LOYOLA UNIVERSITY CHICAGO

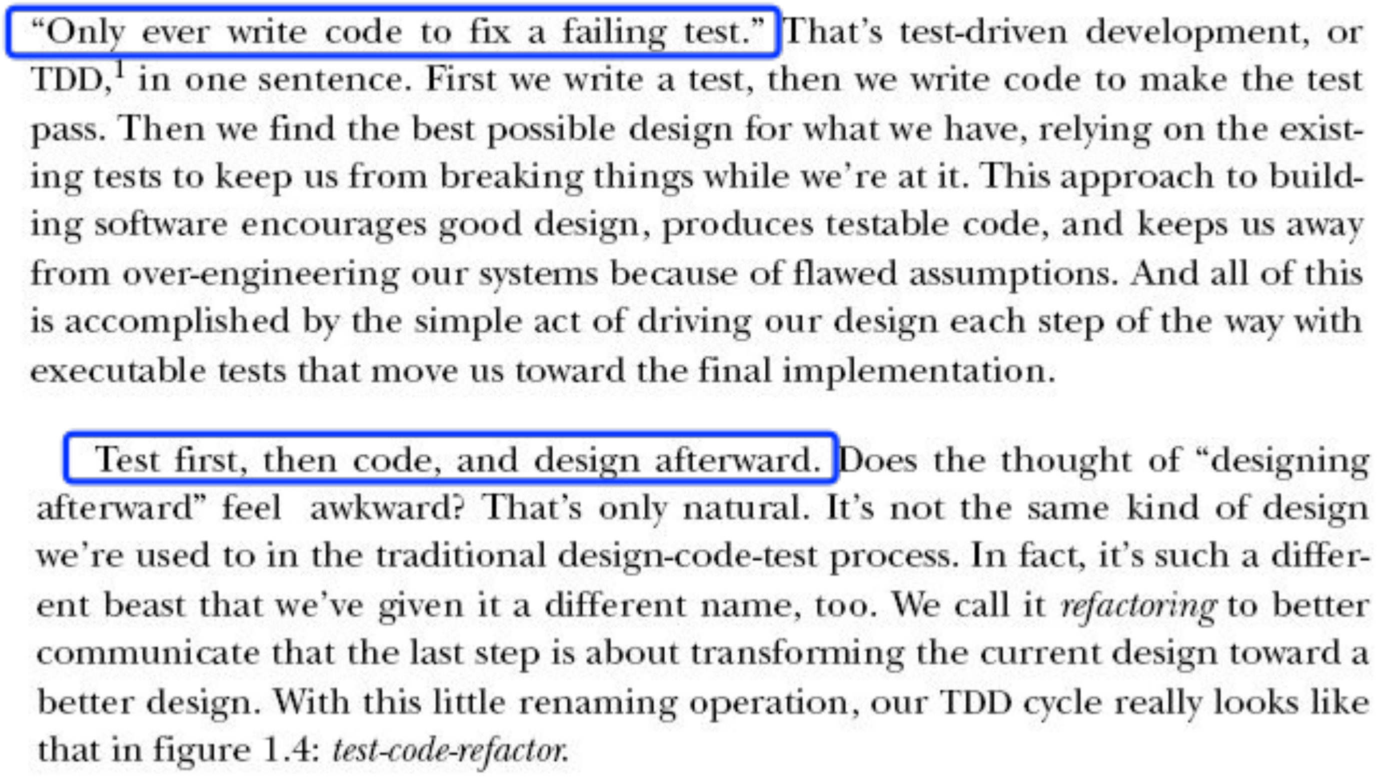
COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Week 1 Topic: Requirements

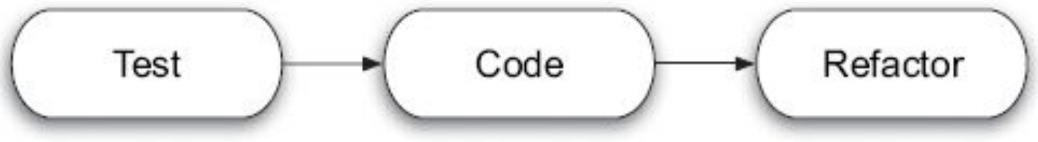
- Requirements – functional vs. nonfunctional
  - http://en.wikipedia.org/wiki/Non-functional_requirement
  - Functional vs. nonfunctional requirements overviews
- The importance of testing
  - http://www.examiner.com/article/importance-of-testing-software-development
  - Testable requirements material
- Test-Driven Development (TDD) and Junit
  - http://en.wikipedia.org/wiki/Test-driven_development

And Now for Something Completely Different … Test-Driven Development (TDD)



"Only ever write code to fix a failing test." That's test-driven development, or TDD,[1] in one sentence. First we write a test, then we write code to make the test pass. Then we find the best possible design for what we have, relying on the existing tests to keep us from breaking things while we're at it. This approach to building software encourages good design, produces testable code, and keeps us away from over-engineering our systems because of flawed assumptions. And all of this is accomplished by the simple act of driving our design each step of the way with executable tests that move us toward the final implementation.

Test first, then code, and design afterward. Does the thought of "designing afterward" feel awkward? That's only natural. It's not the same kind of design we're used to in the traditional design-code-test process. In fact, it's such a different beast that we've given it a different name, too. We call it *refactoring* to better communicate that the last step is about transforming the current design toward a better design. With this little renaming operation, our TDD cycle really looks like that in figure 1.4: *test-code-refactor.*



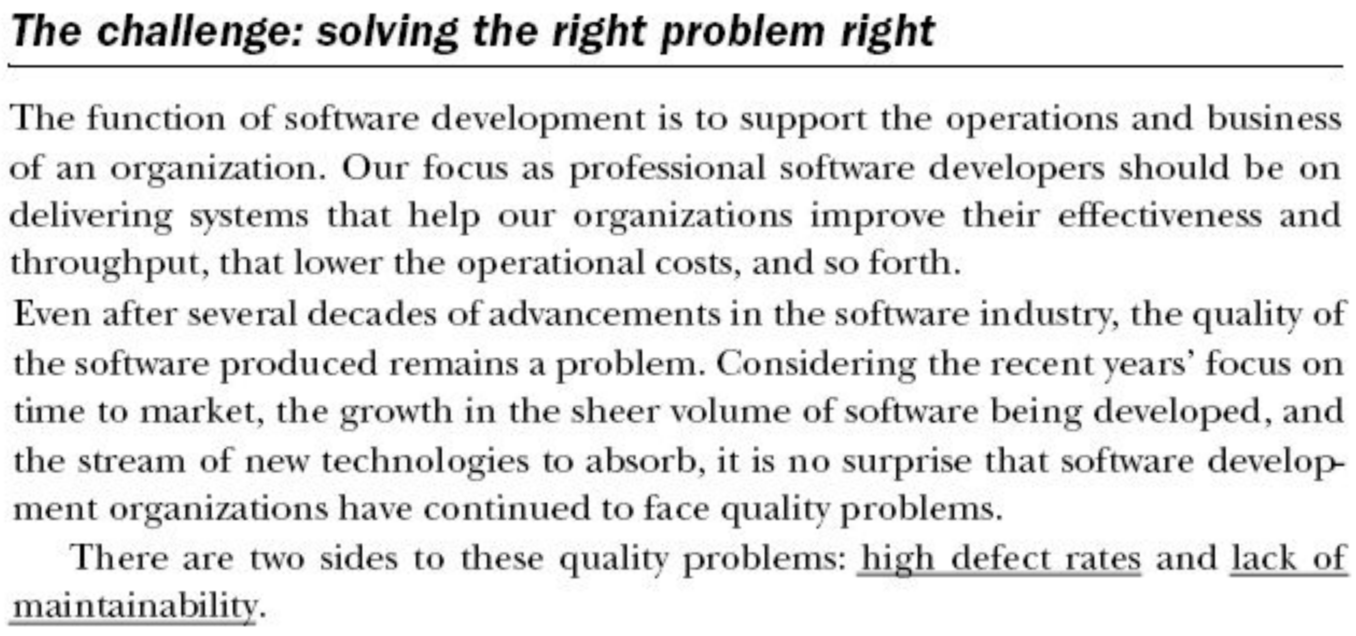
Note: we must reuse our tests to make sure the code continues to work after the refactoring!

Figure 1.4 Test-code-refactor is the mantra we test-driven developers like to chant. It describes succinctly what we do, it's easy to spell, and it sounds cool.

Source: Koskela: Test Driven: Practical TDD and Acceptance TDD for Java Developers

Why Do Test-Driven Development?

- There are two major quality-related problems in development: high defect rates and lack of maintainability.



The challenge: solving the right problem right

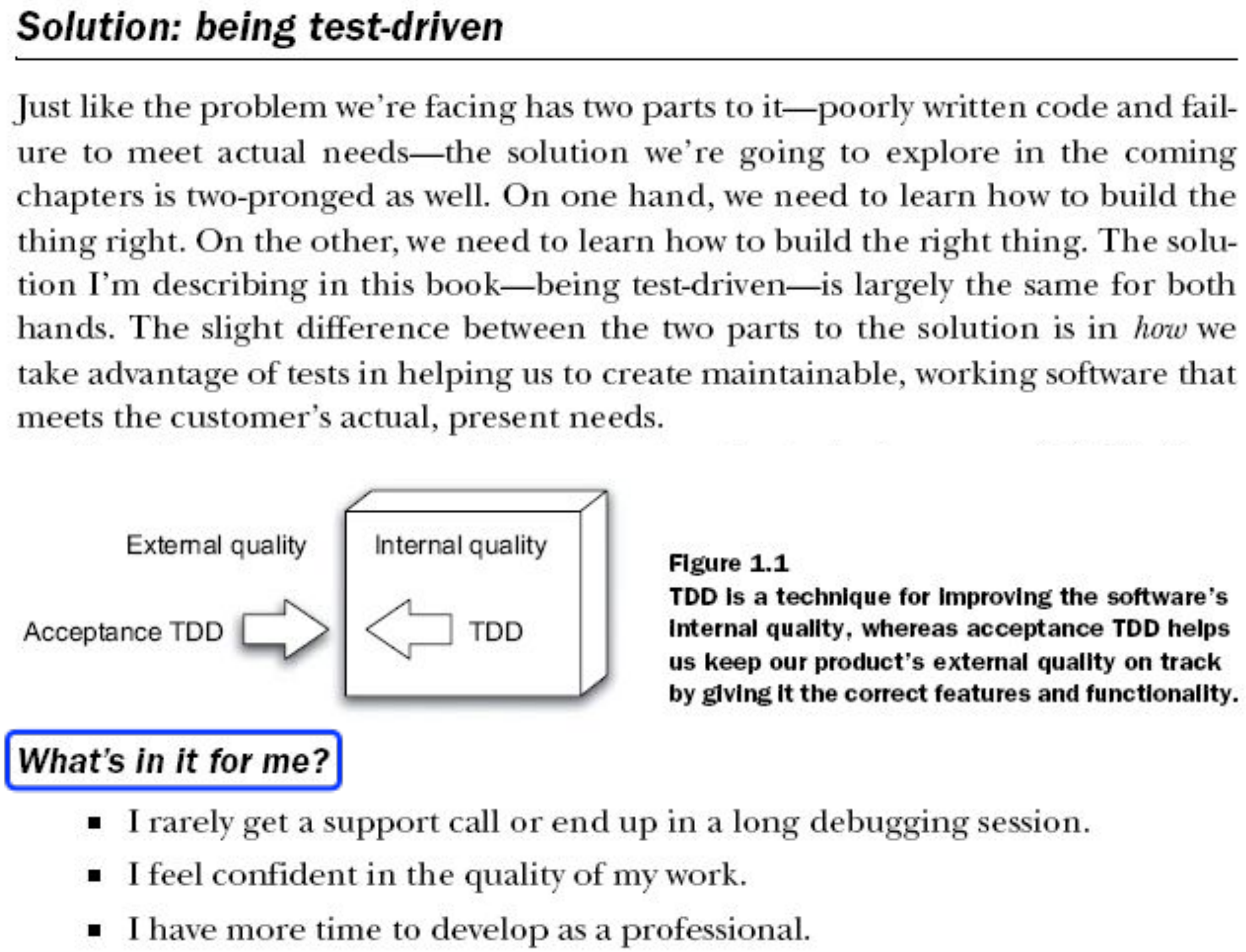
The function of software development is to support the operations and business of an organization. Our focus as professional software developers should be on delivering systems that help our organizations improve their effectiveness and throughput, that lower the operational costs, and so forth.

Even after several decades of advancements in the software industry, the quality of the software produced remains a problem. Considering the recent years' focus on time to market, the growth in the sheer volume of software being developed, and the stream of new technologies to absorb, it is no surprise that software development organizations have continued to face quality problems.

There are two sides to these quality problems: high defect rates and lack of maintainability.

Why Do Test-Driven Development?

- **TDD can be used to solve both of these problems**



Solution: being test-driven

Just like the problem we're facing has two parts to it—poorly written code and failure to meet actual needs—the solution we're going to explore in the coming chapters is two-pronged as well. On one hand, we need to learn how to build the thing right. On the other, we need to learn how to build the right thing. The solution I'm describing in this book—being test-driven—is largely the same for both hands. The slight difference between the two parts to the solution is in *how* we take advantage of tests in helping us to create maintainable, working software that meets the customer's actual, present needs.

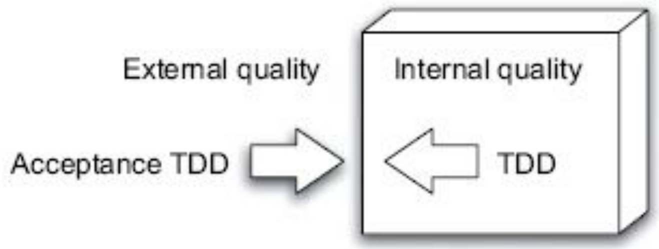


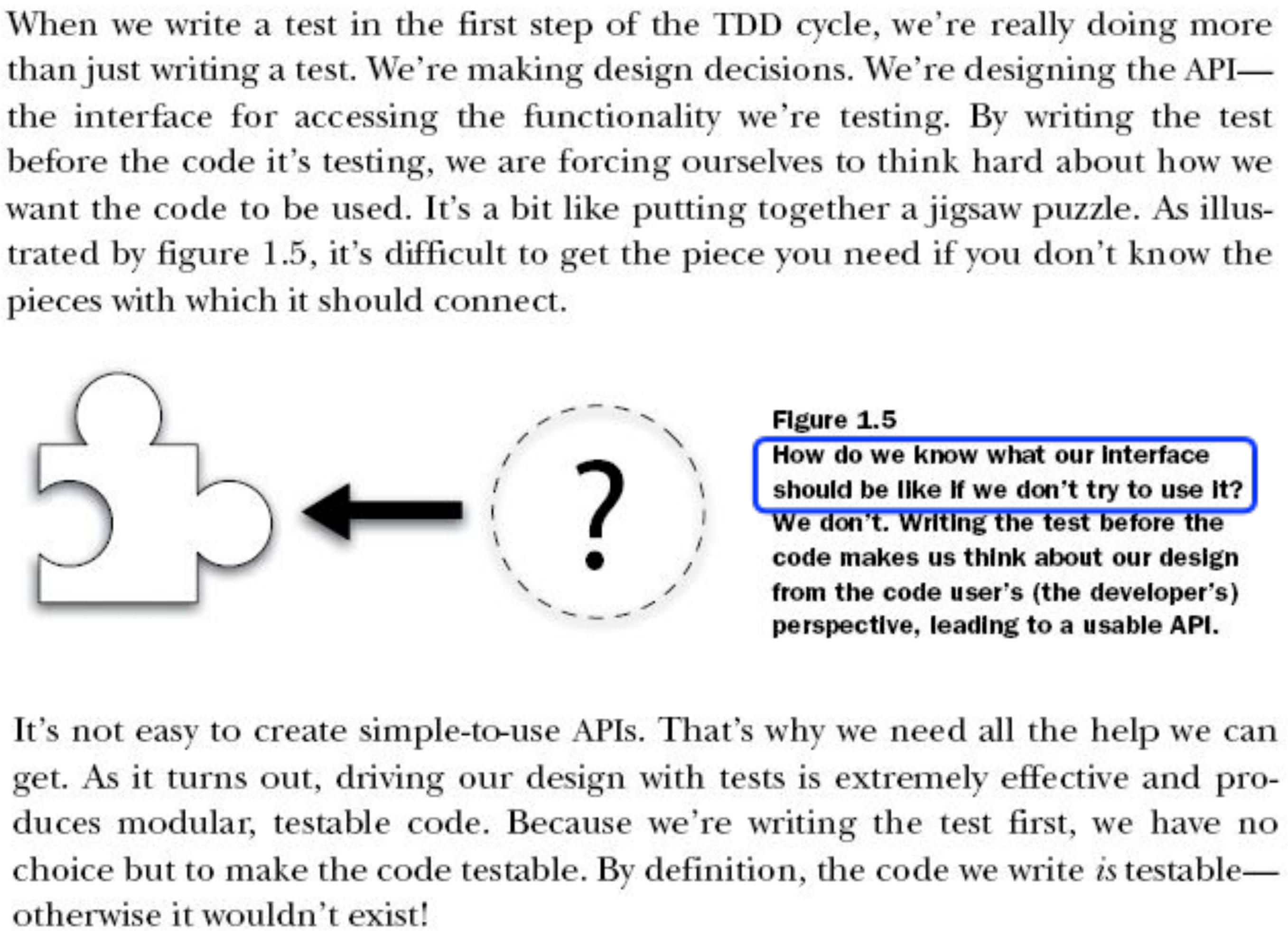
Figure 1.1
TDD is a technique for improving the software's internal quality, whereas acceptance TDD helps us keep our product's external quality on track by giving it the correct features and functionality.

What's in it for me?

- I rarely get a support call or end up in a long debugging session.
- I feel confident in the quality of my work.
- I have more time to develop as a professional.

TDD and API Design

- **Writing tests first is key to good program API design**



When we write a test in the first step of the TDD cycle, we're really doing more than just writing a test. We're making design decisions. We're designing the API—the interface for accessing the functionality we're testing. By writing the test before the code it's testing, we are forcing ourselves to think hard about how we want the code to be used. It's a bit like putting together a jigsaw puzzle. As illustrated by figure 1.5, it's difficult to get the piece you need if you don't know the pieces with which it should connect.

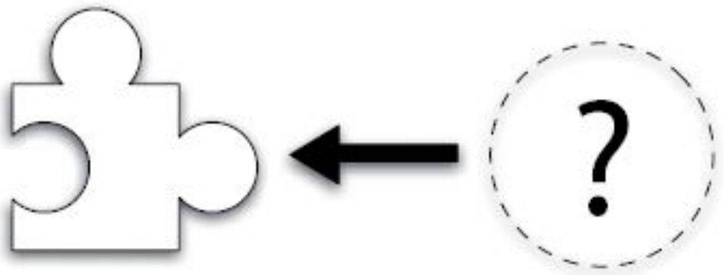
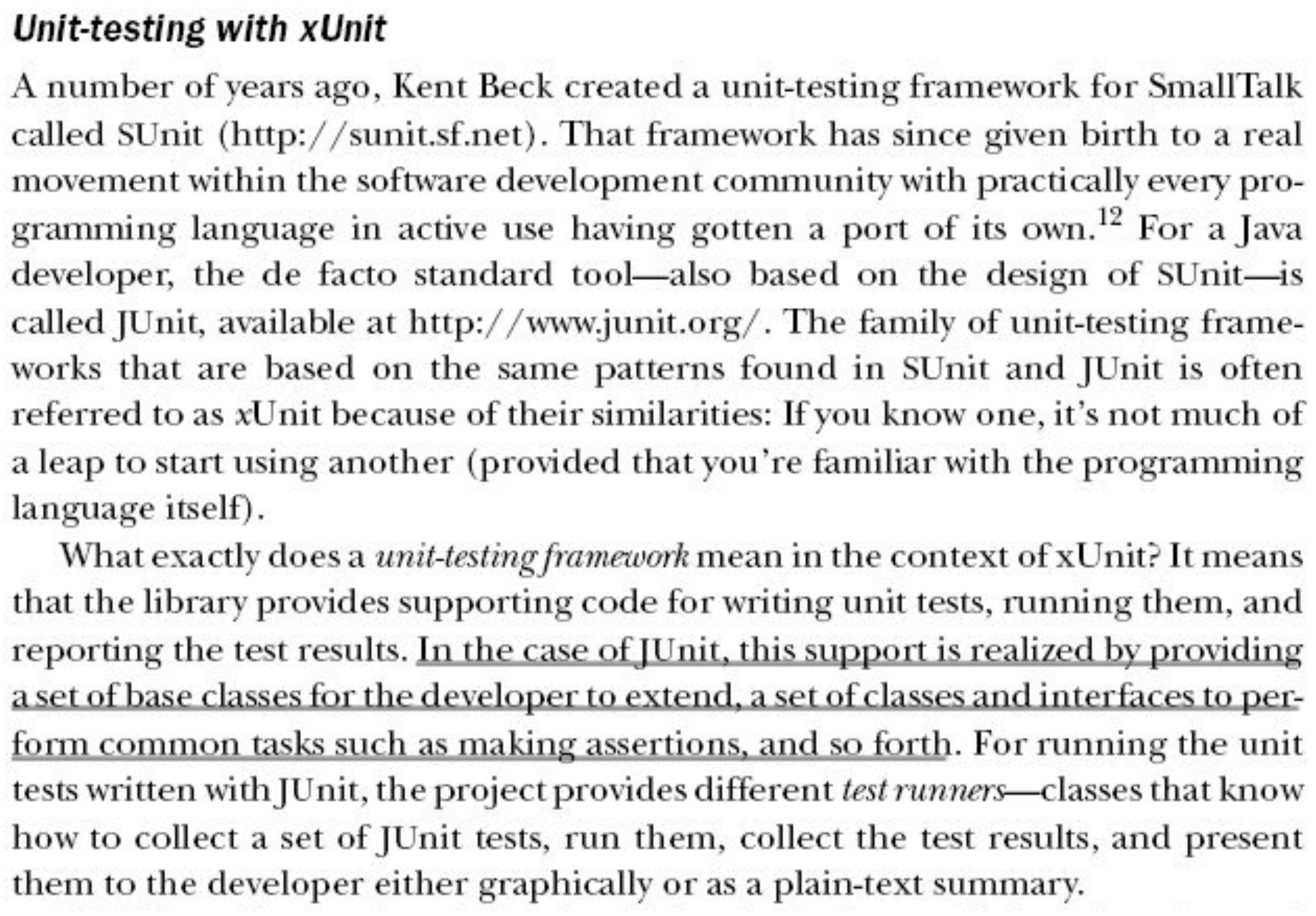


Figure 1.5
How do we know what our interface should be like if we don't try to use it? We don't. Writing the test before the code makes us think about our design from the code user's (the developer's) perspective, leading to a usable API.

It's not easy to create simple-to-use APIs. That's why we need all the help we can get. As it turns out, driving our design with tests is extremely effective and produces modular, testable code. Because we're writing the test first, we have no choice but to make the code testable. By definition, the code we write *is* testable—otherwise it wouldn't exist!

Tool Support for TDD in Java – JUnit

- **JUnit is a unit testing framework for Java**



Unit-testing with xUnit

A number of years ago, Kent Beck created a unit-testing framework for SmallTalk called SUnit (http://sunit.sf.net). That framework has since given birth to a real movement within the software development community with practically every programming language in active use having gotten a port of its own.[12] For a Java developer, the de facto standard tool—also based on the design of SUnit—is called JUnit, available at http://www.junit.org/. The family of unit-testing frameworks that are based on the same patterns found in SUnit and JUnit is often referred to as *x*Unit because of their similarities: If you know one, it's not much of a leap to start using another (provided that you're familiar with the programming language itself).

What exactly does a *unit-testing framework* mean in the context of xUnit? It means that the library provides supporting code for writing unit tests, running them, and reporting the test results. In the case of JUnit, this support is realized by providing a set of base classes for the developer to extend, a set of classes and interfaces to perform common tasks such as making assertions, and so forth. For running the unit tests written with JUnit, the project provides different *test runners*—classes that know how to collect a set of JUnit tests, run them, collect the test results, and present them to the developer either graphically or as a plain-text summary.