DOWNLOAD PDF TEST DRIVEN DEVELOPMENT BY EXAMPLE C

Chapter 1 : .NET TDD (Test Driven Development) by example - Part 1 - CodeProject

Test-Driven Development starts with designing and developing tests for every small functionality of an application. In TDD approach, first, the test is developed which specifies and validates what the code will do.

The following sequence of steps is generally followed: This article will focus on the actual test and implementation using variation of one of the Roy Osherove Katas. This excercise is best done when not all requirements are known in advance. Below you will find the test code related to each requirement and afterwards the actual implementation. Try to read only one requirement, write the tests and the implementation yourself and compare it with the results from this article. Remember that there are many different ways to write tests and implementation. This article is only one out of many possible solutions. The following input is ok: The first line is optional. If there are multiple negatives, show all of them in the exception message stop here if you are a beginner. Make sure you can also handle multiple delimiters with length longer than one char Even though this is a very simple program, just looking at those requirements can be overwhelming. Forget what you just read and let us go through the requirements one by one. Create a simple String calculator Requirement 1: The method can take 0, 1 or 2 numbers separated by comma ,. From here on, for brevity reasons, only modified parts of the code will be displayed. Whole code divided into requirements can be obtained from the GitHub repository tests and implementation. Run all the tests again and see them pass. However, once tests are executed, the first test failed. Allow the Add method to handle new lines between numbers instead of commas. Support different delimiters To change a delimiter, the beginning of the string will contain a separate line that looks like this: We split the code into 2 methods. Initial method parses the input looking for the delimiter and later on calls the new one that does the actual sum. Since we already have tests that cover all existing functionality, it was safe to do the refactoring. If anything went wrong, one of the tests would find the problem. If there are multiple negatives, show all of them in the exception message. First one checks whether exception is thrown when there are negative numbers. The second one verifies whether the exception message is correct. Numbers bigger than should be ignored Example: There are 3 more requirements left. I encourage you to try them by yourself. Delimiters can be of any length Following format should be used: Allow multiple delimiters Following format should be used: Make sure you can also handle multiple delimiters with length longer than one char Give TDD a chance This whole process often looks overwhelming to TDD beginners. One of the common complains is that TDD slows down the development process. It is true that at first it takes time to get into speed. However, after a bit of practice development using TDD process saves time, produces better design, allows easy and safe refactoring, increases quality and test coverage and, last but not least, makes sure that software is always tested. Another great benefit of TDD is that tests serve as a living documentation. It is enough to look at tests to know what each software unit should do. That documentation is always up to date as long as all tests are passing. Together they are covering both unit and functional tests, serving as full documentation and requirements. TDD makes you focus on your task, code exactly what you need, think from outside and, ultimately, a better programmer. It was a long, demanding, but very rewarding journey that resulted in a very comprehensive hands-on material for all Java developers interested in learning or improving their TDD skills. It contains extensive tutorials, guidelines and exercises for all Java developers eager to learn how to successfully apply TDD practices. You can download a sample or purchase your own copy directly from Packt or Amazon.

Chapter 2 : Test Driven Development (TDD): Example Walkthrough | Technology Conversations

test driven development by example c Wed, 31 Oct GMT test driven development by example pdf - Ã¢â‚¬Å“CleverÃ¢â‚¬ play on words in the title.

In essence, unit tests are pieces of code which assert that your production code works as intended. With a sufficient amount of unit tests it is possible to refactor your production code at any time, because the safety net of knowing when you broke something gives you courage. Typically, programmers write code and test it afterwards. Code which is not easily unit testable is not tested or integration tested. Execution paths are overlooked, boundary cases get ignored. There is not much safety for future changes, because there is no machine-executable specification of how less well-known regions of the code are supposed to behave. Enter test-driven development TDD. The essence of test-driven development TDD has been around for a few years. It is the state-of-the-art programming style when it comes to work efficiency and software maintainabilty. The basic idea is to change the order in which production code and test code are written: Write a small amount of test code which your current production code cannot pass. Make sure your test fails. This step is important since it tests your test; you gain confidence that your test can detect errors in your program code. Add just enough production code to pass the test you just wrote and all previous tests. Let all your tests run again. Make sure all of them pass even though you know they will. Look at your both test and production code. Identify code and data duplication and remove it. Apply other refactorings as you see fit. Yet again, check all tests pass. Repeat this cycle until you are done. All you need is a compiler, a unit test framework such as CppUnit, and a simple task to start practicing. I recommend to pick something not related to your work to get yourself started. For example, write a function which converts arabic numbers to Roman numerals. The task should be easy so that you can focus on learning TDD, but it should not be trivial. When you start writing tests, think about the smallest possible steps. Here are some things you could do: Include a header file in your test which shall hold your production code once you are done. The compiler will complain about not being able to find the file in question. Fix this test by creating an empty header file. Use a non-existing class or function in your test. The compiler will complain about the unknown symbol. Add the declaration to your header file. The linker will complain about the symbol not being defined anywhere. Add a source file which holds the definition of your symbol. Use a non-existent namespace in your test. Again, the compiler cannot do its job unless you add the namespace to your production code. Write simple tests which assert the return value of a function. Fake your implementation by returning the very same value. Write another test for the same function. Improve on your fake implementation to let both tests pass at the same time. Really, is TDD that slow? From above items you might infer that TDD is a ridiculously slow technique and this article is not worth the bytes it is encoded in. I might be biased regarding the latter conclusion, but TDD is not for working slowly. TDD empowers you to work as fast as you currently can: Never forget to write tests before production code, though, and never write production code which is not covered by your current tests. If you are tired and you are working on a difficult problem, decrease your step size until you feel confident. Slowing down is not a sign of weakness or lack of skill. Everybody can step on the gas. The best race drivers, though, know exactly when to brake. Costs of working test-driven Some programmers complain that writing tests takes time which would be better invested in implementing new features. This does not imply that these programmers do not test their code. They do, but they do it on-the-fly: Never kill the messenger. Besides well-tested production code, you also have a set of tests which illustrate how to use your code. At a later stage, you simply need to reevaluate your tests to prove your code works. In my experience, the best side-effect of TDD is that you design your code to be easily testable. Units become smaller, functions more cohesive, and classes obey the single responsibility principle. In a word, your code becomes cleaner. Fun and productivity I guess it is safe to say that most of us have made programming our profession or hobby because we enjoy it. Oh boy was I wrong. With TDD, a sense of immediate progress is your steady companion. It is like playing a good computer game, there is action all the time. More than once I missed my ride back home because I just wanted to write one more test. For me, TDD is the most enjoyable style of programming. For my employer, it is the

most efficient one. Because I write tests before code, I think more about boundaries. Whenever I make a mistake and introduce a bug in my production code, my tests immediately provide feedback and I can fix the issue while my knowledge of the code is still fresh. Months later, I can add new features without breaking anything.

Chapter 3 : tdd - Test Driven Development with C++ - Stack Overflow

Basics of test-driven development Test-driven development. Test-driven development is a software development process that relies on the repetition of a very short development cycle.

Start reading Book Description Clean code that works--now. This is the seeming contradiction that lies behind much of the pain of programming. Test-driven development replies to this contradiction with a paradox--test the program before you write it. Since the dawn of computing, programmers have been specifying the inputs and outputs before programming precisely. Test-driven development takes this age-old idea, mixes it with modern languages and programming environments, and cooks up a tasty stew guaranteed to satisfy your appetite for clean code that works--now. Developers face complex programming challenges every day, yet they are not always readily prepared to determine the best solution. More often than not, such difficult projects generate a great deal of stress and bad code. To garner the strength and courage needed to surmount seemingly Herculean tasks, programmers should look to test-driven development TDD , a proven set of techniques that encourage simple designs and test suites that inspire confidence. By driving development with automated tests and then eliminating duplication, any developer can write reliable, bug-free code no matter what its level of complexity. Moreover, TDD encourages programmers to learn quickly, communicate more clearly, and seek out constructive feedback. Readers will learn to: Solve complicated tasks, beginning with the simple and proceeding to the more complex. Write automated tests before coding. Grow a design organically by refactoring to add design decisions one at a time. Create tests for more complicated logic, including reflection and exceptions. Use patterns to decide what tests to write. Create tests using xUnit, the architecture at the heart of many programmer-oriented testing tools. This book follows two TDD projects from start to finish, illustrating techniques programmers can use to easily and dramatically increase the quality of their work. The examples are followed by references to the featured TDD patterns and refactorings. With its emphasis on agile methods and fast development strategies, Test-Driven Development is sure to inspire readers to embrace these under-utilized but powerful techniques.

Chapter 4 : Test Driven / First Development by Example - CodeProject

Introduction - Defining the Battlefield This tutorial is an short introduction to using Test Driven Development (TDD) in Visual Studio (VS) with C#.

Better code quality through Red, Green, Refactor Documentation that grows as we develop and remains up to date Automatic regression test harness This will primarily involve creating unit tests first, having them fail, making them pass and then refactoring the code to be of better quality and then re-running the tests. It also helps the thought processes while designing and developing an application or feature to be more targeted. We will focus on a subset of TDD that encourages developer testing and aids tremendously in shipping software rather than traditionally having testing as a secondary phase or responsibility of a tester and promotes testing as a first class citizen in our everyday software development lifecycle SDLC. How many times have you intended to write unit tests after a feature has been created and due to time constraints ended up leaving it and moving onto the next part of the application with an uncertain feeling that it would have been better to have them in place before adding more complex layers? Following a TDD approach eliminates this as the tests are the first thing to consider as part of an initial implementation. Development Costs Fixing bugs after software is shipped has been proven to be much more expensive than having unit tests in place that can be run each time code is about to be checked into source control for minor or substantial changes to the system. Although not a direct replacement for any of the above, having a large set of unit tests in place gives piece of mind that each days development has been a cost effective one and the code is still in good shape. It consists of two C class libraries. The first contains an MS Test class library which will contain our unit tests and the second is a standard class library which we will use to develop the functionality. The unit test project class library is kept separate to ensure we are only testing the public parts of our business logic, without exposing internals to the tests, which are likely to change over time. The Tools There are numerous unit testing frameworks available. Using NUnit , one of my favourites, or other frameworks would work also and is down to personal preference or the infrastructure you work in. Using MS Test helps the article code to run with no other dependencies. Visual Studio MS Test included as part of Visual Studio Creating the libraries Because we are working with a TDD approach, it has forced me to think a little ahead of time, even before naming my project. I know I want to provide a library for roman numerals but if I want to test it, the more loosely coupled I keep it, the more it will remain testable. Next we will add the Unit Test project as show below: Lastly for this step, we will create the class library to hold our business logic under test as shown below: We can go ahead and perform the following: It further uses this reference with the using namespace and adds attributes to both the class and method, allowing it to know what classes should be used for testing runs and which classes just act as helpers or are just our own code. In TDD, we are encouraged to first make our test fail with as minimal code as possible. As you get more familiar with the process you can use some reasonable judgment on this and my particular take is to create the minimal piece of usable code as a first step and ensure the test fails. Going Back to the Requirements My loose requirements dictate that I can get a roman numeral for numbers between 1 and So here we could also say that entering numbers less than 1 and greater than should fail both the library but more importantly at this stage the test as we are working in test first. Our First Two Tests - RED We will leave the existing test method in place and in the code editor, add two new tests, following the same attribute guidelines and the results will be as follows: Do the same for the method definition as well. At this stage we have a new. The test methods are now satisfied and the project builds. The new class is shown below: The Test Explorer allows drilling down into individual tests and aids in diagnosing the reason for a test failure. Have a play with the test explorer and get familiar with it. You can run tests in different ways also, you will find what works best for you. The amended code is shown below: Green One thing to note at this point is that it is generally good practice to perform the test as a single statement, rather than having branching logic inside the test, this really aids in ensuring the tests are still valid and also deterministic each time you run them. In this case the test is always expecting one thing to happen, an IndexOutOfRange exception. Refactor Refactoring is an important step in the TDD lifecycle. When we refactor we are making changes to the

internals of our business logic without affecting the public API that our tests are consuming. This step may involve a complete re-write of our implementation, renaming variables, adding further abstractions or design patterns. The important thing to remember though is our test library is a client or consumer of our business library as are other clients. Not changing the public external functionality or API will ensure our tests will still validate our system under test, regardless of internals. In fact this acts as a great way to ensure that when we use tools such as resharper to make our code smarter, we can quickly determine if anything has broken by simply re-running the tests. Our first stab at making something work when designing and developing a new feature is most often quite different looking than our final code that has been reviewed, sanity checked and is considered a final implementation. This really helps with not getting too bogged down trying to create a perfect set of code first time round but rather focus on the design of the API or public interface and on what you want to test, leaving refactoring to this point in the lifecycle. If of course in reality you can craft a useful first implementation in a reasonable time frame as your first cut, then go for it. These are guidelines and should be adopted to suit making you productive but at the same time create a testable system. Make Some Changes At this point we can get things tidied up, work some more on the implementation and add features, all with the safe knowledge that we can test at least two scenarios after making changes.

Chapter 5 : C programming and TDD - Stack Overflow

Test -driven development 4 of and members, coupling and cohesion. Either that or we are trying to give ourselves a glimpse through a tiny keyhole at an eternal realm of dynamic order.

Another benefit is that many tools expect that those conventions are followed. There are many naming conventions in use and those presented here are just a drop in the sea. The logic is that any naming convention is better than none. Most important is that everyone on the team knows what conventions are used and is comfortable with them. Separate the implementation from the test code Benefits: Common practice is to have at least two source directories. In bigger projects number of source directories can increase but the separation between implementation and tests should remain. Build tools like Maven and Gradle expect source directories separation as well as naming conventions. Place test classes in the same package as implementation Benefits: Knowing that tests are in the same package as the code they test helps finding them faster. For example, examples in this article are in the package com. As stated in the previous practice, even though packages are the same, classes are in the separate source directories. Name test classes in a similar fashion as classes they test Benefits: One commonly used practice is to name tests the same as implementation classes with suffix Test. If, for example, implementation class is StringCalculator, test class should be StringCalculatorTest. Often, number of lines in test classes is bigger than number of lines in corresponding implementation class. There can be many test methods for each implementation method. To help locate methods that are tested, test classes can be split. Use descriptive names for test methods Benefits: Using method names that describe tests is beneficial when trying to figure out why some test failed or when the coverage should be increased with more tests. It should be clear what conditions are set before the test, what actions are performed and what is the expected outcome. There are many different ways to name test methods. Given describes pre conditions, When describes actions and Then describes the expected outcome. If some test does not have preconditions usually set using Before and BeforeClass annotations , Given can be skipped. An example of BDD format for naming test methods would be: Do NOT rely only on comments to provide information about test objective. Comments do not appear when tests are executed from your favorite IDE nor do they appear in reports generated by CI or build tools. In the example screenshot, both failed tests have the same code inside. The only difference is in the name of the method. Test1 does not give much info regarding the failure. Processes TDD processes are the core set of practices. Successful implementation of TDD depends on practices described in this section. Write the test before writing the implementation code Benefits: By writing or modifying test first, developer is focused on requirements before starting to work on a code. This is the main difference when compared to writing tests after the implementation is done. Additional benefit is that with tests first we are avoiding the danger that tests work as quality checking instead of quality assurance. Only write new code when test is failing Benefits: If new functionality is indeed missing then test always passes and is therefore useless. Test should fail for the expected reason. Even though there are no guarantees that test is verifying the right thing, with fail first and for the expected reason, confidence that verification is correct should be high. Rerun all tests every time implementation code changes Benefits: Every time any part of the implementation code changes, all tests should be run. Ideally, tests are fast to execute and can be run by developer locally. Once code is submitted to version control, all tests should be run again to ensure that there was no problem due to code merges. This is specially important when more than one developer is working on the code. Continuous Integration tools like Jenkins , Hudson , Travis and Bamboo should be used to pull the code from the repository, compile it and run tests. All tests should pass before new test is written Benefits: It is sometimes tempting to write multiple tests before the actual implementation. In other cases, developers ignore problems detected by existing tests and move towards new features. This should be avoided whenever possible. In most cases breaking this rule will only introduce technical debt that will need to be paid with interests. One of the goals of TDD is that the implementation code is almost always working as expected. Some projects, due to pressures to reach the delivery date or maintain the budget, break this rule and dedicate time to new features leaving fixing of the code associated with failed tests for later. Those projects usually end

up postponing the inevitable. Refactor only after all tests are passing Benefits: In most cases there is no need for new tests. Small modifications to existing tests should be enough. Expected outcome of refactoring is to have all tests passing both before and after the code is modified. Development practices Practices listed in this section are focused on the best way to write tests. Write the simplest code to pass the test Benefits: It states that most systems work best if they are kept simple rather than made complex; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided. Write assertions first, act later Benefits: Once assertion is written, purpose of the test is clear and developer can concentrate on the code that will accomplish that assertion and, later on, on the actual implementation. Minimize assertions in each test Benefit: If multiple assertions are used within one test method, it might be hard to tell which of them caused a test failure. This is especially common when tests are executed as part of continuous integration process. When one assert fails, execution of that test method stop. If there are other asserts in that method, they will not be run and information that can be used in debugging is lost. Last but not least, having multiple asserts creates confusion about the objective of the test. This practice does not mean that there should always be only one assert per test method. If there are other asserts that test the same logical condition or unit of functionality, they can be used within the same method. By reading method name and looking at the assert it should be clear what is being tested. First assert is confirming that exception exists and the second that its message is correct. When multiple asserts are used in one test method, they should all contain messages that explain the failure. This way debugging of the failed assert is easier. In case of one assert per test method, messages are welcome but not necessary since it should be clear from the method name what is the objective of the test. It is unclear what is the functionality and if one of them fails it is unknown whether the rest would work or not. It might be hard to understand the failure when this test is executed through some of CI tools. Do not introduce dependencies between tests Benefits: Developers should be able to execute any individual test, set of tests or all of them. Often there is no guarantee that tests will be executed in any particular order. If there are dependencies between tests they might easily be broken with introduction of new tests. Tests should run fast Benefits: Benefit of fast tests, besides fostering their usage, is fast feedback. Sooner the problem is detected, easier it is to fix it. Knowledge about the code that produced the problem is still fresh. If developer already started working on a next feature while waiting for the completion of the execution of tests, he might decide to postpone fixing the problem until that new feature is developed. On the other hand, if he drops his current work to fix the bug, time is lost in context switching. Mocks are prerequisites for fast execution of tests and ability to concentrate on a single unit of functionality. By mocking dependencies external to the method that is being tested developer is able to focus on the task at hand without spending time to set them up. In case of bigger teams, those dependencies might not even be developed. Also, execution of tests without mocks tends to be slow. Good candidates for mocks are databases, other products, services, etc. Mock objects are a big topic and will be described in more details in a future article. Use setup and tear-down methods Benefits: In many cases some code needs to be executed before test class or before each method in a class. For that purpose JUnit has BeforeClass and Before annotations that should be used as the setup phase. BeforeClass executes the associated method before the class is loaded before first test method is run. Before executes the associated method before each test is run.

Chapter 6 : calendrierdelascience.com: Customer reviews: Test Driven Development: By Example

A lot has been written on the subject of test driven development, and especially on the idea that tests ought to be written first. This is an ideal for which I strive. However, I have a tendency to write the unit tests afterwards. Some people learn better by example. This article, rather than going.

Unit testing is a method by which individual units of source code are tested to determine if they are correctly working. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are usually created by developers. The goal of unit testing is to isolate each part of the program, and show that the individual parts are correctly working. A unit test is a strict, written contract that the piece of code must satisfy. Use of unit tests has several benefits: By testing the parts of a program first, and then testing the sum of its parts, integration testing becomes much easier; unit testing provides a sort of living documentation for the system. Unit testing frameworks To simplify development of unit tests, unit test frameworks are usually used. Unit testing framework should provide following functionality: Writing of unit tests should be simple and obvious for new users. Framework should allow advanced users to perform nontrivial tests. Test module should be able to have many small test cases and developer should be able to group them into test suites. At the beginning of the development users may want to see verbose and descriptive error message, whereas during the regression testing they may just want only to know if are any failed tests. For small test modules execution time should prevail over compilation time: Execution of individual tests should be independent on other tests. Almost any programming language now has several unit testing frameworks. Each of such frameworks consists from: How to organize tests Usually unit tests should be created for all publicly exposed functions â€" free functions not declared as static, and all public functions of classes, including public constructors and operators. Unit tests should cover all main paths through functions, including different branches of conditionals, loops, etc. You can find more advices on unit tests code organization in following article. Test cases are often combined into test suites by some criteria â€" common functionality, different use cases for same functions, common fixtures, etc. Fixtures are used to perform setup and cleanup of data that are needed to perform test cases â€" this also allows unit tests to be very short and easy to understand. There are some recommended ways to implement tests: Some people argues, that combining of all test cases into big functions, improves readability of code, and make it more concise. But there are arguments against this approach some of them are mentioned in following document: Testability of code also depends on its design. There are some suggestions on how code should be written to allow easier writing of unit tests for it: More advices on writing testable code you can find in following blog post. Mocking In a unit test, mock objects can simulate behavior of complex, real non-mock objects and they are very useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object instead: 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 that will be returned. Thus, the behavior of a complex object, such as a network socket, can be emulated by a mock object, allowing the programmer to discover whether the object being tested responds appropriately to the wide variety of states, such objects may be in. Typical workflow looks following way: Inside this test case you do following: Currently most popular are Boost. Test has following features: Progress visualization Cross-platform works on all platforms, supported by Boost licensed under Boost License, that allows to use it anywhere without restriction Has very good documentation , including tutorials The only drawback is that it lacks of mocking features, although Google Mocking framework could be used together with it. Test could be used differently, depending on complexity of tests. User can either write test functions themself, and register them manually, forming a hierarchy of tests, or he can use special macros, that will register test automatically. Usually, code written for Boost. Test, consists from several objects: Developer may control behaviour of execution monitor through command-line options, environment variables, or from source code. Tests with automatic registration For simple tests the use of Boost. Minimal test program Here is minimal example, that defines one test: After

compilation you can run this program, and it will print following on the screen Boost. Running 1 test case Use of test suites If you have many test cases in one program, then their maintenance could be hard. Test provides many different testing tools "Checkers". Execution of test case is aborted. This check should be used for things, like checking is object, that will be used later in the code, was created. Complete list of checkers you can find in reference. Test will report line in source code where this happened, and what condition was specified. For example, look onto following code: Test provides several macros that solve this task you need to include additional header file to use them: In some situations, you need to check, does your code throws exception or not. Another task, automated by Boost. Test is testing of output results. Test provides special output class, compatible to std:: You can also create a file with "desired output", and use data from this file to compare against output, produced by your code. In some cases, it could be also useful to get checkpoints, where test case was in normal state. Test provides 2 macros for this task: Test also provides special macros that allows to simplify use of fixtures. There is also additional advantage over direct use of fixtures in your code â€" you have direct access to public and protected members of fixture, for example: This macro accepts fixture name as second parameter, and separate fixture object is created for every test in given test suite. There is also 3rd type of fixtures, supported by Boost. Output of results Usually Boost. Test prints only messages about errors and exceptions, but you can control what will be printed by using different options, described below. There are also compile time options , that allows to control output, for example, threshold level, etc. Test prints results in human-readable format, but it can also output data in XML, so you can feed them into database, or dashboard. There is also macro, that provides explicit printing of data. Execution control Tests are executed by execution monitor, that takes list of registered tests, execute them creating fixtures, if necessary , and count number of failures. By default execution monitor handles all exceptions, including system problems, like wrong memory access. Test provides many run-time options that control behaviour of execution monitor some of these options also have compile-time equivalents. There are two ways to specify run-time configuration option â€" from command line or via setting environment variable. When test program is initializing, execution monitor analyzes command-line options, and excludes from them all options, belonging to its own configuration. Here is list of most important options, that are recognized by test programs, that are using standard execution monitor in parentheses are names of corresponding environment variables: You can use this to see which test is currently executing, together with related information. User can list test names, or use mask. Test display progress indicator during execution of test cases, or not. Description of other options you can find in documentation â€" they can control format of output, which additional details will be shown, etc. Google mocking framework has pretty good documentation, that is available as wiki. You can find tutorial in following document , and find more in cookbook , cheatsheet , and FAQ. I assume, that google mock library is already installed on machine. Google mock follows standard workflow of mocking: There is also a tool, that can generate mock definition from your source code; you create test case that will use your mock class, and inside it you do following: This should be virtual class, so Google mock will able to override methods in it: We create an instance of mocked class called mholder, and will set expectations on it. First expectation is that function GetProperty will be called once with parameter "test", and mocked object should return for this call. The second expectation specifies that SetProperty function will be called with two arguments â€" "test2" and After setting expectation, we create an instance of our TestClass and pass reference to mocked object to it. And last line â€" call of function doCalc, that uses functions from PropHolder class: After that we can run our test program and get results. More information you can find in documentation for Google mock framework, where you can also find many examples of its usage. Additional information There is a lot of additional sources of information â€" books, study courses, articles, etc.

Chapter 7 : A Test Driven Development Tutorial in C#

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.

Download source files - This is an ideal for which I strive. However, I have a tendency to write the unit tests afterwards. Some people learn better by example. This article, rather than going into great length about the principles of test driven development, will walk the reader through the process of building and testing an algorithm by writing the tests first, then changing the method being tested so that it fulfills the tests. The final code and all the unit tests can be found in the accompanying download. This will require NUnit and Visual Studio The use of NUnit in this example is purely incidental, this example could just have easily used the unit test framework fin Visual Studio Team System or any number of other unit test frameworks. The example shows how to build a suite of tests rather than how to use a unit test framework. The reader is expected to understand the basics of using a unit test framework. The specimen problem I once saw a demo of how to create unit tests up front for a simple method. The method took a positive integer and turned it into roman numerals. The rules for this may change depending on the language, so if English is not your only language, you may like to try to repeat this exercise in another language. So, if the integer is 1, the result will be "one". If the integer is 23 the result will be "twenty three" and so on. So, for the result in words is "one hundred and one". In American English it would be "one hundred one". The walk through The algorithm will also be refined through refactoring techniques. Agile development methodologies, especially eXtreme Programming, suggests that you do the simplest thing possible to get the thing working. First, get it to return "one", then "one" or "two" depending on the input, and so on. Once 21 is reached it should become obvious where some refactoring can take place and so on. The final solution will work for 32 bit integers only. Getting Started Visual Studio has some features that can help with writing the tests first. A test can be written that calls into the class under test and the smart tags will prompt you with an offer to create the message stub for you. The stub looks like this: NumberToEnglish 1 ; Assert. NUnit reports the error like this: The method or operation is not implemented. The next thing to do is to ensure that the code satisfies the demands of the unit test. Agile methodologies, such as XP, say that only the simplest change should be made to satisfy the current requirements. In that case the method being tested will be changed to look like this: Test "two" Since the overall requirement is that any integer should be translatable into words, the next test should test that 2 can be translated. The test looks like this: NumberToEnglish 2 ; Assert. Expected the result to be "two" String lengths are both 3. Strings differ at index 0. Test "three" to "twenty" A third test can now be written. It tests for an input of 3 and an expected return of "three". Naturally, at this point, the test fails. The code is updated again and now looks like this: The code will eventually look like this: After the tests for 21 and 22 have been written, the code is refactored to look like this: Test "thirty" to "thirty nine" 30 is a different story. The test will fail like this: Expected the result to be "thirty" String lengths differ. Strings differ at index 1. Expected the result to be "thirty one" String lengths differ. Strings differ at index 6. Expected the result to be "forty" String lengths differ. Of course, the pattern could have been quite easily predicted, but since this code is being built by the simplest change only rule, the pattern has to emerge before it can be acted upon. The pattern repeats itself until it gets to By this point the public method looks like this: The failure message is: Expected the result to be "one hundred" String lengths differ. That test fails like this: Expected the result to be "one hundred and one" String lengths differ. In Visual Studio , it is very easy to highlight some code and extract it into a new method, thus allowing it to be reused by being called from multiple places. Now the public method looks like the following and all previously successful tests continue to be successful. Because it would take too long to write all those tests, it is possible to write just the edge cases and one or two samples from the middle of the range. That should give enough confidence to continue onwards. In this case, the tests are for , , and The code is then re-written to support those tests: For that reason, the details of the intermediate steps are skipped until all positive integers can be written. If, however, you wish to read about these intermediate steps then you can

read the unabridged version of this article on my website. The limitations of an integer Int32 mean that this section reaches the upper limits of Unless an Int64 is used there is no continuation to the trillion range. Final stages To this point, all positive integers are successfully being translated from an integer into a string of words. At this point, through code reuse, it should be a fairly simple matter to refactor the code to work with negative numbers and zero. Zero is easy enough. The unit test is put in place: NumberToEnglish 0 ; Assert. The public method is changed so that it does a quick check at the start: Next is to permit negative numbers. NumberToEnglish -1 ; Assert. Abs number ; return string. But what about that final edge case? The test is written but it fails. The reason is that Math. So, as this is a one off case, the easiest solution is to put in a special case into the public method: The tests continually prove that the developer is on the right path. As the code is built the constant rerunning of existing tests prove that any new enhancements do not break existing code. If, for any reason, it is later found that the code contains a bug, a test can easily be created that exercises that incorrect code and a fix is produced. The full final code is available in the associated download along with a set of NUnit tests. This is the abridged version of the article. For the full article visit here Version 1.

Chapter 8 : Test-driven development and unit testing with examples in C++

Test-driven development (TDD) is a software development technique that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

Over the course of the last two years, our company has been promoting, conducting training, and consulting in XP eXtreme Programming. We believe that XP is the best technique for delivering high-quality software because of its emphasis on communication, simplicity, testing, and rapid feedback. XP achieves its results through implementation of about a dozen proven practices. The TDD technique can be, and should be, applied in any software development environment. The example presented in this article demonstrates how this works. A recent client of ours wanted to do TDD. In fact, they wanted to do all of the XP practices. The problem was that they were a C shop, and they were going to be using C for the foreseeable future, so we had to roll up our sleeves and figure out how to get the benefits of TDD in C. In TDD, one module is developed at a time. To properly test a given module, it must be isolated from most other modules with which it interacts. These interfaces also provide a testing environment for the module. We need these same capabilities in C. In these environments, we would have to stub out the function calls that provided access to the missing hardware. This stubbing technique is just what we need to get the benefits that polymorphism provides in languages that support it. In fact, stubbing is polymorphism, and we refer to it as link-time polymorphism. We then write enough code to get everything to compile, but purposely code it so that our test fails. We then write code to make the test pass. Each of these steps is critical to the TDD process. We strive to make our code have the best possible design for the functionality we have implemented so far, but for nothing more than that functionality. Only after we have refactored toward these goals do we go on to writing another test. We repeat this cycle of test-code-refactor in rapid succession throughout our development day. How often have you spent an hour writing code, only to spend the entire day getting it to compile and debugged? Instead of programming a possibly large piece of functionality, TDD dictates that we work on a very small problem, solve it, and then move on to the next very small problem. After all, every large complex problem is just a bunch of little problems. Solving small problems, one after another, is sustainable, enjoyable, and highly productive. The key to these series of successes is rapid feedback. We run the compiler just as soon as we have enough code that stands a chance of compiling. We also run our tests as often as possible. Our rule is that no more than 10 minutes may go by without having all of our tests running. Five minutes is better. One minute is better still. This obsession with running our tests frequently forces us to partition our work into a series of very small problems. Have you ever tried to test code after it was written? By writing our tests first, we guarantee that there are tests and that the system is designed to be testable. Since adopting TDD, we have almost never used a debugger on the code we have written. Our response to the supposed need for debugging is to simply throw away the last few minutes worth of work. We then try again, taking even smaller steps. A sales clerk passes items over a scanner that reads the barcode. A display monitor shows information on each item as it is scanned and also tracks the total for each sale of one or more items to a given shopper. Our First User Story[1] The total price of all items in the sale is available. We have the concept of a Sale, and we know that it has a total. We are essentially coding our requirements document so that it executes as a suite of unit tests! This is an important concept when practicing XP. We do not produce thick paper requirements or design documents that never seem to match the working code. Yet documenting requirements and design is essential to software development, so we do it in code instead. Writing tests in code is essential for a second reason: Ultimately it becomes impossible to keep up with executing all tests against even a modest body of code. In TDD, a testing framework is an essential tool. It not only provides us with a means for automating the tests, but it is also an integral part of the minute-to-minute development process. Such a tool need not be an expensive investment. The tool we use is CppUnitLIte , which is a freely available macro-based tool.. For our user story, we first create the file SaleTest. CppUnitLite requires a little boilerplate code: The basic idea is that the runAllTests method is responsible for extracting and executing all of the tests we have defined. Now that the boilerplate

code is in place, we can write our first test. Anything defined with this macro is automatically picked up by runAllTests. The macro takes two parameters: It represents the state of a new Sale before any Items are added to it. It seems reasonable that before anything is purchased the total of the Sale should be zero, so we express this in our test. We compile anyway as our first level of feedback. If SaleTest compiles, we know we have a serious configuration problem. Our purpose is to get to compilation so that we can link and run our tests. To run the test, we simply execute SaleTest: This is as expected: We want to see a failure, though, as part of our feedback-driven, test-first design cycle. A reminder of the cycle: Write a test for nonexistent code. Expect that the test will fail. Write just enough code to make the test pass. Every once in a while, the test that we expect to fail passes. This should be a shock! For our Sale application, we can make the test pass by simply returning 0 from GetTotal. We have a failing test case, and we want to get it to pass as quickly as possible. But the fact that we provided just enough code to make the current test pass means that we will have to write more tests. These tests will fail, proving that a more complicated algorithm is necessary. We increase our test coverage this way at a steady pace. More tests are a good thing. There were no test failures. Our first test is complete. We know the state a Sale should be in when nothing has been done to it. Now we want the capability to sell at least a single item, so we write the test sellOneItem. For our test, we make up an item whose barcode is "a" and cost is cents. We devise the function BuyItem to accomplish this. Buying item "a" should increase the total of the Sale to We compile, expecting failure: Time to add the code to make this test pass. At this point, the simplest thing that will work is to introduce a variable to track the total. We declare it in Sale. This gives the variable visibility only at file scope. After each new test passes, we always pause and look over the code to be sure that it communicates. Since the tests will evolve and be maintained along with the code that is being tested, we want them to serve as excellent documentation on how to manipulate the Sale functions. This obsession we have with clearly communicating code is far better than placing comments in the code, which quickly become outdated. Introducing local variables in the test yields: Note that we already have a need to reuse the constant declarations of milkBarcode and milkPrice, so we move them out of sellOneItem and declare them as static. Building the correct production code in Sale. Instead of just setting the total to the price of milk, we add the price into the total:

Chapter 9 : Test Driven Development in C/C++ | Dr Dobb's

test-driven development kent beck test driven driven development writing tests using tdd extreme programming unit testing testing framework design patterns write test read this book years ago unit tests software development easy to follow simple example easy to understand well written development process.

A graphical representation of the test-driven development lifecycle The following sequence is based on the book Test-Driven Development by Example: Add a test In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. 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: Run all tests and see if the new test fails This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. Write the 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. The programmer must not write code that is beyond the functionality that the test checks. Run 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. Refactor 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 recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code. 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 codeâ€”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, [4] 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. By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods. 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. 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. This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Keep the unit small[edit] 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 â€" When test failures are detected, having smaller units aids in tracking down errors. Self-documenting tests â€" Small test cases are easier to read and to understand. With ATDD, the development team now has a specific target to satisfy â€" the acceptance tests â€" which keeps them continuously focused on what the customer really wants from each user story. Test structure[edit] Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and 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. This step is usually very simple. Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT. Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one. The purpose of Wikipedia is to present facts, not to train. Please help improve this article either by rewriting the how-to content or by moving it to Wikiversity , Wikibooks or Wikivoyage. May This section is in a list format that may be better presented using prose. You can help by converting this section to prose, if appropriate. Editing help is available. May 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 oracle 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 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. It may be helpful to review this section during your discussion. 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 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. An oracle 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. Benefits[edit] A study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive. 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. 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. 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: