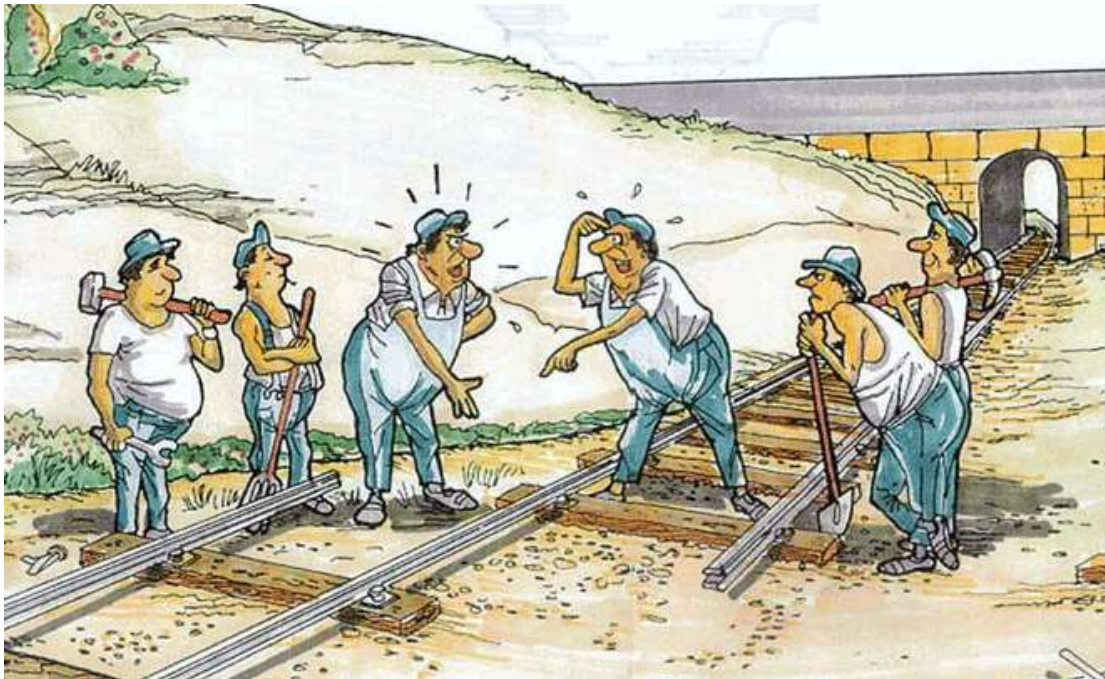


פרק 10

מימוש, שילוב ובדיקות



פעילות הקידוד ובדיקות היחידה

- מטרת הפעילות

- יצירת מרכיבי הקוד של התוכנה ובדיקתם

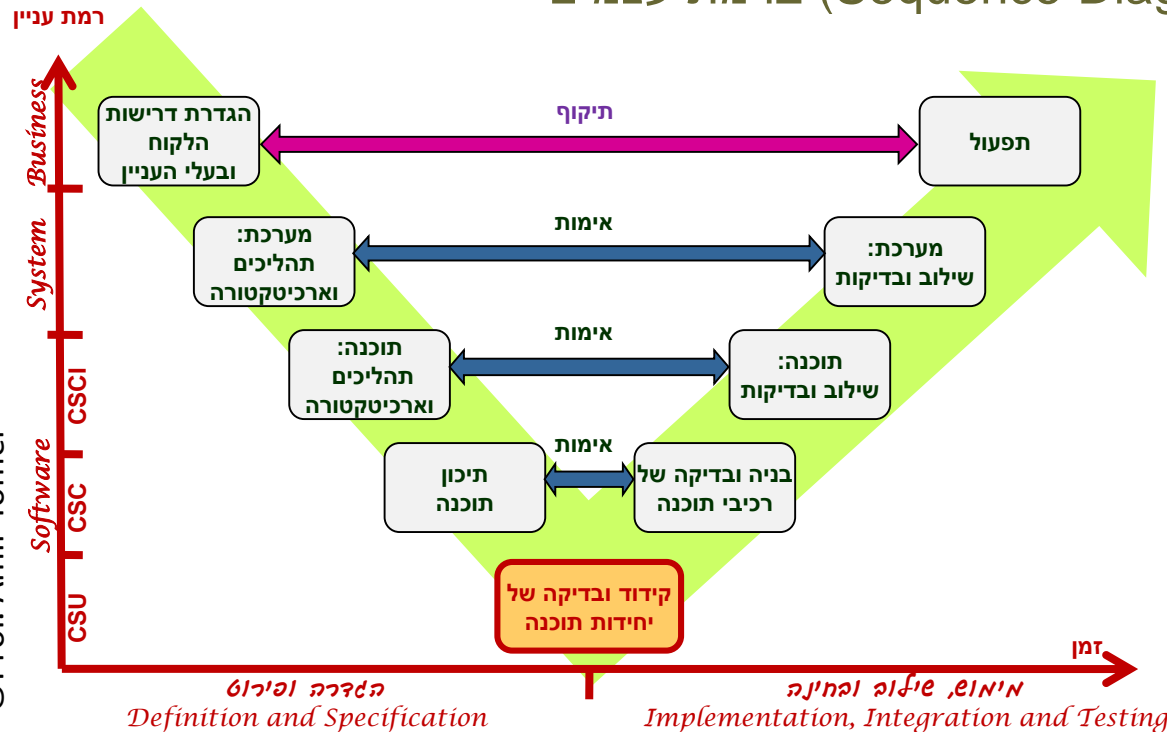
- קלט

- מודל מחלקות (Class Diagram)

- תהליכי תוכנה (Sequence Diagrams) ברמת עצמים

- תוצרים

- מודולי תוכנה בדוקים



עקרונות לקידוד נכון

- תוכנית מחשב נועדה לקריאה ע"י בני-אנוש

- שמות

- שמות משמעותיים ועקביים

- מבנה

- עריכה חזותית של הקוד

- כתיבה מדורגת (indented)

- שורות ריקות

- מבנים ברורים וברי-עקיבה

- מבני בקרה

- מבני נתונים

- הקטנת סיבוכיות הקוד

- לכידות (cohesion) וצימוד (coupling)

- הערות (comments)

- הבהרות של כל מה שלא מובן מאליו

שימוש נכון בשמות



- שמות משמעותיים

- קשר בין השם לבין מה שהוא מייצג

- שמות מלאים, אך ללא סירבול מיותר

```
The_DataBase_Capacity_Counter = The_DataBase_Capacity_Counter + 1;
```

- חד משמעיות

- מה מייצג המשתנה `?fstpt`

first point? (`first_point`, `frstpnt`, `point1`)

file start pointer? (`file_start_ptr`)

fast prototype? (`fastPT`)

- שמירה על עקביות

- דוגמה גרועה: ייצוג המונח תדירות (`frequency`) באופנים שונים

`freqAverage`, `frequencyMaximum`, `minFr`, `frqncyTotl`

קוד המתעד את עצמו (self documenting code)*

- מה עושה התוכנית הבאה*

```
for ( i = 2; i <= num; i ++ ) {  
meetscriteria[ i ] = true;  
}  
for ( i = 2; i <= num / 2; i++ ) {  
j = i + i;  
while ( j <= num ) {  
meetscriteria[ j ] = false;  
i = j + i;  
}  
}  
for ( i = 2; i <= num; i ++ ) {  
if ( meetscriteria[ i ] ) {  
system.out.println ( i + " meets criteria." );  
}  
}
```

* Steve McConnell, Code Complete, 2nd Edition, Microsoft, 2004

אותה תוכנית בסגנון של self-documenting code

```
For ( primecandidate = 2; primecandidate <= num; primecandidate++ ) {
    isPrime[ primecandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
    int factorableNumber = factor + factor;
    while ( factorableNumber <= num ) {
        isprime[ factorableNumber ] = false;
        factorableNumber = factorableNumber + factor;
    }
}

for ( primecandidate = 2; primecandidate <= num; primecandidate++ ) {
    if ( isprime[ primecandidate ] ) {
        system.out.println( primecandidate + " is prime." );
    }
}
```

CHECKLIST: Self-Documenting Code Classes (1)

- **Classes**

- Does the class's interface present a consistent abstraction?
- Is the class well named, and does its name describe its central purpose?
- Does the class's interface make obvious how you should use the class?
- Is the class's interface abstract enough that you don't have to think about how its services are implemented?
 - Can you treat the class as a black box?

- **Routines**

- Does each routine's name describe exactly what the routine does?
- Does each routine perform one well-defined task?
- Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- Is each routine's interface obvious and clear?

CHECKLIST: Self-Documenting Code Classes (2)

- **Data Names**

- Are type names descriptive enough to help document data declarations?
- Are variables named well?
- Are variables used only for the purpose for which they're named?
- Are loop counters given more informative names than i, j, and k?
- Are well-named enumerated types used instead of makeshift flags or boolean variables?
- Are named constants used instead of magic numbers or magic strings?
- Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

- **Data Organization**

- Are extra variables used for clarity when needed?
- Are references to variables close together?
- Are data types simple so that they minimize complexity?
- Is complicated data accessed through abstract access routines (abstract data types)?

- **Control**

- Is the nominal path through the code clear?
- Are related statements grouped together?

תיעוד בגוף הקוד (comments) – סוגי הערות

✗ חזרה על הקוד

- מביאות יותר טרחה מאשר תועלת
- ```
X = X+1; /* add 1 to X */
```

## ✓ הסבר של הקוד

- נועדו להסביר קוד מתוחכם, מסובך או רגיש
- קוד שקשה להסביר עדיף לשנות מאשר לתעד

## ✓ סימונים (זמניים) בתוך הקוד

- תזכורות שיש לשים לב אליהן, אך להסירן בשלב מאוחר יותר
  - כתובות בצורה "מאירת עיניים"
- ```
return NULL; // ***** NOT DONE! FIX BEFORE RELEASE
```

✓ סיכום / תקציר של הקוד

- מספר שורות קצרות

✓ תאור כוונת הקוד

- הסבר ברמת הבעיה, לא ברמת הפתרון
- ```
-update employeeRecord object ולא - get current employee information
```

## ✓ מידע שלא ניתן לביטוי ברור על ידי הקוד עצמו

- דקויות ההבנה הקיימות בזמן הכתיבה, לא יהיו מובנות מאליהן בשלבים מאוחרים יותר, או ע"י אנשים אחרים!

# תחזוקת התיעוד

- גם התיעוד בגוף הקוד עומד להשתנות עם שינויי הקוד, ולכן הוא צריך להיות כתוב בצורה נוחה לתחזוקה
  - קוד קשה לתחזוקה

```
/******
* class: GigaTron (GIGATRON.CPP) *
* author: Dwight K. coder *
* date: July 4, 2014 *
* *
* Routines to control the twenty-first century's code evaluation *
* tool. The entry point to these routines is the Evaluatecode() *
* routine at the bottom of this file. *
*****/
```

– קוד קל לתחזוקה

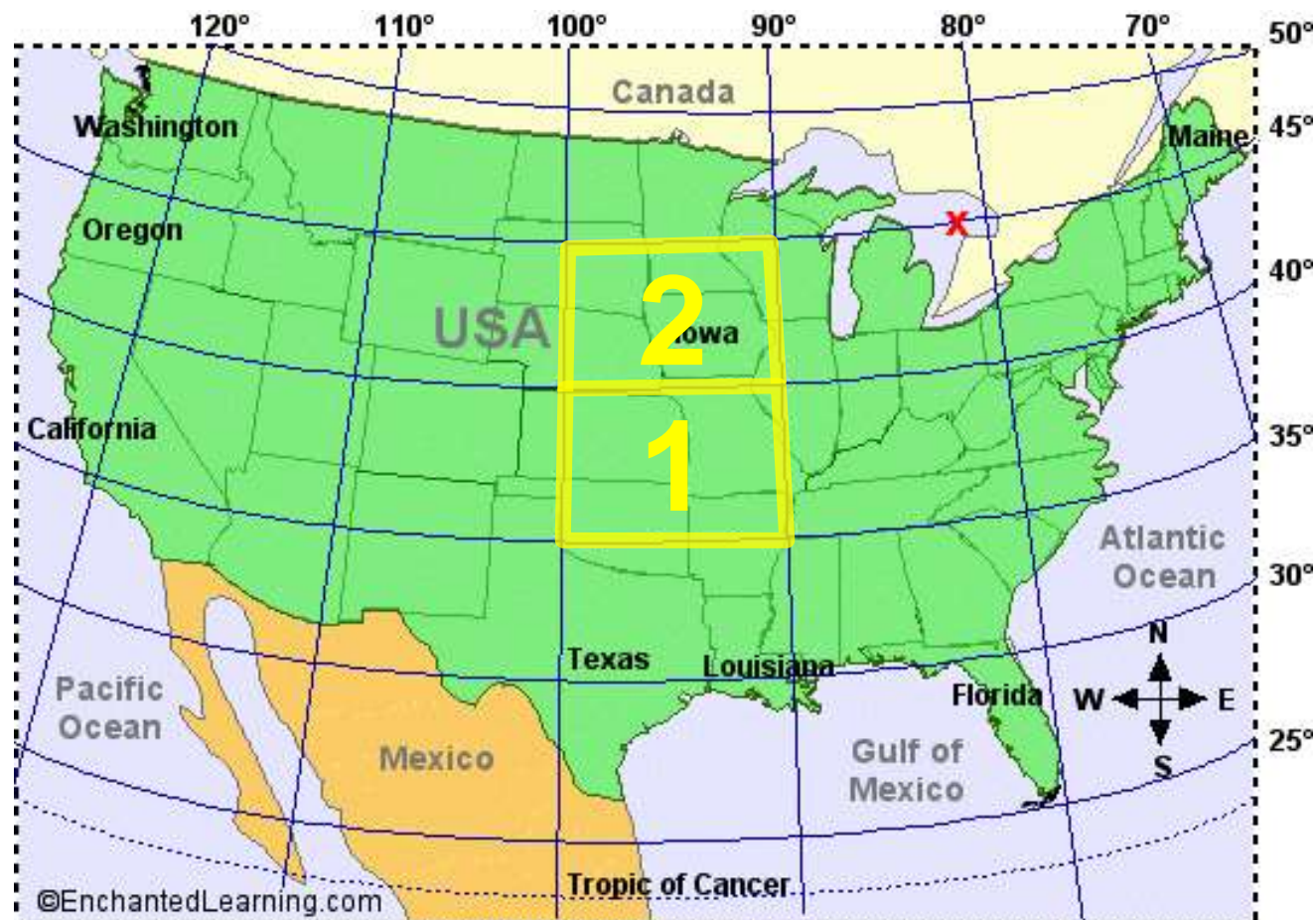
```
/******
class: GigaTron (GIGATRON.CPP)

author: Dwight K. coder
date: July 4, 2014


Routines to control the twenty-first century's code evaluation
tool. The entry point to these routines is the Evaluatecode()
routine at the bottom of this file.
*****/
```

## משפטי תנאי ברורים

- האם קואורדינטה  $[X, Y]$  נמצאת באיזור המסומן (והיכן)?




## משפטי תנאי ברורים - המשך



```
if (latitude > 35 && longitude > 90)
{
 if (latitude <= 40 && longitude <= 100)
 mapSquareNo = 1;
 else if (latitude <= 45 && longitude <= 100)
 mapSquareNo = 2;
 else
 System.out.println("Not on the map");
}
else
 System.out.println("Not on the map");
```

```
if (longitude > 90) && longitude <= 100 &&
 latitude > 35 && latitude <= 40)
 mapSquareNo = 1;
else if (longitude > 90 & longitude <= 100 &&
 latitude > 40 && latitude <= 45)
 mapSquareNo = 2;
else
 System.out.println("Not on the map");
```



## משפטי תנאי ברורים - עקרונות

- משפטי תנאי מקוננים - קשים לקריאה

if-if -

if-else-if -

- בדרך כלל יש שקילות בין שני המבנים:

```
if <condition1>
```

```
 if <condition2>
```

```
if <condition1> && <condition2>
```

- כלל אצבע:

– יש להמנע ממשפטי תנאי מקוננים בעומק של יותר מ-3 רמות.

# CHECKLIST: Using Conditionals\*

- **if-then Statements**

- Is the nominal path through the code clear?
- Do *if-then* tests branch correctly on equality?
- Is the *else* clause present and documented?
- Is the *else* clause correct?
- Are the *if* and *else* clauses used correctly-not reversed?
- Does the normal case follow the *if* rather than the *else*?

- **if-then-else-if Chains**

- Are complicated tests encapsulated in boolean function calls?
- Are the most common cases tested first?
- Are all cases covered?
- Is the *if-then-else-if* chain the best implementation-better than a *case* statement?

- **case Statements**

- Are cases ordered meaningfully?
- Are the actions for each case simple-calling other routines if necessary?
- Does the *case* statement test a real variable, not a phony one that's made up solely to use and abuse the *case* statement?
- Is the use of the default clause legitimate?
- Is the default clause used to detect and report unexpected cases?
- In C, C++, or Java, does the end of each case have a *break*?

---

\* Steve McConnell, *Code Complete*, 2<sup>nd</sup> Edition, Microsoft, 2004

# רמות בדיקה של מערכת/מוצר תוכנה

## • במהלך מחזור החיים קיימות 3 רמות בדיקה עקרוניות

### – בדיקות יחידה

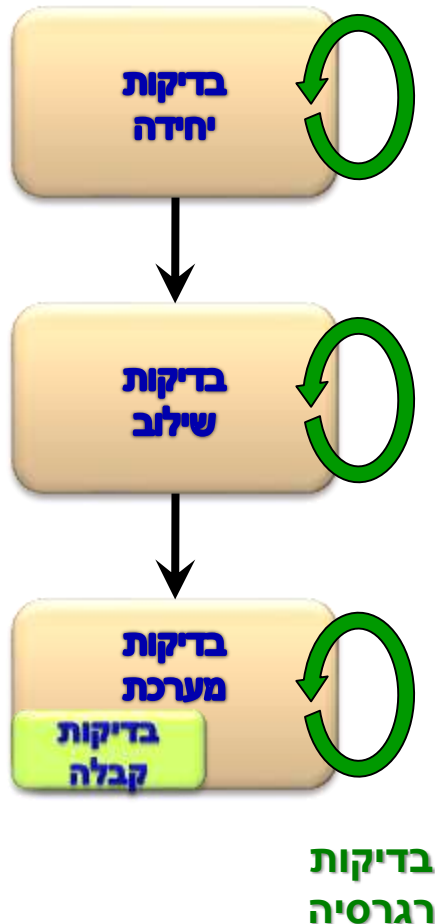
- מתבצעות לכל יחידה בנפרד במקביל לקידוד
- יש לחזור עליהן בכל פעם שהיחידה השתנתה

### – בדיקות שילוב (אינטגרציה)

- מתבצעות לכל תת-קבוצה של פריטים לאחר שילובם
- כוללות חזרה על בדיקות של רמת שילוב קודמת, לוודא שהשילוב החדש לא "קלקל" את מה שעבד קודם

### – בדיקות מערכת / מוצר

- סוגים שונים של בדיקות (פירוט בהמשך) למוצר הסופי
- יש לחזור על חלק מהן בכל פעם שחלו שינויים במוצר



- הגדרה (ד. גלין)

– בדיקות תוכנה הוא תהליך **פורמאלי**, המבוצע בידי **צוות בדיקות מומחה**, אשר במהלכו יחידת תוכנה, מספר יחידות תוכנה משולבות או מערכת תוכנה שלמה נבדקות באמצעות **הרצת התוכנה על גבי מחשב**. כל הבדיקות מבוצעות על פי נוהלי בדיקה (test procedures) **מאושרים** מעל מקרי בדיקה (test cases) **מאושרים**.

- מטרות הבדיקות

– מטרות ישירות

- לזהות ולחשוף מספר רב ככל האפשר של שגיאות בתוכנה הנבדקת
- להביא את התוכנה הנבדקת, לאחר תיקון השגיאות המזהות ובדיקה חוזרת, לרמת איכות קבילה
- לבצע את הבדיקות הנדרשות ביעילות ובאפקטיביות, בגבולות הזמן והתקציב

– מטרה עקיפה

- לאסוף רישום של שגיאות תוכנה לצורך שימוש במניעת שגיאות עתידיות (באמצעות פעולות מתקנות ופעולות מונעות)



# דירוג שגיאות תוכנה ("באגים") על פי חומרתן

| חומרה      | תיאור                                                                                                                                                                               |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5 (קריטית) | (1) מונעת ביצוע של יכולות חיוניות<br>(2) מסכנת את הבטיחות, הבטחון או דרישות קריטיות אחרות                                                                                           |
| 4          | (1) משפיעה לרעה על ביצוע פעולות חיוניות<br>(2) משפיעה לרעה על סיכוני עלות, לו"ז או סיכונים טכניים של הפרויקט, או של אחזקת המערכת, כאשר לא ידוע פתרון העוקף את הבעיה (work-around)   |
| 3          | (1) משפיעה לרעה על יכולות ביצוע פעולות חיוניות, כאשר ידוע פתרון עוקף<br>(2) משפיעה לרעה על סיכוני עלות, לו"ז או סיכונים טכניים של הפרויקט, או של אחזקת המערכת, כאשר ידוע פתרון עוקף |
| 2          | (1) גורמת אי-נוחות למשתמש/למפעיל, אשר אינה משפיעה על יכולות חיוניות של המשימה או התפעול<br>(2) גורמת אי-נוחות לצוות הפיתוח או האחזקה, אך אינה מונעת מהם לממש את אחריותם             |
| 1 (מזערית) | משפיעה בכל צורה אחרת.                                                                                                                                                               |

# 7 העקרונות של Meyer לבדיקות תוכנה

**Bertrand Meyer,  
Seven Principles of Software Testing,  
IEEE Computer, August 2008**

## 1. הגדרה

- לבדוק תוכנית זה לגרום לה להיכשל

## 2. בדיקות לעומת מפרטים

- בדיקות אינן תחליף למפרטים

## • בדיקות רגרסיה

- כל ביצוע שנכשל חייב להניב מקרה-בדיקה (test case), אשר יישאר כחלק קבוע מחבילת הבדיקות של הפרויקט

## • תוצאות הבדיקות הן "אורקלים" (test oracles)

- קביעת ההצלחה או הכשלון של בדיקה חייב להיות תהליך אוטומטי (אורקל)

## • מקרי בדיקה ידניים ואוטומטיים

- תהליך בדיקה אפקטיבי חייב לכלול הן בדיקות ידניות והן בדיקות אוטומטיות

## • הערכה אמפירית של אסטרטגיות הבדיקה

- הערך כל אסטרטגיית בדיקה, אטרקטיבית ככל שתהיה, באמצעות הערכה אובייקטיבית תוך שימוש בקריטריונים מפורשים בתהליך בדיקה בר-חזרה

## • קריטריוני הערכה

- התכונה החשובה ביותר של אסטרטגיית בדיקה היא מספר השגיאות שהיא מצליחה לגלות כפונקציה של הזמן.

# בדיקתיות (Testability)

- הגדרה

– המידה בה מערכת או רכיב מאפשרים את ההגדרה של קריטריוני בדיקה ואת הביצוע של בדיקות אשר יקבעו האם קריטריונים אלה הושגו.

- שתי משמעויות (קשורות אך שונות)

– מה הסיכוי שהתוכנה תחשוף תקלה במהלך בדיקות, אם התקלה קיימת  
– עד כמה ניתן בקלות לעמוד בקריטריון כיסוי\* במלואו

- \*קריטריוני כיסוי

– כיסוי פונקציונלי – מעבר דרך כל הפונקציות בתוכנה  
– כיסוי משפטים – מעבר דרך כל משפטי התוכנה  
– כיסוי תנאים – מעבר דרך כל צמתי ההחלטה וההסתעפויות שבתוכנה  
– כיסוי מסלולים – מעבר דרך כל המסלולים האפשריים בתוכנה  
– כיסוי כניסות/יציאות – מעבר דרך כל הקריאות והחזרות בתוכנה

# כללים להבטחת בדיקות התוכנה\* (1)

*“The better we can control the software, the more the testing can be automated and optimized”*

- יכולת בקרה (Controlability)

- קיים ממשק בו ניתן להגדיר תרחישי בדיקה, או אמצעי להפעלת בדיקות

- debugger

- כלי בדיקה מסחרי

- מצבי התוכנה והחומרה וכן משתני המערכת ניתנים לשליטה באופן ישיר ע"י מהנדס הבדיקות

- אובייקטים, מודולים או שכבות פונקציונליות ניתנים לבדיקה באופן בלתי-תלוי

---

\* J. Bach, Heuristics of Software Testability, 2003

## כללים להבטחת בדיקתיות התוכנה (2)

*“What you see is what can be tested”*

### • שקיפות (Observability)

- מצבי-עבר וערכים היסטוריים של משתני מערכת הינם גלויים או בני-תשאול (queriable)
  - כגון transaction logs
- פלט שונה מיוצר עבור כל קלט
- מצבים ומשתני מערכת הינם גלויים או בני-תשאול תוך כדי ריצה
- כל הגורמים המשפיעים על הפלט הינם גלויים
- פלט לא תקין - מזוהה בקלות
- שגיאות פנימיות מתגלות באופן אוטומטי ומדווחות ע”י מנגנוני בדיקה עצמית

## כללים להבטחת בדיקתיות התוכנה (3)

*“To test it, we have to get at it”*

### • זמינות (Availability)

- בתוכנה קיימים כמה באגים
  - באגים מוסיפים תקורה של ניתוח ודיווח לתהליך הבדיקות
- באגים אינם מפריעים להרצת בדיקות
- המוצר מתפתח בשלבים פונקציונליים
  - מאפשר פיתוח ובדיקה באופן סימולטני
  - חלקים שכבר נבדקו יכולים לשמש לבדיקת חלקים אחרים
- קיימת נגישות לקוד המקור

## כללים להבטחת בדיקות התוכנה (4)

*“The simpler it is, the less there is to test”*

- פשטות (Simplicity)

- התכן שומר על עקביות עצמית

- פשטות פונקציונלית

- למשל, כמות ה-features הינה המינימום הנדרש כדי לענות על הדרישות

- פשטות מבנית

- למשל, המודולים הם בעלי לכידות גבוהה וצימוד רופף

- פשטות הקוד

- למשל, הקוד איננו מפותל עד כדי כך שקורא חיצוני אינו יכול לסקור אותו בצורה אפקטיבית

## כללים להבטחת בדיקות התוכנה (5)

*“The fewer the changes, the fewer the disruptions to testing”*

### • יציבות (Stability)

- שינויים לתוכנה אינם שכיחים
- שינויים לתוכנה מבוקרים ומפורסמים
- שינויים לתוכנה אינם הופכים בדיקות אוטומטיות לבלתי-תקפות



## כללים להבטחת בדיקות התוכנה (6)

*“The more information we have, the smarter we will test”*

### • מידע (Information)

- התכן דומה למוצרים קודמים שאנו מכירים
- הטכנולוגיה עליה מבוסס המוצר מובנת היטב
- התלויות בין רכיבים חיצוניים, פנימיים ומשותפים - מובנות היטב
- מטרת התוכנה מובנת היטב
- משתמשי התוכנה מובנים היטב
- הסביבה בה ישתמשו בתוכנה מובנת היטב
- התיעוד טכני נגיש, מדויק, מאורגן היטב, ספציפי ומפורט
- דרישות התוכנה מובנות היטב

# בדיקות יחידה

- unit

– חלק קוד שיכול לעבור קומפילציה בנפרד וניתן לבדיקה עצמאית

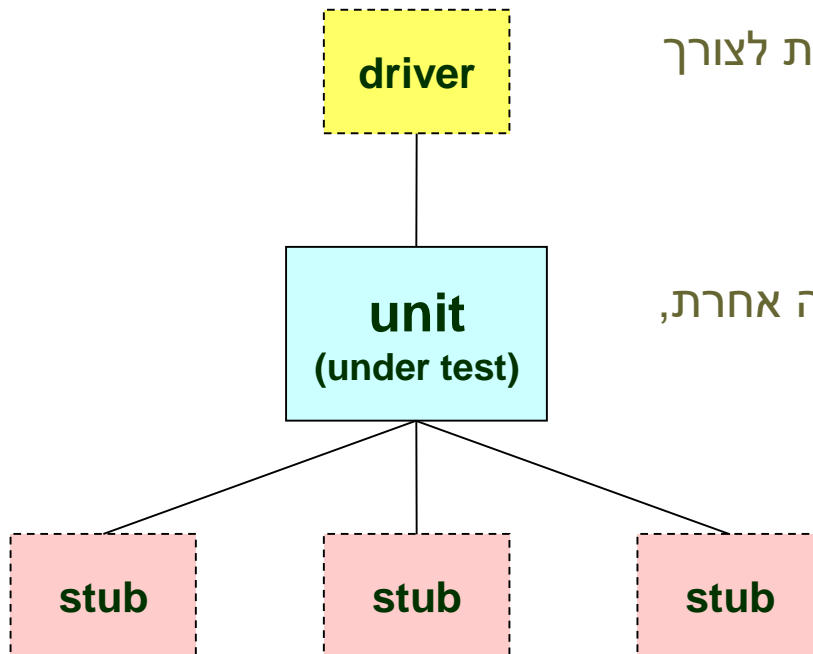
- בדרך כלל נכתב ונבדק ע"י תכנת בודד

- driver

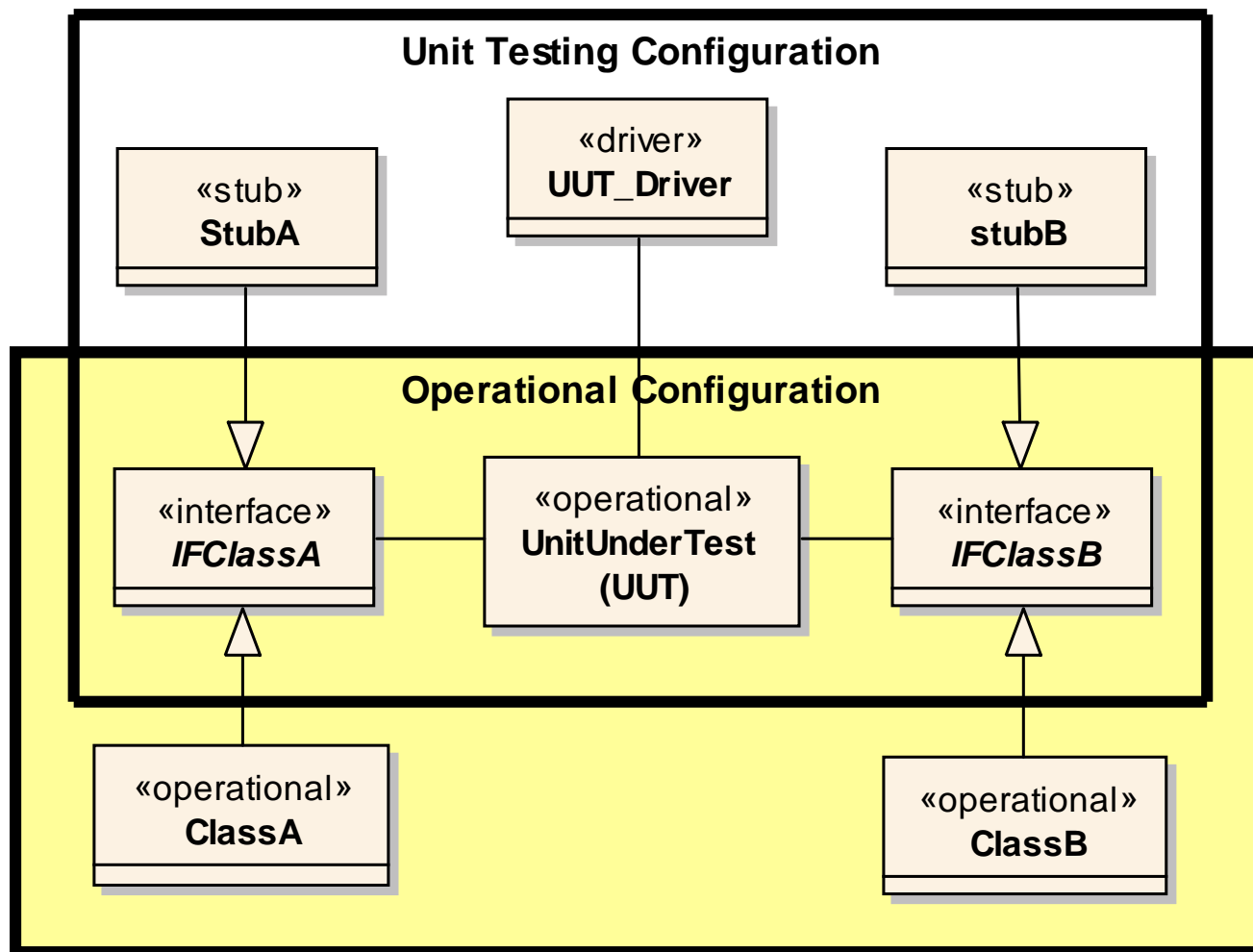
– יחידת "דמה" המפעילה יחידה אחרת לצורך בדיקת היחידה **המופעלת**

- stub

– יחידת "דמה" המופעלת מתוך יחידה אחרת, לצורך בדיקת היחידה **המפעילה**



# סביבת בדיקות יחידה למחלקה



# שיטות לבדיקת יחידה



- קופסה שחורה (black box)

– בחינת התפקוד של היחידה בתוך המערכת

- נכונות הפלט / התגובה

- ערכים "חוקיים"

- ערכים "לא חוקיים"

- מהירות התגובה

- קופסה לבנה (white box, glass box)

– בחינת המבנה הפנימי של היחידה

- מסלולי החישוב

- נכונות החישובים

- נכונות ההחלטות הלוגיות

בכל אחת מהשיטות יש להכין  
קובץ נתוני בדיקה (test data)  
לכיסוי כל המקרים השונים

# בדיקות יחידה – קופסה שחורה

## בדיקות כיסוי לפונקציה $f(X_1, \dots, X_n)$

- נתוני הבדיקה צריכים לייצג את כל מחלקות-השקילות (equivalence classes) של ערכי הארגומנטים

– עבור כל ארגומנט  $X$  :

- אם היחידה אמורה להגיב באופן מוגדר לערכי  $X$  בתחום נתון  $[L, U]$  אזי יש לבדוק את  $f$  עם חמישה ערכים שונים של  $X$ :

–  $X > U$  ,  $X = U$  ,  $L < X < U$  ,  $X = L$  ,  $X < L$

- לדוגמה

– הפונקציה  $\text{power}(\text{Range}, \text{Speed})$  מחשבת עוצמה (בסולם HML) כתלות בטווח ובמהירות, על פי הטבלה הבאה:

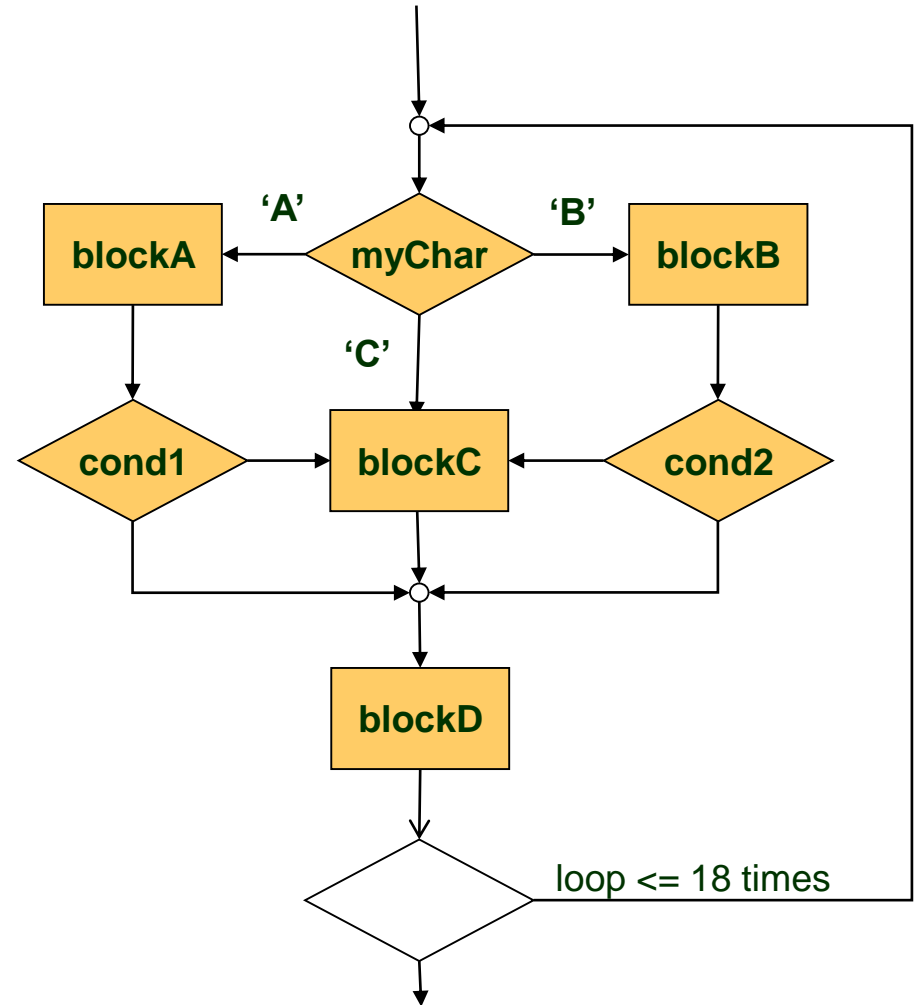
| Speed \ Range         | Range   |                   |         |
|-----------------------|---------|-------------------|---------|
|                       | $R < 1$ | $1 \leq R \leq 5$ | $R > 5$ |
| $S < 100$             | L       | L                 | M       |
| $100 \leq S \leq 200$ | L       | M                 | H       |
| $S > 200$             | M       | H                 | H       |

## בדיקות יחידה – וקטור בדיקות לפונקציה

| R   | S   | Expected Result |
|-----|-----|-----------------|
| 0.5 | 50  | L               |
| 0.5 | 100 | L               |
| 0.5 | 150 | L               |
| 0.5 | 200 | L               |
| 0.5 | 250 | M               |
| 1   | 50  | L               |
| 1   | 100 | M               |
| 1   | 150 | M               |
| 1   | 200 | M               |
| 1   | 250 | H               |
| ... |     |                 |

## בדיקות יחידה – קופסה לבנה

```
read (kmax) //1 <= kmax <= 18
for (k=0; k < kmax; k++) do
{
 read(myChar)
 switch (myChar)
 {
 case 'A':
 blockA;
 if (cond1) blockC;
 break;
 case 'B':
 blockB;
 if (cond2) blockC;
 break;
 case 'C':
 blockC;
 break;
 }
 blockD
}
```



$$5^1 + 5^2 + 5^3 + \dots + 5^{18} = 4.77 \times 10^{12}$$

• מספר המעברים האפשריים:

# בדיקת מסלולי-בסיס (Base Path testing)

(McCabe, 1976)

- שיטה קלאסית לבדיקת "קופסה לבנה"
  - כיסוי כל המעברים האפשריים בתוך היחידה
- בניית גרף זרימה (flow graph) של היחידה
  - צומת - יחידת חישוב ללא הסתעפות
  - קשת - הסתעפות בחישוב
- מסלול בלתי-תלוי
  - מסלול מהתחלה לסיום, הכולל לפחות צומת אחד חדש, לעומת מסלולים אחרים



# בדיקת מסלולי-בסיס - דוגמה

## a procedure in PDL

```
1: do while records remain
 read record;
2: if record field 1 = 0
3: then process record;
 store in buffer;
 increment counter;
 else
4: if record field 2 = 0
5: then reset counter;
6: else process record;
 store in file;
7: endif
 endif
8: enddo
9: end
```

## base paths

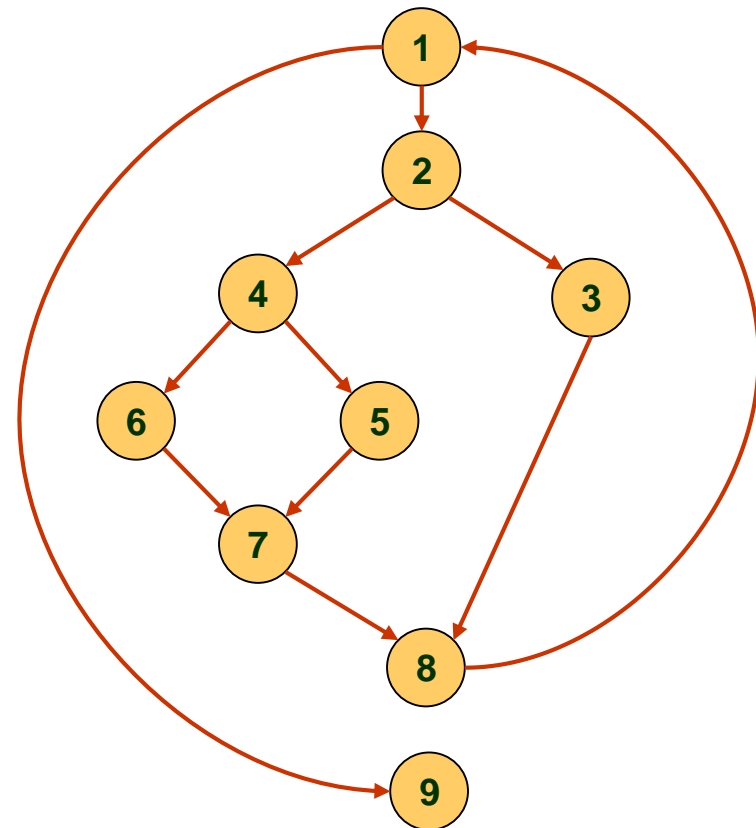
path 1: 1 - 9

path 2: 1 - 2 - 3 - 8 - 1 - 9

path 3: 1 - 2 - 4 - 5 - 7 - 8 - 1 - 9

path 4: 1 - 2 - 4 - 6 - 7 - 8 - 1 - 9

## flow graph



## V(G) - הסיבוכיות הציקלומטית (Cyclomatic Complexity) של הגרף G

### • מספר מסלולי הבסיס הוא סופי (וקטן!)

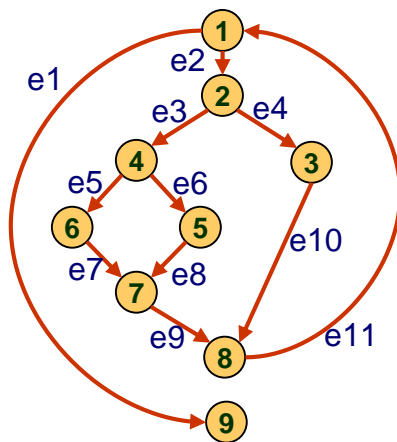
– המספר המקסימלי של מסלולים בלתי תלויים

$$V(G) = E - N + 2 \quad (E = \text{קשתות}, N = \text{צמתים})$$

$$= P + 1 \quad (P = \text{מספר הצמתים מהן יוצאת יותר מקשת אחת}^*)$$

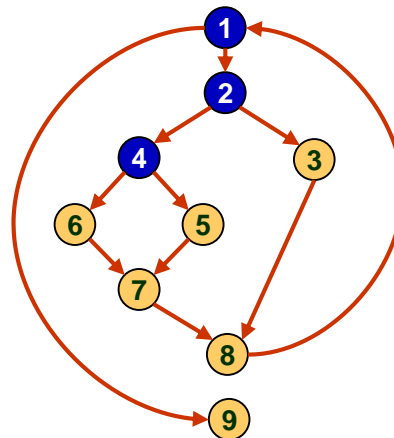
\* לגרפים עם דרגת יציאה 2 לכל היותר

$$= R \quad (R = \text{מספר האיזורים הסגורים} + \text{האיזור "הפתוח"})$$



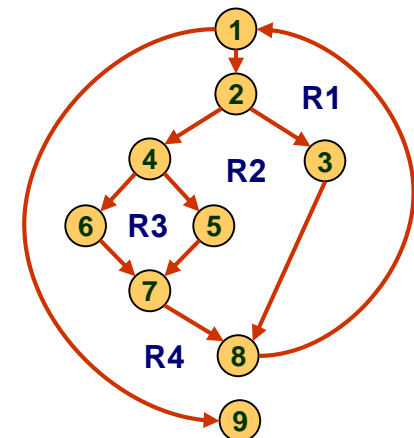
$$11 - 9 + 2$$

=



$$3+1$$

=



$$4$$

# פעילות השילובים והבדיקות של התוכנה

- מטרת הפעילות

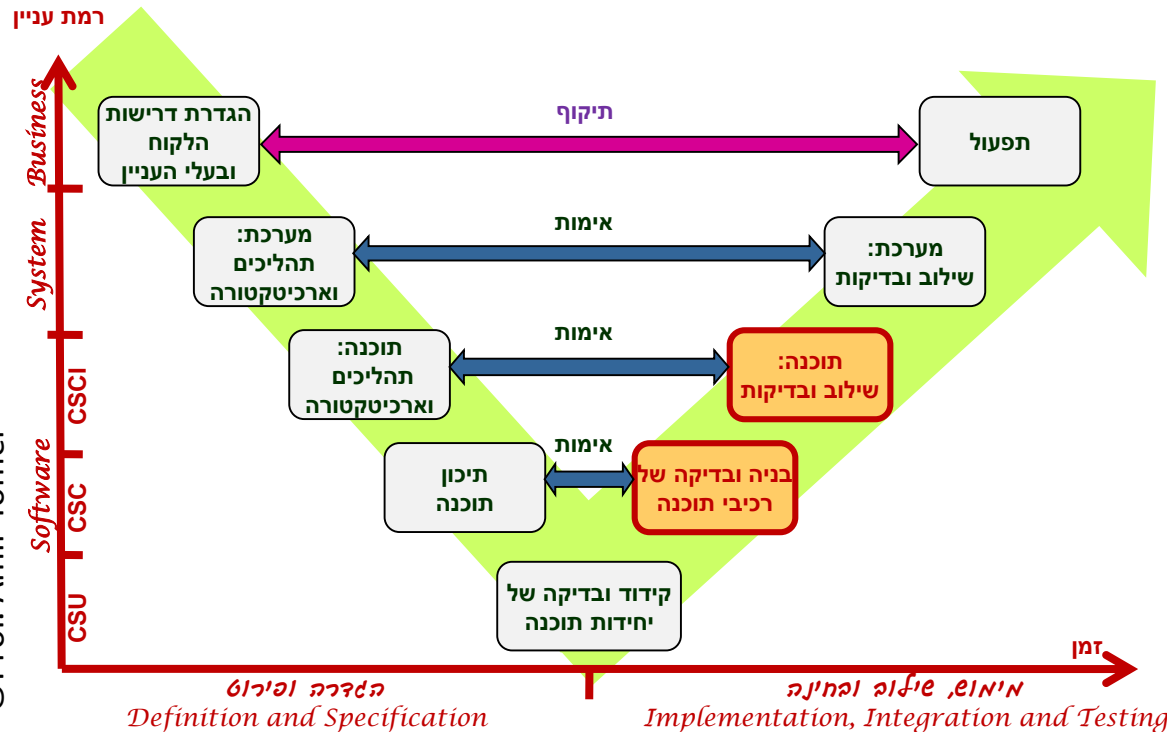
– יצירת פריטים (יישומים) עובדים ובדוקים של תוכנה

- קלט

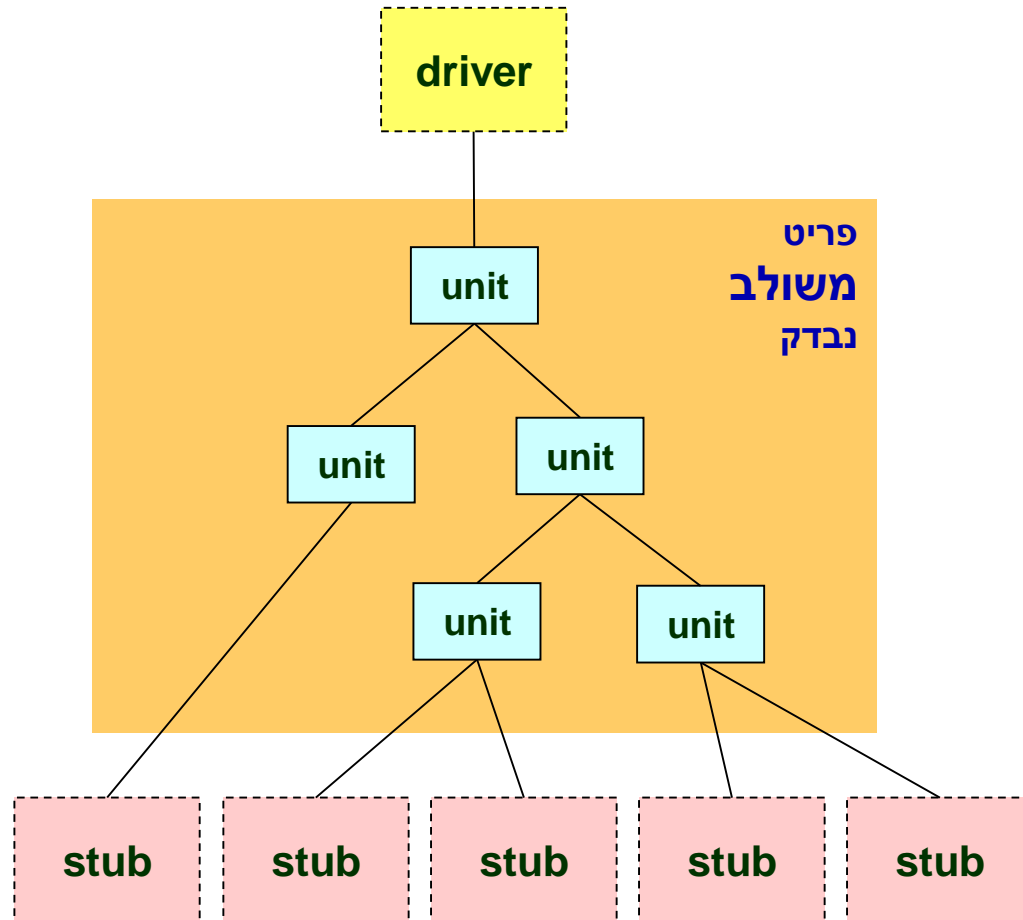
– יחידות תוכנה בדוקות

- תוצרים

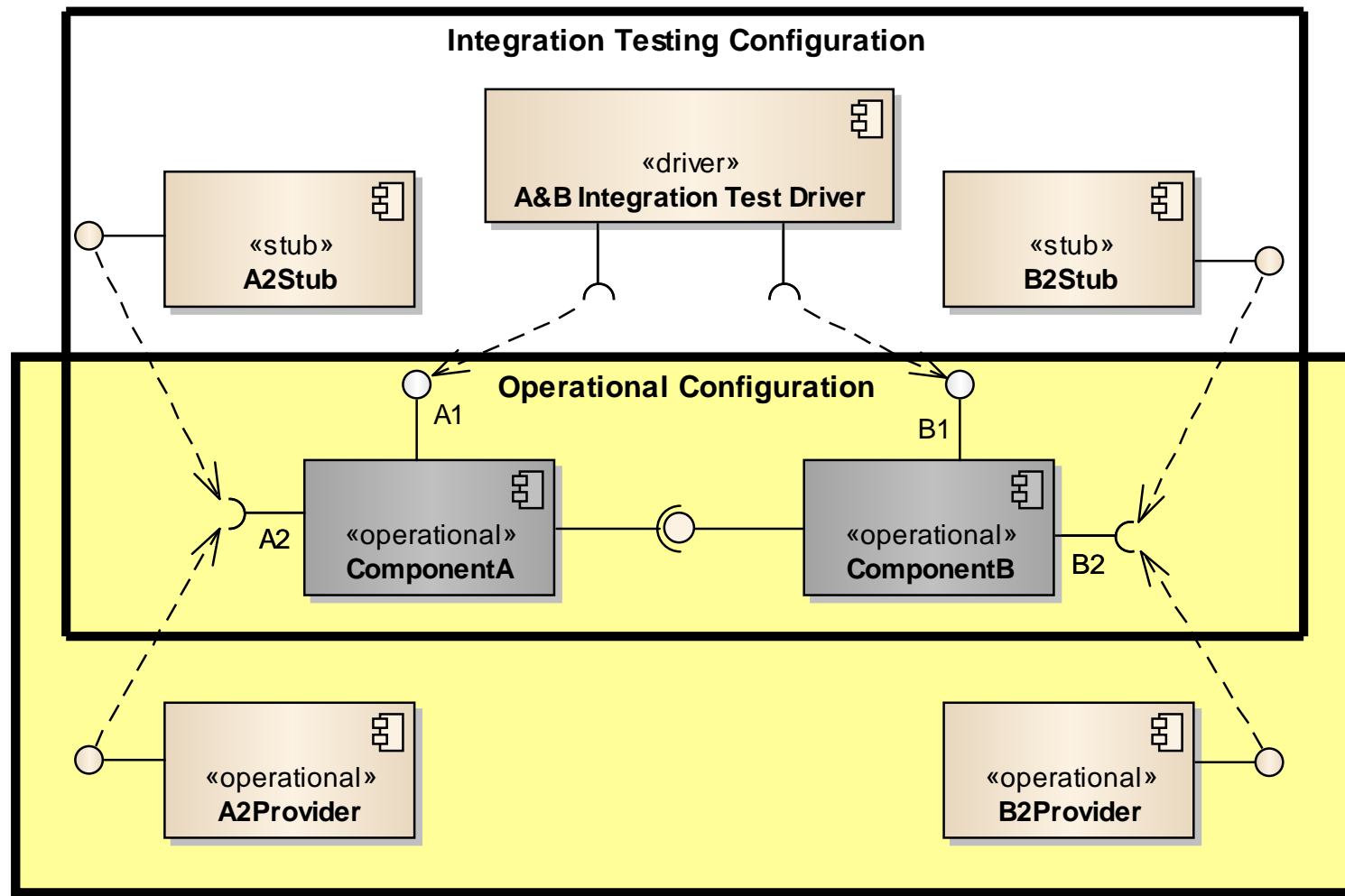
– פריטי תוכנה בדוקים



# סביבת בדיקה לפריט משולב

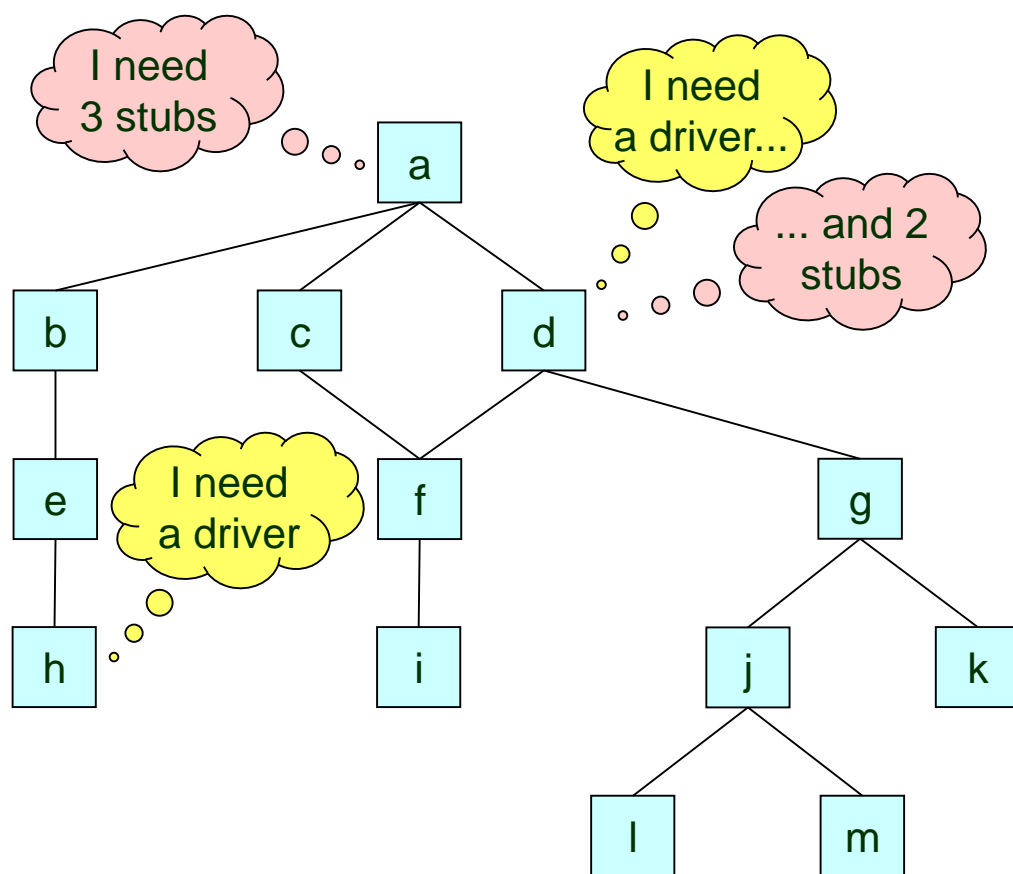


# סביבת בדיקות שילוב (לרכיבים) – יצוג UML



# מימוש ושילוב – גישת "המפץ הגדול"

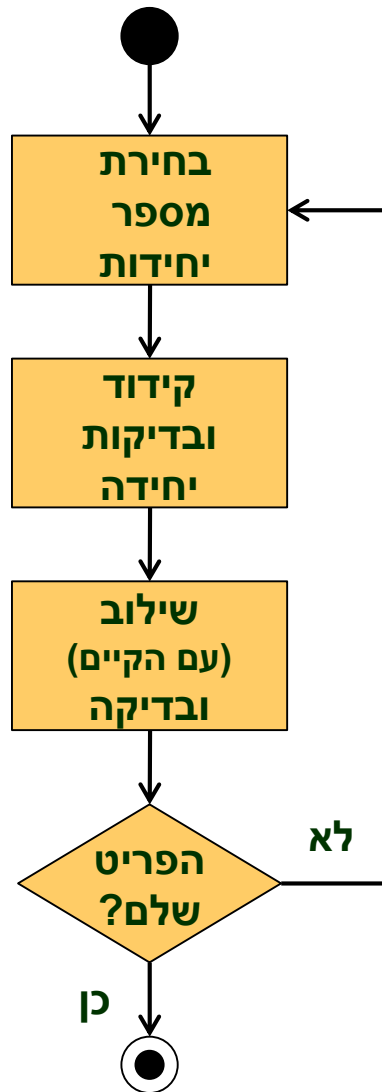
- שלב א': קידוד ובדיקה של כל יחידה בנפרד
- שלב ב': קישור כל 13 היחידות ובדיקת המוצר השלם



השקעה בסביבת בדיקות:

drivers21  
13 stubs

# מימוש ושילוב בסבבים



## • בשיטת "המפץ הגדול" קיימות שתי בעיות עיקריות:

- בעיה מס' 1
  - יש לכתוב stubs ו-drivers ולהשליך אותם לאחר מכן

- בעיה מס' 2
  - קושי לבודד תקלות
  - התקלה יכולה להימצא בכל מקום

– 13 יחידות

– 13 ממשקים

## • הפתרון לשתי הבעיות

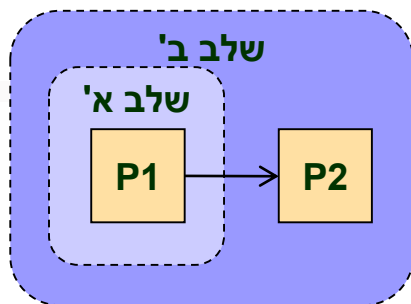
- מימוש ושילוב בסבבים

## • השאלות:

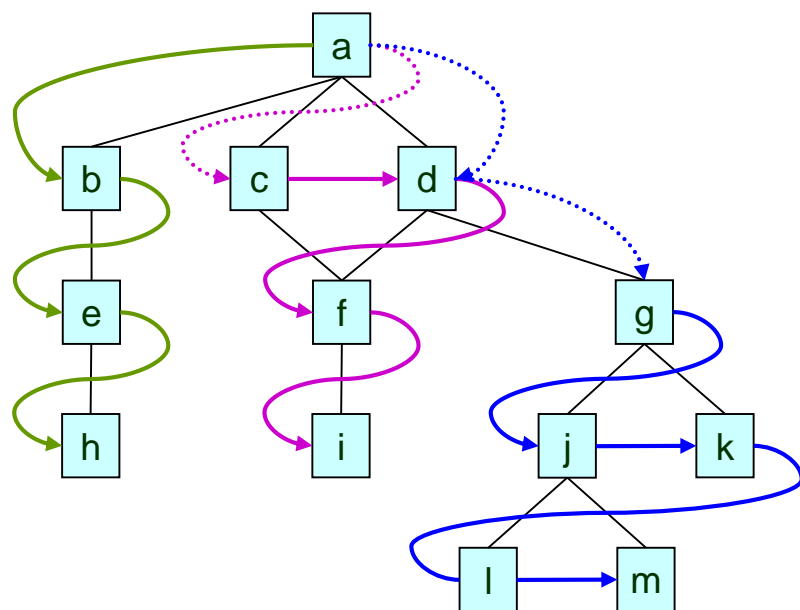
- אילו יחידות לבחור בכל סבב?
- באיזה סדר לשלב?

# מימוש ושילוב top-down

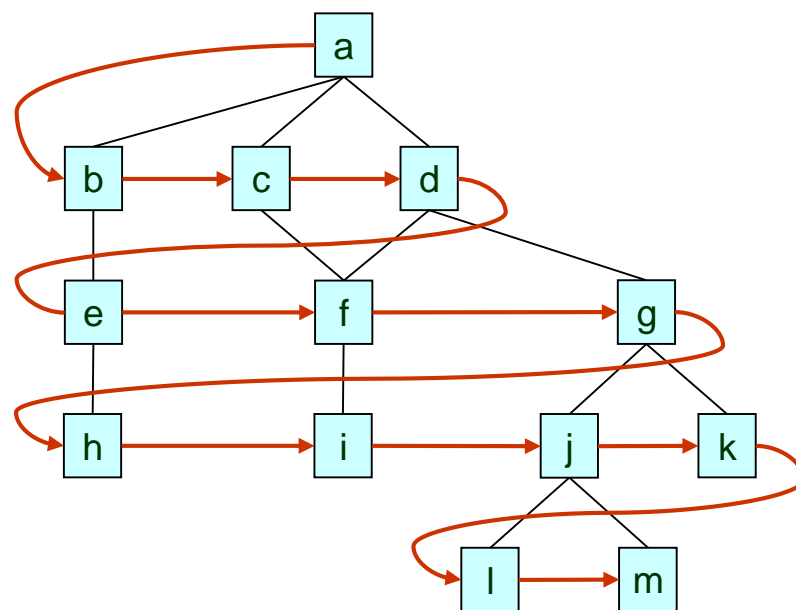
- אם פריט  $p_1$  קורא לפריט  $p_2$  אזי  $p_1$  ימומש וישולב לפני  $p_2$



פעילות מקבילית



פעילות סדרתית

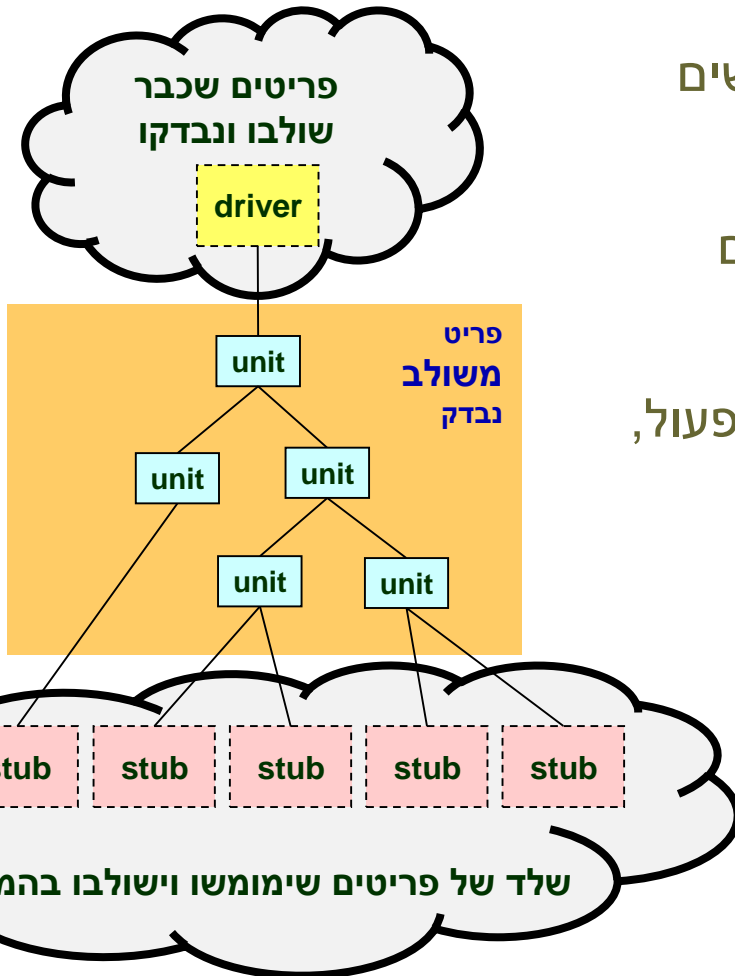




# מימוש ושילוב בסבבים – Top-Down

## • יתרונות

- הפריטים שכבר שולבו ונבדקו משמשים כ-drivers לפריטים החדשים
- ניתן לממש חלקית (שלד) של פריטים עתידיים שימשו כ-stubs
- בעיות עקרוניות (למשל בלוגיקת התפעול, בתפריטים הראשיים, ...) מתגלות בשלבים מוקדמים



# מימוש ושילוב top-down: חסרונות

- חסרון 1: סיכון בשימוש חוזר (reuse) של פריטים

– פריטים ביצועיים

- נמצאים בתחתית העץ, ולכן נבדקים פחות
- מיועדים לשימוש חוזר יותר מאשר פריטים לוגיים

```
if (A >= 0)
 B = computeSquareRoot (A) ;
```

תכנות "הגנתי"  
defensive programming

הפתרון:  
פריטים מגנים  
על עצמם!

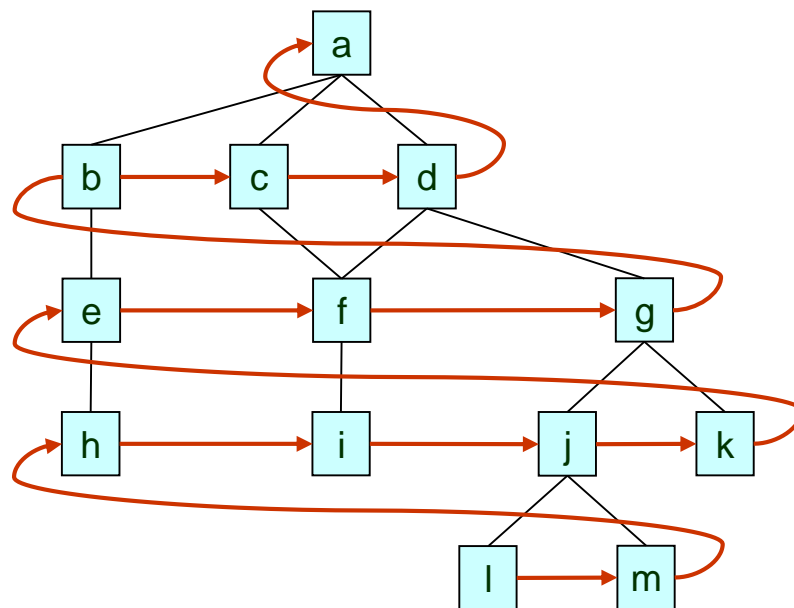
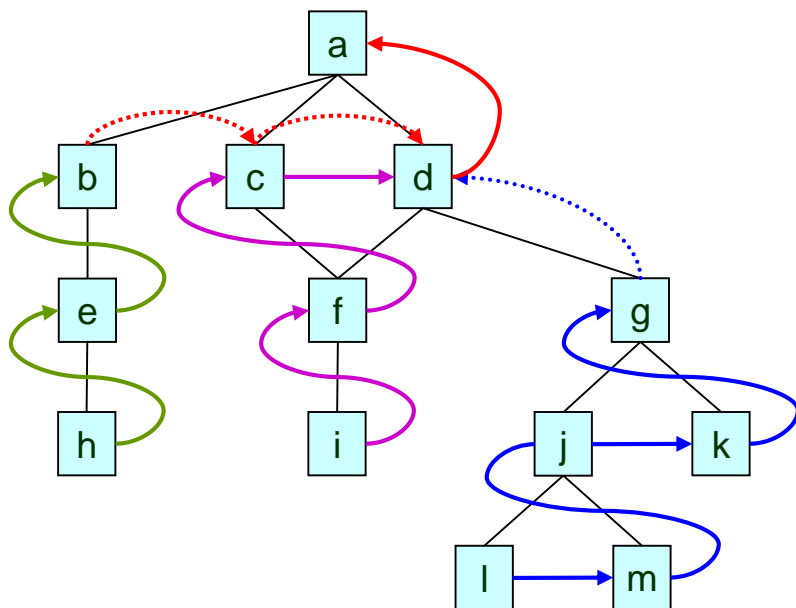
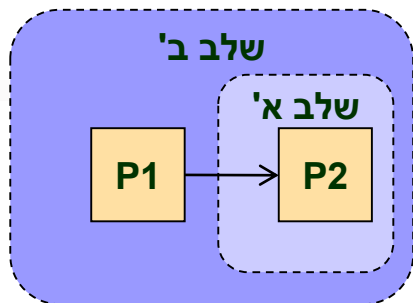
```
real computeSquareRoot (real X)
{
 ...
}
```

מתודה זו לא נבדקה  
מעולם עם פרמטר שלילי!



# מימוש ושילוב bottom-up

- אם פריט  $k_1$  קורא לפריט  $k_2$  אזי  $k_2$  ימומש וישולב לפני  $k_1$

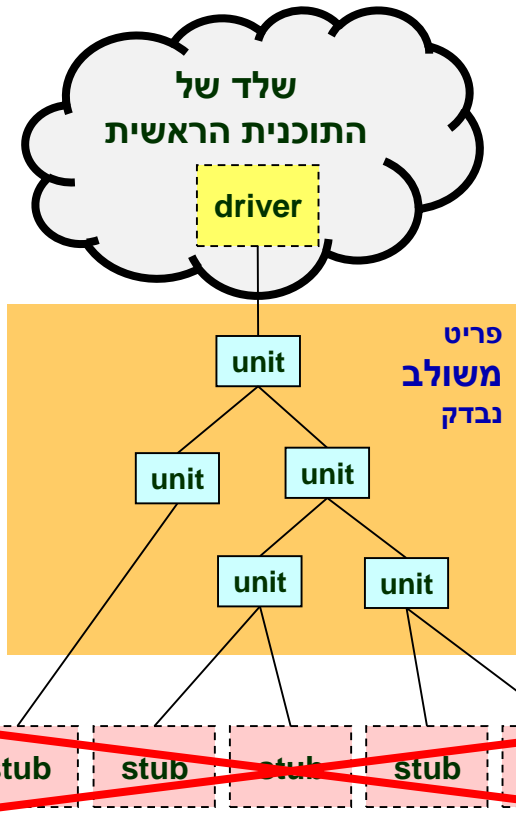


# מימוש ושילוב בסבבים – Bottom-Up

- יתרונות

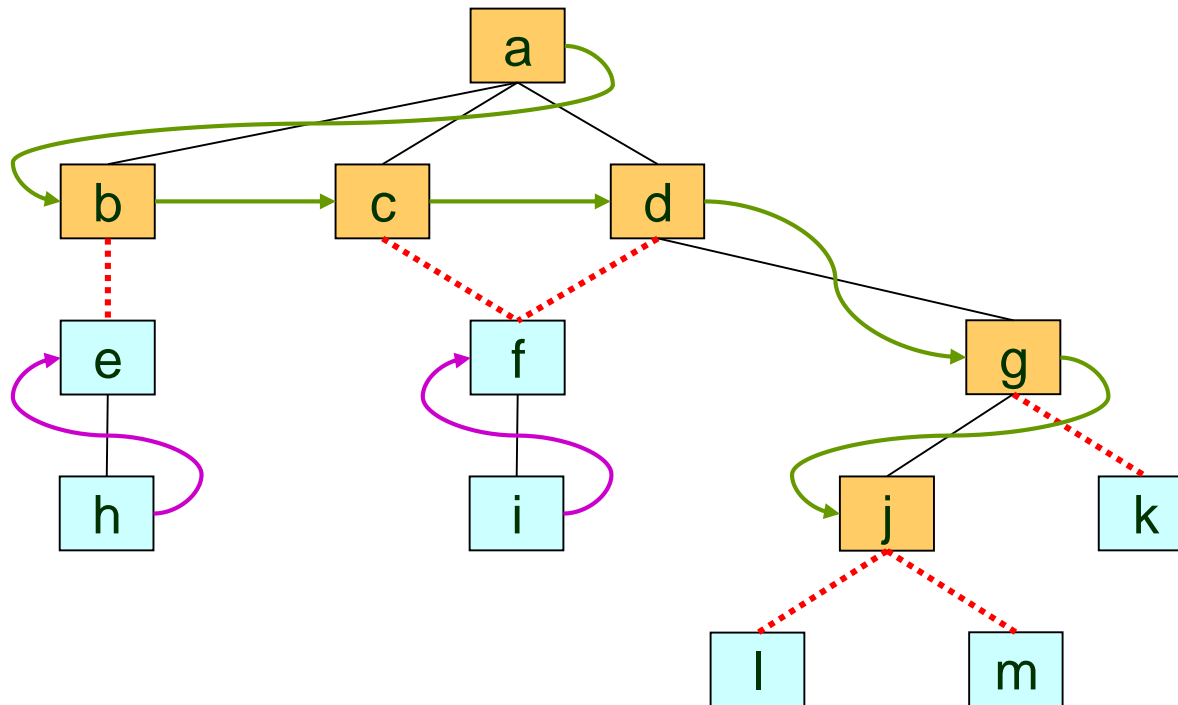
- אין צורך ב-stubs

- מתאים במיוחד לשילוב רכיבים בשימוש חוזר (reuse)



# מימוש ושילוב בשיטת הסנדוויץ' (inside-out)

- פריטים לוגיים ממומשים ומשולבים top-down
- פריטים ביצועיים ממומשים ומשולבים bottom-up
- הממשקים בין שתי הקבוצות נבדקים אחרונים



# מימוש ושילוב בשיטת הסנדוויץ': יתרונות

- יתרון 1: בידוד התקלות

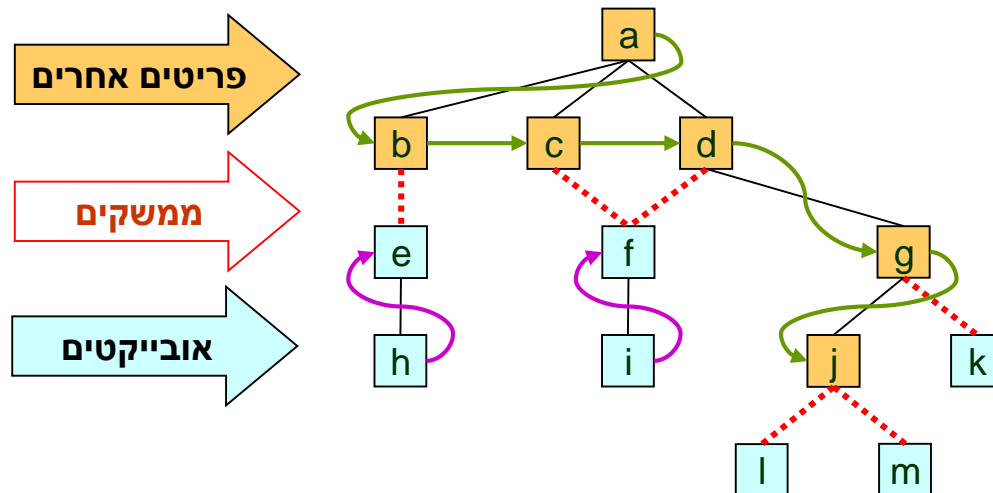
– בכל שלב ושלב נוספים פריטים חדשים על אלה שנבדקו כבר

- יתרון 2: בדיקה יסודית של פריטים ביצועיים

– מאפשרת שימוש חוזר בהם בעתיד

- יתרון 3: איתור תקלות תכן וארכיטקטורה

– תקלות תכן וארכיטקטורה מתגלות בשלבים מוקדמים

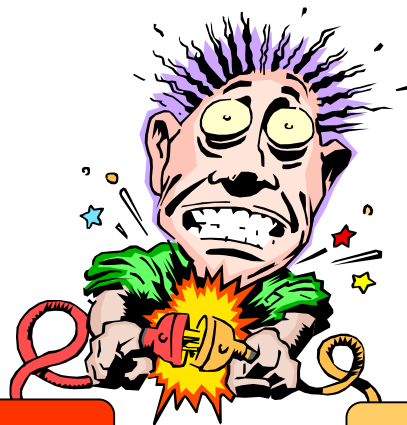


מימוש ושילוב  
מונחה-עצמים  
נעשה, מטבעו,  
בשיטת הסנדוויץ'



# אינטגרציות - ההיבט הניהולי

“ניפגש באינטגרציה...”



`procedure f(x,y)`

`call f(x,y,z)`

- עקרונות בסיס לשילוב מוצלח:

- תכנון נכון של סדר השילוב
- ניהול הממשקים לכל אורך הדרך
- בדיקה מקדמית של היחידות המשולבות

# פעילות השילוב, האימות והתיקוף של המערכת

## • מטרת הפעילות

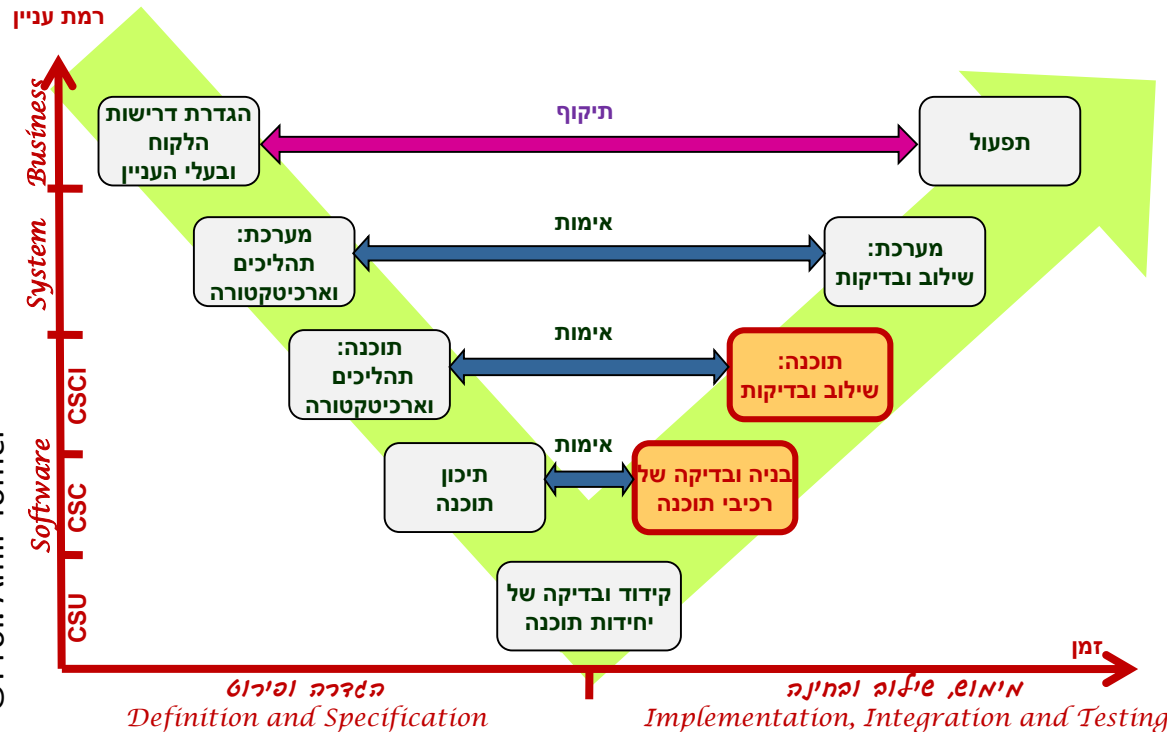
- יצירת מערכת העומדת בבדיקות הקבלה שהוסכמו עם הלקוח
- תפעול המערכת בסביבתה, איסוף נתונים ותיקון ליקויים

## • קלט

- תוכנה בדוקה, חומרה בדוקה

## • תוצרים

- מערכת עובדת ובדוקה





# הוכחת המוצר – אימות ותיקוף

- יש להוכיח כי המוצר (הפריט) המפותח

- עונה לדרישות המוגדרות ממנו (אימות – Verification)

- ממלא את ייעודו בסביבתו המיועדת (תיקוף – Validation)

- לדוגמה, מערכת כספומטים

- דרישות

- משתמש יוכל למשוך כסף רק אם היתרה בחשבונו גדולה מסכום המשיכה

- במקרה של טעות חוזרת בהקשת המספר הסודי לא יוחזר הכרטיס למשתמש

- ייעוד

- לספק שירותי בנקאות מקוונת ל-200,000 לקוחות ביממה, מבלי לחרוג מכללי השימוש,

- ותוך שמירה מלאה על שלמות ונכונות המידע הבנקאי (integrity)

**תיקוף (validation)**

**האם אנחנו מפתחים  
את המוצר הנכון ?**

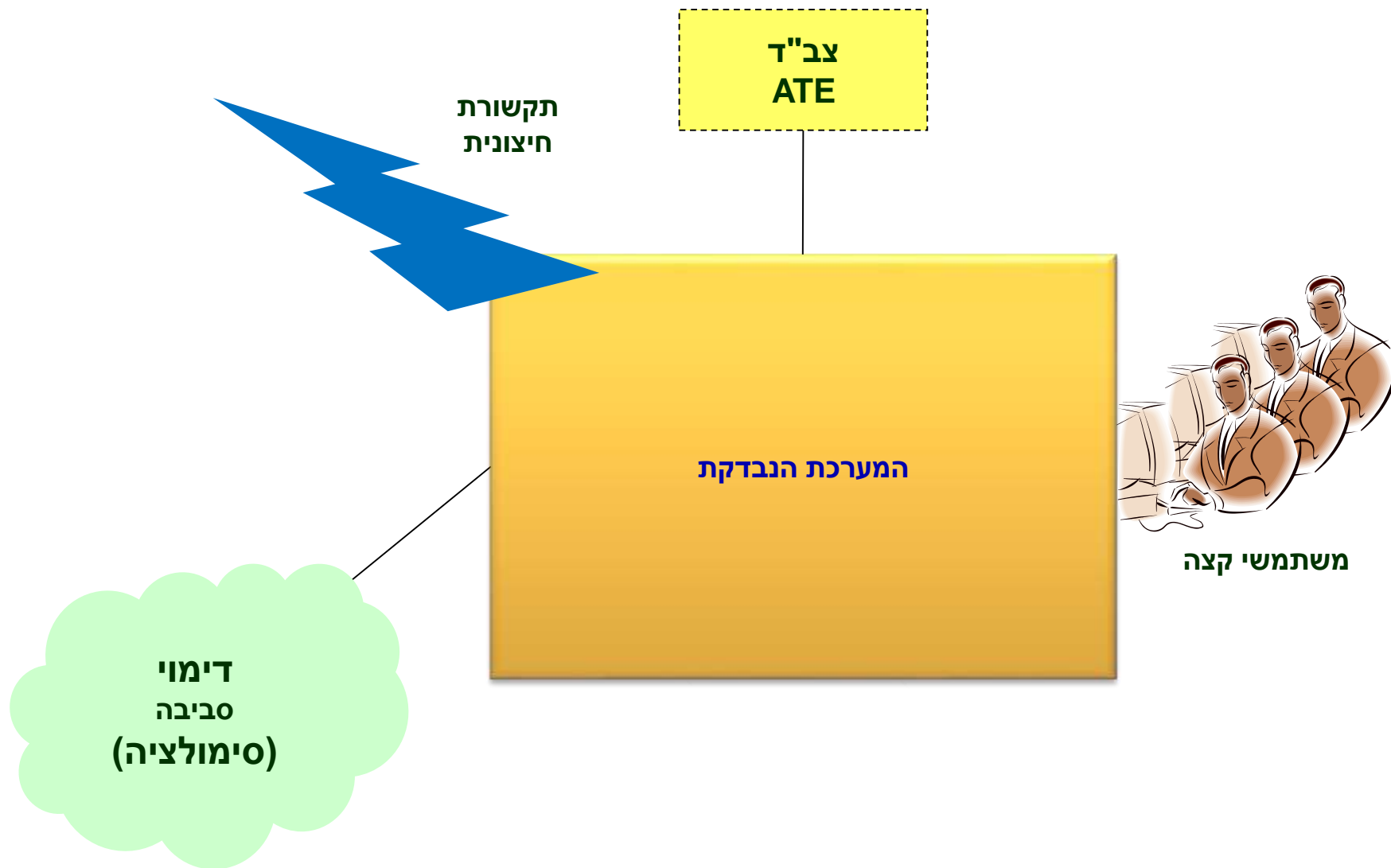
**אימות (verification)**

**האם אנחנו מפתחים  
נכון את המוצר?**

# שיטות להוכחת המוצר

- **בדיקות / ניסויים (Tests / Field Tests)**
  - הפעלת המוצר, או חלק ממנו, על מנת לבחון את ביצועיו בפועל
  - למשל: בדיקת תוכנת הכספומט בתרחישי ההפעלה המוגדרים
- **אבטיפוס / מדגים (Prototype / Demo)**
  - בניית שלד חלקי של הפתרון על מנת לצפות בהתנהגותו המיועדת
  - למשל: בניית demo של מסכים ולוגיקת מעברים של כספומט במחשב אישי
- **סקירה (Review)**
  - בחינת הייצוג של הפתרון המוצע
  - למשל: קריאת מפרט הדרישות, הבנת ה-Use Cases והקשר בינם לבין הדרישות התפעוליות
- **סימולציה (Simulation)**
  - בניית מודל של הפתרון לבחינת התנהגותו
  - למשל: סימולציה של טיסה בחלל על בסיס אלגוריתמי הניווט וההיגוי שבתוכנת החללית
- **ניתוח (Analysis)**
  - הוכחה תיאורטית לנכונות הפתרון
  - למשל: בניית עץ מצבים המוכיח כי מצבים "בלתי רצויים" אינם ניתנים להשגה

# סביבת בדיקה למערכת



# טבלת אימות ותיקוף (V&V Table)

• הגדרת אופן ההוכחה של כל אחת מהדרישות

• שימושי הטבלה

– עקיבות הדרישות לבדיקות

– כיסוי הדרישות באמצעות בדיקות

– בסיס לתכנית הבדיקות (הניסויים)

• דוגמה

| אופן הוכחת הדרישה |             |             |           |                 | NFR | FR | נוסח הדרישה                                                                             |
|-------------------|-------------|-------------|-----------|-----------------|-----|----|-----------------------------------------------------------------------------------------|
| תיקוף             | אימות מערכת | אימות תוכנה | סקר תיכון | אנליזה/סימולציה |     |    |                                                                                         |
|                   |             |             | V         |                 | HC  |    | בתוך כל מעלית נמצאים 10 כפתורים ... וכמו כן ... כפתור לעצירת חירום וכפתור להזעקת חילוץ. |
|                   |             | V           |           |                 |     | OR | בעקבות הלחיצה על <b>כפתור מעלית</b> למעלית נוספת בקשה לעצירה בקומה המתאימה              |
| V                 |             |             |           |                 | PR  |    | מעלית כלשהי הנמצאת בכיוון הנסיעה המבוקש תגיע לקומה, תוך דקה לכל היותר                   |

# בדיקות אימות למערכת עתירת תוכנה

- **בדיקות התאמה / בדיקות פונקציונליות (Conformance testing)**
  - מקרי-בדיקה לצורך תיקוף ההתנהגות של המערכת / הרכיב בהתאם לדרישות הפונקציונליות
  - ניתנות לביצוע גם בכל הרמות: יחידה, שילוב ומערכת
- **בדיקות רגרסיה (Regression Testing)**
  - בדיקה-חוזרת סלקטיבית של מערכת או רכיב על מנת לוודא ששינויים לא גרמו לתופעות בלתי מכוונות ושהמערכת או הרכיב עדיין מתאימים לדרישותיהם
- **בדיקות קבלה / כשירות (Acceptance / qualification testing)**
  - הבדיקות הסופיות לפני כניסת המוצר לשימוש מבצעי
  - המטרה העיקרית: עמידה בדרישות הלקוח
  - במקרים רבים מתבצעות בשני סבבים
- אצל המפתח לפני האספקה (FAT = Factory Acceptance Tests)
- באתר הלקוח לאחר ההספקה (SAT = Site Acceptance Tests)
- **השגת אמינות (Reliability achievement)**
  - בדיקות אקראיות של פונקציונליות המערכת, במקרי בדיקה שונים
    - למשל, נתוני בדיקה אקראיים בשיטת monte-carlo
  - המטרה: להגדיל את ההסתברות לגילוי שגיאות במקרים סינגולריים בלתי צפויים

# בדיקות תיקוף למערכת עתירת תוכנה

- **בדיקות התקנה (Installation testing)**
  - בדיקת המערכת לאחר התקנתה בסביבת המטרה
  - למעשה, ביצוע חוזר של בדיקות הקבלה בסביבת החומרה
  - בדיקת פרוצדורות ההתקנה
- **בדיקות ביצועים (Performance Testing)**
  - בדיקות ספציפיות לבחינת העמידה בדרישות הביצועים (P)
- **בדיקות שימושיות (Usability testing)**
  - בחינת קלות ונוחות השימוש במערכת ("ידידותיות") ובתיעוד
  - הערכת עקומת הלמידה לשימוש במערכת
  - בחינת האפקטיביות של המערכת בהשגת מטרות המשתמש
  - בחינת יכולת המערכת להתאושש מטעויות משתמש
- **בדיקות אלפא/ביתא (Alpha/Beta testing)**
  - בדיקה נסיונית של התוכנה ע"י בודקים מזדמנים
    - אלפא = בדיקה פנימית אצל המפתח, הבודקים מתוך החברה
    - ביתא = בדיקה "בשטח", בודקים מזדמנים ואקראיים
  - אין תכנית בדיקות – הבודק מחליט מה ואיך לבדוק

- **תכנית הבדיקות – Software Test Plan**

- סדר השילוב והבדיקות
- מקרי בדיקה (test cases)
- תרחישי בדיקה
- צירוף תנאים / נתונים לבדיקה
- תכנון סביבת הבדיקות

- **מפרט בדיקות – Software Test Description**

- שגרות בדיקה (test procedures)
- פירוט אופן ביצוע הבדיקות בסביבת הבדיקות

- **סביבת הבדיקות**

- רכיבי בדיקה (test components)

- **הבדיקות נגזרות ישירות מהדרישות**

- גם אם הדרישות לא כתובות כתרחישים, הבדיקות מבוצעות ע"י הפעלת תרחישים שונים

- מפרט בדיקה (test procedure) חייב לציין במפורש

- pre-conditions (תחת אלו תנאים ניתן לבצע את הבדיקה)

- trigger (כיצד מפעילים את הבדיקה)

- steps (צעדי האינטראקציה של הבודק / כלי הבדיקות עם המערכת)

- expected results / post-conditions (התוצאות הצפויות)

- **לבדיקות שתי מטרות עיקריות**

- בדיקות "חיוביות"

- להוכיח שהמערכת אכן עושה את מה שנדרש (מגיבה נכון לאינטראקציה החוקית עם הסביבה)

- בדיקות "שליליות"

- להוכיח שהמערכת לא נכשלת כאשר הסביבה מבצעת אינטראקציה לא חוקית איתה



# מפרט בדיקות (ידני) – STD – Software Test Description

## Test Case Example1 (simple test)

Test Case #: 2.2

System: ATM

Designed by: ABC

Executed by:

Short Description: Test the ATM Change PIN service

Test Case Name: Change PIN

Subsystem: PIN

Design Date: 28/11/2004

Execution Date:

Page: 1 of 1

### Pre-conditions

The user has a valid ATM card - The user has accessed the ATM by placing his ATM card in the machine

The current PIN is 1234

The system displays the main menu

| Step | Action                        | Expected System Response                                                                                                  | Pass/Fail | Comment |
|------|-------------------------------|---------------------------------------------------------------------------------------------------------------------------|-----------|---------|
| 1    | Click the 'Change PIN' button | The system displays a message asking the user to enter the new PIN                                                        |           |         |
| 2    | Enter '5555'                  | The system displays a message asking the user to confirm (re-enter) the new PIN                                           |           |         |
| 3    | Re-enter '5555'               | The system displays a message of successful operation<br>The system asks the user if he wants to perform other operations |           |         |
| 4    | Click 'YES' button            | The system displays the main menu                                                                                         |           |         |
| 5    | Check post-condition 1        |                                                                                                                           |           |         |

### Post-conditions

1. The new PIN '5555' is saved in the database

## רכיבי בדיקה test components

- **רכיב תוכנה לביצוע אוטומטי של שגרת בדיקה**

- תכנית בשפת תכנות

- script

- בדיקה “מוקלטת” באמצעי בדיקה אוטומטי

- **משולב בביצוע**

- מייצר קלט

- בקרה ומעקב אחר הביצוע של הרכיב הנבדק

- דיווח / בחינה של התוצאות

# שימוש ב-UML לבדיקות

- Use Case Model

- ניתן להשתמש ב-Use Cases כ-Test Cases
- אפשר להפעיל את שיטת מסלולי הבסיס כדי לכסות את כל תרחישי ה-UC (MSS, חלופות, חריגות)

- Class Model

- הגדרת אובייקטי-בדיקה

- Sequence Diagrams

- הגדרת תרחישי בדיקה

- שילוב של אובייקטי בדיקה ב-SD המקוריים

- Component Diagram / Deployment Diagram

- סדר אינטגרציה, אינטגרציות חלקיות
- שילוב רכיבי בדיקה עם רכיבי מערכת (חומרה / תוכנה)

## אפקטיביות של בדיקות תוכנה\*

כמות השגיאות שהתגלו בסבב הבדיקות

= אפקטיביות

סך כל השגיאות שהתגלו במחזור החיים, שמקורן בתוצר הנבדק

| סבב בדיקות             | נמוך | חציון | גבוה |
|------------------------|------|-------|------|
| בדיקות יחידה           | 10%  | 25%   | 50%  |
| בדיקת פונקציות חדשות   | 20%  | 35%   | 55%  |
| בדיקות שילוב           | 25%  | 45%   | 60%  |
| בדיקות מערכת           | 25%  | 50%   | 65%  |
| בדיקות "ביתא" חיצוניות | 35%  | 40%   | 75%  |

\* Capers Jones, SOFTWARE QUALITY IN 2008

