחריו. המערכת מפותחת במחזור פיתוח אחד. בשיטה זו נותנים דגש רב על תכנון ועיצוב מוקדם של התוכנה, לפני שמתחילים לממש. כאשר יש שינוי באחד השלבים שסיימנו יש לחזור על כל השלבים העוקבים מהשלב בו בוצע השינוי עד לשלב הנוכחי. בפועל, פעמים רבות, עד כדי שליש או מחצית מזמן הפיתוח מוקדש להפקה של מסמכים ותיעוד ולא לכתיבת קוד.

שיטת עבודה זו נחשבת למיושנת, במיוחד שכיום החומרה והתוכנה משתנים כל הזמן ודרושה שיטת עבודה דינאמית. בנוסף, כיום קיימים כלים המאפשרים לבצע שלבי פיתוח במקביל בלי שייקח זמן לשנות במידה ויש שינוי בדרישות. לכן שיטה זו פחות רלוונטית כיום, אולם עדיין נפוצה במקומות רבים בעולם.



יתרונות:

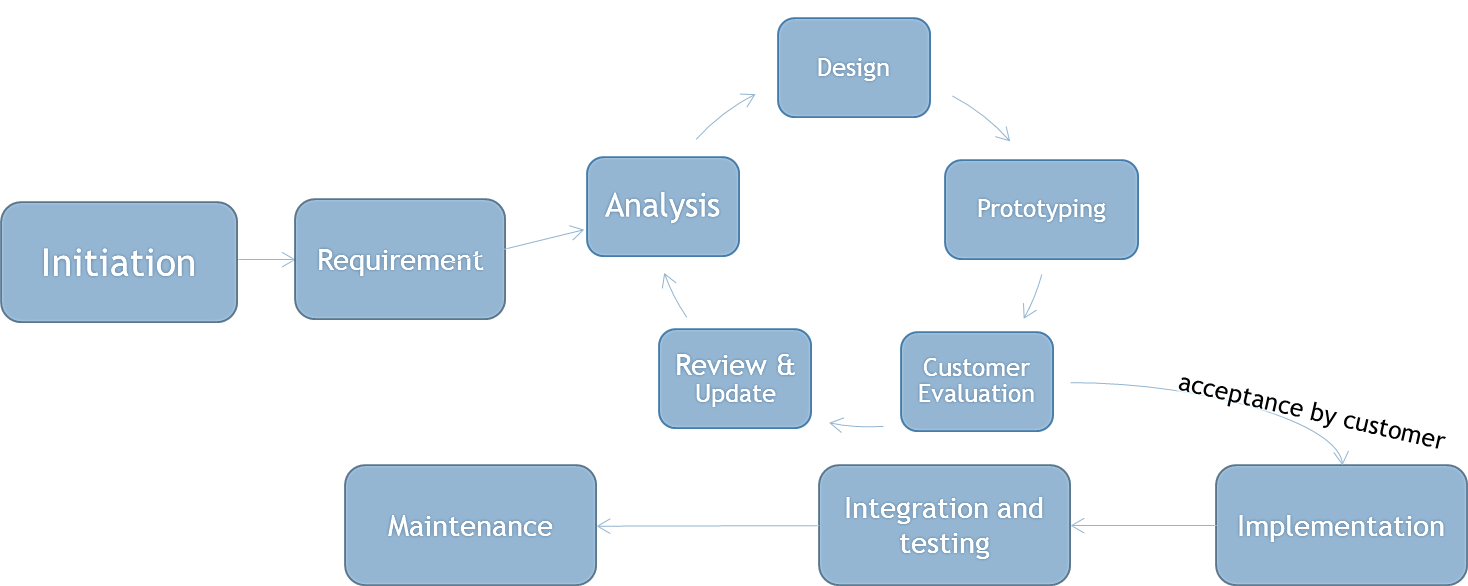
* פשוט, קל להבנה ולשימוש.
* קל לניהול, בגלל שהכל קשוח.
* אבני דרך ברורות.
* תהליך מתועד היטב.
* אחזקה קלה יותר.

חסרונות:

* שלב הפיתוח מגיע מאוחר מאוד בתהליך.
* סיכון גבוה: יכול להימשך זמן רב שבו משקיעים יפרשו ויכולה לעלות תחרות.
* לא טוב לפרויקטים מורכבים, ארוכים או דינאמיים.
* קשה למדוד את ההתקדמות באמצע שלב.
* אין גמישות לשינויים.

מסקנה: שיטה זו יעילה לפרויקטים שאינם מורכבים ודינאמיים, אין צורך לסיימם מהר והסדר בהם מאוד חשוב.

## אב טיפוס מהיר

שיטת עבודה ליניארית כמו מפל המים, כלומר עובדת שלב אחר שלב, אלא שמייצרת אבטיפוס הכי בסיסי ללקוח כדי שיאשר ורק לאחר מכן מוציאה את הגרסה המלאה. מודל זה בא לענות על מצב שבו דרישות הלקוח לא ברורות או אינן סופיות, ולכן מוציאים גרסה ראשונית שתעזור להגדיר את הדרישות למערכת הסופית. שלבי הניתוח והעיצוב הראשוניים הם רק עבור האבטיפוס ולא ממומשים בקוד אלא רק בצורה גרפית. אם הלקוח ביקש שינוי חוזרים על התהליך, ואם הסכים עוברים למימוש ותחזוקה.

יתרונות:

* מבהיר את דרישות הלקוח.
* מקל במקרים של קשיים טכניים.

חסרונות:

* גורר עלויות נוספות.
* לא מקל על סיכונים שמתגלים בתהליך המימוש.

מסקנה: שיטה זו יעילה כשאר דרישות הלקוח לא ברורות ומימוש המערכת פשוט מאוד.

## V-Model

מודל לינארי שבא לפשט ולשפר את שיטת מפל המים. בשיטה זו כל שלב מבוצע קודם ברמה הכללית ורק לאחר מכן יורדים לפרטים הקטנים. בנוסף, לכל שלב מוסיפה שלב נוסף של בדיקות בסדר הפוך לתהליך הבנייה, כלומר בודקים כל שלב מהמקרה הפרטי אל המקרה הכללי.

סדר הבדיקות הוא:

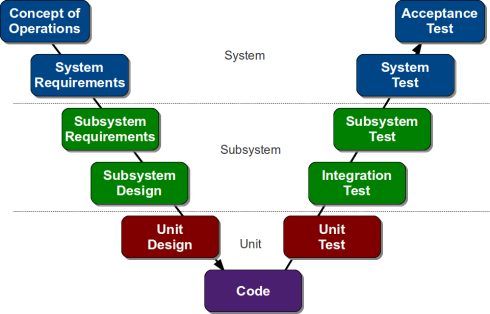
בדיקת יחידה - כדי להיפטר מבאגים בקוד.

בדיקות אינטגרציה - לוודא כי היחידות שנוצרו באופן עצמאי מתקשרות ביניהן.

בדיקות מערכת - מבטיחה כי הציפיות של הלקוח מהמערכת מתקיימות.

בדיקות קבלה - מוודא שמערכת שהועברה עומדת בדרישת המשתמש והמערכת מוכנה.

מלבד הוספת סדר לפיתוח ונתינת דגש לבדיקות מסודרות מודל זה זהה למפל המים מבחינת זמן הפיתוח והגמישות לשינויים.

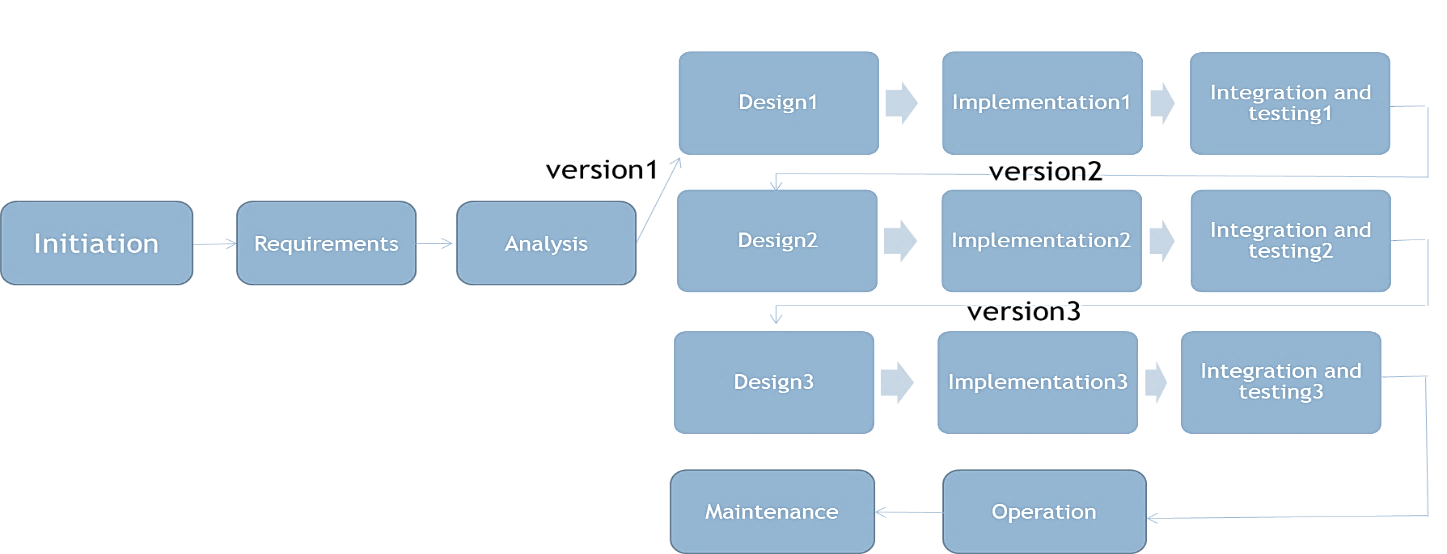


יתרונות:

* קל להבנה ולשימוש.
* קל לניהול בגלל התיעוד והסדר.
* השלבים ואבני הדרך מוגדרים היטב.
* חסכון בכסף ובמשאבים עקב תחילת תהליך הבדיקות בשלב מוקדם של הפרויקט.
* חסרונות:
* תהליך הפיתוח מתחיל מאוחר בשלבי הפרויקט.
* סיכון גבוה: יכול להימשך זמן רב שבו משקיעים יפרשו ויכולה לעלות תחרות.
* לא טוב לפרויקטים מורכבים, ארוכים או דינאמיים.
* אין גמישות לשינויים במהלך הסבב.

מסקנה: שיטה זו יעילה לפרויקטים שאינם מורכבים ודינאמיים, אין צורך לסיימם מהר והסדר בהם מאוד חשוב.

## Iterative Model

זהו מודל איטרטיבי בסיסי, שבו המטרה היא להגיע לשלב המימוש מהר. העקרון שעובד לפיו הוא שלא ניתן לייצר מערכת מוכנה בבת אחת, כמו המודל הליניארי, בגלל שהדרישות יכולות להשתנות תוך כדי הפיתוח. לכן הגיוני שנתחיל לממש את המערכת ונצטרך לשנות בהתאם לאילוצים ושינויים בדרישות. בשיטה זו יש מספר שלבים קבועים, ואז בכל איטרציה מפתחים גרסה של המערכת על ידי חזרה על שלבי העיצוב והמימוש. כבר בגרסה הראשונה יש מערכת מאוד בסיסית מוכנה, ובכל גרסה לאחר מכן מוסיפים עוד נדבך למערכת.

יתרונות:

* ניתן לראות תוצאות בשלב מוקדם של הפיתוח.
* אפשר לתכנן פיתוח מקבילי.
* ניתן למדוד את התקדמות הפרויקט.
* עלויות נמוכות יותר שכן כבר בהתחלה אפשר לראות טעויות מובנות בתכנון.
* יותר קל לבדיקה כי יש פחות בעיות בכל איטרציה.

חסרונות:

* דורש משאבים גבוהים כי צריך הרבה אנשי צוות.
* גמישות מועטה ביחס למה שכבר הוחלט.
* דורש ניהול צמוד וקפדני כי לא מוגדר היטב.
* אין תמונה מלאה של המערכת הנדרשת לפני המימוש.

מסקנה: שיטה זו יעילה במערכת שדי בור איך אמורה להיראות וצריך לייצר לה גרסה ראשונית מהר.

## Incremental model

כמו המודל האיטרטיבי אלא שבכל איטרציה בונים חלק אחר (מודול) של המערכת. בשיטה זו מתכננים את המערכת הסופית כבר בהתחלה, ובכל איטרציה בונים חלק אחר של המערכת, כך שכל חלק שבונים הוא סופי. בסוף כל איטרציה משלבים את מה שעשינו עם איטרציות קודמות.



יתרונות:

* יצירת תוכנה עובדת מהר ובשלב מוקדם של הפיתוח.
* גמיש יותר לשינויים.
* קל לבדוק איטרציות קטנות.
* הלקוח יכול להגיב לכל מודול בנפרד.
* קל יותר לנהל סיכונים, כי חלקים מסוכנים מוגדרים ומטופלים באיטרציה שלהם.

חסרונות:

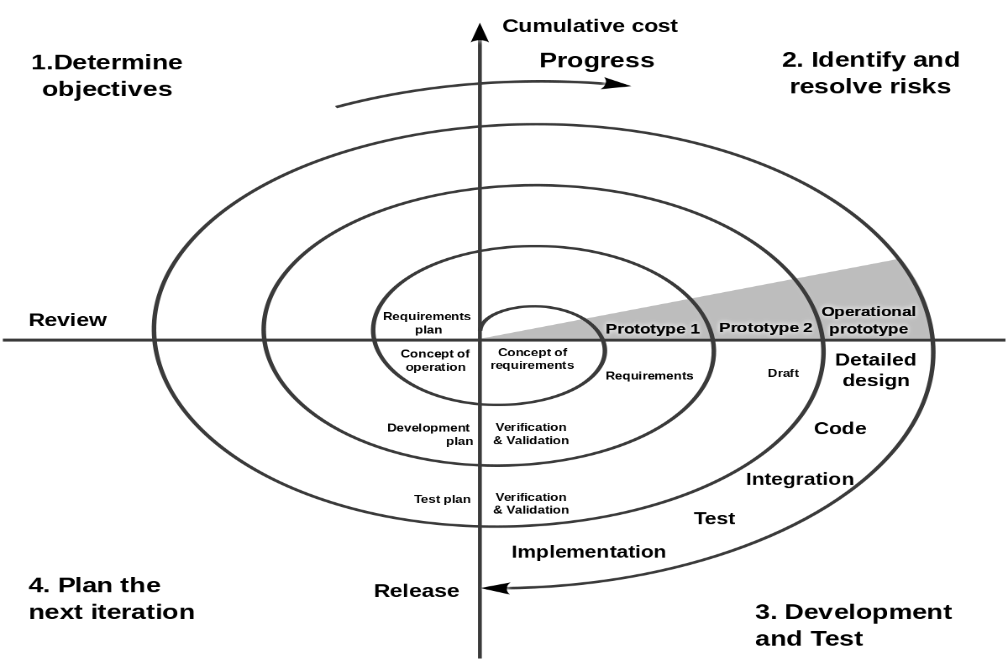
* דרוש תיכנון ועיצוב קפדניים בהתחלה.
* יש צורך להבהיר ולהגדיר את **כל** המערכת לפני שאפשר לחלק אותה למודולים.
* העלות הכוללת כנראה תהיה גבוהה יותר ממפל המים.

מסקנה: שיטה זו יעילה במערכות שיש להם כמה חלקים נפרדים שאינם תלויים יותר מדי אחד בשני.

## Spiral Model

כמו המודל האינקרמנטלי, כלומר פיתוח המערכת אינו נעשה בבת אחת וכל איטרציה מפתחים חלק אחר במערכת, אלא שהמודל הספירלי נותן דגש לניתוח סיכונים. כל איטרציה מבוססת על הקודמת ומחולקת לארבעה שלבים:

1. הגדרת מטרות לסבב הנוכחי.
2. זיהוי סיכונים ומציאת פתרונות.
3. פיתוח והטמעה (שחרור גרסה)
4. הערכת מה שפיתחנו באמצעות בדיקות ותכנון הסבב הבא.

עבור כל סיכון מנתחים מה ההסתברות שיתרחש, מה עלות ההפסד במידה ויתרחש, ומכפילים ביניהם כדי לקבל את הערכת הסיכון. כך נוכל לתעדף את כל הסיכונים ולנהל אותם, כלומר עבור כל סיכון לציין מה יש לעשות כדי להימנע ממנו.

יתרונות:

* ניהול סיכונים ברמה גבוהה.
* אפשר להתחיל מגרסה פשוטה ולהוסיף מורכבות בהמשך.
* הפיתוח מתחיל מוקדם בתהליך.

חסרונות:

* עלויות ניתוח גבוהות יותר לפרויקט.
* הצלחת הפרויקט תלויה בעיקר בשלב ניתוח הסיכונים.
* לא מתאים לפרויקטים קטנים.

מסקנה: שיטה זו מתאימה לפרויקטים גדולים שניתן לחלק אותם לחלקים נפרדים ויש בהם סיכון רב.

## מודל Agile

Agile  היא מתודולוגית פיתוח תוכנה איטרטיבית וזריזה, שהותאמה לפיתוח תוכנה בצוותים קטנים תוך שימת דגש על יעילות, זריזות ואיכות. העקרון המנחה הוא שלא ניתן להגדיר בצורה מלאה או לחזות מראש את כל הצרכים של תוכנה לפני תהליך הפיתוח בפועל, ולכן ומתמקדת במקום זאת בשיפור יכולתו של הצוות ומנהל הצוות לספק תוצרים ביעילות ובזריזות, ולהגיב לדרישות העולות תוך כדי הפיתוח.

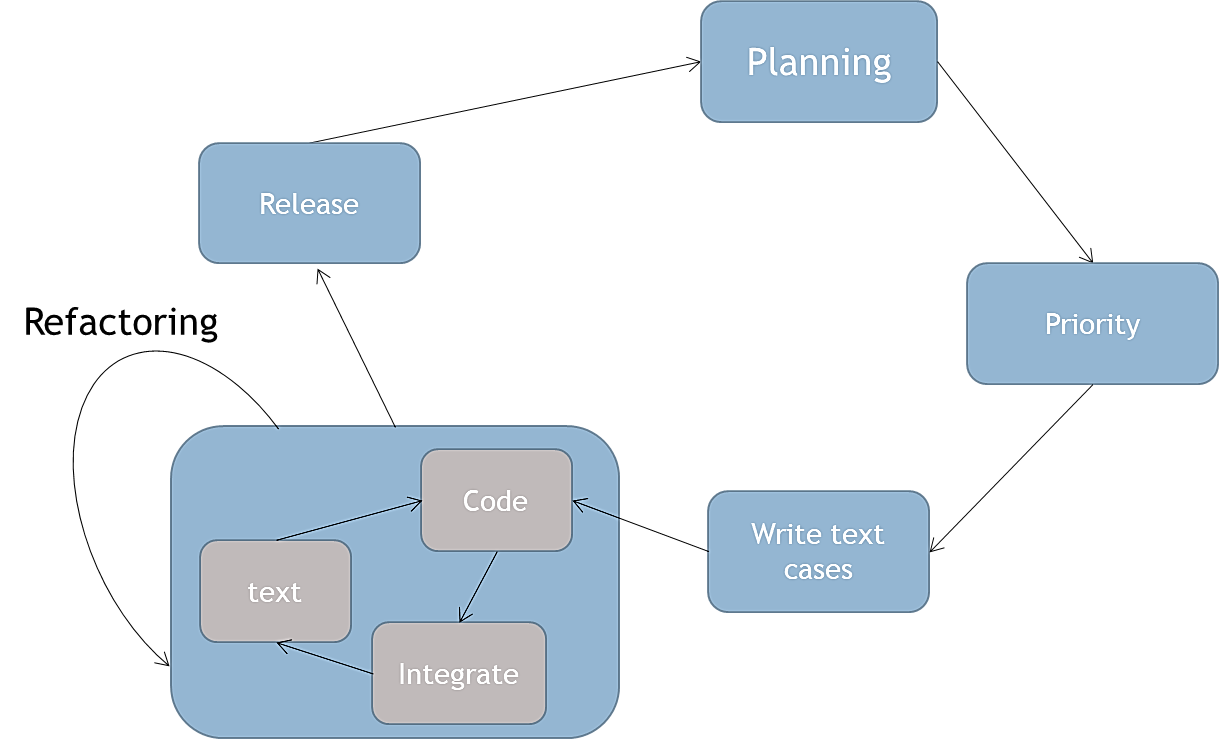
12 העקרונות של שיטה זו כלולות ב-Agile Manifesto והם:

1. העדיפות העליונה היא לשחרר ללקוח גרסה עובדת ובעלת ערך כמה שיותר מוקדם.
2. שינויים בדרישות הם טריגר לשיפור התוכנה ולכן נקדמם בברכה.
3. נרצה לשחרר גרסה עובדת לעיתים קרובות.
4. נדרש שיתוף פעולה וקשר יומיומי בין הלקוח, מומחי היישום והמפתחים לאורך כל הפרויקט.
5. יש לתת בחברי הצוות אמון שיבצעו את העבודה ולהחדיר בהם מוטיבציה.
6. צוותי הפיתוח הם האחראיים הבלעדיים לארכיטקטורה ולעיצוב.
7. הצוות משתפר ונהיה יותר אפקטיבי לאורך זמן.
8. תקשורת פנים אל פנים היא הדרך היעילה והטובה ביותר להעברת מידע.
9. תוכנה עובדת היא המדד העיקרי להתקדמות.
10. תחזוקה המערכת היא חלק בלתי נפרד מפיתוחה, וכל בעלי העניין והמפתחים צריכים לעשות זאת.
11. מצוינות טכנית, עיצוב ותכנון נכון מגדילים את היכולת האג'ילית.
12. יש לפתח את מה שהכרחי ולא לפתח את מה שלא.

מתודולוגיית Agile מתאימה בעיקר לפרויקטים בהם ישנה חוסר ודאות לגבי המוצר הסופי, מאחר והיא משלבת תהליכי למידה ושיפור מתמידים, בד בבד עם התקדמות הפרויקט. יש מספר שיטות עבודה שפותחו לפי עקרונות ה-Agile. בפרק זה נדון על שלושה שיטות מרכזיות: XP, Scrum, Kanban.

## XP - Extreme Programming

XP זו שיטת עבודה ששומרת על עקרונות Agile שפירטנו לעיל: פיתוח מוצר מהר בגרסאות קטנות, גמישות לשינויים, קשר רציף עם הלקוח, צוותים קטנים בעבודה פנים מול פנים וכו', אלא שמוסיפה שורה של פרקטיקות שימושיות בהן יש לפתח את המערכת:

* Pair Programming - פיתוח בזוגות. אחד כותב את הקוד והשני עוקב אחריו, מייעץ ומאתר תקלות בקוד. מתחלפים כל כמה זמן. הזוגות דינאמיים, ניתן להחליף זוגות ואף להחליף משימות בין זוגות.
* 40-hour week - לא להעמיס על המפתחים, הם צריכים לעבוד בזמן סביר.
* Continuous Testing - בדיקות יחידה נכתבות בטרם נכתב הקוד אותו הן בודקות. נדרש 100% הצלחה בבדיקות לפני שעוברים לשלב הבא.
* Simple Design - ההנחה היא שהתוכנה הטובה ביותר היא זו שעומדת בכל הדרישות, הקוד שלה מובן למתכנתים ויש בה המינימום הנדרש של מחלקות ושיטות, ולא זאת המתוחכמת ביותר.
* Coding Standards - יש לשמור על אחידות בקוד לפי כל הכללים, כך שלא יהיה ניתן להבחין מי כתב מה. בנוסף, כל המתכנתים מכירים את כל הקוד ולא רק מה שהם כתבו.
* Refactoring - שיפור קוד קיים על ידי שימוש בטכניקות שנועדו לשפר את המבנה הפנימי של הקוד מבלי לשנות את ההתנהגות החיצונית שלו.

יתרונות:

* מעורבות הלקוח מגדילה את הסיכוי שהמערכת תענה על צרכיו של המשתמשים.
* מקטין את הסיכון בפרויקט.
* בדיקות ואינטגרציה מתמשכות מסייעות להגביר את איכות העבודה.
* זמני הפיתוח קצרים יותר, כי מכינים את הבדיקות לפני.

חסרונות:

* מיועד לפרויקט אחד, שמפותח ומתוחזק ע"י צוות אחד.
* פגיע במיוחד בצוות שבו יש מפתחים שלא עובדים טוב עם אחרים. או שמפתחים לא יכולים לעבוד יחד.
* לא יעבוד בסביבה שבה לקוח או מנהל מתעקש על מפרט מלא או עיצוב לפני שלב המימוש.

## Scrum

מתודולוגיה לתהליך פיתוח תוכנה זריז המבוססת על עקרונות ה-Agile. המתודולוגיה מבוססת על ההבנה שפיתוח תוכנה היא לא בעיה שניתן לפתור ע"י חיזוי או ניתוח, אלא בעיה אמפירית שיש לפתור אותה בשיטה של ניסוי וטעיה, נתחיל לממש בהדרגה, נלמד מטעויות ונשפר את המערכת כל הזמן.

בשיטת Scrum יש לתחזק רשימת פריטי העבודה לביצוע, מסודרים לפי קדימויות, המכונה עתודת המוצר - Product Backlog. ב-Scrum עובדים במחזורים קצרים, נקראים ״ספרינטים״, האורכים שבועיים עד חמישה. בכל ספרינט מגדירים מסמך Sprint Backlog (נלקח מתוך Product Backlog), המכיל את כל מה שאנו רוצים להוסיף למוצר בספרינט נוכחי. בסוף הספרינט כל המשימות שהוגדרו בתחילת הספרינט עומדות בהגדרה של "בוצע", המוצר שמיש ובעל פוטנציאל לשחרור. ספרינט חדש מתחיל מיד לאחר סיום הספרינט הקודם. צוות הפיתוח עורך פגישות רבות פנים מול פנים.

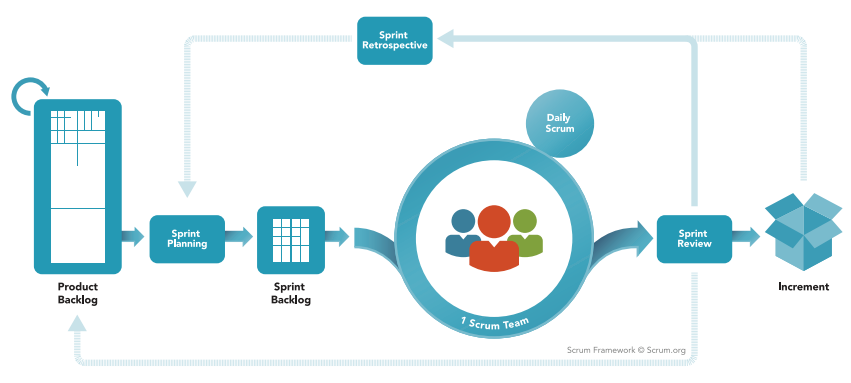
### בעלי תפקידים

ב-Scrum מוגדרים מספר בעלי תפקידים המחולקים לשתי קבוצות: מחויבים ומעורבים. יש מי שמחויבים לפתח את התוכנה באופן מחזורי ולספק תוצרים באופן סדיר, כל השאר הם מעורבים, יש להן עניין בפרויקט אך משקל דעותיהן זניח. אם הפרויקט יכשל, יהיו אלה המחויבים שישלמו את המחיר. השיטה מתחשבת בצרכים, ברצונות וברעיונות של המעורבים, אך אינה מתירה להן להפריע לפרויקט בכל דרך שהיא. בעלי העניין המעורבים הם המשתמשים, המנהלים והיועצים. למחויבים יש תפקיד רגיש יותר והם:

* **צוות ה-Scrum**: צוות קטן המונה 5 עד 9 אנשים בעלי מיומנויות רב-תחומיות המבצעים בפועל את העבודה.
* **Scrum Master**: מוודא שתהליך ה-Scrum מתבצע כנדרש.
* **בעל המוצר**: מייצג את אינטרס הלקוח כדי להבטיח שצוות ה-Scrum עושה את הדברים הנכונים מנקודת המבט העסקית.

### פגישות

לאורך הספרינט הצוותים מקיימים מספר סוגים של פגישות פנים מול פנים:

* **Sprint Planning** - מתבצעת בתחילת הספרינט ומטרתה להחליט איזה תוספות למוצר ניתן לספק בספרינט נוכחי וכיצד לעשות זאת. התוצר של פגישה זו הוא ה-Sprint Backlog.
* **Daily Scrum** - פגישה קצרה (15 דק') של חברי הצוות המתקיימת בכל יום ובה מתכננים את העבודה ליום הקרוב.
* **Sprint Review** - פגישה המתקיימת בסיומו של הספרינט על מנת לבדוק את התוספת שפותחה ועל מנת להתאים את ה-Backlog Product במידת הצורך.
* **Sprint Retrospective** - פגישה המתרחשת לאחר ה-Review Sprint ולפני ה-Planning Sprint הבא, ומטרתה לתת הזדמנות לצוות ה-Scrum לבדוק את עצמו וליצור תכנית שיפורים שיחולו על ה-Sprint הבא.

### בעיות ב-Scrum

* חוסר בהירות בנוגע לתכולה הסופית של הפרויקט. הלקוח עלול להמשיך לבקש עוד פונקציונליות חדשות.
* צוות שמנהל את עצמו לא יעבוד אם חברי הצוות לא מחויבים למטרה והפרויקט ייכשל.
* עבודה ביחידות קטנות יכול לגרום לבעיות כאשר חלק עובדים יותר וחלק פחות. בנוסף, חברי הצוות צריכים להיות מיומנים מאוד בשביל שהתהליך יצליח.
* מצריך שינויים מאד גדולים בארגון, גם ברמת ההנהלה וגם ברמת הצוותים המפתחים.
* תפקיד ה-scrum master קשה לביצוע וקשה לאיוש.

## Kanban

שיטה לתהליך פיתוח תוכנה זריז המבוססת על עקרונות ה-Agile. בשיטה זו יוצרים לוח המתאר איזה משימות כבר בוצעו, מה בתהליך ביצוע ומה עוד נשאר לבצע. כל חבר צוות כאשר מסיים משימה, ניגש ללוח מעדכן את המשימה שסיים ולוקח משימה חדשה. השיטה מעודדת ביצוע שינויים קטנים והדרגתיים במערכת.

הלוח צריך להיות מוחשי (פיזי או וירטואלי). רצוי לתאר כל סוג משימה בצבעים שונים. יש אפליקציות רבות שניתן דרכן לבנות לוח Kanban. דוגמה לאפליקציה כזו היא Trello.

יתרונות:

* מגביר את הגמישות לשינויים.
* מצמצם בזבוז זמן מפני שתמיד ממתינה עוד משימה.
* קלה להבנה ולהטמעה.
* מקצר את זמני המסירה של משימות.

חסרונות:

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

בשנת 2018 Scrum ו-Kanban הוציאו מדריך משותף לשיטה חדשה אשר משלבת את היתרונות בשניהם.

# ניהול פרויקט תוכנה

## הקדמה

בפרק זה נתאר את האופי של מנהל פרויקט תוכנה ומהם התפקידים הנדרשים ממנו בכל שלב. בנוסף, נתאר כלים שונים המייעלים את עבודתו של מנהל הפרויקט.

## תכונות חשובות למנהל פרויקטים

* תקשורת טובה עם אנשים - רוב זמן המנהל פרויקט הוא לתווך בין גורמי הפיתוח השונים.
* יודע לנהל משא ומתן בצורה טובה ובהרגשה נעימה ל-2 הצדדים.
* מנהיגות.
* יכולת ניהול צוות.
* יודע לנהל זמן.
* בן אדם מאורגן - לא יכול לארגן אחרים אם לא מאורגן בעצמו.
* יכולת ניהול סיכונים.
* מומחיות בנושא.
* חשיבה ביקורתית - יודע לנתח יתרונות וחסרונות של פעולה לפני שמבצעים אותה.
* מיומנויות תכנון וידע בשימוש בכלי תכנון.

## שלב הייזום

* הכרה בצורך לבצע - מה הרעיון שלנו ואיזה בעיה אמורה להיפתר.
* הגדרת ציפיות הלקוחות, ההנהלה ויתר בעלי העניין בפרויקט.
* הגדרת מטרות העל של הפרויקט.
* הגדרת היקף הפרויקט.
* בחירת הצוות הראשוני לביצוע הפרויקט.

## שלבי הדרישות והביצוע

* חלוקת המטרות העל ליעדים הניתנים למדידה.
* חלוקת הפרויקט לתתי משימות.
* קביעת לו"ז לביצוע המשימות.
* הערכת עלויות הפרויקט.

## שלבי העיצוב והמימוש

השלב הכי מקושר למנהל הפרויקט. הביצוע תלוי במידה רבה בשלב התיכנון. בשלב זה המנהל אחראי על:

* התאמה לכל חבר צוות את המשימה שמתאימה לו.
* הסבר על המשימות - לדאוג שהכל מובן ואם לא לדאוג להדרכה.
* מיקוד חברי הצוות במשימות.
* עמידה על הוצאה לפועל של התיכנון.
* שמירה על המשאבים המוקצים.
* יצירת קשר עם הלקוחות, חברי הצוות וההנהלה.
* סיפוק תוצרים ללקוח.

### בקרה

השוואה מתמדת בין הביצוע לתכנון תוך נקיטת פעולות מתקנות על מנת להקטין פערים בלתי רצויים. סוגי בקרות בפרויקט:

בקרת תכולה (שאכן מממשים את כל מה שמתוכנן), בקרת לו"ז, בקרת תקציב, בקרת סיכונים, בקרת איכות ובקרת שינויים.

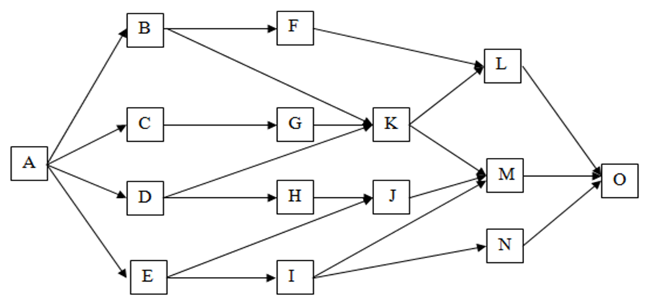
## שלב הסגירה

שלב זה קורה אחרי שמספקים ללקוח את המערכת. בשלב זה עושים ניתוח על התהליך שעברו חברי הצוות. המטרה היא ליצור צוות ותהליך טובים יותר לפרויקט הבא או כדי למכור את המערכת שפותחה ללקוחות נוספים. בשלב זה שואלים מה עבד? מה לא עבד? למה? איך המנהל יכול להשתפר? וכו'.

## ×ª××× × ×§×©××¨×WBS

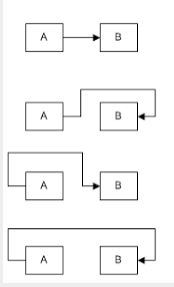
WBS (Work Breakdown Structure) הוא תרשים (בדר"כ של עץ) שבו מחלקים את הפרויקט לתת משימות הנותנות ביחד את התמונה המלאה של המערכת. הרמה העליונה מייצגת את המוצר הסופי, וברמה התחתונה נמצאים חבילות העבודה שכל אחת מהן ניתן להקצות לאדם מהצוות לביצוע. כל חבילת עבודה אומדים את המשאבים הדרושים לה מבחינת זמן עבודה ועלות. את החלוקה לחבילות העבודה ניתן לעשות במספר דרכים, לדוגמה: חלוקה לאבני דרך בתהליך, חלוקה לתוצרים, חלוקה לתחומי התמחות, לאתרי פעילות ועוד.

מסמך תכולת העבודה המוכן הוא מסמך מרכזי עבור מנהלי הפרויקט. מסמך זה משמש בעיקר לתכנון ומעקב, אך משמש כבסיס ליצירת כלי ניהול נוספים שנלמד בהמשך.

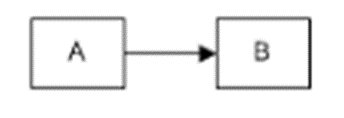


## תרשים רשת

כלי להצגת וניהול שרשרת המשימות בפרויקט. כל שרשרת מתארת רצף של משימות התלויות זו בזו, כלומר שלא ניתן להתחיל משימה לפני שמסיימים את הקודמת לה בשרשרת. שני משימות בשרשרות שונות יכולות להתבצע במקביל. תרשים זה מסייע למנהל הפרויקט לתכנן את סדר ביצוע המשימות בצורה יעילה ומאורגנת, בפיקוח על בקרת הפרויקט ובמציאת נתב קריטי כמו שנלמד בסעיף הבא.

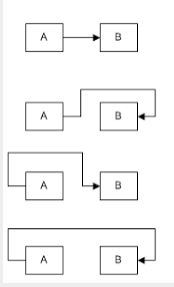


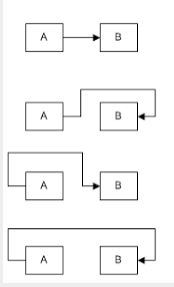
(2)



(1)

יש ארבעה סוגים של תלויות בתרשים רשת:

1. Finish to Start - B יכול להתחיל רק כאשר A מסתיים. זו ברירת המחדל.
2. Finish to Finish - A חייב להסתיים לפני B.



(4)

(3)

1. Start to Start - A חייב להתחיל לפני B.
2. Start to Finish - B מסתיים רק כאשר A מתחיל.

## CPM

CPM (Critical Path Method) זוהי שיטה ליצירת נתיב קריטי, שהוא שרשרת של משימות התלויות זו בזו, אשר כל שינוי בזמנים שלהם ישפיע על מועד סיום הפרויקט. משך הזמן של הנתיב הקריטי הוא גם הזמן המינימלי של הפרויקט. מאוד חשוב למנהל הפרויקט לזהות את הנתיב הקריטי כדי שיוכל להשקיע יותר משאבים במשימות הנמצאות בו כדי לא לעכב את זמן ביצוע הפרויקט. בפרויקט בו כל המשימות תלויות זו בזו ולא מתבצעות פעולות במקביל, הפרויקט כולו בנתיב הקריטי.

בפרויקט בו מתבצעות פעולות במקביל ישנם מספר נתיבים, כך שיכולים להיווצר מרווחים (slacks) המאפשרים לדחות פעילויות מסוימות בלי לדחות את סיום הפרויקט. נוכל להשתמש במרווחים אלו כדי להאריך את מועד הביצוע של משימות מסוימות וכן לבצע השלמות למשימות שלא הושלמו במלואן. המרווחים על הנתיב הקריטי הם תמיד 0.

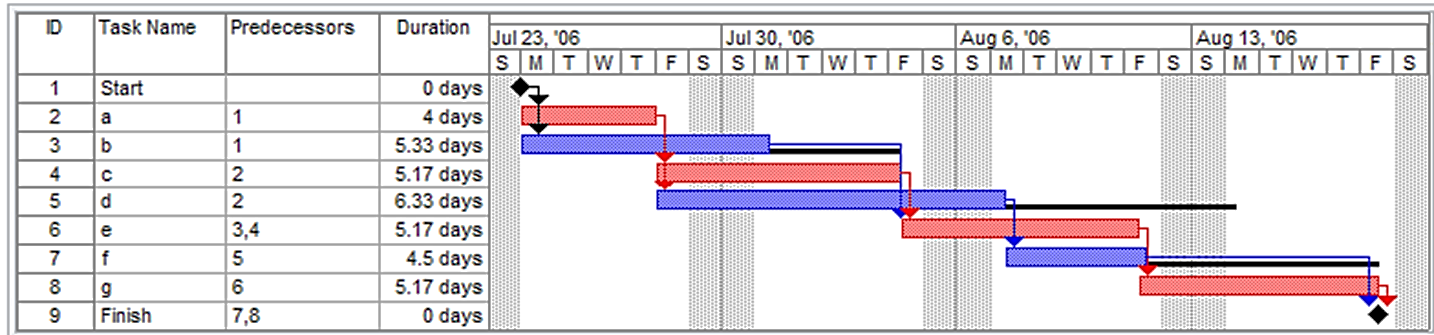
כדי למצוא את הנתיב הקריטי יש לחשב לכל משימה את הזמנים הבאים:

* **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.
* **Slack**: זמן שבו המשימה יכולה להמתין ללא עיכוב בפרויקט.

## תרשים Gantt

זו השיטה הנפוצה ביותר להצגת לו"ז הפרויקט והתרשים השימושי ביותר על ידי מנהלי פרויקטים. מייצרים גרף שבו הציר האופקי מייצג את הזמן והציר האנכי מייצג את חבילות העבודה. כל שיטת עבודה מצוינת בגרף באורך הזמן שלוקח לבצע אותה, ומתחילה ומסתיימת לפי חבילות העבודה שהיא תלוי בהן. חישוב הזמן לכל חבילת עבודה הוא לפי נוסחת Pert.

בגף זה בדרך מייצגים בנוסף לחבילות העבודה גם אבני דרך, שהם זמנים מיוחדים המהווים נקודות בקרה במהלך פיתוח המערכת. יש אלמנטים נוספים שנהוג לציין בתרשים Gantt.



## תרשים Pert

Pert (Program Evaluation Review Technique) זוהי נוסחה באמצעותה ניתן לקבוע אומדנים של עלות וזמן למשימות בפרויקט. משתמשים בזמנים שמקבלים מנוסחה זו כדי להעריך את הזמן של כל משימה בתרשים רשת. עבור כל משימה יש לתת 3 אומדנים: אופטימי, סביר ופסימי. נוסחת Pert המספקת את האומדן היא:

יתרונות:

* מאפשר לנתח ולהעריך את הזמן, העלות והמשאבים שדרושים לפרויקט כולו.
* נותן תמונה מלאה על הפרויקט למנהל הפרויקט ולשאר בעלי העניין.
* מאפשר מעקב אחרי ביצוע הפרויקט.

חסרונות:

* לא נותן עלות מדויקת אלא משוערכת.
* ממוקד בנושא הערכות הזמנים ופחות בדברים האחרים (לכן זה מולבש על תרשים רשת).
* שיטה מורכבת וקשה לביצוע בצורה טובה - דרוש מקצוען שייתן את שלושת הערכות.

יש אומדנים נוספים להעריך זמן ועלויות שתיארנו [בשלב הייזום](#_בדיקת_ישימות_ועלות).

## ערך מזוכה

הביצועים הנוכחיים הם האינדיקטור הטוב ביותר לבדיקת הביצועיים העתידיים, ולכן באמצעות ניתוח המגמה העכשווית ניתן לחזות את עלות הפרויקט ואיחור בלו"ז בשלב מוקדם של הפרויקט. השיטה הנפוצה ביותר היא שיטת ערך מזוכה (Earned Value Method). בשיטה זו מחשבים ערך מזוכה סביב נקודת בקרה ספציפית (נקבעת נקודת חיתוך בזמן), אשר מהווה נקודת ייחוס אחידה לכל הפרמטרים השונים שנפרט מיד. השיטה למעשה תקפה רק לאחר שהפרויקט החל.

* BAC (Budget At Completion) - עלות כוללת המתוכננת לפרויקט.
* AC (Actual Cost) - העלות בפועל עד נקודת בקרה t.
* PV (Planned Value) - העלות שתוכננה עד זמן t.
* EV (Earned Value) - העלות שתוכננה עד הגעה למצב נוכחי.

במידה והשלבים לא אחידים בעלות שלהם, זמן הביצוע הוא ביחס לעלות של כל שלב.

* CV (Cost Variance) - ערך שלילי מייצג כמה חרגנו מהתקציב, וערך חיובי כמה חסכנו מהתקציב. **.**
* SV (Schedule Variance) - הפרש בין עלות שתוכננה עד הגעה למצב נוכחי לעלות המקורית עד t. **.**
* CPI (Cost Performance Index) - יחס בין מה שסיימנו למה שהיינו אמורים לסיים לפי העלות עד כה.
* SPI (Schedule Performance Index) - יחס בין מה שסיימנו לכמה שתכננו לסיים עד זמן t.
* EAC (Estimated at Completion) - תחזית עלות הפרויקט כולו לפי מצב נוכחי.
* ETC (Estimated to Complete) - כמה עוד תקציב צריך להיות לנו.
* VAC (Variance at Completion) - תחזית כמה אנו נחרוג סה"כ מהתקציב לפי מצב נוכחי. .

## כיצד לצמצם לו"ז

* **Re-estimation** - בדיקה מחודשת של ההנחות, אולי יש שיטה או כלי מהירים ותר ממה שאנו משתמשים.
* **Crashing** - הוספת יותר משאבים לנתיב הקריטי, תוך שמירה על התכולה. דורש הגדלה של התקציב.
* **Fast Tracking** - ביצוע מטלות בנתיב הקריטי במקביל. מגביר סיכונים וסיבוכיות. לא אידיאלי, אבל שימושי לעיתים.
* **משולש האילוצים** - זהו משולש המורכב מארבעה חלקים שכל אחד מהם מייצג אילוץ אחר בפרויקט: לו"ז, תכולה, תקציב ואיכות. השקעה באחד האילוצים בהכרח תבוא על חשבון אילוץ אחר. בהקשר כאן, נוכל לקצר את הלו"ז על ידי צמצום התכולה, הורדה באיכות או השקעה בתקציב. זוהי כמובן שיטה לא רצויה ונשתמש בה כמוצא אחרון.