תוכנה 1

מתי שמרת

ליאור וולף

בית הספר למדעי המחשב

אוניברסיטת תל אביב

על סדר היום

- דרישות זיכרון של מבני נתונים
- בדיקות תוכנה (Testing)
- שכתוב מבני (refactoring)

דרישות זיכרון

קופסאות קטנות?

- מהו יחס גודל הזיכרון בין Integer ל-int?

1. 1:1
2. 1.33:1
3. 2:1
4. ?

חפצים קטנים?

- כמה בתים ב-String המכיל שמונה תווים?

1. 8
2. 16
3. 28
4. ?

גדול יותר

- איזה מהמשפטים הבאים על היחס בין HashSet ל-HashMap נכון

1. פחות פונקציונאליות, קטן יותר
2. יותר פונקציונאליות, קטן יותר
3. פונקציונאליות דומה, גודל דומה
4. ?

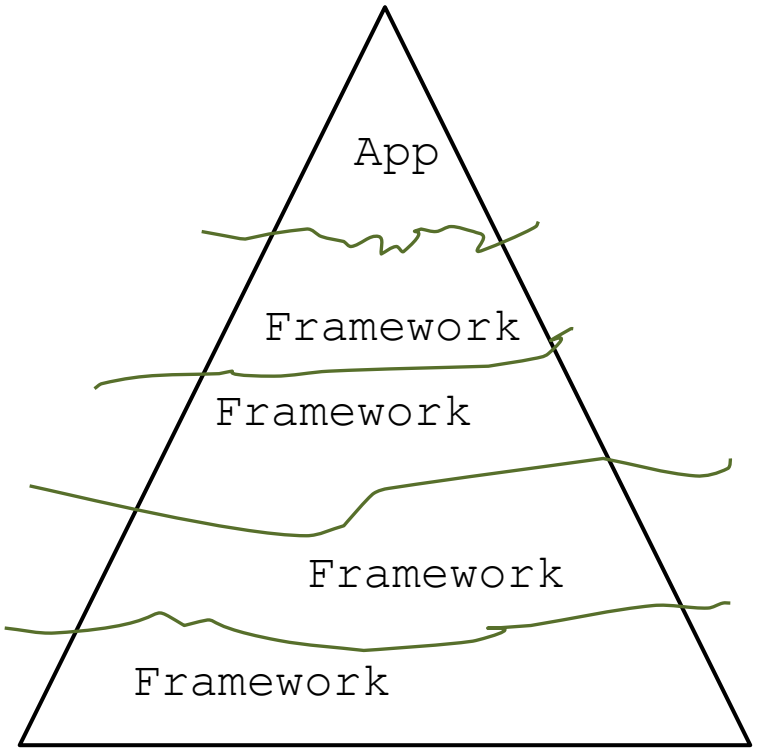
מבני נתונים

- הניחו כי בכל אחד ממבני הנתונים נשמרים שני אלמנטים.
- סדרו את מבני הנתונים לפי גודל הזיכרון הנדרש (קטן לגדול)

ArrayList, HashSet ,LinkedList, HashMap

אפקט הקרחון

- יותר אבסטרקציה = פחות מודעות לעלות

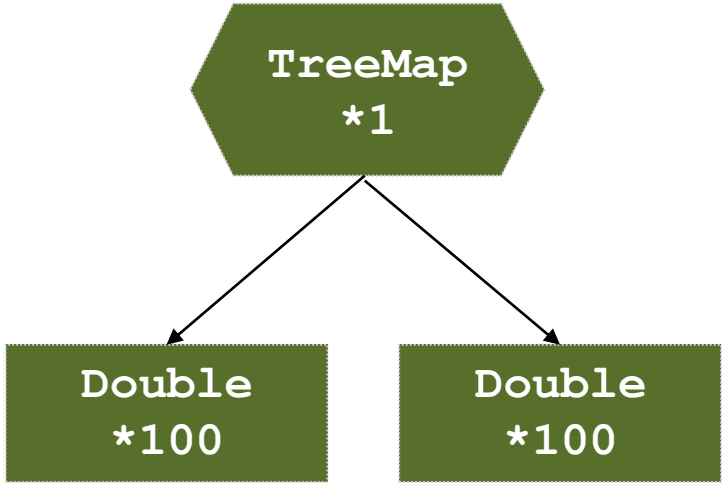


מיתוסים

- אובייקטים (String, HashSet, ….) הם זולים
- ספריות נכתבות ע"י מומחים ולכן הן יעילות (למקרה שלי!)
- ה-JVM וה-GC ידאגו להכל

כמה?

- TreeMap<Double, Double> (100 entries)



3.9KB

2.3KB * 2

~8.6KB

One Double

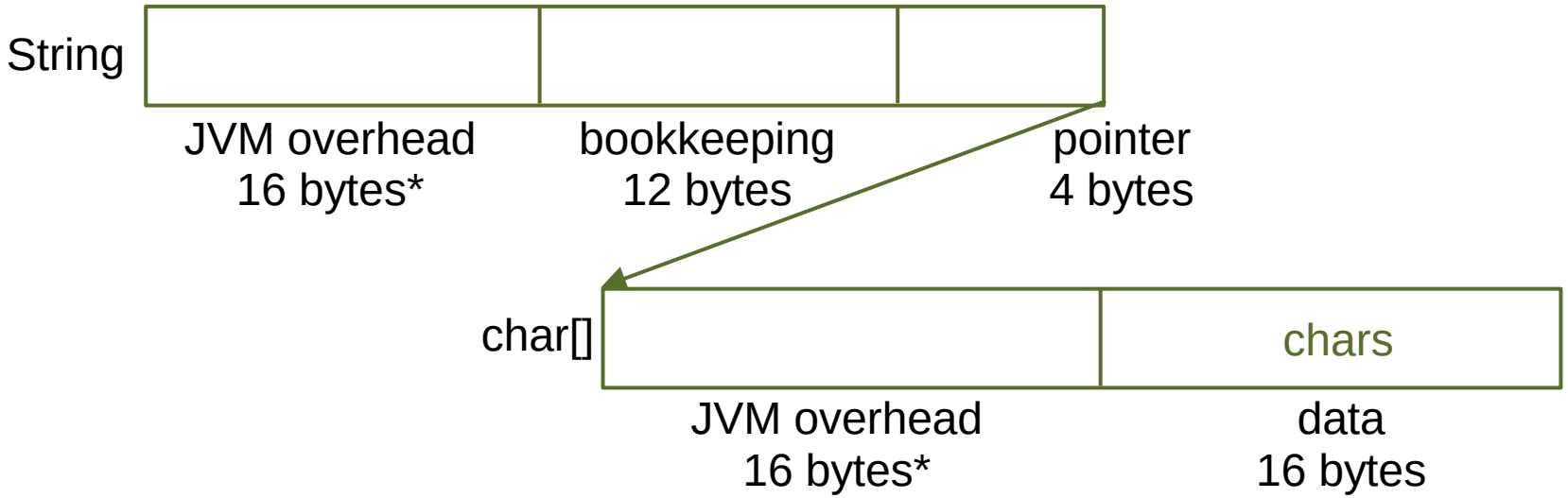
- Double

- 33% actual data
- 67% representation overhead

* JVM dependant

String

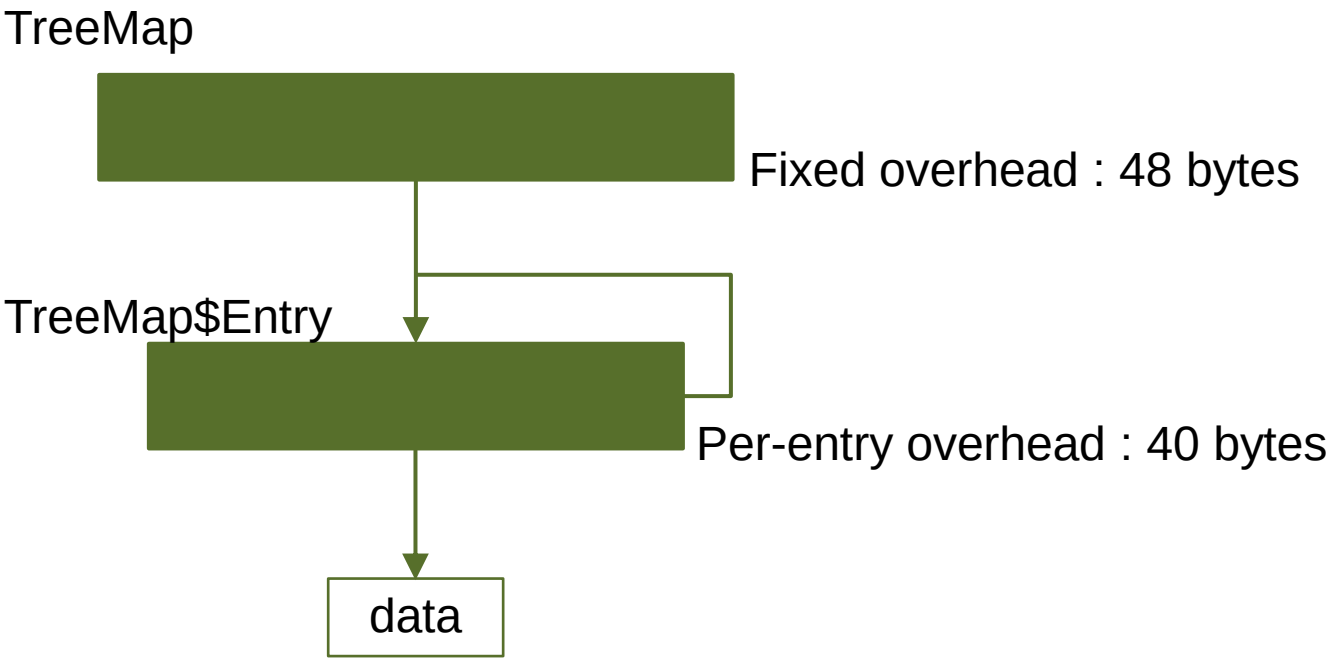
- 8 character string



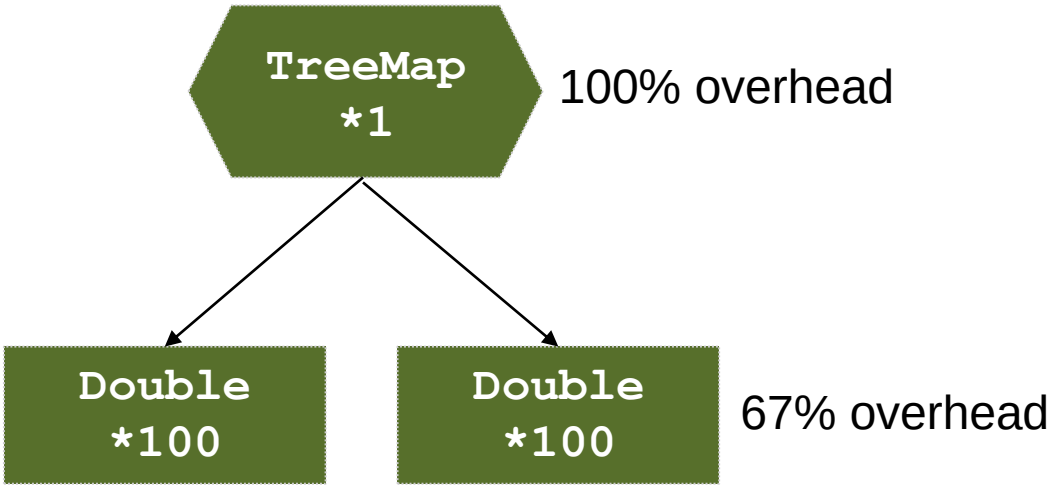
- 25% actual data, 75% representation overhead

TreeMap

- כיצד "מבזבז" מבנה הנתונים את הזיכרון
  - עלויות קבועות ומשתנות



TreeMap<Double, Double>



- 82% overhead overall
- Enables updates while maintaining order
- Does it worth the cost?

Alternative Implementation

```
double[]
x1 = 816 bytes
```

```
double[]
x1 = 816 bytes
```

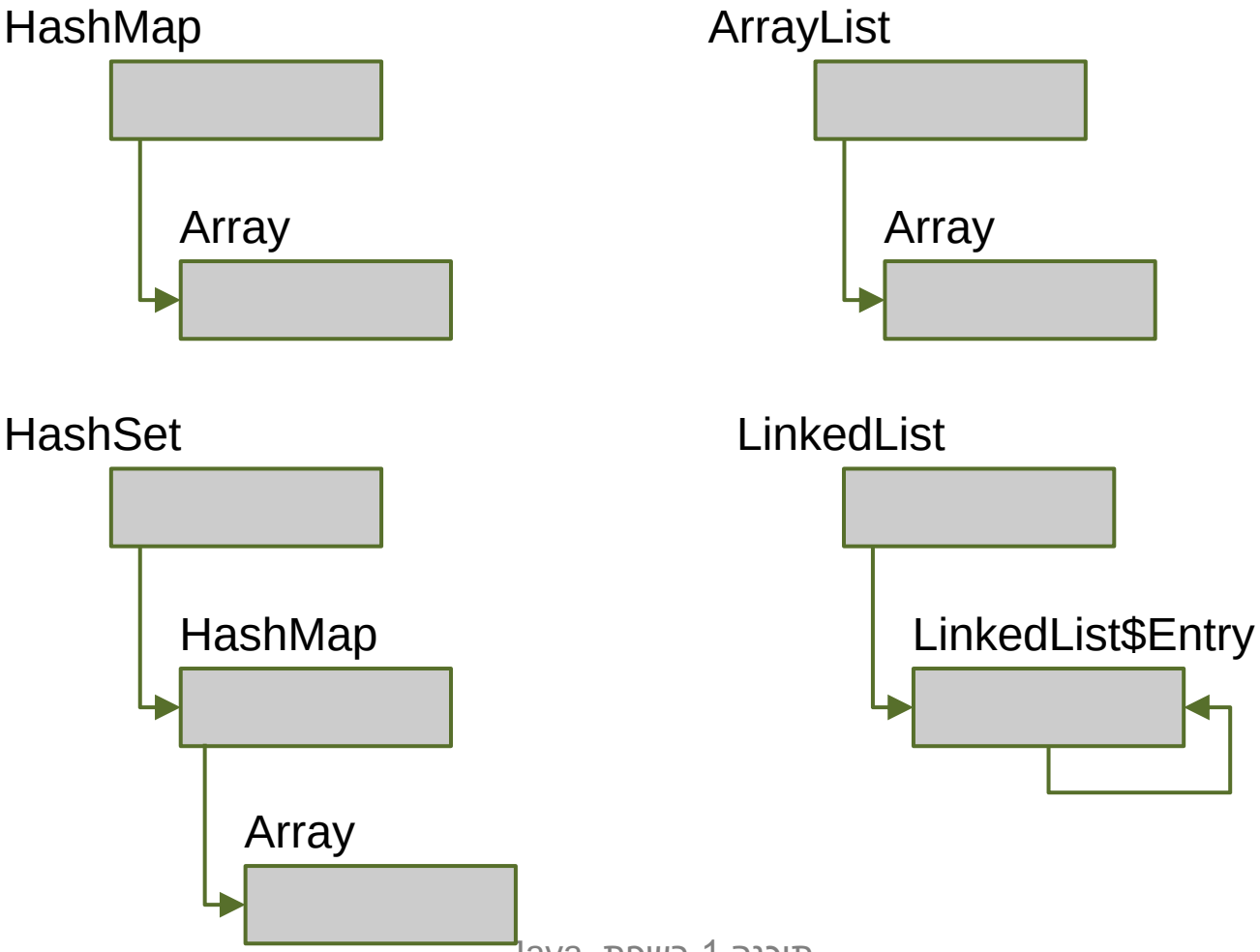
2% overhead

- Binary search against sorted array
- Less functionality

Scalability

- TreeMap
  - High constant cost per element
  - constant overhead 82%
- Alternative
  - Cost per element 16 bytes pure data
  - overhead starts at 2% and quickly goes to 0

האוסף ה(לא כל כך)ריק



המחיר של אוסף "ריק"

	Minimal size	Default size	Default # of slots
LinkedList	48	48	1 sentinel entry
ArrayList	48	48 or 80	0 or 10
HashMap	56 or 120	120	16
HashSet	72 or 136	136	16

סיכום

- אובייקטים הם לא זולים
- שימוש בהאצלה משמעותו עוד זיכרון
- תכנון המערכת צריך לתת מענה לא רק להיבטים הפונקציונאליים של המערכת
  - הספריות הסטנדרטיות של ג'אוה "מעדיפות" זמן על פני מקום
- יש לשים לב במעבר מ 32 ביט ל 64 ביט

בדיקות

איך יודעים שמודול או תוכנית נכונים?

- **אימות**: תהליך שמיועד לוודא באופן פורמאלי נכונות של מודול או תוכנית ביחס לחוזה
- **אימות פורמאלי אוטומאטי** אינו אפשרי במקרה הכללי. למרות זאת קיימים כלים שלעיתים מצליחים עבור מקרים פרטיים.
- **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 ביטחוננו בנכונותה
  - לא מבטיח נכונות, אבל יותר טוב מכלום, ומועיל מאוד באופן מעשי להקטנת מספר הפגמים

אל תירה בשליח

- כאשר המכונית 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
- **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)
  - הקוד נבדק מול החוזה, לא תלוי במימוש
  - אותו סט בדיקות תקף לכל המימושים של מנשק מסוים, גם העתידיים, ובפרט לשינויים ותיקונים במימוש הנוכחי
- **בדיקות כיסוי** (coverage tests או glass-box tests)
  - דואגות שבזמן הבדיקות, כל פיסת קוד תרוץ, ובמקרים מסוימים, תרוץ ביותר מכמה 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
- **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

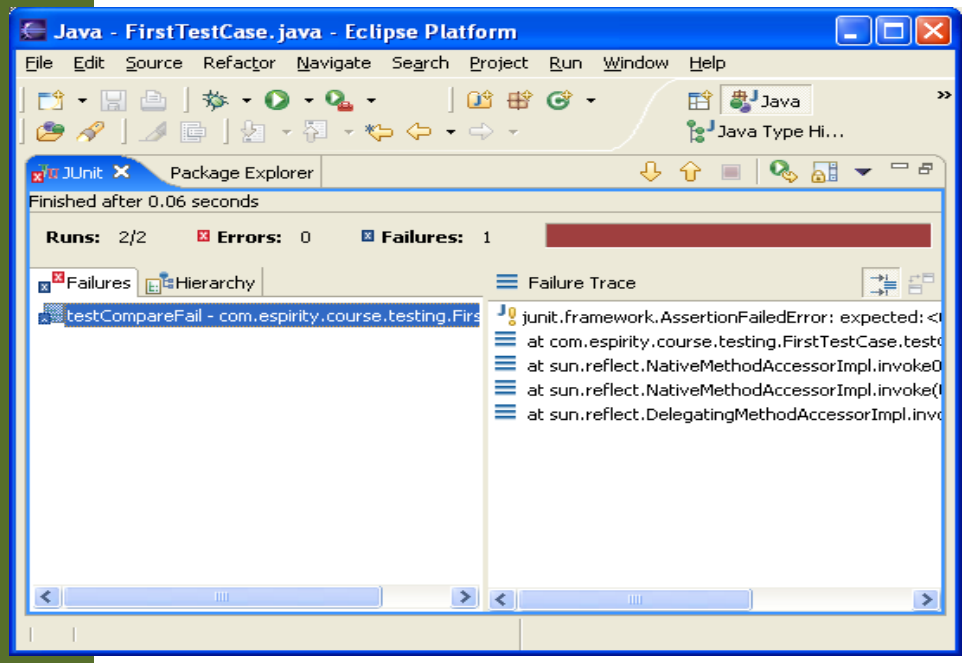
בדיקות רגרסיה

- בכל פעם שמגלים פגם בתוכנה (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 חריגים ואילו

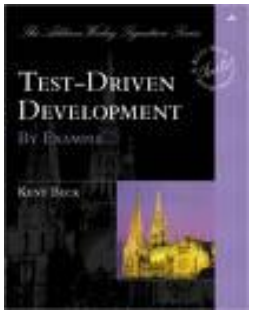
פיתוח מונחה בדיקות

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 complition 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 complition 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 complition of operation
            ...unlock the object
    }
}
```

The wrong way(6)

- **Persistence ? Cache consistency ?**

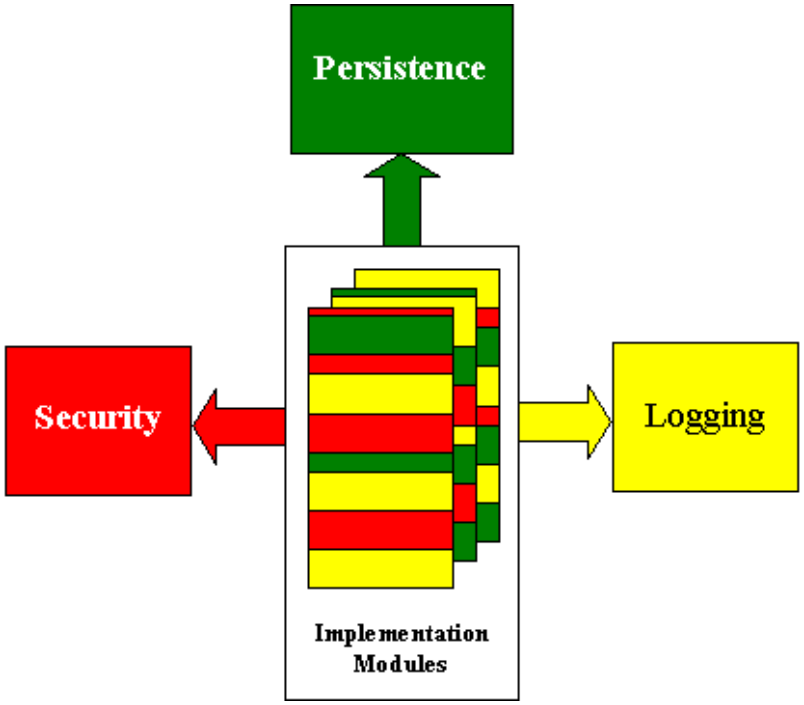
```
public class SomeBusinessClass extends OtherBusinessClass {

     // 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 complition of operation
             ...unlock the object
     }
     public void save(PersitanceStorage ps) {...}

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

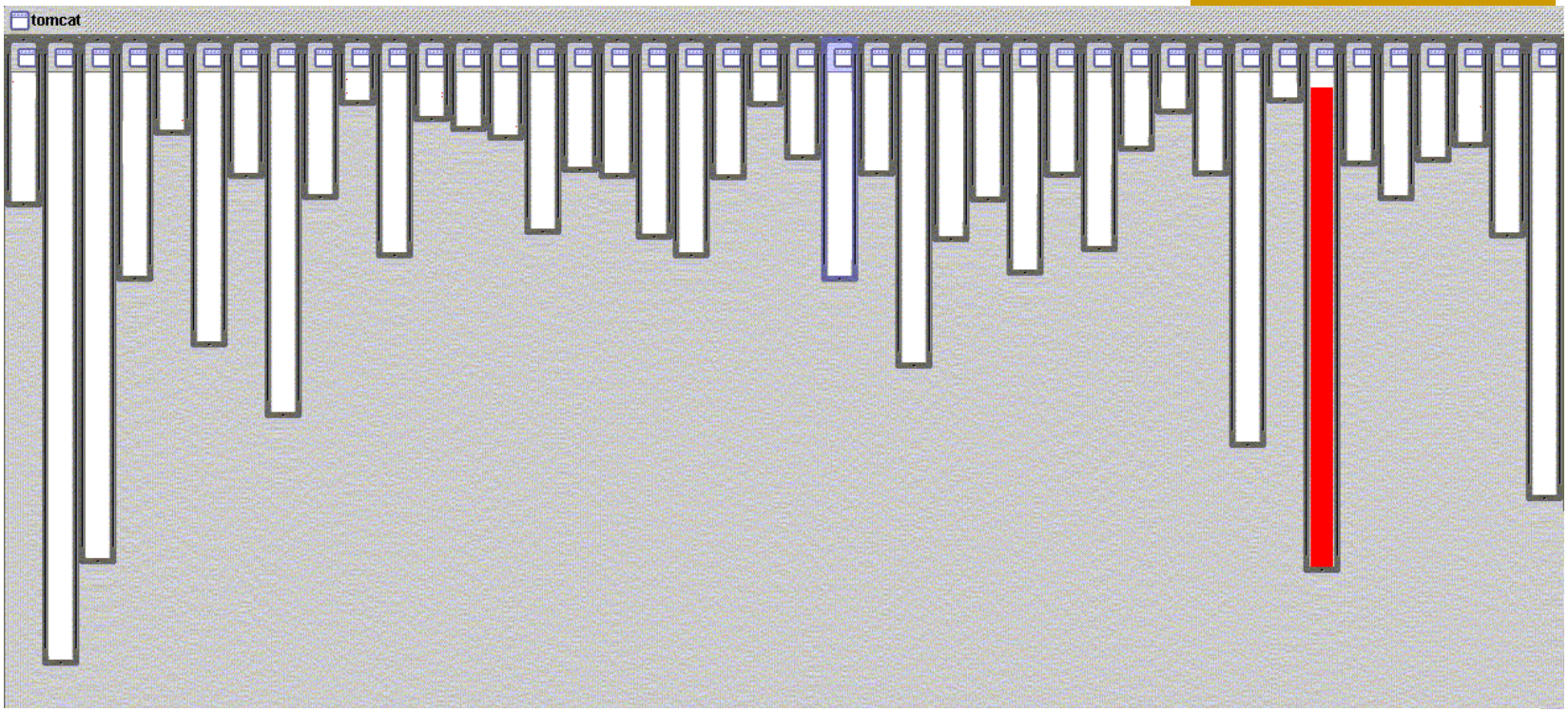
מה קיבלנו?



בלאגן בשתי רמות:

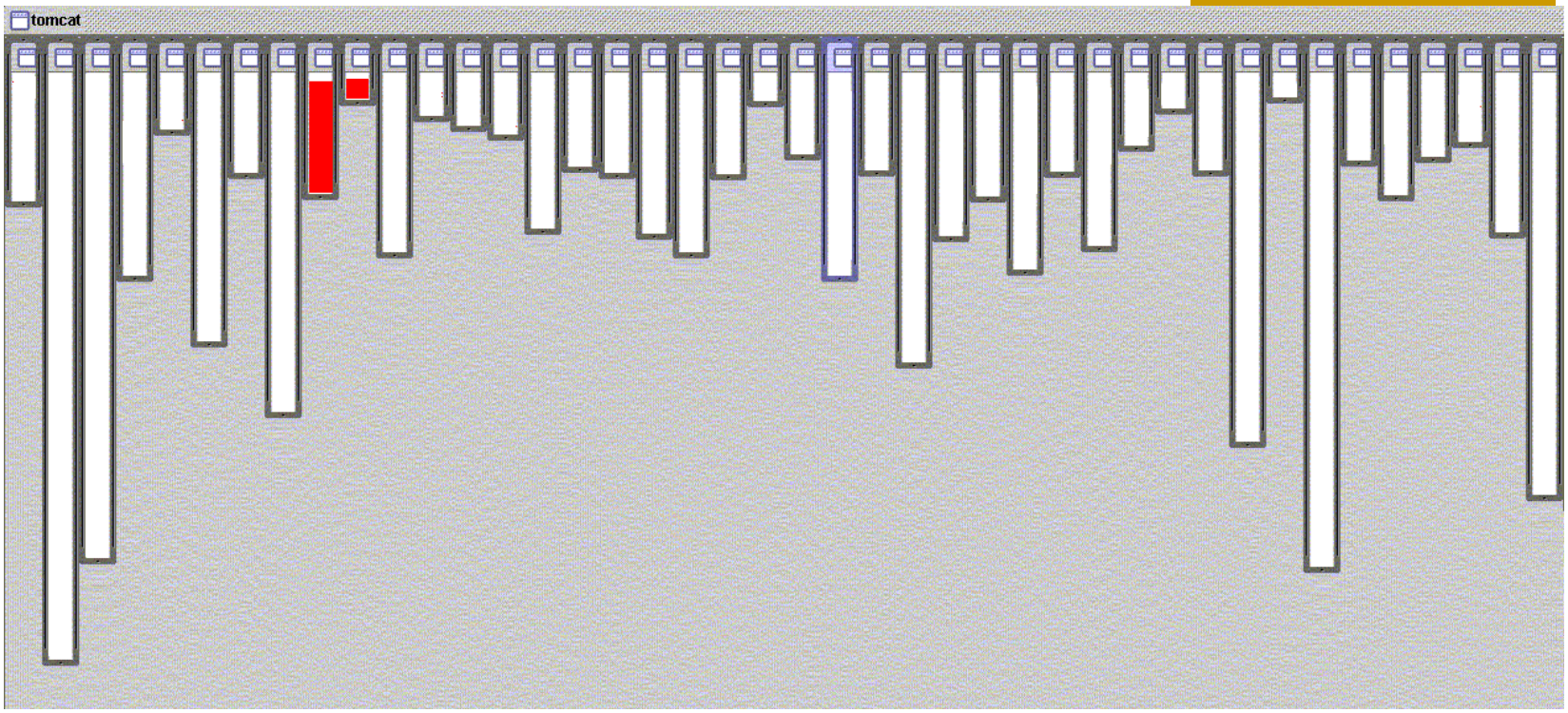
- ברמת המיקרו (השרות הבודד):
  - Code Tangling
  - הוא כבר לא עושה "רק משהו אחד" - לא מודולרי
  - ראו תרשים =>
- ברמת המאקרו (מערכת התוכנה):
  - Code Scattering
  - שכפול קוד, קטעי קוד קשורים אינם מופיעים יחד
  - ראו תרשימים גם בשקפים הבאים
- שבירת המודולריות נוצרת בגלל אופי הספק-לקוח של תכנות מונחה עצמים

good modularity
XML parsing



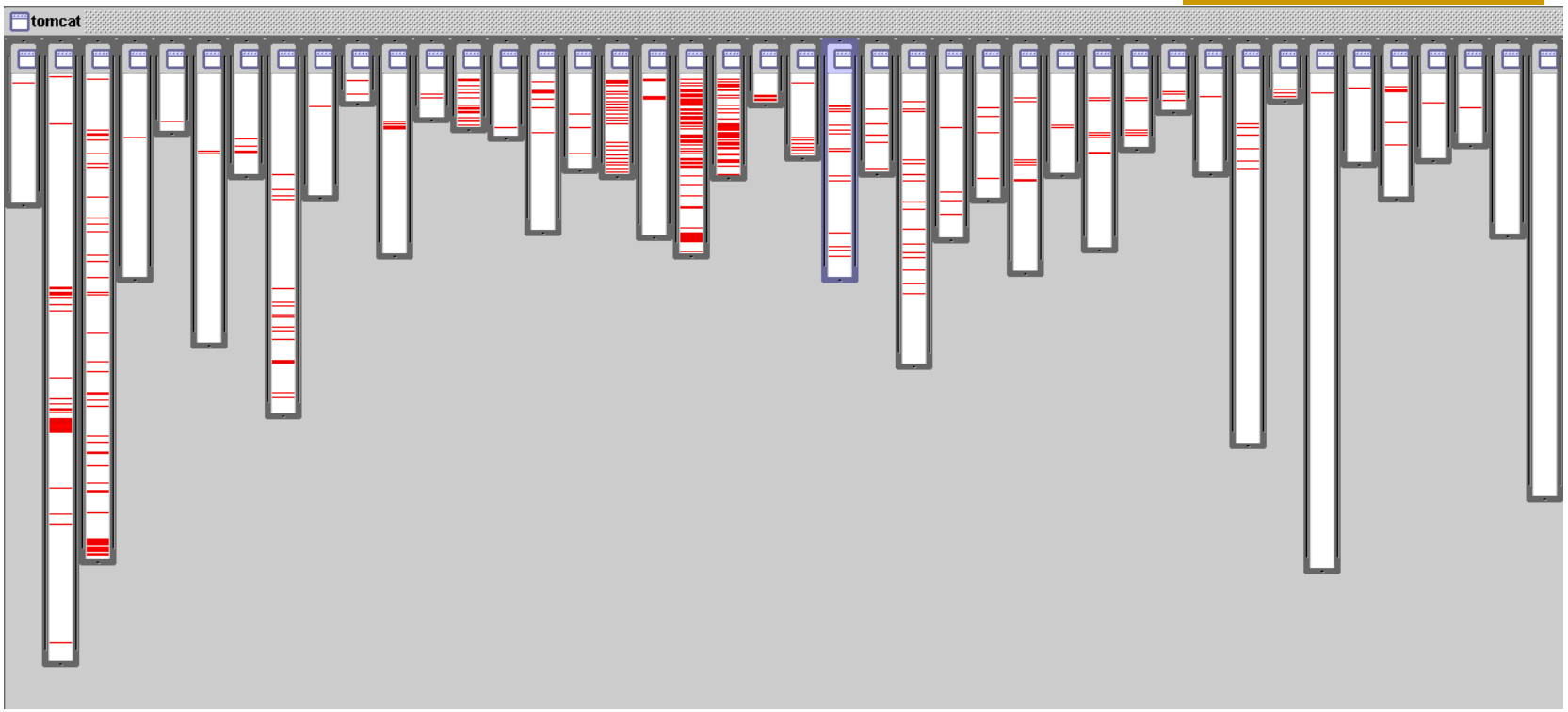
- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

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

StandardSession

SessionInterceptor

StandardManager

StandardSessionManager

ServerSession

ServerSessionManager

שבירת המודולריות

- נזכיר 3 **גישות** לפתרון הבעיה:
  - מעבר לשימוש **ברכיבים** (components) במקום עצמים
    - כגון: Servlets או EJB's
    - חסרון: Domain Specific Framework
  - פתרונות ברמת **שפת התכנות ותבניות העיצוב**:
    - כגון: Mixin או Dynamic Proxy
    - חסרון: דורש "תחזוקה ידנית" של העיצוב
  - מעבר **לשפת תכנות** בפרדיגמה התומכת ביחסים נוספים בין מחלקות
    - כגון: AspectJ או שפת E
    - חסרון: לימוד שפה חדשה

שכתוב מבני

refactoring

שכתוב מבני (refactoring)

- refactoring הוא תהליך של שינוי תוכנה כך שהתנהגותה החיצונית לא תשתנה, אך המבנה הפנימי שלה ישתפר.
- "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 ?

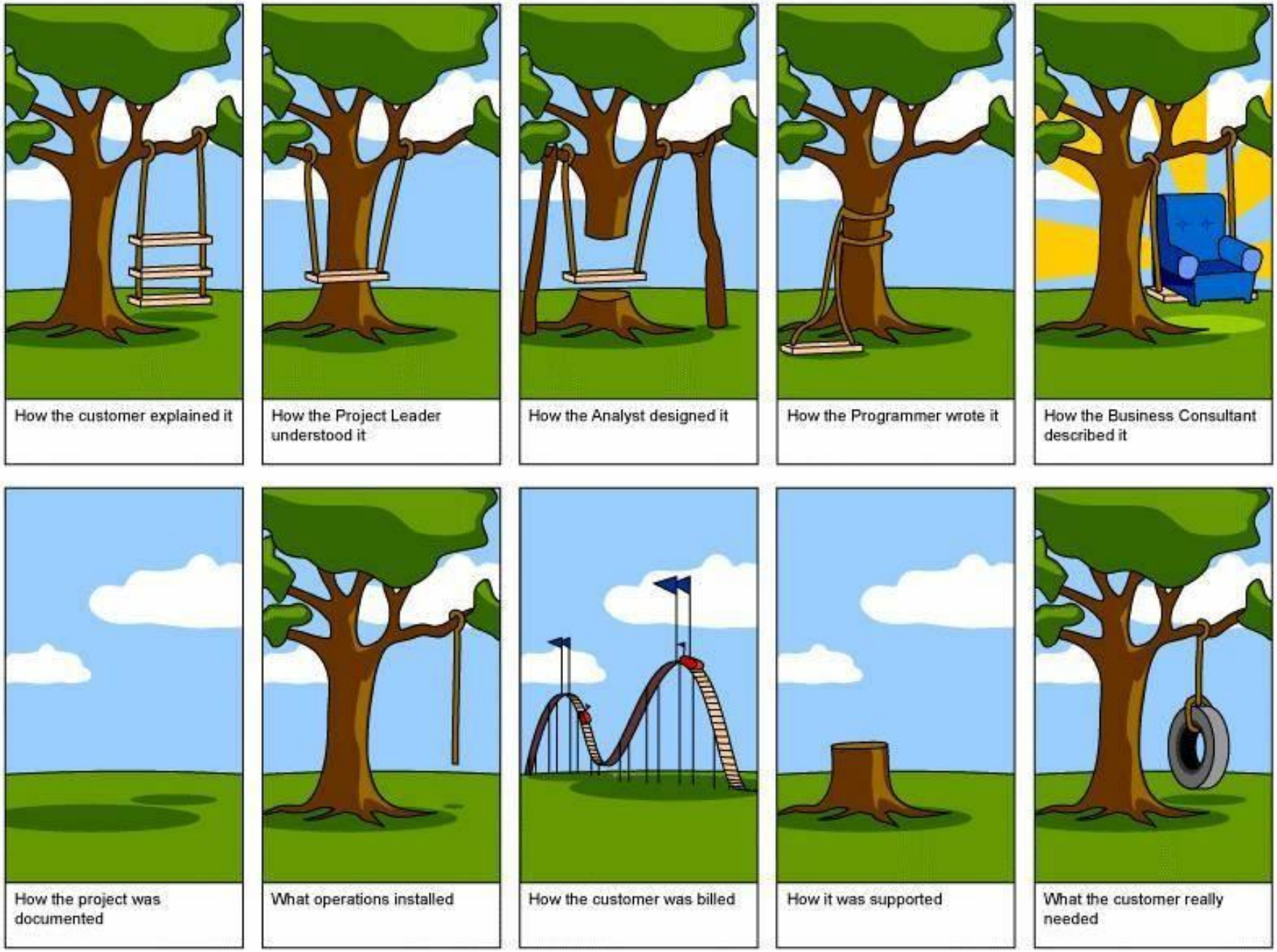
- כאשר מוסיפים פונקציונליות למערכת - "אם הקוד היה כתוב כך, היה קל יותר להוסיף את הפעולה".
- כאשר צריך למצוא שגיאה - בכל פעם שמסתכלים על קוד ומתקשים להבין אותו יש לבדוק האם ניתן לשפר.
- תוך כדי סקר קוד (Code review)
- באופן כללי, כל פעם שמגלים קוד ש"מריח לא טוב" (code smells). לדוגמא:
  - כפילות בקוד, 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).
- 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


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed