HSLU Hochschule Luzern

Software Architecture and Techniques

Refactoring

Alexandra Junghans, Marcel Baumann, tangly llc

Lecture Content

- *Why Agile Architecture and Design?*
- *Evolution of Software Architecture over the last Decades*
- *What is Agile Architecture?*
- *Agile Approaches with Scrum, XP, LeSS*
- **Refactoring**
- **Errors, Vulnerabilities, Smells in Source Code**
- **Architecture of Components and Subsystems**
- **Verify Functional Features**
- **Validate Quality Attributes of Software Architecture**
- Architecture Documentation
- Architecture Trends I
- Architecture Trends II
- Domain Driven Design Workshop
- Team and Technical Excellence for Architects

Refactoring (1/3)

Refactoring is a **disciplined** technique for **restructuring** an existing body of code, altering its internal structure **without changing its external behavior**.

Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but **a sequence of these transformations can produce a significant restructuring**. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Refactoring (2/3)

By **continuously** improving the design of code, we make it **easier** to work with.

This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features.

If you get into the hygienic habit of refactoring continuously, you'll find that it is **easier to extend and maintain code**.

Joshua Kerievsky, Refactoring to Patterns

Refactoring (3/3)

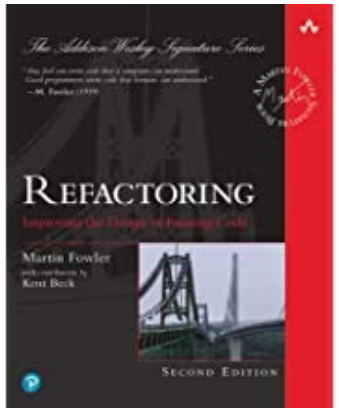
- *Why*: Improve the legibility and maintainability of the source code of your product
- *Who*: every developer does refactoring
- *When*: Always
- *Where*: Any code you write or modify
  ***never** in code you do not need to change. This would be **waste**.*

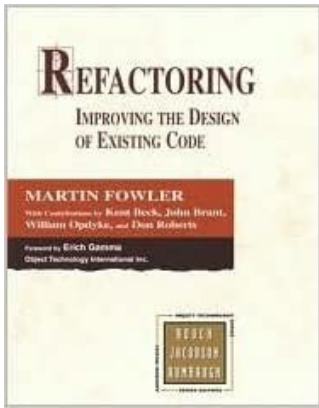
Reality (1/2)

- Refactoring is around for decades
- Quite a few refactoring recipes are simple, but
  - Majority of teams are not doing it
  - They are unprofessional and sloppy
  - Why?

Reality (2/2)

- Martin Fowler Refactoring 1st Edition was published in 1999,
- Martin Fowler Refactoring 2nd Edition was published in 2018,
- Please just **refactor your code**





Misconceptions

- You do **not** need to ask to refactor
- You do **not** need a backlog item to refactor

Refactoring is not a special task that would show up in a project plan.

Done well, it is a regular part of programming activity.

Simplistic Refactoring

- Code formatted following coding guidelines
- High quality naming of methods and variables
- Import statements are accurate
- No out-commented code – git was invented for this purpose 😉
- No TODO, FIXME
- No empty methods (*empty constructors are acceptable)*
- No empty catch blocks

Mechanical Refactoring

- Automatic improvement supported by your IDE
  - Zero risk to break the code
  - Just do it → every time you see an improvement in the code you are working, do it!
- Refactor each time you extend, correct or edit source code
- Upon each small set of changes commit in git – *can be every two minutes*

Mechanical Refactoring

- You need an advanced IDE
  - Renaming through the project
  - Move to another class or package
  - Change parameter list, or parameter name or type
  - Extract interface, method, class
- An advanced IDE shall hint about mechanical refactoring like a compiler finds errors

Typical Mechanical Refactoring

- Rename
- Move
- Extract
- Inline
- Change signature
- Delegate
- Remove superfluous keywords
- Use advanced loop
- Use streams
- Use method reference
- Use try with resources
- Multiple exception in catch
- Use unchecked exceptions
- Etc.

Typical does not mean trivial

- Why should you rename?
  - The name is not descriptive enough
  - The class/method/variable name doesn't match what it really represents
  - Something new has been introduced, requiring existing code to have a more specific name
- You should have short methods

Advanced Refactoring

- You change code to improve it
- You need a security net to guaranty the code behavior stays the same
  → TDD and ATDD are a must
- You need at least 60% coverage for a reasonable efficient security net.
  The value is backed through a few empirical studies.

Refactoring Examples (1/2)

- Never have public fields
- Never use parameters as local variables
- Never return null → either an empty collection or an optional
- Use standard library classes and exceptions
- Prefer private predicate methods to abstract complex *if* conditions

Refactoring Examples (2/2)

- Replace loops with streams
- Replace conditions with filters
- Prefer immutable classes (*records*)
- Design with interfaces and sealed classes
- *No checked exceptions*
- *Use modern switch expressions*

Test Driven Approach

If it is worth building, it is worth testing.

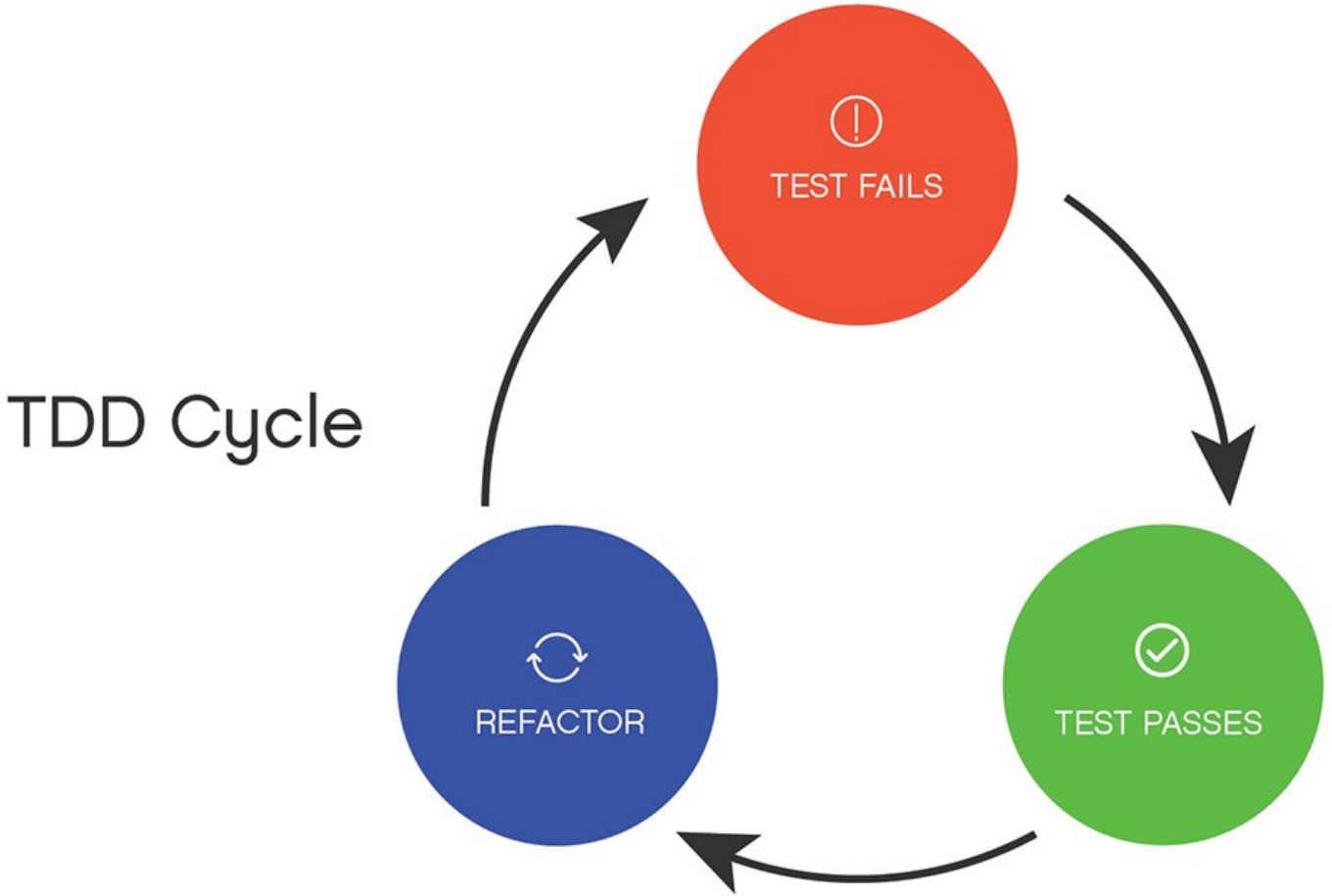
If it is not worth testing, why are you wasting your time working on it?

Scott Ambler

Test Driven Development

The duration of the cycle is a few minutes, never hours.

Upon completion of a cycle you should commit your changes.



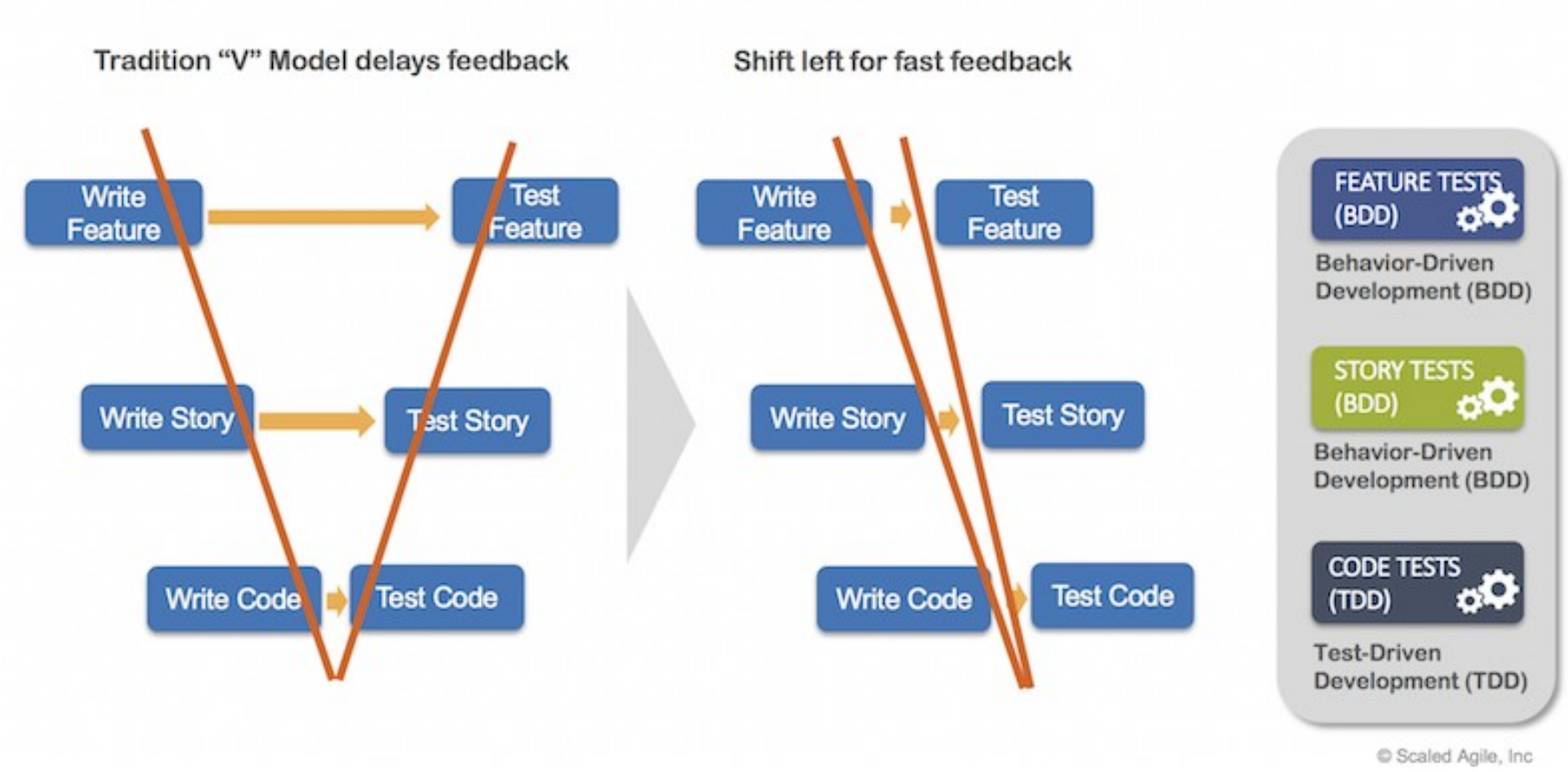
Legacy Code Unit Testing

- You can add unit tests to any existing class
- The steps are
  - Define context – *this can be hard if singletons and god design approach were used. You have to mock (e.g. Mockito library)*
  - Start with one method
    - Start testing the last statement in the method → down and right statement in your IDE,
    - Extend your tests to cover the left statements, and when move up.

Java Tools

- Junit 5
- AssertJ
- Mockito
- Helpers
  - Flyway, JSONassert, jimfs, BDD libraries

The Modern Way of Testing



Remainder

- *Scrum uses product backlog items and stories*
- *Stories have acceptance criteria.*
  Acceptance criteria are ***specifications by example***
- *Acceptance criteria are implemented as automated tests*
- *Automated tests are executed in the CI/CD pipeline*

Refactoring Catalog

- Online catalog from Martin Fowler
- Explore the catalog
- Learn which refactoring is automated in your IDE
- It is a **sin** to leave out refactoring opportunities before pushing your changes

Youtube and other Links

- Clean Code - I
- Clean Code - II
- Clean Code with .NET C# (and Resharper)

- Blog *Agile Code is Clean Code*

Exercises

- Study the refactoring catalog
  Become a refactoring expert!
- Read the cheat sheet *Clean TDD*
- Refactor code on your product and master your IDE
  - Why is it an improvement?
- Write unit tests on your code
  - What is your gain?