Codurance.com

Agile Technical Practices

Agenda

- Introduction
- XP practices
- Pair programming
- Classic TDD

“A journey of a thousand miles begins with a single step”

Lao-Tzu

Introduction - what are technical practices?

Agile practices:

- Scrum & Kanban
    - Organizational, non-technical
- eXtreme Programming
    - Principles, organizational AND technical

Why do we need them?

Because lack of technical practices allows technical debt to grow into the codebase and spread under the radar - until it becomes too painful to be tackled when evidence arises.

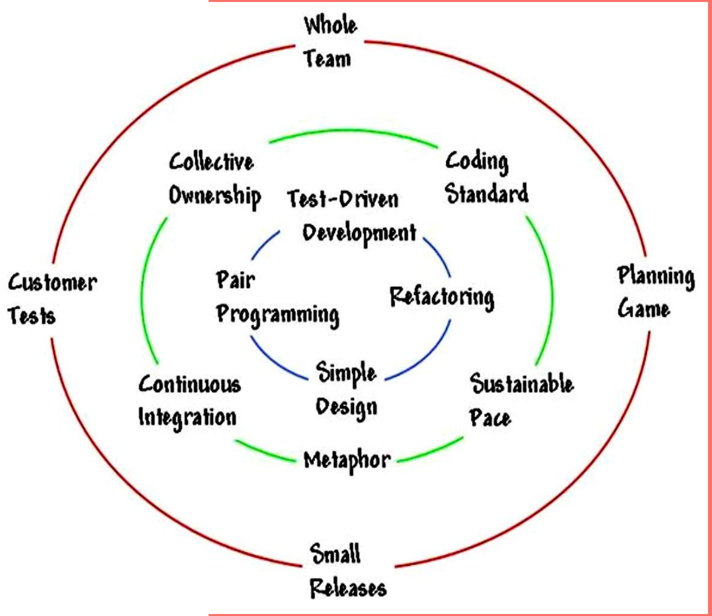
A set of techniques that constantly monitor and reduce the amount of technical debt in a project prevents its accumulation. The key point here is dealing with debt as it emerges: so we have FEEDBACK technical practices (monitoring) and DESIGN technical practices (removing/enhancing).

eXtreme Programming practices

- ➔ **Feedback practices**
  - ◆ TDD
  - ◆ Pair Programming
- ➔ **Design practices**
  - ◆ Refactoring
  - ◆ Simple Design

Design and feedback practices are key points for experimentation: only a quick feedback cycle can let the team take informed decisions about what is done and what to do next.

XP practices

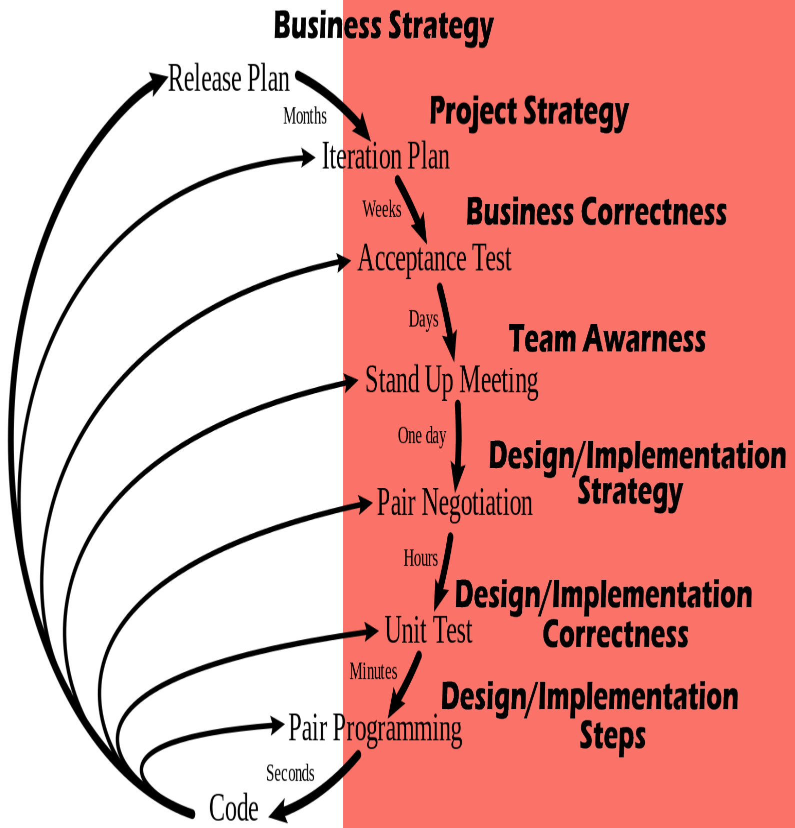


Feedback loops

“Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, courage, and respect. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation”

“Optimism is an occupational hazard of programming: feedback is the treatment”

Kent Beck



codurance

“The adjustment period from solo programming to collaborative programming was like eating a hot pepper. The first time you try it, you may not like it because you are not used to it. However the more you eat it, the more you like it.”

Anonymous XP Practitioner

Pair programmers:

- Keep each other on task.
- Brainstorm refinements to the system.
- Clarify ideas.
- Take initiative when the partner is stuck, thus lowering frustration.
- Hold each other accountable to the team's practices.

Pair programming roles

Navigator

- Responsible for reviewing the driver's work. Should catch incidental mistakes, but also make strategic considerations about design, duplication and dependencies with other parts of the system
- Should keep note of all activities that come to mind but get deferred because the current task is not finished
- When navigator is lost, they must let the driver know immediately

Driver

- Responsible for typing, moving the mouse, etc.
- Speaking and telling your peer what you are doing while coding is a very good exercise for clarifying it also to yourself

Driver/navigator switch techniques

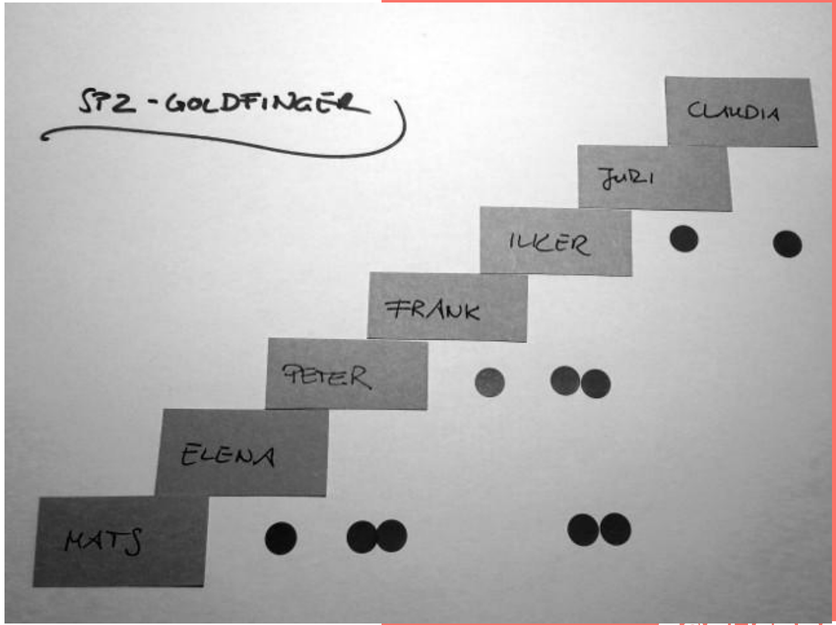
Both partners should spend an equal amount of time in each role. How?

- **Chess clock**
  - Roles switch after a predefined time interval
- **Ping pong**
  - Dev A writes a new RED test. Dev B writes code to pass the test. Now Dev B writes a new RED test. Dev A writes code to pass it. And so on.
- **Pomodoro**
  - Set a pomodoro timer at x minutes and when it rings take a short break and switch. After 4 pomodoros take a longer break.

Pair rotation

It is important in a team that all pairing combinations work together for some time without too much imbalance.
To keep track of this the “pair stair” board can be used.

- Whenever two team members pair up, they mark their “pairing” with a dot.
- The goal is to fill the voids, which are the pairs who have not yet worked together.



Classic TDD - the classicist approach

“Any fool can write code that a computer can understand.
Good programmers write code that ***humans*** *can understand”*
Martin Fowler

The Classicist approach is the original approach to TDD created by Kent Beck.
It’s also known as Chicago/Detroit School of TDD.

Classic TDD - benefits

- **Design**
  - Following correctly the rules of TDD guarantees a solution composed of testable modules. Another word for “testable” is “decoupled” or “loosely coupled”. That means the design is more flexible, more maintainable and just much cleaner.

- **Documentation**
  - Have you ever integrated a third party package using a manual? Do you read all the text or just skip to the code examples? Well written unit tests are just like that manual, without all the noisy text. The tests are real examples of how to use the production code, written in the language programmers understand (code), unambiguous and cannot get out of sync with production.

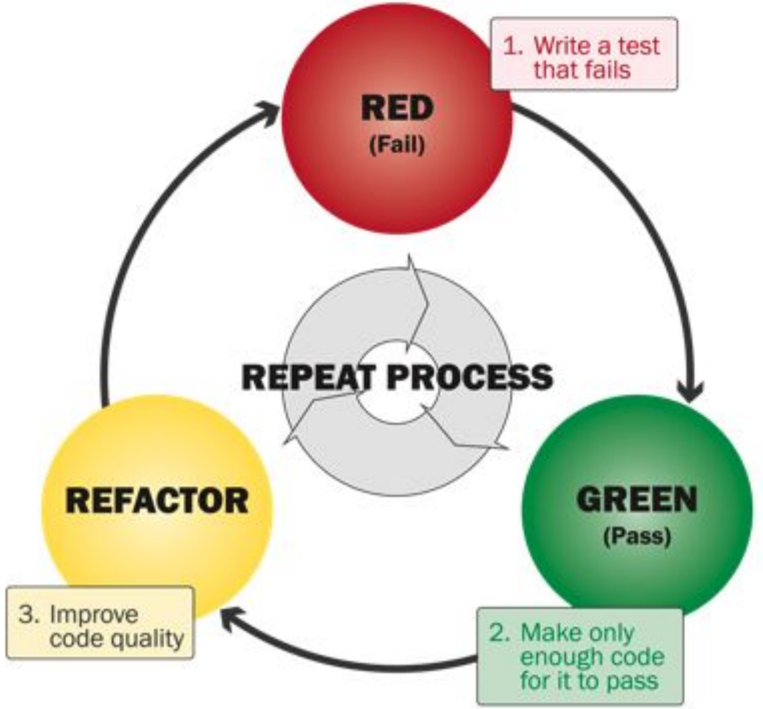
Classic TDD - benefits

- **Debugging**
  - If you are just a few minutes away from having a working system, most of the time there is no need to use the debugger. And when you need to debug, you can always do it just for the test you are investigating, reducing the debug time and effort to the very minimum.
- **Courage**
  - Ever been afraid to touch some opaque, spaghetti-like legacy code fearing you'd break it? What if you had a suite of tests giving you an immediate feedback about your changes? That would be the safety net you needed for gathering the courage to actually go and clean up the mess!

The three laws of TDD / baby steps

1) You are not allowed to write any production code, unless it is for making a failing unit test pass
2) You are not allowed to write any more of a unit test than is sufficient to fail. (compilation failure is a failure)
3) You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Refactoring -> use the *Rule of Three*:
Extract duplication only when you see it for the third time


RED
(Fail)
1. Write a test that fails
GREEN
(Pass)
2. Make only enough code for it to pass
REFACTOR
3. Improve code quality
REPEAT PROCESS

Baby steps - the three ways forward

1. Fake implementation
When you hard code exactly the value you need to pass the test

2. Obvious implementation
When you are sure of the code you need to write. This is what you will be using more often to move forward quickly.

3. Triangulation with the next test
When you want to generalize a behaviour but are not sure how to do it. Starting with fake implementation and then adding more tests will force the code to be more and more generic. Complete one dimension first and then move on the next one with another test case.

Main characteristics

- Design happens during refactoring.
- Test are usually state-based.
- When refactoring, the unit under test can grow to multiple classes.
- Mocks are rarely used, usually only for isolating external systems.
- No upfront design assumptions. Design emerges completely from code, hence solving over-engineering problems.
- Easy to understand due to state-based tests and no upfront design.
- Often used with the 4 Rules of Simple Design.
- Good for exploration when only input/output couples are known (black box).
- Great when we can’t rely on a domain expert or domain language (algorithms, data transformation, etc.).

Main problems

- Exposing state for test purposes only.
- When unit under test become bigger than a class, using real collaborators instead of mocks will make the tests vulnerable, as those collaborators evolve life by their own and can cause completely unrelated tests to break.
- Refactoring step is often skipped by inexperienced practitioners and left as the final big refactoring step.
- The public interface of the units under test are created accordingly to the criteria “I think I will need these public methods”, which doesn’t always fit well with the rest of the system.
- Can be slow and wasteful when we know that a class has too much responsibility and other classes could be extracted early on.

Fizz Buzz kata
Write a function that takes numbers from 1 to 100 and outputs them as a string, but for multiples of three returns "Fizz" instead of the number and for the multiples of five returns "Buzz". For numbers which are multiples of both three and five returns "FizzBuzz".

Leap year kata

Write a function that returns true or false depending on whether its input integer is a leap year or not.

A leap year is defined as one that is divisible by 4, but is not otherwise divisible by 100 unless it is also divisible by 400.

For example, 2001 is a typical common year and 1996 is a typical leap year, whereas 1900 is an atypical common year and 2000 is an atypical leap year.

Classic TDD

Nth Fibonacci

Write a function that generates the Fibonacci number for the nth position implementing:

```
int Fibonacci(int position)
```

First Fibonacci numbers in the sequence are:

Position	0	1	2	3	4	5	6	7	8	9
Fibonacci	0	1	1	2	3	5	8	13	21	34

Further reading

Online

http://en.wikipedia.org/wiki/Extreme_programming

http://www.extremeprogramming.org/

http://www.extremeprogramming.org/rules.html

http://ronjeffries.com/xprog/what-is-extreme-programming/

Books

http://www.amazon.co.uk/Extreme-Programming-Explained-Embrace-Change/dp/0321278658/ref=sr_1_3?ie=UTF8&qid=1428489551&sr=8-3&keywords=kent+beck