Test-Driven Development

Aziz Nanthaamornphong

Department of Computer Science
University of Alabama

Test-Driven Development: Introduction & Background

- Basic idea:
  - Write automated tests
  - Prior to developing functional code
  - Small rapid iterations
- Part of the *agile* software development approach
  - Short iterations
  - Little up-front design
  - Lightweight documentation
  - Refactoring
  - Pair programming

Test-Driven Development: Overview

- Focus on unit tests
  - Traditionally written after code is completed
  - In TDD tests are written before code
- Often require
  - Test drivers
  - Test stubs
- Can be automated or manual
- Can be performed by developers or testers

Test-Driven Development: Motivation

- Programming practice that instructs developers to:
  - Write code only if a test has failed
  - Eliminate duplication

- Test-Driven Development
  - Leads to analysis, design and programming decisions
  - Writing a test is one of the first steps in deciding what a program should do (analysis)

Test-Driven Development: Definition

- From the Agile Alliance

Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

Test-Driven Development: Additional Thoughts

- Refactoring
  - Additional step after coding
  - Code becomes complex
  - Tests still pass, but code is simpler
- Not a software development methodology
- Provides automated test
  - Not thrown away
  - Become part of the development process
  - If a change breaks something that worked before, developer knows immediately

Test-Driven Development: Refactorings

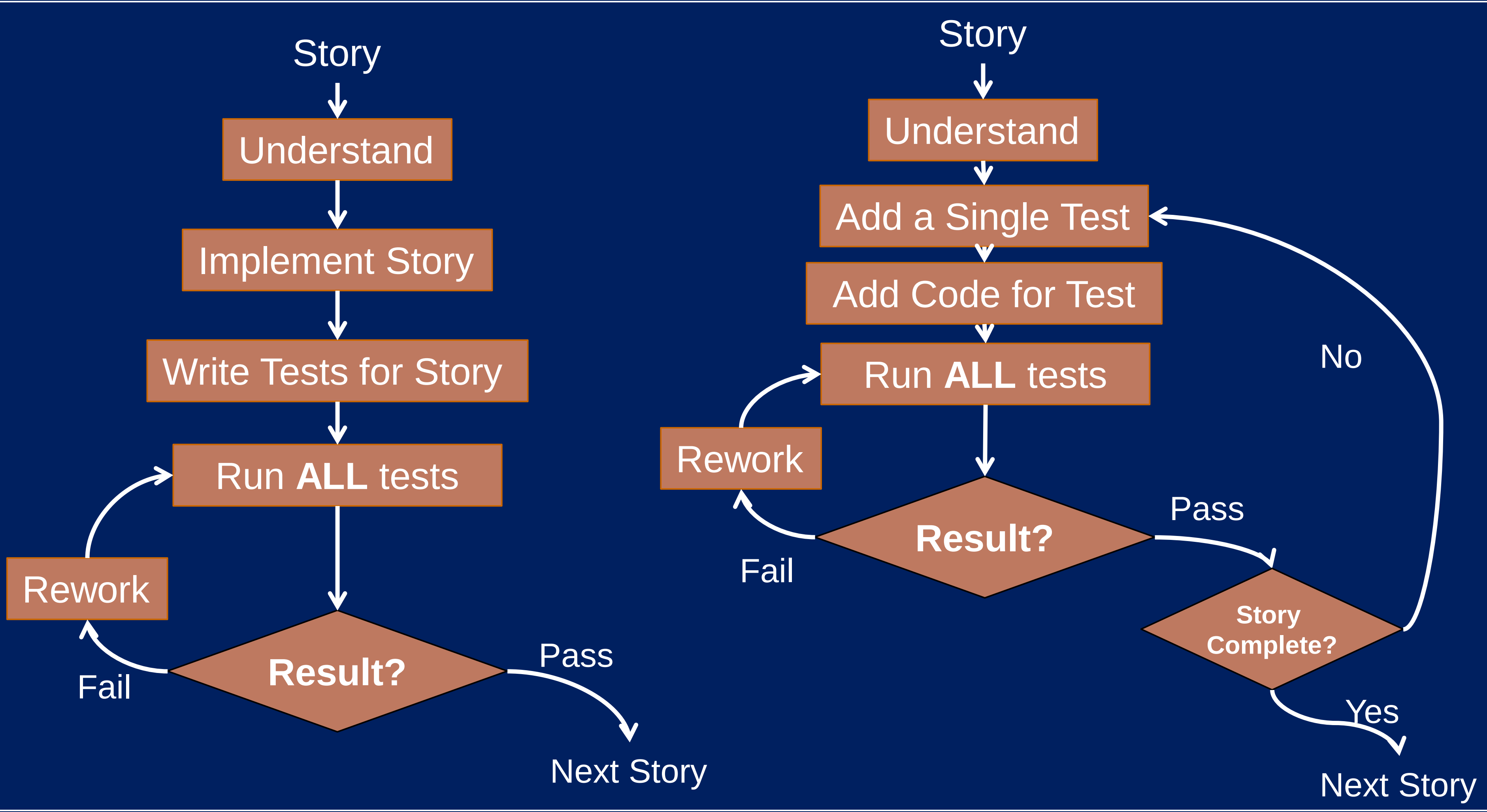
- Example of refactorings

Techniques	Description
Extract Method	If a code fragment can be grouped together, turn it into a method whose name explains the purpose of the method and replace the fragment with a call to the new method.
Pull Up Method	Methods with duplicated code in two subclasses of a common ancestor X can be refactored as follows: extract a method in both classes and put it into the superclass X.
Push Down Method	Behavior on a superclass is relevant only for the subclass. Push Down Method is the opposite of Pull Up Method.
Parameterization	Several methods do similar things but with different values contained in the method body, one method that uses a parameter for the different values can be created.
Extract Superclass	You have two classes with similar features. Create a superclass and move the common features to the superclass.

Test-Driven Development: Additional refactorings

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static
- Convert Static to Dynamic
- Form Template Method
- Hide Delegate
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass

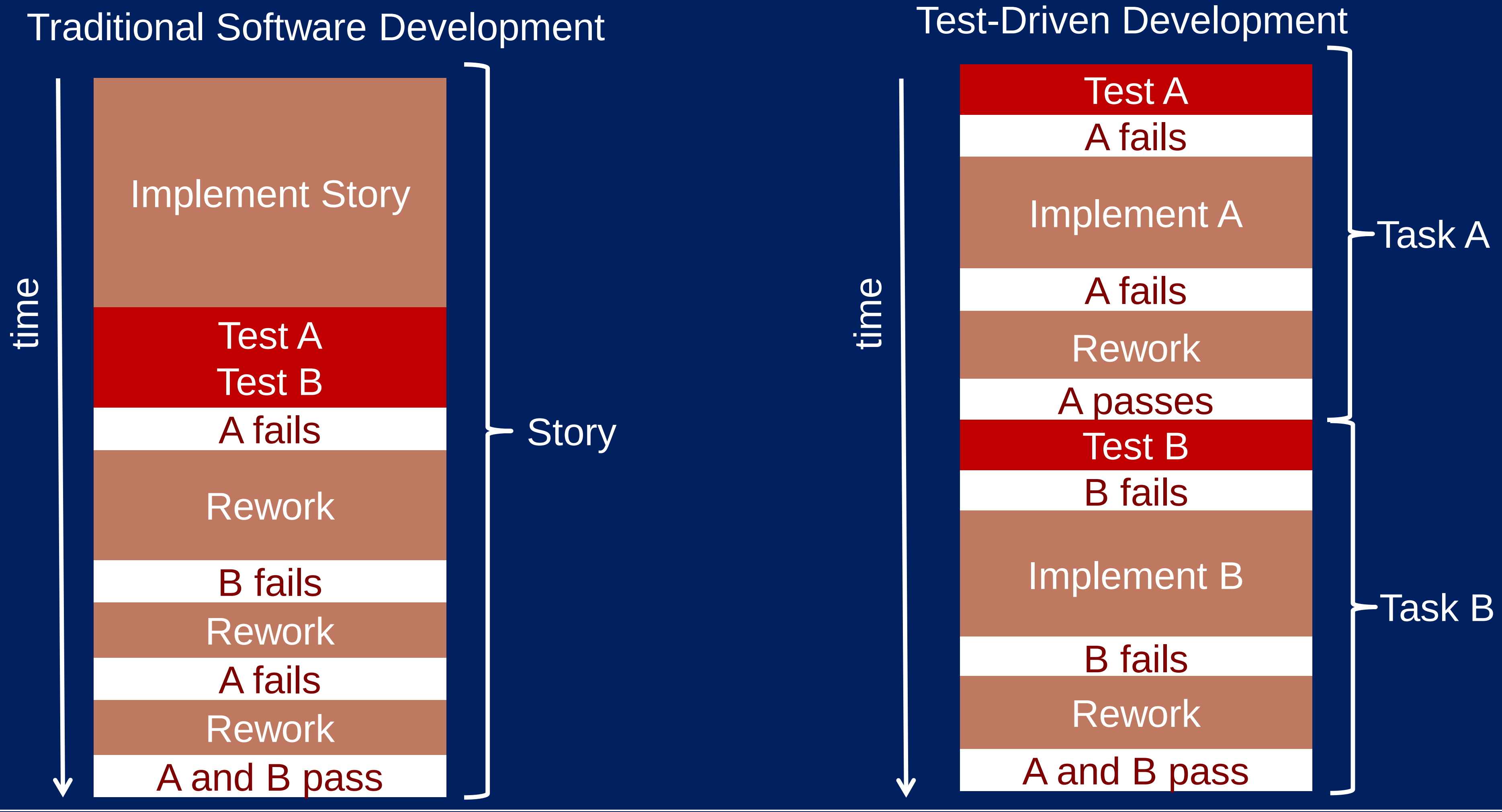
Test-Driven Development: Traditional vs. TDD



Traditional Software Development

Test-Driven Development

Test-Driven Development: Example Story



Test-Driven Development: Automated Testing

- TDD assumes the presence of an automated testing framework
- Test Harnesses
- xUnit
  - Lets users write tests to initialize, execute, and make assertions about code being tested
  - Tests can serve as documentation

Test-Driven Development: Challenges to Adoption

- Requires discipline by programmers
- TDD is misunderstood – many think it addresses only testing and not design
- Does not fit every situation

Test-Driven Development:
Example : Object-oriented programming

- Design a system to perform financial transactions with money that may be in different currencies

- For example –
  - If the exchange rate from Swiss Francs to US Dollars is 2 to 1, then we can calculate
    - 5 USD + 10 CHF = 10 USD
    - 5 USD + 10 CHF = 20 CHF

Example:
Starting Point

- How do we start?
- Write a list of things we want to test
- List can be any format, just keep it simple
- Example
  - 5 USD + 10 CHF = 10 USD if rate is 2:1
  - 5 USD * 2 = 10 USD

Example:
First Test

- Second item is easier, start there
  - 5 USD * 2 = $10

- First write a test case

Example:
Test Case Discussion

- What benefits does this provide?

- Target class plus some of its interface
  - Design the interface of the Dollar class by thinking about how we would want to use it

- Testable assertion about the state of the Dollar class after a particular sequence of operations

Example:
Next Step

- Test case revealed some issues with the Dollar class that must be cleaned up
  - The amount is represented as an integer, making it difficult to handle things like 1.5 USD; how do we handle rounding of fractions?
  - Dollar.amount is public; violates encapsulation
  - Side effects?
    - We first declared our variable as “five”, but after we performed the multiplication, it equals “ten”
- Update Test List

Example:
First Version of Dollar Class

- Our test will not compile
  - What compile errors will we encounter?
  - Fix compile errors
  - Create skeleton of Dollar class

- Now our test compiles and fails

Example:
Too Slow?

- Process
  - Do the simplest thing to get the test to compile
  - Now do the simplest thing to get the test to pass

- Is this process too slow?
  - Yes
    - As you get familiar with the TDD lifecycle you will gain confidence and make bigger steps
  - No
    - Small simple steps help avoid mistakes
    - Beginning programmers try to code too much before compiling
    - Spend the rest of their time debugging!

Example:
Make the Test Case Pass

- First do simplest thing to get test case to pass
- The test now passes
- Now, we need to refactor to remove duplication
  - Where is the duplication?
  - Hint: Its between the Dollar class and the test case

Example: Refactoring

- To remove the duplication of the test data and the hard-wired code of the times method, we think the following
  - *I am trying to get at 10 at the end of my test case. I've been given a 5 in the constructor and a 2 was passed as a parameter to the times method*
- Let's connect things

Example:
First Version of Dollar Class

- Refactor Dollar class
- Now our test compiles and passes, and we didn’t have to cheat!
- One TDD loop complete
  - Update testing list
  - Move on to next item

Example:
Second Loop

- Address the “Dollar Side-Effects” item

- Next test case
  - When we called the times operation on our variable, “five” was pointing at an object whose amount equaled “ten”; not good
    - The times operation had a side effect which was to change the value of a previously created “value object”
    - This doesn’t make sense, you can’t change a $5 bill into a $10 bill; the $5 bill remains the same throughout transactions
  - Rewrite test case

Example:
Test Fails

- Won’t compile
  - How do we fix this problem?

- Change the signature of the times method; previously it returned void and now it needs to return Dollar

- The test compiles but still fails – progress
  - How do we fix this problem?

Example:
Test Passes

- To make the test pass, we need to return a new Dollar object whose amount equals the result of the multiplication
- Test passes, cross “Dollar Side-Effects” off of the testing list.
- No need to refactor here
- Move on to next test item

Example 2 : procedural programming

- Write a function to calculate the sum of integers from min to max inclusive. If the result equals zero, then calculate the multiplication of min and max values

Expected function: *sum(int min, int max)*

Example 2:
Write a test

```
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    assert(sum(3,7)==25);
   assert(sum(-2,2)==-4);
   assert(sum(-4,2)==-9);
    cout << “Congratulations!!” << endl;
}
```

Compiler will report “error” because “sum()” is not implemented

Example 2:
Make the test pass

```
int sum(int min, int max)
{
    int sum = 0;
    for(int i=min; i<=max; i++)
    {
        sum += i;
    }
    if (sum == 0){ sum = min * max;}
    return sum;
}
```

Compiler will not blame

Test pass! executing the program with this output "Congratulations!!"

Example 2: Refactoring

```
int sum(int min, int max)
{
     int sum = 0;
     for(int i=min; i<=max; i++)
     {
          sum += i;
     }
     if (sum == 0){ sum = min * max;}
     return sum;
}
```

Extract Method

```
int sum(int min, int max)
{
     int sum = 0;
     for(int i=min; i<=max; i++)
     {
          sum += i;
     }
     if(sum == 0){
        sum = multiply(min, max)
     }
     return sum;
}
int multiply(int min, max){
     int multiply = min * max;
     return multiply;
}
```

Make sure the test still passes!!

References

- Jansen, D. and Saiedian, H. “Test-Driven Development: Concepts, Taxonomy, and Future Direction.” *Computer*. Sept. 2005. p. 43-50

- Erdogmus, H., Morisio, M., and Torchiano, M. “On the Effectiveness of the Test-First Approach to Programming.” *IEEE Transactions on Software Engineering*. 31(5): 226-237. March 2006.

- Example taken from
  - Kenneth Anderson, Univ. of Colorado, Boulder