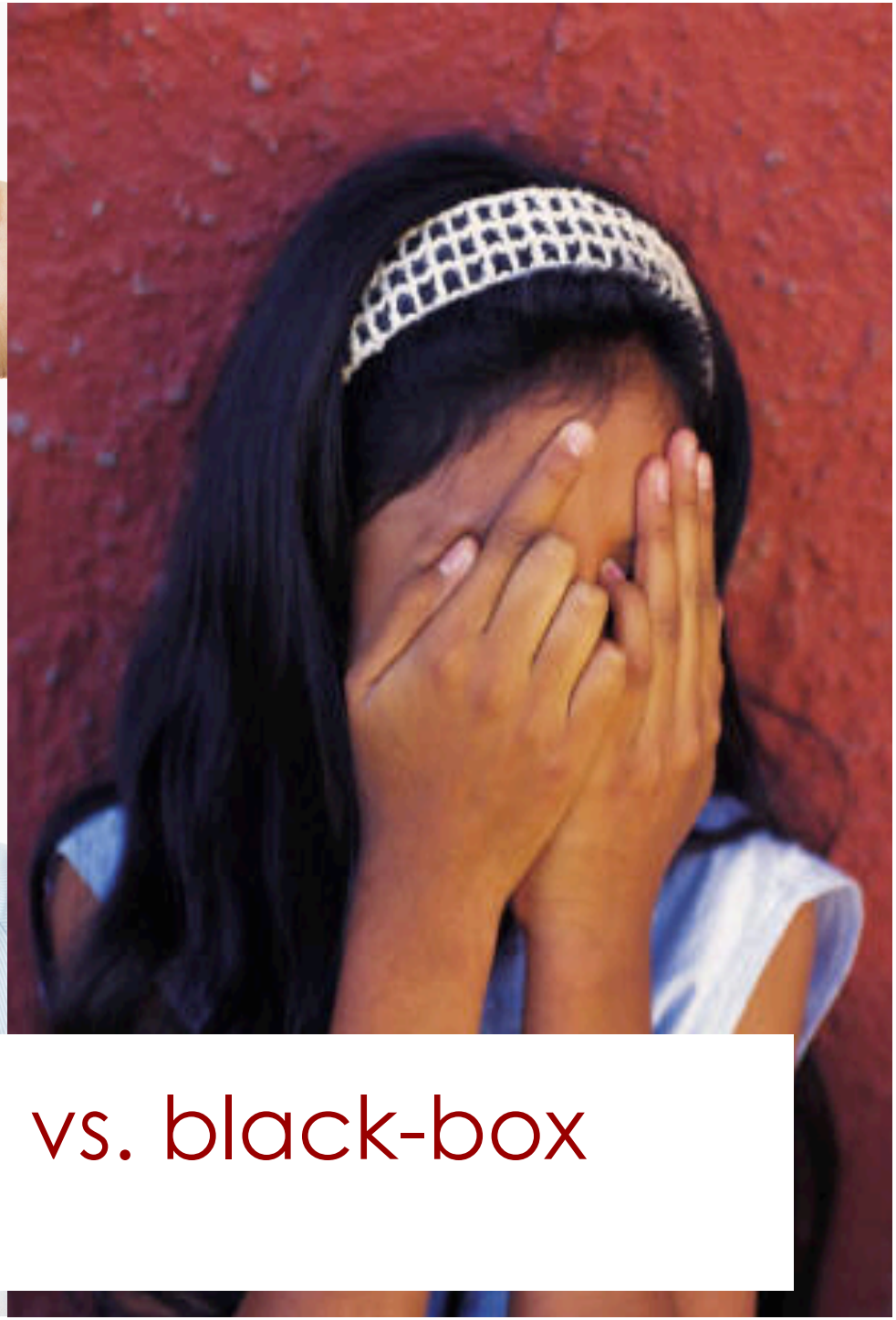
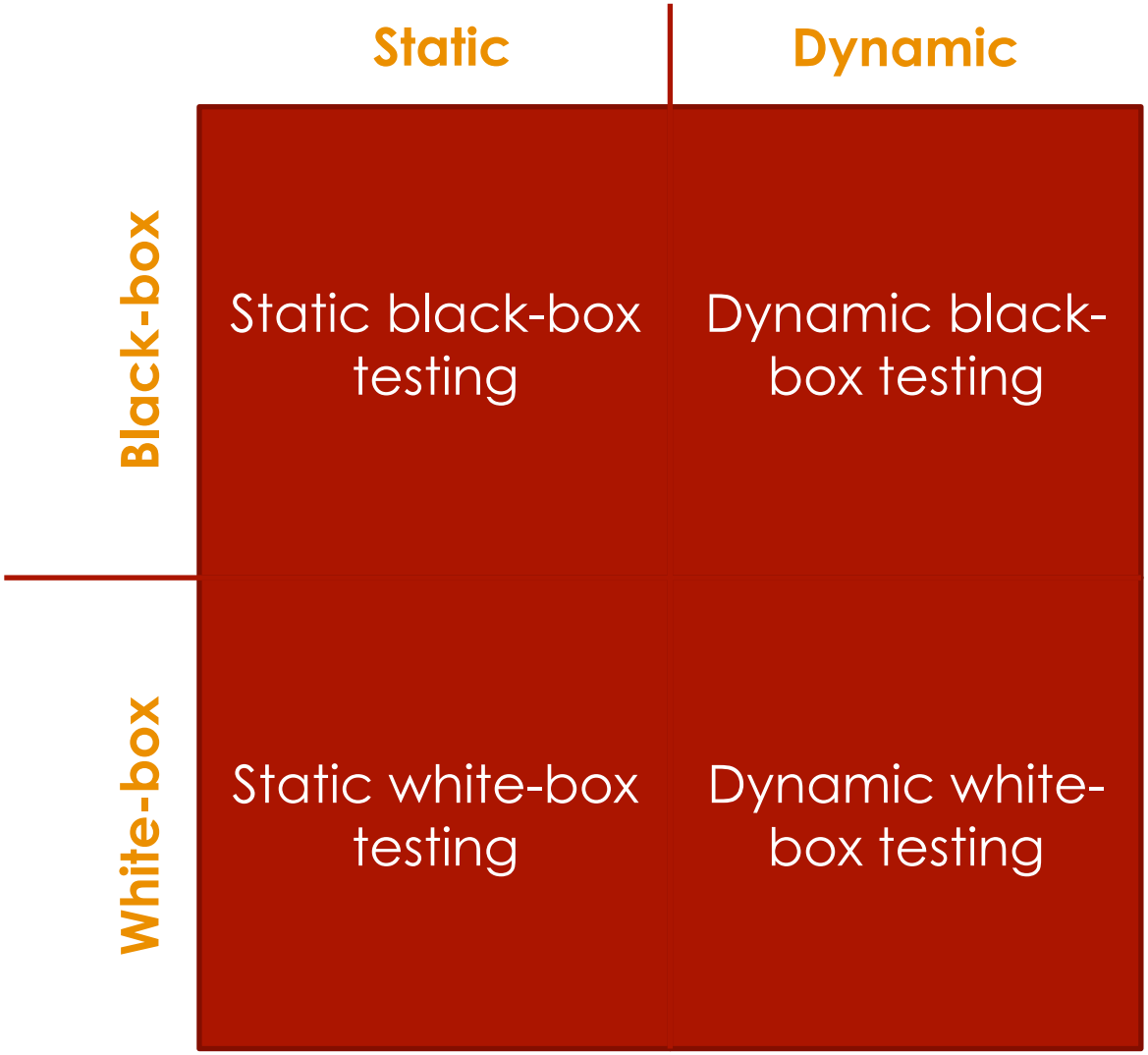
# Introduction to Unit Testing

# Static vs. Dynamic testing



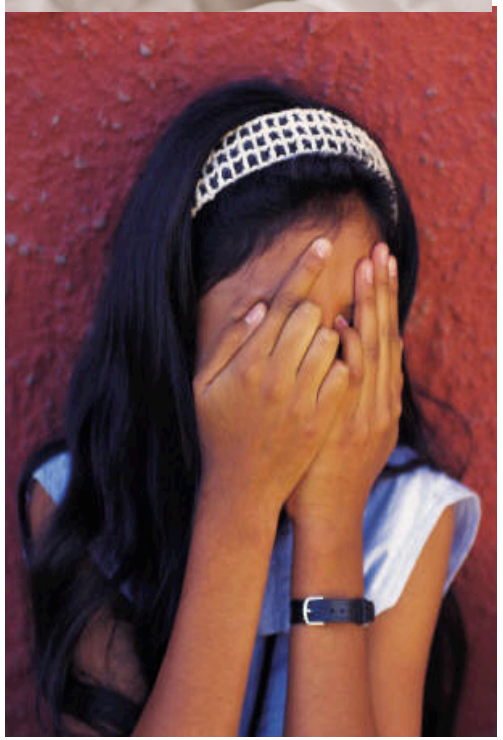
# White-box testing vs. black-box testing

# There are four quadrants, then



# Static black-box testing

- Can't see the code
- Can't run the program
- So, what can you do?
- You can evaluate the requirements and plans

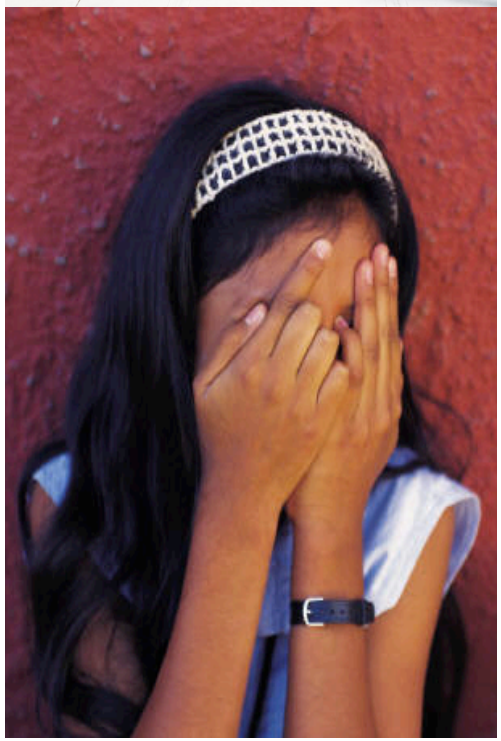


# Static white-box testing

- Can see the code
- But can't run it
- What does this entail?
- Code reviews & Code analysis tools

# Dynamic black-box testing

- Can't look at the code
- But you can run it
- So, what is this?
- Running test cases



# Dynamic white-box testing

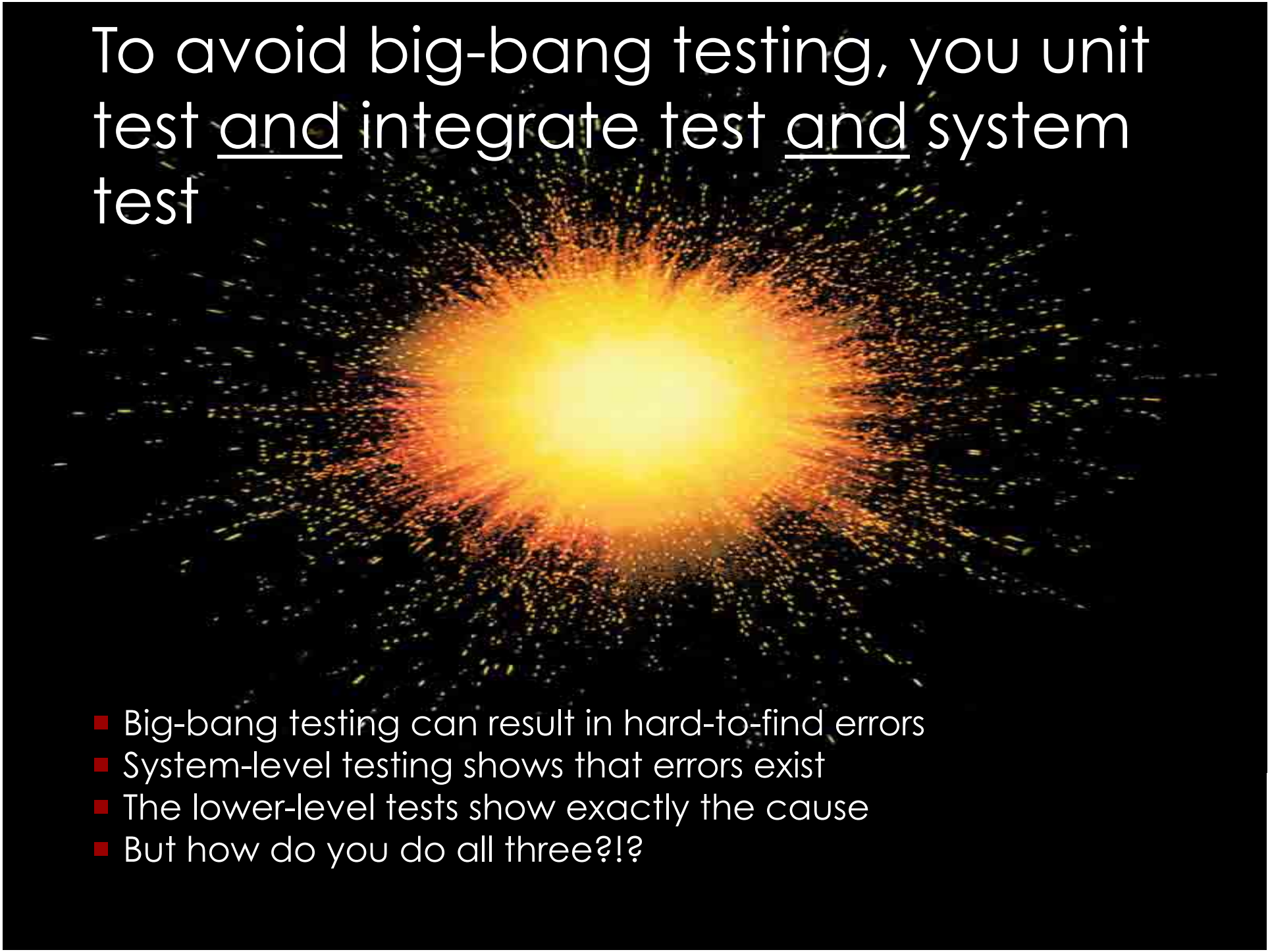
- You run the code …
- While examining it
- What is this called?
- Unit testing or integration testing

… but it isn't debugging

# Levels of dynamic white-box testing

- Unit testing
  - Lowest possible level
  - Completely isolated from anything else
- Integration testing
  - Next level up
  - Two or more units tested together
- System testing
  - Highest level
  - All units tested together

# To avoid big-bang testing, you unit test and integrate test and system test



- Big-bang testing can result in hard-to-find errors
- System-level testing shows that errors exist
- The lower-level tests show exactly the cause
- But how do you do all three?!?

# A unit test tests one unit of work

- One requirement for one method
- They are:
  - Isolated from other code
  - Isolated from other developers
  - Targeted
  - Repeatable
  - Predictable

# What is NOT unit testing? ... Anything else!

- > 1 requirement
- > 1 method
- > 1 project
- > 1 system
- Affects > 1 developer

# Benefits of unit testing

- Fewer errors
- Better designed
  - Unit testing requires good practices
  - Repository pattern
  - Dependency injection
  - Inversion of Control
  - Single responsibility
- Built-in regression testing
- Puts a stop to recurring bugs
  - When you discover a bug you write tests there and elsewhere to detect that same error

# Unit testing protects the database

- If you run against the database, you can corrupt the database with every test, especially failing ones
- One test may cause the next one to fail

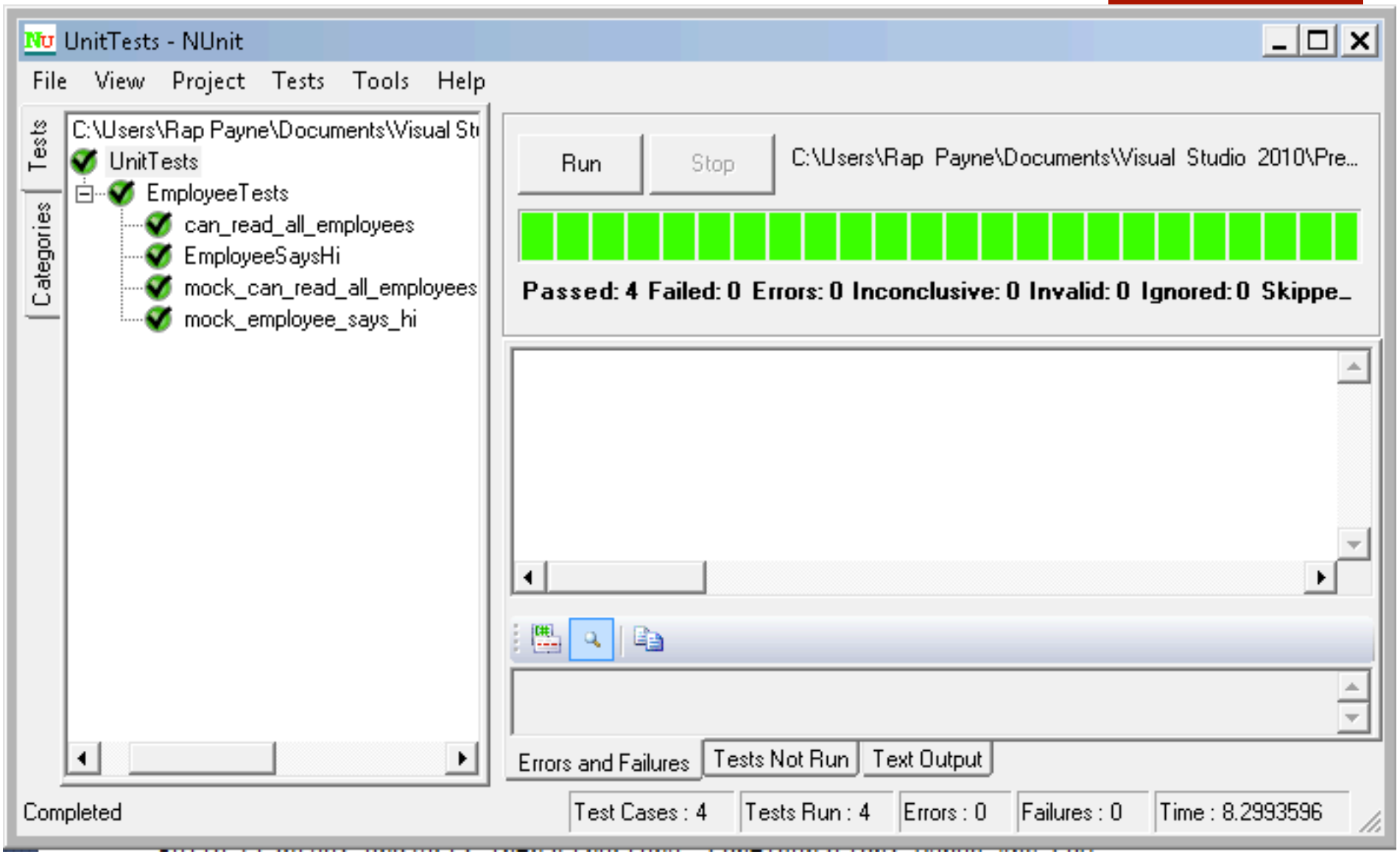
# Other types of tests

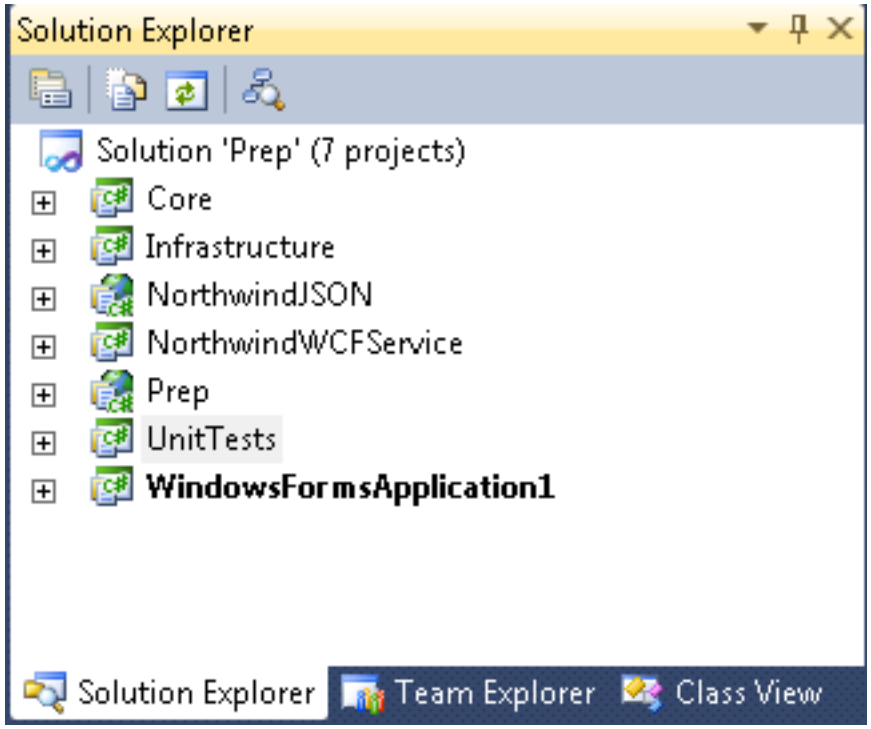
- UI
- Integration
- Stress
- UAT

# So, unit testing is awesome, but isn't it a lot of work?

- If only we had a framework that would allow us to easily format our tests
- If only we had some pre-written software that would allow us to see the results of those tests
- A unit testing framework!
- NUnit
- MbUnit
- xUnit
- MSTest

# A brief look at NUnit



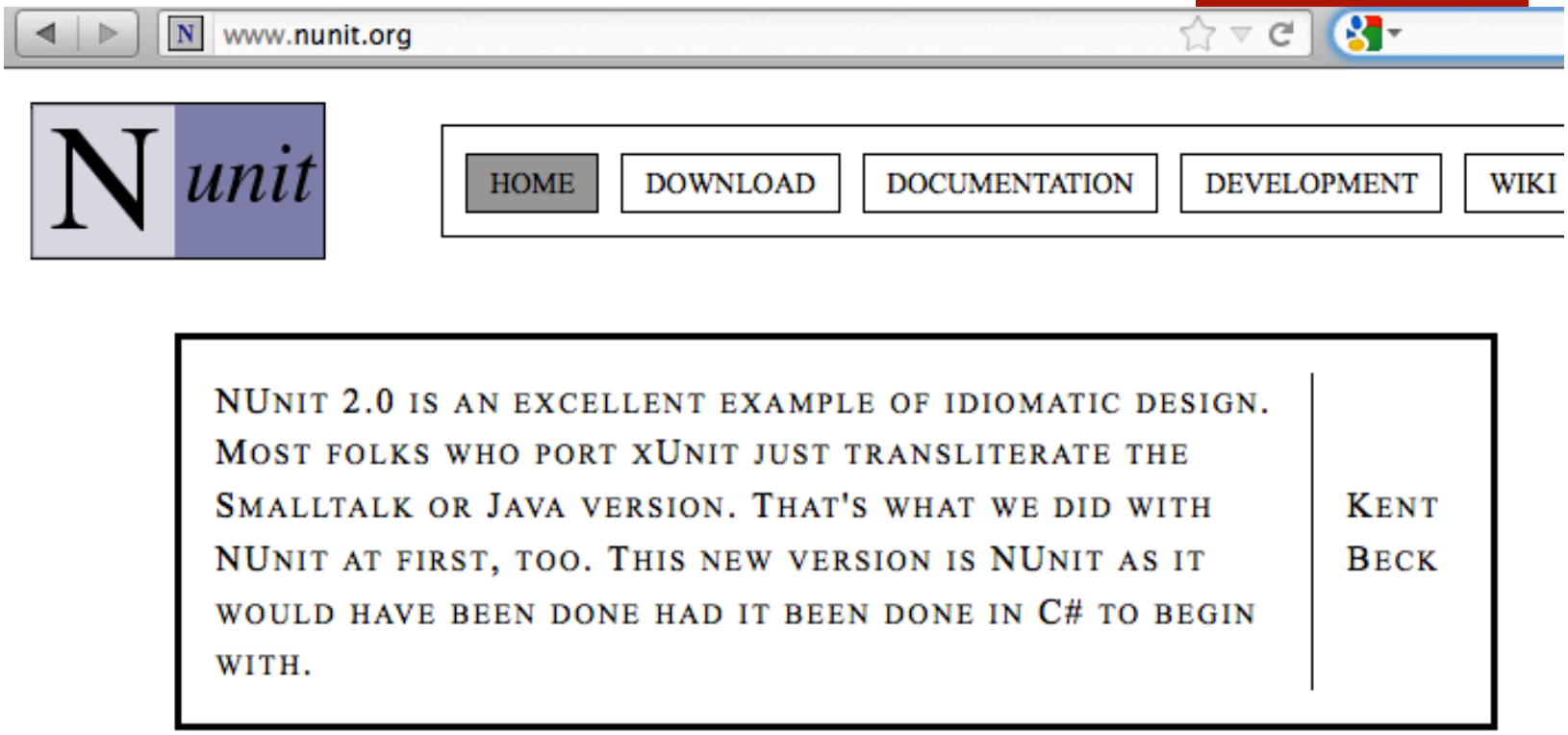



# Open your solution

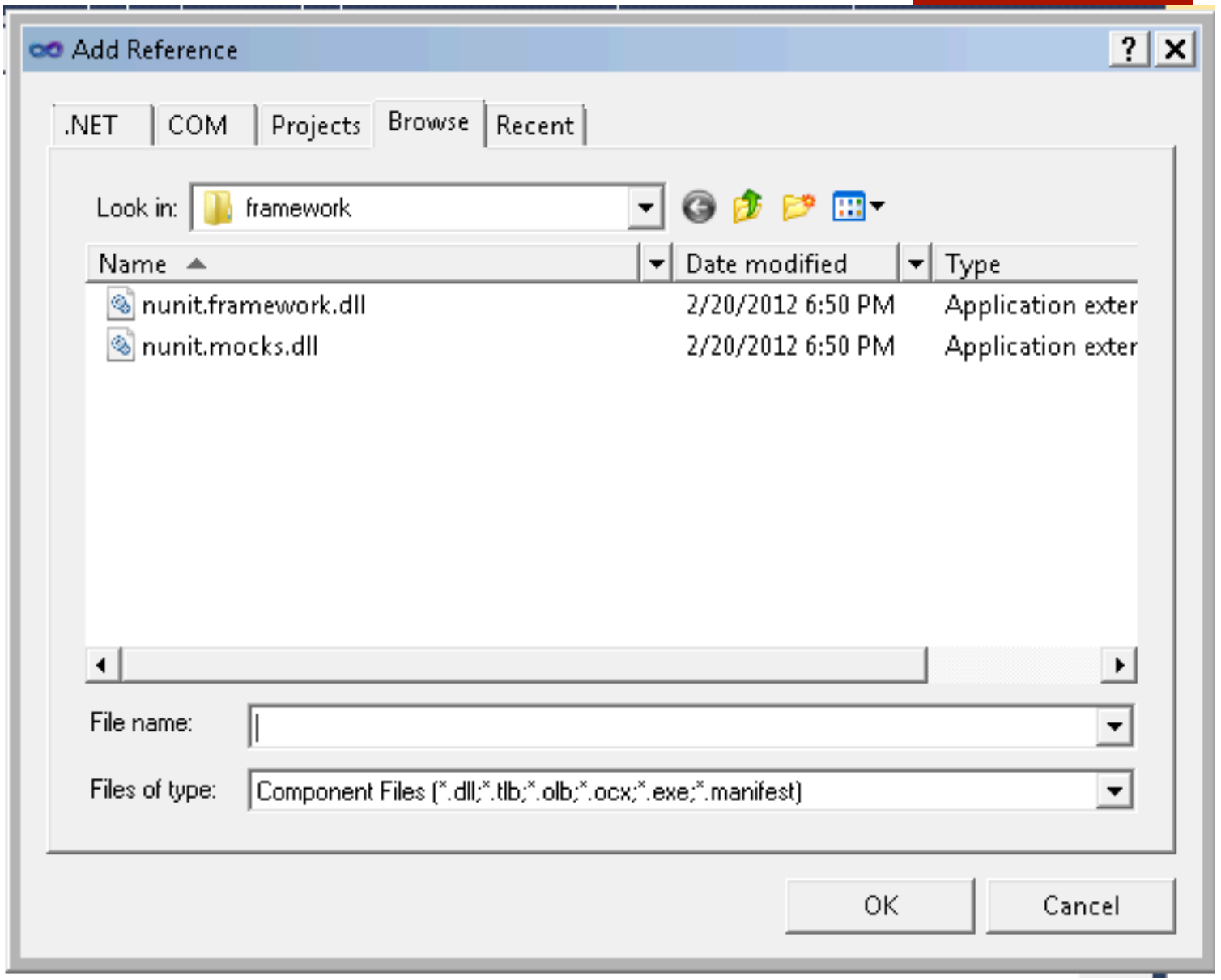
# Add a test project

- No user interface

# Download and install NUnit



# Add a reference to NUnit



# Add using statements

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using NUnit;
6  using NUnit.Framework;
7  using NUnit.Framework.Constraints;
8
```

# Mark the class as a TestFixture

```
namespace UnitTests
{
    [TestFixture]
    public class EmployeeTests
    {
…
```

# Write a test and mark it as a Test

```
[Test]
public void EmployeeSaysHi()
{
    var e = new Employee();
    var s = e.SayHi();
    Assert.AreEqual(s, "Hello");
}
```

# Comparison Assertions

- AreEqual(x, y)
- AreNotEqual(x, y)
- AreSame(x, y)
- AreNotSame(x, y)
- Greater(x, y)
- GreaterOrEqual(x, y)
- Less(x, y)
- LessOrEqual(x, y)

# Boolean Assertions

- False(bool)
- IsFalse(bool)
- True(bool)
- That(bool)
- IsTrue(bool)

# Exception Assertions

- Catch(exception)
- Throws(exception)
- DoesNotThrow(exception)

# Assert.*

- IsAssignableFrom(Type, object)
- IsEmpty(collection)
- IsInstanceOf(Type, object)
- IsNull(object)
- IsNullOrEmpty(string)
- Null(object)
- Contains(x, collection)
- IsNaN(number)
- IsNotAssignableFrom(Type, obj)
- IsNotEmpty(collection)
- IsNotInstanceOf(Type, object)
- IsNotNull(object)
- IsNotNullOrEmpty(string)
- NotNull(object)

# Hardcode Assertions

- Pass()
- Fail()
- Inconclusive()
- Ignore()

# Sometimes we want to ensure that an exception is thrown

- We sometimes have requirements that an exception should be thrown under certain circumstances
  - Age can't go negative
  - Etc.
- We can mark a test as expecting that an exception will be thrown:

```
[ExpectedException(
   ExpectedException = typeof(Exception),
   ExpectedMessage = "Any message."
   )]
public void HereIsMyTest()
{
   …
}
```

# Debugging the unit tests

1. Go Debug-Attach to process.
2. Find NUnit's agent in the process list.
3. Set a breakpoint
4. In NUnit, hit Run

# Summary

- Unit testing only occurs in complete isolation from every other requirement, method, user, etc.
- Unit testing frameworks are pretty much indispensible
- They allow us to make assertions