Effective TDD
for Complex Embedded Systems

Version 1.09
April 30, 2012
# Table Of Contents

**Executive Summary** .......................................................... 2

**Motivation** ........................................................................... 2

**Introduction to TDD** .......................................................... 4
  - What is the Unit? .............................................................. 4
  - Basic Theory of TDD ........................................................... 4
  - Benefits of Test Driven Development ...................................... 4
  - The TDD micro-cycle .......................................................... 5

**Making it Work in the Real World** ........................................... 5
  - Architecting for Testability .................................................. 6
  - Managing Tests for Large Teams .......................................... 7

**Effective xUnit Testing** ........................................................ 8
  - The xUnit Framework ........................................................ 8
  - Making A Good Test .......................................................... 8
  - The Quick Testing Do’s and Don’ts ........................................ 9

**Separating The System from The Platform** .............................. 10
  - Multi-Platform Testing ........................................................ 10
  - Leveraging Architectural Layers .......................................... 11
  - Dummy/Stub/Spy/Mocking/Simulation (Test Doubles) ............. 11
  - The Payoff ......................................................................... 13

**References** ........................................................................... 14
Executive Summary

Software Development methods allow groups to share techniques that have been found to deliver good results. However, a method that brings success for one team or on a specific class of problem may not work well in a different context until it has been adapted to meet the unique needs of the new context. Test-Driven Development (TDD) is an approach that has been proven to be effective for a broad range of projects – can it work for your team?

If you develop complex embedded systems, certain refinements in TDD techniques can deliver better results for your team more consistently. This paper introduces TDD with specific adaptations to help your team address the unique challenges of building these types of systems.

Motivation

Why apply TDD? One of the most important driving forces for many organizations is to cut time-to-market. Simply put, time is the most precious resource to complex embedded software development efforts. Getting to market faster can mean more market share, happier customers, more time to build additional features, or maybe just the opportunity to go home by dinnertime. To illuminate how TDD brings products to market more quickly, let’s start by highlighting some key principles that pervade the complex embedded software industry:

- **Team productivity trumps individual productivity** - Individual efforts to reduce team-wide hurdles pay off. Conversely, when an individual skips engineering disciplines that help others work more efficiently in order to save themselves a few minutes, the project usually takes longer to complete overall.
- **Integrating a complex embedded system is like running a marathon** - It requires extensive preparation and a healthy respect for the challenge. Without this discipline and perspective, it likely will take a lot longer than expected, can be very painful, and you may not finish at all.
- **Poor quality software is like a zombie** - Sometime in the future it will rise up and torment you, destroying schedules and profits, always at an inconvenient time.

Software quality is important for all kinds of systems, but it is critical for complex embedded systems. Quality is the foundation upon which many other commonly sought benefits are built. Productivity, speed-to-market, reliability, longevity, flexibility, and extensibility all rely on a high quality base of software capability. In addition, some industries require quality certification from regulatory agencies.

How does software quality affect time-to-market? When building complex systems, the Integration Phase is the first big challenge to software quality. Integration happens when individual errors get in the way of the progress made by others, multiplying the cost of repair by the number of people impacted. To understand why
the Integration Phase is so important, let us contrast it with its preceding phase – Individual Component Development:

• It is harder to identify and isolate code issues in a larger body of code, as encountered first in Integration
• Resolving issues during Integration is more expensive than during Development because more people are involved and the time elapsed from defect genesis is greater, thus clouding recollections and hiding true root causes.
• Integration occurs later in the schedule than Development, so there is less time to recover from unanticipated challenges.

We can see it is a good thing to avoid problems during Integration. Typically, for complex embedded systems, even a substantial investment in effort during Development to prevent Integration issues can result in faster time-to-market and improved productivity.

### Signs Integration is Going Well

To really understand how TDD delivers value we need to uncover how TDD improves quality. TDD is about testing, so building lots of tests improves quality - right? While testing is an essential element of an overall quality approach, it cannot improve software quality by itself. Testing only illuminates the current quality state of software. The quality improvements driven by TDD are instead derived from focusing on the things that can adversely affect quality at an early enough stage to affect improved quality. When testing is an intimate part of development from the very start, the tests that fail can now provide the direct and immediate jolt needed to avoid leaving defects in software. Key practices and disciplines in TDD build fundamental improvements in quality culture:

• TDD brings an early test focus which rapidly catches wayward coding and gets the developer immediately back on track.
• Test construction for a feature requires a mastery of requirements for that feature. This avoids a common source of coding errors – misunderstood requirements - and occasionally improves the requirements themselves.
• Steadily building and growing a base of regression tests provides a strong and early indication that software quality is not being eroded in subtle ways. Effective regression test bases empower a team to refactor more often, extensively, and effectively.
Simply by making testing a primary partner activity to coding, this subtle but pervasive shift increases each developer’s focus on quality.

In these ways TDD boosts overall product quality and productivity, and cuts time-to-market.

**Introduction to TDD**

Test-Driven Development is a process of test-first development in which test cases are built prior to each aspect of feature implementation and behavioral design. This practice primarily relies on test cases being added to the system as unit tests and developed to be automatically executed and validated. By developing a set of functional test cases first, developers can be assured that newly implemented features work as expected and that the associated modifications to the code base did not break existing behavior, as manifested by the tests base. In doing this, the software is always in a healthy state, or at least vocal about the ways it might be defective.

**What is the Unit?**

For TDD, a unit is most commonly defined as a class or group of related functions, often called a module. Keep units relatively small provides critical benefits:

- **Reduced Debugging Effort** – Smaller units aid in tracking down issues introduced when test failures are detected.
- **Self-Documenting Tests** – Small test cases have improved readability and facilitate rapid understandability.

**Basic Theory of TDD**

A driving philosophy of TDD is “Red-Green-Refractor”, a short mantra describing the iterative development micro-cycle in which the tests and behavior are implemented. First, newly written test cases try to exercise the unwritten code and consequently fail, verifying test correctness (Red). Then the developer implements the feature in the simplest manner that causes the test to pass (Green). Now with a function system in hand, developers refactor their code for clarity and simplicity. (Refactor).

**Benefits of Test Driven Development**

*Quality and its advantage*

Quality improves through the development of a regression test base rooted in the requirements. Building the regression tests from the beginning allows for the continual verification of the impact of changes, giving developers and product owners a greater confidence that when changes are made, there are no negative side effects. Also the Red-Green-Refactor micro-cycle ensures that test code functions properly in both pass and fail cases, keeping this valuable software asset healthy.
**Improvements to Written Code**

Most TDD projects also build an improved code base in terms of simplicity, understandability, readability. The more complex the problem, the more important it is to ensure each unit is coded in as simple a manner as possible. TDD helps developers avoid over-engineering solutions with its short development micro-cycle, and also helps to break apart the problem into modular units and produce a solution that can be tested as well as implemented.

Ultimately, a simpler design code is easier to read and understand. Additionally, the ability to examine tests corresponding to a unit can aid in understanding the expected behavior of that unit.

**Testability and its advantage**

With a test focus throughout the development lifecycle developers naturally construct testable code. This eliminates typical late-cycle efforts to “add-in” testability, reducing the overall effort required to deliver tested code.

**The TDD micro-cycle**

The TDD micro-cycle provides rapid feedback to the developer – as soon as a code change is completed it is tested. If the code has an error it is immediately caught. The immediacy and locality of the feedback is very valuable. An effective micro-cycle requires quick compilation and execution of test cases. This requirement is central to sustaining the pace of development, and reducing the obtrusiveness of tests. A key technique here is to enable the automatic execution of test cases as a post-build activities, combing building and testing into a single developer step. This reduces manual steps and blends building and testing into one coinciding activity.

**Making it Work in the Real World**

The real world of building complex embedded systems is where tidy theories can fail to deliver positive results due to the inconvenient realities of building complex, high-performance embedded systems. Some of these problems include:

- High feature complexity on a resource-constrained execution platform
- Short time to hit market windows
- Limited development budget and schedule
- Concurrent development of mechanical and electrical aspects of the system
- Large legacy code bases of murky design and dubious quality
- Stakeholders who are skeptical of newfangled software methods claiming to “boost” anything
Teams that are able to successfully surmount these challenges recognize that TDD needs to be part of a cultural shift to a development perspective that embodies a holistic quality focus. Large organizations broaden their gains with TDD through incremental successes facilitated by expert training and mentoring.

*The Cycle of Incremental Success Feeding Culture Change*

**Architecting for Testability**

A complex system requires 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**, ensuring each unit provides a set of related capabilities, making the tests of those capabilities easier to maintain
- **Low Coupling**, so each unit can be effectively tested in isolation
- **Published Interfaces**, restricting Component access and serving as contact points for tests, which facilitates test creation and ensures the highest fidelity between test and production unit configuration.

*Modular, Testable Architecture*

A key technique for building effective modular architecture is Scenario Modeling, where a sequence chart is constructed with a focus on a single system-level
execution scenario. The Scenario Model provides an excellent vehicle for creating the pattern 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/functions that a component must provide, but it also dictates the order that these components and services will interact together. Constructing TDD tests to validate proper component behavior in a Scenario context can greatly facilitate Integration.

Managing Tests for Large Teams

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

During Integration, the body of TDD tests serves as an excellent regression test. Since building the entire body of software for a large system can be time consuming, it is essential to structure the test code so a single build can incorporate the regression base for all system components. To be effective this test base must avoid awkwardly organized tests that require manual execution, elaborate scaffolding or individual components that require unique test framework configurations. Regression tests are more effective when the entire system can be constructed for regression testing in a single executable, minimizing build time for test executables.
Effective xUnit Testing

The xUnit Framework

Many TDD practitioners use xUnit frameworks to provide assertion-style test validation capabilities and result reporting. These capabilities are critical to automation, moving the burden of execution validation from an independent post-processing activity to one that is included in the test execution. This concept of built-in test oracles helps to reduce unit test maintenance burden by requiring maintenance on only one artifact, and eliminating the independent validation of often complex and fragile output. Additionally, these test frameworks tend to provide an execution framework in which to automate the execution of all system test cases or various subsets along with various other features.

Example xUnit Frameworks include:
- Junit (3 & 4)
- cppUnit
- UnitTest++
- Unity (C)

Making A Good Test

Structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smoothes 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 capturing all output such as return values and output parameters. This is usually a very simple section.
• **Validation**: Ensure that the results of the test are correct. Results may include explicit outputs captured during Execution, or state changes in the UUT.

• **Cleanup**: Restore the Unit Under Test (UUT) or the overall test system to the pre-test state. This permits another test to execute immediately after this one.

  **Oracle**

An Oracle is a body of code that inspects explicit outputs captured during Execution, or state changes in the UUT, compares them to expected results, and reports a simple success or outlines a failure and it’s context. Often assertion-style comparison primitives are provided by test frameworks to make automatically report errors via comparison primitives:

```c
ASSERT_EQUAL( expected value, actual value);
```

Simple, robust and repeatable tests are best. Avoid:

• Random Numbers

• Variable values the won’t be consistent between runs, like timestamps and absolute file names

• Superfluous information - only inspect and report what is necessary to validate the test case

• Timing Dependent Activities that precisely measure elapsed execution time, unless you have explicit control over your timing reference

  **Automation**

Large bodies of automatically executed tests – without manual intervention are essential. Unit tests should not include manual set-up, input, or validation steps. If your UUT requires explicit configuration of the test environments, use tools and scripts to automatically establish the right context.

**The Quick Testing Do’s and Don’ts**

There are a number of patterns and practices proven to produce a flexible and durable test base. Industry experience also has highlighted some anti-patterns which result in brittle test cases and increased maintenance and debugging costs.

**Don’ts**

1. Do not have test cases depend on system state manipulated from previously executed test cases.

   • A test suite where test cases are dependent upon each other is brittle and complex. Execution order has to be specifiable and/or constant. Basic refactoring of the initial test cases or structure of the UUT cause a spiral in increasingly pervasive impacts in associated tests.
• Interdependent tests can cause cascading false negatives, as a failure in an early test case breaks a later test case with no actual fault exists in the UUT, increasing defect analysis and debug efforts.

2. Do not test **precise** execution behavior **timing** or performance.

3. A common error for many TDD practitioners is to build **all-knowing oracles**. An oracle that inspects more than is required is more expensive and brittle over time than it needs to be. This very common error is also very dangerous because it causes very subtle but pervasive time sink across a complex project.

**Do’s**

1. Separate **common** set up and teardown logic into test support services utilized by the appropriate test cases.

2. Keep each test **oracle focused** only on the results necessary to validate its test.

3. Design **time-related** tests to allow **tolerance** for execution in a non-real time operating systems. A common practice is allowing a 5-10% margin for late execution, reducing the potential number of false negatives in test execution.

4. 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, be readable and maintainable.

5. Get together with your team and **review** your **tests** and **test practices** to share effective techniques and catch bad habits. (When you do, re-read this section.)

**Separating The System from The Platform**

A common embedded coding perspective is to write code only for the target platform. If you are building code that is delivered on a single platform, why bother testing it on a different platform? While there are a range of benefits that accrue to platform-independent code, we find universally that the most important is it **Saves Time**. By making testing and development quicker, making more test platforms available, and fundamentally facilitating automated testing, the benefits of building well partitioned and portable systems far exceed the costs.

**Multi-Platform Testing**

At the start of many embedded projects, the final hardware is typically unavailable to software developers for various reasons – it may still be in development, or the selection COTS components haven’t been finalized. Even when it finally becomes available, custom developed hardware is buggy in its initial releases. Software
developers cannot wait for reasonably reliable hardware to start testing, and still meet their schedules.

Functional behavior can readily be exercised on typical development environments. Nearly all initial tests – and for many systems the majority of all tests – can be completed in this context. The development platform is nearly always cheaper and far more plentiful than the target environment. Tools like Visual Studio and Eclipse CDT provide debugging capabilities that surpass most embedded environment development platforms, and have the added benefit of debugging directly onto your primary workstation, and often providing far more computing horsepower.

**Leveraging Architectural Layers**

*Hardware Abstraction*

Architecting software for platform independence provides the ability to separate the majority of the system from interfaces to unstable custom hardware base. Explicitly defining a hardware abstraction layer provides an opportunity for simulated/stubbed hardware behavior. This can greatly delay the need for actual hardware, and throughout the project life provides the ability to execute in both the development and target environments.

*Portability Layer*

At both the architectural level and the individual module coding level, building a system to be independent of a single target OS delivers a range of benefits. In addition to general architectural benefits of reduced coupling and greater large-grained flexibility, having a disciplined approach to portability also allows for simpler and more effective optimization efforts, and reduced duplication of code.

From a test perspective, portability offers the simple and substantial benefit of being able to exercise and validate the system on a number of platforms. While not always needed, for some projects the discipline of writing code to use only the OS portability layer can then allow test code to control aspects of the test environment not normally under application control. For example, OS portability layer socket communications can be replaced by test code to simulate delays or errors. Tests can control time services – an element that is often at the heart of most RTOSs. A test can speed up or slow down the passage of time in addition to facilitating date and time specific testing.

An OS portability layer is beneficial to system development and good design, and the reduction of overall testing effort.

*Thread Of Execution Design*

Effective architecture for complex embedded systems generally applies multiple, parallel threads of execution. In another case where best design also results in best test design, industry experience shows the simplest multi-thread approach that work generally works best. Keeping most modules in your system simple, and isolating multi-thread control to a single layer can greatly reduce issues. Minimizing the number of threads and localizing their control facilitates tests and greatly reduces the risk of inter-thread conflicts and subtle dependencies. From a testing perspective, inter-thread issues are generally the most difficult types of defects to repeatable illuminate.

**Dummy/Stub/Spy/Mocking/Simulation (Test Doubles)**
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 in which test doubles can be introduced into a system; linker 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, where doubles for the hardware level code are required 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, typically through the use of reassignment of known function pointers or object replacement.

There are a number of different types of test doubles of varying complexity:

- **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 validate to test-specific behavior, checking parameter values and call sequencing.
- **Simulator** – A comprehensive component providing a higher-fidelity approximation of the target capability (the thing being doubled). A simulator typically requires significant additional development effort.

**Spotlight on Mock Objects**

**What Mocking is**

A Mock Object is specified in the context of an individual test case. It checks parameter values and call sequencing specific to that case. Unlike other Test Doubles, a Mock is tailored to the needs of a single test and therefore provides the detailed capability needed by that test case, but in the simplest manner possible. In this way a mock for one test case does not depend on any other test case.

A Mock Framework provides scripting capabilities, giving the test writer a full set of invocation validation primitives.

**What Mocking isn’t**

Mock Objects are not a simulation tool, as they have no understanding of the subject matter represented by the object being mocked. Mocks only understand function calls, parameters, call counts, and return values.

**What to do With a Mock**

Mock objects can test service invocation parameter values and counts. Mocks can also provide advanced stubbing capabilities for scripting the results of service invocations to drive testing down different logical paths.

Some mock frameworks provide the ability to validate advanced scenarios by specifying the sequence of ordering in which methods of the Mock Object have to be invoked.
Mocking frameworks provide advanced features, such as redirecting function calls to other methods and invoking callbacks via function pointers passed as parameters.

The Payoff

While adopting a Test-Driven Development approach takes significant effort and may subject a development organization to the pain of culture change, the results can far outweigh these costs. Ultimately, the most significant business gains delivered by TDD come from substantially and fundamentally boosting the quality of the software components of your system. Building the capability for your team to consistently build higher-quality components will deliver these key business benefits:

- Boosting overall product quality
- Delivering products to market faster
- Rapidly delivering anticipated product updates

**Boost Overall Product Quality**

With a focus on testing first, developing in small increments, and requiring tests to pass, the TDD approach forces each component to steadily build capability at a consistently higher level of quality. Once individual component development switches to integration, this high level of component quality leads to rapid integration and more effective testing. Finally, the effort to build a broad base of automated tests incrementally throughout the development cycle yields a far more complete and valuable base of tests than a rushed - and often truncated – test effort at the end. Throughout the development lifecycle, the increased focus on testing helps maintain a higher focus on quality in general. This continual investment in quality yields significant dividends at the overall system level.

**Deliver Products to Market Faster**

The TDD cycle is shorter than the traditional development cycle. By focusing on building only what is needed to get the tests to pass and using immediate feedback to reduce regression errors, the development cycle can quickly be reduced to its essential core. And while it is true that the quality improvements gained at the component level will speed development of the component itself, they also slash the
time needed to integrate components into the complete system by a substantial degree.

Typically, integration delays for systems with average component quality will bloat software development and delivery schedules. When applying a TDD approach instead, most component issues are resolved much earlier in the process or eliminated outright, thus shrinking integration times and quickly getting the completed product to market.

![TDD Time to Market Payoff](image)

**TDD Shortens Code and Integration Phases**

**Rapidly Deliver Unanticipated Product Updates**

Current realities driving business in the realm of embedded software include tightening competition, reduced schedules, and tighter market windows. Opportunities don’t wait for convenient times to emerge – they must be seized immediately. TDD boosts a software development organization’s ability to rapidly respond to changing requirements by facilitating shorter development and integration cycles, and supporting the rapid refinement of new requirements. A solid TDD culture with a rich test base is the best preparation to rapidly seize opportunities and beat competitors to market.

**References**


