
פיתוח מערכות תוכנה מבוססות Java

אוהד ברזילי

ohadbr@mta.ac.il

המכללה האקדמית תל אביב – יפו

**Based on: K. Beck: Extreme Programming Explained.
E. M. Burke and B.M. Coyner: Java Extreme Programming Cookbook.
L. Crispin and T. House: Testing Extreme Programming**
<http://www.extremeprogramming.org>

**And slides of: Kent Beck and Ward Cunningham,
Laurie Williams, Vera Peeters and Pascal Van Cauwenberghe,
Ian Sommerville:**
<http://www.comp.lancs.ac.uk/computing/resources/lanS/SE7/Presentations/index.html>

The Rules

1. On Site Customer

- At least one **customer is always present**.
- This **customer is available full-time** to:
 - Answer questions about the system.
 - Negotiate the timing and scheduling of releases.
 - Make all decisions that affect business goals.
- The customer writes functional tests (with the help of **Development**).

2. Pair Programming

- All programming is done with **two coders at the same machine**.
 - The **programmers must share** one mouse, keyboard, screen, etc.
- At least two people are always intimately familiar with every part of the system, and every line of code is reviewed as it's written.

Here is how pair programming works:

- You pick out a user story for your next task
 - A user story is a requirement from the customer.
 - Stories are typically written on index cards, and the customer decides which stories are the most important
- You ask for help from another programmer.
- The two of you work together on a small piece of functionality:
 - Try to work on small tasks that take a few hours.
 - After the immediate task is complete, pick a different partner or offer to help someone else

Pair Programming

<http://www.pairprogramming.com/>

Pair-programming has been popularized by the eXtreme Programming (XP) methodology



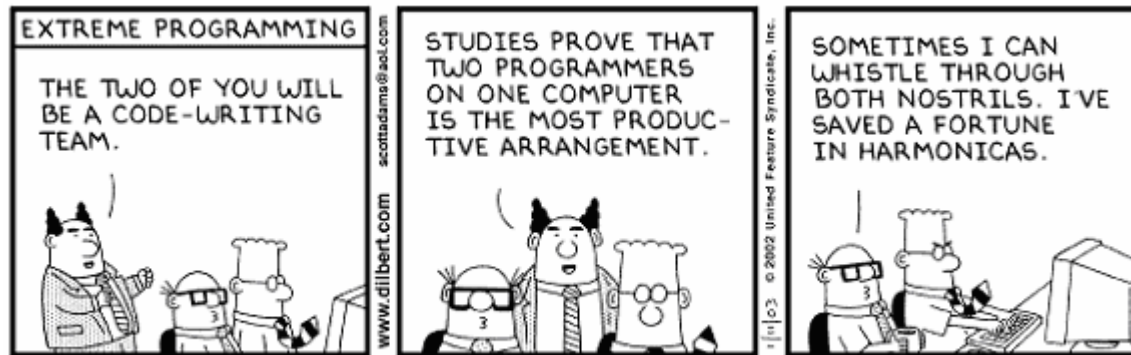
With pair-programming:

- Two software engineers work on one task at one computer
- One engineer, **the driver**, has control of the keyboard and mouse and creates the implementation
- The other engineer, **the navigator**, watches the driver's implementation to identify defects and participates in on-demand brainstorming
- The roles of driver and observer are periodically rotated between the two software engineers

Research Findings to Date

- Strong anecdotal evidence from industry
 - “We can produce near defect-free code in less than half the time.”
- Empirical Study
 - Pairs produced higher quality code
 - 15% **less defects** (difference statistically significant)
 - Pairs completed their tasks in about half the time
 - 58% of elapsed **time** (difference not statistically significant)
 - Most programmers reluctantly embark on pair programming
 - Pairs **enjoy** their work more (92%)
 - Pairs **feel more confident** in their work products (96%)
- India Technology Company
 - 24% increase in **productivity** (KLOC/Person-Month)
 - 10-fold reduction in **defects**.

Pair Programming



Copyright © 2003 United Feature Syndicate, Inc.

3. Coding Standards

- Agree upon **standards for coding styles**.
- Promotes **ease of understanding and uniformity**.
- **No idiosyncratic quirks** that could complicate understanding and refactoring by the entire team.

4. Metaphor

- Use metaphors to **describe how the system should work.**
- These **analogies** express the functionality of the system.
- Provides a simple way to remember naming conventions.

5. Simple Design

- The code should **pass all tests** and fulfill certain functionality while maintaining:
 - **Best communicate the intention (cohesion).**
 - **No duplicate code.**
 - **Fewest possible classes and methods.**
 - **“Say everything once and only once.” (DRY)**

Simplest thing:

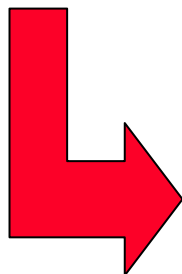
- XP developers always do **the simplest thing that could possibly work**.
- They never solve a more general problem than the specific problem at hand.
- They never add functionality sooner than needed.

6. Refactoring

- The code may be changed at any time to provide:
 - **Simplification.**
 - **Flexibility.**
 - **Reduced redundancy.**
- Automated unit tests are used to verify every change.

שרותים והפשטה דוגמא:

```
public static void printOwing(double amount) {  
    //printBanner  
    System.out.println("*****");  
    System.out.println("*** Customer Owes ***");  
    System.out.println("*****");  
  
    //print details  
    System.out.println ("name:" + name);  
    System.out.println ("amount" + amount);  
}
```



```
public static void printOwing(double amount) {  
    printBanner();  
    printDetails(amount);  
}  
  
public static void printBanner() {  
    System.out.println("*****");  
    System.out.println("*** Customer Owes ***");  
    System.out.println("*****");  
}  
  
public static void printDetails(double amount) {  
    System.out.println ("name:" + name);  
    System.out.println ("amount" + amount);  
}
```



When to refactor?

- **Refactor constantly**, throughout the lifetime of a project.
- Each time you fix a bug or add a new feature, look for overly complex code. Look for:
 - **Chunks of logic that are duplicated** and refactor them into a shared method.
 - Try to **rename methods and arguments** so they make sense.
 - Try to **migrate poorly designed code** towards better usage of design patterns.
- Writing unit tests is a great way to identify portions of code that need refactoring. When you write tests for a class, **your test is a client of that class**.

How to refactor?

1. Make sure you have a working unit test for the feature you are about to refactor.
2. Do the refactoring, or a portion of the refactoring.
3. Run the test again to ensure you did not break anything.
4. Repeat steps 2-4 until you are finished with the refactoring.

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/>

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

7. Testing

- Tests are **continuously written** with the system.
- **All tests are run together at every step.**
- **Customers write tests** that will convince them the system works.
- Don't proceed until current **system passes ALL tests.**

Testing

- Every piece of code has a set of **automated unit tests**, which are released into the code repository along with the code.
- The programmers **write the unit tests before they write the code**, then add unit tests whenever one is found to be missing.
- No modification or refactoring of code is complete until **100% of the unit tests have run successfully**.
- **Acceptance tests** validate larger blocks of system functionality, such as user stories.
- When all the acceptance tests pass for a given user story, **that story is considered complete**.

Unit tests

- A **unit test** is a programmer-written test for a single piece of functionality in an application.
- Unit tests should be fine grained, testing small numbers of closely-related methods and classes.
- Unit tests should not test high-level application functionality.
- Testing application functionality is called **acceptance testing**, and acceptance tests should be designed by people who understand the business problem better than the programmers.

Writing tests

All tests must be pass/fail style tests.

Grouping tests into **test suites**:

Now Testing Person.java:

Failure: Expected Age 2, but was 1 instead

Now Testing Account.java:

Passed!

Now Testing Deposit.java:

Passed!

Summary: 2 tests passed, 1 failed.

- The entire suite of unit tests must always pass at 100% before any code is integrated into the source repository.
- Acceptance tests do not have to pass at 100%.

8. Continuous Integration

- Newly finished code is **integrated immediately**. Unit tests must run 100% successfully, both before and after each integration.
- System is **rebuilt from scratch** for every addition.
- New system must **pass all tests** or new code is discarded.
- Additions and modifications to the code are integrated into the system on at least a daily basis.

9. Small Releases

- A **functional system is produced** after a few months.
- System is released **before the whole problem is solved.**
- New **releases regularly** (daily to monthly).

Small releases

- The smallest useful feature set is identified for the first release.
- Releases are performed as early and often as possible.
- Each release: a few new features added each time.

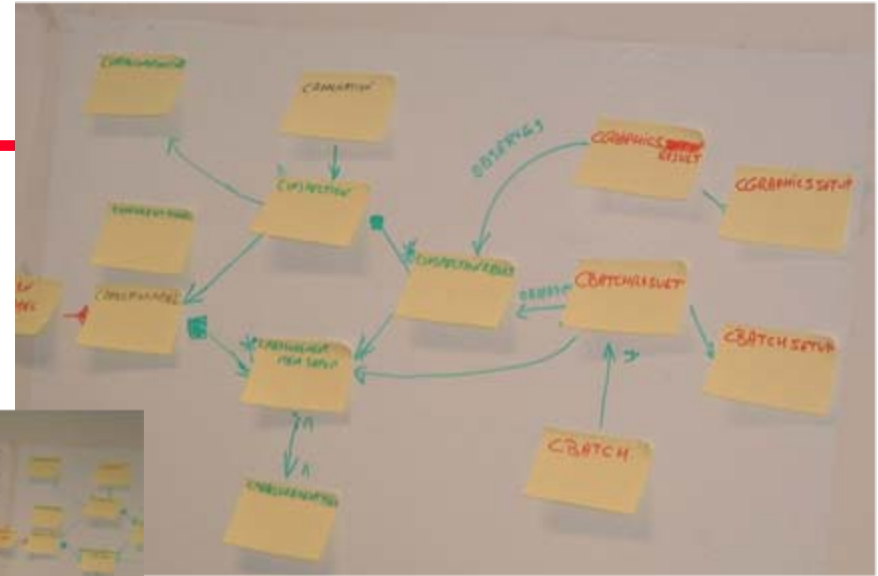
10. The Planning Game

- Schedule small tasks to be completed during the current completed iteration.
- Programmers will focus their attention on the tasks at hand.
- List of tasks is updated regularly.

11. Collective Ownership

- All workers **can access any of the code.**
- **Any programmer can change any part of the system** if an opportunity for improvement exists.
- The TEAM makes the product.
- It works...
- ... in disciplined XP teams.

More info: <http://www.xp.be/>



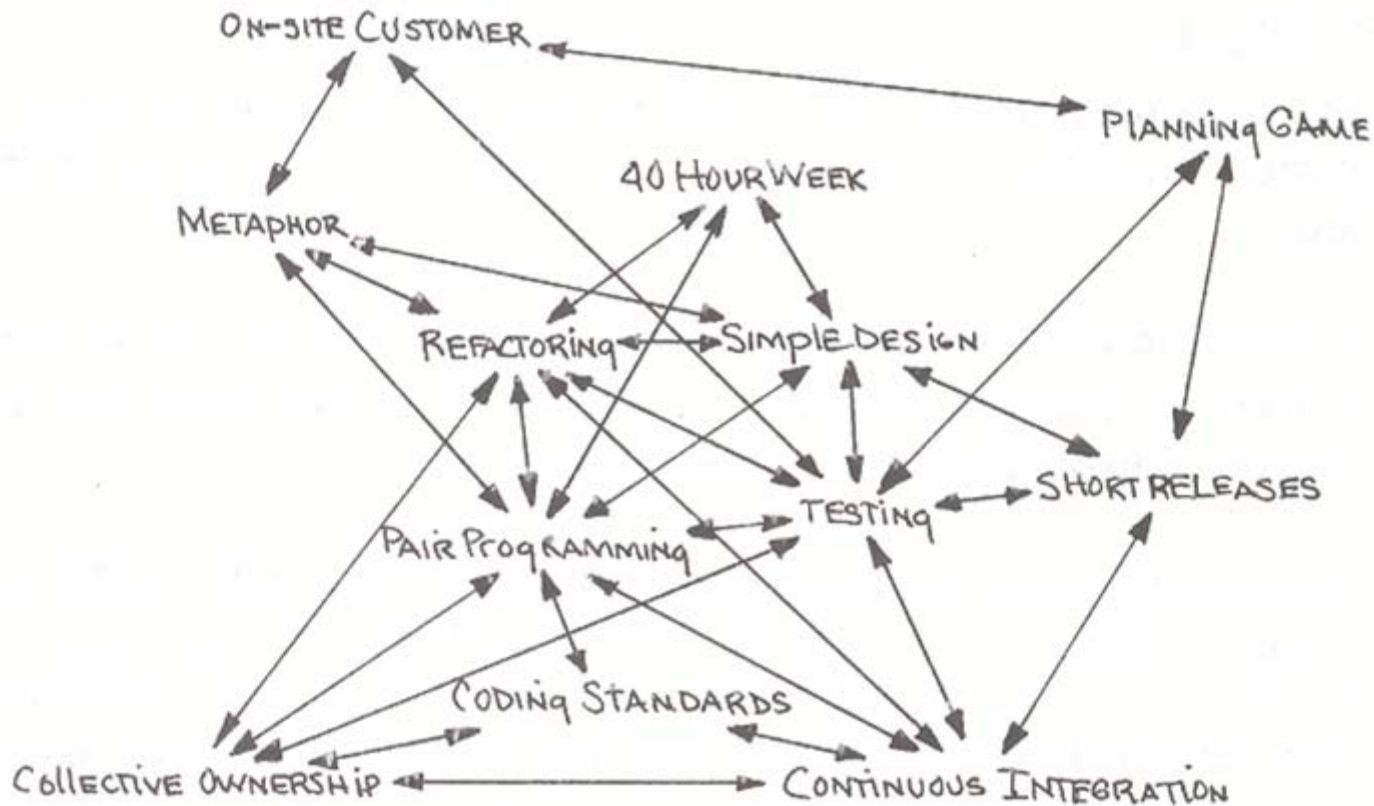
12. Sustainable pace: 40 Hour Weeks

- **Consecutive weeks of overtime is not allowed.**
- The need for overtime is a **symptom of a deeper problem.**

Just Rules

- These rules are **just rules**.
- XP teammates agree to **follow all of the rules**.
- An agreement can be made **to change the rules**.
 - Must address side effects of rule change.

Dependency of Practices



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

Open Workspace

- Work on computers set up in the **middle of a large room** with cubicles around the edges.
- Question: With how many people do you want to work in one room?

Daily Standup Meeting

- Stand up to keep it short.
- Everybody
 - Agrees what they will work on
 - Raises problems & difficulties
 - Knows what's going on
- Initial pairing.

More info: <http://www.xp.be/>



More info: <http://www.xp.be/>



More info: <http://www.xp.be/>

