10 Best Practices for
Agile Software Development

How to develop high-quality software?

Prof. Fabio Kon
Department of Computer Science
University of São Paulo

MIT Senseable City Lab
January, 11th, 2019

What is Software Development?

– Modeling	(Jacobsen)
– Engineering	(Meyer)
– Discipline	(Humphreys)
– Poetry	(Cockburn)
– Craft	(Knuth)
– Art	(Gabriel)

(from Alistair Cockburn)

- Common mistake: look at software as only one of the above items.

Conventional Software Development

Waterfall model (pervasive from 1960s to early 2010)

1. Requirement elicitation
2. Requirement analysis
3. Design
4. Implementation
5. Tests
6. Maintenance

Old Assumptions

- one must do the best possible job in one stage before starting the next one

- it’s very costly to change something in a previously completed step

But the world is now different

- Requirements change very rapidly
- The customer doesn’t know what he/she wants
- It’s easy to change (well written) software
  - new languages, frameworks, methods, tests

In the Agile mindset

- You’re always ready to:
    - change everything (requirements, code, plans)
    - interact with your customer
        - to show what you did and get feedback
        - to receive new requests
    - replan the next steps

In the Agile mindset

- You perform all the steps of the waterfall everyday (or every week):

  Customer negotiation

  Design

  Implementation

  Tests

  Maintenance

10 Agile Best Practices

- Intention-revealing names
- Design Patterns
- Customer Involvement
- Management & Planning
- Automated Tests
- Code Reviews
- Version Control
- Development, Homologation (Acceptance), and Production environments
- Continuous Delivery
- When and how to optimize

1. Intention-revealing Names

Meaningful Names

Names are vital!

- Code is basically names and reserved words
- Choosing good names takes time but saves more than it takes
- Names should be expressive and should answer questions

Example

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if(x[0] == 4)
      list1.add(x);
  return list1;
}
```

Meaningful Names

Example

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if(x[0] == 4)
            list1.add(x);
    return list1;
}
```

Many doubts arise...

1. What does this method get?
2. What kinds of things are in theList?
3. What is the importance of the zeroth position?
4. What is the significance of the value 4?

Meaningful Names

Example

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if(x[0] == 4)
            list1.add(x);
    return list1;
}
```

What about this code?

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if(cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Meaningful Names

Example

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if(cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Problem solved!

1. What does this method get? It gets all flagged cells!

2. What kinds of things are in theList? theList is a gameBoard filled with cells!

3. What is the importance of the zeroth position? That's the Status Value!

4. What is the significance of the value 4? It means it is flagged!

Meaningful Names

Example

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if(cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Going further...

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if(cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

This is pretty much what you expected!

Meaningful Names

summarizing

- seek ***intention-revealing*** names
- good names are neither too short nor too long (do not promote obscurity to save a couple of keystrokes)

- **if you find a bad name, change it now!**

2. Design Patterns

on the shoulders of giants

- *Design Patterns: elements of reusable object-oriented software* - GoF book

- Architectural Patterns (MVC, Pub/Sub, etc.)

- Implementation Patterns (Beck and Martin)

Design Patterns

on the shoulders of giants

- *Design Patterns: elements of reusable object-oriented software* - GoF book

- Architectural Patterns (MVC, Pub/Sub, etc.)

- Implementation Patterns (Beck and Martin)

3. Customer Involvement

don’t be shy:
talk to all stakeholders

- Show preliminary, icremental versions of your software to:
  - client
  - user
  - other stakeholders
  - (in academia: colleagues, conferences)

- Get frequent feedback

4. Management and Planning

- An Agile software development team:
  - 2 to 10 members
  - coach (experienced developer)
  - product owner (customer)
  - other developers:
    - testing manager, devops manager, DB manager, planning manager, etc.
    - (but everybody does everything, the manager simply makes sure it’s being done)

Management and Planning

- Agile Planning happens everyday.
- Layered approach:
  - Long-term planning is very vague, just a vision
  - Medium-term planning is vague
  - Short-term planning (monthly) is more detailed
  - Very-term-planning (weekly) is very detailed
- Story Cards are written by customer to describe requirements
- On the Back of story cards, developers list the tasks that are required to implement that card

Management and Planning

- Tool Support for Agile Planning:
  - Trello
  - GitLab / GitHub issue tracker
  - JIRA, Pivotal Tracker, etc.

- Release planning
  - Periodic meeting (with the entire team) to plan the next release
  - Customer defines priorities
  - Developers define development costs

5. Automated Tests

if it’s not tested, it doesn’t exist

- Each relevant line of code should have an automated test associated with it.
- Unit tests
- Acceptance tests
- Integration tests
- Smoke tests
- Performance tests
- Stress tests

Automated Tests

if it’s not tested, it doesn’t exist

- If you are a beginner, I suggest you start with Unit tests
- Use a framework for your specific language
  - pytest, JUnit, CPPUnit, etc.
  - Web: Selenium
- A large project should have thousands of automated tests and >90% of testing coverage
- The testing suite should be executed everyday (may times per day)

Automated Tests

MAJOR BENEFITS

- Communication
- (Self-checking) Documentation
- Safety net for changes/refactorings
- Helps one developer undestand the code written by the others

6. Code Reviews

- Collective code ownership
- Periodic code peer-review
- Pair programming

7. Software Execution environments

- Development Environment
  - e.g., your notebook
  - should standardize in the team

- Homologation (Acceptance) Environment
  - for the customer to try/test/play with
  - should be as similar as possible to production

- Production Enviroment
  - for the real users with real data

8. Code version control

- Code must be maintained in a repository, not in your [notebook, dropbox, server file system]
- The repository should use a modern *Version Control System (VCS)*
- git is a bit tricky but it’s very powerful VCS
- github, gitlab, are good online repositories

9. Continuous Delivery

Automate the entire process:

1) Write some new code

2) Run automated tests

3) if all pass -> push to Repository

4) Deploy new version in Homologation Environment

5) Run Smoke tests

6) Deploy new version in Production Environment

7) Shut down old version in Production Environment and redirect users to new version

10. When and how to optimize

Premature optimization is the root of all evil in programming (or at least most of it)

Donald Knuth

1) Only optimize if a functional or non-functional requirement is not met
2) Run a profiler and identify where’s the bottleneck
3) optimize just that bottleneck and go back to step 1)

The End

fabiokon@mit.edu

kon@ime.usp.br