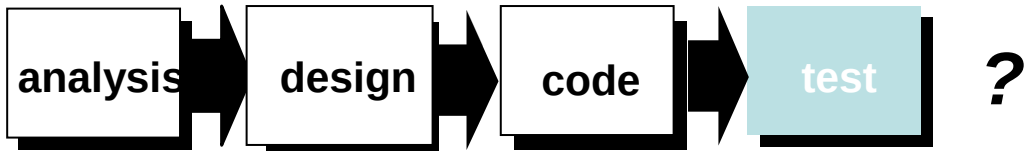
Test-Driven Development (TDD)

Branko Marović

Testing?

- Programmers dislike testing
- They will test reasonably thoroughly the first time
  - The second time, testing is usually less thorough
  - The third time, well..
- Testing is “boring”
- Testing is the job of another department / person
- Developers like to
  - Design?
  - Code
  - Make code beautiful?
  - Run
  - Get feedback by seeing their code working
  - Be in control over machine

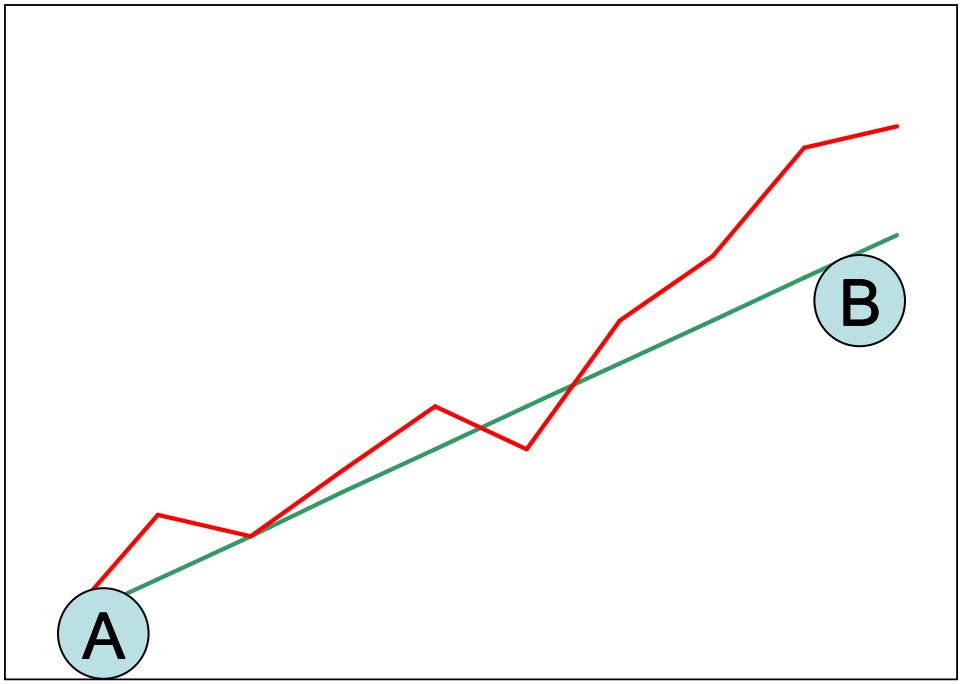
TDD vs. late testing



- When testing should take place in the lifecycle?
  - Test-driven development advocates early testing
  - Implementation and testing as two parallel activities
- “Before you write code, think about what it will do. Write a test that will use the methods you haven’t even written yet.”
- Test-Driven Development alternative names
  - Domain-Driven Development
  - Coding by Example

Who should write the tests?

- The programmers!
  - Cannot wait for somebody else to write them
  - They know what could go wrong
- Discover errors as soon as possible
- Iterative development in many small steps



TDD: Stories

- **Narrative user stories** iteratively describe user/business requirements covering use cases or Scrum product backlog items
- User stories and corresponding (new) behaviors are **expressed in programming language** by writing corresponding unit tests
- Developers immediately try to interpret the story and required behavior – forced to think from perspective of **users** and **features**

5

Stories and Examples

- Describe the most important scenarios (with the customer)
- Agree on what functionality needs to be implemented in order to realize each scenario
- A set of well defined examples – a testing plan for the user story
- Expected results – acceptance criteria – for each test. Must include both
  - Expected output
  - Expected new system state
- Do not expect all scenarios to be known in advance

TDD: Examples

- Write examples/tests **before** any implementation code
  - Developers must think about **interfaces** first (instead of implementation)
  - Tests provide a specification of **intent**: what a piece of code is supposed to do
  - Tests provide illustrative **examples of usage** of target code
- Tests drive or dictate the developed code
  - **“Do the simplest thing that could possibly work”**
  - Developers have less choice in what they write
  - Less guesswork or dramatic decisions
  - No over-engineering in implementations and interfaces
  - Code is easier to understand and maintain
- Coding in iterative increments
  - Immediate consequences of design decisions
  - Quick feedback and visible results – happier developers
- Strong emphasis on **refactoring**!

Writing examples

- First try write to an example how it should work
- Structure it using "**given, when, then**" scheme
- **One example per test!**
  - Make several tests for a one example is if it makes tests more readable
  - Some even propone only one assert per test!
  - Simple check of corresponding failure may be added to the same test
  - Check complex failures separately
- Generate methods/class declarations from example
  - Ctrl+1 –Eclipse suggestion / quick fix
  - Alt+Enter – NetBeans suggestion / hint

- Also
  - Eclipse
    - Ctrl+Space – auto complete
    - Alt+Enter – content assist – generate constructor, getters, setter
    - Shift+Space – incremental assist
    - Ctrl+Shift+Space – parameters hint
    - Ctrl+2, L on expression – variable declaration and assignment
  - NetBeans
    - Ctrl+Space – code completion (Alt+Enter full assignment usage)
    - Alt-Insert – generate constructor, getters, setter

Key points for writing tests

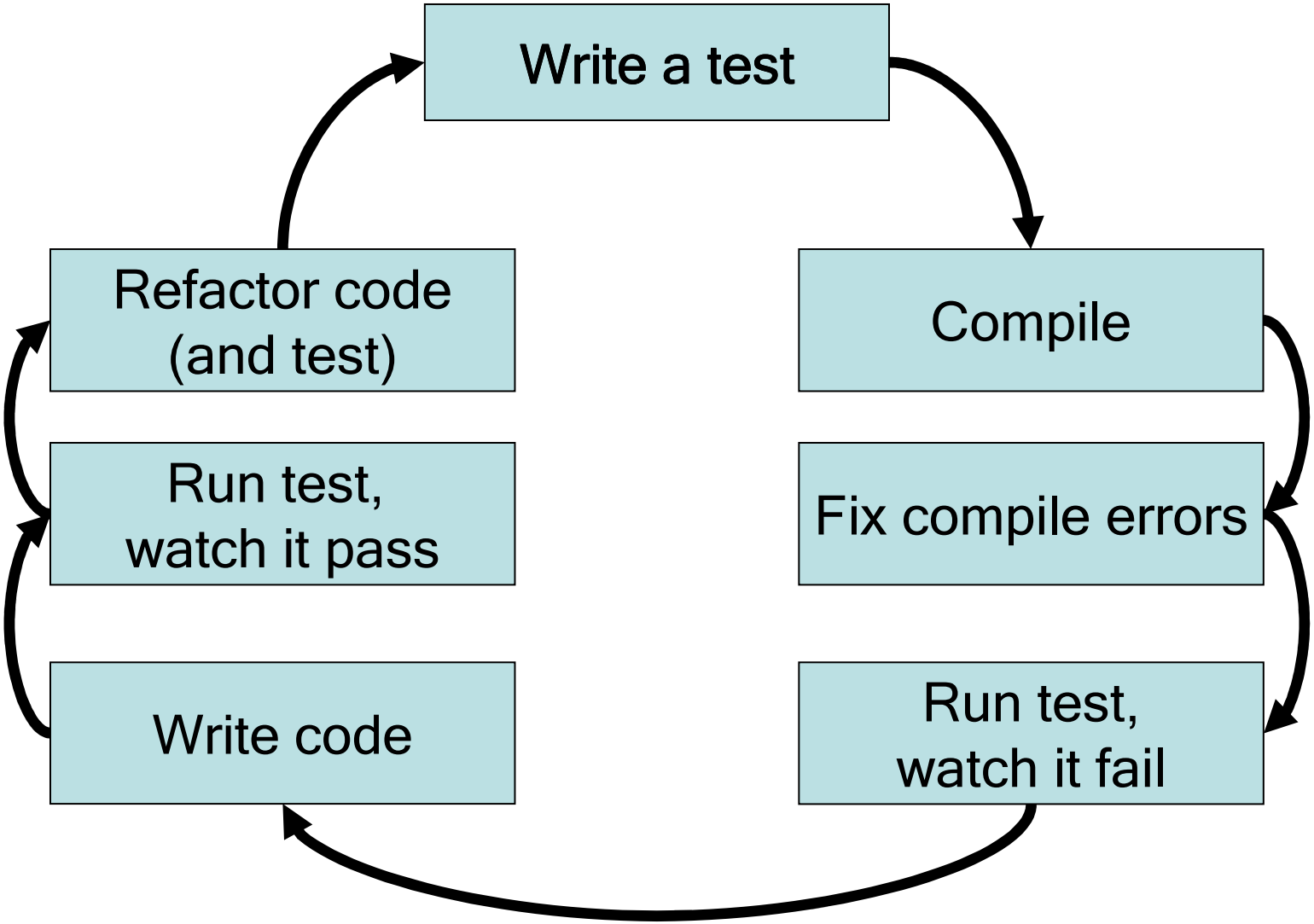
- Write examples of stories, not tests
- Do the simplest thing
- One intention/statement (“should”)
  - (One) example
  - Each (JUnit) test method should be associated with a **single** example
- In test method always group code in “ ” sections – AKA “3A Pattern”
  - Arrange = Given
  - Act = When
  - Assert = Then
- Do not make test mutually dependant – prepare test data in #given

Workflow

1. Write a single test
2. Compile, fail – missing behavior
3. **Refactor** using Ctrl+1, Alt+Enter just to make it compile
4. Compile, pass
5. Run test, fail
6. **Write minimal (simplest possible) implementation** to pass the test – make it work
7. Run test, pass
8. **Refactor** – change internal design for clarity and “once and only once” – make it better
9. Run test, pass
10. Repeat until done

- Let compiler and automated test tell you about errors and omissions
- Regularly run all tests
  - Automate: Maven, continuous integration…
  - … and check test results!

TDD Stages


Write a test
Compile
Fix compile errors
Run test,
watch it fail
Write code
Run test,
watch it pass
Refactor code
(and test)

Live with tests

- Refactor both used code and test regularly
- New tests guide implementation or extension of already existing interfaces and features of target code
- For user interaction there are Swing and web tools and JUnit test generators, e.g. Selenium
  - Test and implement needed business methods
  - Make GUI
  - Make GUI test
  - Not that straightforward!

xUnit Testing Frameworks

- Ported to various languages and platforms
  - JUnit, CppUnit, DUnit, VBUnit, RUnit, PyUnit, Sunit, HtmlUnit, …
  - Good list at www.xprogramming.com
- Standard test architecture
  - A test structure definition
  - Assertions
  - Tools to run tests (all, some…)
  - Tools to asses results

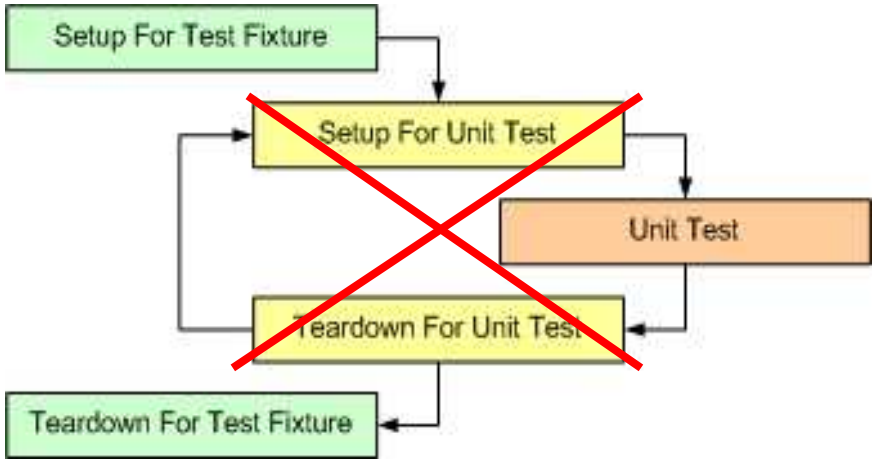
Refactoring

- When the code and/or intent is **not clear**
- **Duplication**
  - Once And Once Only (OAOO) AKA DRY (Do not Repeat Yourself)
- Code **smells**
  - Need for extensive comments
  - Large methods or classes
  - Inappropriate intimacy between classes
  - Too many responsibilities
  - Spread responsibility
  - …

Refactor tests to keep them readable

- Make minimal test suite fixture setup/teardown
- Localize context/data setup within test
- Apply builder pattern
- Use fluent interfaces
- Name and type data – avoid long arrays of numbers, strings or parameters
- Keep magic numbers under control
- Minimize coupling between tests
- Use a mocking framework to access or simulate external or missing parts of the system
- xUnits provide basic assertions – consider using assert frameworks to increase simplicity and readability!

Making effective tests



- A fixture is a set of objects that we have instantiated for our tests to use ~ piece of code that sets scene for tests.
- Use #given instead of test setup method
- If a feature is difficult to test, refactor it
- Address complex architectures with layered tests
- Builder and fluent interfaces can be used to create a DSL (Domain Specific Language)
- Make assertions readable (FEST framework) – avoid “if (...)” : use assertions DSL
- Consider adding checks elsewhere: preconditions (Google Guava), Java assertions, mocks

Example

- Three stories on translation…

Test doubles

- A test double is an object “stand-in”
  - Looks like the real thing from the outside
  - Execute faster
  - Easier to develop and maintain
- Dummies, **stubs,** fakes, **mocks, spies**
- Useful in
  - State based testing
  - Interaction based testing

Mocks

- A mock can replace or simulate complex, inaccessible or missing part of the system
- Useful to insulate test targets
  - Can test an object without writing/setting all its environment
  - Allows a stepwise implementation as we successively add more and more parts
  - On failure, we can be almost sure that the problem is in
    - New object (replacing mock)
    - Changed object (using mock)

Mockito framework

- Can track invocations and responses of a real object
- Stub can be set to return
  - Hardcoded values
  - Sequences of values
  - Conditional values
  - Exceptions on specific calls
- Verifications (on stubs or real spied objects) of
  - Invocations with specific arguments
  - Numbers of invocations
  - Invocation order
  - Unused or last invocations
- Good example of a fluent interface

Example

- Mocks usage

Embracing TDD

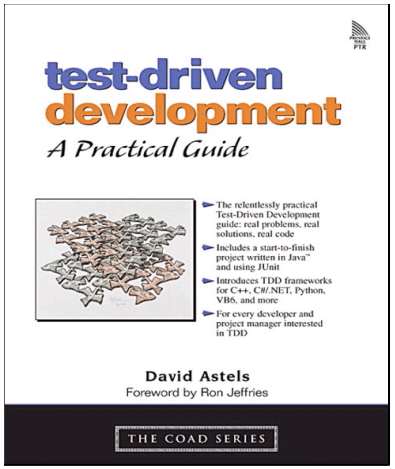
- Do not start big!
- Start new tasks with TDD
  - Add tests to code that you need to change or maintain – but only to small parts
  - Grow an architecture
  - Use technology migration or project handover as an opportunity
- TDD limitations
  - Some documentation is still needed
  - GUI must be implemented before being tested
  - Does not fully replace traditional (functional, integration, acceptance…) tests

Summary

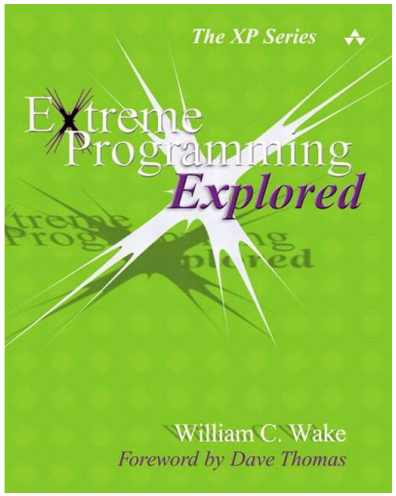
TDD promotes the development of high-quality code!

- Short feedback loop
- Detailed specification through examples
- Design emerges through development in small and safe steps – refactoring
- Clean design by focusing on implementation of callable and testable features
- TDD supports evolutionary development and reduces fear of changes
- No code without tests, as they:
  - Dictate the code
  - Verify it
  - Provide evidence that the software works
  - Act as documentation – working examples of how to invoke some code
- Confidence boost – “It works!”
- Reduced reliance on the debugger
- Some user acceptance tests be included into TDD process

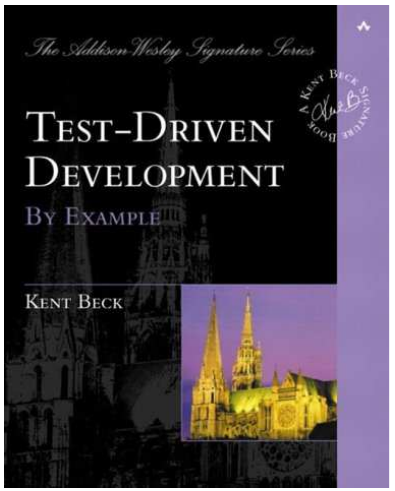
Resources (Books)



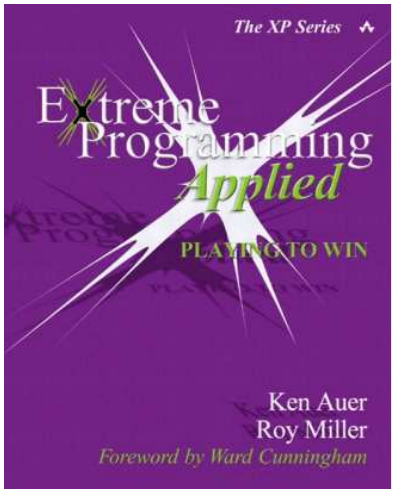
test-driven development:
A Practical Guide
Dave Astels



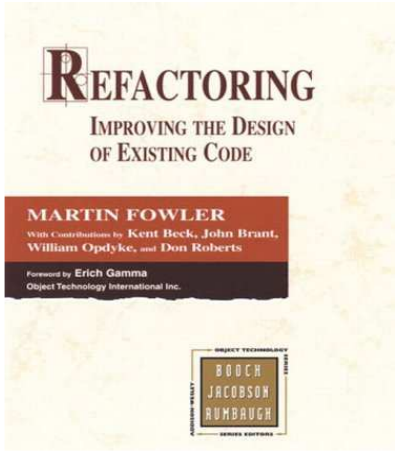
Test-Driven Development:
By Example
Kent Beck



Extreme Programming Explored
(The Green Book)
Bill Wake



Extreme Programming Applied:
Playing To Win (The Purple Book)
Ken Auer, Roy Miller



Refactoring:
Improving the Design of Existing Code
Martin Fowler

Resources (web)

- Behaviour driven development: http://behaviour-driven.org
- Mock Objects: http://www.mockobjects.com
- Mockito open source framework for Java, http://mockito.org/
- FEST – family of Java libraries for easy software testing, including assertions, Java reflection, Swing: http://code.google.com/p/fest/
- JUnit testing framework: http://www.junit.org/

Credits

Material for some slides originates from

- Craig Murphy, http://www.CraigMurphy.com