Test-Driven Development

Hakan Erdogmus
Kalemun Research, Ottawa, Ontario, Canada

Grigori Melnik
Microsoft Corporation, Redmond, Washington, U.S.A.

Ron Jeffries
XProgramming.com, Pinckney, Michigan, U.S.A.

Abstract
Test-driven development (TDD) is a software development approach using a growing scaffold of tests that guide and support the production of code. This entry describes TDD, explains the underlying dynamics, provides a small worked example, and offers a theory of how and why it works. It relates TDD to other approaches that employ a similar style of development and presents a summary of the empirical results about TDD's effectiveness. The entry also raises some of the known concerns and challenges about this development practice and provides pointers to solutions. TDD is a practice that has widespread impact on the software development lifecycle. Adopting it takes a great amount of discipline. However, we cannot say whether TDD is right for everyone and for all types of software: this entry should help the reader decide whether to explore further.

INTRODUCTION

Test-driven development (TDD)[1,2] is an incremental software development approach. It relies on automated regression tests, alternately written and made to work in a short cycle, to steer development activities. TDD was popularized by extreme programming,[3] of which it remains a central practice. In TDD, tests precede the production code that they exercise. The unique dynamic that defines TDD largely follows from this particular sequencing of activities. That production code is written after test code is TDD's most distinguishing aspect.

The term "test-driven" sometimes causes confusion. TDD is not a testing technique per se. It is a production technique—or strictly speaking, classical TDD is a coding practice—that relies on tests and continuous regression testing. As will be illustrated, *programmer tests* written during TDD differ in many aspects from tests produced during separate quality assurance activities. Programmer tests are compact, limited in scope, expressive, and execute fast. They are comprehensive only in their association, as examples, with the intended functionality, and programmer tests do not aspire for perfect coverage. They are not exploratory in intention: they don't aim to reveal very unusual interactions. Rather programming tests are guiding.

Fig. 1 illustrates gradual progression to TDD from a traditional phased, test-last pattern. The light and dark portions respectively represent production- and testing-related activities. In a strictly sequential development setting, such testing follows the completion of all implementation activity. With an incremental approach, development may proceed in smaller discrete chunks, with each chunk, or increment, being composed of a small production, or implementation, step followed by a corresponding unit-testing step. The developers may implement a distinguishable and testable piece of functionality, and follow up with a set of tests that exercise that functionality. This process, although it could be called *test oriented*, is not test-driven, for tests do not lead or guide the associated production activity. Inverting the sequence of production and testing activities for each increment gives rise to a process that is test-driven in nature if not conformant to full-out, ideal TDD. This near-ideal version is characterized by writing a set of tests for a set of related features, and then implementing in production code what the last batch of tests prescribes. The last level of progression consists in refining the process so that the increments are small enough to be captured by a single test: write a single test and follow it up by implementing just enough functionality to satisfy that test, all without breaking the previous tests. As testing moves to the forefront of development and becomes intertwined and integrated with coding, its nature and purpose also change. Additional traditional quality assurance activities, including validation and verification that incorporate other types of testing, may still follow to complement such a process.

Even though TDD is centrally a production technique rather than a design technique, it impacts the end product in significant ways. TDD forces developers to think first in

Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044180

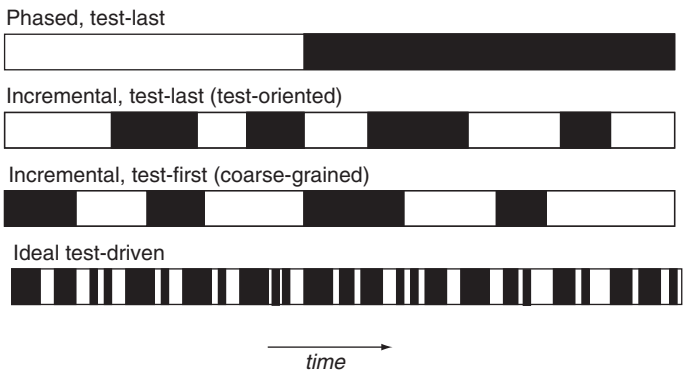

Fig. 1 Progression to TDD.

terms of interfaces rather than implementation mechanisms. This emphasis shapes the low-level design of the system, helping to keep it simple and clear. The result is a natural design with hooks that allow the system's functions to be exercised independently. Thus TDD leads to testable software.

The next section describes the mechanics of TDD at a high level. The "Aliases and Variations" section discusses related approaches and the aliases under which TDD is known. This is followed in the "Why Test-Driven Development Works" section by a theory explaining the factors that make TDD a plausible technique. The "Perceptions and Misconceptions" section focuses on common impressions of TDD. The "An Example" section provides a worked example of TDD's application to a small programming task. The "Dynamics" section follows up on the "How Test-Driven Development Works"section by elaborating on the resulting dynamics in terms of the underlying process's granularity and the resulting distribution of effort between alternating activities. The "Evidence of Effectiveness" section summarizes the empirical evidence on TDD: some of this evidence challenges the theory. Cognitive, social, and technical problems are covered in the "Challenges" section. Tool support is addressed in the "Tool Support" section and conclusions are provided in the final section.

HOW TEST-DRIVEN DEVELOPMENT WORKS

At its simplest, TDD is the practice of writing each piece of production code in direct response to a test. The test fails; we write the production code lines until the test passes. This is however only a rough description; the actual dynamics is more complicated.

In TDD, we do write tests before we write code, but we don't write all the tests before we write code. In general, we write just one small test at a time, make it work as quickly as possibly (typically in a few minutes), then write another test, make it work, and so on. The tests grow one at a time, and the code grows to meet the requirements and constraints that the tests provide. The tests get more and more challenging, and the code becomes more and more capable. We'll examine a real code example later on, but here's a scenario to illustrate how such a process works in general.

Suppose that, in a payroll situation, we were to pay people their base pay for up to 40 hr a week, one-and-a-half times their base for all overtime—work over 40 hr—and double their base pay for overtime work on Sunday. These requirements sound pretty simple, and we could imagine coding it all up and then testing a bit. But doing TDD, we would proceed differently. First, we might write a test to process the pay for someone who worked, with a base of 10 Euros per hour, for 40 hr. The test would assert that the person should earn 400 Euros. The test wouldn't run at all at first, because we have no code yet. But the code is easy enough: pay equals hours times base. Now the assertion is met.

Then we might write a test to process the pay for some other number of hours, perhaps 55. This would assert that the person's pay was 40 times 10, plus 15 (55 minus 40) times 10 times 1.5. That's 625. We run our test and it fails: it pays not 625, but 550. So we modify the code to account for overtime, perhaps compute pay as hours up to 40 times 10, plus hours above 40 times 10 times 1.5. Again the test runs.

We might go on now to a test about Sunday, but we would be wiser instead to look at our code first, now that it works, and see if it is clean enough. We might observe some shortcuts that we took, or we might improve some variable names or method signatures. We might even change the algorithm a bit. It may make more sense to compute the pay this way: all hours times base, plus hours above 40 times half the base. Either way, when the tests run, we take the occasion to look at the code and see if it could use a little improvement. And it usually can.

Then we'd move on to the Sunday calculation, and so on. If we had already seen the possibility of changing the code as described in the regular overtime calculation, we'd probably find the Sunday change to be a bit simpler, though it may still be somewhat tricky. If we hadn't, then the code to make Sunday overtime would begin to look a bit awkward, and on our next pause, we would see more reasons to improve it.

This is the basic flow of TDD. Write a test that fails, make it work, improve the code. Because most of the TDD

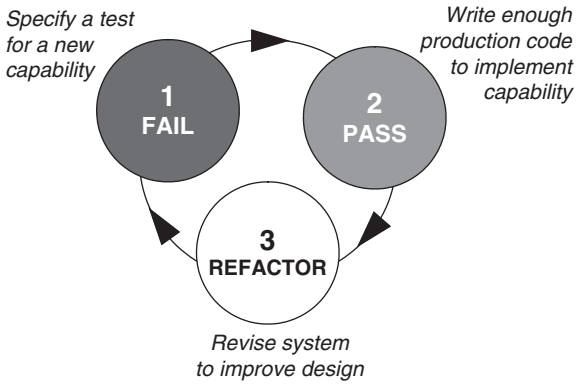

Fig. 2 TDD's flow.

tools use a red signal when tests don't work, and green when they do, we refer to this flow, illustrated in Fig. 2, as Red–Green-Refactor. : write a failing test and step into red; make the test succeed, going green; improve the code by refactoring and staying in the green; repeat. (Refactoring is the term of art for improving the design of existing running code, as described in Martin Fowler's book *Refactoring: Improving the Design of Existing Code*.[4] As Fowler puts it: "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 transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.")

There is more to TDD than this explanation, including interesting variations, but the essential idea of beginning with a simple test, making that test work, and improving the code remains constant.

At first TDD might appear tedious or slow. Yet most people who develop skill at applying TDD report that it provides a pleasing rhythm to the work, together with a sense of confidence that comes from the code always being supported by a scaffolding of tests. The scaffolding ensures that the code continues to do what we intended. Better yet, when we return to this code later, as we so often seem to do, changes are made easier for two reasons. First, we have the tests, which allow us to *regress* the system against them whenever we are in doubt. Second, because we've been keeping the code clean as part of our process, it's likely to be clear and easy to understand.

A successful software project is often built on the foundation of many practices, not just a few. Teams that use TDD find that its use will impact their planning process, which can become much more iterative as the work naturally falls into smaller pieces. They find that their specification activities become more streamlined, as they focus more and more on precise examples rather than on text descriptions that are too easily misunderstood. Our focus here is on TDD alone, although TDD is often used in practice in conjunction with other synergistic practices, refactoring[4,5] being the most obvious one.

ALIASES AND VARIATIONS

Since its introduction, the classical TDD process has seen both deliberate and inadvertent adaptations. It has come to represent practices with sometimes subtle and other times significant variations in style, granularity, level of application, and underlying dynamic. It has also become to be known under slightly different names, each stressing a different aspect. In this entry, TDD refers to the original, classical version as described by Beck and others.[1,6] This is the particular practice that we focus on. For clarity, it is nevertheless worthwhile to mention common aliases and variations, highlighting the differences.

Common Aliases

In folk usage, the various aliases result from the substitution of "test first" for "test-driven" and "programming" and "design" for "development". Hence the combinations test-first development, test-first programming, test-driven design, and test-first design.

The term "test first" correctly stresses that tests come before implementation. The implication is that if tests are systematically written after the production code that they exercise, the resulting technique would not be compliant even if the code, both production and test, is built iteratively and incrementally. The term "programming" stresses that the technique is essentially a code development practice and meant to be applied by programmers. While neither emphasis is wrong, none of the resulting combinations have an explanatory advantage over the original term.

The use of the term "design" is more problematic. The term attempts to emphasize the position that programming and program design go hand in hand, and that the practice makes (as opposed to helps) the design to emerge organically rather than being determined up front. TDD certainly includes that characteristic, but it isn't TDD per se that prescribes the design: it is the developer. TDD provides focus to the developer so that design decisions can be taken along the way with the help of tests. However, TDD doesn't provide any technical guidance on what those decisions might be. In fact, TDD works the same way regardless of whether there is an implicit or explicit design, or whether the design emerges gradually or is determined up front. Instead, in TDD, the developer is open to design and need not be fully committed to any preconceived notions, nor even to what is currently in the code. The developer can start from no design, very little design, or a well-articulated design. TDD does not stipulate one or the other, but if there is a target, it helps to reach that target by

providing a direction, without the burden of a commitment. For TDD to work, the position that it is a replacement for up-front design activity is not necessary. It may even be harmful. This point is subtle, but important. Consequently, we do not advocate the use of "test-driven design" or "test-first design" as an alias for TDD.

Common Variations

An inadvertent variation obtained by reversing the sequence of the main TDD activities is common in practice. Instead of writing tests first, why not still write them incrementally, but after implementing a small bit of functionality? After all, wouldn't such a strategy be more natural, more consistent with what we have been taught: design, implement, then test? And why should the order matter so long as the developer writes those tests? This unnamed variation, which we could call test-oriented development, could very well be effective. It just is not TDD. While in the course of applying TDD, this pattern may happen occasionally for valid reasons or due to slippage, when it dominates, the nature and rhythm of the practice change significantly. The consequences are also different: tests no longer affect how the developer thinks about the solution and focuses on a small bit. In fact, they no longer drive development. They may also suddenly become optional, and portions of the code risk becoming untestable. Test-oriented development is probably less sustainable than TDD in the long term. We consider it to be a different practice inspired by TDD rather than a legitimate variation faithful to TDD's main tenets.

Faithful variations do emerge along an entirely different dimension when a TDD-like dynamic is applied at higher levels, for example, at the system, subsystem, or service levels. Instead of focusing on the isolated behavior of relatively fine-grained program units (methods, classes, components, and modules), the emphasis is shifted to crosscutting, end-to-end functionality affecting compositions of such program units. As such the practice gets increasingly tied to addressing integration and ensuring successful implementation of user-level requirements. The tests' granularity gets much coarser than they are at the unit level. In effect, the practice moves from one of technical and individual nature to that of a team, requiring team-wide collaboration and the involvement of project roles beyond programming, such as customers, project managers, requirements experts, and business analysts. An inevitable side effect is that the extremely tight feedback loop that exists in TDD is relaxed. In *acceptance-test-driven development*[7]—and its subtle terminological variations *story-test-driven development* and *example-driven development*[8]—user-level requirements drive the development of new features. Scenarios (acceptance tests, story tests, or examples) capturing these requirements are expressed in a notation that customers or their proxies can more easily understand. Alternatively, the scenarios can be written using a developer-oriented framework collaboratively with the customers, users, or their proxies.

A close cousin of TDD is *behavior-driven development (BDD)*.[9] For some, BDD is more aptly named than TDD for its terminology helps decouple it from quality-assurance-type testing. BDD is a conceptual progression of TDD, with a more problem-friendly than solution-friendly vocabulary and notation. BDD generalizes the notion of a test to that of a behavioral specification. However the specifications are, rather than being generic, expressed in terms of concrete instances. Thus the specifications are much akin to TDD test cases than to traditional formal specifications that admit quantifiers and bound variables. BDD, in addition, encourages the use of the application domain's language in such specifications.

While BDD, being at the same level as TDD, is a substitute for classical TDD, acceptance TDD and its cousins are orthogonal and complementary to TDD.

WHY TEST-DRIVEN DEVELOPMENT WORKS

A Cutter Consortium survey of 196 companies rated TDD as the topmost influential practice on project success.[10] The only other practices and factors that had a statistically significant impact on a project's success were inspections and project duration, but frequent use of TDD was found to have the largest impact. So why should TDD deserve such attention? We first present a theoretical framework that explains why TDD works. We discuss empirical results regarding TDD's effectiveness in a later section.

Leading with tests (*test first*), *incremental development*, and frequent automated *regression testing* are TDD's foundational principles. In Fig. 3, these principals are placed at the top. The interplay between them is thought to result in a web of chain effects that impact development outcomes. These outcomes—programmers' *productivity*, software's *quality*, and software's *adaptability* (resilience to change)—are on the bottom of Fig. 3. Intermediate factors are shown in the middle.

Incremental development promotes decomposition of programming tasks into small, manageable pieces. This increases programmer's focus, with a potentially positive impact on productivity. Leading with tests ensures that the decomposed tasks are formalized before they are implemented as program code. When all programming activity is led by tests, the resulting software begins its life as testable and necessarily remains so, with the positive quality implications of the ability to independently exercise program behaviors. In addition, writing tests in advance makes tests obligatory and an integral part of the development process. Consequently, the amount of test assets increases. The amount of test assets in turn correlates positively with quality.

Test assets enable automated regression testing. Regression testing acts as a safety net. Tests can be run any

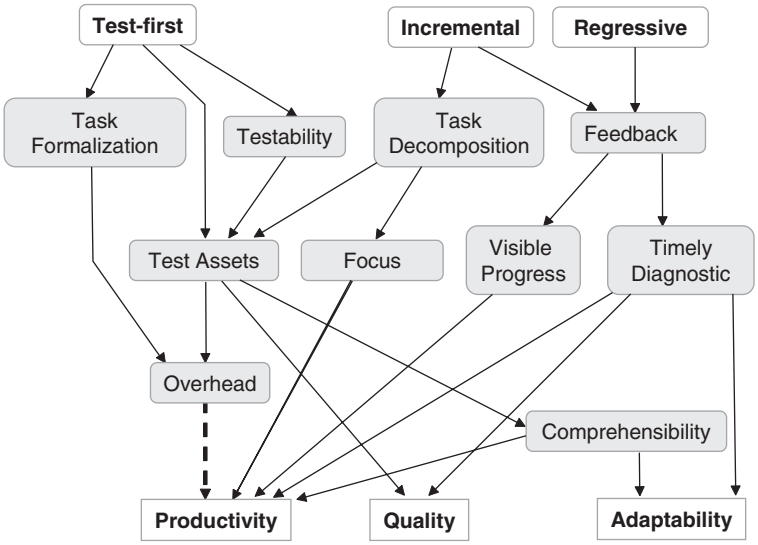

Fig. 3 A theory of TDD.

time to ensure correct program behavior. In incremental development, these tests can be executed (hence the system is regressed) after each bit of implementation activity. Frequent execution of tests provides the programmer with concrete feedback in two ways. First, the programmer knows whether the last task has been completed successfully as specified by the associated tests. The programmer then has a means of objectively gauging progress, a psychological factor that impacts productivity. Second, the programmer knows whether the last burst of programming activity has broken any previously implemented behavior. With timely diagnostic, and the resulting fault localization ability that concrete feedback provides, faults have a lesser chance of propagating and escalating. Early detection and fixing of faults in turn affect both downstream productivity and quality positively. Simultaneously, testable code, the presence of test assets, and the timely diagnostic these assets afford increases software's adaptability: the code can be maintained more easily, and accommodating future changes becomes more and more feasible. Test assets also increase the code's comprehensibility, helping programmers understand its usage by way of concrete examples. Tests double as low-level, executable documentation that improves adaptability.

On the downside, the mechanics of test creation result in extra effort. Down the road, test assets also require maintenance and must be managed, amplifying these efforts. The resulting overhead affects productivity negatively, as indicated by the dashed arrow in Fig. 3. TDD works by trading off this up-front productivity penalty against downstream productivity and quality gains.

PERCEPTIONS AND MISCONCEPTIONS

TDD is a controversial practice. This is partly because it is misunderstood. The main misunderstanding stems from the name: the adjective "test-driven" conjures in many the perception that TDD has more to do with testing and quality assurance than it has to do with development and programming. An extreme form of this misconception equates TDD with plain unit testing. Testing is a means for TDD, but it's not the purpose. Even though TDD leverages unit-testing frameworks, it is simultaneously different and more than unit testing.

TDD is controversial because it also blurs the sometimes revered separation between ordinarily distinct activities. TDD indeed does incorporate and blends activities, or some aspects thereof, traditionally associated with requirements, design, testing, implementation, and documentation.

Even when its main purpose of building software is unambiguously understood, some critiques of TDD question both the need for and the capacity of programmers to write tests. After all, why should programmers get mixed up with testing, a responsibility that belongs with a separate group? And don't testing and programming require different skills? TDD has significant quality side effects and can be seen as an in-process quality assurance aid. However, programmer tests written during TDD are not indented as a complete substitute for other types of independent testing or quality assurance where such activities are appropriate. Most programmers do possess the skills necessary to apply TDD, whereas they may not possess the skills to perform exploratory testing, performance testing, stress testing, formal verification, or system testing.

Similar arguments apply to the relationship between TDD and traditional design and modeling that precede implementation. In reality, TDD is both orthogonal and complementary, rather than a complete substitute to design and modeling.

The most severe criticism of TDD is based on its perceived negative impact on productivity. As reported

in the "Dynamics" section, a number of studies have observed a productivity penalty ranging from mild to significant with variants of TDD, often when compared to a control in which in-process testing is effectively optional. However, the jury is still out regarding TDD's impact on long-term productivity that accounts for downstream rework costs. Many of its expert practitioners characterize TDD mainly as a productivity technique rather than a quality technique.

Other street criticisms of TDD center on cognitive factors. Many argue that TDD is too difficult to learn and apply, that it requires a certain propensity for a particular and counterintuitive way of reasoning and problem solving. In studies of TDD, subjects often express similar sentiments when first exposed to TDD. Some dislike it intensely and abandon the practice altogether. In fact, TDD does require proficiency, discipline, and a change of attitude. Also common are claims that TDD is for junior or disorganized programmers. In reality, motivational and cultural factors are probably more influential than skill level or reasoning style.

AN EXAMPLE

To apply TDD, we need a unit-testing framework designed to work with the programming language being used. A good unit-testing framework is the bare minimum. The sidebar "The xUnit Family of Testing Frameworks" describes the general organization of a family of popular testing framework available for different programming languages. All of these frameworks operate based on the same principles. For Java, the de facto unit-testing framework is JUnit. A modern integrated development environment that supports incremental compilation and refactoring also helps a great deal. We assume an incremental compiler is available. For serious use in a team project, of course, version control and build support are also indispensable, although we won't need them in the following example. The example is developed with Java SE 5 and JUnit 4.4. Some steps have been abbreviated or combined, and several have been omitted to save space. Import declarations are also omitted. Compiler errors are underlined.

Sidebar: The xUnit Family of Testing Frameworks

JUnit is the first of a family of unit testing frameworks referred to as xUnit. In an xUnit framework, unit tests are represented as *test methods*. Each test method exercises a bit of production code functionality and verifies the effects using one or more *assertions*. An assertion stipulates that the actual effect of exercising a piece of production code matches the expected effect. If an assertion of a test method fails, the test method itself fails.

In general, a test method comprises four sections. In the setup section, the system under test is brought to the desired state in preparation for exercising a piece of its functionality. Then the method calls a series of operations of the system under test to exercise the functionality being tested. One or more assertions, written as assertion statements follow to check the effects. Finally, a cleanup may be performed to reverse any side effects or reset the tested system's state.

Test methods are organized into *test cases*. A test case is a special class whose instances are executed by executing all the test methods included in that class once. Test methods within a test case should be independent to avoid side effects so that the sequence in which the test methods are executed does not matter. Some programming languages and earlier versions of xUnit frameworks distinguish test methods of a test case from the ordinary methods of the test case using special naming conventions (e.g., the method name starts with "test"), while other languages and modern versions of xUnit frameworks use metadata attributes, or annotations (as in "@Test" annotation in JUnit 4). Ordinary methods can encapsulate utility functions and setup and cleanup code shared by several test methods.

These extra bits and pieces and other special methods, such as a setup code that needs to be executed for each test method of a test case, collectively make up the test case's *fixture*.

Independent test cases can further be grouped into a *test suite* to give the test code additional structure.

The example's purpose is to create a programming interface for computing taxes and discounts for a commerce application. The scenario illustrates the nature and granularity of the TDD steps involved in solving a typical programming task from scratch. With the programming interface, the users should be able to:

- Create a sale record.
- Compute the federal tax (FT) on the sale.
- Compute the provincial tax (PT) on the sale.
- Compute the discount, if any, applicable to a sale.
- Compute a sale's total amount due after any discount, including taxes.
- Set and reset the current FT rate.
- Set and reset the current PT rate for a given province.

To simplify, we use only integer arithmetic for all currency operations, although dealing with currency and taxes clearly requires real arithmetic.

To kick off, we should be able to create a sale record—an object that represents a sale—with an associated currency amount. We should be able to ask for the amount back. We are not worried about taxes yet. We start with a single test named `createSale` that we decide to insert in

a test case called SaleApiTest:

```
public class SaleApiTest{
   private int amount1 = 100;
   @Test
   public void createSale(){
         assertEquals(amount1, new
          Sale(amount1).amount());
   }
}
```

The test encapsulates the following design decisions:

- There exists a class called Sale.
- To create an instance of Sale, use a constructor that accepts a single parameter. The parameter represents the sale's underlying amount.
- Use the amount() method to ask for the sale's amount back.

Here we express, by an example, the invariant that when we ask a Sale object its amount, it should return the value of the parameter with which it was created. All of this is done using JUnit's assertEquals statement. The first parameter of assertEquals represents the expected value of a computation, and the second parameter represents the actual value that the program code called by the test generates. (JUnit, like many other unit-testing frameworks, provides several other types of *assertions* that can be used in test cases.)

This fragment does not compile for the Sale class, which does not yet exist. Think of the compilation error as a type of test failure that prompts the next action. In this case, the failure states: "the test expects a class Sale to exist, but it does not." Let's quickly fix this problem by stubbing the class out.

```
public class Sale {
   public Sale (int saleAmount){
   }
   public int amount(){
         return 0;
   }
}
```

Now the fragment compiles (and thus part of the test passes). Even if we don't have the right behavior in place, the quick fix allows us to run the test and see it fail.

```
Failure in createSale: expected:<100> but
   was:<0>
```

It's time to add the correct behavior, which calls for a private field.

```
public class Sale{
   private int amount;
   public Sale (int saleAmount){
         amount = saleAmount;
   }
   public int amount(){
         return amount;
   }
}
```

Running the test, we see that it passes. Our first task, and with it the associated TDD episode, is thus complete.

The next test guides us in a similar fashion to add behavior for specifying an FT rate and computing the FT on a sale amount.

```
private int pctRate1 = 6;
private int amount2 = 200;
@Test
public void computeFt(){
       assertEquals(12, new Sale(amount2,
        pctRate1).ft());
}
```

The test states that the FT on a sale with an amount 200 and a FT rate of 6% should be 12. The fragment fails to compile as it should. The failure prompts us to add an extra constructor and a new method stub to Sale, with a dummy return value, allowing the code to compile and the old test createSale to continue to pass, but causing the new test computeFt to fail. To fix the failure, we add the correct behavior to the Sale class.

```
private int ftRate;
public Sale (int saleAmount, int
   applicableFtRate){
      amount = saleAmount;
      ftRate = applicableFtRate;
}
public int ft(){
       return ftRate* amount()/100;
}
```

Now both tests pass. The old constructor should be removed. To be able to do this, first we fix the test associated with the obsolete constructor:

```
@Test
public void createSale(){
     assertEquals(amount1, new
       Sale(amount1, pctRate1).amount());
}
```

Then we delete the obsolete constructor and run both tests to make sure no dangling references exist. Back in the green, we can proceed.

The next task consists in setting the FT rate independently of a sale record and letting all newly created sale records to use that set rate. First, let's take care of setting the FT rate:

```
@Test
public void setAndGetFtRate(){
      FtRate.set(pctRate1);
      assertEquals(pctRate1, new
        FtRate().get());
}
```

Compilation errors lead us to stub the new `FtRate` class out, with a static `set` and a public `get` method. The code then compiles, but the new test fails. The failure signals that the test does not pass trivially. `FtRate's` behavior is straightforward, and we omit it: we implement just enough behavior to make the last test pass. Now we can refactor the old tests to take advantage of the new behavior. We no longer wish to specify the FT rate explicitly when recording a sale. Instead we first set the FT rate independently, and then pass an instance of the `FtRate` class to a new sale object.

```
@Test
public void createSale(){
      FtRate.set(pctRate1);
      assertEquals(amount1,
            new Sale(amount1, new
                  FtRate()).amount());
}
@Test
public void computeFt(){
      FtRate.set(pctRate1);
      assertEquals(12,
         new Sale(amount2, new
               FtRate()).ft());
}
```

The compiler errors alert us to refactor the constructor of `Sale` to accept an `FtRate` object instead of a primitive.

```
public Sale (int saleAmount, FtRate
  applicableFtRate){
      amount = saleAmount;
      ftRate = applicableFtRate.get();
}
```

Finally the new test passes, and nothing else is broken. At this point, noticing the duplication in the test case, we realize that it is possible to set the FT rate once and for all tests by moving the associated statement to a `setUp` method executed *before* each test. After this cleanup, `SaleApiTest` looks like this:

```
public class SaleApiTest{
      @Before
      public void setUp(){
            FtRate.set(pctRate1);
      }
      @Test
      public void createSale(){
            assertEquals(amount1,
                  new Sale(amount1, new
                  FtRate()).amount());
      }
      ...
}
```

Running the modified tests ensures that the tests still work. The next test expresses the requirement that when the FT rate is reset, a previously created `Sale` object retain the FT rate with which it was created.

```
private int pctRate2 = 10;
@Test
public void saleRetainsFtRate(){
      Sale sale1 = new Sale(amount1, new
        FtRate());
      FtRate.set(pctRate2);
      assertEquals(6, sale1.ft());
}
```

The test passes: we don't need to touch the production code.

Using equally tiny steps, we continue to add behavior that allows the user to create a sale record with a PT rate, calculate the PT amount on a sale, calculate the total amount due including all taxes, specify an optional discount rate for a sale record, compute the discount amount when it exists, and adjust the total amount in the presence of a discount. As we implement these features, both the production and the test code are constantly being refactored to remove any introduced duplication and improve clarity. Suppose we end up with a `Rate` superclass that specifies how a rate specified as a percentage is applied to a base amount, whether it is a tax rate or a discount rate. Specialized classes `FtRate`, `PtRate`, and `DiscountRate` inherit from `Rate`. These specialized classes allow the `Sale` class to acquire new constructors representing the possible ways with which a new sale record is created, with the PT rate and discount rate being optional. In the end, the fixture of the test case looks like this:

```
private int pctRate3 = 7;
private int pctRate4 = 20;
@Before
public void setUp(){
      FtRate.set(pctRate1);
      sale2WithDiscount = new Sale
        (amount2, new FtRate(),
```

```
            new PtRate(pctRate3), new
            DiscountRate(pctRate4));
    sale1NoPt = new Sale(amount1, new
        FtRate());
    sale2WithPt = new Sale(amount2,
        new FtRate(), new PtRate
          (pctRate3));
}
```

In the process, we might be required to handle different error conditions, for example, when a sale's total amount due is requested, but the PT rate is unknown (because the sale's originating province is not yet specified). Here is how one might handle such a condition by leading with a test:

```
@Test (expected = PtException.class)
public void canNotComputePt() throws
  PtException{
      sale1NoPt.pt();
}
```

The test states that asking for the PT amount from a sale object with an unspecified PT rate should throw an exception that alerts user of the situation. It prompts us to create the class `PtException`:

```
public class PtException extends
  Throwable{
}
```

Then the new test fails as expected.

```
Failure in canNotComputePt: Unexpected
exception, expected<PtException> but
was <jNullPointerException>
```

Now we can handle the error condition explicitly in the affected methods of `Sale`:

```
public int pt() throws PtException{
      if (ptRate == null){
            throw new PtException();
      } else
      return ptRate.applyTo(amount() -
        discount() + ft());
}
public int totalDue() throws PtException{
      return amount() - discount() +
        ft() + pt();
}
```

This is not quite enough though. Other tests that call `pt()` or `totalDue()` also need to be changed by either propagating a `PtException` or handling it. We opt for the former tactic as in:

```
@Test
public void computePt() throws
  PtException{
      assertEquals(14,
sale2WithPt.pt());
}
```

The change makes all tests to pass, moving us back into the green.

To conclude the example, we show how PT rates can be handled more naturally. Since the PT rate depends on the province and there are only a handful of provinces, we can specify a PT rate in terms of the province, like this:

```
@Test
public void setPtRateByProvince() throws
 PtException{
      Sale saleWithPt = sale1NoPt.
        setPtRate(new PtRate(Province.
        ONTARIO));
      assertEquals(8, saleWithPt.pt());
}
```

The test calls for an enumeration type, so we create it.

```
public enum Province{
      ONTARIO, QUEBEC, ALBERTA
}
```

The stipulated new constructor of `PtRate` needs a stub. After making the new test compile and fail, we fake it for a quick fix:

```
public PtRate(Province province){
      super(8);
}
```

Then we refactor `PtRate` to add the correct behavior.

```
private static HashMap<Province,
  Integer> rates
      =new HashMap<Province, Integer>();
static {
      rates.put(Province.ONTARIO, 8);
}
public PtRate(Province province){
      super(rates.get(province));

}
```

The new test passes for real, along with all the other tests. The last step introduces the ability to set or override the PT rate for a given province, like this:

```
@Test
public void setPtRateForProvince() throws
  PtException{
```

```
        PtRate.setRate(Province.ALBERTA,
          pctRate3);
        Sale saleWithPt = sale1NoPt.
           setPtRate(new PtRate(Province.
           ALBERTA));
        assertEquals(pctRate3,
          saleWithPt.pt());
}
```

Adding the static method `setRate` to the class `PtRate` provides the desired behavior while keeping all tests in the green.

```
public static void setRate(Province
  province, int rateValue){
       rates.put(province, rateValue);
}
```

The above sequence illustrates the train of thought underlying TDD. Although not illustrated in the example, modern integrated development environments (IDEs) provide built-in or third-party refactoring support that ease burden of frequent updates to the production and test code.

DYNAMICS

> *We are what we repeatedly do. Excellence, then, is not an act, but a habit.*
>
> *—Aristotle*

TDD proceeds in short feedback cycles and results in a fair amount of test code. But how short or long can these cycles, or *episodes*, get? Roughly how much test code is a "fair amount of test code"? And how much effort is spent writing test code relative to writing production code? These questions are commonly asked by those who are new to TDD.

The specific patterns regarding episode length and test-code volume vary during the course of development, depending on the stage of development and the underlying tasks' nature. Still, general patterns underlie typical TDD scenarios.

We define an *episode* as the sum of activities that take place between successive stable states of a piece of code under development. In a stable state, 100% of the programmer tests that pertain to the code pass. An episode is thus temporally delimited by "green" states. *Episode length* is the net duration of an episode, measured as the elapsed time between the episode's beginning and end. The developer may take breaks and perform tasks not necessarily directly related to the task at hand (e.g., converse with colleagues, check e-mail, browse the Internet), so the measurement is adjusted by deducting the time during which the development environment has been inactive for at least a predetermined duration. Inactivity refers to absence of changes in the state of the development environment (no navigation, changes of focus, edits, compilations, saves, test runs, or builds). In the episodes we illustrate, the inactivity threshold was set to two minutes: hence if the elapsed time is 10 min, but the developer has not manipulated the development environment for 4.5 min, the recorded measurement equals 5.5 min.

The data associated with the patterns illustrated in the following subsections have been collected using an instrumentation tool that was developed at National Research Council Canada (NRC).[11] The data were collected from one of the authors working on a small Java programming task and an additional developer who was working on a subsequent release of the tool. The second developer was a recently graduated computer engineering student who had a good understanding of TDD, but limited experience with it. We consider these patterns to be representative of TDD's typical application by developers with at least a fair understanding of the technique and moderate programming experience.

Episode Length

TDD experts advise programmers to strive to keep the green-to-green cycles, the episode length, as short as it's practicable to increase visibility, feedback frequency, and confidence in progress. Fig. 4 shows the distribution of episode length for a development scenario not unlike the one presented in the "An Example" section. In the figure, episode length is calculated by deducting the idle time during which the development environment remains inactive above a preset threshold from elapsed time to account for breaks and distractions. The distribution has a very thin tail. The majority of the episodes are very short, lasting less than 5 min. Episodes that last less than 2 min dominate. This might sound too short, but it is the signature of steady and masterful application of TDD. The example in the "An Example" section illustrates how to achieve episodes in this short range. A small improvement to the code, such as changing a variable name, extracting a method, or making an assignment statement more readable, may very well take less than a minute. Similarly, handling obvious behavior, such as performing a straightforward calculation, may also result in very short episodes. Some TDD experts are against leading obvious or trivial behavior by tests and encapsulating them in separate episodes, thus discouraging what they consider to be superfluous tests and artificially short episodes.

The ideal rhythm may take a while to reach in the beginning of a new project. Or it may be interrupted from time to time with changes in task type and focus. The tail part of the distribution, consisting of longer episodes of 10 or more minutes, represents these occasional deviations. For example, long episodes or slippage may occur if the developer is integrating local work with a larger application or experimenting with an unfamiliar application programming interface (API). Despite the resolve to maintain a fast TDD rhythm, breaking a complex task into smaller steps is not always straightforward. In some cases, the developer may not figure out a natural test-first strategy and resort to a

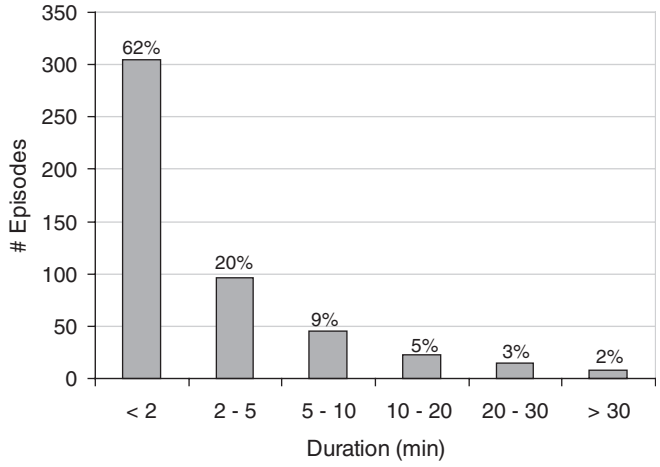

Fig. 4 Episode length distribution in TDD.

traditional test-last strategy, hence digressing from TDD. Or expressing the steps in terms of tests may take more time, and reaching a stable state consequently becomes elusive.

Occasional long episodes and deviations from the ideal TDD rhythm, especially during activities not obviously amenable to TDD or when exploring uncharted territory, are natural and should be expected. While in the long term, it is best to devise TDD strategies that address recurring obstacles, surprises, and new challenges are inherent to software development. We must also note that not all tasks are equal and long episodes might be caused by the complexity of a task at hand while still conforming to the TDD's normal fail–pass-refactor sequencing.

As the level of TDD mastery increases, average episode length tends to get shorter and long episodes and deviations tend to get less and less frequent.

Test-Code Volume

The amount of test code created relative to production code is more predictable and consistent with TDD than episode length. In general, TDD developers expect to write at least as much test code as production code. Some practitioners report test-code-to-production-code ratios of up to two, amounting to a twofold increase in the size of the total code base. Fig. 5 shows how the test-code-to-production-code ratio varied as a function of time during the middle stages of a development scenario. The vertical bars represent episode boundaries, or stable states. Code size was measured in normalized source lines of code.

The ratio hovers around unity within a range of .7–1.2. Observe the drop in the relative volume of tests during the long episodes between minutes 20 and 60. This period corresponds to a temporary departure from TDD, resulting in a loss of rhythm. As the TDD rhythm is regained, the ratio increases gradually to its previous level. Further down the timeline, transient drops in the ratio reappear to coincide with increases in the length of neighboring episodes (space between two vertical bars), and conversely peaks in the ratio coincide with decreases in neighboring episodes' length. These effects are manifested as sparse depressions and dense peaks.

Handling obvious behavior, for example, whether to write tests for getters and setters, influences test-code footprint as it does episode length. The decision is one of weighing the possibility of a future change breaking

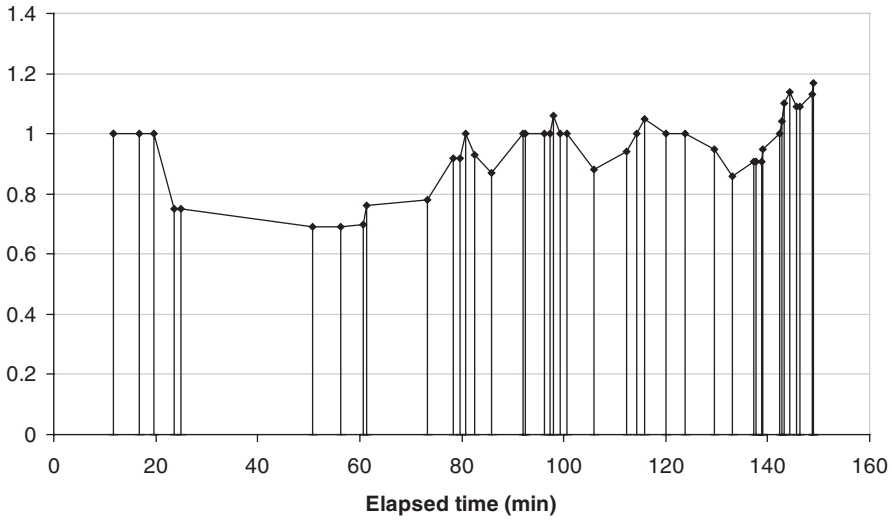

Fig. 5 The ratio of test code to production code as a function of time (size was measured in lines of code).

obvious behavior against having a compact test suite that is easy to manage. Test code is also code: test-code management implies balancing the amount of tests with the principle of traveling light by avoiding spurious tests and duplication.

Test-Code Effort

Writing test code comparable in volume to production code does not necessarily imply expending an equivalent amount of programmer effort for test-code- and production-code-related activities. Some tests involve making important strategic decisions and take more time to figure out and specify. Others guide more tactical tasks involving familiar, recurring problems or express simple input–output relationships, thus taking less time to code.

Fig. 6 compares the effort expended for manipulating test code to effort expended for manipulating production code. The horizontal axis tracks the sequence number of the episodes recorded. For each episode, the dark bar extending upward toward the positive side of the vertical axis represents production-code activity. The light bar extending downward toward the negative side of the vertical axis represents test-code activity. The bars' height indicates the portion of the episode's length attributed to the associated activity.

Production-code activity dominates the long erratic episodes in the beginning as well as the steadier rhythm in the middle section of recorded scenario. The observations for the long episodes in the beginning (until about episode 40) are typical: as the episode length increases, the proportional contribution of test-code activities to the total effort tends to decrease. As a developer moves away from idealized short TDD episodes, the expectation is to spend more time manipulating production code than test code. Test-code effort in the middle, steadier TDD phase of the scenario (from episode 50 to about 80) is very small compared to production-code effort (some of the lower bars in this region are barely noticeable). In the scenario recorded, this stage corresponds to implementation of features that mainly entail algorithmic logic with straightforward, localized computations. It is representative of relatively less design-centric code where underlying design decisions tend to be simple, as in choosing a sensible API signature. Moreover, for such code, tests tend to express input–output relationships of required computations, rather than conditions on complex application states.

The situation changes toward the scenario's end, from about episode 140 onward. Test-code activity dominates this region. The pattern is observed when refactoring the tests, implementing design-rich features, handling cross-cutting behavior, or tackling application logic under composite states. In these circumstances, the crux of the work involves making design decisions that are more strategic and persistent than tactical in nature. In TDD such decisions are captured in the tests.

Does the developer's productivity decrease as test-code effort increases relative to production-code effort? Some developers have this perception when introduced to TDD. The perception stems from seeing tests as pure overhead. If tests constitute overhead, any extra effort expended on manipulating test code could be seen as discretionary. This is not necessarily so. Effort-intensive tests capture important design decisions (even if such decisions tend to be low level). If the developer is spending less time on test-code-related activities, the intellectual work is probably happening elsewhere.

EVIDENCE OF EFFECTIVENESS

So how effective is TDD according to researchers who studied its various aspects? Empirical studies provide somewhat conflicting answers on the two dimensions, productivity and quality, along which the effectiveness of a development technique is commonly evaluated. Table 1 is adapted from.[12] It extends an earlier account with additional, more recent work to summarize a total 23 studies published between 2001 and early 2008. The differences in findings stem from the multiplicity of context factors that influence the outcome variables measured, the variables' definitions, how these variables are measured, and the study design. The most important context factors include the technique against which TDD is evaluated; the

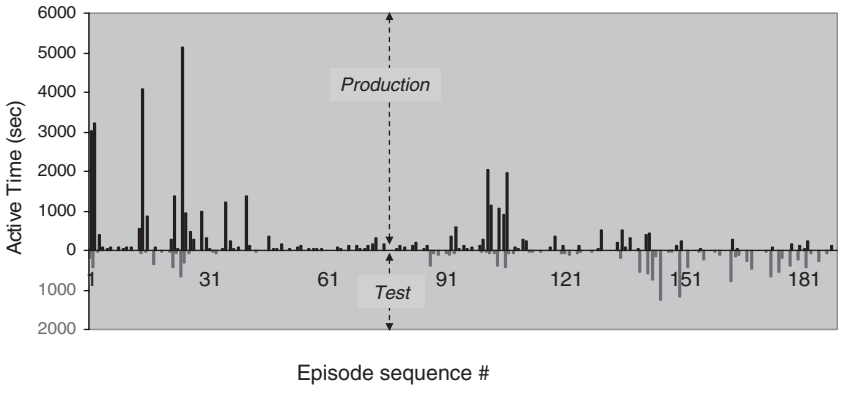

Fig. 6 Effort distribution in TDD: test-code activity vs. production-code activity.

Table 1 Summary of TDD study findings (2001–2008).

Study authors and year	Study type	Duration of observations	Participants	Software built	Productivity effect	Quality effect
1. Jansen and Saiedian, 2008[14]	Experiments and case studies	3–12 months	Five professionals at a Fortune 500 Company & 19 students at University of Kansas	Small Web-based applications (Java)	N/A	Improved test coverage; resulted in less complex code, smaller classes; effect on coupling and cohesion inconclusive[c] (↑)
2. Madeyski and Szala, 2007[15]	Experiment	112 hr	One student at Wroclaw University of Technology	Small Web-based paper submission system (Java/AspectJ)	Improved initially by 87–177%, then when TDD withdrawn, stayed the same (↑)	N/A
3. Siniaalto and Abrahamsson, 2007[16]	Experiment	9 weeks	13 students with industrial experience at VTT Technical Research Center	Small mobile stock market browser application (Java)	N/A	Improved test coverage; cohesion may have decreased; effect on coupling inconclusive[c] (↑)
4. Gupta and Jalote, 2007[17]	Controlled experiment	20–55 hr	22 students at Indian Institute of Technology Kanpur	Toy student registration and ATM applications (Java)	Improved overall productivity (↑)	Inconclusive
5. Sanchez et al., 2007[18]	Case study	5 yr	9–17 professionals at IBM	Medium-size point-of-sale device driver with legacy components (Java)	Increased effort 19% (↓)	40%[a] (↑)
6. Bhat and Nagappan, 2006[19]	Case studies	4–7 months	Five to eight professionals at Microsoft	Small to medium Windows Networking common library, MSN Web services (C/C++/C#)	Increased effort 15–35% (↓)	62–76%[a] (↑)
7. Canfora et al., 2006[20]	Experiment	5 hr	Professionals at Soluziona Software Factory	Toy text analyzer (Java)	Increased effort by 65% (↓)	Inconclusive based on quality of tests
8. Damm and Lundberg, 2006[21]	Case studies	1–1.5 yr	100 professionals at Ericsson	Medium-size components for a mobile network operator application with legacy components (C++/Java)	Total project cost increased by 5–6% (↓)	5–30% decrease in fault slip-through rate; 55% decrease in avoidable fault costs
9. Melis et al., 2006[22]	Simulation	49 days (simulated)	Four simulated subjects based on calibration data from KlondikeTeam & Quinary	Medium-size market information project (Smalltalk)	Increased effort 17% (↓)	36% reduction in residual defect density (↑)

(Continued)

Table 1 Summary of TDD study findings (2001–2008). *(Continued)*

Study authors and year	Study type	Duration of observations	Participants	Software built	Productivity effect	Quality effect
10. Flohr and Schneider, 2006[23]	Experiment	40 hr	18 students at University of Hanover	Small graphical workflow library with legacy components (Java)	Improved productivity by 27% (↑)	Inconclusive
11. Müller, 2006[24]	Artifact analysis	Unknown	Unknown number of students and professionals	Various Small to medium-size open-source and student projects (Java)	N/A	Projects developed using TDD had better assignment controllability (indicating better testability) and lower coupling, but were less cohesive[c] (↑)
12. Mann, 2004[25]	Case study	8 month	Four to seven professionals at PetroSleuth	Medium-size Windows-based oil and gas project management application with elements of statistical modeling and legacy components (C#)	N/A	Reduced external defect ratio by 81%[d]; customer & developers' perception of improved quality (↑)
13. Erdogmus et al., 2005[26]	Experiment	13 hr	24 students at Politecnico di Torino	Toy bowling game application (Java)	Improved normalized productivity by 22% (↑)	No difference
14. Abrahamsson et al., 2005[27]	Case study	30 days	Three students with industrial experience and One professional at VTT	Small mobile application for global markets (Java)	Increased effort by 0–30% with highest increase in early iterations (↓)	No value perceived by developers
15. Melnik and Maurer, 2005[28]	Case studies	4-month projects over 3 yr	240 students at University of Calgary/ SAIT Polytechnic	Various small Web-based systems: surveying, event scheduling, price consolidation, travel mapping (Java)	N/A	73% of respondents perceive TDD improves quality (↑)
16. Madeyski, 2005[29]	Experiment	12 hr	188 students at Wroclaw University of Technology	Toy accounting application (Java)	N/A	–25–45%
17. Geras et al., 2004[30]	Experiment	≈3 hr	14 professionals at various companies	Small simple database-backed business information system (Java)	No effect	Inconclusive based on the failure rates; Improved based on number of tests & frequency of execution
18. Edwards, 2004[31]	Artifact analysis	2–3 weeks	118 students at Virginia Tech	Toy introductory programming assignment (Java)	Increased effort 90% (↓)	45%[b] (↑)

19. George and Williams, 2003[32]	Experiment	≈5 hr	24 professionals at John Deer, Role Model Software, Ericsson	Toy bowling game application (Java)	Increased effort 16% (↓)	18%[b] high test coverage (↑)
20. Pančur et al., 2003[33]	Experiment	4.5 mo	38 students at University of Ljubljana	Four toy programming assignments (Java)	N/A	No difference
21. George, 2002[34]	Experiment	1¼ hr	138 students at North Carolina State University	Toy bowling game application (Java)	Increased effort 16% (↓)	16%[b] (↑)
22. Müller and Hagner, 2002[35]	Experiment	≈10 hr	19 students at University of Karlsruhe	Toy graph library (Java)	No effect	No effect, but better reuse & improved program understanding
23. Ynchausti, 2001[36]	Case study	≈9 hr	Five professionals at Monster Consulting	Small coding exercises	Increased effort 60–100% (↓)	38–267%[a] (↑)

[a] Reduction in the internal defect density.
[b] Increase in percent of functional black-box tests passed (external quality).
[c] Evaluated design quality only.
[d] Cannot be solely attributed to TDD, but to a set of practices.
(↑) improvement.
(↓) deterioration.

selection and level of the developers studied; the type, size, realism, domain, and complexity of the objects (applications, projects, or piece of software developed) studied; and the duration of the development period evaluated.

On the quality front, the results are more compelling, if not resoundingly in agreement. Of the 22 studies that evaluated some aspect of internal or external quality with vs. without TDD, 13 reported improvements of various degrees, 4 were inconclusive, and 4 reported no discernable difference. Only one study reported a quality penalty for TDD. Studies that evaluated defect density report most dramatic improvements. Test coverage, not surprisingly, also appears to improve substantially with TDD. For studies that evaluated TDD against a technique that did not involve any testing or an alternative quality technique, improvements in defect density and test coverage are naturally expected. TDD's overperformance is most meaningful when the alternative technique against which TDD is compared effectively involves and enforces testing or a substitute for testing. However, this was the case only for a small number of studies (notably, studies 1, 3, 13, 16, 20 from Table 1). Most studies conducted with professional developers reported a significant quality advantage for TDD (studies 5, 6, 8, 12, 19, 23).

Studies that evaluated design quality (as opposed to defect density), for example, in terms of object-oriented design metrics, are less conclusive and more controversial. Their results indicate that although program complexity and coupling may decrease with TDD, cohesion may suffer or fail to improve when TDD is viewed as a substitute for up-front design.

While the case for improved external quality is growing, the same cannot be said of productivity, at least not in the short term. Often quality and productivity are traded off against each other, although this need not be the case in every situation. Of the 17 studies that evaluated productivity, only 4 reported an improvement with TDD, while 2 reported no important difference, and the remaining 11 studies reported a penalty ranging from minor to significant. The extent to which participants adhered to TDD and the choice of the alternative technique against which TDD is compared are likely determinants of whether TDD incurs a productivity benefit or penalty. In cases where the alternative technique does not involve testing or a viable substitute, or in which ultimately testing becomes optional, a productivity penalty should be expected. This was the case in the majority of the studies. In addition, significant differences in the way productivity is measured can account for the differences in relative results. An example is the granularity of the output measure chosen—user stories vs. source lines of code delivered. Except for study 8, none of the studies appear to have evaluated net or long-term productivity, that is, productivity including rework effort. Lack of consideration for the downstream cost of poor quality may significantly alter findings: we do not know whether the penalties suffered could have ultimately been compensated by future rework savings. Erdogmus and Williams[13] argue that moderate quality gains can compensate for dramatic productivity shortfalls.

Since it is inherently difficult to pool data from different studies, at this point we cannot draw sweeping, strong conclusions regarding TDD's effectiveness. The big picture points to a potentially material quality advantage with TDD, with an initial productivity penalty. Some studies, such as 1 and 4 from Table 1, argue that the quality advantage is an indirect effect of TDD's enforcement of tests.

CHALLENGES

Social and Cognitive Challenges

Applying TDD requires discipline. Even though TDD may be an efficient way to develop, it proceeds at a steady pace, without giving a sense of racing ahead, sometimes with no exciting great leaps. For some developers, it's more fun to work without a safety net. It can also be difficult to think of an easy way to write the next test. The temptation to proceed without can be overwhelming.

A related obstacle a developer new to TDD faces is the overhead of writing tests: one must write significantly more lines of code with TDD when test code is counted. This overhead may exceed 100% in terms of the total footprint. The pressure to "go faster" by skipping the tests can thus be strong, from management, from peers, and from oneself. Overcoming such temptations and keeping the discipline of leading with tests present a significant barrier to long-term adoption.

In a team environment, TDD may turn out to be an all-or-nothing proposition. Effective application is unlikely in an environment in which TDD is not insistently advocated as standard practice. Only a few keen developers may be writing tests and continuously regressing the system, while the rest of the team is uncooperative. The frustration may eventually prompt those few keen developers to abandon the practice altogether.

Technical Challenges

Test code is also code. Test suite and test execution management become imperative when the number of tests grows. It is not feasible to continuously regress a system if running the tests takes more than a few minutes. Even a few minutes are disruptive enough to spoil a developer's rhythm. In addition, as the application and test suite grows, tests may become brittle and start failing in clusters. The tests need to be minimally redundant and the whole test suite must be organized into decoupled test cases and suites with limited and disjoint scopes so that individual test cases, and suites thereof, can be run with varying frequencies in different stages. Meszaros's test

organization patterns are a good starting point for effective test suite and test execution management.[37]

Other technical challenges arise from applying TDD in the development of front-end and back-end software components. On the front end, the main difficulty stems from reduced ability to capture the software's interactions with the environment, whether users or external hardware, through robust tests. User interfaces, real-time, and embedded software don't lend themselves as naturally to TDD as does a text manipulation library or the business logic of an enterprise application.

On the back end, the main difficulty is tackling the cost and stability of frequent interactions with persistent, distributed, or shared components. Such components include databases, Web services, middleware, operating systems, and other communication and application servers. Setting up such components and precisely controlling their state inside automated tests requires care. In addition, these components often represent expensive or scarce resources, prohibiting frequent interaction with actual instances. Non-deterministic behavior also requires special treatment.

Mock objects[38] are a frequently used technique that allows programmer tests to express behavior that requires interaction with components having complex, slow, or uncontrollable behavior. In tests, these objects emulate in controlled ways the behavior of the real components that they stand for.

Applying TDD to evolve a legacy system with no tests is particularly challenging. The system may need to be refactored carefully to make it sufficiently testable and create an initial safety net to grow. Feathers[39] describes several techniques to achieve this prerequisite state and later leverage it. Meszaros also[37] discusses several useful ways to deal with persistent and legacy components.

Patterns and workarounds that help adapt TDD to the needs of specific contexts are plentiful. *IEEE Software*'s special focus section published in the May/June 2007 issue[12] describes applications of TDD and its variants in the development of relational databases, real-time systems, and graphical user interfaces as well as in handling performance requirements. It also provides a reading list for those who wish to learn and increase their proficiency in TDD.

TOOL SUPPORT

Developers are not without help when applying TDD. Modern IDEs offer incremental compilation and built-in refactoring support. The refactoring capabilities can be further enhanced by plug-ins that facilitate TDD.

Besides basic compilation and refactoring support integrated into the development environment, a plethora of testing frameworks directly support TDD-style development for almost every imaginable environment, platform, and language. Additionally, several frameworks are available to support mocking, a strategy for emulating interactions with complex, resource-intensive, or external components. A selection of TDD tools and frameworks, available as of writing, is listed in Table 2. Frameworks that are listed under the unit-testing category (U) offer low-level, vanilla TDD support, at the module or component level. Frameworks under the acceptance-testing category (A) offer high-level support suitable for applying TDD to implement end-to-end functionality at the requirements, system acceptance, or integration level. Those listed under category B specifically support an advanced variation of TDD known as behavior-driven development discussed in the "Common Variations" section Finally frameworks listed under the category M support mocking.

The number of aids for TDD and TDD-style development are quickly increasing. The majority of offerings are open source or otherwise freely available. If your favorite language or environment is not included in the above lists, check the Web. It is likely that by the time of reading, one will have been created by an enthusiastic colleague or community.

CONCLUSIONS

Test-driven development organically embodies elements of design, coding, and testing in an iterative and incremental style based on a fundamental principle: the developer leads the next increment with a test and avoids writing code except what's necessary to make that test pass. TDD is used in conjunction with continuous refactoring, the practice of improving the code's design. Extrapolated to a higher level, the same style of development (in the form of variants such as acceptance TDD or story TDD) helps with requirements discovery, clarification, and communication when domain experts specify tests before the system's features are implemented.

TDD and its variations and adaptations are used across many kinds of software development projects, from control systems and line-of-business applications to database and rich-client development. A system developed with TDD is naturally testable and incrementally deliverable. The presence of tests guarantees a certain level of quality and allows the system to be changed without the fear of inadvertently breaking it. TDD also helps the low design of the system to emerge rather than be decided in advance. Despite these benefits, TDD is not a silver bullet. It requires discipline and mastery. It is not a complete substitute for traditional assurance or up-front design. However, used properly, TDD can help many developers become more effective.

Table 2 Selection of TDD tools and frameworks (available as of writing).

Framework or framework family	Languages/platforms supported	Categories
Junit	Java	U
TestNG	Java	U
NUnit	.NET	U
TestDriven.NET	.NET	U
CUnit	C	U
xUnit.NET	.NET	U
CppUnit	C++	U
PerlUnit	Perl	U
PyUnit	Python	U
Test::Unit	Ruby	U
PHPUnit	PHP	U
VBUnit	Visual Basic	U
SUnit	SmallTalk	U
Visual Studio Team Test	.NET, C++	U
FIT	Java, C, C++, .NET, Python, Ruby, Objective C, Smalltalk	A
FitNesse	.NET, Java	A
Selenium	Java, .NET, Perl, Python, Ruby	A
Watir/Watij/Watin	Ruby, Java, .NET	A
Exactor	Java	A
TextTest/xUseCase	Python, Java	A
GreenPepper	Java	A
RSpec, ActiveSpec	Ruby	B
BooSpec	Boo, .NET	B
NSpec	.NET	B
JSSpec	JavaScript	B
jMock, EasyMock	Java	M
RSpec, Mocha	Ruby, Rails	M
NMock	.NET	M
MockPP	C++	M
Smock	Smalltalk	M
Test::MockObject	Perl	M
RSpec, Mocha	Ruby, Rails	M
Smock	Smalltalk	M

U: unit-testing framework.
A: acceptance-testing framework.
B: framework supporting behavior-driven development.
M: Mocking framework.

REFERENCES

1. Beck, K. *Test Driven Development—by Example*; Addison-Wesley: Boston, MA, 2003.
2. Astels, D. *Test* NJ *Driven Development: A Practical Guide*; Prentice Hall: Upper Saddle River, NJ, 2003.
3. Beck, K. *Extreme Programming Explained: Embrace Change*, 2nd Ed.; Addison-Wesley: Boston, MA, 2004.
4. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Reading, MA, 1999.
5. Kerievsky, J. *Refactoring to Patterns*; Addison-Wesley: Upper Saddle River, NJ, 2004.
6. Astels, D.; Miller, G.; Novak, M. *A Practical Guide to Extreme Programming*; Prentice Hall: Upper Saddle River, NJ, 2002.
7. Mugridge, R.; Cunningham, W. *Fit for Developing Software: Framework for Integrated Tests*; Prentice Hall: Upper Saddle River, NJ, 2005.
8. Reppert, T. Don't just break software: Make software. Better Softw. **July/August 2004**, 18–23.
9. Humphries, C.; Barker, K. *Foundations of RSpec: Behavior-driven Development with Ruby and Rails*; Apress: Berkeley, CA, 2008.
10. Khaled El Emam, *Finding Success in Small Software Projects*, Agile Project Management Executive Report, Vol. 4, No. 11, Cutter Consortium, Arlington, Massachusetts.
11. Wang, Y.; Erdogmus, H. The role process measurement in test-driven development. In *Extreme Programming and Agile Methods, XP/Agile Universe 2004,* Zannier, C., Erdogmus, H., Lindstrom, L., Eds., Lecture Notes in Computer Science (LNCS), Springer; Calgary, Alberta, 2004; *3134,* 32–42.
12. Jeffries, R.; Melnik, G. TDD: The art of fearless programming. IEEE Softw. **May/June 2007**, 24–30.
13. Erdogmus, H.; Williams, L. The economics of software development by pair programmers. Eng. Econ. **2003**, *48* (4), 283–319.

14. Janzen, D.; Saiedian, H. Does test-driven development really improve software design quality? IEEE Softw. **2008**, *25* (2), 77–84.
15. Madeyski, L.; Szala, L. The impact of test-driven development on software development productivity—An empirical study. In *Software Process Improvement—14th European Conference, EuroSPI 2007*; Potsdam, Germany, 2007; 200–221.
16. Siniaalto, M.; Abrahamsson, P. A comparative case study on the impact of test-driven development on program design and test coverage. In *1st International Symposium on Empirical Software Engineering and Measurement*; Madrid, Spain, 2007.
17. Gupta, A.; Jalote, P. An experimental evaluation of the effectiveness and efficiency of the test-driven development. In *1st International Symposium on Empirical Software Engineering and Measurement*; Madrid, Spain, 2007.
18. Sanchez, J.C.; Williams, L.; Maximilien, E.M. On the sustained use of test-driven development practice at IBM. In *Agile 2007 Conference*, Washington, DC, 2007; 5–14.
19. Bhat, T.; Nagappan, N. Evaluating the efficacy of test-driven development: Industrial case studies. In *5th ACM/IEEE International Symposium on Empirical Software Engineering—ISESE 2006*; Rio de Janeiro, Brazil, 2006.
20. Canfora, G.; Cimitile, A.; Garcia, F.; Piattini, M.; Visaggio, C.A. Evaluating advantages of test driven development: A controlled experiment with professionals. In *5th ACM/IEEE International Symposium on Empirical Software Engineering—ISESE 2006*; Rio de Janeiro, Brazil, 2006.
21. Damm, L.-O.; Lundberg, L. Results from introducing component-level test automation and test-driven development. J. Syst. Softw. **2006**, *79* (7), 1001–1014.
22. Melis, M.; Turnu, I.; Cau, A.; Concas, G. Evaluating the impact of test-first programming and pair programming through software process simulation. Softw. Process: Improv. Pract. **2006**, *11* (4), 345–360.
23. Flohr, T.; Schneider, T. Lessons learned from an XP experiment with students: Test-first needs more teachings. In *7th International Conference on Product-Focused Software Process Improvement, PROFES 2006*; Amsterdam, The Netherlands, 2006; 305–318.
24. Müller, M. The effect of test-driven development on program code. In *7th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2006*; Oulu, Finland, 2006.
25. Mann, C. *An Exploratory Longitudinal Study of Agile Methods in a Small Software Company*. Master's Thesis; Department of Computer Science, University of Calgary, 2004.
26. Erdogmus, H.; Morisio, M.; Torchiano, M. On the effectiveness of the test-first approach to programming. IEEE Tran. Softw. Eng. **2005**, *31* (3), 226–237.
27. Abrahamsson, P.; Hanhineva, A.; Jäälinoja, J. Improving business agility through technical solutions: A case study on test-driven development in mobile software development. In *Business Agility and Information Technology Diffusion, IFIP TC8 WG 8.6 International Working Conference*; Atlanta, Georgia, 2005; 227–243.
28. Melnik, G.; Maurer, F. A cross-program investigation of students' perceptions of agile methods. In *27th International Conference on Software Engineering, ICSE 2005*; St. Louis, MO, 2005; 470–478.
29. Madeyski, L. Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. In *Software Engineering: Evolution and Emerging Technologies*, *Frontiers in Artificial Intelligence and Applications* Zieliński, K., Szmuc, T., Eds.; IOS Press: Amsterdam, The Netherlands, 2005; Vol. 130, 113–123.
30. Geras, A.; Smith, M.; Miller, J. A prototype empirical evaluation of test driven development. In *10th International Symposium on Software Metrics, METRICS 2004*; Chicago, IL, 2004; 405–416.
31. Edwards, S.H. Using software testing to move students from trial-and-error to reflection-in-action. In *35th SIGCSE Technical Symposium on Computer Science Education*; Norfolk, Virginia, 2004; 26–30.
32. George, B.; Williams, L. An initial investigation of test driven development in industry. In *ACM Symposium on Applied Computing*; Melbourne, Florida, 2003; 1135–1139.
33. Pančur, M.; Ciglaric, M.; Trampus, M.; Vidmar, T. Towards empirical evaluation of test-driven development in a university environment. In *Computer as a Tool, IEEE Region 8 Conference, EUROCON 2003*; Ljubljana, Slovenia, 2003; 83–86.
34. George, B. *Analysis and Quantification of Test-Driven Development Approach*. Master's Thesis; Department of Computer Science, North Carolina State University, 2002.
35. Müller, M.M.; Hagner, O. Experiment about test-first programming. In *Empirical Assessment in Software Engineering (EASE)*; Keele, UK, 2002.
36. Ynchausti, R.A. Integrating unit testing into a software development team's process. In *International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2001)*; Sardinia, Italy, 2001; 79–83.
37. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*; Addison Wesley Professional, Upper Saddle River: , 2007.
38. Freeman, S. ; Mackinnon, T. ; Pryce, N. ; Walnes, J. Mock roles, not objects. In *Companion to the Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, 2004; 236—246.
39. Feathers, M. *Working Effectively with Legacy Code*; Prentice Hall: : Upper Saddle River, NJ, 2004.