

תוכנה 1 בשפת Java
שיעור מספר 14: "מה להנדסה ולזה?"

פרופ' ליאור וולף

על סדר היום

- מעבר לתכנות בשפת Java ותכנות מונחה עצמים
- תוכנה אינה רק תכנות (גם בדיקות גם תיכון)
- תבניות תיכון (design patterns) קלאסיות בתכנות מונחה עצמים
- חתכי רוחב (crosscutting concerns)
- שכתוב מבני (refactoring)



מבוא להנדסת תוכנה

מבוא להנדסת תוכנה

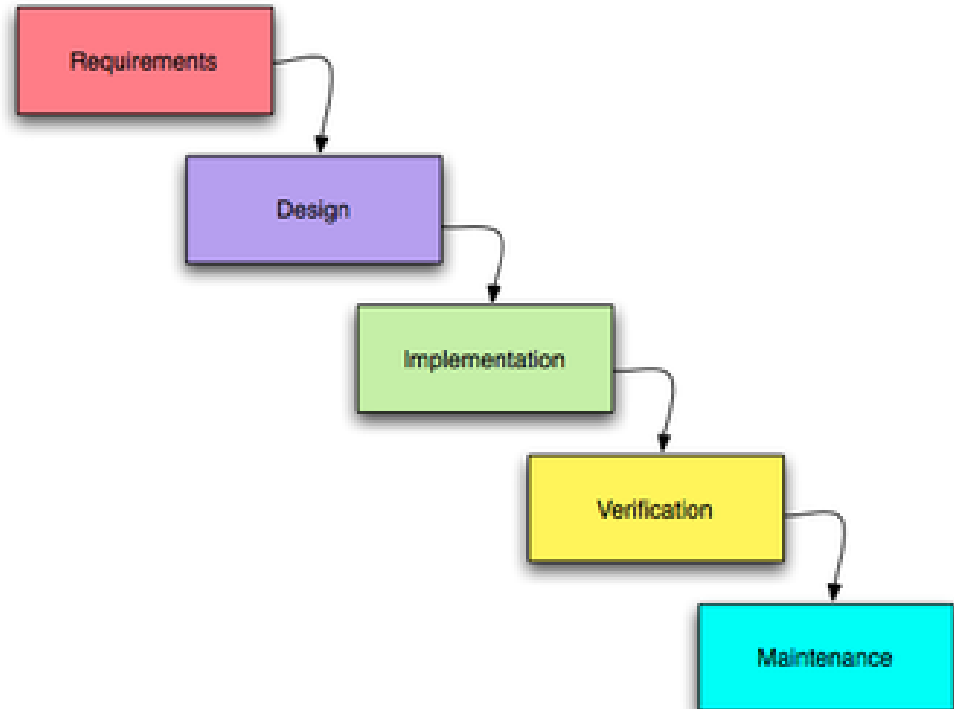
- תהליך הפיתוח של תוכנה אינו מורכב רק מתכנות ובדיקות
- התהליך מתחיל לפני הפיתוח ונמשך גם אחרי שהפיתוח הסתיים
- הנדסת תוכנה מתיימרת להיות תחום הנדסי העוסק בכל ההיבטים של יצירת מערכות תוכנה.
- בחלק הזה של הקורס נדון בקצרה בשלבים שלפני ואחרי הפיתוח, במה שמשותף להם ולפיתוח ובמה ששונה
- הדיון יהיה תמציתי ולא ממצה; הנושא רחב מדי
- קיימים קורסי בחירה מתקדמים בחוג המתמקדים בשלבים השונים
- הדיון אינו ספציפי לתכנות מונחה עצמים

מחזור החיים של תוכנה

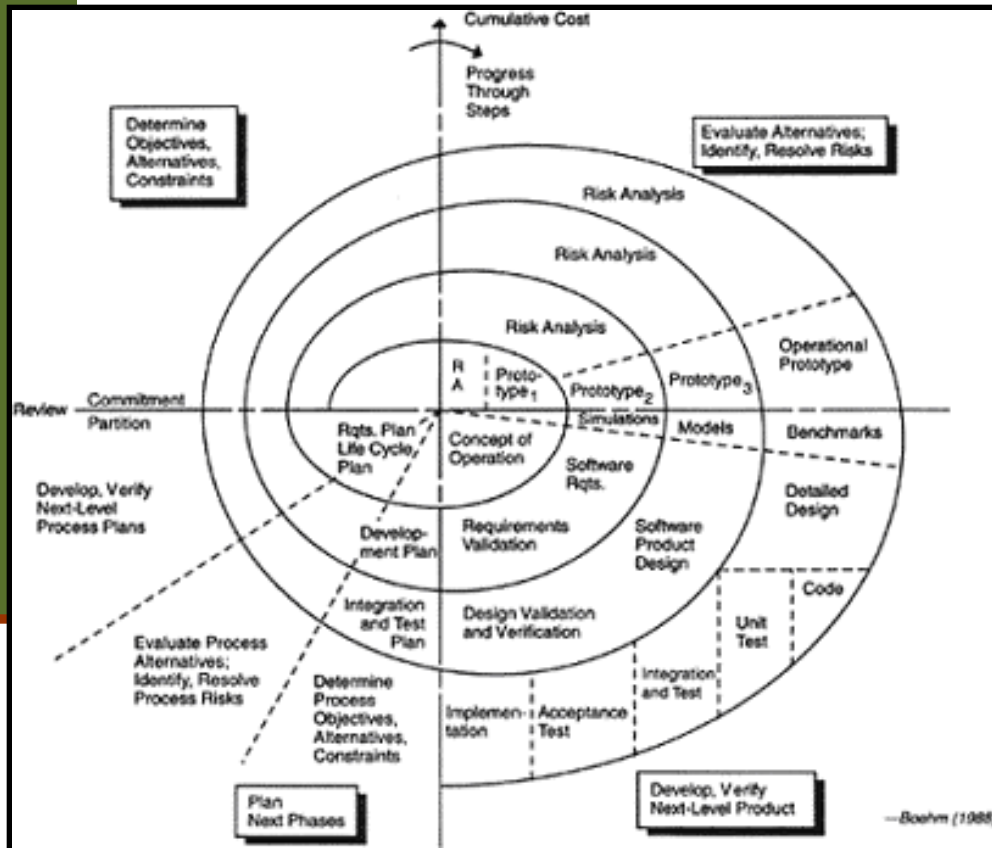
- ניתוח דרישות (requirements analysis)
 - תיכון (design)
 - מימוש (Construction, implementation or coding)
 - שילוב (integration)
 - בדיקות וניפוי שגיאות (Testing and debugging aka: verification)
 - בדיקות קבלה
 - ייצור (production)
 - הפצה והתקנה (deployment and installation)
 - תחזוקה ושינויים (maintenance)
- התייחסות מיוחדת למקרה שמערכת התוכנה היא חלק ממערכת ממוחשבת הכוללת חומרה ותוכנה.

מודל המפל

- המודל המסורתי של מחזור חיים נקרא מודל מפל המים (waterfall model, Royce 1970) - כל שלב מתבצע לאחר שקודמו הסתיים (אך ניתן לחזור לשלב קודם לצורך תיקון).



מודל ספירלה



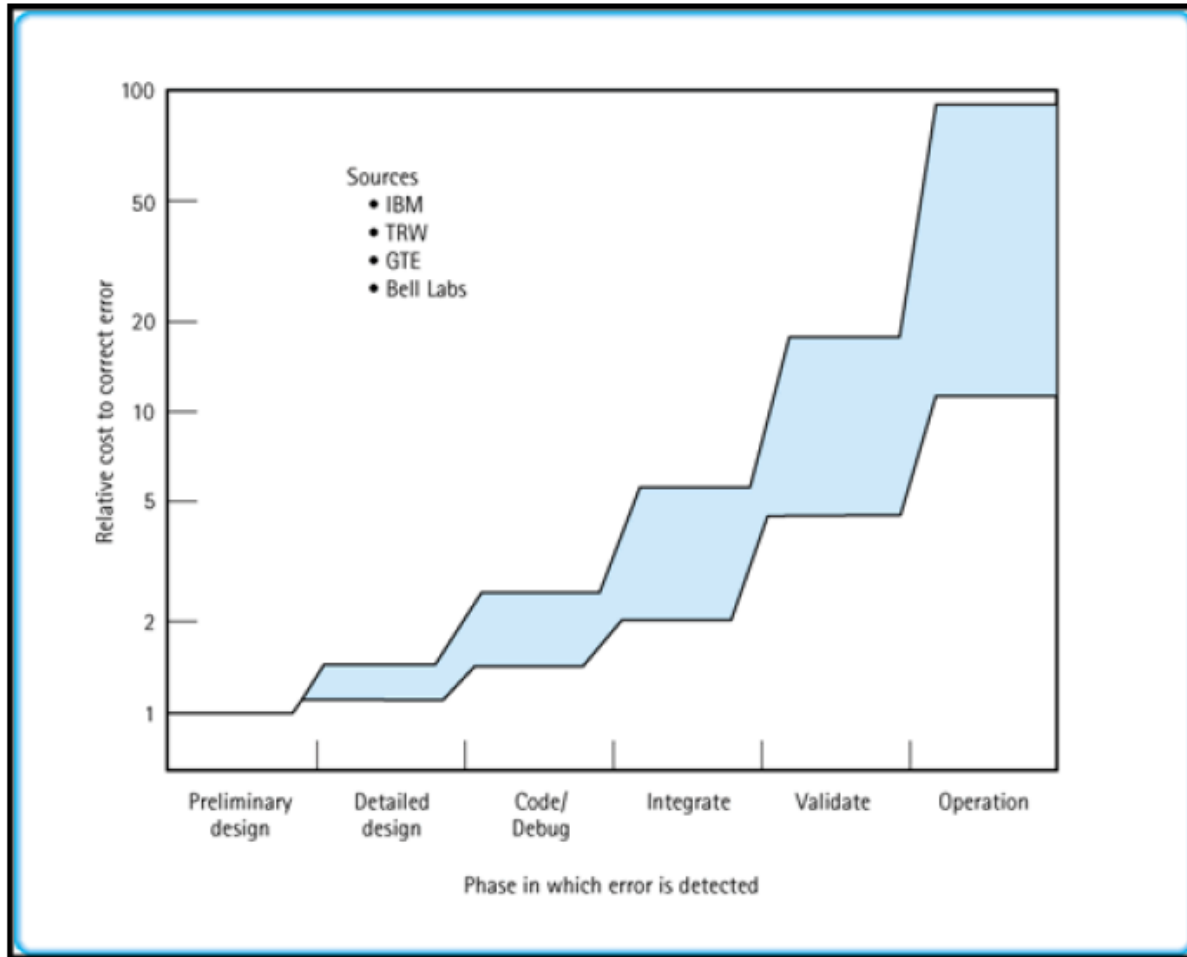
מודל הספירלה (spiral model) שהוצע מאוחר יותר (Barry Boehm, 1988) מפתח את המערכת באופן אבולוציוני.

מתחילים מפיתוח מערכת מינימלית, ומבצעים את כל השלבים. לאחר סיום מעריכים את המוצר הנוכחי, מחליטים מה להוסיף, וחוזרים על כל השלבים

מחירן של טעויות

- ככל שטעות מתגלה מוקדם יותר, מחיר תיקונה קטן יותר
- נניח שטעינו בניתוח הדרישות ושכחנו פעולה מסוימת שהתוכנה צריכה לבצע
 - אם נגלה את הטעות לפני המעבר לתיכון, המחיר יהיה מינימאלי, אולי עיכוב קטן בלוח הזמנים
 - אם נגלה בזמן התיכון, נצטרך אולי לזרוק חלק מהתיכון שלא יתאים לדרישות המתוקנות
 - אבל אם נגלה את הטעות רק בזמן בדיקות הקבלה, נצטרך אולי לזרוק חלקים גדולים מהתיכון ומהמימוש!
- עדיף לגלות טעויות מוקדם; לשם כך צריך לתכנן בקפדנות את תהליך הפיתוח הכולל, ולהשתדל להשתמש בשיטות שימזערו טעויות ואת הצורך לחזור אחורה לשלב קודם

מחירן של טעויות



מפל או ספירלה?

- מודל הספירלה מאפשר לראות מוצר חלקי ולהעריך אותו
- אבל מפל המים משקף את הרצוי: רצוי לא לטעות.

- שינויים בתוכנה אינם רק תוצאה של שגיאות, **שינוי הוא דבר מובנה** בתהליך הפיתוח

- פיתוח תוכנה תוך הערכות לשינויים עתידיים הביא למודלים נוספים לתהליך הפיתוח:
 - בשנים האחרונות עולה הפופולריות של משפחת המודלים הקלילה (agile)
 - הנציג הבולט של המשפחה הזו הוא eXtreme Programming (תכנות קיצוני)

תכנות קיצוני (XP)

■ מעצבי השיטה (Kent Beck, Ward Cunningham, Ron Jeffries, 1996) ניסחו 4 ערכים:

■ משוב (feedback)

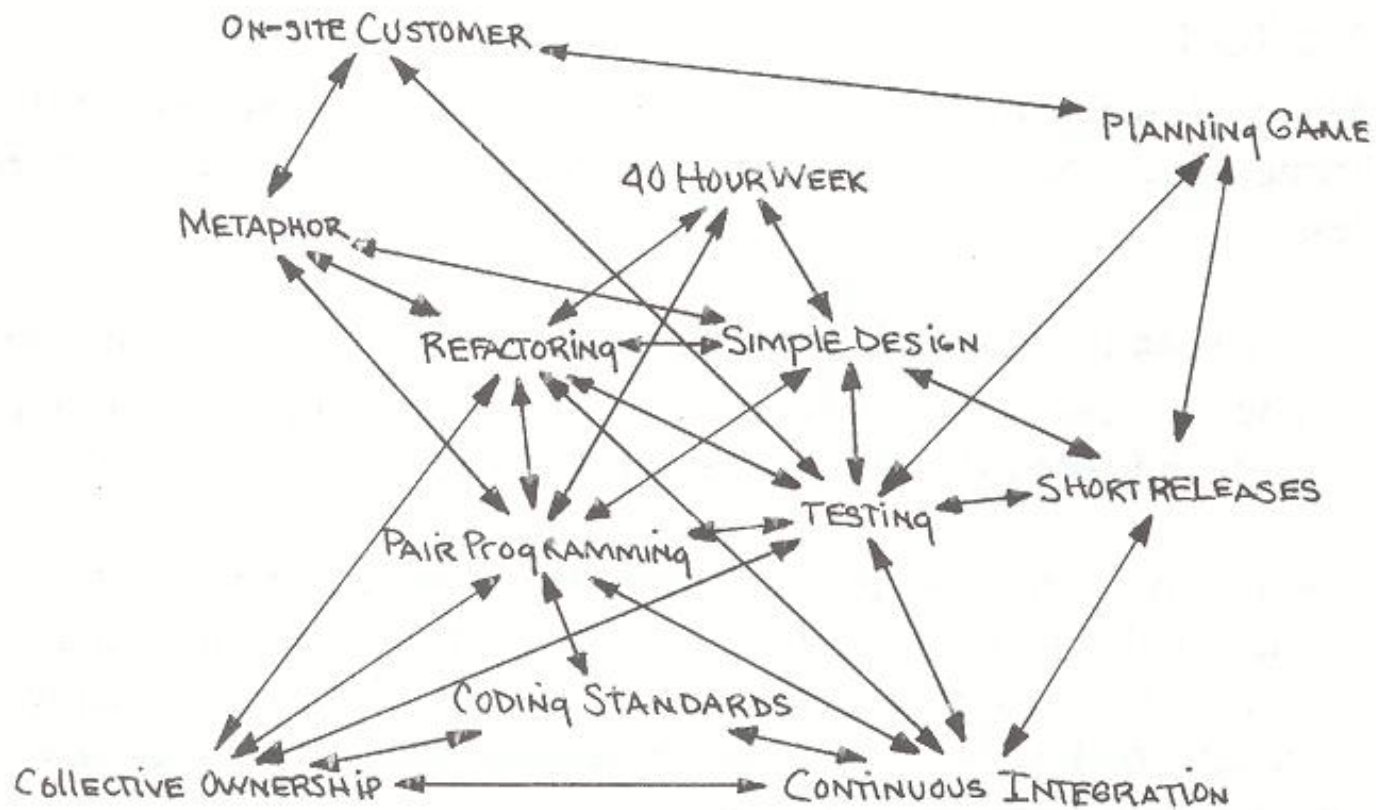
■ פשטות (Simplicity)

■ תקשורת (Communication)

■ אומץ (Courage)

■ ערכים אלו מבוטאים ב 12 מיומנויות תוכנה

■ מכיוון שהערכים והמיומנויות הוכחו כטובים, נלקח כל אחד מהם לקיצוניות



Source: Beck, K. (2000). *eXtreme Programming explained*, Addison Wesley.





How the customer explained it

בדיקות תוכנה

איך יודעים שמודול או תוכנית נכונים?

- **אימות:** תהליך שמיועד לוודא באופן פורמאלי או לא פורמאלי נכונות של מודול או תוכנית ביחס לחוזה
- **אימות פורמאלי אוטומאטי** אינו אפשרי במקרה הכללי (לא כריע). למרות זאת קיימים כלים פורמליים שלעיתים אינם מצליחים.
- **אימות פורמאלי ידני** יקר מדי לרוב המערכות פרט אולי למערכות שחיי אדם תלויים בהן ישירות (רפואיות, מוטסות, וכולי, אבל גם שם יש פחות אימות ממה שהיה ראוי)
- **בדיקות (testing):** ביצוע סדרת הרצות של התוכנה שמיועדות למצוא פגמים, אם יש, ולהגדיל את בטחוננו בנכונותה
 - לא מבטיח נכונות, אבל יותר טוב מכלום, ומועיל מאוד באופן מעשי להקטנת מספר הפגמים

אל תירה בשליח

- כאשר המכונת לא עוברת טסט, זה כמובן מעצבן, אבל זה בדרך כלל לא **כישלון** של מכון הרישוי שביצע את הטסט
- **כישלון והצלחה** של בדיקה הם נפרדים לחלוטין מאלה של הקוד הנבדק!
- בדיקה **מצליחה** אם היא מגלה פגם
- בדיקה **נכשלת** אם היא לא מגלה פגם או מדווחת על פגם לא קיים
- אם בדיקה מדווחת על פגם נאמר שהקוד לא עבר את הבדיקה, ולא נאמר שהבדיקה נכשלה
- דווח על פגם הוא אירוע חיובי (לא משמח אולי, אבל חיובי) כי הוא מספק אפשרות לתיקון פגם לפני שהוא גורם עוד נזק

שלושה סוגי בדיקות

■ **בדיקות יחידה (unit tests)** בודקות מודול בודד (שרות, מחלקה אחת או מספר מחלקות קשורות)

■ **בדיקות אינטגרציה** בודקות את התוכנית כולה, או קבוצה של מודולים ביחד; מתבצעת תמיד לאחר בדיקות היחידה של המודולים הבודדים (כלומר על מודולים שעברו את בדיקות היחידה שלהם)

■ **בדיקות קבלה (acceptance tests)** מתבצעות על ידי הלקוח או על ידי צוות שמתפקד בתור לקוח, לא על ידי צוות הפיתוח

■ גם לאחר כניסה לשימוש, התוכנה ממשיכה למעשה להיבדק, אבל אצל משתמשים אמיתיים; רצוי שיהיה מנגנון דיווח לתקלות ופגמים שמתגלים בשלב הזה, ורצוי לתקן את הפגמים הללו

קופסאות שחורות וקופסאות פתוחות

על כל מודול תוכנה צריך לבצע שני סוגים של בדיקות יחידה:

■ **בדיקות קופסה שחורה (black-box tests)**

- בודקים את הקוד מול החוזה שהוא מבטיח לקיים, והן אינן תלויות במימוש
- בדיקות קופסה שחורה לא תלויות במימוש ולכן אותו סט בדיקות תקף לכל המימושים של מנשק מסוים, גם העתידיים, ובפרט לשינויים ותיקונים במימוש הנוכחי

■ **בדיקות כיסוי (glass-box tests או coverage tests)**

- דואגות שבזמן הבדיקות, כל פיסת קוד תרוץ, ובמקרים מסוימים, תרוץ יותר בכמה צורות
- בדיקות כיסוי צריך לעדכן כאשר מעדכנים את הקוד

איך בודקים?

- בבדיקות מעורבים שני סוגי קוד: מנועים ורכיבים חלופיים
- **מנוע** (driver) הוא קוד שמדמה לקוח של המודול הנבדק וקורא לו
- **רכיב חלופי** (stub) מחליף ספק שמשרת את המודול הנבדק
- למשל מחלקה A משתמשת ב-B שמשתמשת ב-C
- בדיקת יחידה ל-B תדמה לקוח של B ותספק מחלקה חלופית ל-C, על מנת שניתן יהיה לבדוק את B בנפרד מ-A ו-C
- רכיב חלופי צריך להיות פשוט ככל האפשר
- לפעמים הרכיב החלופי לא יכול להיות משמעותית יותר פשוט מהמודול שאותו הוא מחליף, ואז כדאי להשתמש במודול האמיתי לאחר בדיקות יסודיות שלו



תוכנה 1 בשפת Java
אוניברסיטת תל אביב

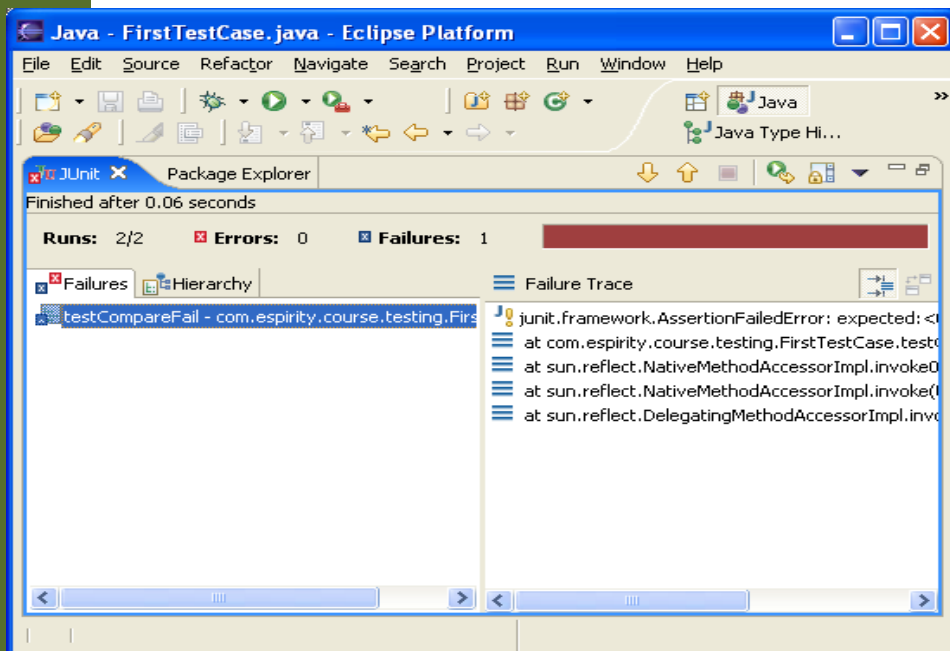
בדיקות רגרסיה

- בכל פעם שמגלים פגם בתוכנה, בכל שלב של חיי התוכנה (גם לאחר שנכנסה לשימוש) יש להוסיף **בדיקה שחושפת את הפגם**, כלומר שנכשלת בגרסה עם הפגם אבל עוברת בגרסה המתוקנת
- לפעמים הבדיקה תתווסף לבדיקות הקופסה השחורה ולפעמים לבדיקות הכיסוי (אם הפגם קשור באופן הדוק למימוש ולא לחוזה)
- את סט הבדיקות השלם, כולל כל הבדיקות הללו שנוצרו בעקבות גילוי פגמים, **מריצים לאחר כל שינוי** במודול הרלוונטי, על מנת לוודא שהשינוי לא גרם לרגרסיה, כלומר להופעה מחודשת של פגמים ישנים
- סט הבדיקות מייצג, כמו התוכנה המתוקנת, **ניסיון מצטבר** ויש לו ערך טכני וכלכלי משמעותי

בדיקות צריכות להיות אוטומטיות

- בדיקה שדורשת התערבות של אדם היא בדיקה לא טובה, כי קשה ויקר לחזור עליה אחרי כל שינוי בתוכנה
- לכן, כל בדיקה בדידה צריכה להיות **אוטומטית**
- צריך מנגנון (תוכנה) שמריץ את כל הבדיקות ומדווח על כל הפגמים שהתגלו
- לפעמים צריך להריץ אולי רק חלק, למשל אם ביצענו שינוי קטן בתוכנה; אבל אם הבדיקות מהירות כדאי להריץ את כולן

תמיכה בסביבת הפיתוח

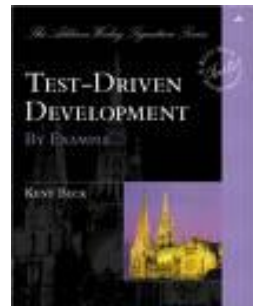


- כלים נוחים לבדיקות יחידה קיימים לכל שפות התכנות ולכל סביבות הפיתוח (JUnit, NUnit, CPPUNIT), הכלים מגדירים את המושג Test Suite ליצירת סדרת בדיקות
- הסביבה מספקת מידע נוח לגבי אלו בדיקות בוצעו אילו עברו ואילו נכשלו
- קל לראות האם נזרקו חריגות ואילו
- קל לנווט בקוד ישירות למקור הבעיה
- אדום = בעיה

פיתוח מונחה בדיקות

מתודולוגיה ששמה דגש על הבדיקות כגורם המניע את התהליך.
חוזרים שוב ושוב על התהליך הבא:

- הוסף במהירות בדיקה.
 - הרץ את כל הבדיקות וראה שהחדשה לא עוברת.
 - בצע שינוי קטן בקוד.
 - הרץ את כל הבדיקות וראה שכולם עוברות.
 - בצע refactoring לביטול כפילות בקוד.
- Kent Beck, Test-Driven Development By example, Addison-Wesley



פיתוח מונחה בדיקות

- הבדיקה מקדימה את הפונקציה!
- הפונקציה הנכתבת היא מינימלית - מטרתה לגרום לבדיקה להצליח
- היבט פסיכולוגי
- לכל מחלקה ולכל מתודה נכתוב מחלקת בדיקה ומתודת בדיקה.
- לדוגמא את המתודה `func` של המחלקה `MyClass` נבדוק בעזרת המתודה `testFunc` של המחלקה `TestMyClass`



חתכי רוחב בתוכנה

Crosscutting Concerns

No Silver Bullet

- בתוכנה אין פתרונות קסם
- גם לתכנות מונחה עצמים יש החסרונות שלו וצריך להיות ערים להם
- חסרון בולט קשור לניהול של חתכי רוחב (crosscutting concerns) במערכת תוכנה
- נניח שכתבנו תוכנה שעושה משהו
 - במערכת התוכנה נמצא את המחלקה `SomeBusinessClass` עם השרות `someOperation`
 - למשל המחלקה `BankAccount` עם השרות `withdraw` (רק לצורך הדוגמא – הדבר תקף כמעט לכל תוכנה אמיתית)

The wrong way

```
public class SomeBusinessClass extends OtherBusinessClass {  
    // Core data members  
    // Override methods in the base class  
    public void someOperation(OperationInformation info) {  
        // ==== Perform the core operation ====  
    }  
  
    ...  
}
```

The wrong way(2)

■ But what about logging capabilities ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
    }  
}
```

The wrong way(3)

- **Actually, we want it multithreaded...**

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```

The wrong way(4)

■ Who enforces your contract ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...ensure info satisfies contract  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```

The wrong way(5)

■ Authorization ? Authentication ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...ensure authorization  
        ...ensure info satisfies contract  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```

The wrong way(6)

■ Persistence ? Cache consistency ?

```
public class SomeBusinessClass extends OtherBusinessClass {

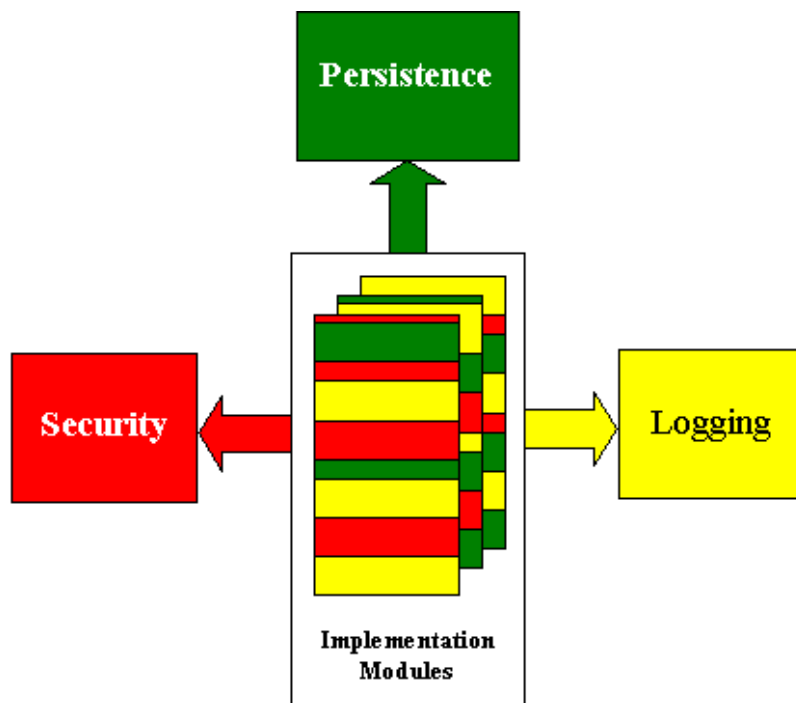
    // Core data members
    ...Log stream ;
    ...cache_update_status ;
    // Override methods in the base class

    public void someOperation(OperationInformation info) {
        ...ensure authorization
        ...ensure info satisfies contract
        ...lock the object - thread safety
        ...ensure cache is up to date
        ...log the start of operation
        // ==== Perform the core operation ====
        ...log the completion of operation
        ...unlock the object
    }

    public void save(PersitanceStorage ps) {...}

    public void load(PersitanceStorage ps) {...}
}
```


מה קיבלנו?



בלאגן בשתי רמות:

■ ברמת המיקרו (השרות הבודד):

■ Code Tangling

■ הוא כבר לא עושה "רק משהו אחד" - לא מודולרי

■ ראו תרשים <=

■ ברמת המאקרו (מערכת התוכנה):

■ Code Scattering

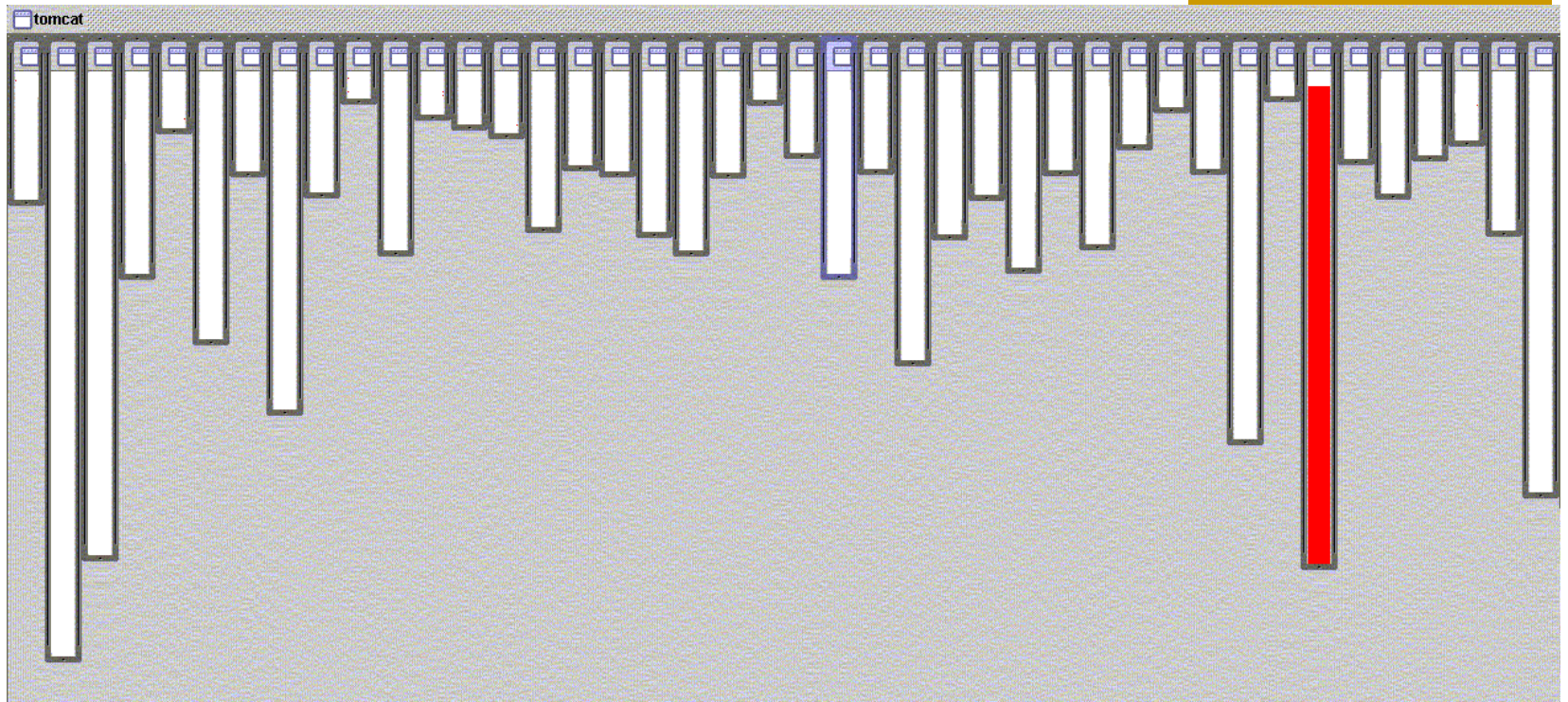
■ שכפול קוד, קטעי קוד קשורים אינם מופיעים יחד

■ ראו תרשימים גם בשקפים הבאים

■ שבירת המודולריות נוצרת בגלל אופי הספק-לקוח של תכנות מונחה עצמים

good modularity

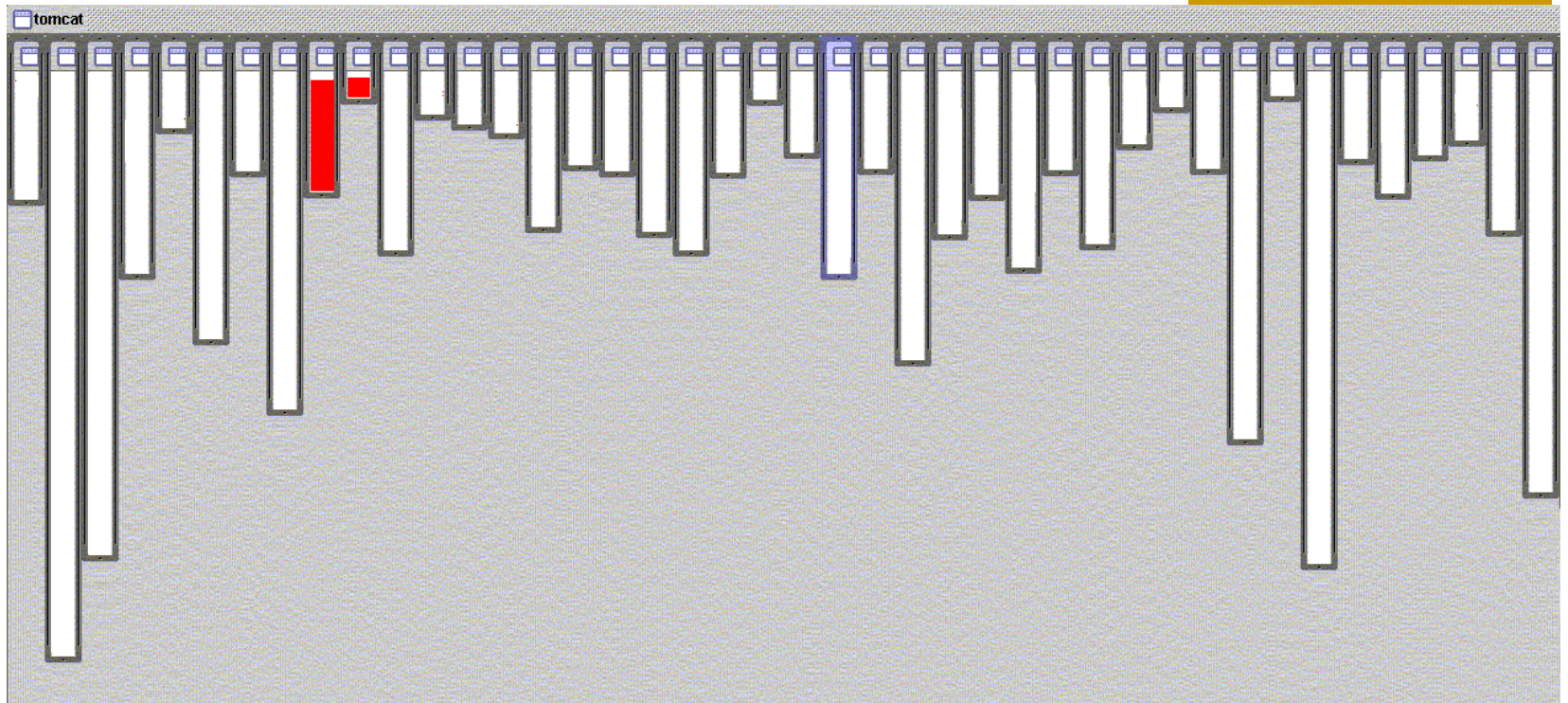
XML parsing



- XML parsing in org.apache.tomcat
 - red shows relevant lines of code
 - nicely fits in one box

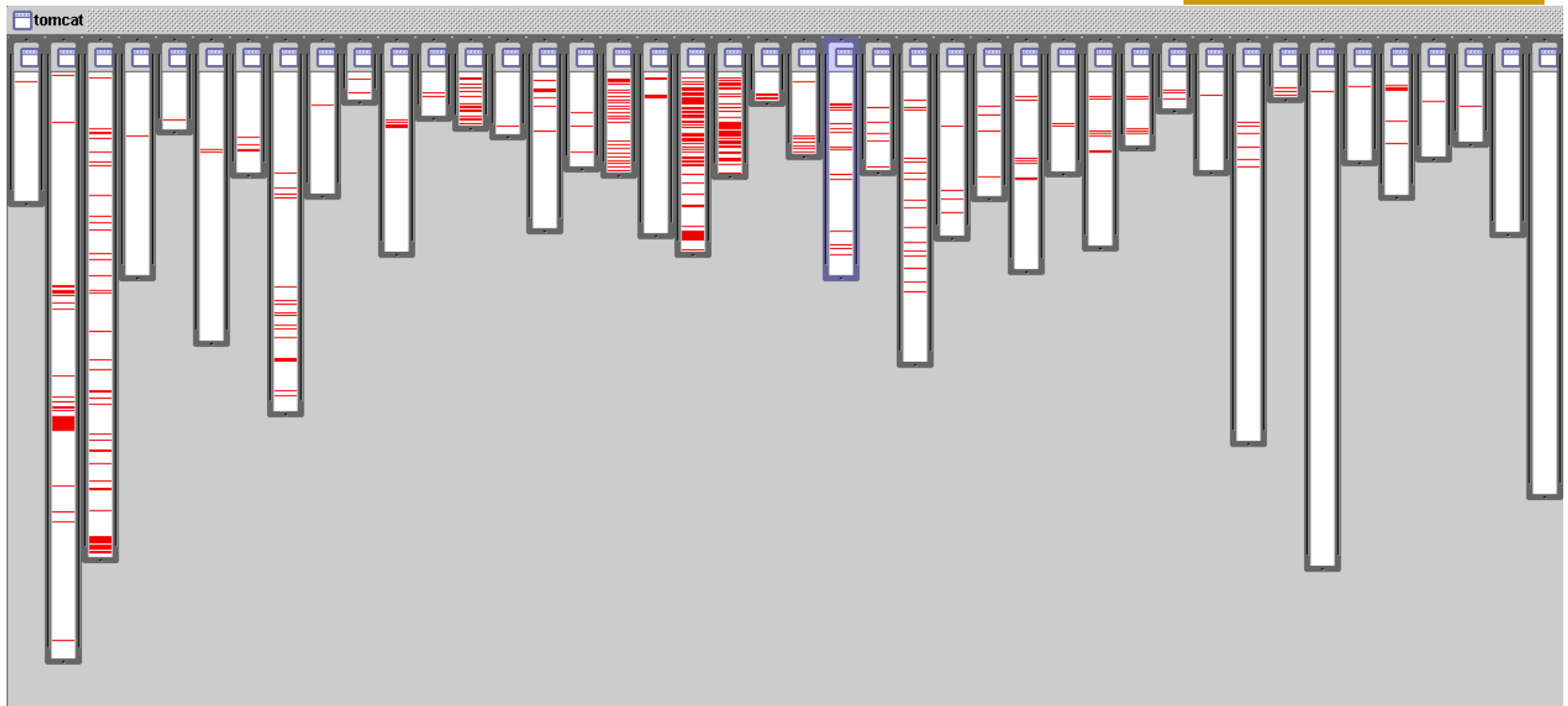
good modularity

URL pattern matching



- URL pattern matching in org.apache.tomcat
 - red shows relevant lines of code
 - nicely fits in two boxes (using inheritance)

logging is not modularized...



- where is logging in org.apache.tomcat
 - red shows lines of code that handle logging
 - not in just one place
 - not even in a small number of places

אילו רק יכולנו...

ApplicationSession

```
public class ApplicationSession extends Session {
    private Application application;
    private boolean isApplicationSession;

    public ApplicationSession(Application application) {
        this.application = application;
        this.isApplicationSession = true;
    }

    public Application getApplication() {
        return application;
    }

    public boolean isApplicationSession() {
        return isApplicationSession;
    }

    // ... other methods ...
}
```

StandardSession

```
public class StandardSession extends Session {
    private boolean isStandardSession;

    public StandardSession() {
        this.isStandardSession = true;
    }

    public boolean isStandardSession() {
        return isStandardSession;
    }

    // ... other methods ...
}
```



SessionInterceptor

```
public class SessionInterceptor implements Interceptor {
    public void intercept(Action action) {
        // ... interceptor logic ...
    }
}
```

StandardManager

```
public class StandardManager implements Manager {
    // ... manager logic ...
}
```

StandardSessionManager

```
public class StandardSessionManager implements SessionManager {
    // ... session manager logic ...
}
```

ServerSession

```
public class ServerSession extends Session {
    // ... server session logic ...
}
```

ServerSessionManager

```
public class ServerSessionManager implements SessionManager {
    // ... server session manager logic ...
}
```



שכתוב מבני

refactoring

שכתוב מבני (refactoring)

- refactoring הוא תהליך של שינוי תוכנה כך שהתנהגותה החיצונית לא תשתנה, אך המבנה הפנימי שלה ישתפר.
- לנקות ולשפר את הקוד בלי להכניס לשגיאות.
- "שיפור התיכון אחרי שהקוד נכתב" סותר לכאורה את העקרונות שמנחים פיתוח תוכנה.
- אבל מכיר בעובדה שבמשך הזמן, שינויים בקוד (למשל להוספת תכונות) גורמים לכך שהמבנה נפגע ומסתבך.
- ב refactoring מבצעים בכל פעם שינוי קטן, טרנספורמציה שמשמרת נכונות (כלומר לא משנה את ההתנהגות החיצונית).
- לאחר כל שינוי יש לבדוק היטב שהשינוי היה נכון - להריץ את אוסף הבדיקות שצברנו.

מקורות

- האנשים שזיהו את חשיבות הרעיון :
 - Ward Cunningham, Kent Beck
- ספר:
 - Martin Fowler, Refactoring, Improving the Design of Existing Code, Addison Wesley 2000. (2nd edition 2005)
- אתר:
 - <http://www.refactoring.com/>
- קשור ל Extreme Programming

למה refactoring ?

- לשפר את תיכון התוכנה – אחרת מבנה המערכת **נשחק** עם הזמן.
- לעשות את התוכנה **קריאה יותר** – הקריאות חיונית למתחזקים.
- לעזור למצוא **שגיאות** – קשה למצוא שגיאה בקוד מסורבל.
- לזרז את כתיבת הקוד – כל השיפורים הללו יקטינו את הזמן שיידרש בהמשך.

מתי לעשות refactoring ?

- כאשר מוסיפים פונקציונליות למערכת - "אם הקוד היה כתוב כך, היה קל יותר להוסיף את הפעולה".
- כאשר צריך למצוא שגיאה - בכל פעם שמסתכלים על קוד ומתקשים להבין אותו יש לבדוק האם ניתן לשפר.
- תוך כדי סקר קוד (Code review)
- באופן כללי, כל פעם שמגלים קוד ש"מריח לא טוב" (code smells). לדוגמא:
 - כפילות בקוד, שרות ארוך מדי, מחלקה גדולה מדי, רשימת פרמטרים ארוכה, סימפטומים של צימוד חזק מדי בין מחלקות....

קטלוג של refactorings

- הספר של Fowler כולל קטלוג של refactorings שכל אחד כולל שם, סיכום קצר, מוטיבציה, תהליך השינוי, ודוגמא.
- חלק מה refactorings ניתנים לאוטומציה ע"י סביבות הפיתוח
 - הכלים מאפשרים לראות כיצד ייראה הקוד אחרי השינוי, ולהחליט (וכן לבטל שינוי שנעשה).
 - הכלים יכולים לציין מתי מובטח שהשינוי נכון (כלומר לא משנה התנהגות).
 - למשל ב eclipse
- אפילו דוגמא פשוטה - שינוי שם של שרות - קשה מאד לשינוי ידני ללא שגיאה. (שינוי גלובלי בעורך טקסט לא יהיה נכון בהכרח).

דוגמאות מקטלוג ה refactorings

- extract method / inline method
- Introduce Explaining Variable
- Move method/Field
- Rename method
- Add/Remove Parameter
- Pull up/Push down Field/Method
- Extract Subclass/Superclass/Interface
- Collapse Hierarchy
- Replace Inheritance with Delegation / vice versa