Backwards Is Forward: Making Better Games with Test-Driven Development

Game Developers Conference 2006

Noel Llopis
High Moon Studios
llopis@convexhull.com

Sean Houghton
High Moon Studios
sean.houghton@gmail.com

The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. Frederick P. Brooks, Jr.

1. Introduction

Part of the appeal of programming is that, as Brooks describes, it allows us to build ornate castles in the air, where our imagination is the only limit. As our target platforms increase in power and memory, our castles become larger and more intricate.

However, it's the same facility to weave code out of thin air that can become our greatest danger. Code has a tendency to grow beyond initial expectations, quickly surpassing our capacity to fully understand it. Our castles in the air can quickly become snowballs that are too large to be moved and shaped to fit our needs.

We have all experienced how development slows down to a crawl towards the end of a project. We have seen first-hand the difficulty of squashing insidious many-headed bugs. We have wrestled with somebody else's code, just to give up or fully re-write it in despair. We have sat in frustration, unable to do any work for several hours while the game build is broken. We have seen countless hours of work go up in smoke as code is thrown away and started from scratch for the next project.

Code can get too complex for its own good. Milestone pressures, a fluctuating game industry, growing teams and budgets, and the breakneck pace of hardware change don't help an already difficult situation.

This is where test-driven development comes in.

2. What is test-driven development?

Let's start by looking at the traditional programming process. Once you decide to implement a specific piece of functionality, you break the problem down mentally into smaller problems, and you start implementing each of them. You write code, and you compile it to know whether things are going well. After a while, you might have written enough code that you can run the game and try to see if the feature works. Maybe you run through a level, checking that feature and making sure nothing else looks obviously broken, or maybe you even step in the debugger to make sure your code is getting called and doing what it's supposed do. Once you're happy with it, you check it into source control and move on to

another task.

Test-driven development (TDD) turns the programming process around: As before, you break down a problem mentally into smaller problems (or at least a single smaller problem), but now you write a unit test for that small feature, see it fail (since you still haven't implemented anything), and then write the code to make that test pass. Then you repeat the process with another, very small piece of functionality that will get you closer to the full feature you're trying to implement. The cycles are very short, perhaps only a minute or two.

Let's have a more detailed look at the TDD cycle:

- Write a single unit test for a very small piece of functionality.
- Run it and see it fail (in C++, it's likely that it won't even compile).
- Write the simplest amount of code that will make the test compile and pass.
- Refactor the code and/or tests. Run tests and see them pass.

A *unit test* is a test that verifies a single, small behavior of the system (also called a developer test). Specifically, in this context, each unit test deals with something almost trivially small, which can probably be done in less than a minute. For example, it can test that a health pack doesn't add more health past the player's maximum, or it can test that a particular effect disables depth writes. We'll see an example of a simple test in the next section.

How does TDD help us deal with the complexity of software, and, more specifically, how does it solve some of the problems listed in the introduction? These are the benefits, listed roughly in our order of importance:

Better code design

What is the first thing you do when writing some code with TDD? You are forced to use that code in a test. That simple fact makes it so all code is created with the user of the code in mind, not the implementation details. The resulting code is much easier to work with as a result, and it directly solves the problems it was intended to fix. Also, because the code was first created by testing it in isolation from the rest of the code, you will end up with a much more modular, simpler design.

Safety net

With TDD, just about every bit of code has some associated tests with it. That means you can be merciless about refactoring your code, and you can still be confident that it will work if all the tests continue to run. Even though we find refactoring to be an extremely important part of the development process, the safety net goes way beyond refactoring. You can confidently apply obscure performance optimizations to squeeze that last bit of performance out of the hardware and know that nothing is broken. Similarly, changing functionality or adding new feature towards the end of the development cycle suddenly becomes a lot less risky and scary.

Instant feedback

The unit tests provide you with instant feedback up to several times per minute. If at any point you thought tests should be passing and they aren't, you know something has gone wrong: not an hour ago, not ten minutes ago, but sometime in the last minute. In the worst case, you can just revert your changes and start over. This is quite subjective, but the constant feedback has a surprising moral-boosting effect.

No problem seems too large as long as you're taking small steps in its direction and getting feedback that you're in the correct track. For some of us, TDD has rekindled the same joy of programming that we discovered in the 8-bit computers we cut our teeth on.

Documentation

What's worse than uncommented code? Code with outdated comments. We all know how easy it is for comments to get out of date, yet there is very little we can do about it. The unit tests created through TDD serve as a very effective form of documentation. You can browse through them to see what kind of use a class is intended to have, you can look up a particular test to see what kind of assumptions a function makes about its parameters, or you can even comment out some code that makes no sense and see what tests break to give an idea of what it does. The best thing about unit tests as documentation: they can never get out of date. We have found that in codebases that are developed with TDD, comments have almost disappeared, being used only to explain why something was done in a particular way, or to document what paper an algorithm we implemented came from.

One thing that should be clear and is important to stress is that TDD is not just writing unit tests. TDD is a development methodology, not a testing one. That's why TDD's benefits deal with better code design and structure, ease of refactoring, etc, and not with correctness. TDD ensures that your code does whatever you wanted it to do, not that it does it correctly.

3. Implementing TDD

Let's put TDD in practice by following the normal practice and writing our first test. Let's assume we already have a game up and running, and our task is to implement health powerups. The very first thing we want to implement is that if the player walks over a health powerup, he receives some amount of health. That's actually even more complicated than what we want for our small testing steps, so let's start with an even simpler test that requires the least amount of effort: if we have a player and a powerup, and the player isn't anywhere near the powerup, his amount of health doesn't change. Simple, right?

Ideally, how would we like to test that? Probably by writing some code like this:

```
World world;
const initialHealth = 60;
Player player(initialHealth);
world.Add(&player, Transform(AxisY, 0, Vector3(10,0,10));
HealthPowerup powerup;
world.Add(&powerup, Transform(AxisY, 0, Vector3(-10,0,20);
world.Update(0.1f);
CHECK_EQUAL(initialHealth, player.GetHealth());
```

And that's exactly what the test will look like, only we'll have to surround it with a macro to take care of all the bookkeeping and to give it a descriptive name. Our full test looks like this:

```
TEST (PlayersHealtDoesNotIncreaseWhileFarFromHealthPowerup)
{
      World world;
      const initialHealth = 60;
      Player player(initialHealth);
      world.Add(&player, Transform(AxisY, 0, Vector3(10,0,10));
      HealthPowerup powerup;
      world.Add(&powerup, Transform(AxisY, 0, Vector3(-10,0,20);
```

```
    world.Update(0.1f);
    CHECK_EQUAL(initialHealth, player.GetHealth());
}
```

The macros `TEST` and `CHECK_EQUAL` are part of a unit testing framework. The framework is intended to make the task for writing and running unit tests as simple as possible. As you can see, it can't get much easier than that. There are a variety of freely-available unit-test frameworks for C++ and other languages. We recommend UnitTest++, which is a lightweight C++ unit-test framework that supports multiple platforms, is easy to adapt and port, and was created after using TDD in games for several years.

Now we compile this code and we get the following output:

```
Running unit tests...
1 tests run
There were no test failures.
Test time: 0 seconds.
```

The `CHECK_EQUAL` macro is the part that performs the actual check, and it takes as parameters the expected value and the actual value. If they are different, it will mark the test as failed, and output a message with as much information as possible. As a bonus, UnitTest++ formats the failed test message so it can be parsed by Visual Studio and it is easy to navigate to the location of the test.

The reason we have macros like `CHECK_EQUAL` is that unit tests need to be fully automated. There should be no need for visual inspection or reading some text. The test program should know whether any tests failed or not without any ambiguity, that way it can be easily integrated into the build process.

4. How to test

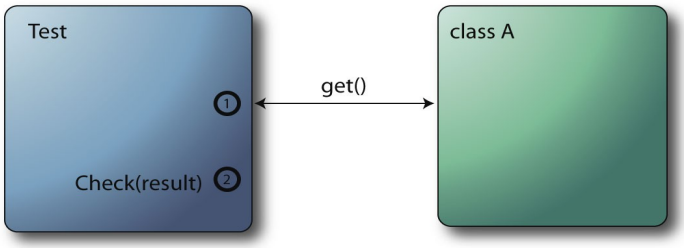
When writing unit tests, there are three main ways of testing your code:

Return value

Make a function call, and check the return value. This is the most direct way of testing, and it works great for functions that do computations and return the computed value. Because this is the easiest and most straightforward way of testing, we should use this approach whenever possible.

For example, a function called `GetNearestEnemy()` would be a perfect candidate to be tested this way. We can easily set up a world with a couple of enemies, call that function, and verify that it returns the enemy entity we expected.

Beware of testing functions that return boolean results indicating if the function failed or succeeded. You really want to test that the function does things correctly, not just that it reports it did them.

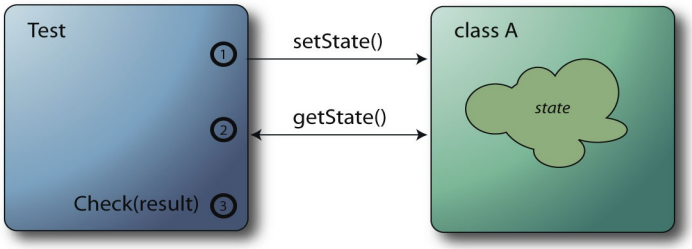


Object state

Make a function call, then check that the state of the object or some part of the system has changed correctly. This tests that state changes directly, so it's also a very straightforward form of testing.

For example, we could send an event of nearby noise to an AI in “idle” state and check that its state changes to “alert” in response.

Sometimes it might force you to expose some state that would otherwise be private, but if it's something that you need to test from the user point of view, then making it public is probably not a bad idea anyway. It might lead to larger interfaces than absolutely necessary, but it also sometimes shows that a class really should be split into two, and one of the classes should contain the other one.

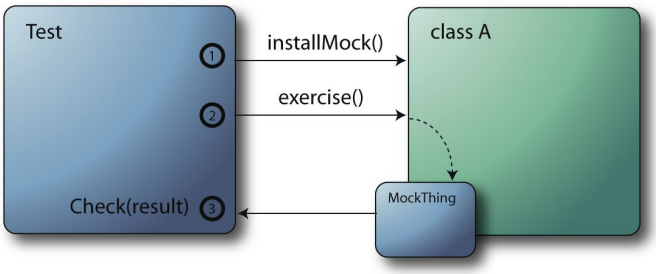


Object interaction

This is the trickiest aspect to test. Here we make a function call, and we want to test that the object under test did a sequence of actions with other objects. We don't really care about state, just that a certain number of function calls happened. The most common testing pattern for this situation is a *mock object*. A mock object is an object that implements the interface of another object, but its only purpose is to help with the test. For example, a mock object could record what functions were called and what values were passed, or could be set up to return a set of fixed values when one of its functions is called.

Because this is the most complex form of testing, we recommend using it only when the other two forms of testing are not possible. Using mock objects frequently could be an indication that the code relies too much on heavy objects with complex interactions instead of many, loosely-coupled, simpler objects.

A good situation for using a mock object could be testing HUD rendering. We want to verify that the HUD elements are rendered in a specific order, so we create a mock for the rendering canvas which will keep an ordered list of the elements that get passed to it. We pass the mocked rendering canvas to the HUD renderer and make the render call. Then we can examine that the list in our mocked canvas matches our expectations.



5. Best practices

These is a set of best practices we have found to be particularly important for all of our development with TDD.

Run tests frequently

Being able to write tests easily is a requirement for successfully rolling out TDD. Nobody wants to spend extra time writing unit tests when they can be implementing the features that are due for this milestone. But even more importantly, tests should run very frequently, and a failed test should be treated the same way as a failed build.

At High Moon, every library has a separate test project that creates an executable that links with the library and runs all the tests. We have hooked up running the executable itself as the post-build step in Visual Studio (or as the last command in a make file). That means that making any changes to the library or tests and triggering a compile will also run all the tests. Additionally, since the test executable returns a value with the number of failed tests, a failing test will return a non-zero value, which is interpreted by the build chain as a failed build. This makes it so everybody is running unit tests for all code all the time, which greatly improves the build stability.

In addition to running tests locally, the build server also runs all the unit tests at the same time it does code builds. In our case, since the tests are a postbuild step, there was nothing special we had to do in the build server, and a failed test would be reported just as code that didn't compile correctly.

Test only code under test

This is a key practice for writing good unit tests. The most important reason to minimize the amount of code involved in a test is to keep things simple. Ideally, when something breaks, you want only a small number of tests failing so you can quickly pinpoint the problem. If some tests involve a large amount of the codebase, they will constantly break for unrelated reasons.

Also, if you are able to use the code under test with a minimal amount of other code or libraries, it means you are creating very modular, self-contained code, which will help with the overall design. Finally, another very good reason to avoid involving extra code is that tests that only work on a small set of code are typically much faster than tests that involve large systems and complex initialization/shutdown sequences.

Continuing with the health powerup example, notice that the test involved a `World` object, a `Player` object, and a `HealthPowerup` object. There was no graphics system initialization, databases, etc. The `World` object can be a very lightweight container for game entities without any other dependencies. Try setting something like that in your current engine and you might be surprised by how many implicit dependencies you find in different parts of the engine.

A test that involves a lot of code across many different systems is usually referred to as a *functional test*

(also called a *customer test*). Functional tests are extremely useful, especially if they are fully automated, but they fill a very different role than unit tests.

Keep tests simple

This is related to the previous best practice. You want a test to check one thing and only one thing; that way, when something breaks, it's immediately clear what went wrong.

A good start is to label each test clearly with a name that describes exactly what the test is supposed to do. For example, a test named `PlayerHealth` is not very helpful. A much better name would be `PlayerHealthGoesUpWhenRunningOverHealthPowerup`.

Keeping each unit test to just a handful of lines makes it much easier to understand at a glance. In our case, it is unusual to have unit tests that are more than 15 lines long. Needing to write overly long unit tests is usually a sign that the test is involving too much code and could probably be re-written in a better way.

We also found that limiting each unit test to a single check statement or two resulted in tests that were the easiest to understand. Sometimes that means you'll have to write some duplicate setup code between two tests, but it's a small price to pay to keep the tests as simple as possible.

Whenever you have some common code in two or more tests, you can use a fixture. A fixture is a set of common code that is executed before and after each test that uses that fixture. Using fixtures can cut down tremendously on the amount of test code you have to write. Any decent unit test framework should support fixtures, so use them whenever they're needed.

Keep tests independent

Unit tests should be completely independent of each other. Creating an object in a test and reusing it in a different test is asking for trouble for the same reasons as we discussed earlier: when a test fails, you want to be able to zero in on the failing test and the problem that is causing it to fail. Having tests depend on each other will cause chains of failing tests, making it difficult to track the problem down to the source.

Keep tests very fast

The unit tests we write are compiled and executed as a postbuild step. We have hundreds or thousands of tests per library, so that means they have to run blazingly fast or they'll get in the way. If unit tests are only dealing with the minimum amount of code, they should be able to run really fast. That means they shouldn't be talking to the hardware, they shouldn't be initializing and shutting down expensive systems, and they most definitely should not be doing any file I/O.

All our unit tests are timed (another feature of UnitTest++), and the overall time for the test run is printed after it runs. As a general rule, whenever a set of unit tests goes over two seconds, it means something is wrong and we try to fix it (even if a unit test takes a full ms, which is a huge amount for a unit test, you can still run 1000 of them in a single second).

6. TDD and game development

Applying TDD to game development has its own unique challenges that are not usually discussed in the TDD literature.

Different platforms

Most game developers today need to develop for a variety of platforms: PCs (Windows, Macs, or Linux), game consoles, handhelds, etc. Even though at High Moon we develop console games, we use Windows as our primary development environment because of the good development tools and the fast iteration time. That means we always have a version of our engine and tools that runs under Windows, which makes running unit tests very easy and convenient.

We also wanted to run the unit tests in each platform we develop for, but we had to make a few changes to compensate for the shortcomings of those platforms. The minimum support that we needed was the ability to run an executable through the command line and capture the output and, ideally, the return code. Surprisingly, none of the game console environments we develop for supports that simple operation, so we were forced to write a small set of programs using the system API to do exactly that.

After we wrote those small utility programs, we were able to run unit tests on the consoles just like we did under Windows. The main difference was that running any program on the consoles had a noticeable startup time delay. It's a matter of just a few seconds, but it was too long to run the unit tests automatically as a postbuild step after every compilation. Besides, we don't yet have one dev kit for every developer station, so we couldn't count on always having a target platform available. Because of that, our unit tests on platforms other than Windows are only executed manually, and by the build server. It is not ideal, but the large majority of the problems show up in the Windows tests, so as long as those continue being run all the time, we catch most problems on time.

Graphics, middleware, and other APIs

Probably the biggest barrier that people see to doing unit testing and TDD with games is how to deal with graphics. It's certainly not the textbook-perfect example you'll read about in TDD books, but it's certainly possible.

The first thing to realize is that graphics are just part of a game. It is common sense and a good software engineering practice to keep all the graphics-related code in one library or module, and make the rest of the game independent of the platform graphics API and hardware. Once we had that organization, we were able to test any part of the game or engine without having to worry about graphics.

Ideally, we wanted to develop the graphics renderer library with TDD as well, and there's no way to avoid dealing with platform-specific graphics calls. We tried three different approaches, from most involved to least involved:

- **Catch all graphics function calls**. We inserted a layer between the graphics renderer and the graphics API that exactly mimicked the platform API. That way we could make any graphics API calls that we needed without having to worry about the underlying hardware. As a bonus, that layer could report which functions were called and with what parameters. This approach made for extremely thorough tests. The testing layer could be removed in the final build and the functions would call directly into the graphics API so there would be no performance penalties. However, it was quite labor-intensive, especially for the Direct3D API because it uses classes and not just plain functions like OpenGL.
- **Check for state**. Another approach is to actually work on the graphics hardware and check that

things are working correctly by querying the graphics API state. For example, rendering a certain mesh should set a specific vertex declaration. This approach is much simpler, but there are some things we couldn't test. For example, we could render a mesh, but we wouldn't be able to find out how many triangles were sent to the hardware. Also, since OpenGL is so state-based, it was important for most functions to clean up after themselves, so it was often very hard to check for any state. On platforms in which the internals of the graphics API are more open, you might be able to examine more states by looking at the graphics command buffer.

- **Isolate graphics calls**. When all else fails, this is a useful technique for dealing with calls into external APIs. Isolate an API function call or a group of API function calls into a single function, and test that your function is called at the right time. The function that calls into the graphics API itself won't be tested, but all it does it make some straight calls. Remember, what we really want to test is that our code behaves correctly, not that the graphics API works as documented.

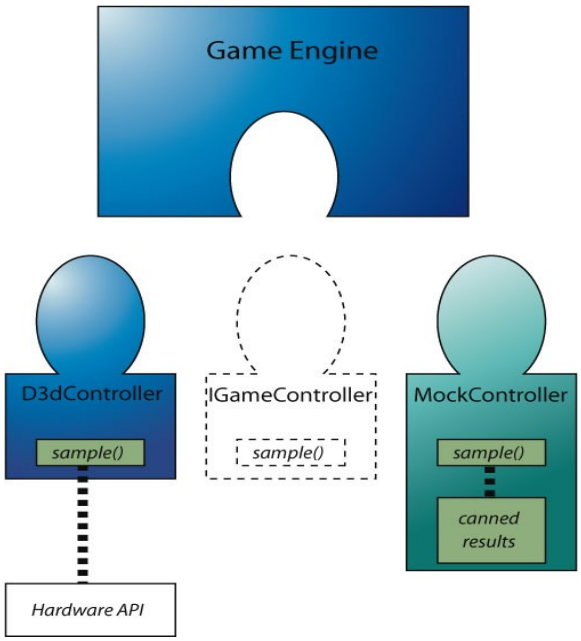
We started with the first approach, wrapping the API as we were using it, but it the amount of extra work we had to do to wrap complex APIs like Direct3D and OpenGL quickly became overwhelming. Maybe if we were working on a commercial graphics rendering middleware it would be worth the effort. For us, after doing that for a few weeks, we realized that the benefits we were deriving from it weren't worth the time we were spending. It was also discouraging us from using new API functions just because they hadn't been wrapped before.

In the end, we ended up settling for a combination for the second and third approaches: testing the state whenever possible, and isolating the calls the rest of the time. One of the drawbacks of this approach is that we're working directly with the hardware, so tests actually do initialize and shutdown the graphics system every time (except for those platforms out there that can only initialize the graphics system once), so they can be more time consuming. On the positive side, because the tests are exercising the graphics API and hardware directly, we catch things that the first approach wouldn't have caught (for example, sending incorrect parameters to a function).

The same three approaches can be used for any middleware or external API. The important thing to remember is to make sure you're testing your code, not the API itself. Keeping that in mind helps to write true unit tests and not functional tests for the API.

A good example of this approach is how we wrote our input system with TDD. The input system deals with getting input values from gamepads and other controllers. Even though at first glance it might seem like the whole system is about making system calls to poll the data, there is a lot of common code that is totally platform independent: button mappings, edge detection, filtering, plugging/unplugging controllers, etc.

In our system, we have an interface named `GameController` with a Sample() function. Each platform implements a platform-specific version of the `GameController` (for example `D3DController`) and does the raw sampling of all buttons and axis through platform-specific API calls. That part is fully untested. The rest of the input system works through the `GameController` interface, but for all the tests we provide a `MockGameController()` which allows us to control what input values we feed to the tests.



We are hoping that as TDD becomes more common in the games industry, middleware providers will make their APIs more TDD friendly and even ship with their unit tests.

Third-party game engines

If dealing with APIs was not straightforward, working with a full third-party game engine that was not developed with TDD is even more challenging. After all, an API is just a list of classes or functions, and you have a lot of control over how and when they get called. Working with a full game engine, you might end up writing a small module that is fully surrounded by the engine code. If the engine was not developed with TDD, it can be very difficult to use your code in isolation or figure out how to break things up so they can be tested.

At High Moon, we are working on several projects with the Unreal Engine 3, and we're using TDD on them. Even though the engine is not TDD-friendly at all, we still find more benefit from doing TDD with it. Imagine then how useful TDD can be on a full engine developed with TDD from the start.

The most important thing to do in this situation is to separate the code we write as much as possible from the engine code. Not only does this allow us to unit-test our code in isolation, but we also keep future merges with Epic's codebase as simple as possible. However, that severely limits the amount of large-scale refactorings we can do to keep things as testable as possible, so it's a tough trade-off..

One unique aspect of the Unreal Engine is that it makes heavy use of its scripting language, UnrealScript. Because a lot of the high-level game engine is written in UnrealScript, we had little choice but to use it to write most of the game code. The first thing we had to do was to create a unit-testing framework for UnrealScript. The resulting framework is called UnUnit and is freely available for download through UDN (Unreal Developer Network). Several companies working with the Unreal Engine are currently using UnUnit for their game projects.

UnrealScript is a surprisingly good language for unit testing. By default, all functions are virtual, so overriding behaviors and creating mocks is simpler than C++. Also, compilation is very fast, which keeps iteration times low (large, badly organized C++ codebases can take a long time to link). All of that

makes TDD with UnrealScript not just possible, but very effective.

Randomness and games

Most games involve a fair amount of randomness: the next footstep sound you play can be any one of a set of sounds, the next particle emitted has a random speed between a minimum and a maximum, etc. As a general rule, you want to remove the randomness from your tests. A few times we caught ourselves starting to write a unit test that called a function many times in a loop and then averaged the result, but that's totally the wrong way to go. Unit tests are supposed to be simple and fast, so any loops in a test are usually very suspect.

A better approach is to separate the random decision from the code that uses it. For example, instead of just having a `PlayFootstep()` function that takes care of computing a random footstep and then playing it, we can break it into int `ComputeNextFootstep()` which is just a random function call, and a `PlayFootstep(int index)`, which we can now test very easily.

Another approach is to take control over the random number generator at the beginning of the test and rig the output so we know what sequence of numbers is going to come up. We can do this either by mocking the random number generator object or by setting some global state on the random number generator, depending on how it is implemented. Then tests can count on a specific sequence of numbers being generated and check the results accordingly.

High-level game scripts

How far is it worth taking TDD? Should TDD be used for every single line of code? How about game-specific script code? The answer depends on your priorities and game.

As a general rule, we find that if any other part of the game is going to depend on the code we are writing, then it's probably worth doing it with TDD and having a full set of unit tests for it. Otherwise, if it's a one-shot deal with the highest-level code, then it's probably fine without TDD. Also, code at that level is often written by designers in a game-specific scripting language, so TDD might not be a viable option.

An example of some code that we would not use TDD for is trigger code: when the player goes around the corner, wake up two AIs and trigger a different background music.

Functional tests are a great complement to unit tests, so it's important not to forget about them. Automated functional tests can be extremely useful catching high-level problems, gathering performance and memory utilization data, and removing some of the mechanical testing from QA and letting them concentrate on issues such as gameplay balance and flow.

7. Lessons learned

Design and TDD

One of the most important benefits we get from TDD is the better code design that it creates. For this to

happen, it's important to let the tests guide the code and not the other way around. It can be disconcerting to always be looking only a few minutes into the future and implementing the “simplest thing that could possibly work,” but it really works. The key to working with TDD is to realize that refactoring is an integral part of the development process and should happen regularly every few tests. Good design will come up through those refactorings as needed by the tests we have written and the code we have implemented.

Does this mean you shouldn't do any design ahead of time? That depends on your situation and experience, and the type of code you're writing. In general, the less known or more likely to change something is, the less design we do. If you're working on the 10th iteration of a well-known sports game franchise, for a known platform, and you know exactly what you're going to do, up front design can be more beneficial.

In our case, we prefer to discuss very rough concepts of where we expect something to go, what it should do in the future, and maybe some very, very rough organizational structure. In general, we prefer not to even think of classes or draw UML diagrams on whiteboards before starting, because such discussions can then lead implementation too much. We prefer to let the tests guide us, and if at some point we're going away from what we had in mind at the beginning, we can stop to reconsider if we're heading in the right direction. Most of the time we are, and we simply hadn't thought things through enough at the beginning (or thought through them too much and we simply didn't need that level of flexibility).

TDD and high-level code

One of the questions we had when we jumped into TDD is whether it was going to hold for high-level code. We had seen in practice from previous projects that we can certainly do TDD to create low-level and intermediate-level libraries (math, collision, messaging, etc). But would it really work for high-level code that would build on low-level code?

The answer is an unconditional yes. We have developed a full codebase doing TDD from the start, and we had no difficulty writing high-level code with TDD. Things like character state machines, game flow, or specific game entities were done through TDD without any problems, and greatly benefited from the TDD approach.

Two tips that helped us apply TDD to high-level code:

- Keep tests as true unit tests. When working on high-level code, it can be tempting to let tests degenerate into functional tests that involve the whole game engine. Don't do that. Even if it takes a few more minutes to make an enemy character testable in isolation, it is well worth it in the long run.
- Keep the engine architecture as flat as possible. This is a general good practice, but it is more so with TDD. Clearly, avoid having your engine as one big, intertwined module. But even if it's broken down into libraries or modules, keep them as independent of each other as possible instead of as one long list of dependencies. For example, there's no reason the AI module needs to know anything about graphics or sound, but it will need to know about a world representation and have access to the messaging system.

Tests as a measure of progress

Software developers have been struggling for a long time to find a good measure of progress or work done. Some projects use code line counts to determine progress, others use features completed, while others just look at the number of hours spent at the office. Clearly, none of those approaches are ideal.

We have found that counting the number of unit tests is a really good measure of progress. Especially when tests are developed through TDD, they deal less with corner cases and more with program features. If a library has 500 tests associated with it, we can say that it's roughly half as complex as a library with 1000 tests.

As an added benefit, people tend to behave based on how they think they are being measured. So if this is made clear, people will be much more likely to be strict about applying TDD and not falling back on writing code without tests. In our groups, we have a test chart, which is updated every day and shows the total count of tests. If tests aren't going up, or they're going up more slowly than other times, we know something is wrong and we're not making much progress.

Build stability

As we expected, having almost full code coverage with unit tests greatly improves the code stability. Broken builds happen much less frequently, and they're usually caused by a missing file or a different platform not compiling correctly. What was a pleasant surprise is that broken builds are much easier to fix. By following our best practices for unit tests, it becomes really obvious when something breaks, and we can usually check in a fix right away. Builds are rarely broken for more than a few minutes at a time, which can completely change how you organize your code in source control.

Amount of code

It will probably come as no surprise to anybody that writing code with TDD results in more code being written. Sometimes the test code is as large as the code under test itself. Even though this can initially sound like a scary proposition, it really isn't a problem. The extra code does not take significantly longer to write initially, and it allows us to move a lot faster once we have accumulated more code and we need to refactor or optimize the code in any way.

Just because we have twice as much code it doesn't mean we have twice the complexity. Quite the contrary. The test code has been written so it's extremely simple, so its complexity is minimal. Its only job is to check the non-test code, so it's keeping an eye out for us, helping us, and letting us know when something breaks. Additionally, the TDD approach results in much simpler, decoupled code. Overall, the complexity of the codebase is greatly reduced with TDD.

A good analogy to TDD is scaffolding in a building construction. It's not something you're going to deliver to your customer, but it's an absolute necessity, it needs some time commitment to set it up, and you wouldn't dream of doing any complex building without it.

Development speed

This is the million-dollar question: Does TDD slow development? We're not doing TDD because it's a

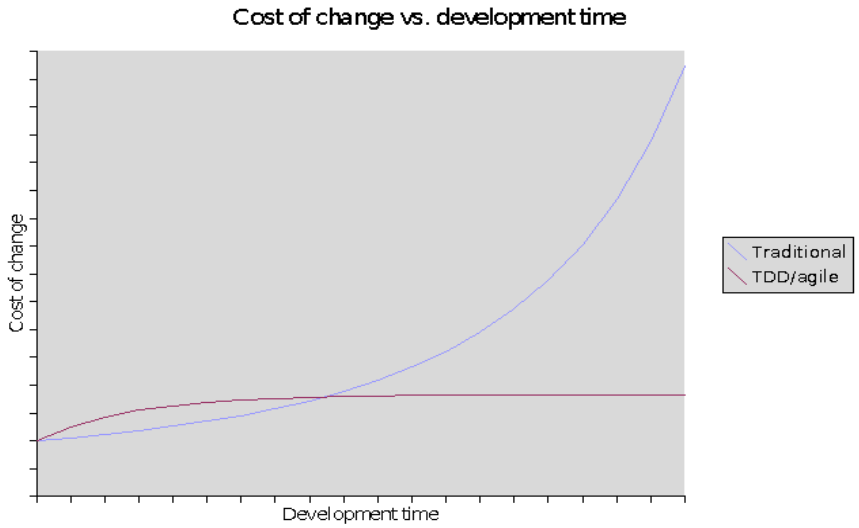
“good” thing to do, but because we want to ship a better-quality product faster and cheaper than we did before. If TDD doesn't help with this, then there's very little point to it (other than keeping programmers happy).

As with other aspects of software development, it is difficult to make an objective study and measure exactly the effects of TDD versus a control group. Things change too much from project to project and team to team. Still, some initial studies have some interesting initial findings (http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf), even if the study was not very rigorous and had a very small sample.

Our experience is that TDD, like any other new development technique, slows development down at the beginning while the team is learning it and becoming familiar with it. This can take as long as a couple of months. Once the team is over the hump, and given the right tools and development environment, the impact of TDD on development speed is minimal.

It is possible simply write code faster than it is to write the tests first, but as soon as that code needs to be refactored, debugged, used by somebody else, or simply gets more complex, any time savings quickly disappear. The larger the team and the more complex the problem, the more TDD saves time in the long run. It's very important to not fall for the temptation of skipping TDD for a very short-term gain (aka milestone of the month).

Ideally, TDD and refactoring can flatten out the classical cost-of-change over time curve into something like this. Notice the trade-off between slightly slower short-term speed vs. massive gains later on.



Adopting TDD

We started with TDD by first trying it with a small group on a separate project. In our particular case, not only were we doing TDD, but we were doing all the extreme programming practices (pair programming, continuous integration, collective code ownership, etc). When doing this, it can be extremely useful to have somebody on board with previous TDD experience who can help guide the process, set up the environment, and avoid common early mistakes.

Applying it on a small scale initially was very useful in many different ways. It let the team become

more comfortable with TDD and get to the point where they are as productive as they were before. It also makes any bad practices apparent early on and they can be dealt with before rolling it out (making overly complex tests or functional, rather than unit tests).

It also had the effect of creating new evangelists for the new technique. Members of that team were then moved on to other teams, where they became the resident TDD expert.

TDD fits best with other agile development practices. Having an agile mindset fits very well with the idea of finding the design through tests. Pair programming can be extremely helpful, especially at the beginning while rolling out TDD and getting everybody on board. Pair programing also has the added advantage of making programmers less likely to skip writing tests, which can be a common reaction early on.

If you're interested in applying TDD but you don't have a commitment from your manager or lead, it is possible to start doing it on the side on your assigned tasks. The most important thing is to make sure your tests run automatically (remember the postbuild trick). Slowly, other people might see how useful the tests are, or how much easier it is to refactor that code, and eventually the time might be right to roll out TDD.

A few people can have a hard time switching over to the TDD mentality, but to take full advantage of TDD, it is best to let the design emerge from the tests and the refactoring rather than trying to plan everything up front. It is very interesting to note though, that most programmers at High Moon quickly accepted TDD, and soon became very enthusiastic about it and started using it in all their code, including their home projects.

There is no doubt that the best situation to roll out TDD is with a fresh new codebase. Not everybody has that luxury, so it is important to learn how to apply TDD even with an existing legacy codebase. Michael Feathers' book *Working Effectively with Legacy Code*, explains exactly that situation and gives some very good guidelines on how to go about unit testing with a codebase without existing unit tests. Sometimes it just takes a little bit of refactoring of the existing code and it becomes a lot easier to add new tests.

8. Conclusion

Test-driven development can be a very effective development technique. We have successfully applied it to game development in a variety of situations, and we're convinced of the many benefits it has provided us. Right now, the idea of writing code without writing tests first feels quite alien to most of us, and we treat TDD like the scaffolding in building construction: a necessary tool that will not be shipped to the customer but that helps tremendously during development.

9. Resources

This is required reading before you start doing any TDD:

- Beck, Kent, *Test Driven Development: By Example.* Addison-Wesley, 2002

These other books will also come in handy:

- Fowler, Martin, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999
- Astels, David, *Test Driven Development: A Practical Guide.* Prentice Hall, 2003
- Beck, Kent, and Cynthia Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley, 2004

Very useful mailing lists and web sites:

- TestDriven.com (http://www.testdriven.com)
- TDD mailing list (http://groups.yahoo.com/group/testdrivendevelopment)

Resources dealing with TDD and agile game development:

- Noel's blog, Games from Within (http://www.gamesfromwithin.com)
- Clinton Keith's blog, Agile Game Development (http://www.agilegamedevelopment.com/blog.html)

C++ unit-testing frameworks:

- UnitTest++ (http://unittest-cpp.sourceforge.net)
- CppUnit (http://cppunit.sourceforge.net/cppunit-wiki)
- Comparison of C++ unit-test frameworks (http://www.gamesfromwithin.com/articles/0412/000061.html)