

---

# Human Aspects of Agile Software Engineering: Cognitive and Social Analysis

---

Orit Hazzan

Technion - Israel Institute of Technology

Based on my joint work with [Yael Dubinsky](#), [Jim Tomayko](#),  
[Uri Leron](#), [Irit Hadar](#) and [Meira Levy](#).

---

# Problems with software development

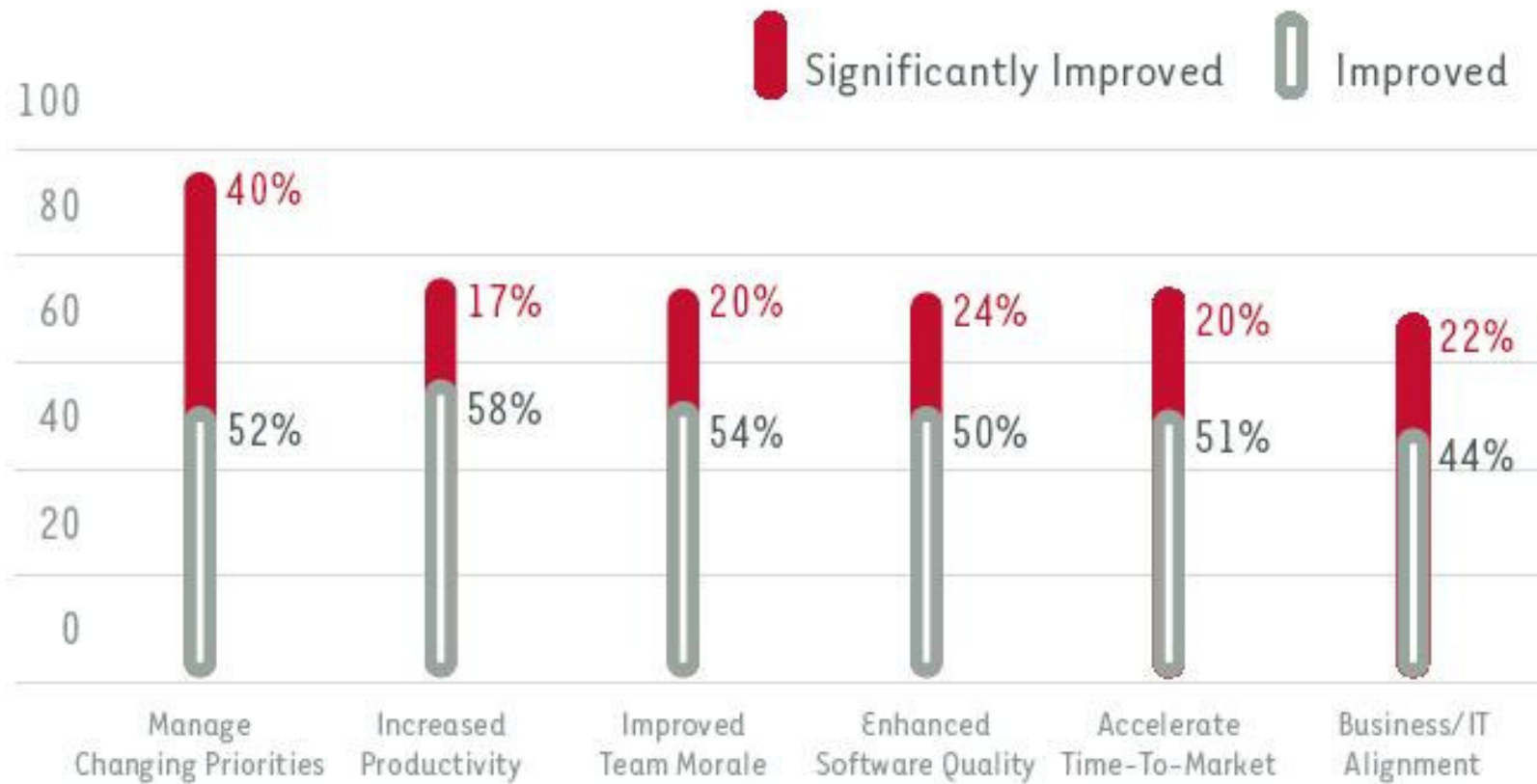
- **Google: "problems with software development"**
  - Requirements are complex
  - Clients usually do not know all the requirements in advance
  - Requirements may be changing
  - Frequent changes are difficult to manage
  - Process bureaucracy (documents over development)
  - It takes longer
  - The result is not right the first time
  - It costs more
  - Applying the wrong process for the product

---

# Main Message

- Software intangibility
  - How does *agile* software development improve software projects' results?
    - Increases transparency
    - Reduces cognitive complexity
  - Main Topics
    - Collaboration
    - Abstraction
    - Testing
-

# VersionOne Survey



source: VersionOne/Agile Alliance Survey August 2006

*Value Actually Realized from Agile*

---

# Part A - Collaboration

---

# Bonus Allocation

	Personal Bonus (% of the total bonus)	Team Bonus (% of the total bonus)	How will this option influence the collaboration of the team members?
a	0	100	
b	20	80	
c	50	50	
d	80	20	
e	100	0	

X% personal, Y% team means that X% of the total bonus is divided on a personal basis and Y% is divided equally between the team members.

---

# Bonus Allocation

- **A conflict:** Individual vs. group interests
  - **Game Theory:**
    - analyzes human behavior
    - decision making
    - decisions are mutually interdependent
    - players wish to maximize their profit
  - **Prisoner's Dilemma:** Illustrates cooperation in situations in which people can not verify that their cooperation is reciprocated.
-

# The Prisoner's Dilemma: A's perspective

	B cooperates	B competes
A cooperates	+5 No evidence	-10 Evidence against A; B is released
A competes	+10 Evidence against B; A is released	0 Evidence against both; punishment is decreased because they cooperated with the police

**The Prisoner's Dilemma: The case of software development - A's perspective, Bonus**

	B cooperates	B competes
A cooperates	50%	20%
A competes	80%	0%

---

**The Prisoner's Dilemma:** The case of software development - **A's perspective, Bonus**

	B cooperates	B competes
A cooperates	+5	-10
A competes	+10	<b>-20</b>

---

# The Prisoner's Dilemma: The case of software engineering

- Usually, team members are asked to cooperate.
  - **Software intangibility:** Team members can't verify that their cooperation will be reciprocated.
  - **According to the Prisoner's Dilemma:** Team members will tend to compete.
  - **In software development:** Such behavior leads to the worst results for **all** team members.
-

---

# The Prisoner's Dilemma: The case of agile software development

- How *cooperation* and *trust* can be achieved?
    - establish an environment which enables to verify that one's cooperation is reciprocated.
      - *Transparency*
      - *A set of specific activities* (practices) that all team members are committed to apply.
-

---

The Prisoner's Dilemma: The case of a transparent process - A' and B 's perspective

	B cooperates
A cooperates	+5



---

# Agile Software Development: Transparency

- Whole team:
    - development environment
    - daily stand-up meetings
  - Short releases: planning sessions
  - Measures
  - Customer involvement
-

---

# Kent Beck

- Kent Beck explains when XP is not appropriate:
    - "[i]t's more the social or business environment. If your organization punishes honest communications and you start to communicate honestly, you'll be destroyed."
  - An interview with Kent Beck, June 17, 2003, *Working smarter, not harder*, IBM website: <http://www-106.ibm.com/developerworks/library/j-beck/>
-

---

# Exercise

- Select an agile practice.
    - How does it contribute to the process transparency?
-

---

# Questions to think about

- The Prisoner's Dilemma implies a specific application with respect to SDMs - increase transparency.
  - Relevant questions:
    - May other SDMs be analyzed in a similar way?
    - May other theories help in a similar way?
-

---

# Part B - Abstraction

---

---

## What is common to the following statements?

- “I need to gain a global view at the application in order to know how this method fits into it”.
  - “I truly believe that if I had a minute to think about these two objects more abstractly, I’d have come up with the conclusion that they can be extracted into one class. But I must move on to the next development task”.
-

---

## What is common to the following statements?

- "I need some time to think about the code without being swamped with all the details. I'm almost sure that if I could leave now and go to swim, I could have come up with a solution. But I must stay late as all the others on my team".
  - "I wish I could join the programmer when s/he writes the code. You ask why? I'm not sure if this design can be implemented in c++."
-

---

## What is common to the following statements?

- “I need to gain a **global view** at the application in order to know how this method fits into it”.
  - “I truly believe that if I had a minute to think about these two objects **more abstractly**, I’d have come up with the conclusion that they can be extracted into one class. But I must move on to the next development task”.
-

---

## What is common to the following statements?

- “I need some time to think about the code without **being swamped with all the details**. I’m almost sure that if I could leave now and go to swim, I could have come up with a solution. But I must stay late as all the others on my team”.
  - “I wish I could join the programmer when s/he writes the code. You ask why? I’m not sure if this design **can be implemented** in c++.”
-

---

## What is common to the following statements?

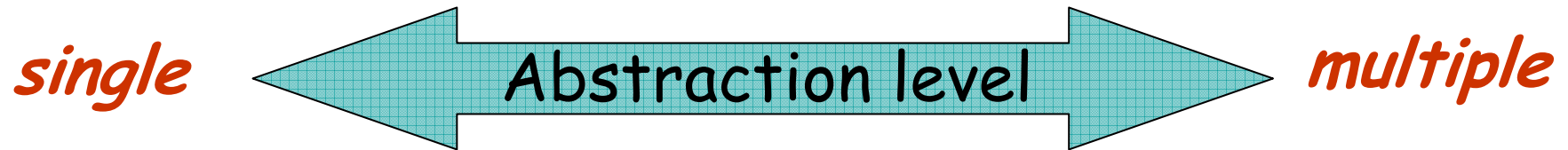
- "I need to gain **global view** ..."
- "I truly believe that had I had only a minute to think about these two objects **more abstractly**..."
- "I need some time to think about the code without being swamped with all the **details**..."
- "...I'm not sure if this design can be **implemented** into C++."

---

**The need to move between abstraction levels**

---

# Abstraction



---

# Multiple abstraction levels in agile software development environments

## ■ Planning Sessions

- the **release** planning session: **high** abstraction level;
  - the **iteration** planning session: **lower** abstraction level.
  - the **entire team** participates in the planning game; developers see the **entire picture** of the system as well as **its parts**.
-

---

# Multiple abstraction levels in agile software development environments

- **Short Releases**

- guide not to stay for too long a time in too high or too low level of abstraction

- **Pair Programming**

- **Refactoring**

---

---

# Questions to think about

- What other agile practices guide moving between abstraction levels?
  - What other cognitive mechanisms does agile software development offer?
-

---

# Part C - Testing

---

---

# Questions about testing

- Who likes testing? Why?
  - Who dislikes testing? Why?
-

---

# Outline - Testing

- Why people don't like testing?
  - cognitive, social, affective, managerial reasons
- How Test Driven Development (TDD) may help?



---

# Why people don't like testing?

- **Observation 1:** In traditional development environments, testing appears as one of the last stages and is usually done under pressure.
-

---

# Software Engineering/Hans Van Vliet



- P. 386-397: ... the testing activity often does not get the attention it deserves. **By the time the software has been written, we are often pressed for time, which does not encourage thorough testing.**
- p. 397: Postponing test activities for too long is one of the most severe mistakes often made in software development projects. This postponement makes testing a rather costly affair.

**Yet, this book presents the traditional life cycle of software development processes.**

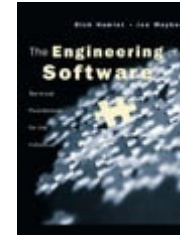
---

---

# Why people don't like testing?

- **Observation 2:** Testing may give a **negative feedback** (and who likes it?)
-

# Hamlet and Maybee



**T**esting a program ought to be more fun than designing and coding it, certainly more fun than negotiating with its users about the requirements. For the first time in development, you get to see it work! There's no denying that watching all that information processing horsepower in action is interesting, even exciting. After so much work, running the program should be a reward.

Maybe the reason testing is not always thought of as fun is that there's a flip side: the program may not work. In the earlier parts of development, things can go wrong, but failures are not as absolute and graphic as they are in testing. A developer can even (unconsciously or on purpose) sweep problems under the rug during requirements, specification, design, and coding—but when those problems show up as failed tests, it's no longer possible to kid yourself.

Or maybe the fun of testing diminishes if it isn't your own code being tried. But testing is still detective work that would be a serious challenge for Sherlock Holmes. The program and its specification and design documentation contain the clues to the crime (the lurking crashes), and the tester must find them as a triumph of deduction. Then again, not everyone likes detective work, and many people have no talent for it. Dr. Watson had none.

---

# Why people don't like testing?

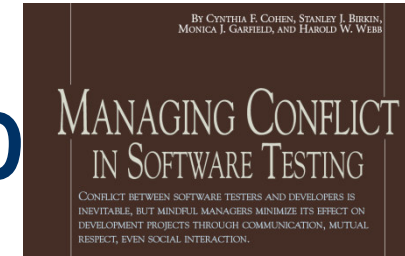
- **Observation 3:** **Testing** in traditional environments **is done by someone else**
    - The responsibility is transferred
      - Especially, if it gives a negative feedback!
-

---

# Why people don't like testing?

- **Observation 4:** Testing in traditional software development environments is carried out at the end of the production line, and, inspired by tradition working class jobs, gets **low status**.
-

# Managing Conflict in Software Testing, *Cohen et al. CACM, 20*



- Though most organizations recognize the need for high-quality testers and their specialized skill set, **testers still struggle to win the respect they deserve.** One manager told us, "If you had a diagram with God at the top, the engineers [developers] would put themselves above that." Many testers feel they struggle to maintain their place relative to that of developers.
- P. 80: **The lack of status and support makes the tester's job more difficult and time consuming, as the struggle for recognition becomes part of the job itself.**

---

# Additional Observations

- **Managerial difficulties:**
    - Testing slows down development
    - It's hard to manage testing (when to test)
  - **Cognitive difficulties:**
    - It's hard to know what to test
    - It's hard to know how much testing should be done
-

---

# Intermediate Summary

- Why people don't like testing?
  - Testing is difficult
    - Time pressure
    - Testing gives negative feedback
    - The responsibility is transferred
    - Low status of testing
    - More...
  - How may TDD help?
-

Kent Beck (2001).

*Extreme Programming Explained*, p. 116



Remember the principle “Work with human nature, not against it.”

That is the fundamental mistake in the testing books I’ve read. They start with the premise that testing is at the center of development. You must do this test and that test and oh yes this other one, too. If we want programmers and customers to write tests, we had better make the process as painless as possible, realizing that the tests are there as instrumentation, and it is the behavior of the system being instrumented that everyone cares about, not the tests themselves. If it was possible to develop without tests, we would dump all the tests in a minute.

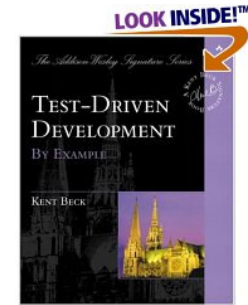
Reduce complexity

Increase transparency

---

# About the book: Kent Beck

## Test Driven Development: By Example



- **From Google:** Clean code that works - now. This is the seeming contradiction that lies behind much of the pain of programming. Test-driven development replies to this contradiction with a paradox - test the program before you write it. A new idea? Not at all. Since the dawn of computing, programmers have been specifying the inputs and outputs before programming precisely. Test-driven development takes this age-old idea, mixes it with modern languages and programming environments, and cooks up a tasty stew guaranteed to satisfy your appetite for clean code that works - now.
-

---

# How TDD copes with the challenge

- Time pressure
    - TDD is done all the time and is not postponed to the end of the process
  - Testing gives negative feedback
    - TDD ends with a success
-

---

# Practitioners' reflection

- Code Unit Test First

Why don't people like testing? Well, the traditional way of testing is tough to take. You write what seems to be perfectly sensible code, then you write a test and the test tells you that you failed. No one wants to hear that.

Let's turn it around. Write the test first; run it. Of course it fails.. You haven't written the code under test yet. Start writing code.. keep testing. Soon, the test will tell you that you've succeeded! MichaelFeathers

---

---

# How TDD copes with the challenge

- **Time pressure:** TDD is done all the time and is not postponed to the end of the process
  - **Testing gives negative feedback:** TDD ends with a success
  - **The responsibility is transferred:** TDD is done by the developers who write the code
  - **Low status of testing:** All developers are testers
-

---

# Additional Observations

- Managerial difficulties:
    - Testing slows down development
      - TDD fosters development processes
      - Automatic (not manually) process
    - It's hard to manage testing (when)
      - TDD turns development (and testing) to be a controlled process
-

---

# Practitioner's reflection

- A key aspect of this process: don't try to implement two things at a time, don't try to fix two things at a time. Just do one. When you get this right, development turns into a very pleasant cycle of testing, seeing a simple thing to fix, fixing it, testing, getting positive feedback all the way. Guaranteed flow. And you go so fast! Try it, you'll like it.

[RonJeffries](#)

---

---

# Additional Observations

- **Cognitive difficulties:**
    - ▣ TDD improves understanding of what you develop
      - What to test
      - How much testing should be performed
-

---

# Practitioner's reflection

- <http://xp.c2.com/UnitTest.html>

It's a wonderful ... experience ... .

I don't write code any other way anymore.

**My code has less problems, I have more confidence and management has more confidence. - sg**

---

---

## Conclusion - Part C

- TDD may help in coping with
    - Cognitive
    - Affective
    - Social
    - Managerialchallenges of testing.
-

---

# Conclusion

Main challenge of Software Engineering:

**Software intangibility**

- The agile approach:
    - Increases transparency
    - Reduces cognitive complexity
-

---

Thank you.

Questions?

---

Orit Hazzan

Email: [oritha@techunix.technion.ac.il](mailto:oritha@techunix.technion.ac.il)