1. מבוא

1.1 סקירה היסטורית:
אין הנדסת תוכנה. יש פיתוח ותחזוקה של מערכות תוכנה גדולות.

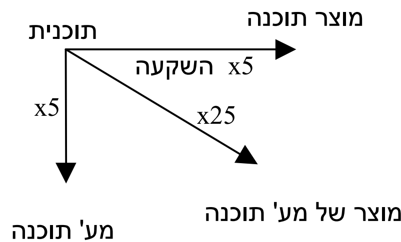
הגדרה 1.1:
הנדסת תוכנה היא הביסוס והשימוש בשיטות ועקרונות הנדסאיים בריאים כדי להשיג תוכנה כלכלית ואמינה שיכולה להתבצע על מחשבים אמיתיים.
הגדרה 1.2:
הנדסת תוכנה היא פיתוח ותחזוקה של תוכנה בעלת גרסאות מרובות (ולא גרסה חד-פעמית) ע"י צוות (ולא מפתח יחיד).

הגדרה 2.1:
תהליך הוא סדרה של משימות שמביאות לתוצאה רצויה אם מבצעים אותן בהצלחה.
הגדרה 2.2:
תהליך של הנדסת מוצר הוא תהליך לפיתוח או תחזוקה של תוכנה. תהליך של הנדסת תהליכים הוא תהליך לפיתוח ושכלול של תהליך להנדסת מוצר. תהליך תוכנה הוא או תהליך של הנדסת מוצר או תהליך להנדסת תהליכים.

Safety Critical: חיי אדם בסכנה כאשר מפעילים את המערכת (למשל: Therac).
Mission Critical: המשימה בסכנה אם המערכת לא מתפקדת (למשל: שליחת טיל לא מאויש).

1.2 סיבות לכשלונות ולבעיות:

תוכנית: עושה דבר מוגדר אחד.
מע' תוכנה: מורכבת ממודולים רבים.



אם איש בודד יכול לסיים את הפרוייקט תוך 25 שנה ⇍ 25 מתכנתים יסיימו תוך שנה, זאת בגלל:
1. לא ניתן להטיל משימות תכנות על מתכנתים, כמו משימות בנושאים מקביליים.
2. ככל שיש יותר בנ"א שעובדים על אותו פרוייקט אי הסדר גדל (אנתרופיה גדלה).

2. מודלים לתהליכי תוכנה:
2.1 מודלים למחזור חיים
(1) תכנת ותקן.

(2) (שרטוט) מודל מחזור החיים הקלאסי (מודל המפל): כולם משתמשים במודל, אבל הוא לא שווה דבר. למה?
שלבי המודל:
א. Requirements Definition: מנסחים מסמך דרישות. כיצד מנסחים אותו? דבר ראשון קובעים את בעלי העניין בארגון (stake holders). מראיינים אותם ושואלים אותם מה צריך להיות בתוכנה. המודל נכשל כבר בקביעת הדרישות. הבעיות:
1. לא יודעים מי הם כל בעלי העניין ולא שואלים את כולם.
2. לא כל בעלי העניין מסכימים זה עם זה.
3. IKIWISI – I Know It When I See It
4. חוסר שלמות – לא זוכרים דברים בזמן הראיון.
5. חוסר קונסיסטנטיות.
6. במודל המפל רואים את התוכנה אחרי שנתיים.

עקרון COTS: Commercial Of The Shelf: לא מפתחים את כל רכיבי המערכת. בודקים האם יש רכיבים שניתן לרכוש.

ב. בשלב התחזוקה צריך לגשת לדרישות ולעבור דרך העיצוב והתיעוד. כלומר, צריך קשר דו-סטרי בכל השלבים. אולם כל השלבים לוקחים המון זמן ואף פעם לא מבצעים אותם עד הסוף.

Gold Plating: השאיפה שלא יחסר שום דבר במערכת ← המחיר יעלה ושלב הפיתוח יעלה.

למה משתמשים בו? נוהל מפת"ח הולך לפי מודל זה. מודל המפל קובע אוטומטית אבני דרך: זה טוב למפתח ולממן הפרויקט. בכל שלב ושלב ידוע מה המוצר ולפי זה נותנים כסף.
המודל טוב למערכות מידע.

(3) (שרטוט) מודל אב-הטיפוס:
א. האבולוציוני: בניית אב-טיפוס יציב. אם משקיעים מספיק באב-טיפוס, אזי כשמגיעים לאיטרציה האחרונה (לאב-טיפוס האחרון) וכיוון שהלקוח כל הזמן בתמונה ← כל הדרישות מולאו.
ב. המהיר: בניית אב-טיפוס לא יציב. הרעיון: להציג בפני הלקוח כמה שיותר מהר (IKIWISI), כך שהלקוח יוכל להגיד האם לזה או התכוון. משקיעים פחות מאמץ ולכן פחות זמן בפיתוח כל פעם. כשמראים ללקוח את המע' האחרונה צריך להסביר לו שהמע' אינה יציבה ורק עם אישור הלקוח מסתיים שלב הדרישות ואח"כ ממשיכים הלאה במודל המפל.
2 האפשרויות טובות, כיוון ששתיהן נותנות בטחון שהדרישות מולאו. מודל אב-הטיפוס המהיר דומה למודל *תכנת ותקן*, אך כעת אנו עושים את המודל הזה כדי להגיע לדרישות הסופיות, ואילו במודל תכנת ותקן לא היה מוכר המושג הזה.
המודל הזה חשוב יותר היום כי המנשק כיום חשוב וצריך להיות ידידותי למשתמש.

(4) (שרטוט) מודל הטרנספורמציה:
זהו מודל אוטופי שלא קיים. הרעיון: כתיבת דרישות המע' בלוגיקה מסדר ראשון (פרולוג). כיוון שכותבים את זה בשפת תכנות ← זוהי המע' וסיימנו. הרעיון לא פועל כי זה לא יעיל. רעיון המודל:
1. מפרט פורמלי בלוגיקה מסדר ראשון.
2. מכניסים את המפרט הפורמלי (T) לתוכנית שהופכת אותו למשהו יותר יעיל (R) (התוכנה הזו לא קיימת).
מפעילים את שלב 2 עוד כמה פעמים ובסוף מקבלים תוכנה מוכנה ויעילה.
לכל הטרנספורמציות הנ"ל אסור לשנות את הדרישות. צריך בדיקה שהטרנספורמציה שומרת על השלמות (P).

(5) (שרטוט) המודל הספירלי:
הרכבה של מודל המפל ומודל אב-טיפוס. יש שלב פיתוח נוסף של *הערכת סיכונים*.
הרעיון: לפני כל שלב עושים הערכת סיכונים של אותו שלב: מה הסיכויים שבשלב איסוף הדרישות אני אצליח?
אם מחליטים שאין סיכון גדול בשלב הזה מבצעים אותו והולכים לפי מודל המפל (בוקטור ישר מהמרכז החוצה).
אם מחליטים שנושא הדרישות מעורפל אזי בונים אב-טיפוס רק כדי לקבל את הדרישות (עושים סיבוב מלא שמתחיל ב-planning ומסתיים בהערכת הלקוח).
במודל זה ניתן לבנות אב-טיפוס בכל שלב ושלב.

(6) (שרטוט) מודל ה-Win-Win:
מודל זה נובע מהעובדה שלבעלי העניין יש דיעות סותרות. בכל שלב צריך להגיע למצב win-win, שכולם מרוצים. צריך לפשר בין הדיעות.

<u>2.2 הגדרת התהליך ומודל עבורו</u>
<u>דוגמה: דיווח טעויות ותהליך שינוי:</u>
1. פרוייקט תוכנה מתנהל כשחלק מהמע' עוצב, תוכנת, עבר בדיקה ו-baselined (כאשר יש גרסה שהיא נטולת בגים שמים אותה ב-baseline. רק אנשי צוות איכות המוצר מכניסים ל-baseline ולא המתכנת עצמו).
2. tester מדווח על בעיה [איך מדווחים על הבאג? למי זה נמסר?] שנמצאת במודול שכבר נמצא ב-baseline.
3. המודל המצריך תיקון נמצא בשימוש ע"י משתמשים אחרים. מודיעים להם על הבעיה.
4. מוציאים את המודל מה-baseline. ממשיכים להשתמש בגירסה הישנה של המודול.
5. משנים את המודול כדי לפתור את הבעיה.
6. בודקים את המודול. כשהבעיה נפתרה מתקיימים תהליכים לאישור/דחיית המודול. כשהמודול מאושר עושים regression testing על כל המודולים שמשתמשים במודול הנוכחי.
7. מכניסים את המודול חזרה ל-baseline.
הבעיה היא שהשלבים אינם מפורטים: בכל שלב יש חור שלא כתוב מי צריך לעשות ומה לעשות. המודלים למחזור חיים הם גסים (גרעון גס מדי) ולא מתייחסים לבנ"א.

הגדרה 3:
1. תחזוקה מתקנת עוסקת בתיקון שגיאות שנכנסו תוך כדי תהליך התוכנה (הן בשלב הפיתוח והן בשלב התחזוקה).
2. תחזוקה מתאמת עוסקת בהתאמת מערכת תוכנה לשינויים בדרישות ומפרטים של המערכת (התחזוקה המתאמת חייבת להיות כל הזמן).
3. תחזוקה משכללת עוסקת בשיפור הביצועים של המערכת.
4. תחזוקה יצירתית עוסקת בשילוב רעיונות חדשים במערכת.

<u>מה כן אפשר לעשות?</u>

פורמליזציה:

שלשות Hoare: { P} S { Q }:
P – תנאי לוגי של תנאי מוקדם: pre condition.
S – סדרה של פעולות/פקודות.
Q – תנאי לוגי של תנאי מאוחר: post condition.

משמעות כל שלשה: נניח שיודעים ש-P מתקיים, מבצעים פעולות S ואז נובע שהתנאי Q יתקיים.
<u>חסרון:</u> חוסר גמישות. עוסקים בבנ"א: זה אף פעם לא סטטי, אלא דינמי. צריך גמישות שמגיעה לעתים גם לאלתורים. גישה כזו פורמלית היא בעלת גרעון עדין מדי.

הגדרה 4:
שלשה של Hoare: {P}S{Q} היא טענה על תוכנית S ועל הנוסחאות הפרדיקטיביות P, Q. P נקרא תנאי מוקדם ו-Q נקרא תנאי מאוחר. הסמנטיקה של שלשת Hoare: אם P אמת לפני הביצוע של S אזי Q אמת לאחר הביצוע של S.

<u>*2.3 CPT: Chief Programmer Team*</u>
גישה לתהליך תוכנה שפותחה ונוסתה ב-IBM.
<u>רקע:</u> ננסה להטיל על אדם אחד לפתח תוכנה לבד. כיוון שזה ייקח הרבה זמן נעמיד לרשותו צוות, שהוא יעמוד בראשו. המתכנת הראשי אחראי על הכל (תכנות, תיעוד, קידוד). בצוות יש co-chief, מנהלן (administrator), מזכיר, מכשירן (אם המתכנת הראשי זקוק לפונקציה ספציפית, המכשירן עושה את זה), מומחה לשפה (בלשן), עורך לשוני, ספרן.
המטרה: שלא ייקח לתוכניתן 20 שנה לבנות את זה. ייקח לו יותר זמן מאשר ל-20 איש, אבל עם פחות טעויות.
<u>חסרונות:</u> 1. המנהל צריך להיות הבוס ולא מספר 3.
2. מי מבטיח שהסיבה להצלחה היא השימוש ב-CPT? אולי היה פרמטר אחר שגרם להצלחה.
3. לא הצליחו למצוא אנשים אחרים שיהיו Chief Programmer.

חוק Brooks: תוספת כ"א לפרוייקט נמצא באיחור יגרום לאיחור נוסף.
למה? הפרוייקט באיחור ולכן כולם בלחץ. כשמוסיפים מישהו, אנשים אחרים יבזבזו זמן להסביר לו במה מדובר. הוא יכול לגרום לבאג שאותו יצטרכו לתקן ולבזבז עוד זמן. אי-הסדר גדל (אנתרופיה גדלה).

<u>*3. פורמליזמים ויזואליים I*</u>

<u>*3.1 FSM, דיאגרמות למעברי מצבים:*</u> [דוגמאות: עמ' 11-12]
הרכבת רכיבים ב-State Transition Diagram היא כפלית, כלומר: אם יש לי k מצבים לרכיב n ואני רוצה n כאלו אזי צריך $n_1 * n_2 * \ldots * n_k$. לא ניתן לתאר מקביליות ב-FSM.
<u>הרעיון:</u> הדרישות מנוסחות בשפה טבעית. יש נטייה לעשות פורמליזציה כדי למנוע עמימות וחורים. הדיאגרמה נבנית כדי לגלות שגיאות, לראות שהיא מתאימה לדרישות וכו'.
כל עיגול ב-FSM הוא מצב.

הגדרה 5:
1. מכונת מצבים סופית היא שלשה סדורה (S, I, f) כאשר S, I הן קבוצות סופיות ו-f: S x I → S היא פונקציה חלקית.
2. דיאגרמת מעברי מצבים המתאימה ל-FSM: (S, I, f) היא גרף מכוון בעל צלעות עם תוויות המתקבל בצורה הבאה:
   א. לכל איבר של הקבוצה הסופית S מתאים קודקוד של הגרף.
   ב. קיים צלע מקודקוד a לקודקוד b אם"ם קיים $i \in I$ כך שמתקיים f(a, i) = b.
   ג. הצלע המכוון מקודקוד a לקודקוד b הוא בעל התווית i.
3. הסמנטיקה של FSM = (S, I, f) היא כדלהלן: S היא קבוצה של מצבים של מערכת, I היא קבוצה של מעברים ממצב אחד למצב אחר, כאשר המעברים האפשריים מוגדרים ע"י f.

<u>*3.2 רשתות Petri:*</u> [דוגמאות: עמ' 12-14]
כל עיגול כאן הוא תנאי: p_i – תנאי מסוים. $p_i – t_i$ הוא תנאי עמ"נ שניתן יהיה לבצע את המעבר t_i.
אסימון (נקודה שחורה): אם יש אסימון ← מתקיים התנאי והמעבר אפשרי. אחרי המעבר, האסימון עובר גם כן.
בחלק מהמקרים אין חיבור אסימונים, כלומר: אם מגיע עוד אסימון למצב בו יש אסימון – יהיה רק אסימון אחד בסוף. האסימונים יכולים לשאת אינפורמציה.
ברשת פטרי לא ניתן לדעת איזה תהליך יתבצע. יש אי-דטר'. ע"י רשת פטרי קל יותר לגלות חוסר קונסיסטנטיות, חורים.
הרכבת רכיבים ברשת פטרי הוא אדדיטיבי ולא כפלי. פה תיתכן מקביליות.

הגדרה 6:
1. רשת פטרי היא רביעייה סדורה PN = (S, T, F, MI), כאשר S קבוצה סופית של 'מקומות', T קבוצה סופית של 'מעברים', F קבוצה סופית של 'חצים' ו-MI סימון התחלתי של ה'מקומות' ע"י 'אסימונים'.

2. הסמנטיקה של PN = (S, T, F, MI) היא כדלהלן: ה'מקומות' מייצגים תנאים של מערכת, ואילו הימצאות 'אסימון' ב'מקום' מסוים מסמן קיום אותו תנאי. 'מעבר' אפשרי אם כל ה'חצים' המובילים ל'מעבר' באים מ'מקומות' עם 'אסימונים'. כאשר מתרחש 'מעבר', מועברים ה'אסימונים' מה'מקומות' הקשורים ב'חצים' נכנסים ל'מקומות' הקשורים ב'חצים' יוצאים מה'מעבר'.

3.3 State Charts (Higraphs) [דוגמאות: עמ' 14-16]

2 סוגי של Higraphs: activity chart ו-state chart. Higraph מורכב מדיאגרמת ון וגרף. בגרף היחס הוא בינארי. ב-Hypergraph היחס הוא טרנארי: יש על-צלע שאומר שזה לא 2 צלעות נפרדות, אלא צלע אחת משותפת.

כל קבוצה היא מלבן מעוגל שמסמן מצב של המע'. קבוצה שלא מסומנת ואין לה שם אינה קיימת. כל שטח שקיים חייב להיות מסומן ובעל שם. קו מקווקו = מכפלה קרטזית. בתרשימים מכפלה קרטזית היא קומוטטיבית. מכפלה קרטזית מציינת שלמצב יש 2 אספקטים. חץ עם נקודה בתחילתו מציין מצב התחלתי.
g(c) – מה שכתוב בסוגריים זה תנאי שחייב להתקיים כדי שמעבר יוכל להתקיים.
m / e – המעבר m גורר אחריו את המעבר e היכן שהוא מופיע.
f [in(G)] – תנאי: המעבר f ייתכן רק אם אנחנו במצב G ברכיב מסוים.

דיאגרמת E-R: כל ישות היא מלבן. מציינים יחסים בין ישויות ותכונות של ישויות ע"י מעוינים.
ב-Higraph כל ישות היא קבוצה, יחס הוא קשת ← יותר תמציתי וקריא.

הגדרה 7:
דיאגרמת ישויות-יחסים היא שלשה סדורה (E, R, A), כאשר E קבוצה של ישויות, R קבוצה של יחסים בין הישויות ו-A קבוצה של תכונות של הישויות ושל היחסים.

Activity Chart: המלבנים המעוגלים מציינים פעולות. הצלעות מסמנות זרימת נתונים.

הגדרה 8:
1. גרף הוא זוג סדור (V, E) כאשר V קבוצה של קודקודים ו-E קבוצה של צלעות בין הקודקודים.
2. גרף משרה יחס בינארי על V: קיים יחס f(a, b) אם"ם יש צלע מ-a ל-b.
3. על-גרף (hypergraph) הוא קבוצה V של קודקודים ויחס f (לאו דווקא בינארי) על V. קיים על-צלע מקודקוד a לקודקודים b, c,… אם"ם קיים f(a, b, c, …).
4. higraph הוא פורמליזם ויזואלי של קבוצות (המתוארות ע"י מלבנים מעוגלים) ויחסים (לאו דווקא בינאריים) המגדירים צלעות ועל-צלעות בין הקבוצות. כן מוגדרים תת-קבוצות, חיתוך ומכפלה קרטזית בין הקבוצות.
5. State chart הוא higraph עם הסמנטיקה הבאה: הקבוצות הן מצבים של מערכת והצלעות (על-צלעות) מסמנות מעברים בין המצבים. מכפלה קרטזית של שתי קבוצות מסמלת מצב מורכב משני המצבים המיוצגים ע"י הקבוצות.
6. activity chart הוא higraph עם הסמנטיקה הבאה: הקבוצות הן פעולות או פונקציות, ותת-קבוצות מסמנות תת-פעולות (אין מכפלות קרטזיות). הצלעות מסמנות זרימת נתונים.

4. פורמיליזמים ויזואליים 2: UML

5. שיטות פורמליות

בקביעת המפרט, בפורמליזציה, הלקוח בד"כ אינו משתתף. את המפרט נותנים למתכנת. אחרי התכנות בודקים את המע' עפ"י המפרט. אף אחד לא מבטיח שהקשר בין המפרט לעולם האמיתי הוא שלם, לכן יש צורך לעשות בדיקות של המע' עם העולם האמיתי. כלומר: גם אם בנינו את המע' על-סמך מפרט פורמלי ⇍ המע' תהיה טובה, כי צריך גם לבדוק אותה לפי העולם האמיתי.

5.1 מפרטים תיאוריים:

מתארים תכונות/התנהגות של המע'.

5.1.1 מפרטים אקסיומטיים:

יש מנשק (=פרמטרים ותוצאה) והתנהגות. ההתנהגות מתוארת ע"י שלשות Hoare. דוגמה:
Interface: gcd(integer, integer) returns integer.
Behavior: {x > 0 & y > 0} gcd(x,y) { divides(gcd(x,y), x) & divides(gcd(x,y), y), …}
<u>למה משמש המפרט?</u>
אחרי הקידוד בודקים שהמע' מקיימת את המפרט.
<u>בעיה:</u> לפעמים ניסוח הבעיה הוא טריוויאלי, אולם הפתרון מסובך, כי הפתרון צריך להביא בחשבון מס' אלגוריתמים.

5.1.2 מפרטים אלגבריים: [דוגמאות: עמ' 22-24]
יש חלוקה ל-4 שכבות:
1. Elem אומר שאני מגדיר מפרט אלגברי עבור משתנים שהם מסוג Elem.
   imports – שימוש במפרט אחר.
2. הגדרה בלתי פורמלית (בשפה טבעית) של האובייקט והפונקציות הקשורות לו.
3. Signature: עבור כל פונקציה: סוג הפלט וסוג הקלט.
4. אקסיומות: בניית האקסיומות מתבצעת ע"י הפעלת **פונקציות Inspection** על פונקציות **Construction**.
   ניתן לחלק את כל הפונקציות במפרט ל-2 סוגים: Inspection Operators, Construction Operators.

אם רוצים להגדיר השוואה בין 2 טיפוסים של Elem יש לציין זאת בחלק העליון.
instantiates: נותן ערך למחלקה ע"י השמת סוג משתנה במקום Elem.
undefined – דבר שבמקומו ניתן לשים Elem.
*** enrich - דומה להורשה: מקבלים את מה שיש ב-*** ומוסיפים עליו דברים.
ניתן להגדיר בשכבה 4 גם exceptions.

<u>הגדרת התהליך של בניית מפרט אלגברי:</u>
1. קבע שם ופרמטרים גנריים (למשל: שם = list, פרמטר = elem).
2. בחר פעולות חשבון: inspection operators and construction operators.
3. כתוב מפרט לא פורמלי.
4. הגדר את הסיגנטורות של הפונקציות.
5. הגדר אקסיומות: בד"כ, הפעל כל אופרטורי ה-inspection על כל אופרטורי ה-construction.
6. קבע טיפול בשגיאות: א. undefined ב. error ג. exception
7. שימוש חוזר במפרטים: א. instantiation ב. ירושה

5.2 מפרטים תפעוליים: [דוגמאות: עמ' 25-27]
<u>שפת Z</u>: שפה לכתיבת מפרטים תפעוליים.
Z – המספרים השלמים.
N – מסמנת type שהוא מס' טבעי.
Δ – הסכמה המצוינת עוברת שינוי.
x? – x הוא קלט.
x! – x הוא פלט.
x’ – הערך לאחר השינוי.
Ξ– הסכמה לא תשנה את הסכמה המצוינת.
seq – סדרה.
IP – קבוצת חזקה.
⇸ פונקציה חלקית: לכל איבר בתחום יש תמונה.
↦ בניית זוג סדור.
dom(x) – התחום של x.
▷ - צמצום: לוקחים את הפונק' הכללית שמקשרת בין x ל-y ומצמצמים את הטווח.

<u>הגדרת התהליך של כתיבת סכמות ב-Z:</u>
1. מפרט בלתי-פורמלי.
2. פירוק המערכת.
3. עבור כל מרכיב של המע':
   א. הגדר קבוצה וטיפוסים.
   ב. הגדר משתני מצב.
   ג. הגדר פעולות תקינות.
   ד. הגדר פעולות לא-תקינות.
   ה. הגדר exceptions.
   ו. הרכב סכמות בחזרה.

יש 3 סוגי סכימות:
1. ***State Schema***
2. ***Operation Schema***
3. ***Observation Schema***

הגדרה 9:
1. התחום של מפרט נתון spec, domain(spec), הוא קבוצת כל ערכי <u>הקלט</u> שעבורם המפרט מוגדר. התחום של תוכנית נתונה prog, domain(prog), הוא קבוצת כל ערכי הקלט שעבורם התוכנית מוגדרת.
2. spec(x) יסמן <u>הפלט</u> שתוכנית אמורה לתת עבור x∈ domain(spec), ו-prog(x) יסמן הפלט המתקבל מביצוע התוכנית prog עבור x ∈domain(prog).

3. תוכנית prog מספקת מפרט spec, Sat(spec, prog), אם"ם מתקיימים התנאים הבאים:
   $domain(spec) \subseteq domain(prog)$
   $prog(x) = spec(x)$ עבור כל $x \in domain(spec)$.
4. למפרט נתון spec יכולים להיות מספר פגמים פורמליים:
   א. ייתכן כי קיים ערך x ולא ברור האם $x \in domain(spec)$.
   ב. ייתכן כי $\exists\, x \in domain(spec)$ אך עבורו spec(x) אינו מוגדר.
5. spec נקרא עקבי אם"ם קיים לפחות prog אחד המקיים Sat(spec, prog).
6. spec נקרא רב-משמעי אם $\exists\, x \in domain(spec)$ ומוגדרים שני ערכים שונים עבור spec(x). במקרה זה:

$$\exists\, prog_1, prog_2 : \quad Sat(spec, prog_1), Sat(spec, prog_2), \quad prog_1(x) \neq prog_2(x)$$

הגדרה 10:

1. אלגברה הומוגנית $(S, O_1, O_2, \ldots)$ היא קבוצה בודדת S ופעולות חשבון $O_1, O_2, \ldots$, המוגדרות על S.
2. אלגברה הטרוגנית $(S, T, \ldots, O_1, O_2, \ldots)$ היא אוסף של קבוצות S, T, … ופעולות חשבון $O_1, O_2, \ldots$ המוגדרות מעל לקבוצות S, T, ….

הגדרה 11:

1. מפרט תפעולי מגדיר את ההתנהגות הדינמית של המע'. מפרט תיאורי מגדיר את התכונות של המע'.
2. קיימים שני סוגי מפרטים תיאוריים פורמליים, הנקראים גם מפרטים מכווני תכונות: מפרטים אלגבריים ומפרטים אקסיומטיים. מפרטים תפעוליים פורמליים נקראים גם מפרטים מכווני מודלים.

5.3 אימות תוכניות:

<u>הרעיון:</u> להוכיח הוכחה מתמטית לוגית שמע' התוכנה מתנהגת בהתאם למפרט.
{P}S{Q} – אם ידוע שיש תנאי לוגי P ואנו מבצעים פקודות S אזי לאחריהן יתקיים תנאי לוגי Q.

כללי הוכחה: הרעיון: אם מה שכתוב במונה נכון, אזי נובע מהכלל שמה שכתוב במכנה נכון גם.

First consequence rule:

$$\frac{\{P\}S\{Q\},\ Q \Rightarrow R}{\{P\}S\{R\}}$$

Second consequence rule:

$$\frac{P \Rightarrow Q,\ \{Q\}S\{R\}}{\{P\}S\{R\}}$$

Assignment rule:

$$\{P^x_e\}\ x := e\ \{P\}$$

Compound rule:

$$\frac{\{P_{i-1}\}S_i\{P_i\},\ i = 1, 2, \ldots, n}{\{P_0\}\ \text{begin}\ S_1, \ldots, S_n\ \text{end}\ \{P_n\}}$$

לפי כלל ההרכבה רואים שאם רוצים לאמת תוכנית שלמה ← ניתן לעשות את זה קטע-קטע.

Conditional rules:

$$\frac{\{P \,\&\, B\}S\{Q\},\ P \,\&\, \text{not } B \Rightarrow Q}{\{P\}\ \text{if } B \text{ then } S\ \{Q\}}$$

$$\frac{\{P \,\&\, B\}S_1\{Q\},\ \{P \,\&\, \text{not } B\}S_2\{Q\}}{\{P\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Case rule:

$$\frac{\{P \,\&\, (x = k_i)S_i\{Q\}}{\{P \,\&\, x \in \{k_1, \ldots, k_n\})\}\ \text{case } x \text{ of } k_1: S_1, \ldots, k_n: S_n \text{ end}\ \{Q\}}$$

First loop rule:

$$\frac{\{P \,\&\, B\}S\{P\}}{\{P\}\text{while } B \text{ do } S\ \{P \,\&\, \text{not } B\}}$$

Second loop rule:

$$\frac{\{P\}S\{Q\},\ Q \,\&\, \text{not } B \Rightarrow P}{\{P\}\ \text{repeat } S \text{ until } B\ \{Q \,\&\, B\}}$$

ניתן להוכיח את הכללים הנ"ל ע"י שימוש בתרשימי זרימה. ע"י תרשימי זרימה ניתן להוכיח נכונות חלקית ולא שלמה.

מיתוסים של שיטות פורמליות:

1. שיטות פורמליות יכולות להבטיח שהתוכנה היא מעולה. *כמו שיש באגים בתוכנית, יכולים להיות באגים בהוכחות פורמליות.*
2. שיטות פורמליות הן אימות תוכניות. *שיטות פורמליות קשורות גם בכתיבת מפרטים פורמליים.*
3. שיטות פורמליות יעילות רק עבור מע' safety-critical. *אין מניעה להשתמש בשיטות פורמליות למע' תוכנה פשוטות.*
4. שיטות פורמליות דורשות מתמטיקאים מוכשרים.
5. שיטות פורמליות מעלות את עלות הפיתוח. *אף אחד לא הוכיח ששיטות פורמליות מעלות את מחיר הפיתוח.*
6. שיטות פורמליות לא מקובלות על המשתמשים. *למשתמש לא איכפת איך פיתחו את התוכנה.*
7. לא משתמשים בשיטות פורמליות על מערכות גדולות אמיתיות.
8. שיטות פורמליות דוחות את תהליך הפיתוח. *יכול להיות שזה מעכב את הפיתוח, אבל חוסך זמן אח"כ.*
9. אין מספיק כלים לשיטות פורמליות. *היה נכון בעבר, כיום יש כלים אוטומטיים.*
10. שיטות פורמליות מחליפות את שיטות העיצוב ההנדסי המקובלות. *הן בנוסף. הן לא מחליפות דבר.*
11. שיטות פורמליות מיועדות רק לתוכנה. *משתמשים בשיטות פורמליות גם בחומרה (VLSI).*
12. שיטות פורמליות אינן הכרחיות.
13. לשיטות פורמליות אין תמיכה.
14. אנשים שמתעסקים בשיטות פורמליות, משתמשים רק בשיטות פורמליות.

הגדרה 12:

1. יהי spec מפרט פורמלי ויהי prog תוכנית מתאימה. אימות פורמלי של prog הוא הוכחה לוגית שקיים Sat(spec, prog).
2. באימות ע"י שלשות Hoare: הוכחת {P}S{Q} נקראת נכונות חלקית של S. נכונות שלמה דורשת הוכחת סופיות הביצוע של S עבור כל ערכי קלט המקיימים את P.
3. משפט: תהי $\{x_i\}$ סדרה מונוטונית יורדת של מספרים שלמים אי-שליליים. אזי הסידרה סופית.

6. מידות תוכנה

6.1 שורות קוד:

רוב הדברים שניתן למדוד הם פונקציה של מס' שורות הקוד שבתוכנית: $\alpha * DSL^{\beta}$ (מודל להערכת משאבים: זמן + כסף).
(DSL = Deliverd Source Lines). הבעיה: איך סופרים את מס' שורות הקוד: האם סופרים גם את השורות שהן הערות? בנוסף, ניתן לכתוב בשורה אחת או ב-4 שורות – איך נספור את זה?

סיבוכיות של תוכנה: כמה קשה לבנ"א להבין את אותה מע' תוכנה.

6.2 מידת הסיבוכיות של McCabe: [דוגמאות: עמ' 30-31]

הרעיון: נגדיר את *גרף זרימת **הבקרה** של התוכנית*: לוקחים את התוכנית. אם יש פקודות סדרתיות אחת אחרי השנייה מקבצים אותן והן הופכות לקודקוד. מפקודות תנאי (if … else) עושים פיצול לכל כיוון (ל-then ול-else). אם יש לולאה אזי חוזרים מהקודקוד לקודקוד אחר. case: פיצול ל-x מקרי ה-case.
מדברים על פונקציות ולכן יש נקודת כניסה אחת ונקודת יציאה אחת.
להבין פונקציה פירושו להבין את גרף הבקרה של הפונקציה ← עיקר הסיבוכיות היא ב**בקרה**, בפיצולים. להבין את גרף הבקרה זה לעבור בכל המסלולים האפשריים מנקודת ההתחלה ועד הנקודה הסופית. כיוון שלעבור על כל המסלולים זה בלתי אפשרי, הגדירו **מסלולים בלתי תלויים**. לעבור על המסלולים הבלתי תלויים זה אפשרי וזה מספר קטן יחסית של מסלולים.
מידת הסיבוכיות של McCabe: מס' המסלולים הבלתי תלויים. רצוי שהסיבוכיות לא תהיה גדולה מ-10.

פקודות הכרעה: פקודת הכרעה פשוטה = if. אם יש case – נפצל לכמה if. גם לולאות for ניתן להעביר ל-if.

בעיות:

1. שתי תוכניות זהות: אחת עם 2 if מקוננים, השנייה עם if שהתנאי שלו יותר מורכב (ובעצם כולל את ה-if המקונן). לפי McCabe, if מקונן יותר קשה להבנה ואילו התנאי המורכב לא תורם לסיבוכיות.
2. שתי תוכניות להם יש אותה סיבוכיות לפי McCabe: אחת עם if מקוננים, השנייה עם if סדרתיים.

הסיבוכיות של המע':

הצעה 1: סכום הסיבוכיות של פונקציות במע'. נניח שהמע' מורכבת מ-3 פונקציות בה יש רק קוד סדרתי ← סכום הסיבוכיות הוא 3. אולם אם נבנה מע' שיש בה פונקציה אחת שהכל שם סדרתי ← הסיבוכיות היא 1. כלומר: ***הסיבוכיות אינה חיבורית (additive).***

הצעה 2: כדי להגדיר את סיבוכיות המע' נגדיר את הסיבוכיות המצומצמת לפי גרף הבקרה המצומצם: חיתוך החלק של גרף הבקרה שאין לו שייכות למודולים האחרים. לכל פונקציה נחשב סיבוכיות מצומצמת ובסוף, אם n זה מס' המודולים, אזי ***הסיבוכיות המצומצמת של המערכת*** =

$$\sum_{i=1}^{n} v_{i(reduced)} - n + 1$$

מדוע להחסיר n? אם יש פונקציות שלאחר הצמצום נקבל עבורן קודקוד בודד נקבל $\sum_{i=1}^{n} v_{i(reduced)} = n$ הנורמליזציה תגרום לכך שבסוף נקבל 1.

הגדרה 14:

1. גרף הבקרה של תוכנית הוא גרף מכוון המתקבל עבור פונקציה בדרך הבאה: נקודת הכניסה והיציאה של הפונקציה מתאימות לקודקוד כניסה וקודקוד יציאה בהתאמה. כל קטע קוד עם זרימה סדרתית מתאים לקודקוד בודד. צלעות הגרף מתאימות להסתעפויות של פקודות התניה.
2. המספר הציקלומטי של גרף מכוון קשיר בעל n קודקודים ו-e צלעות הוא: e – n + 1 (המס' הזה הוא חסם עליון למס' המסלולים הב"ת). *[המספר המדויק הוא e – n + 2, שכן מוסיפים את הצלע מנק' היציאה חזרה לנקודת הכניסה]. [הוכחה: אינדוקציה].*
3. גרף נקרא קשיר חזק אם"ם יש מסלול מכוון מכל קודקוד לכל קודקוד (לכן מוסיפים צלע מנקודת היציאה לנקודת הכניסה בגרף זרימת הבקרה).
4. משפט: מספר המסלולים הסגורים הבלתי תלויים בגרף קשיר חזק שווה למספר הציקלומטי.
5. מספר המסלולים הבלתי תלויים דרך גרף הבקרה של פונקציה שווה לכל היותר למספר הציקלומטי + 1.

הגדרה 15:

1. הסיבוכיות הציקלומטית של פונקציה היא מספר המסלולים הב"ת דרך גרף הבקרה שלה (McCabe).
2. משפט: הסיבוכיות הציקלומטית שווה למספר פקודות ההכרעה הפשוטות + 1.

הגדרה 16:

1. גרף נקרא מישורי אם אפשר להציג אותו במישור בלי שאף שתי צלעות יחתכו אחת את השנייה.
2. משפט Euler: בגרף מישורי בעל n קודקודים, e צלעות ו-f דפנות מתקיים: n – e +f = 2.
3. משפט: הסיבוכיות הציקלומטית של פונקציה שווה למס' הדפנות של גרף הבקרה המתאים.

הגדרה 17:

1. הסיבוכיות המצומצמת של פונקציה היא הסיבוכיות לאחר מחיקת כל הקודקודים של גרף הבקרה אשר יש להם אך ורק השפעה פנימית.
2. סיבוכיות האינטגרציה של מערכת היא: $V_{int} = \sum v_r(M_i) - n + 1$, כאשר M_i הן פונקציות של המע' ו-$Vr(M_i)$ זה הסיבוכיות המצומצמת של הפונקציה M_i.
3. V_{int} שווה למספר פקודות ההכרעה הפשוטות של המערכת לאחר צמצום + 1.

הוכחת 3:

$$\sum v_{i(reduced)} = \sum [\text{מספר פקודות הכרעה פשוטות לאחר צמצום} + 1]$$

$$V_{int} = \sum v_i - n + 1 = \sum [\text{מספר פקודות הכרעה פשוטות לאחר צמצום} + 1] - n + 1$$

$$= n + \sum [\text{מספר פקודות הכרעה פשוטות לאחר צמצום}] - n + 1$$

$$= \sum [\text{מספר פקודות הכרעה פשוטות לאחר צמצום}] + 1$$

Flow in מודד מה סיבוכיות הפונקציה לפי הזרימות פנימה (= לפי מקורות המידע מהם הפונקציה יונקת = מס' הפרמטרים הנכנסים). אפשרות נוספת לזרימה פנימה: שימוש במשתנה גלובלי, דוגמה:

struct a
f(int x, y, z) { … := a… }

Flow out: דוגמה:

struct a
f(int x, y, z) { … a :=… }

משתנה יכול לתרום גם ל-*flow in* וגם ל-*flow out*.

נניח שיש מבנה עם 3 שדות. כיצד נספור אותו: כ-3 או כ-1? כנראה סופרים כ-1. כנ"ל בנוגע למערך באורך 1,000 – נספור את המערך כולו כ-1.

הגדרה 18:

Flow-in = מספר הזרימות (פרמטרים) המקומיות לתוך הפונקציה + מספר מבני הנתונים החיצוניים (גלובליים) שמהם הפונקציה מקבלת מידע.

Flow-out = מספר הזרימות המקומיות מהפונקציה + מספר מבני הנתונים החיצוניים המקבלים ערך מהפונקציה.

Information Flow Metric : Complexity $_{H\text{-}K} = (Flow_{in} * Flow_{out})^2$

הערה: **מידת Goedle:** כתיבת הפונקציה כמחרוזות של 0, 1, כאשר המידה שווה לערך הבינרי.

הגדרה 19: קריטריונים אפשריים למידת סיבוכיות – Weyuker:

p, q – פונקציות, m – מידת הסיבוכיות:

1. $\exists p, \exists q, p \neq q : \quad m(p) \neq m(q)$ לא לכל התוכניות יש אותה מידה.
נכון עבור כל המידות.

2. $\forall n \geq 0$ קיימות מספר סופי של תוכניות בעלות מידת סיבוכיות m. על המידה לא להיות יותר מדי coarse grained (גרעין גס מדי).
אקסיומה זו אינה נכונה, למשל, עבור:
א. LOC.
ב. McCabe.

3. $\exists p, \exists q, p \neq q : \quad m(p) = m(q)$ על המידה לא להיות יותר מדי fine-grained (Goedel's Measure).
נכון לגבי LOC, McCabe.

4. $\exists p, \exists q, spec(p) = spec(q) : \quad m(p) \neq m(q)$ על המידה להיות מסוגלת להבחין בין 2 מימושים שונים של התוכנית. (הערה: תכונה 4 ← תכונה 1). *המידה של הסיבוכיות היא לא מידה של האלגוריתם, אלא מידה של התוכנית.*

5. $\forall p, \forall q : \quad m(p) \leq m(p+q), \quad m(q) \leq m(p+q)$ קטע תוכנית לא יכול להיות יותר מסובך מהתוכנית השלמה.
לגבי LOC, Goedle זה מתקיים. לגבי McCabe נקבל שוויון. לא מתקיים ב-Helstead's effort measure.

6. $\exists p, \exists q, \exists r : \quad m(p) = m(q)\ and\ m(p+r) \neq m(q+r)$ תכונה זו גורמת למידות להיות context sensitive (רגישה להקשר).
לא קיים ב-LOC, ב-McCabe וב-Goedle. קיים ב-Helstead's effort measure.

7. $\exists p, \exists q, q = permutation(p) : \quad m(p) \neq m(q)$ גם תכונה זו גורמת למידות להיות context sensitive ביחס ל-features בתוכנית.
לא נכון לגבי LOC ו-McCabe. נכון ב-Goedle.

8. *If q is a renaming of p then: m(p) = m(q). [renaming = למשל: החלפת שמות משתנים]*
נכון לגבי LOC ו-McCabe. לא נכון ב-Goedle.

9. $\exists p, \exists q : \quad m(p) + m(q) < m(p+q)$ תכונה זה משקפת את העובדה שיכולה להיות אינטרקאציה בין חלקים מחוברים, מה שיגדיל את הסיבוכיות.
לא קיים ב-LOC וב-McCabe. נכון ב-Goedle.

הגדרה 20: OOP (Object-Oriented) Complexity Measures

1. *WMC = Weighted Methods per Class = Sum of measures of methods of the class.*
2. *DIT = Depth of Inheritance = # of superclasses up to the root.*
ככל שהעומק גדול יותר ← הסיבוכיות גדלה, מהסיבה שכדי להבין מה יש במחלקה יורשת צריך להבין את כל מה שהיא יורשת.
3. *NOC = Number of Children = # of subclasses immediately below measured class.*
הסיבה לכך היא שכאשר כותבים את המתודות והמחלקות של מחלקה, חושבים מה יהיה בתתי המחלקות שלה.
4. *Two objects are coupled if methods of one uses methods or instance variables of the other.*
CBO = Coupling between Objects = # of non-inheritance related couplings with other classes.
5. *RFC = Responses for a Class = # of methods of class + # methods called from other classes.*
6. *LCOM = Lack of Cohesion in Methods = # of disjoint sets formed by intersection of the sets of instance variables used by each method.*

דוגמה:

class A { int $a_1, a_2, \ldots ,a_n$;
function f_1, function f_2, … function f_n }

מסתכלים על המתודות ובודקים באילו ממשתני המחלקה משתמשים במחלקות השונות. אם, למשל, רק f_1, f_2 משתמשות ב-a_1,a_2 ורק f_3, f_4 משתמשות ב-a_3, a_4 זי יש איזושהי הפרדה, ואולי היה צריך להגדיר 2 מחלקות שונות: אחת עם f_1, f_2 והשנייה עם f_3, f_4 ומשתניהם.

<u>החידוש של OOP:</u> Information hiding, Encapsulation, Inheritance, Abstraction.

6.3 CMM: The Capability Maturity Model: מודל בגרות היכולת:

רמות המודל:

1. <u>Initial</u>: רמת תוהו ובוהו. חברה שהיא ברמה 1 מתאפיינת בזה שאין הגדרה לתהליך התוכנה.
2. <u>Repeatable</u>: חברה ברמה זו כבר הצליחה לבצע פרוייקט מסוים ומסוגלת לחזור על פרוייקט דומה. מה צריך להיות בתוך החברה כדי שהחברה תאושר לרמה זו:
   ניהול תצורה/גרסאות, אבטחת טיב איכות, קבלני משנה, מעקב אחרי פרוייקט התוכנה, ניהול דרישות.
3. <u>Defined</u>: תהליך התוכנה מוגדר. יש:
- *peer reviews*:בכל שלב של מחזור החיים הקלאסי קובעים מועדים שבו כל איש צוות שעוסק באותו שלב מזמין אנשים אחרים שהם בדרגה וברמה שלו, מציג בפניהם מה שעשה מתוך מגמה שהם יגלו באגים.
   - Intergoup coordination
   - Software product engineering.
   - Training program: הכשרת האנשים <u>ב</u>תהליך.
   - Software process definition: קבוצה להגדרת התהליך.
   - Software process focus.

   <u>הבעיה:</u> יש קבוצה שלמה שעוסקת בתהליכים הנ"ל ← הרבה אנשים. מה עם חברות קטנות?
4. <u>Managed</u>: אם התהליך מוגדר אז יש קבוצה שמודדת את התהליך מהאספקט האיכותי: האם התהליך שהגדרנו מבטיח התאמה בין צוותים שונים.
5. <u>Optimizing</u>: הסקת מסקנות משלב 4 וכוונון הדברים בהתאם.

<u>חסרונות:</u>

1. כדי שחברה תמלא אחר כל הדרישות היא לא יכולה להיות חברה קטנה.
2. מי אומר מה הסדר שבו הדברים צריכים להתקיים?
3. המודל לא גמיש.
4. הרבה מאוד חברות תוכנה הן ברמה 1 ובכל זאת הן מצליחות לפתח תוכנה. כיצד ניתן להסביר את זה? חברה שמצליחה, מצליחה מפני שיש בה מס' אנשים שהם "כוכבים" שמצליחים להביא את הצוות להישגים. כל הגדרת התהליך באה להבטיח הצלחה גם אם אין "כוכבים".

7. Testing

<u>שלבים ב-testing:</u>

1. *unit testing*: בדיקת יחידה. בדיקות אלו נעשות ע"י זה שפיתח את היחידה. אחרי הבדיקה ה-unit עובר ל-baseline.
2. *integration testing*: לקיחת כל היחידות וחיבורן יחד. שיטות לחיבור:
   א. לחבר את כל היחידות יחד למע' אחת וזהו – שיטת ה-*<u>big-bang</u>*.
   ב. *<u>bottom up</u>*: כותבים driver זמני שמפעיל units שצריכים להתחבר יחד ל-unit שאינו קיים עוד ולאט-לאט עולים כלפי מעלה. המחיר: צריך לכתוב הרבה drivers.
   ג. *<u>top down</u>*: כותבים קודם כל את היחידה העליונה וכדי לבחון אותה כותבים stubs. כותבים עוד ועוד stubs עד שמגיעים לרמה התחתונה.
3. *system testing*: איחוד הכל.

יש חלוקה נוספת בהקשר של הצוות שמבצע את הבדיקה:
- הצוות הראשוני הוא האדם שכתב את ה-unit.
- צוות אבטחת איכות.

<u>בדיקות α:</u> בדיקות שמתבצעות בחברה עצמה. חשוב שאנשים שמבצעים בדיקות α לא יהיו אלו שכתבו את התוכנה.

<u>בדיקות β:</u> מתקינים את המע' אצל הלקוחות האמיתיים ונותנים להם להשתמש בזה.

<u>2 סוגי בדיקות:</u>

<u>1. White Box Testing:</u> מסתכלים על קוד התוכנה ובהתאם לכך בוחרים ערכים לקלט, להרצה. בדיקת נכונות הפונקציה.

bebugging: ניקח מע' תוכנה. נכניס, למשל, 100 באגים ונרשום איפה הכנסנו אותם. ניקח מתכנתים ונבקש מהם לגלות באגים. נניח שהם מצאו 200 באגים, כאשר 13 מתוכם הם אלו שנשתלו. אזי סה"כ מס' הבאגים הוא N, כך ש: $\frac{13}{200} = \frac{100}{N}$

שיטה זו אינה מדויקת: א. גם מתכנתים מנוסים עשויים לגלות בגים איפה שאין בגים.
ב. איך דואגים שה-bebugging יהיה הומוגני?

שיטת הכיסויים: שיטה לבחירת ערכי קלט ב-White box testing:
כיסוי פקודות: בחירת ערכי קלט כך שעל כל פקודה עוברים לפחות פעם אחת. [דורש הכי פחות]
כיסוי צלעות: בחירת ערכי קלט כך שעל כל צלע ב<u>גרף זרימת הבקרה</u> נעבור פעם אחת.
כיסוי תנאים: בתוך if בודקים לא רק את כל הביטוי של התנאי אלא את האמת והשקר של כל אחד מהמרכיבים. למשל:

if (a & b) then… else…

בכיסוי תנאים ניתן ל-a ול-b אפשרות להיות T/F: סה"כ 4 תנאים.
כיסוי מסלולים: עוברים על כל המסלולים האפשריים בתוכנית (כולל לולאות). [דורש הכי הרבה]

גם אם עוברים על כל התנאים זה לא מבטיח שלא יהיו בגים.

2. *<u>Black Box Testing</u>*: לא מסתכלים על מה שיש בפנים. רק בודקים לפי ערכי קלט ופלט (לפי הדרישות, המפרט). מסתכלים על ערכי הקלט המותרים בכלל. בוחרים קבוצה של ערכי קלט ומריצים. בד"כ ניתן לחלק את ערכי הקלט לקבוצות שבהן x יכול לקבל ערכים. נבחר x בכל אחד מהקבוצות. חלק מהבדיקות יעשות בקצוות המרווח, או בקצוות $\pm\varepsilon$. בדיקה מהצד של המשתמש.

<u>8. אבולוציית מערכות תוכנה: חוקי Lehman:</u>

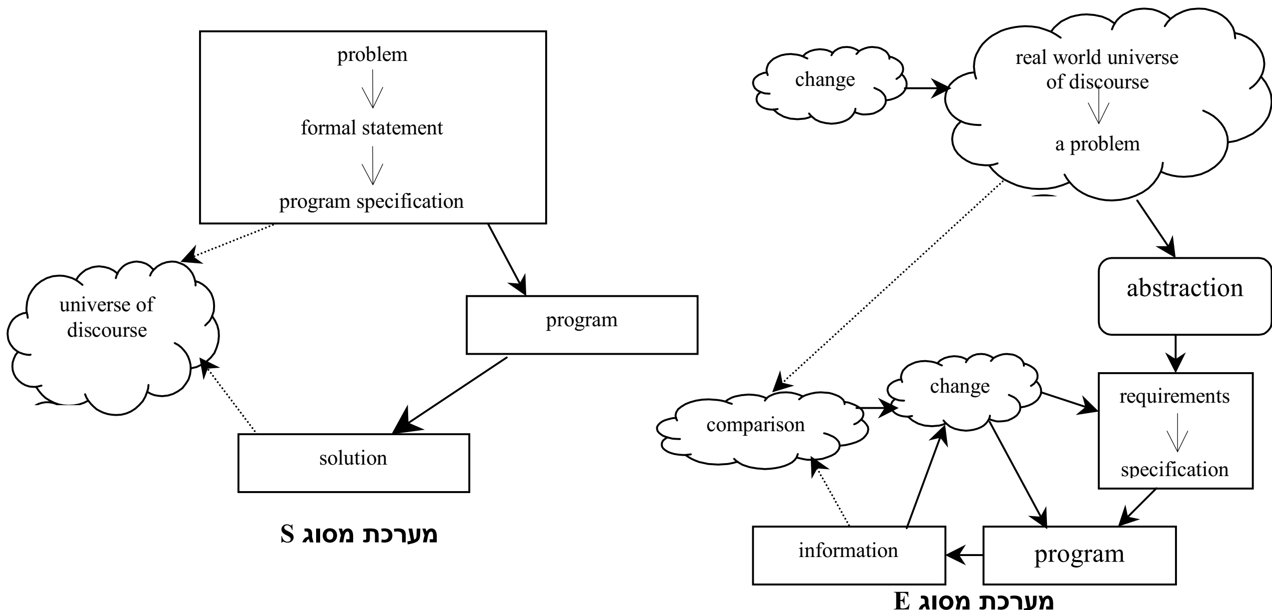
חוקי Lehman מהווים ניסיון למצוא חוקים שנותנים תיאוריה להנדסת תוכנה.

מערכות מסוג S:

מנסחים בעיה ← ניסוח פורמלי של הבעיה ← מפרט. על סמך המפרטים כותבים תוכנית שמהווה פתרון. יש איזשהו קשר בין הבעיה המקורית והפתרון לעולם האמיתי. הפתרון הוא תיאורטי ולכן הקשר מסומן בקווים מקווקווים (דוגמה: בעיית 8 המלכות). (מע' סטטיות).

מערכות מסוג E:

בעולם האמיתי ישנה בעיה. עקב הבעיה ← הפשטה ← דרישות ומפרט ← תוכניות ← מידע מהרצת התוכנית. עקב המידע ניתן לעשות השוואה עם העולם האמיתי. כאן יש קשר הדוק לעולם האמיתי, בעוד במערכות מסוג S אין. אחרי ההשוואה ניתן לעשות שינוי שגורם לשינוי בדרישות. יש לולאה עד שהפתרון שיתקבל יספק או יתאים לדרישות. גם העולם האמיתי משתנה ← כל העסק משתנה.
יש מעגל חיצוני: שינויים בעולם האמיתי, ומעגל פנימי: שינויים עקב הפעלת התוכנית (דוגמה: תוכנית שמשחקת מחשב: אין תוכנית שרצה בזמן נורמלי שתמיד תנצח ← עושים הנחות מסוימות. עקב שינויים בעולם האמיתי, שכלול מחשבים, נצטרך לעשות שינוי וכיו"ב).
<u>מערכת מסוג E למעשה אף פעם לא נגמרת</u>. בעיה מסוג E מורכבת מתת-בעיות מסוג S.



<u>מערכת תוכנה מסוג P:</u> יש בעיה שנוצרת בעולם האמיתי, על-סמך השקפות בעלי העניין בונים מודל וקובעים requirements and specifications ואז בונים מע' תוכנה שתופעל בעולם האמיתי.

1. ***חוק השינוי המתמיד:*** מערכת מסוג E שמשתמשים בה חייבת להשתנות כל הזמן, אחרת היא נהפכת פחות ופחות משביעת רצון.
2. ***חוק הסיבוכיות העולה:*** ככל שמע' התוכנה מתפתחת הסיבוכיות שלה עולה, אלא אם כן נעשית עבודה לשמור על הסיבוכיות או להקטין אותה.
3. ***חוק הוויסות העצמי:*** תהליך אבולוצית התוכנה מווסת את עצמו בהסתברות הקרובה לנורמלית למידות של המוצר והתהליך (← כל דבר שקשור לתהליך ולמוצר שמודדים הוא קבוע בכל המע').
4. ***חוק היציבות הארגונית:*** ממוצע קצב הפעילות הגלובלית של מערכת מתפתחת הוא קבוע לאורך זמן חיי המוצר.
5. ***חוק שמירת ההיכרות:*** במהלך זמן החיים של תוכנה מתפתחת, תוספת התוכן מגרסה לגרסה הוא באופן סטטיסטי קבוע.
6. ***חוק הצמיחה הקבועה:*** התוכן הפונקציונלי של המע' חייב לגדול באופן מתמיד על-מנת לשמור על שביעות רצון המשתמש לאורך זמן.
7. ***חוק האיכות היורדת:*** מערכות מסוג E ייתפסו כבעלות איכות יורדת, אלא אם כן יקפידו לתחזק אותן ולהתאים אותן לסביבת הפעלה משתנה.
8. ***חוק המשוב:*** תהליכי תוכנה של מערכות מסוג E מהווים מע' משוב רב-לולאות רב-רמות ויש להתייחס אליהן כך על-מנת להצליח לעדכן ו/או לשפר בצורה מוצלחת.

ניתן להסביר חלק מהחוקים עפ"י אינרציה, מומנטום ומשוב. מע' מסוג E הן מע' ענקיות אזי יש בהן אינרציה ומומנטום. לאחר העשייה יש משוב.