FIRST Principles

Produced by:

Dr. Siobhán Drohan (sdrohan@wit.ie)

Eamonn de Leastar (edeleastar@wit.ie)

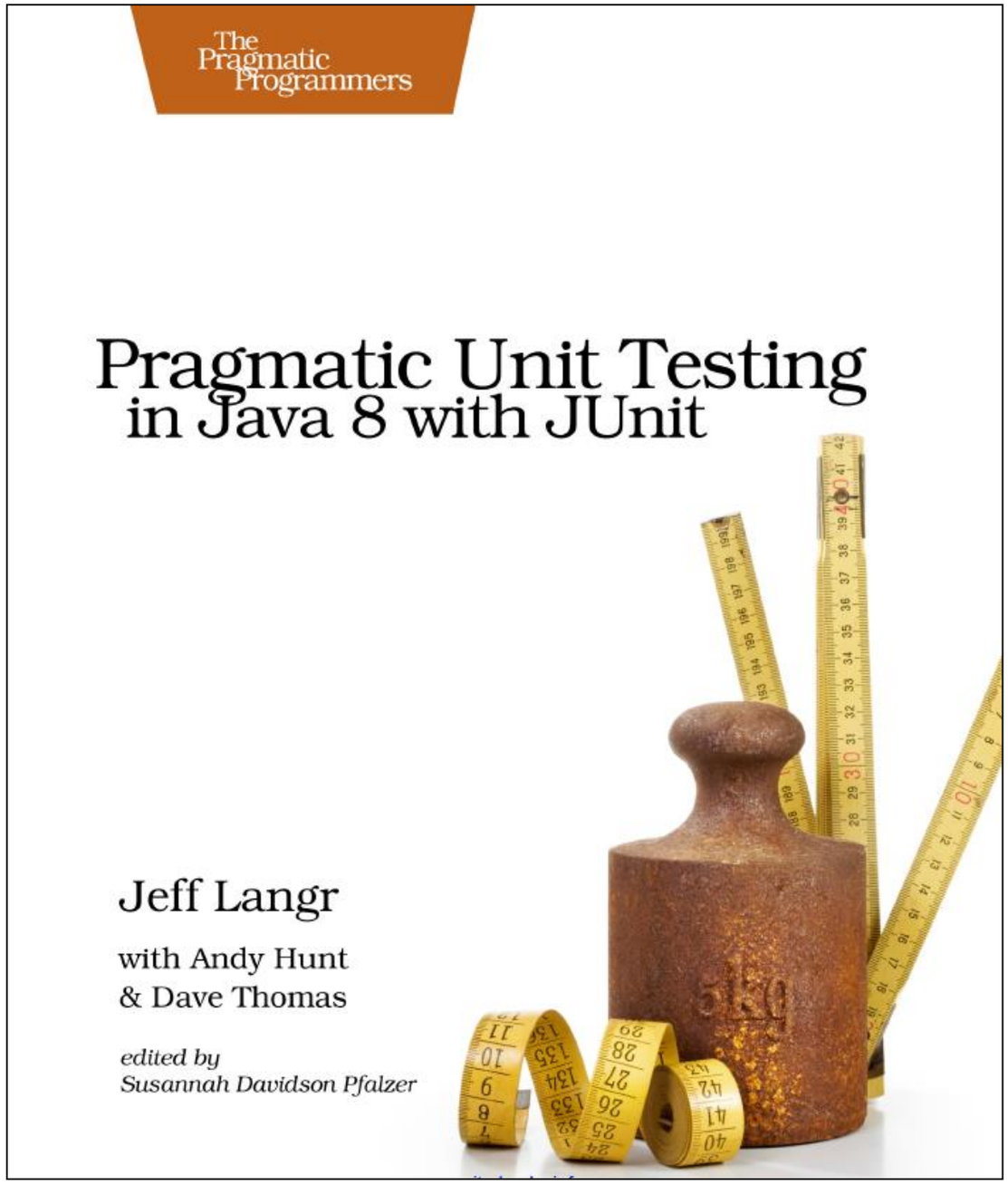
Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

FIRST Principles

Characteristics of quality tests:

- [F]ast
- [I]solate your tests
- [R]epeatable
- [S]elf-validating
- [T]imely



Source Code: https://pragprog.com/titles/utj2/source_code

[F]IRST: [F]ast

- Consider this scenario:
  - 2500 unit tests
  - Average test takes 200 ms
    → approx. 8 minutes to run test suite.

- Are you going to run an 8-minute suite of tests multiple times a day?

[F]IRST: [F]ast

- Consider this scenario:
  - 2500 unit tests
  - Average test takes 200 ms
    → approx. 8 minutes to run test suite.

- Are you going to run an 8-minute suite of tests multiple times a day?

- As your system grows, your unit tests will take longer and longer to run: 8 minutes easily turns into 15 or even 30.

[F]IRST: [F]ast

- **Tipping Point:** When your unit tests reach the point where it's painful to run them more than a couple times per day, you've tipped the scale in the wrong direction.

[F]IRST: [F]ast

- **Tipping Point:** When your unit tests reach the point where it's painful to run them more than a couple times per day, you've tipped the scale in the wrong direction.

- **Value/Health Diminishes:** The value of your suite of unit tests diminishes as their ability to provide continual, comprehensive, and fast feedback about the health of your system also diminishes.

[F]IRST: [F]ast

- **Tipping Point:** When your unit tests reach the point where it's painful to run them more than a couple times per day, you've tipped the scale in the wrong direction.

- **Value/Health Diminishes:** The value of your suite of unit tests diminishes as their ability to provide continual, comprehensive, and fast feedback about the health of your system also diminishes.

- **Confidence Diminishes:** When you allow your tests to fall out of favour, you and your team will question the investment you made to create them.

[F]IRST: [F]ast → Recommendations

- Keep your tests fast!
    - Keep your design clean
    - minimize the dependencies on code that executes slowly.

[F]IRST: [F]ast → Recommendations

- Keep your tests fast!
    - Keep your design clean
    - minimize the dependencies on code that executes slowly.

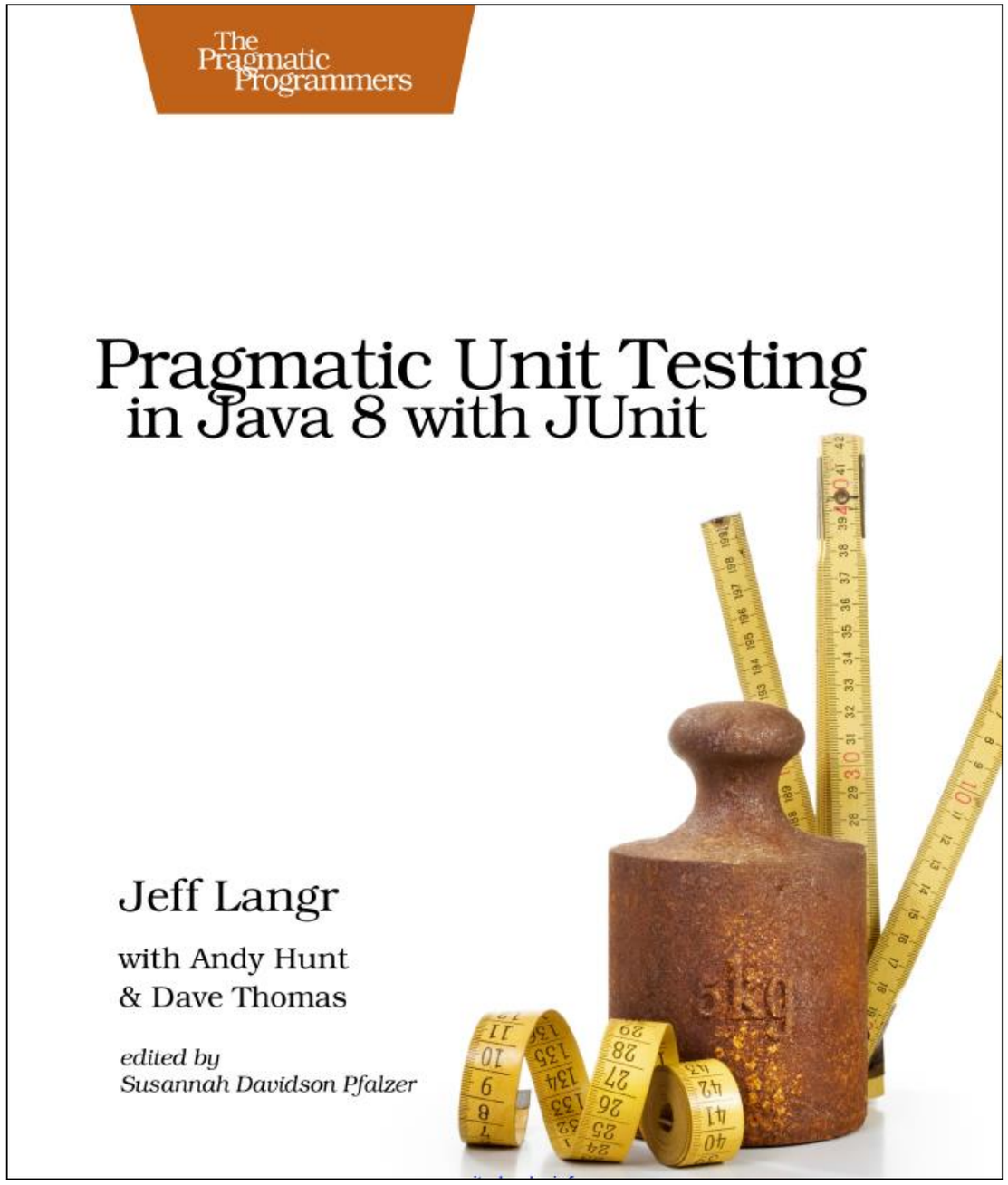
- If all your tests interact with code that ultimately always makes a database call, all your tests will be slow.

- This is where the Mock Objects technique can excel (more on this later).

FIRST Principles

Characteristics of quality tests:

- [F]ast
- [I]solate your tests
- [R]epeatable
- [S]elf-validating
- [T]imely



Source Code: https://pragprog.com/titles/utj2/source_code

F[I]RST: [I]solate your tests

- **Good unit tests focus on a small chunk of code to verify (i.e. cohesive):**
  - That's in line with our definition of *unit*; the more code that your test interacts with, directly or indirectly, the more things are likely to go awry.

F[I]RST: [I]solate your tests

- **Good unit tests focus on a small chunk of code to verify (i.e. cohesive):**
  - That's in line with our definition of *unit*; the more code that your test interacts with, directly or indirectly, the more things are likely to go awry.
- **You should be able to run any one test at any time, in any order:**
  - Good unit tests also don't depend on other unit tests (or test cases within the same test method). You might think you're speeding up your tests by carefully crafting their order so that several tests can reuse some of the same expensively constructed data. But you're simultaneously creating an evil chain of dependencies. When things go wrong—and they will—you'll spend piles of time figuring out which one thing buried in a long chain of prior events caused your test to fail.

F[I]RST: [I]solate your tests

Single Responsibility Principe (SRP) and Testing:

- SRP: classes should have only one reason to change.

F[I]RST: [I]solate your tests

Single Responsibility Principe (SRP) and Testing:

- SRP: classes should have only one reason to change.
- SRP provides great guideline for your test methods also. If one of your test methods can break for more than one reason, consider splitting it into separate tests. When a focused unit test breaks, it’s usually obvious why.

F[I]RST: [I]solate your tests

Single Responsibility Principe (SRP) and Testing:

- SRP: classes should have only one reason to change.
- SRP provides great guideline for your test methods also. If one of your test methods can break for more than one reason, consider splitting it into separate tests. When a focused unit test breaks, it's usually obvious why.
- It's easy to keep your tests focused and independent if each test concentrates only on a small amount of behaviour.
    - When you start to add a second assert to a test, ask yourself, "Does this assertion help to verify a single behaviour, or does it represent a behaviour that I could describe with a new test name?"

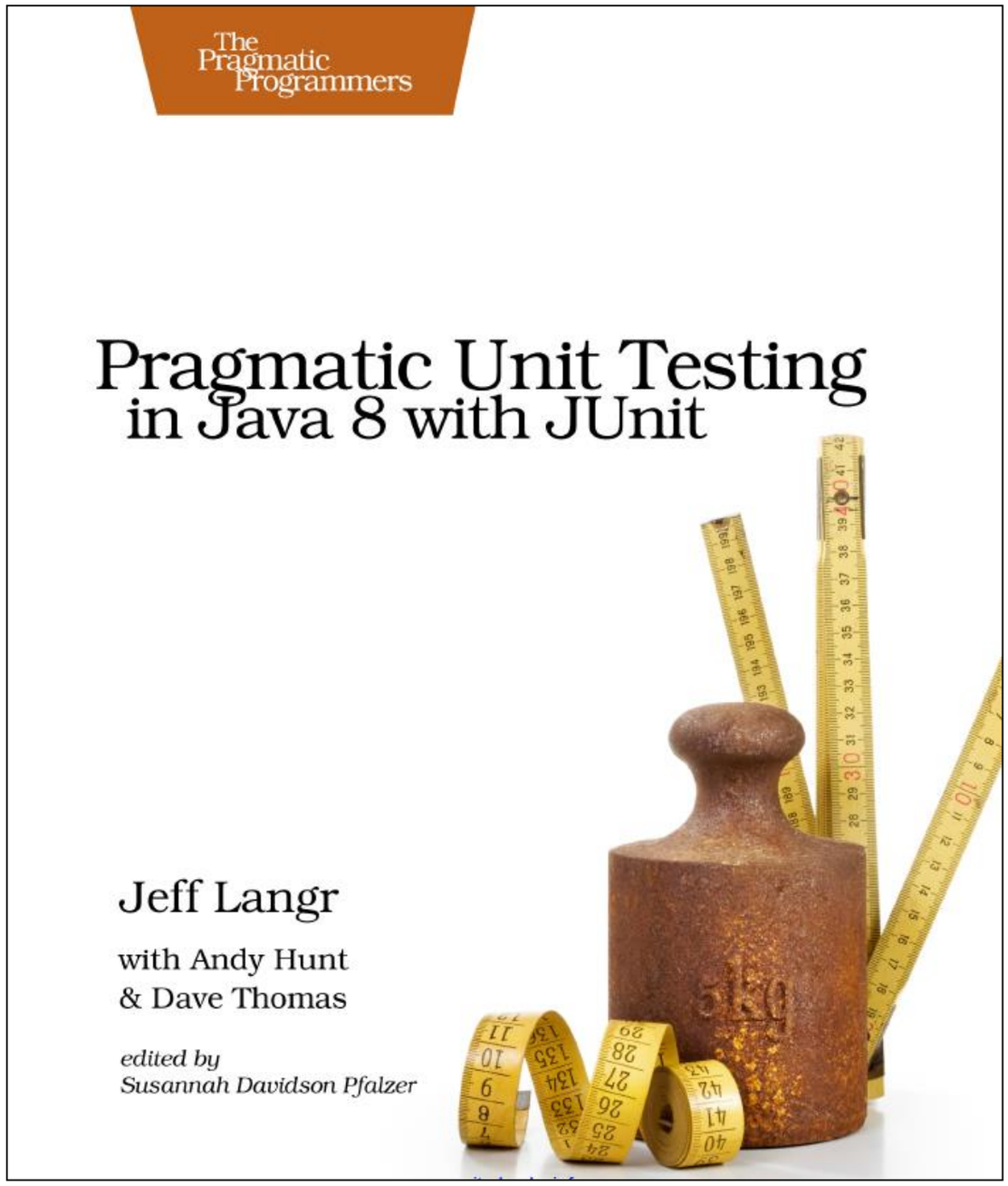
F[I]RST: [I]solate your tests

- Consider this database access scenario:
  - The code you're testing might interact with other code that reads from a database.
  - Data dependencies create a whole host of problems. Tests that must ultimately depend on a database require you to ensure that the database has the right data.
  - If your data source is shared, you have to worry about external changes (maybe out of your control) breaking your tests. Don't forget that other developers are often running their tests at the same time! Simply interacting with an external store increases the likelihood that your test will fail for availability or accessibility reasons.

FIRST Principles

Characteristics of quality tests:

- [F]ast
- [I]solate your tests
- [R]epeatable
- [S]elf-validating
- [T]imely



Source Code: https://pragprog.com/titles/utj2/source_code

FI[R]ST: Good tests should be [R]epeatable

- In test design, you provide an assertion that specifies what the outcome should be ***each and every time*** the test is run.

FI[R]ST: Good tests should be [R]epeatable

- In test design, you provide an assertion that specifies what the outcome should be ***each and every time*** the test is run.

- A repeatable test is one that produces the same results each time you run it.

- Without repeatability, you might be in for some surprises at the worst possible moments.
  - What's worse, these sort of surprises are usually bogus—it's not really a bug, it's just a problem with the test.
  - You can't afford to waste time chasing down phantom problems.

FI[R]ST: Good tests should be [R]epeatable

- To accomplish repeatable tests, you must ***isolate*** them from anything in the external environment not under your direct control; your system will inevitably need to interact with elements not under your control, however.

- We can use a ***mock object*** as one way to isolate the rest of the code under test and keep it independent from the volatility of time.

FI[R]ST: Good tests should be [R]epeatable

Consider this scenario…testing timestamps:

- Timestamps are moving targets, making it a bit of a challenge to assert what the creation timestamp should be.
- Well, we can't stop time, but we *can* fake it out. Or rather, we can fake out our code to think it's getting the real time, when it instead obtains the current time from a different source.

We will use the Java 8, java.time.Clock object to demonstrate faking.

FI[R]ST: Good tests should be [R]epeatable

java.time.Clock

- Allows alternate clocks to be plugged in as and when required.
- Simplifies testing by modelling a single instantaneous point on the time-line.

`static Clock`	**`systemUTC()`** Obtains a clock that returns the current instant using the best available system clock, converting to date and time using the UTC time-zone.

`abstract Instant`	**`instant()`** Gets the current instant of the clock.

FI[R]ST: Good tests should be [R]epeatable

The code we want to test

```
public class QuestionController {

private Clock clock = Clock.systemUTC();
    // ...
    public int addBooleanQuestion(String text) {
        return persist(new BooleanQuestion(text));
    }

    void setClock(Clock clock) {
        this.clock = clock;
    }
    // ...
    private int persist(Persistable object) {
        object.setCreateTimestamp(clock.instant());
        executeInTransaction((em) -> em.persist(object));
        return object.getId();
    }
}
```

iloveyouboss/16-branch-persistence/src/iloveyouboss/controller/QuestionController.java

FI[R]ST: Good tests should be [R]epeatable

Creates an **Instant** instance and stores it in the **now** local variable.

iloveyouboss/16-branch-persistence/test/iloveyouboss/controller/QuestionControllerTest.java

```
@Test
public void questionAnswersDateAdded() {
    Instant now = new Date().toInstant();
    controller.setClock(Clock.fixed(now, ZoneId.of("America/Denver")));
    int id = controller.addBooleanQuestion("text");

    Question question = controller.find(id);

    assertThat(question.getCreateTimestamp(), equalTo(now));
}
```

Injects the **now Instant** into the controller via the setter method.

When asked for the time, it will always return the **now Instant**—because the test previously *injected* it into the controller through the setter method.

FI[R]ST: Good tests should be [R]epeatable

- The persist() method obtains an instant from the injected clock instance and passes it along to the setCreateTimestamp() method.
- If no client code injects a Clock instance using setClock(), the clock defaults to the systemUTC clock as initialized at the field level.

```
private int persist(Persistable object) {
    object.setCreateTimestamp(clock.instant());
    executeInTransaction((em) -> em.persist(object));
    return object.getId();
}
```

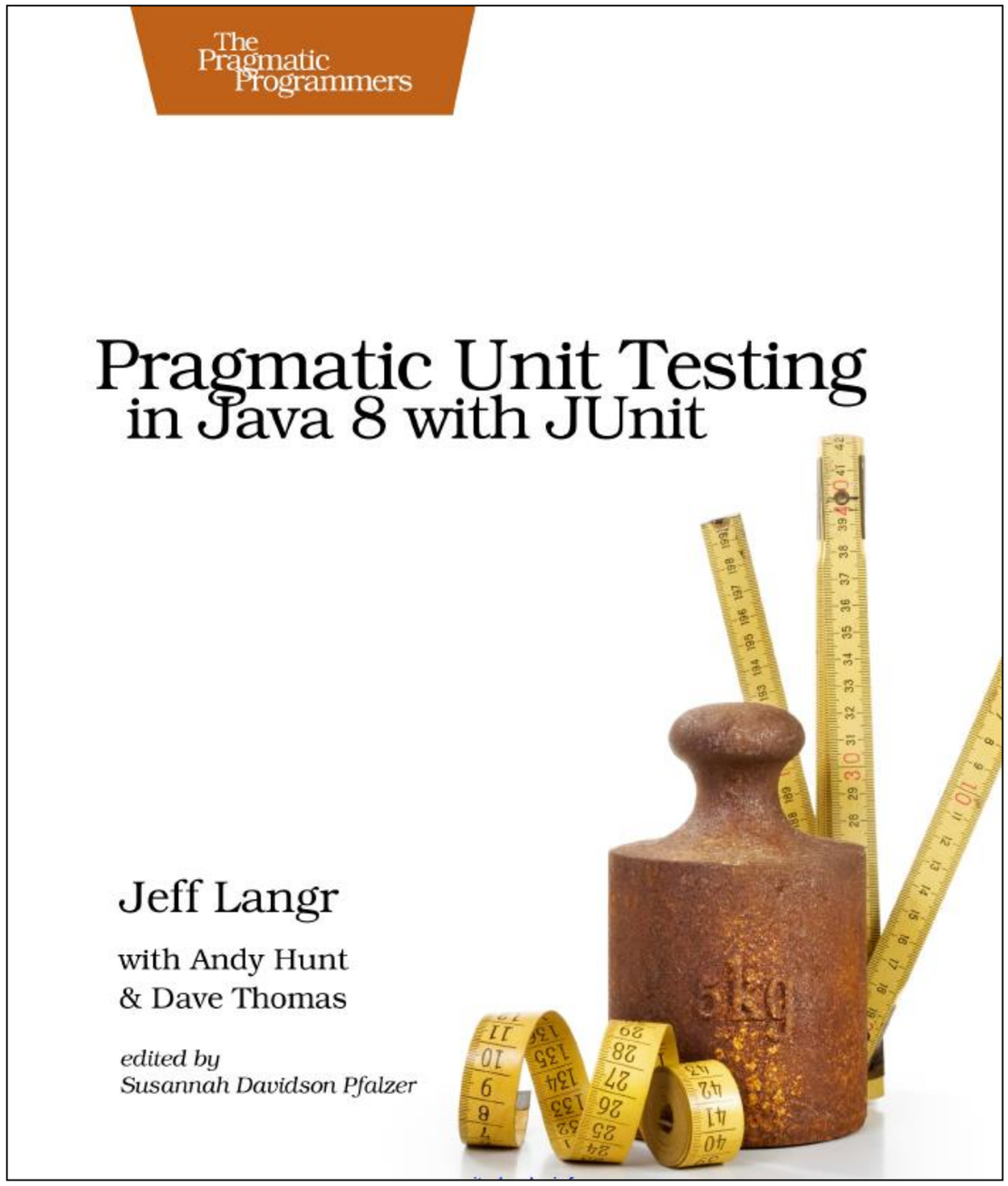
- Voila! The QuestionController doesn't know anything about the nature of the Clock, only that it answers the current Instant.

  - The clock used by the test acts as a *test double*—a stand-in for the real thing.

FIRST Principles

Characteristics of quality tests:

- [F]ast
- [I]solate your tests
- [R]epeatable
- [S]elf-validating
- [T]imely



Source Code: https://pragprog.com/titles/utj2/source_code

FIR[S]T: [S]elf-Validating

- Tests aren't tests unless they ***assert*** that things went as expected:
  - Avoid the temptation to manually verify the results of tests.
- Test should also be self-arranging; you must automate any setup your test requires. But remember the [I]solated part of FIRST:
  - tests requiring external setup (e.g. use of external db) violates [I].
  - any setup must ensure that you can run any one test at any time, in any order.

FIR[S]T: [S]elf-Validating

- For self-validating, the sky's the limit…as an ideal, imagine a system where:
  - you write tests for all changes you make.
  - whenever you integrate the code into your source repository, a build automatically kicks off and runs all the tests (unit and otherwise), indicating that your system is acceptably healthy.
  - the build server takes that vote of confidence and goes one step further, deploying your change to production.

- Embracing such continuous delivery (CD) approaches can significantly reduce the overhead of taking a need from inception to deployed product.

Infinitest tool for IDEs

As you make changes to your system, Infinitest identifies and runs (in the background) any tests that are potentially impacted. With Infinitest, testing moves from being a proactive task to being a gating criterion, much like compilation, that prevents you from doing anything further until you've fixed a reported problem.

CI Tools – Jenkins, TeamCity

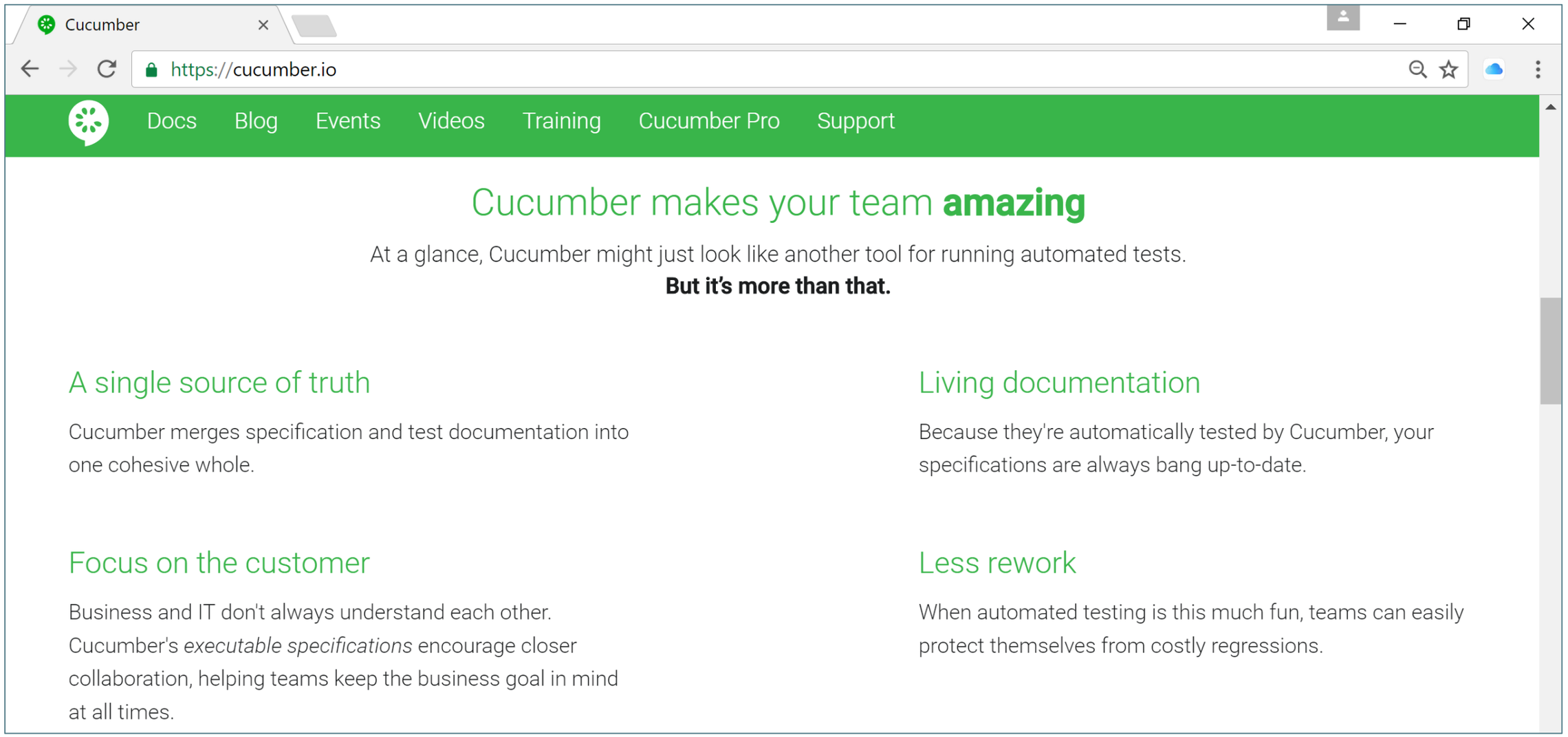
- On an even larger scale, you can use a continuous integration (CI) tool such as Jenkins or TeamCity.
- A CI tool watches your source repository and kicks off a build/test process when it recognises changes.

https://www.jetbrains.com/teamcity/

Suite of Videos on TeamCity:
https://www.jetbrains.com/teamcity/documentation/

http://jenkins-ci.org/

Behaviour Driven Development (BDD) tool



- A tool that facilitates BDD: https://cucumber.io/school

JIRA (https://www.atlassian.com/software/jira)

Go agile with ease

Whether you're a seasoned agile expert, or just getting started, JIRA Software unlocks the power of agile

Flexible planning

Scrum? Check. Kanban? Check. Mixed methodology? Check. JIRA Software's rich planning features enable your team to flexibly plan in a way that works best for them.

Accurate estimations

Estimations help your team become more accurate and efficient. Use story points, hours, t-shirt sizes, or your own estimation technique. JIRA Software supports them all.

Value-driven prioritization

Order user stories, issues, and bugs in your product backlog with simple drag and drop prioritization. Ensure stories that deliver the most customer value are always at the top.

Transparent execution

Whether your team is across the table or around the world, JIRA Software brings a new level of transparency to your team's work and keeps everyone on the same page.

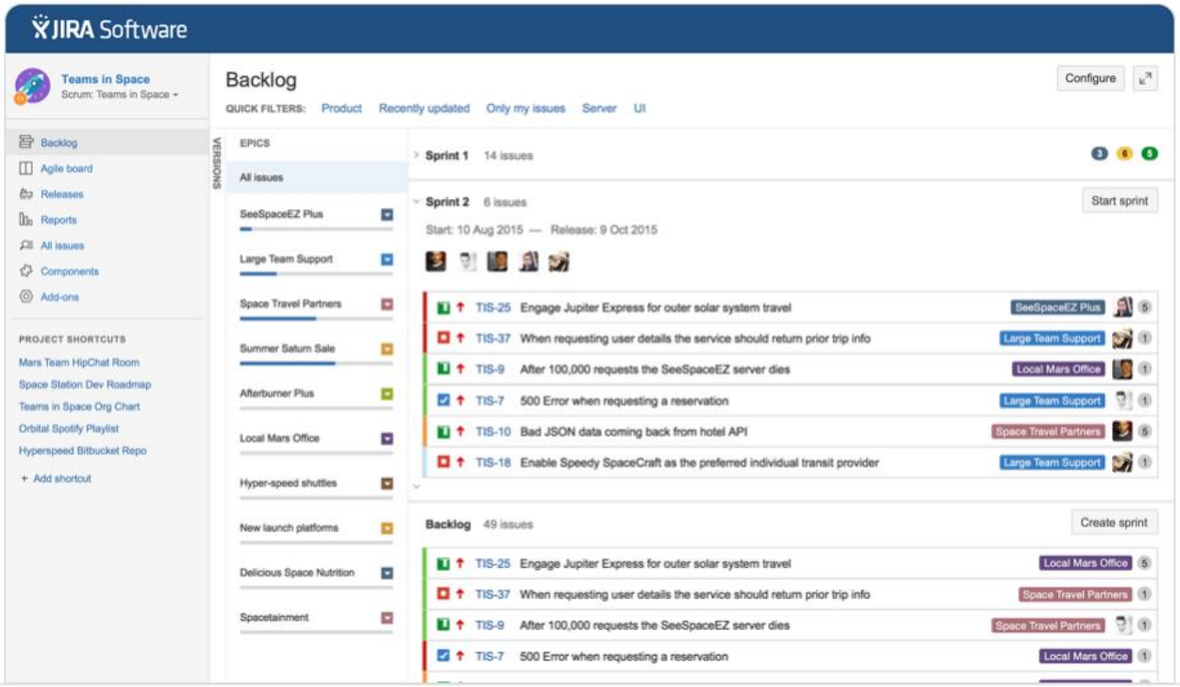
Actionable results

Extensive reporting functionality gives your team critical insight into their agile process. Backed by data, retrospectives are more data-driven and actionable than ever before.

Scalable evolution

Add and change issue types, fields, and workflows as your team evolves. JIRA Software is agile project management designed for teams of every shape and size.

JIRA (https://www.atlassian.com/software/jira)



Plan

Create user stories and issues, plan sprints, and distribute tasks across your software team.

Track

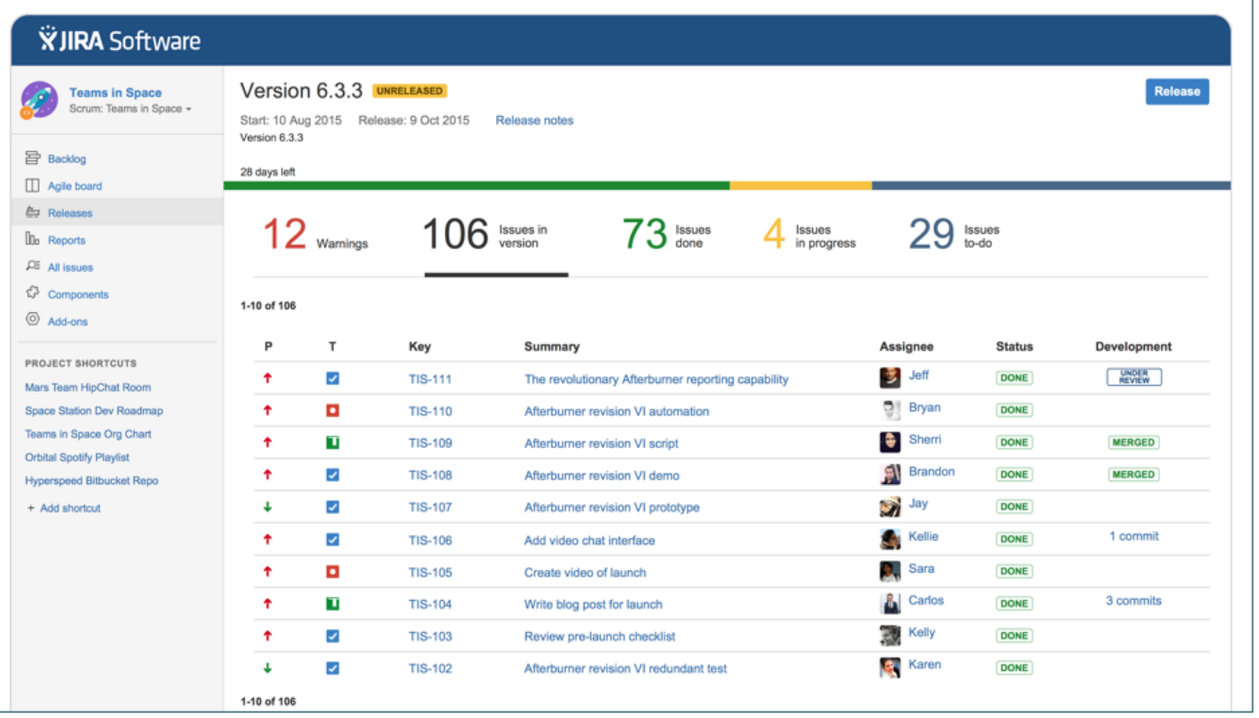
Prioritize and discuss your team's work in full context with complete visibility.

Release

Ship with confidence and sanity knowing the information you have is always current.

Report

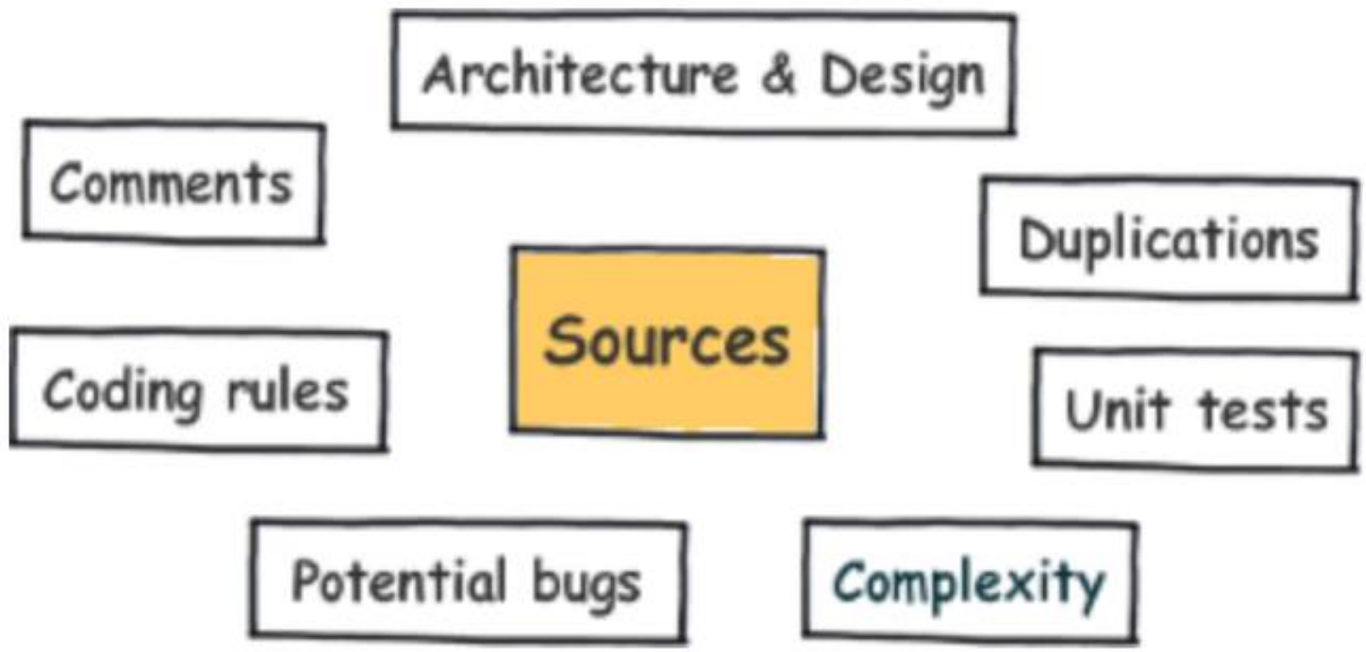
Improve team performance based on real-time, visual data you can use.



SonarCube (http://www.sonarqube.org/)

All in one

SonarQube is an open platform to manage code quality. As such, it covers the 7 axes of code quality:



SonarCube helps control your Technical Debt.

Wikipedia: Technical debt is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution."

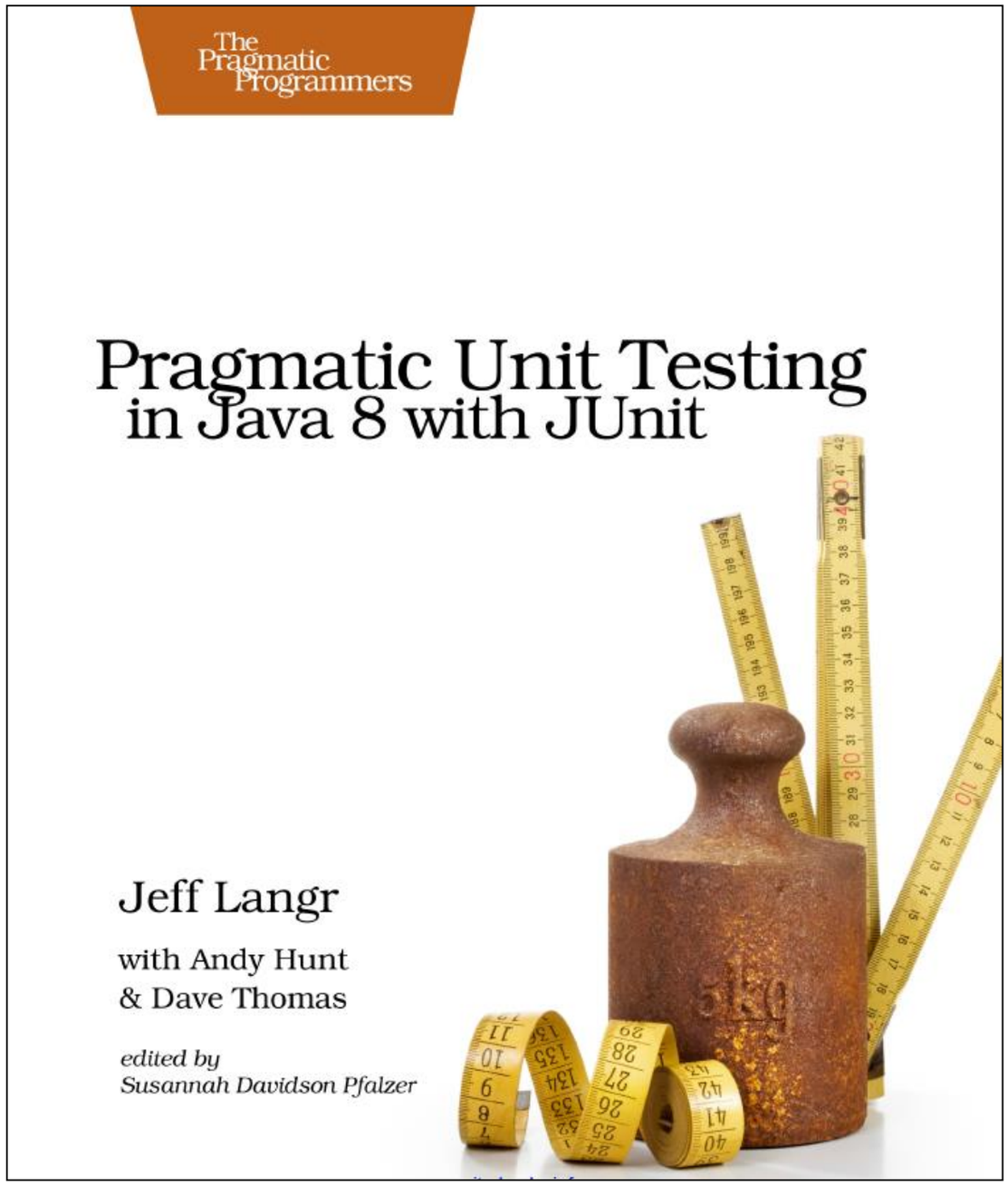
Languages covered

More than 20 programming languages are covered through plugins including Java, C#, C/C++, PL/SQL, Cobol, ABAP…

FIRST Principles

Characteristics of quality tests:

- [F]ast
- [I]solate your tests
- [R]epeatable
- [S]elf-validating
- [T]imely



Source Code: https://pragprog.com/titles/utj2/source_code

FIRS[T]: [T]imely

- You can write unit tests at virtually any time. You should focus on writing unit tests in a timely fashion.

- Many test-infected dev teams have guidelines or strict rules around unit testing. Some use review processes or even automated tools to reject code without sufficient tests.

- Keeping atop good practices like unit testing requires continual vigilance.