KEVLIN HENNEY

Testing is one way of making the essentially invisible act of software development more visible. Kevlin Henney presents a practical view of unit testing

Driven to tests

TDD (test-driven development) has increased in popularity both as a practice and as a term. Although there is a strong association with eXtreme Programming (XP), more generally there is a certain set of practices that can be homed in on and applied across various development models. Decoupling TDD from XP, which specifically focuses on the terms *programmer tests* and *test-first programming*, and understanding it as a set of microprocess practices in its own right makes it a more general and useful tool for programmers, regardless of the development macroprocess they are in.

Unit testing and more

The term TDD is often mistakenly used as a synonym for "unit testing". The relationship between the two is that unit testing forms part of TDD, but there is more to TDD than just unit testing. Both critics and advocates often miss that metonymic distinction.

Unit testing is a general notion that can be employed and practised in many ways, whether upfront documentation of test plans associated with upfront documentation of detailed designs in the classic waterfall approach (think Niagara Falls and people in barrels) or more informally by individual programmers seeking confidence in their own work and applying the simple tool of an assertion in a test case that exercises some code they have written.

TDD places unit testing in the context of agile development by motivating it in a different way from traditional testing approaches, which typically borrow from the classic V model of testing – a smokily mirrored view of the waterfall development lifecycle that emphasises testing but divorces it from other activities making up the main flow of development [1].

Effective unit testing can be seen to rest on the foundation of *programmer testing responsibility*, *automated tests* and *example-based tests*. TDD drives that through the design-focused activities of *active test writing*, *sufficient design* and *refactoring*.

Programmer testing responsibility

Unit tests are sometimes called *programmer tests* to differentiate them from system-level tests carried out in a separate testing role or department. It is important to remember that testing a unit is not the same as testing a system. Testing a system involves the software combination of the many code units, whatever their granularity, and the elements of the system external to the code, e.g. target software environment, configuration and broad usage. Units that are under test as units, rather than as integrated parts, are isolated from the external environment, hence "unit" as opposed to "system" or "integration" testing.

Of course, there is a natural tension in any development process between the responsibilities each role can include and the control it can exercise. A development role needs to be broad enough to connect to the range of activities involved in software development, but cohesive enough and appropriately sized so that responsibilities are not overwhelming. For the role of developer, the organisational pattern *Developer Controls Process* defines such a focus, and includes the following brief [2]:

"Responsibilities of developers include understanding requirements, reviewing the solution structure algorithm with peers, building the implementation, and performing unit testing."

However, this does not limit or define the whole range of testing activities. The system-level perspective is explicitly covered in patterns such as *Engage Quality Assurance* and *Application Design Is Bounded by Test Design*. The inclusion of a code-testing responsibility in the programmer's canon also receives strong support from more orthodox testing perspectives [3]:

"I find the projects I work on usually go more smoothly when programmers do some unit and component testing of their own code. Through the ascendance of approaches like Extreme Programming, such a position is becoming less controversial… So, a good practice is to adopt a development process that provides for unit testing, where programmers find bugs in their own software, and for component testing, where programmers test each other's software. (This is sometimes called 'code swapping.') Variations on this approach use concepts like pair programming and peer reviews of automated component test stubs or harnesses."

However, sometimes responsibility can also translate to burden. Just stating that unit testing is a programmer's responsibility does not offer guidance as to how the responsibility can be met in practical terms, which is where the other practices fit in.

Automated tests

Automating test execution relieves much of the tedium that fuels the common perception of testing as a dull and menial task. Unit tests focus on things that can be automated within a localised and code-centric view – therefore, by definition, usability tests and system performance tests are excluded. Tests need to be written in executable form rather than being left in the abstract in a programmer's head or in a document. For unit tests, the most natural exe-

At a glance

- Test-Driven Development can be understood in terms of three core unit-testing practices (examined this time) and three design-focused practices (examined next time).
- Programmer testing responsibility ensures that code-centric tests are the responsibility of those closest to the code.
- Automated tests ensure that unit tests are executed as code-on-code rather than by programmer-on-code.
- Example-based test cases ensure that test cases adopt a sustainable black-box testing approach.

cutable form is code in the same language as the unit being tested. And how much more fun is that than the manual approach? The programmer testing responsibility is realised as a coding activity!

Automated unit tests are quite definite and repeatable in their judgement and there is less of "it kinda looks OK, I think", "I think it's OK, but I don't have the time to check" or "it didn't appear to trip any of the debug assertions when I last ran it by hand, so I guess it's OK".

Automation offers continuous and visible feedback: "All tests passed, none failed". Many people underestimate the value of such feedback. It raises the safety net to a comfortable level and gives a more concrete and local indication of status and progress than many other measures: counting lines of code is simply not a useful measure of progress at any level; end-user accessible functionality is a system- and team-level indicator; test cases offer a personal minute-to-minute, day-to-day progress indicator.

Unit test coverage is inevitably incomplete in practice, so by definition one's ability to trap defects is limited to the quality and quantity of those tests. This is not a criticism of testing, just an observation on practical limits. If one were to claim, "we will catch all defects through unit testing" then that would be a flight of fancy that should be shot down in flames by precisely the observation just made. However, I know of no one advocating any form of unit testing who believes such a proposition. What is guaranteed is that when you have no unit tests, you will catch precisely zero defects through unit testing! Defects represent a form of waste that can brake – and even break – development if allowed to accumulate.

It is worth remembering that perhaps one of the most wasteful code-related activities of all is debugging. Pretty much any opportunity to prevent a situation where debugging becomes a normal and necessary activity should be taken – tests, static analysis, reviews, etc. Unlike debugging, all of these activities can be estimated reasonably in terms of the time they take; debugging lacks this basic property and is also labour intensive. Writing a test involves effort, but with a significantly smaller and consistent schedule footprint. Writing a test is much more linear and far easier to estimate, and running an automated test is not a labour-intensive activity. Debugging is not a sustainably cost-effective development practice.

Example-based test cases

The goal of unit tests is to test functional behaviour, which can be expressed using assertions, rather than operational behaviour, e.g. performance. There are many different unit-testing styles. A distinction often drawn is between *black-box testing* and *white-box testing* (also known as *structural testing* or *glass-box testing*). The premise of white-box testing is that tests are based on the code as written and they explore the paths and values based on the internal structure of the code. For classes this means that the question of examining private data and using private methods often raises its head.

And it is this question that highlights some of the shortcomings of the white-box testing approach. White-box testing must by necessity be carried out after the code is written: before it is written there is no structure on which to perform structural testing. Its emphasis on coverage and this sequencing in development can expose it to the cutting edge of schedule pressure. White-box testing is also coupled to the implementation of a concept: private details are exposed, poked and prodded. Changes to the implementation, even when functional behaviour is preserved, will likely break the tests. Thus, the set of test cases is brittle in the face of change, which will discourage programmers either from making internal changes that ought to be made or from carrying out white-box testing in the first place.

The point of partitioning a system into encapsulated parts is to reduce the coupling of one part of a system on another, allowing more degrees of freedom in implementation behind an interface. White-box testing can strip a system and its developers of that freedom. It is interfaces that define the usage and the path of dependencies within a partitioned system. Consequently, it is interfaces that affect the lines of responsibility and communication in development. This suggests that other development activities need both to respect and to support these boundaries and intentions; working against them can introduce unnecessary friction and distortion [4].

Many previous articles have emphasised the contract metaphor as one of the richer ways of reasoning about an interface [5, 6, 7]. Contracts take a black-box view, focusing on interfaces and constraints on – but not details of – implementation. A black-box test is based on asserting expected effects based on testing given inputs in a given situation.

An example-based style of black-box testing focuses on presenting tests of an implementation through specific examples that use its interface. The contract can be formulated, framed and tested through representative samples [8], as opposed to exhaustively and exhaustingly running through all possible combinations of inputs for their outputs.

Next time...

The combination of *programmer testing responsibility*, *automated tests* and *example-based test cases* offers motivation and a platform for practical unit testing. Test-Driven Development stands on this base, employing the combination of *active testing*, *sufficient design* and *refactoring*, to take the role of testing more solidly into design, and vice versa. ■

References

1. Kevlin Henney, "Learning Curve", *Application Development Advisor*, March 2005
2. James O Coplien and Neil B Harrison, *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall, 2005
3. Rex Black, *Critical Testing Processes*, Addison-Wesley, 2004
4. Melvin E Conway, "How Do Committees Invent", *Datamation*, April 1968, available from www.melconway.com/research/committees.html
5. Kevlin Henney, "Sorted", *Application Development Advisor*, July 2003, available from www.curbralan.com
6. Kevlin Henney, "No Memory for Contracts", *Application Development Advisor*, September 2004, available from www.curbralan.com
7. Kevlin Henney, "First Among Equals", *Application Development Advisor*, November 2004, available from www.curbralan.com
8. Kevlin Henney, "Put to the Test", *Application Development Advisor*, November 2002, available from www.curbralan.com

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com