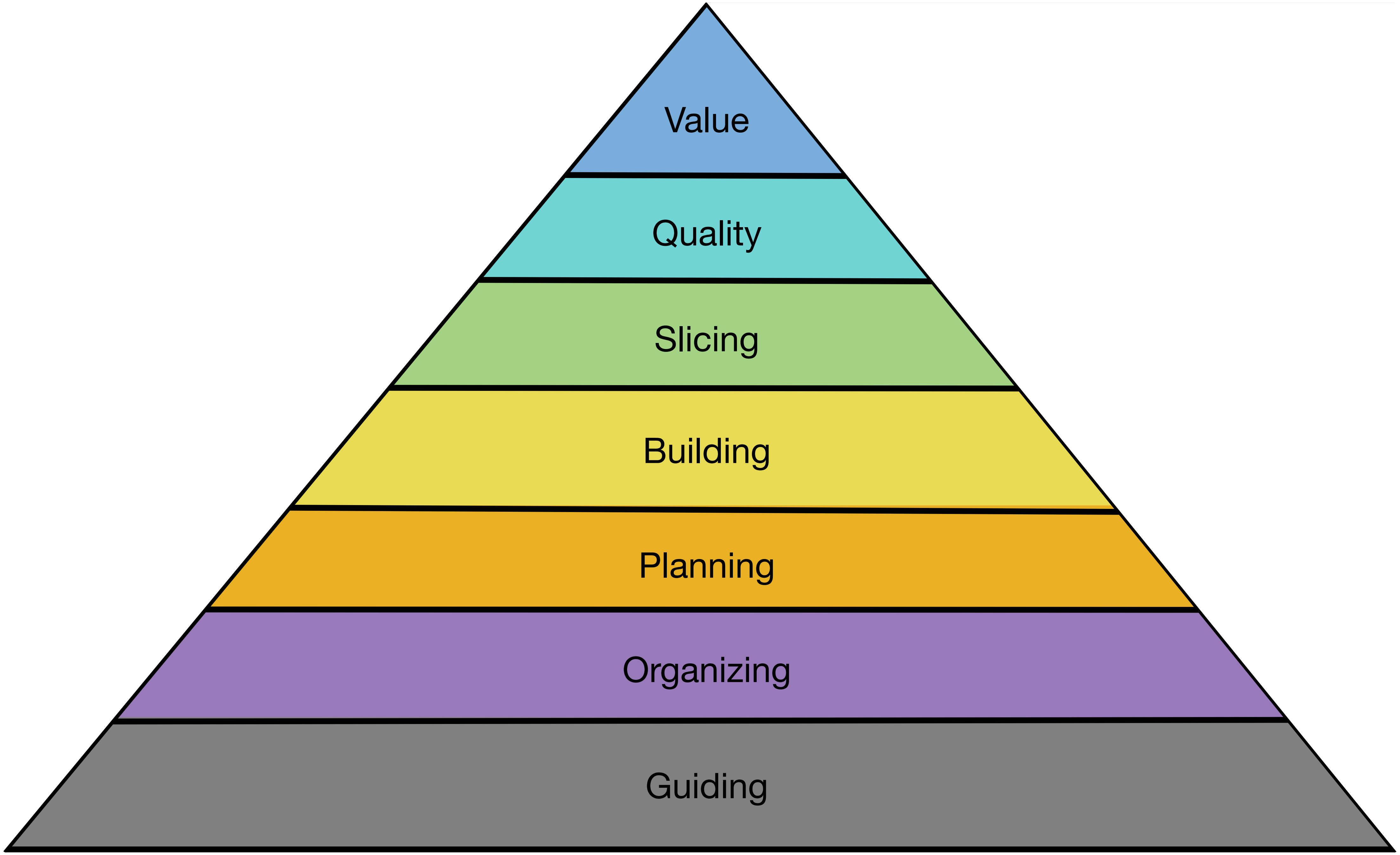
The Nature of Software Development, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 24 — 11/12/2015

Goals

- Cover the material presented in Part One of our Agile textbook
  - Chapters 6 to 9
- Building Software Feature by Feature
- Slicing Features (and then growing them over time)
- Ensuring Code Quality
  - Test Driven Design
  - Refactoring

Reminder: The Natural Way in a Nutshell



Getting the Software Built

- Chapters 6, 7, and 8 talk about getting a software system implemented
  - feature by feature
  - in a sustainable fashion
  - while ensuring code quality
  - (Chapter 9 is just a single page and just summarizes the lessons learned)
- These chapters fulfill Jeffries promise that the natural way is
  - “simple but not easy”
- The issues discussed are tricky and require thought and practice to get right

Building

- Is it possible to build a system feature by feature?
  - Yes! Jeffries asserts that teams have been doing this successfully for decades
- To do it, our work has to occur in short cycles (1-3 weeks)
  - In that cycle, we
    - define the next features to build and we identify how they are tested
    - build the features and then verify that they pass our tests
  - In that cycle we
    - engage in a complete product development life cycle
      - concept, requirements, design, implement, test, deploy

Short Iterations are Hard

- It will take some work to be able to work in this manner
  - When you start down this path, it will be hard; you won't be good at it
  - It is important not to get discouraged and to keep trying
    - Work to understand how to set scope
    - and how to build a complete, deployable feature (user story)
- At the beginning, your meta-goal is to learn
  - learn how much can be done in a short iteration
  - how to test your code properly
  - how to write user stories
  - how to inject quality in the process over time

Keep Working to Identify “What We Want”

- As we progress, we need to constantly engage the customer to ensure we are building valuable features
    - We have to gather information so that ambigious, high level statements get turned into small, easy-to-understand user stories
        - what MUST the system do? vs. what is “nice to have”
- Always work on the highest priority features
    - As new user stories get added and our vision of the system evolves make sure the user is reviewing the priorities of existing stories relative to the new ones
- Be very clear about what “done” means
    - Sometimes there is “done” and then there is “done done”. Be clear!

The Hard Part: Eliminate Test and Fix

- Lots of projects add a “test and fix” cycle at the end of a project
    - Build, build, build, and then test, test, test, test, test, …
- In Agile, we must attempt to avoid this at all costs
    - we have to do what we can to avoid injecting defects into what we build
        - if we build low-quality software then we kill our ability to deliver new features as we spend time fixing old features
            - ones we thought were “done” but weren’t
- For this style of development to work, the software needs to be nearly defect free at the end of an iteration and for each release
    - How do we do that? Stay tuned (Foreshadowing: it’s tough!)

Slicing: Build Features and Foundation in Parallel

- In Chapter 7, Jeffries acknowledges the difficulty of building a feature
    - You need a complete vertical stack for the feature to work
        - UX, app logic, services, data stores
    - And for that to work you need a foundation upon which the features rest
- There are several approaches to do this
    - Build a strong foundation first; then work on features
    - Build lots of features with very little unifying foundation; integrate later
    - Build features/foundation in parallel
- The first two will slow you down; the third is what Jeffries recommends
    - To make this work, you need to “version” your features
        - build a simple version of a feature first with just enough foundation
            - grow both over time

Quality: Bug-Free and Well Designed

- In Chapter 8, Jeffries talks about injecting quality into the agile life cycle
    - At a high-level, his recommendations are
        - test everything
            - have business tests (written by the customer) that confirm the features are working
            - have (lots of) developer tests that are run every day
                - unit tests and integration tests
                - using test-driven development
        - always improve/maintain your design via refactoring

Wrapping Up The First Half of the Book

- We’ve now encountered the various aspects that make up Ron Jeffries’s natural way of develop software
  - Our primary goal is to create value for our customer/user
    - by delivering a working software system with valuable features
      - features that solve a problems they currently have
  - We do this by guiding, organizing, planning, and building the system in an iterative, incremental fashion feature by feature such that
    - it’s easy to measure progress
    - get feedback
    - achieve high quality

Deep Dives

- Let’s continue by taking a look at
  - Refactoring
    - and
  - Test-Driven Development

What is Refactoring

- Refactoring is the process of changing a software system such that
    - the external behavior of the system does not change
        - e.g. functional requirements are maintained
    - but the internal structure of the system is improved
- This is sometimes called
    - “Improving the design after it has been written”
- It is known in Agile circles as helping to pay down “technical debt”
    - Technical debt is defined as the continuous accumulation of shortcuts, hacks, duplication, and other sins that we regularly commit against our code base in the name of speed and schedule.

(Very) Simple Example

- Consolidate Duplicate Conditional Fragments (page 243); This

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send()
} else {
    total = price * 0.98;
    send()
}
```

- becomes this

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

(Another) Simple Example

- Replace Magic Number with Symbolic Constant

```
double potentialEnergy(double mass, double height) {
      return mass * 9.81 * height;
}
```

- becomes this

```
double potentialEnergy(double mass, double height) {
      return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

In this way, refactoring formalizes good programming practices

Refactoring is thus Dangerous!

- Manager’s point-of-view
  - If my programmers spend time “cleaning up the code” then that’s less time implementing required functionality (and my schedule is slipping as it is!)
- To address this concern
  - Refactoring needs to be **systematic**, **incremental**, and **safe**

Refactoring is Useful Too

- The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time and, as a program's requirements change, the design may need to change
    - refactoring provides techniques for evolving the design in small incremental steps
- Benefits
    - Often code size is reduced after a refactoring
    - Confusing structures are transformed into simpler structures
        - which are easier to maintain and understand

A “cookbook” can be useful

- Refactoring: Improving the Design of Existing Code
  - by Martin Fowler (and Kent Beck, John Brant, William Opdyke, and Don Roberts)
- Similar to the Gang of Four’s Design Patterns
  - Provides “refactoring patterns”

Principles in Refactoring

- Fowler's definition
  - Refactoring (noun)
    - a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
  - Refactoring (verb)
    - to restructure software by applying a series of refactorings without changing its observable behavior

Principles, continued

- The purpose of refactoring is
  - to make software easier to understand and modify
- contrast this with performance optimization
  - again functionality is not changed, only internal structure;
  - however performance optimizations often involve making code harder to understand (but faster!)

Principles, continued

- When you systematically apply refactoring, you wear two hats
  - adding function
    - **functionality is added** to the system **without** spending any time **cleaning the code**
  - refactoring
    - **no functionality is added**, but the code is **cleaned up**, made easier to understand and modify, and sometimes is reduced in size

Principles, continued

- How do you make refactoring safe?
  - First, use refactoring “patterns”
    - Fowler’s book assigns “names” to refactorings for you to memorize and use
  - Second, test constantly!
    - This ties into the agile design paradigm
      - you write tests **before** you write code
      - after you refactor, you run the tests and check that they all pass
        - if a test fails, the refactoring broke something **but you know about it right away** and can fix the problem before you move on

Why should you refactor?

- Refactoring **improves the design of software**
  - without refactoring, a design will "decay" as people make changes to a software system
- Refactoring **makes software easier to understand**
  - because structure is improved, duplicated code is eliminated, etc.
- Refactoring **helps you find bugs**
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- Refactoring **helps you program faster**
  - because a good design enables progress

When should you refactor?

- The Rule of Three
  - Three “strikes” and you refactor
    - refers to duplication of code
- Refactor when you add functionality
  - do it before you add the new function to make it easier to add the function
  - or do it after to clean up the code after the function is added
- Refactor when you need to fix a bug
- Refactor as you do a code review

Problems with Refactoring

- Databases
    - Business applications are often tightly coupled to underlying databases
        - code is easy to change; databases are not
    - Changing Interfaces (!!)
        - Some refactorings **require that interfaces be changed**
            - if you own all the calling code, no problem
            - if not, the interface is “published” and can’t change
    - Major design changes **cannot** be accomplished via refactoring
        - This is why agile design says that software devs. need courage!

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?
    - Fowler uses an olfactory analogy (attributed to Kent Beck)
    - Look for **“Bad Smells”** in your code
        - A very valuable chapter in Fowler’s book
        - It presents examples of “bad smells” and then suggests refactoring techniques to apply

Bad Smells in Code

- **Duplicated Code**
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long Method**
  - long methods are more difficult to understand
    - performance concerns with respect to lots of short methods are largely obsolete
- **Comments** (!)
  - Comments are sometimes used to hide bad code
    - "…comments often are used as a deodorant" (!)

Bad Smells in Code

- **Shotgun Surgery**
  - a change requires lots of little changes in a lot of different classes
- **Feature Envy**
  - A method requires lots of information from some other class
    - move it closer!
- **Long Parameter List**
  - hard to understand, can become inconsistent if the same parameter chain is being passed from method to method

Bad Smells in Code

- **Primitive Obsession**
  - characterized by a reluctance to use classes instead of primitive data types
- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
- **Speculative Generality**
  - “Oh I think we need the ability to do this kind of thing someday”

The Catalog

- The refactoring book has 72 refactoring patterns!
    - I’m only going to cover a few of the more common ones, including
        - Extract Method
        - Replace Temp with Query
        - Separate Query from Modifier
        - Introduce Parameter Object
        - Encapsulate Collection

Extract Method

- You have a code fragment that can be grouped together
    - Turn the fragment into a method whose name explains the purpose of the fragment
- Example, next slide

Extract Method, continued

```
void printOwing(double amount) {
    printBanner()
    //print details
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + amount);
}

=============================================

void printOwing(double amount) {
    printBanner()
    printDetails(amount)
}

void printDetails(double amount) {
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + amount);
}
```

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression
  - Extract the expression into a method;
  - Replace all references to the temp with the expression.
  - The new method can then be used in other methods
- Example, next slide

Replace Temp with Query, continued

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;

==============================

if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
…
double basePrice() {
    return _quantity * _itemPrice;
}
```

Separate Query from Modifier

- Sometimes you will encounter code that does something like this
  - **getTotalOutstandingAndSetReadyForSummaries()**
- It is a query method but it is also changing the state of the object being called
  - This change is known as a “side effect” because it’s not the primary purpose of the method
- It is generally accepted practice that queries should not have side effects so this refactoring says to split methods like this into:
  - **getTotalOutstanding()**
  - **setReadyForSummaries()**

Introduce Parameter Object

- You have a group of parameters that go naturally together
  - Stick them in an object and pass the object
- Imagine methods like
  - **amountInvoicedIn(start: Date; end: Date);**
  - **amountOverdueIn(start: Date; end: Date);**
- This refactoring says replace them with something like
  - **amountInvoicedIn(dateRange: DateRange)**
- The new class starts out as a data holder but will likely attract methods to it

Encapsulate Collection

- A method returns a collection
  - Make it return a read-only version of the collection and provide add/remove methods
- Student class with
  - **getCourses(): Map;**
  - **setCourses(courses: Map);**
- Change to
  - **getCourses(): ReadOnlyList**
  - **addCourse(c : Course)**
  - **removeCourse(c : Course)**

Changing the externally visible collection, too, is a good idea to protect clients from depending on the internals of the Student class

Summary for Refactoring

- Refactoring is a useful technique for making non-functional changes to a software system that result in
    - **better code structures**
        - Example: There’s a book out there called “Refactoring to Patterns”
    - **less code**
        - Many refactorings are triggered via the discovery of duplicated code
            - The refactorings then show you how to eliminate the duplication
- **Bad Smells**
    - Useful analogy for discovering places in a system “ripe” for refactoring

Test-Driven Development (I)

- The idea is simple
  - No *production* code is written **except to make a failing test pass**
- Implication
  - You have to write test cases **before** you write code
- Note: use of the word "production"
  - which refers to code that is going to be deployed to and used by real users
- It does not say: "No code is written except…"

Test-Driven Development (II)

- This means that when you first write a test case, you may be testing code that does not exist
    - And since that means the test case will not compile, obviously the test case “fails”
        - After you write the skeleton code for the objects referenced in the test case, it will now compile, but also may not pass
    - So, then you write the simplest code that will make the test case pass

Example (I)

- Consider writing a program to score the game of bowling
- You might start with the following test

```
public class TestGame extends TestCase {
   public void testOneThrow() {
      Game g = new Game();
      g.addThrow(5);
      assertEquals(5, g.getScore());
   }
}
```

- When you compile this program, the test “fails” because the Game class does not yet exist. But:
  - You have defined two methods on the class that you want to use
  - You are designing this class from a client’s perspective

Example (II)

- You would now write the Game class

```
public class Game {
  public void addThrow(int pins) {
  }
  public int getScore() {
    return 0;
  }
}
```

- The code now compiles but the test will still fail: getScore() returns 0 not 5
  - In Test-Driven Design, Beck recommends taking small, simple steps
  - So, we get the test case to compile before we get it to pass

Example (III)

- Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```
public class Game {
    public void addThrow(int pins) {
    }
    public int getScore() {
        return 5;
    }
}
```

- The test case now passes!

Example (IV)

- But, this code is not very useful!
- Lets add a new test case to enable progress

```
public class TestGame extends TestCase {
     public void testOneThrow() {
        Game g = new Game();
        g.addThrow(5);
        assertEquals(5, g.getScore());
     }
     public void testTwoThrows() {
        Game g = new Game()
        g.addThrow(5)
        g.addThrow(4)
        assertEquals(9, g.getScore());
     }
  }
```

- The first test passes, but the second case fails (since 9 ≠ 5)
    - This code is written using JUnit; it uses reflection to invoke tests automatically

Example (V)

- We have duplication of information between the first test and the Game class
  - In particular, the number 5 appears in both places
  - This duplication occurred because we were writing the simplest code to make the test pass
  - Now, in the presence of the second test case, this duplication does more harm than good
  - So, we must now refactor the code to remove this duplication

Example (VI)

```
public class Game {

  private int score = 0;

  public void addThrow(int pins) {

    score += pins;

  }

  public int getScore() {

    return score;

  }

}
```

Both tests now pass. Progress!

Example (VII)

- But now, to make additional progress, we add another test case to the TestGame class

```
…
  public void testSimpleSpare() {
    Game g = new Game()
    g.addThrow(3); g.addThrow(7); g.addThrow(3);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(16, g.getScore());
  }
 …
```

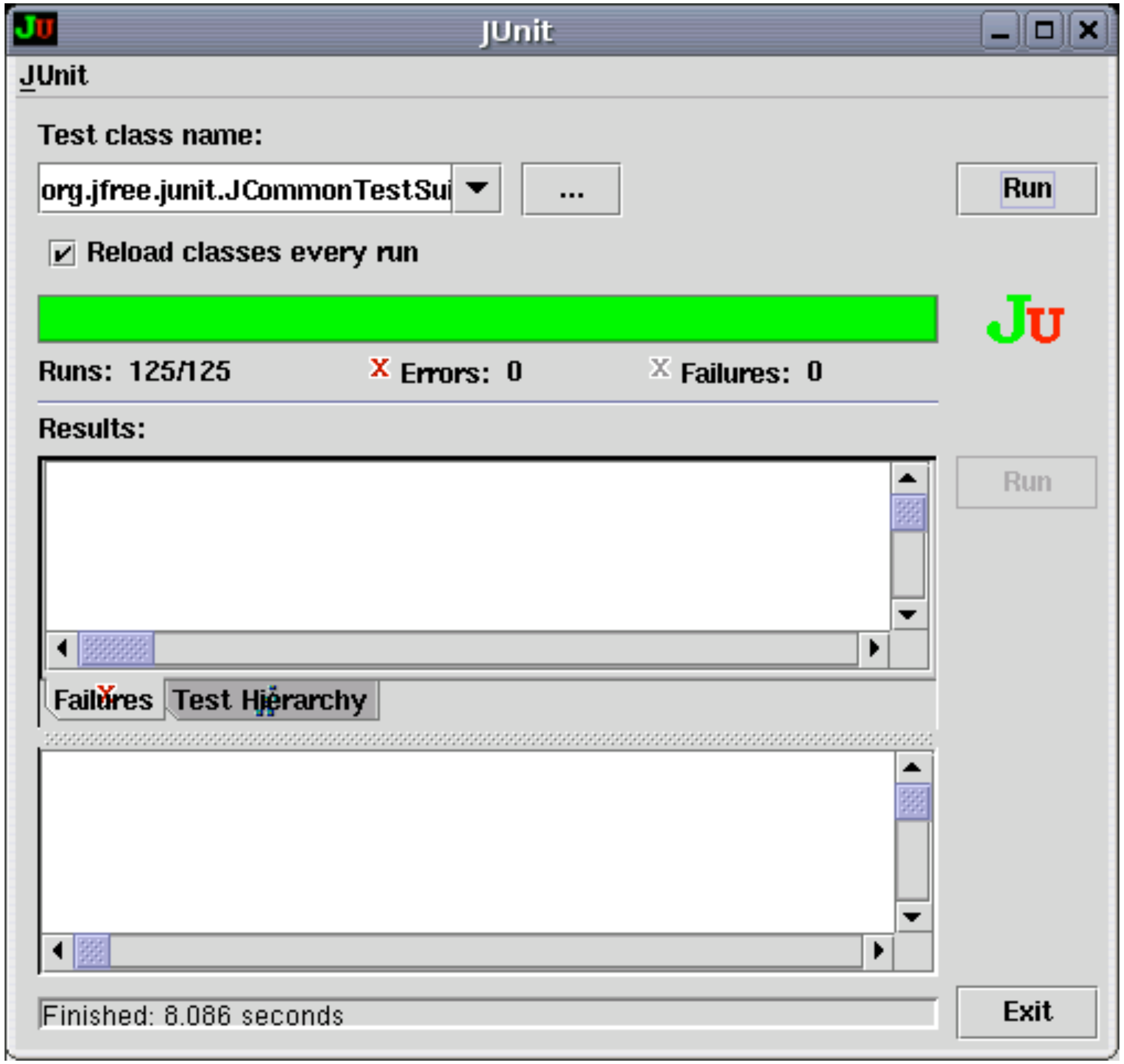
- We’re back to the code not compiling due to scoreForFrame()
  - We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

TDD Life Cycle

- The life cycle of test-driven development is
  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a simple change
  - Run all tests and see them all pass
  - Refactor to remove duplication
- This cycle is followed until you have met your goal;
  - note that this cycle simply adds testing to the “add functionality; refactor” loop covered in refactoring

TDD Life Cycle, continued

- Kent Beck likes to perform TDD using a testing framework, such as JUnit.
- Within such frameworks
  - failing tests are indicated with a "red bar"
  - passing tests are shown with a "green bar"
- As such, the TDD life cycle is sometimes described as
  - "red bar/green bar/refactor"



Test class name: org.jfree.junit.JCommonTestSui

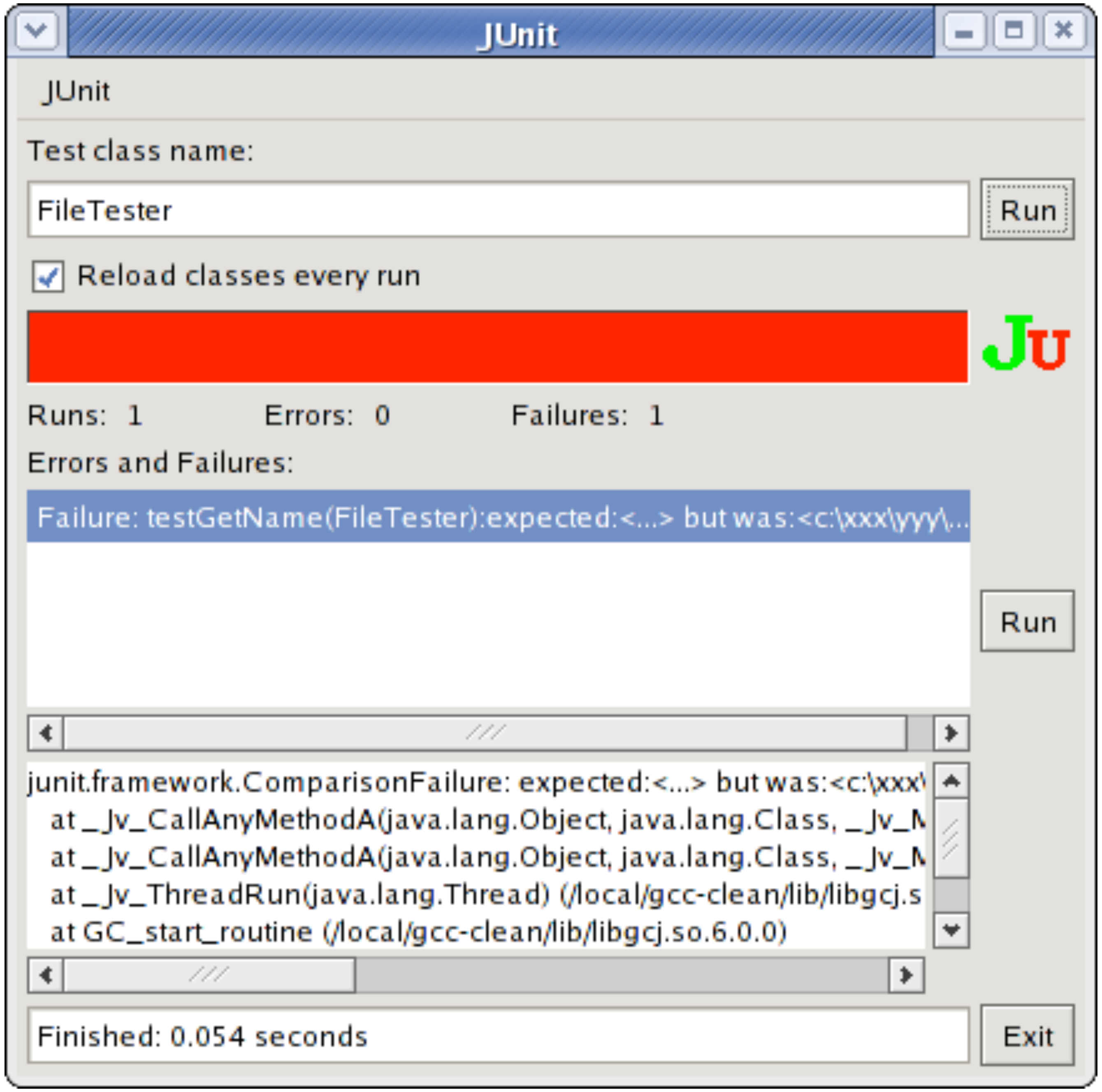
☑ Reload classes every run

Runs: 125/125 Errors: 0 Failures: 0

Finished: 8.086 seconds

JUnit: Red Bar...

- When a test fails:
  - You see a red bar
  - Failures/Errors are listed
  - Clicking on a failure displays more detailed information about what went wrong



```
Failure: testGetName(FileTester):expected:<...> but was:<c:\xxx\yyy\...
```

```
junit.framework.ComparisonFailure: expected:<...> but was:<c:\xxx\
  at _Jv_CallAnyMethodA(java.lang.Object, java.lang.Class, _Jv_M
  at _Jv_CallAnyMethodA(java.lang.Object, java.lang.Class, _Jv_M
  at _Jv_ThreadRun(java.lang.Thread) (/local/gcc-clean/lib/libgcj.s
  at GC_start_routine (/local/gcc-clean/lib/libgcj.so.6.0.0)
```

Finished: 0.054 seconds

Principles of TDD

- Testing List
  - keep a record of where you want to go;
    - Beck keeps two lists, one for his current coding session and one for “later”; You won’t necessarily finish everything in one go!
- Test First
  - Write tests before code, because you probably won’t do it after
  - Writing test cases gets you thinking about the design of your implementation;
    - does this code structure make sense?
    - what should the signature of this method be?

Principles of TDD, continued

- Assert First
  - How do you write a test case?
    - By writing its assertions first!
  - Suppose you are writing a client/server system and you want to test an interaction between the server and the client
    - Suppose that for each transaction
      - some string has to have been read from the server, and
      - the socket used to talk to the server should be closed after the transaction
  - Lets write the test case

Assert First

```
public void testCompleteTransaction {

   …

   assertTrue(reader.isClosed());

   assertEquals(“abc”, reply.contents());

}
```

- Now write the code that will make these asserts possible

Assert First, continued

```
public void testCompleteTransaction {
   Server writer = Server(defaultPort(), “abc”)
   Socket reader = Socket(“localhost”, defaultPort());
   Buffer reply = reader.contents();
   assertTrue(reader.isClosed());
   assertEquals(“abc”, reply.contents());
}
```

- Now you have a test case that can drive development
  - if you don’t like the interface above for server and socket, then write a different test case
  - or refactor the test case, after you get the above test to pass

Principles of TDD, continued

- Evident Data
  - How do you represent the intent of your test data
  - Even in test cases, we’d like to avoid magic numbers; consider this rewrite of our second “times” test case

```
public void testMultiplication() {
  Dollar five = new Dollar(5);
  Dollar product = five.times(2);
  assertEquals(5 * 2, product.amount);
  product = five.times(3);
  assertEquals(5 * 3, product.amount);
}
```

- Replace the “magic numbers” with expressions

Summary of Test Driven Development

- Test-Driven Development is a “mini” software development life cycle that helps to organize coding sessions and make them more productive
  - Write a failing test case
  - Make the simplest change to make it pass
  - Refactor to remove duplication
  - Repeat!