MSO
Testing

Hans Philippi

October 2, 2018

Why test software?

- Test software to try and break things to find bugs
- Test that your software has been installed correctly
- Test that code changes (new features or bug fixes) do not introduce new bugs
- Test performance, security, usability, accessibility, or any non-functional requirements
- . . .

What to test?

- Acceptance testing – test that the user accepts the final product
- System testing – test an entire system, before handing it over to end users
- Integration testing – test that different software components (that may have been tested individually) work together
- Regression testing – tests that run nightly to check for new bugs
- Unit testing – testing individual software units

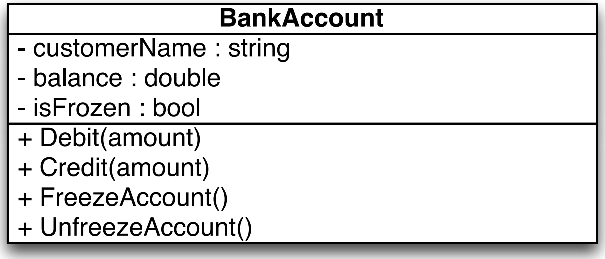
Unit testing

Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use.

Unit testing

- Test one unit at a time
- Don't test code that will not break, like getters and setters – focus on the difficult parts
- Tests are code too
    - Choose meaningful names (`testSuccessfulTransfer` rather than `test3`)
    - Organize tests, just as you would organize other code

Example: testing bank account software



Debit implementation

```
public void Debit(double amount)
{
  if (isFrozen)
  {
    throw new Exception("Account frozen");
  }
  if (amount > balance || amount < 0)
  {
    throw new
      ArgumentOutOfRangeException("amount");
  }
  balance -= amount;
}
```

Credit implementation

```
public void Credit(double amount)
{
  if (isFrozen)
  {
    throw new Exception("Account frozen");
  }

  if (amount < 0)
  {
    throw new
      ArgumentOutOfRangeException("amount");
  }

  balance += amount;
}
```

Testing. . .

- Test that tricky functions compute the right thing. . .
- . . . but also test that exceptional cases get triggered!
- Tests are typically written using *assertions*.
  - An assertion is passed a boolean expression
  - If the expression evaluates to True, nothing happens and the test passes . . .
  - . . . otherwise the assertion throws an error, and possibly reports debugging information

A first unit test

```
public void DebitWithValidAmount()
{
  double beginningBalance = 11.99;
  double debitAmount = 4.55;
  double expected = 7.44;
  BankAccount account =
    new BankAccount("Mr. Walter White"
                   , beginningBalance);

  // act
  account.Debit(debitAmount);

  // assert
  double actual = account.Balance;
  Assert.AreEqual(expected, actual, 0.001);
}
```

Another unit test

```
public void DebitTriggerException
{
  double beginningBalance = 11.99;
  double debitAmount = 20.0;
  BankAccount account =
    new BankAccount("Mr. Walter White", beginningBalance);
  try
  {
      account.Debit(debitAmount);
  }
  catch (ArgumentOutOfRangeException e)
  {
      StringAssert.Contains(e.Message, "amount");
      return;
  }
  Assert.Fail("No exception was thrown.")
}
```

What's missing?

- How would you check that no transactions can be done on frozen accounts?
- What other cases are missing?
- Is it good to have a the String "Amount" in both the test code and the original class?
- What about having duplicate setup code in every test?

How to test?

- The `Assert` class has a long list of basic assertions:
  - compare two primitive types for equality
  - assert that a boolean must be true
  - check that an object is not null
  - fail
  - inconclusive
- The `CollectionAssert` class provides various assertions about generic collections.
- The `StringAssert` class provides assertions related to strings:
  - Contains(String,String)
  - StartsWith(String,String)
  - EndsWith(String,String)
  - Matches(String,Regex)
- Various exception classes for when an assertion fails, when a test fails, when a test expects an exception to be thrown, etc.

Testing in Visual Studio

There is pretty good unit test support in Visual Studio

- You can generate a unit test framework for a class with a few clicks
- Automated tools for running tests and showing code coverage
- Help in writing stubs/mock classes

And lots more. . .

Unit testing complex systems

- Suppose you need to test a complex system that interacts with a database
- Querying the database takes a long time and is complex to set up
- Besides, I don't want my tests to corrupt my data
- How do I write unit tests?

Stubs, fakes, and mocks

- Stub objects always return the same response
  - Every database query returns the same result
- Fake objects behave correctly, but have a simplistic implementation
  - A fake database object has a list of data base entries stored as a C# list
- Mock objects also test that its calls satisfy constraints
  - The mock database knows about the database schema and also tests that queries are valid

Unit testing: best practices

1. Make sure your tests test one thing and one thing only
2. Write unit tests as you go; preferably before you write the code you are testing against
3. Do not unit test the GUI
4. Minimise the dependencies of your tests
5. Mock behaviour with mocks

Test-driven development

With the rise of Agile methodologies, *Test-driven development* has gained a lot of attraction in the past years

1. Start the iteration by writing tests
2. Add code that makes the tests pass
3. Refactor your solution

This approach is sometimes summarized as ‘red-green-refactor’

Question: What might the benefits of this approach be?

Benefits of TDD

- It forces developers to think about *specifications first*
- Strongly coupled or weakly cohesive code is hard to test; writing tests first encourages better design
- Tests are not an afterthought; the end of the iteration may encourage developers to skimp on writing tests
- Code will require less refactoring later

Material covered

- Refactoring. Martin Fowler.
- Verifying Code by Using Unit Tests. MSDN. Available online.