תוכנה 1 בשפת Java

שיעור מספר 14: "מה להנדסה ולזה?"

שחר מעוז

בית הספר למדעי המחשב
אוניברסיטת תל אביב

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 היא תחום הנדסי העוסק בכל ההיבטים של יצירת מערכות תוכנה.

- קיימים קורסי בחירה מתקדמים בחוג המתמקדים בשלבים השונים
- הדיון אינו ספציפי לתכנות מונחה עצמים

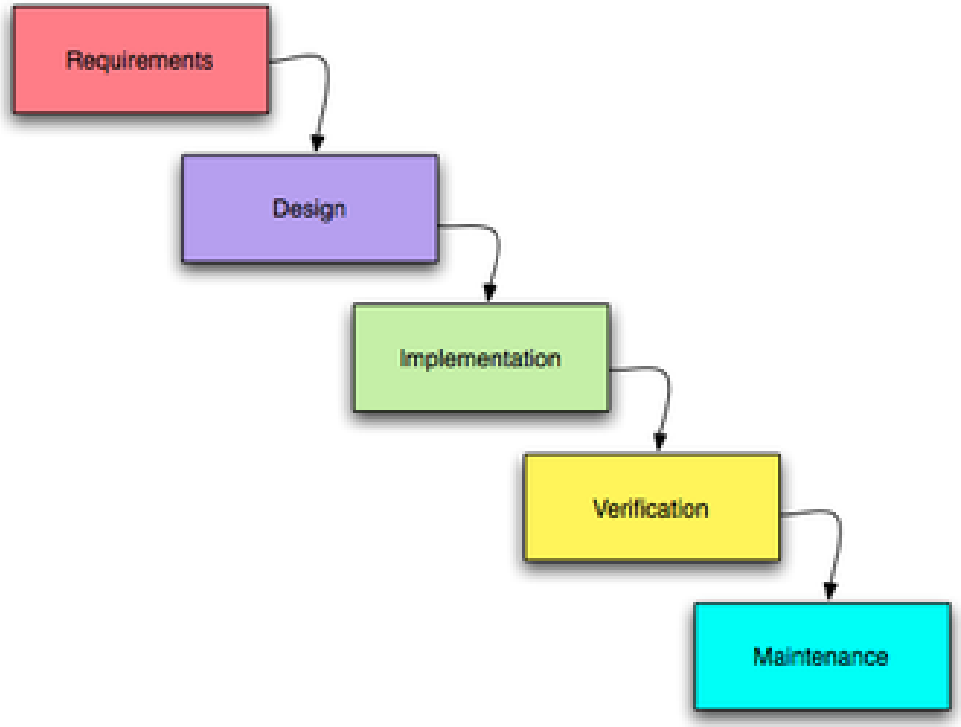
מחזור החיים של תוכנה

- ניתוח דרישות (requirements analysis)
- תיכון (design)
- מימוש (construction, implementation or coding)
- שילוב (integration)
- בדיקות וניפוי שגיאות (testing and debugging aka: verification)
- בדיקות קבלה (acceptance)
- ייצור (production)
- הפצה והתקנה (deployment and installation)
- תחזוקה ושינויים (maintenance)

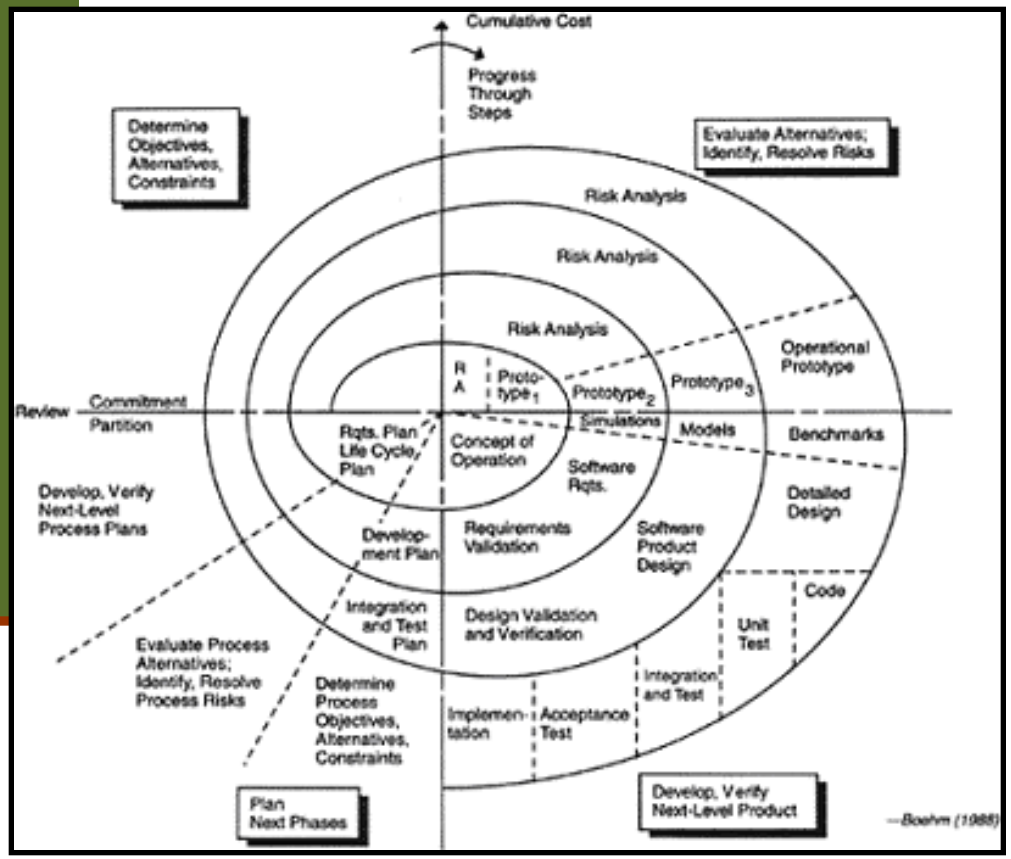
- התייחסות מיוחדת למקרה שמערכת התוכנה היא חלק ממערכת ממוחשבת הכוללת חומרה ותוכנה.

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 ואיטרטיבי.

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

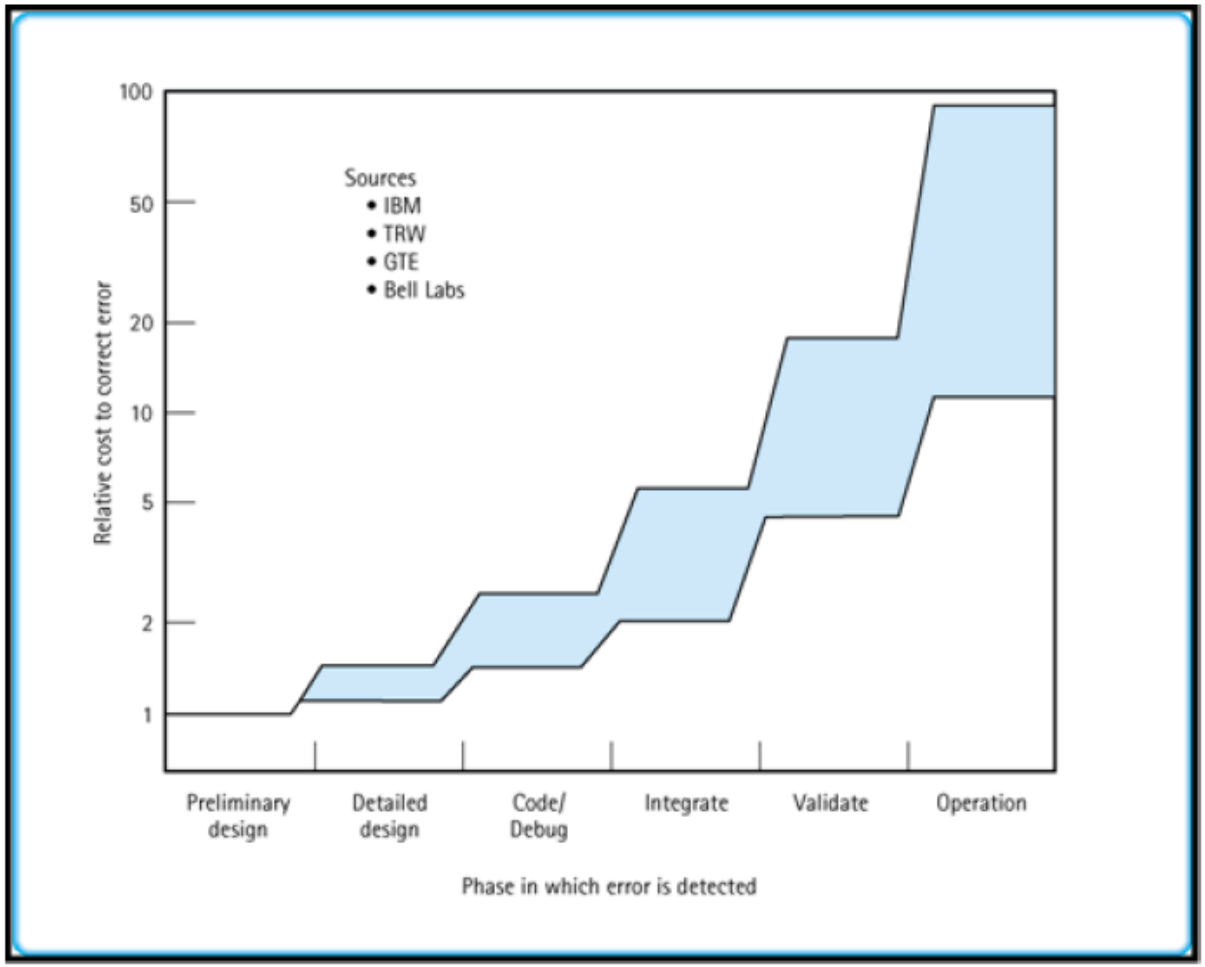
מחירן של טעויות

- **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)


How the customer explained it

תבניות תיכון (design patterns)

תבניות תיכון - מוטיבציה

- בחיי היום יום אנחנו מתארים דברים תוך שימוש בתבניות חוזרות:
  - "מכונית א' היא כמו מכונית ב', אבל יש לה 2 דלתות במקום 4"
  - "אני רוצה ארון כמו זה, אבל עם דלתות במקום מגרות"

- גם בפיתוח תוכנה, אנחנו יכולים להסביר כיצד לעשות משהו ע"י התייחסות לדברים שעשינו בעבר, ובצורה כזאת להקל על התקשורת עם עמיתים.
  - "נאחסן את המבנה בעץ בינרי, ונבצע חיפוש לרוחב"

הגדרות מהספרות:

- "Design Patterns are **recurring solutions to design problems** you see over and over." [Alpert, Brown, Wof, 1998].
- "Design Patterns constitute a set of rules describing how to **accomplish certain tasks** in the realm of software development." [Pree, 1994].
- "A pattern address a **recurring design problem** that arises in specific design situations and presents a solution to it." [Buschmann et al, 1996].
- "Patterns identify and specify **abstractions** that are above the level of single classes and instances, or of components." [Gamma et al, 1993].

מקורות

- המושג נעשה פופולרי בעקבות הספר שמכונה GoF – דוגמאות הקוד ב C++

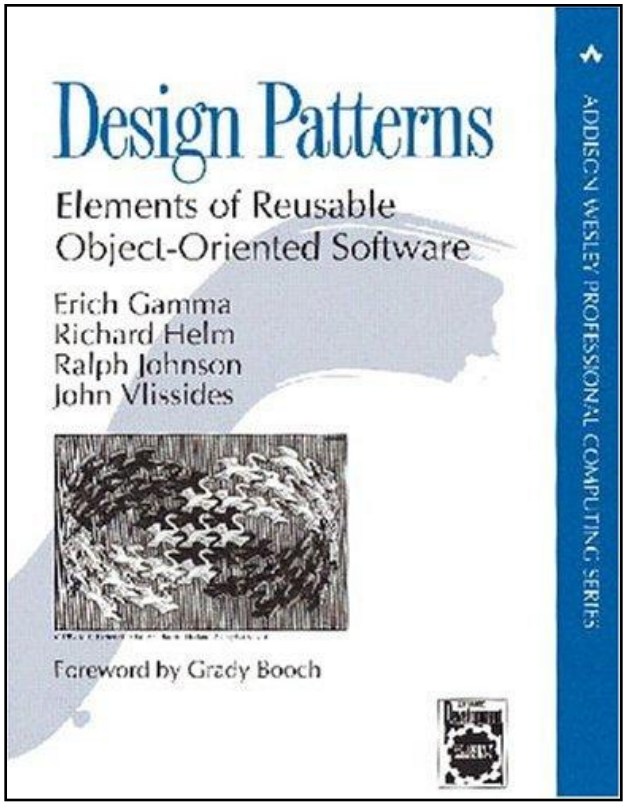
Design Patterns: Elements of Reusable Object-Oriented Software

By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Published by Addison Wesley Professional.

Series: Addison-Wesley Professional Computing Series.

ISBN: 0201633612; Published: Oct 31, 1994; Copyright 1995; Pages: 416; Edition: 1st.



מקורות ב Java

- קיימים כמה מקורות לתבניות עיצוב עם דוגמאות בשפת Java כגון:

- **The Design Patterns Java Companion**
  James W. Cooper
  http://www.patterndepot.com/put/8/JavaPatterns.htm

- **Thinking in Patterns with Java**
  Bruce Eckel
  http://www.mindview.net/Books/TIPatterns/

- תבניות עיצוב רבות אומצו ע"י כותבי הספריות הסטנדרטיות של Java
  - בעיקר (אבל לא רק) בספריות GUI

סיווג תבניות

- הספר של GoF מציג 23 תבניות שמחולקות לשלוש משפחות לפי המטרה שלהן:
  - **תבניות ליצירה Creational**: נוגעות לתהליך היצירה של עצמים.
  - **תבניות מבנה Structural**: עוסקות בהרכבה של מחלקות ועצמים.
  - **תבניות התנהגות Behavioral**: מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים ומחלקים אחריות.

- קלסיפיקציה נוספת מתייחסת לתחום העיסוק של תבנית – מחלקות או עצמים.
- בנוסף, ניתן להתייחס **לקבוצות של תבניות** שמופיעים בדרך כלל ביחד:
  - כאלה שמהווים חלופות שונות לפתרון בעיות דומות

  ולהיפך –
  - פתרונות דומים לבעיות שונות

- לתבניות חשיבות גדולה באפיון הבעיות
- חלק מהתבניות ראינו במהלך הקורס – בהקשר רחב או מקומי

סיווג תבניות

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

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 ?**

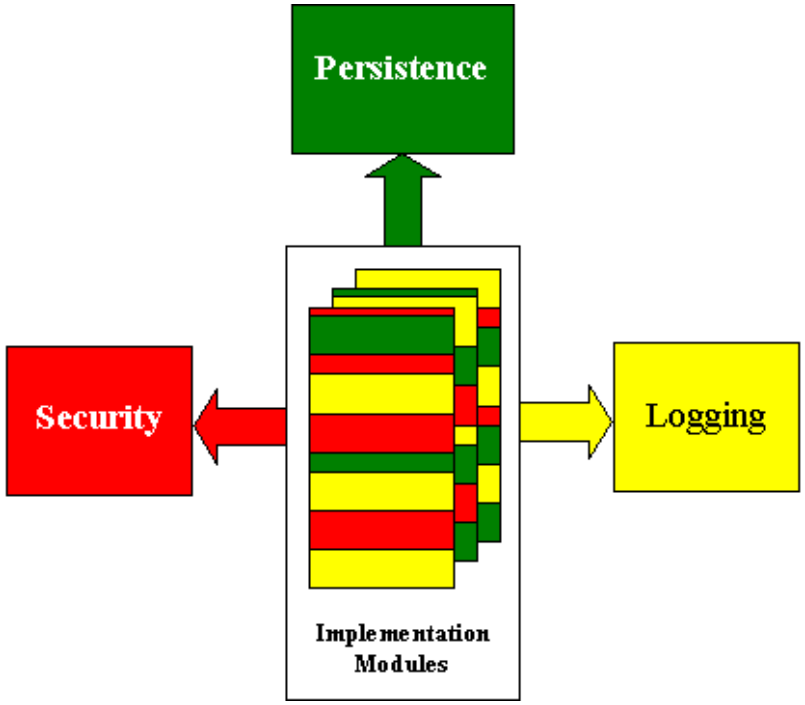
```
public class SomeBusinessClass extends OtherBusinessClass {

      // 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) {...}
}
```

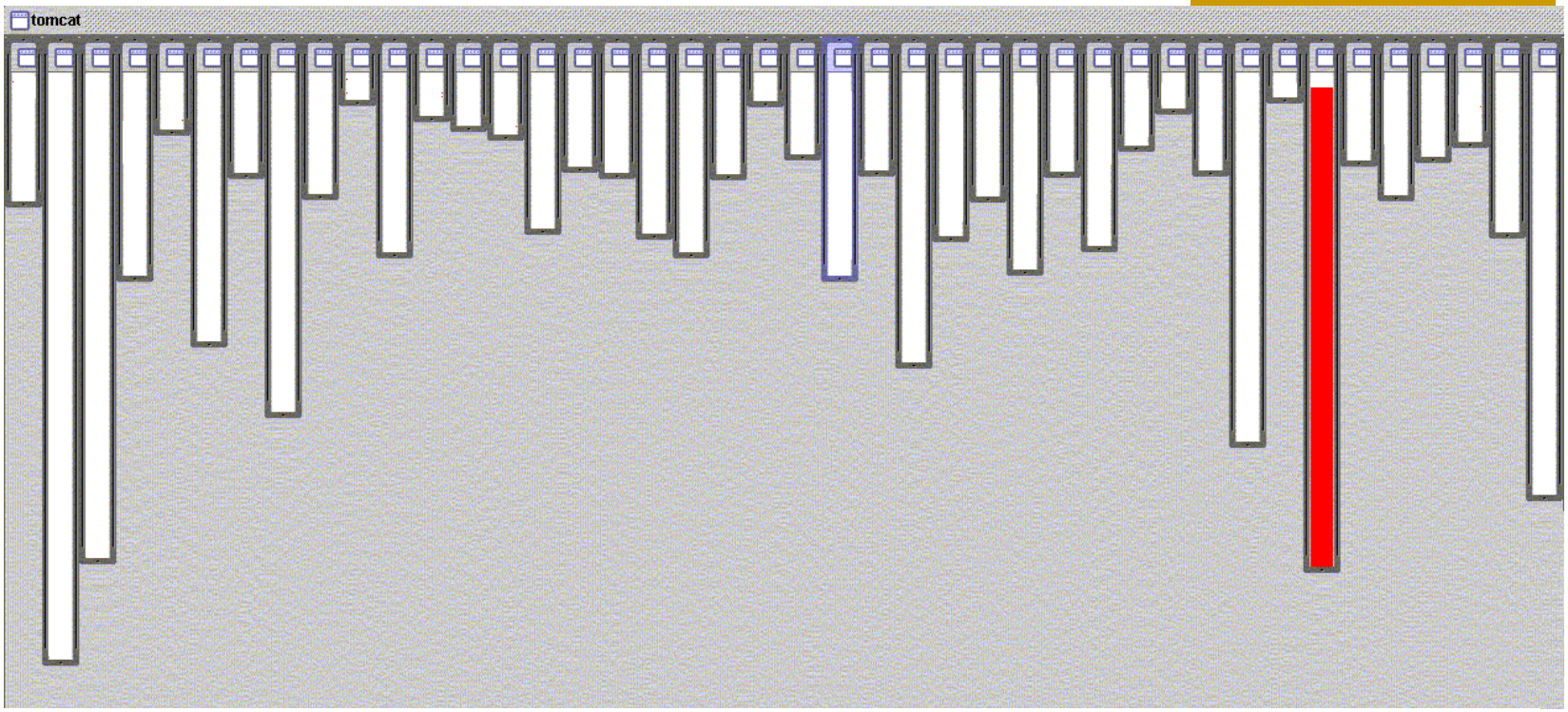
מה קיבלנו?



בלאגן בשתי רמות:

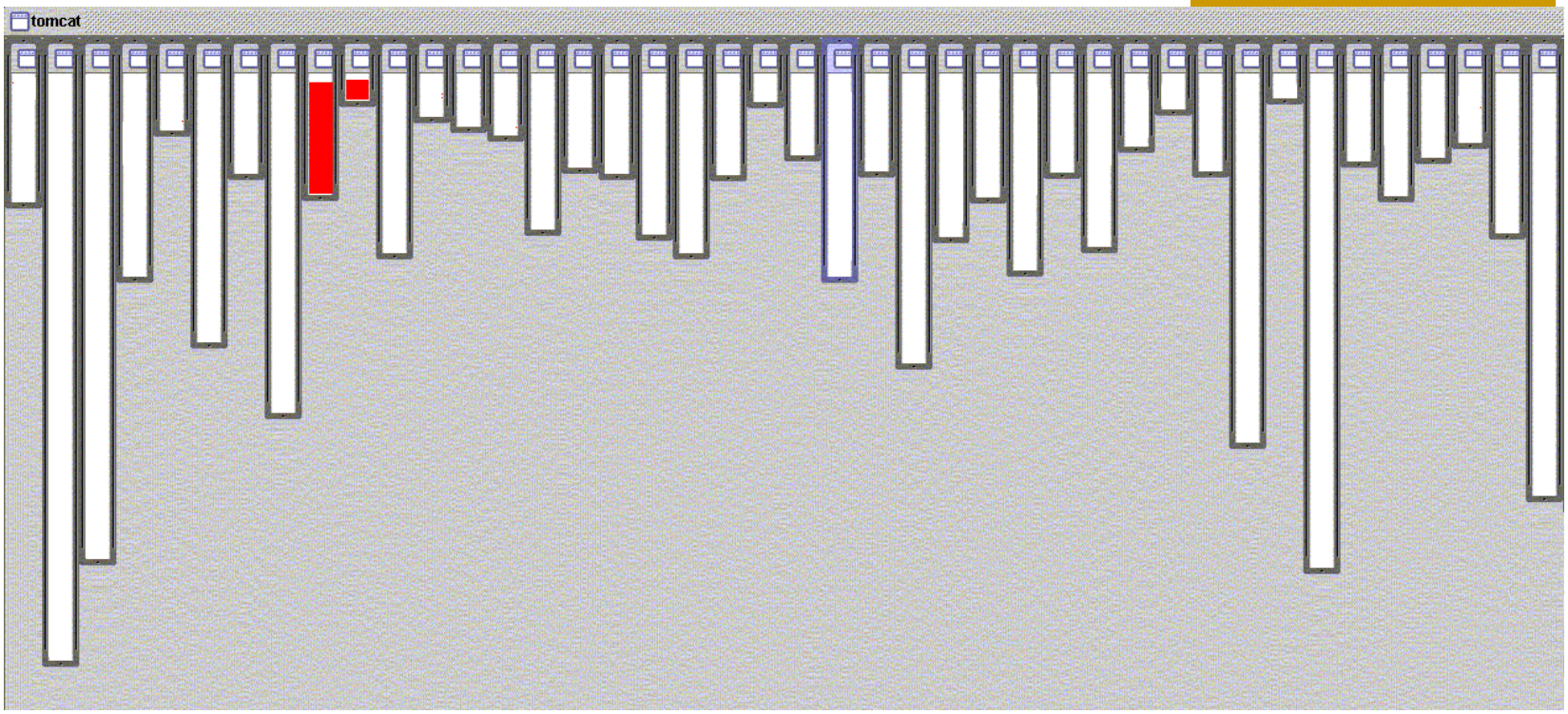
- ברמת המיקרו (השרות הבודד):
  - Code Tangling
  - הוא כבר לא עושה "רק משהו אחד" - לא מודולרי
  - ראו תרשים =>
- ברמת המאקרו (מערכת התוכנה):
  - Code Scattering
  - שכפול קוד, קטעי קוד קשורים אינם מופיעים יחד
  - ראו תרשימים גם בשקפים הבאים
- שבירת המודולריות נוצרת בגלל אופי הספק-לקוח של תכנות מונחה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

Good modularity
XML parsing



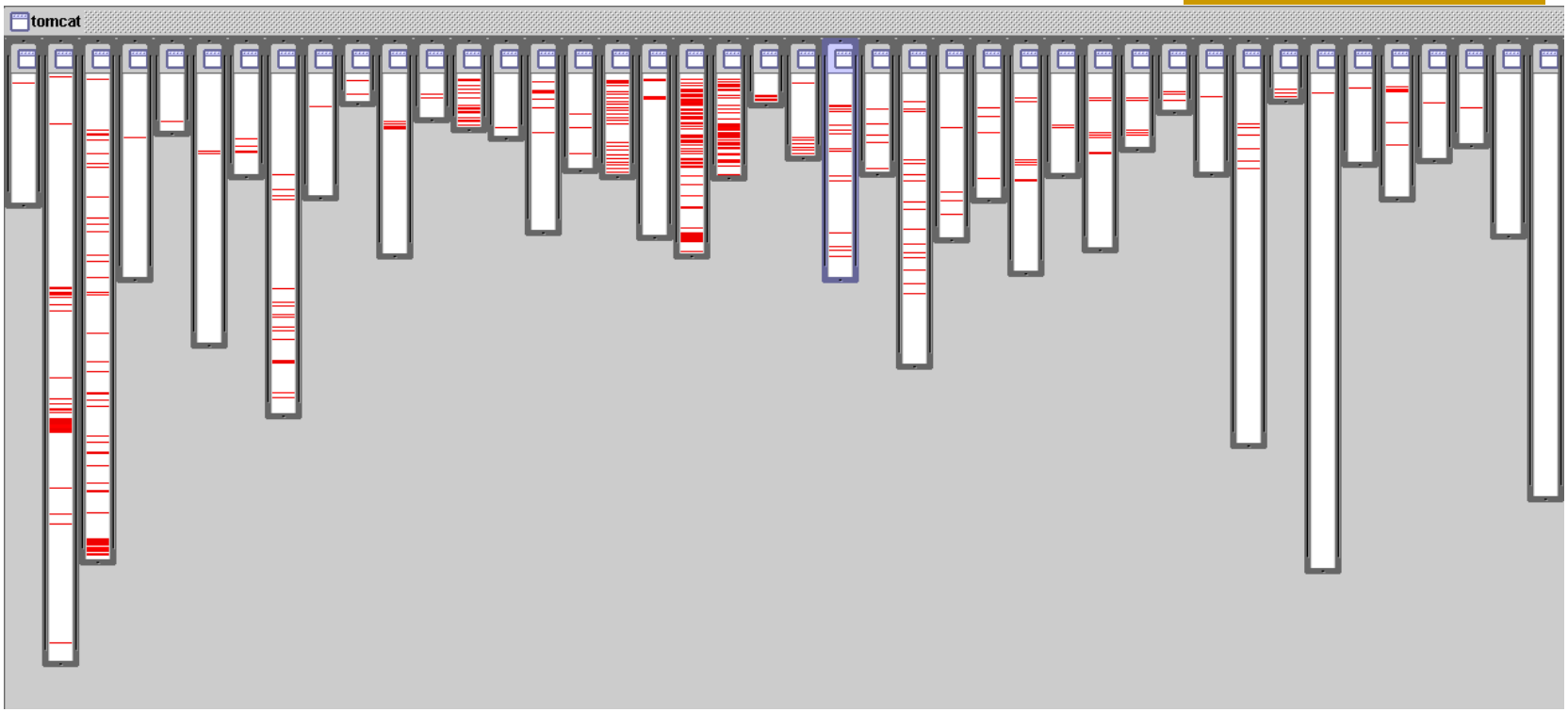
- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

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.
- 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/

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