Test-driven development (TDD)

Test-driven development (**TDD**) is a software **development** process that relies on the repetition of a very short **development** cycle: first the developer writes an (initially failing) automated **test** case that defines a desired improvement or new function, then produces the minimum amount of code to pass that **test**, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered'[1] the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,[3]but more recently has created more general interest in its own right

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

Test-driven development cycle

1. Add a test

In test-driven development, each new feature begins with writing a test. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

2. Run all tests and see if the new one fails

This validates that the test harness is working correctly, that the new test does not mistakenly pass without requiring any new code, and that the required feature does not already exist. This step also tests the test itself, in the negative: it rules out the possibility that the new test always passes, and therefore is worthless. The new test should also fail for the expected reason.

This step increases the developer's confidence that the unit test is testing the correct constraint, and passes only in intended cases.

3. Write some code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test; no further (and therefore untested) functionality should be predicted nor 'allowed for' at any stage.

4. Run All tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor the code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added.

As features are added, method bodies can get longer and other objects larger.

They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognised design patterns. There are specific and general guidelines for refactoring and for creating clean code.

By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging.

Continuous integration helps by providing revertible checkpoints.

When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.The proven librarie features not needed to test in our code.

Development style

There are various aspects to using test-driven development, for example the principles of "**keep it simple, stupid" (KISS)** and "**You aren't gonna need it**" (YAGNI).

By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods. In *Test-Driven Development by Example*, Kent Beck also suggests the principle "**Fake it till you make it**".

To achieve some advanced design concept such as a design pattern, tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Writing the tests first: The tests should be written before the functionality that is to be tested.

This has been claimed to have many benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later. It also ensures that tests for every feature get written. Additionally, writing the tests first leads to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on software quality. When writing feature-first code, there is a tendency by developers and organizations to push the developer on to the next feature, even neglecting testing entirely. The first TDD test might not even compile at first, because the classes and methods it requires may not yet exist.

Nevertheless, that first test functions as the beginning of an executable specification.

Each test case fails initially: This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented.

This has led to the "test-driven development mantra", which is "red/green/refactor," where red means *fail* and green means *pass*. Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring.

Receiving the expected test results at each stage reinforces the developer's mental model of the code, boosts confidence and increases productivity.

Keep the unit small

For TDD, a unit is most commonly defined as a class, or a group of related functions often called a module. Keeping units relatively small is claimed to provide critical benefits, including:

- Reduced debugging effort – When test failures are detected, having smaller units aids in tracking down errors.

- Self-documenting tests – Small test cases are easier to read and to understand

Advanced practices of test-driven development can lead to Acceptance test-driven development (ATDD) and Specification by example where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process.

This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With **ATDD**, the development team now has a specific target to satisfy – the acceptance tests – which keeps them continuously focused on what the customer really wants from each user story.

Best practices

Test structure

Effective layout of a test case ensures

- all required actions are completed,
- improves the readability of the test case,
- smooths the flow of execution.

Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT & UAT.
- Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one

Individual best practices states that one should:

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test focused on only the results necessary to validate its test.

- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits.

Practices to avoid, or "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases.
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
- Testing precise execution behavior timing or performance.
- Testing “multiple features.” An test that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.
- Testing implementation details.
- Saving the test state in target or test objects.
- Spanning the result of test across unit tests.
- Slow running tests.
- Error catching in the test.Errors should be exposed
-

TDD Benefits

Study has found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive. Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.

Programmers using pure TDD on new ("Greenfield") projects reported they only rarely felt the need to invoke a debugger. Used in conjunction with a version control system, when tests fail

unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program.

By focusing on the test cases first, one must imagine how the functionality is used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, the total code implementation time could be shorter based on a model by Müller and Padberg.

Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Madeyski provided an empirical evidence (via a series of laboratory experiments with over 200 developers) regarding the superiority of the TDD practice over the classic Test-Last approach, with respect to the lower coupling between objects (CBO). The mean effect size represents a medium (but close to large) effect on the basis of meta-analysis of the performed experiments which is a substantial finding. It suggests a better modularization (i.e., a more modular design), easier reuse and testing of the developed software products due to the TDD programming practice.

Madeyski also measured the effect of the TDD practice on unit tests using branch coverage (BC) and mutation score indicator (MSI),which are indicators of the thoroughness and the fault detection effectiveness of unit tests, respectively. The effect size of TDD on branch coverage was medium in size and therefore is considered substantive effect.

TDD Limitations

Test-driven development does not perform sufficient testing in situations where full functional tests are required to determine success or failure, due to extensive use of unit tests.Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations.

TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.

Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.

Unit tests created in a test-driven development environment are typically created by the developer who is writing the code being tested. Therefore, the tests may share blind spots with the code: if, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify those parameters. Another example: if the developer misinterprets the requirements for the module he is developing, the code and the unit tests he writes will both be wrong in the same way. Therefore, the tests will pass, giving a false sense of correctness.

A high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.

Tests become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or are themselves prone to failure, are expensive to maintain. This is especially the case with fragile tests.There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs, it may not be detected.

It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase.

Writing and maintaining an excessive number of tests costs time. Also, more-flexible modules (with limited tests) might accept new requirements without the need for changing the tests. For those reasons, testing for only extreme conditions, or a small sample of data, can be easier to adjust than a set of highly detailed tests. However, developers could be warned about overtesting to avoid the excessive work, but it might require advanced skills in sampling or factor analysis.

The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original, or early, tests become increasingly precious as time goes by. The tactic is to fix it early. Also, if a poor architecture, a poor design, or a poor testing strategy leads to a late change that makes dozens of existing tests fail, then it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Test-driven work

Test-driven development has been adopted outside of software development, in both product and service teams, as test-driven work.

Similar to TDD, non-software teams develop quality control checks (usually manual tests rather than automated tests) for each aspect of the work prior to commencing. These QC checks are then used to inform the design and validate the associated outcomes. The six steps of the TDD sequence are applied with minor semantic changes:

1. "Add a check" replaces "Add a test"
2. "Run all checks" replaces "Run all tests"
3. "Do the work" replaces "Write some code"
4. "Run all checks" replaces "Run tests"
5. "Clean up the work" replaces "Refactor code"
6. "Repeat"

TDD and ATDD

Test-Driven Development is related to, but different from Acceptance Test-Driven Development (ATDD).

TDD is primarily a developer's tool to help create well-written unit of code (function, class, or module) that correctly performs a set of operations.

ATDD is a communication tool between the customer, developer, and tester to ensure that the requirements are well-defined. TDD requires test automation. ATDD does not, although automation helps with regression testing. Tests used In TDD can often be derived from ATDD tests, since the code units implement some portion of a requirement. ATDD tests should be readable by the customer. TDD tests do not need to be.

TDD and BDD

BDD (Behavior-driven development) combines practices from TDD and from ATDD.

It includes the practice of writing tests first, but focuses on tests which describe behavior, rather than tests which test a unit of implementation. Tools such as Mspec and Specflow provide a syntax which allow non-programmers to define the behaviors which developers can then translate into automated tests.

Code visibility

Test suite code clearly has to be able to access the code it is testing.

On the other hand, normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

But it is detached from Production code.

In object oriented design this still does not provide access to private data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access private fields and methods.

Alternatively, an inner class can be used to hold the unit tests so they have visibility of the enclosing class's members and attributes.

In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code.

This means the released code is not exactly the same as what was unit tested.

The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private methods and data anyway. Some argue that private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Thus it should be sufficient to test any class through its public interface or through its subclass interface, which some languages call the "protected" interface.

Others say that crucial aspects of functionality may be implemented in private methods and testing them directly offers advantage of smaller and more direct unit tests.

xUnit frameworks[edit]

Developers may use computer-assisted testing frameworks, such as xUnit created in 1998, to create and automatically run the test cases. Xunit frameworks provide assertion-style test validation capabilities and result reporting. These capabilities are critical for automation as they move the burden of execution validation from an independent post-processing activity to one that is included in the test execution. The execution framework provided by these test frameworks allows for the automatic execution of all system test cases or various subsets along with other features.

TAP Results

Testing frameworks may accept unit test output in the language agnostic Test Anything Protocol created in 1987.

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple module may have only ten. The tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code.

Two steps are necessary:

1. Whenever external access is needed in the final design, an interface should be defined that describes the access available. Check the dependency inversion principle for a discussion of the benefits of doing this regardless of TDD.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as “Person object saved” to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected.
3. Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. In a fault mode, a method may return an invalid, incomplete or null response, or may throw an exception. Fake services other than data stores may also be useful in TDD: A fake encryption service may not, in fact, encrypt the data passed; a fake random number service may always return 1.
4. A Test Double is a test-specific capability that substitutes for a system capability, typically a class or function, that the UUT depends on. There are two times at which test doubles can be introduced into a system: link and execution. Link time substitution is when the test double is compiled into the load module, which is executed to validate testing. This approach is typically used when running in an environment other than the target environment that requires doubles for the hardware level code for compilation. The alternative to linker substitution is run-time substitution in which the real functionality is replaced during the execution of a test cases. This substitution is typically done through the reassignment of known function pointers or object replacement.

Test doubles are of a number of different types and varying complexities:

- Dummy – A dummy is the simplest form of a test double. It facilitates linker time substitution by providing a default return value where required.
- Stub – A stub adds simplistic logic to a dummy, providing different outputs.
- Spy – A spy captures and makes available parameter and state information, publishing accessors to test code for private information allowing for more advanced state validation.
- Mock – A mock is specified by an individual test case to validate test-specific behavior, checking parameter values and call sequencing.

- Simulator – A simulator is a comprehensive component providing a higher-fidelity approximation of the target capability (the thing being doubled). A simulator typically requires significant additional development effort.

Integration tests

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the "real" implementations of the interfaces discussed above. These are integration tests and are quite separate from the TDD unit tests. There are fewer of them, and they must be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a "snapshot" of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl,Jenkins,Hudson etc..
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

TDD for complex systems

Exercising TDD on large, challenging systems requires a modular architecture, well-defined components with published interfaces, and disciplined system layering with maximization of platform independence. These proven practices yield increased testability and facilitate the application of build and test automation.

Designing for testability

Complex systems require an architecture that meets a range of requirements. A key subset of these requirements includes support for the complete and effective testing of the system. Effective modular design yields components that share traits essential for effective TDD.

- High Cohesion ensures each unit provides a set of related capabilities and makes the tests of those capabilities easier to maintain.
- Low Coupling allows each unit to be effectively tested in isolation.
- Published Interfaces restrict Component access and serve as contact points for tests, facilitating test creation and ensuring the highest fidelity between test and production unit configuration.

A key technique for building effective modular architecture is Scenario Modeling where a set of sequence charts is constructed, each one focusing on a single system-level execution scenario. The Scenario Model provides an excellent vehicle for creating the strategy of interactions between components in response to a specific stimulus. Each of these Scenario Models serves as a rich set of requirements for the services or functions that a component must provide, and it also dictates the order that these components and services interact together. Scenario modeling can greatly facilitate the construction of TDD tests for a complex system.

Managing tests for large teams

In a larger system the impact of poor component quality is magnified by the complexity of interactions. This magnification makes the benefits of TDD accrue even faster in the context of larger projects. However, the complexity of the total population of tests can become a problem in itself, eroding potential gains. It sounds simple, but a key initial step is to recognize that test code is also important software and should be produced and maintained with the same rigor as the production code.

Creating and managing the architecture of test software within a complex system is just as important as the core product architecture. Test drivers interact with the UUT, test doubles and the unit test framework.

Mock Objects

In object-oriented programming, **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

Reasons for use

In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an actual object has any of the following characteristics, it may be useful to use a mock object in its place:

- the object supplies non-deterministic results (e.g., the current time or the current temperature);

- it has states that are difficult to create or reproduce (e.g., a network error);
- it is slow (e.g., a complete database, which would have to be initialized before the test);
- it does not yet exist or may change behavior;
- it would have to include information and methods exclusively for testing purposes (and not for its actual task).

For example, an alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the test must wait until the alarm time to know whether it has rung the bell correctly.

If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Technical details

Mock objects have the same interface as the real objects they mimic, allowing a client object to remain unaware of whether it is using a real object or a mock object. Many available mock object frameworks allow the programmer to specify which, and in what order, methods will be invoked on a mock object and what parameters will be passed to them, as well as what values will be returned. Thus, the behavior of a complex object such as a network socket can be mimicked by a mock object, allowing the programmer to discover whether the object being tested responds appropriately to the wide variety of states such mock objects may be in.

Mocks, Fakes and Stubs

Classification between mocks, fakes, and stubs is highly inconsistent across literature.Consistent among the literature, though, is that they all represent a production object in a testing environment by exposing the same interface.

Which of the mock, fake, or stub is the simplest is inconsistent, but the simplest always returns pre-arranged responses (as in a method stub). On the other side of the spectrum, the most complex object will fully simulate a production object with complete logic, exceptions, etc. Whether or not any of the mock, fake, or stub trio fits such a definition is, again, inconsistent across the literature.

For example, a mock, fake, or stub method implementation between the two ends of the complexity spectrum might contain assertions to examine the context of each call. For example, a mock object might assert the order in which its methods are called, or assert consistency of data across method calls.

In the book "The Art of Unit Testing mocks are described as a fake object that helps decide whether a test failed or passed by verifying whether an interaction with an object occurred.

Everything else is defined as a stub. In that book, "Fakes" are anything that is not real. Based on their usage, they are either stubs or mocks.

An *external dependency* is an object in your system that your code under test interacts with and over which you have no control. (Common examples are filesystems, threads, memory, time, and so on.)
In programming, you use *stubs* to get around the problem of external dependencies.

A *stub* is a controllable replacement for an existing dependency (or *collaborator*) in the system. By using a stub, you can test your code without dealing with the dependency directly.

The main thing about mocks versus stubs is that mocks are just like stubs, but you assert against the mock object, whereas you do *not* assert against a stub.

Fake: a class that implements an interface but contains fixed data and no logic. Simply returns "good" or "bad" data depending on the implementation.
Mock: a class that implements an interface and allows the ability to dynamically set the values to return/exceptions to throw from particular methods and provides the ability to check if particular methods have been called/not called.
Stub: Like a mock class, except that it doesn't provide the ability to verify that methods have been called/not called.
Mocks and stubs can be hand generated or generated by a mocking framework.

Fake classes are generated by hand.

I use mocks primarily to verify interactions between my class and dependent classes. I use stubs once I have verified the interactions and am testing alternate paths through my code.

I use fake classes primarily to abstract out data dependencies or when mocks/stubs are too tedious to set up each time.

From Martin Fowler about Mock and Stub
Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production
Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
Mocks are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

From xunitpattern:

Mock Object that implements the same interface as an object on which the SUT depends. We can use a Mock Object as an observation point when we need to do Behavior Verification to avoid having an Untested Requirement (see Production Bugs on page X) caused by an inability to observe side-effects of invoking methods on the SUT.

Stub : This implementation is configured to respond to calls from the SUT with the values (or exceptions) that will exercise the Untested Code (see Production Bugs on page X) within the SUT. A key indication for using a Test Stub is having Untested Code caused by the inability to control the indirect inputs of the SUT

Fake: We acquire or build a very lightweight implementation of the same functionality as provided by a component that the SUT depends on and instruct the SUT to use it instead of the real.

Use Mock when it's an object that returns a value that is set to the tested class. and use Stub to mimic an Interface or Abstract class to be tested. In fact, it doesn't really matter what you call it, they are all classes that aren't used in production, and are used as utility classes for testing.

Mock usage in test-driven development

Programmers working with the test-driven development (TDD) method make use of mock objects when writing software. Mock objects meet the interface requirements of, and stand in for, more complex real ones; thus they allow programmers to write and unit-test functionality in one area without actually calling complex underlying or collaborating classes.

Using mock objects allows developers to focus their tests on the behavior of the system under test (SUT) without worrying about its dependencies. For example, testing a complex algorithm based on multiple objects being in particular states can be clearly expressed using mock objects in place of real objects.

Apart from complexity issues and the benefits gained from this separation of concerns, there are practical speed issues involved. Developing a realistic piece of software using TDD may easily involve several hundred unit tests. If many of these induce communication with databases, web services and other out-of-process or networked systems, then the suite of unit tests will quickly become too slow to be run regularly. This in turn leads to bad habits and a reluctance by the developer to maintain the basic tenets of TDD.

When mock objects are replaced by real ones then the end-to-end functionality will need further testing. These will be **integration tests** rather than unit tests.

Mock Limitations

The use of mock objects can closely couple the unit tests to the actual implementation of the code that is being tested.

For example, many mock object frameworks allow the developer to check the order of and number of times that mock object methods were invoked by the real object being tested; subsequent refactoring of the code that is being tested could therefore cause the test to fail even though all mocked object methods still obey the contract of the previous implementation.

This illustrates that unit tests should test a method's external behavior rather than its internal implementation. Over-use of mock objects as part of a suite of unit tests can result in a dramatic increase in the amount of maintenance that needs to be performed on the tests themselves during system evolution as refactoring takes place.

The improper maintenance of such tests during evolution could allow bugs to be missed that would otherwise be caught by unit tests that use instances of real classes. Conversely, simply mocking one method might require far less configuration than setting up an entire real class and therefore reduce maintenance needs.

Mock objects have to accurately model the behavior of the object they are mocking, which can be difficult to achieve if the object being mocked comes from another developer or project or if it has not even been written yet. If the behavior is not modelled correctly then the unit tests may register a pass even though a failure would occur at run time under the same conditions that the unit test is exercising, thus rendering the unit test inaccurate.

More details can be found on

http://xunitpatterns.com/

Martin Fowler on Mocks are not stubs

I first came across the term "mock object" a few years ago in the Extreme Programming(XP) community. Since then I've run into mock objects more and more. Partly this is because many of the leading developers of mock objects have been colleagues of mine at ThoughtWorks at various times. Partly it's because I see them more and more in the XP-influenced testing literature.

But as often as not I see mock objects described poorly. In particular I see them often confused with stubs - a common helper to testing environments. I understand this confusion - I saw them as similar for a while too, but conversations with the mock developers have steadily allowed a little mock understanding to penetrate my tortoiseshell cranium.

This difference is actually two separate differences. On the one hand there is a difference in how test results are verified: a distinction between state verification and behavior verification. On the other hand is a whole different philosophy to the way testing and design play together, which I term here as the classical and mockist styles of Test Driven Development.

Regular Tests

I'll begin by illustrating the two styles with a simple example. (The example is in Java, but the principles make sense with any object-oriented language.) We want to take an order object and fill it from a warehouse object. The order is very simple, with only one product and a quantity. The warehouse holds inventories of different products. When we ask an order to fill itself from a warehouse there are two possible responses. If there's enough product in the warehouse to fill the order, the order becomes filled and the warehouse's amount of the product is reduced by the appropriate amount. If there isn't enough product in the warehouse then the order isn't filled and nothing happens in the warehouse.

These two behaviors imply a couple of tests, these look like pretty conventional JUnit tests.

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

xUnit tests follow a typical four phase sequence: setup, exercise, verify, teardown. In this case the setup phase is done partly in the setUp method (setting up the warehouse) and partly in the test method (setting up the order). The call to order.fill is the exercise phase. This is where the object is prodded to do the thing that we want to test. The assert statements are then the verification stage, checking to see if the exercised method carried out its task correctly. In this case there's no explicit teardown phase, the garbage collector does this for us implicitly.
During setup there are two kinds of object that we are putting together. Order is the class that we are testing, but for Order.fill to work we also need an instance of Warehouse. In this situation Order is the object that we are focused on testing. Testing-oriented people like to use terms like object-under-test or system-under-test to name such a thing. Either term is an ugly mouthful to say, but as it's a widely accepted term I'll hold my nose and use it. Following Meszaros I'll use System Under Test, or rather the abbreviation SUT.
So for this test I need the SUT (Order) and one collaborator (warehouse). I need the warehouse for two reasons: one is to get the tested behavior to work at all (sinceOrder.fill calls warehouse's methods) and secondly I need it for verification (since one of the results of Order.fill is a potential change to the state of the warehouse). As we explore this topic further you'll see there we'll make a lot of the distinction between SUT and collaborators. (In the earlier version of this article I referred to the SUT as the "primary object" and collaborators as "secondary objects")
This style of testing uses **state verification**: which means that we determine whether the exercised method worked correctly by examining the state of the SUT and its collaborators after the method was exercised. As we'll see, mock objects enable a different approach to verification.

Tests with Mock Objects

Now I'll take the same behavior and use mock objects. For this code I'm using the jMock library for defining mocks. jMock is a java mock object library. There are other mock object libraries out there, but this one is an up to date library written by the originators of the technique, so it makes a good one to start with.

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
```

```
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    assertFalse(order.isFilled());
  }
```

Concentrate on testFillingRemovesInventoryIfInStock first, as I've taken a couple of shortcuts with the later test.

To begin with, the setup phase is very different. For a start it's divided into two parts: data and expectations. The data part sets up the objects we are interested in working with, in that sense it's similar to the traditional setup. The difference is in the objects that are created. The SUT is the same - an order. However the collaborator isn't a warehouse object, instead it's a mock warehouse - technically an instance of the class Mock.

The second part of the setup creates expectations on the mock object.The expectations indicate which methods should be called on the mocks when the SUT is exercised.

Once all the expectations are in place I exercise the SUT. After the exercise I then do verification, which has two aspects. I run asserts against the SUT - much as before. However I also verify the mocks - checking that they were called according to their expectations.

The key difference here is how we verify that the order did the right thing in its interaction with the warehouse. With state verification we do this by asserts against the warehouse's state. Mocks use **behavior verification**, where we instead check to see if the order made the correct calls on the warehouse. We do this check by telling the mock what to expect during setup and asking the mock to verify itself during verification. Only the order is checked using asserts, and if the the method doesn't change the state of the order there's no asserts at all.
In the second test I do a couple of different things. Firstly I create the mock differently, using the mock method in MockObjectTestCase rather than the constructor. This is a convenience method in the jMock library that means that I don't need to explicitly call verify later on, any mock created with the convenience method is automatically verified at the end of the test. I could have done this in the first test too, but I wanted to show the verification more explicitly to show how testing with mocks works.
The second different thing in the second test case is that I've relaxed the constraints on the expectation by using withAnyArguments. The reason for this is that the first test checks that the number is passed to the warehouse, so the second test need not repeat that element of the test. If the logic of the order needs to be changed later, then only one test will fail, easing the effort of migrating the tests. As it turns out I could have leftwithAnyArguments out entirely, as that is the default.

Using EasyMock

There are a number of mock object libraries out there. One that I come across a fair bit is EasyMock, both in its java and .NET versions. EasyMock also enable behavior verification, but

has a couple of differences in style with jMock which are worth discussing. Here are the familiar tests again:

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);

    //setup - expectations
    warehouseMock.hasInventory(TALISKER, 50);
    warehouseControl.setReturnValue(true);
    warehouseMock.remove(TALISKER, 50);
    warehouseControl.replay();

    //exercise
    order.fill(warehouseMock);

    //verify
    warehouseControl.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
```

```
    warehouseMock.hasInventory(TALISKER, 51);
    warehouseControl.setReturnValue(false);
    warehouseControl.replay();

    order.fill((Warehouse) warehouseMock);

    assertFalse(order.isFilled());
    warehouseControl.verify();
  }
}
```

EasyMock uses a record/replay metaphor for setting expectations. For each object you wish to mock you create a control and mock object. The mock satisfies the interface of the secondary object, the control gives you additional features. To indicate an expectation you call the method, with the arguments you expect on the mock. You follow this with a call to the control if you want a return value. Once you've finished setting expectations you call replay on the control - at which point the mock finishes the recording and is ready to respond to the primary object. Once done you call verify on the control.

It seems that while people are often fazed at first sight by the record/replay metaphor, they quickly get used to it. It has an advantage over the constraints of jMock in that you are making actual method calls to the mock rather than specifying method names in strings. This means you get to use code-completion in your IDE and any refactoring of method names will automatically update the tests. The downside is that you can't have the looser constraints.

The developers of jMock are working on a new version which will use other techniques to allow you use actual method calls.

The Difference between Mocks and Stubs

When they were first introduced, many people easily confused mock objects with the common testing notion of using stubs. Since then it seems people have better understood the differences (and I hope the earlier version of this paper helped). However to fully understand the way

people use mocks it is important to understand mocks and other kinds of test doubles. ("doubles"? Don't worry if this is a new term to you, wait a few paragraphs and all will be clear.)

When you're doing testing like this, you're focusing on one element of the software at a time - hence the common term unit testing. The problem is that to make a single unit work, you often need other units - hence the need for some kind of warehouse in our example.

In the two styles of testing I've shown above, the first case uses a real warehouse object and the second case uses a mock warehouse, which of course isn't a real warehouse object. Using mocks is one way to not use a real warehouse in the test, but there are other forms of unreal objects used in testing like this.

The vocabulary for talking about this soon gets messy - all sorts of words are used: stub, mock, fake, dummy. For this article I'm going to follow the vocabulary of Gerard Meszaros's book. It's not what everyone uses, but I think it's a good vocabulary and since it's my essay I get to pick which words to use.

Meszaros uses the term **Test Double** as the generic term for any kind of pretend object used in place of a real object for testing purposes. The name comes from the notion of a Stunt Double in movies. (One of his aims was to avoid using any name that was already widely used.) Meszaros then defined four particular kinds of double:

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- **Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Of these kinds of doubles, only mocks insist upon behavior verification. The other doubles can, and usually do, use state verification. Mocks actually do behave like other doubles during the exercise phase, as they need to make the SUT believe it's talking with its real collaborators - but mocks differ in the setup and the verification phases.

To explore test doubles a bit more, we need to extend our example. Many people only use a test double if the real object is awkward to work with. A more common case for a test double would be if we said that we wanted to send an email message if we failed to fill an order. The problem is that we don't want to send actual email messages out to customers during testing. So instead we create a test double of our email system, one that we can control and manipulate.

Here we can begin to see the difference between mocks and stubs. If we were writing a test for this mailing behavior, we might write a simple stub like this.

```
public interface MailService {
  public void send (Message msg);
}
public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

We can then use state verification on the stub like this.

```
class OrderStateTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

Of course this is a very simple test - only that a message has been sent. We've not tested it was sent to the right person, or with the right contents, but it will do to illustrate the point.

Using mocks this test would look quite different.

```
class OrderInteractionTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

In both cases I'm using a test double instead of the real mail service. There is a difference in that the stub uses state verification while the mock uses behavior verification.

In order to use state verification on the stub, I need to make some extra methods on the stub to help with verification. As a result the stub implements MailService but adds extra test methods.

Mock objects always use behavior verification, a stub can go either way. Meszaros refers to stubs that use behavior verification as a Test Spy. The difference is in how exactly the double runs and verifies and I'll leave that for you to explore on your own.

Classical and Mockist Testing

Now I'm at the point where I can explore the second dichotomy: that between classical and mockist TDD. The big issue here is *when* to use a mock (or other double).
The **classical TDD** style is to use real objects if possible and a double if it's awkward to use the real thing. So a classical TDDer would use a real warehouse and a double for the mail service. The kind of double doesn't really matter that much.

A **mockist TDD** practitioner, however, will always use a mock for any object with interesting behavior. In this case for both the warehouse and the mail service.

Although the various mock frameworks were designed with mockist testing in mind, many classicists find them useful for creating doubles.

An important offshoot of the mockist style is that of Behavior Driven Development (BDD). BDD was originally developed by my colleague Dan North as a technique to better help people learn Test Driven Development by focusing on how TDD operates as a design technique. This led to renaming tests as behaviors to better explore where TDD helps with thinking about what an object needs to do. BDD takes a mockist approach, but it expands on this, both with its naming styles, and with its desire to integrate analysis within its technique. I won't go into this more here, as the only relevance to this article is that BDD is another variation on TDD that tends to use mockist testing. I'll leave it to you to follow the link for more information.

You sometimes see "Detroit" style used for "classical" and "London" for "mockist". This alludes to the fact that XP was originally developed with the C3 project in Detroit and the mockist style was developed by early XP adopters in London. I should also mention that many mockist TDDers dislike that term, and indeed any terminology that implies a different style between classical and mockist testing. They don't consider that there is a useful distinction to be made between the two styles.

Choosing Between the Differences

In this article I've explained a pair of differences: state or behavior verification / classic or mockist TDD. What are the arguments to bear in mind when making the choices between them? I'll begin with the state versus behavior verification choice.

The first thing to consider is the context. Are we thinking about an easy collaboration, such as order and warehouse, or an awkward one, such as order and mail service?

If it's an easy collaboration then the choice is simple. If I'm a classic TDDer I don't use a mock, stub or any kind of double. I use a real object and state verification. If I'm a mockist TDDer I use a mock and behavior verification. No decisions at all.

If it's an awkward collaboration, then there's no decision if I'm a mockist - I just use mocks and behavior verification. If I'm a classicist then I do have a choice, but it's not a big deal which one to use. Usually classicists will decide on a case by case basis, using the easiest route for each situation.

So as we see, state versus behavior verification is mostly not a big decision. The real issue is between classic and mockist TDD. As it turns out the characteristics of state and behavior verification do affect that discussion, and that's where I'll focus most of my energy.

But before I do, let me throw in an edge case. Occasionally you do run into things that are really hard to use state verification on, even if they aren't awkward collaborations. A great example of this is a cache. The whole point of a cache is that you can't tell from its state whether the cache hit or missed - this is a case where behavior verification would be the wise choice for even a hard core classical TDDer. I'm sure there are other exceptions in both directions.

As we delve into the classic/mockist choice, there's lots of factors to consider, so I've broken them out into rough groups.

Driving TDD

Mock objects came out of the XP community, and one of the principal features of XP is its emphasis on Test Driven Development - where a system design is evolved through iteration driven by writing tests.

Thus it's no surprise that the mockists particularly talk about the effect of mockist testing on a design. In particular they advocate a style called need-driven development. With this style you begin developing a user story by writing your first test for the outside of your system, making some interface object your SUT. By thinking through the expectations upon the collaborators, you explore the interaction between the SUT and its neighbors - effectively designing the outbound interface of the SUT.

Once you have your first test running, the expectations on the mocks provide a specification for the next step and a starting point for the tests. You turn each expectation into a test on a collaborator and repeat the process working your way into the system one SUT at a time. This style is also referred to as outside-in, which is a very descriptive name for it. It works well with layered systems. You first start by programming the UI using mock layers underneath. Then you write tests for the lower layer, gradually stepping through the system one layer at a time. This is

a very structured and controlled approach, one that many people believe is helpful to guide newcomers to OO and TDD.

Classic TDD doesn't provide quite the same guidance. You can do a similar stepping approach, using stubbed methods instead of mocks. To do this, whenever you need something from a collaborator you just hard-code exactly the response the test requires to make the SUT work. Then once you're green with that you replace the hard coded response with a proper code.

But classic TDD can do other things too. A common style is middle-out. In this style you take a feature and decide what you need in the domain for this feature to work. You get the domain objects to do what you need and once they are working you layer the UI on top. Doing this you might never need to fake anything. A lot of people like this because it focuses attention on the domain model first, which helps keep domain logic from leaking into the UI.

I should stress that both mockists and classicists do this one story at a time. There is a school of thought that builds applications layer by layer, not starting one layer until another is complete. Both classicists and mockists tend to have an agile background and prefer fine-grained iterations. As a result they work feature by feature rather than layer by layer.

Fixture Setup

With classic TDD, you have to create not just the SUT but also all the collaborators that the SUT needs in response to the test. While the example only had a couple of objects, real tests often involve a large amount of secondary objects. Usually these objects are created and torn down with each run of the tests.

Mockist tests, however, only need to create the SUT and mocks for its immediate neighbors. This can avoid some of the involved work in building up complex fixtures (At least in theory. I've come across tales of pretty complex mock setups, but that may be due to not using the tools well.)

In practice, classic testers tend to reuse complex fixtures as much as possible. In the simplest way you do this by putting fixture setup code into the xUnit setup method. More complicated fixtures need to be used by several test classes, so in this case you create special fixture generation classes. I usually call these Object Mothers, based on a naming convention used on an early ThoughtWorks XP project. Using mothers is essential in larger classic testing, but the mothers are additional code that need to be maintained and any changes to the mothers can

have significant ripple effects through the tests. There also may be a performance cost in setting up the fixture - although I haven't heard this to be a serious problem when done properly. Most fixture objects are cheap to create, those that aren't are usually doubled.

As a result I've heard both styles accuse the other of being too much work. Mockists say that creating the fixtures is a lot of effort, but classicists say that this is reused but you have to create mocks with every test.

Test Isolation

If you introduce a bug to a system with mockist testing, it will usually cause only tests whose SUT contains the bug to fail. With the classic approach, however, any tests of client objects can also fail, which leads to failures where the buggy object is used as a collaborator in another object's test. As a result a failure in a highly used object causes a ripple of failing tests all across the system.

Mockist testers consider this to be a major issue; it results in a lot of debugging in order to find the root of the error and fix it. However classicists don't express this as a source of problems. Usually the culprit is relatively easy to spot by looking at which tests fail and the developers can tell that other failures are derived from the root fault. Furthermore if you are testing regularly (as you should) then you know the breakage was caused by what you last edited, so it's not difficult to find the fault.

One factor that may be significant here is the granularity of the tests. Since classic tests exercise multiple real objects, you often find a single test as the primary test for a cluster of objects, rather than just one. If that cluster spans many objects, then it can be much harder to find the real source of a bug. What's happening here is that the tests are too coarse grained.

It's quite likely that mockist tests are less likely to suffer from this problem, because the convention is to mock out all objects beyond the primary, which makes it clear that finer grained tests are needed for collaborators. That said, it's also true that using overly coarse grained tests isn't necessarily a failure of classic testing as a technique, rather a failure to do classic testing properly. A good rule of thumb is to ensure that you separate fine-grained tests for every class. While clusters are sometimes reasonable, they should be limited to only very few objects - no more than half a dozen. In addition, if you find yourself with a debugging problem due to overly

coarse-grained tests, you should debug in a test driven way, creating finer grained tests as you go.

In essence classic xunit tests are not just unit tests, but also mini-integration tests. As a result many people like the fact that client tests may catch errors that the main tests for an object may have missed, particularly probing areas where classes interact. Mockist tests lose that quality. In addition you also run the risk that expectations on mockist tests can be incorrect, resulting in unit tests that run green but mask inherent errors.

It's at this point that I should stress that whichever style of test you use, you must combine it with coarser grained acceptance tests that operate across the system as a whole. I've often come across projects which were late in using acceptance tests and regretted it.

Coupling Tests to Implementations

When you write a mockist test, you are testing the outbound calls of the SUT to ensure it talks properly to its suppliers. A classic test only cares about the final state - not how that state was derived. Mockist tests are thus more coupled to the implementation of a method. Changing the nature of calls to collaborators usually cause a mockist test to break.

This coupling leads to a couple of concerns. The most important one is the effect on Test Driven Development. With mockist testing, writing the test makes you think about the implementation of the behavior - indeed mockist testers see this as an advantage. Classicists, however, think that it's important to only think about what happens from the external interface and to leave all consideration of implementation until after you're done writing the test.

Coupling to the implementation also interferes with refactoring, since implementation changes are much more likely to break tests than with classic testing.

This can be worsened by the nature of mock toolkits. Often mock tools specify very specific method calls and parameter matches, even when they aren't relevant to this particular test. One of the aims of the jMock toolkit is to be more flexible in its specification of the expectations to allow expectations to be looser in areas where it doesn't matter, at the cost of using strings that can make refactoring more tricky.

Design Style

One of the most fascinating aspects of these testing styles to me is how they affect design decisions. As I've talked with both types of tester I've become aware of a few differences between the designs that the styles encourage, but I'm sure I'm barely scratching the surface.

I've already mentioned a difference in tackling layers. Mockist testing supports an outside-in approach while developers who prefer a domain model out style tend to prefer classic testing.

On a smaller level I noticed that mockist testers tend to ease away from methods that return values, in favor of methods that act upon a collecting object. Take the example of the behavior of gathering information from a group of objects to create a report string. A common way to do this is to have the reporting method call string returning methods on the various objects and assemble the resulting string in a temporary variable. A mockist tester would be more likely to pass a string buffer into the various objects and get them to add the various strings to the buffer - treating the string buffer as a collecting parameter.

Mockist testers do talk more about avoiding 'train wrecks' - method chains of style ofgetThis().getThat().getTheOther(). Avoiding method chains is also known as following the Law of Demeter. While method chains are a smell, the opposite problem of middle men objects bloated with forwarding methods is also a smell. (I've always felt I'd be more comfortable with the Law of Demeter if it were called the Suggestion of Demeter.)
One of the hardest things for people to understand in OO design is the "Tell Don't Ask" principle, which encourages you to tell an object to do something rather than rip data out of an object to do it in client code. Mockists say that using mockist testing helps promote this and avoid the getter confetti that pervades too much of code these days. Classicists argue that there are plenty of other ways to do this.

An acknowledged issue with state-based verification is that it can lead to creating query methods only to support verification. It's never comfortable to add methods to the API of an object purely for testing, using behavior verification avoids that problem. The counter-argument to this is that such modifications are usually minor in practice.

Mockists favor role interfaces and assert that using this style of testing encourages more role interfaces, since each collaboration is mocked separately and is thus more likely to be turned into a role interface. So in my example above using a string buffer for generating a report, a

mockist would be more likely to invent a particular role that makes sense in that domain, which *may* be implemented by a string buffer.

It's important to remember that this difference in design style is a key motivator for most mockists. TDD's origins were a desire to get strong automatic regression testing that supported evolutionary design. Along the way its practitioners discovered that writing tests first made a significant improvement to the design process. Mockists have a strong idea of what kind of design is a good design and have developed mock libraries primarily to help people develop this design style.

So should I be a classicist or a mockist?

I find this a difficult question to answer with confidence. Personally I've always been a old fashioned classic TDDer and thus far I don't see any reason to change. I don't see any compelling benefits for mockist TDD, and am concerned about the consequences of coupling tests to implementation.

This has particularly struck me when I've observed a mockist programmer. I really like the fact that while writing the test you focus on the result of the behavior, not how it's done. A mockist is constantly thinking about how the SUT is going to be implemented in order to write the expectations. This feels really unnatural to me.

I also suffer from the disadvantage of not trying mockist TDD on anything more than toys. As I've learned from Test Driven Development itself, it's often hard to judge a technique without trying it seriously. I do know many good developers who are very happy and convinced mockists. So although I'm still a convinced classicist, I'd rather present both arguments as fairly as I can so you can make your own mind up.

So if mockist testing sounds appealing to you, I'd suggest giving it a try. It's particularly worth trying if you are having problems in some of the areas that mockist TDD is intended to improve. I see two main areas here. One is if you're spending a lot of time debugging when tests fail because they aren't breaking cleanly and telling you where the problem is. (You could also improve this by using classic TDD on finer-grained clusters.) The second area is if your objects

don't contain enough behavior, mockist testing may encourage the development team to create more behavior rich objects.

Final Thoughts

As interest in unit testing, the xunit frameworks and Test Driven Development has grown, more and more people are running into mock objects. A lot of the time people learn a bit about the mock object frameworks, without fully understanding the mockist/classical divide that underpins them. Whichever side of that divide you lean on, I think it's useful to understand this difference in views. While you don't have to be a mockist to find the mock frameworks handy, it is useful to understand the thinking that guides many of the design decisions of the software.

The purpose of this article was, and is, to point out these differences and to lay out the trade-offs between them. There is more to mockist thinking than I've had time to go into, particularly its consequences on design style. I hope that in the next few years we'll see more written on this and that will deepen our understanding of the fascinating consequences of writing tests before the code.

The various test doubles usage along with examples:

a) **Dummy** is simple of all. It’s a placeholder required to pass the unit test. Unit in the context (SUT) doesn’t exercise this placeholder. Dummy can be something as simple as passing ‘null’ or a void implementation with exceptions to ensure it’s never leveraged.

```
[TestMethod]
public void PlayerRollDieWithMaxFaceValue()
{
var dummyBoard = new Mock<IBoard>();
var player = new Player(dummyBoard.Object, new Die() ); //null too would have been just fine
player.RollDie();
Assert.AreEqual(6, player.UnitsToMove);
}
```

While the above test would work just fine, it won’t throw any exceptions if RollDie implementation is invoking Board Object. To ensure that Board object isn’t exercised at all you can leverage strict mock. Strict Mock with throw an exception if no expectation is set for member.

```
[TestMethod]
public void PlayerRollDieWithMaxFaceValueStrictTest()
{
var dummyBoard = new Mock<IBoard>(MockBehavior.Strict); //Ensure Board class is never
```

```
invoked
var player = new Player( dummyBoard.Object, new Die() );
player.RollDie();
Assert.AreEqual( 6, player.UnitsToMove );
}
```

b) **Fake** is used to simplify a dependency so that unit test can pass easily. There is very thin line between Fake and Stub which is best described as – “a Test Stub acts as a control point to inject indirect inputs into the SUT the Fake Object does not. It merely provides a way for the interactions to occur in a self-consistent manner. These interactions (between the SUT and the Fake Object) will typically be many and the values passed in as arguments of earlier method calls will often be returned as results of later method calls“. A common place where you would use fake is database access. Below sample shows the same by creating a FakeProductRepository instead of using live database.

```
public interface IProductRepository
{
void AddProduct(IProduct product);
IProduct GetProduct(int productId);
}

public class FakeProductRepository : IProductRepository
{
List<IProduct>
_products = new List<IProduct>();
public void AddProduct(IProduct product)
{
//...
}
public IProduct GetProduct(int productId)
{
//...
}
}

[TestMethod]
public void BillingManagerCalcuateTax()
{
var fakeProductRepository = new FakeProductRepository();
BillingManager billingManager = new BillingManager(fakeProductRepository);
//...
}
```

Fakes can be also be implemented by moq using callbacks.

c) **Stub** is used to provide indirect inputs to the SUT coming from its collaborators / dependencies. These inputs could be in form of objects, exceptions or primitive values. Unlike Fake, stubs are exercised by SUT. Going back to the Die example, we can use a Stub to return a fixed face value. This could simply our tests by taking out the randomness associated with rolling a Die.

```
[TestMethod]
public void PlayerRollDieWithMaxFaceValue()
{
var stubDie = new Mock<IDie>();
stubDie.Setup(d => d.GetFaceValue()).Returns(6).Verifiable();
IDie die = stubDie.Object;
Assert.AreEqual(6, die.GetFaceValue()); //Excercise the return value
}
```

d) **Mock** – Like Indirect Inputs that flow back to SUT from its collaborators, there are also Indirect Outputs. Indirect outputs are tricky to test as they don't return to SUT and are encapsulated by collaborator. Hence it becomes quite difficult to assert on them from a SUT standpoint. This is where behavior verification kicks in. Using behavior verification we can set expectations for SUT to exhibit the right behavior during its interactions with collaborators. Classic example of this is logging. When a SUT invokes logger it might quite difficult for us to assert on the actual log store (file, database, etc.). But what we can do is assert that logger is invoked by SUT. Below is an example that shows a typical mock in action

```
[TestMethod]
public void ModuleThrowExceptionInvokesLogger()
{
var mock = new Mock<ILogger>();
Module module = new Module();
ILogger logger = mock.Object;
module.SetLogger(logger);
module.ThrowException("Catch me if you can");
mock.Verify( m => m.Log( "Catch me if you can" ) );
}
```

e) **Spy** – Spy is a variation of behavior verification. Instead of setting up behavior expectations, Spy records calls made to the collaborator. SUT then can later assert the recordings of Spy. Below is variation of Logger shown for Mock. Focus on this test is to count the number of times Log is invoked on Logger. It's doesn't care about the inputs passed to Log, it just records the Log calls and asserts them. Complex Spy objects can also leverage callback features of moq framework.

```
[TestMethod]
public void ModuleThrowExceptionInvokesLoggerOnlyOnce()
{
var spyLogger = new Mock<ILogger>();
Module module = new Module();
ILogger logger = spyLogger.Object;
module.SetLogger( logger );
module.ThrowException( "Catch me if you can" );
module.ThrowException( "Catch me if you can" );
spyLogger.Verify( m => m.Log( It.IsAny<string>()), Times.Exactly(2) );
}
```

Code refactoring

Code refactoring is the process of restructuring existing computer code – changing the *factoring* – without changing its external behavior. Refactoring improves *nonfunctional* attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source code maintainability and create a more expressive internal architecture or object model to improve extensibility.

Typically, refactoring applies a series of standardised basic *micro-refactorings*, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements. Many development environments provide automated support for performing the mechanical aspects of these basic refactorings. If done extremely well, code refactoring may also resolve hidden, dormant, or undiscovered computer bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, and/or introduce new bugs.

By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.

— *Joshua Kerievsky, Refactoring to Patterns*

Refactoring Overview

Refactoring is usually motivated by noticing a code smell.For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by *refactoring* the source code, or transforming it into a new form that behaves the same as before but that no longer "smells".

For a long routine, one or more smaller subroutines can be extracted; or for duplicate routines, the duplication can be removed and replaced with one shared function. Failure to perform refactoring can result in accumulating technical debt; on the other hand, **refactoring is one of the primary means of repaying technical debt.**

There are two general categories of benefits to the activity of refactoring.

1. Maintainability. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might

be achieved by moving a method to a more appropriate class, or by removing misleading comments.

2. Extensibility. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.[1]

Before refactoring a section of code, a solid set of automatic unit tests is needed. The tests are used to demonstrate that the behavior of the module is correct before the refactoring. If it inadvertently turns out that a test fails, then it's generally best to fix the test first, because otherwise it is hard to distinguish between failures introduced by refactoring and failures that were already there. After the refactoring, the tests are run again to verify the refactoring didn't break the tests. Of course, the tests can never prove that there are no bugs, but the important point is that this process can be cost-effective: good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough.

The process is then an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation. If at any point a test fails, the last small change is undone and repeated in a different way. Through many small steps the program moves from where it was to where you want it to be.

For this very iterative process to be practical, the tests must run very quickly, or the programmer would have to spend a large fraction of his or her time waiting for the tests to finish. Proponents of extreme programming and other agile software development describe this activity as an integral part of the software development cycle.

List of refactoring techniques

Here are some examples of micro-refactorings; some of these may only apply to certain languages or language types. A longer list can be found in Fowler's refactoring book[2] and website.[5] Many development environments provide automated support for these micro-refactorings. For instance, a programmer could click on the name of a variable and then select the "Encapsulate field" refactoring from a context menu. The IDE would then prompt for additional details, typically with sensible defaults and a preview of the code changes. After confirmation by the programmer it would carry out the required changes throughout the code.

- Techniques that allow for more abstraction
  - Encapsulate Field – force code to access the field with getter and setter methods
  - Generalize Type – create more general types to allow for more code sharing
  - Replace type-checking code with State/Strategy[6]
  - Replace conditional with polymorphism
- Techniques for breaking code apart into more logical pieces

    - Componentization breaks code down into reusable semantic units that present clear, well-defined, simple-to-use interfaces.
    - Extract Class moves part of the code from an existing class into a new class.
    - Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
- Techniques for improving names and location of code
    - Move Method or Move Field – move to a more appropriate Class or source file
    - Rename Method or Rename Field – changing the name into a new one that better reveals its purpose
    - Pull Up – in OOP, move to a super class
    - Push Down – in OOP, move to a sub-class

While the term *refactoring* originally referred exclusively to refactoring of software code, in recent years code written in hardware description languages (HDLs) has also been refactored. The term *hardware refactoring* is used as a shorthand term for refactoring of code in hardware description languages. Since HDLs are not considered to be programming languages by most hardware engineers,[8] hardware refactoring is to be considered a separate field from traditional code refactoring.

Sanity check

A **sanity test** or **sanity check** is a basic test to quickly evaluate whether a claim or the result of a calculation can possibly be true. It is a simple check to see if the produced material is rational (that the material's creator was thinking rationally, applying sanity). The point of a sanity test is to rule out certain classes of obviously false results, not to catch every possible error. A rule-of-thumb may be checked to perform the test. The advantage of a sanity test, over performing a complete or rigorous test, is speed.

In arithmetic, for example, when multiplying by 9, using the divisibility rule for 9 to verify that the sum of digits of the result is divisible by 9 is a sanity test—it will not catch *every* multiplication error, however it's a quick and simple method to discover *many* possible errors.

In computer science, a *sanity test* is a very brief run-through of the functionality of a computer program, system, calculation, or other analysis, to assure that part of the system or methodology works roughly as expected. This is often prior to a more exhaustive round of testing.

Continuous test-driven development

Continuous test-driven development (**CTDD**) is a software development practice that extends test-driven development by means of automatic test execution in the background (sometimes called continuous testing).

In CTDD the developer writes a test first but is not forced to execute the tests manually. The tests are run automatically by a continuous testing tool running in the background. This technique can potentially reduce the time waste resulting from manual test execution by eliminating the need for the developer to start the test after each phase of the normal TDD practice: after writing the (initially failing) test, after producing the minimal amount of code for the test to pass and after refactoring the code. It can potentially reduce the time waste resulting from manual test execution.

Behavior-driven development

In software engineering, **behavior-driven development** (**BDD**) is a software development process that emerged from test-driven development (TDD).

Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.[

Although BDD is principally an idea about how software development should be managed by both business interests and technical insight, the practice of BDD does assume the use of specialized software tools to support the development process.

Although these tools are often developed specifically for use in BDD projects, they can be seen as specialized forms of the tooling that supports test-driven development. The tools serve to add automation to the ubiquitous language that is a central theme of BDD.

BDD is largely facilitated through the use of a simple domain specific language (DSL) using natural language constructs (e.g., English-like sentences) that can express the behavior and the expected outcomes. Test scripts have long been a popular application of DSLs with varying degrees of sophistication.

Behavior-driven development is an extension of test-driven development: development that leverages a simple, domain-specific scripting language.

These DSLs convert structured natural language statements into executable tests. The result is a closer relationship to acceptance criteria for a given function and the tests used to validate that functionality. As such it is a natural extension of TDD testing in general.

BDD focuses on:

- Where to start in the process
- What to test and what not to test
- How much to test in one go
- What to call the tests
- How to understand why a test fails

BDD suggests that unit test names be whole sentences starting with a conditional verb ("should" in English for example) and should be written in order of business value. Acceptance tests should be written using the standard agile framework of a User story: "As a [role] I want [feature] so that [benefit]". Acceptance criteria should be written in terms of scenarios and implemented as classes.

BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

At its core, behavior-driven development is a specialized form of Hoare logic applied to test-driven development which focuses on behavioral specification of software units using the Domain Language of the situation.

Test-driven development is a software development methodology which essentially states that for each unit of software, a software developer must:

- define a test set for the unit *first*;
- then implement the unit;
- finally verify that the implementation of the unit makes the tests succeed.

This definition is rather non-specific in that it allows tests in terms of high-level software requirements, low-level technical details or anything in between.

One way of looking at BDD therefore, is that it is a continued development of TDD which makes more specific choices than TDD.

Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit.Borrowing from agile software development the "desired behavior" in this case consists of the requirements set by the business — that is, the desired behavior that has business value for whatever entity commissioned the software unit under construction.

Within BDD practice, this is referred to as BDD being an "outside-in" activity.

Behavioural specifications

Following this fundamental choice, a second choice made by BDD relates to *how* the desired behavior should be specified.

In this area BDD chooses to use a semi-formal format for behavioral specification which is borrowed from user story specifications from the field of object-oriented analysis and design.

BDD specifies that business analysts and developers should collaborate in this area and should specify behavior in terms of user stories, which are each explicitly written down in a dedicated document.[Each user story should, in some way, follow the following structure

Title: The story should have a clear, explicit title.

Narrative

A short, introductory section that specifies

- who (which business or project role) is the driver or primary stakeholder of the story (the actor who derives business benefit from the story)
- what effect the stakeholder wants the story to have
- what business value the stakeholder will derive from this effect

Acceptance criteria or scenarios

a description of each specific case of the narrative. Such a scenario has the following structure:

- It starts by specifying the initial condition that is assumed to be true at the beginning of the scenario. This may consist of a single clause, or several.
- It then states which event triggers the start of the scenario.
- Finally, it states the expected outcome, in one or more clauses.

BDD does not have any formal requirements for exactly **how** these user stories must be written down, but it does insist that each team using BDD come up with a simple,

standardized format for writing down the user stories which includes the elements listed above.

However, in 2007 Dan North suggested a template for a textual format which has found wide following in different BDD software tools.

A very brief example of this format might look like this

Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock

The scenarios are ideally phrased declaratively rather than imperatively — in the business language, with no reference to elements of the UI through which the interactions take place.

This format is referred to as the **Gherkin** language, which has a syntax similar to the above example. The term *Gherkin*, however, is specific to the Cucumber, JBehave and Behat software tools.

BDD Frameworks

- Cucumber (Ruby framework)
- Behat (PHP framework)
- Kahlan (PHP framework)

- Behave (Python framework)
- JBehave (Java framework)
- JBehave Web (Java framework with Selenium integration)
- Jasmine (JavaScript framework)
- Concordion (Java framework)
- Cucumber-js(Javascript framework)
- Squish GUI Tester (BDD GUI Testing Tool for JavaScript, Python, Perl, Ruby and Tcl)
- SpecFlow (.NET framework)
- Spock (Groovy framework)
- Yadda (Gherkin language support for frameworks such as Jasmine (JavaScript framework))

****