Unit Testing

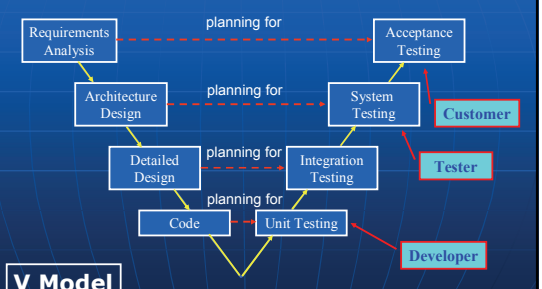
陳偉凱

台北科大資工系、軟體中心

Outline

- Why unit testing?
- Unit Testing Basics (exercise #1)
- More Unit Testing (exercise #2,3)
- Difficult to Test? (exercise #4)
- Mock Objects (exercise #5)
- Conclusions

Software Testing



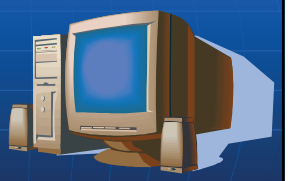
V Model

Software Testing

- Testing should begin "in the small" and progress toward "in the large"
  - Unit testing
    - Concentrates on each unit (i.e., component)
  - Integration testing
    - Building a system from its components and testing the resultant system for detecting component interaction problems
  - System testing
    - Concern with testing an increment to be delivered or entire system
  - Acceptance testing
    - Performed by the clients/users to confirm that the product meets the business requirements

An Analogy

- How do you assemble a computer?
  - CPU
  - Power supply
  - Motherboard
  - RAM
  - Cable
  - Hard Disk
  - Fan
  - …



Unit testing ⇔ System testing

Unit Testing

- Unit Testing
  - Is an adjunct to the coding step
  - Provide a release criterion for a programming task
  - Focuses on the smallest unit of software design
    - Smallest testable unit is a method of a class
    - Every method within the class is tested
    - The state behavior of the class is examined

What is xUnit?

- An automated unit test framework
  - Provides automatic test runs
  - Provides automatic result checks
  - Can be used to do integration testing, etc.
- Available for multiple languages:
  - JUnit
  - CppUnit
  - HttpUnit
  - NUnit
  - GTest
  - …

Unit Testing Basics

Unit Testing: Methodology

Production code

```
class X {
    . . .
    void Foo() {
        . . .
        . . .
    }
};
```

Function under test

Test code

```
class XTest {
    . . .
    void testFoo() {
        X x;
        x.Foo();
        CPPUNIT_ASSERT(…);
    }
    . . .
};
```

Test case(s)

Unit Testing: Methodology

```
class X {
    . . .
    int Max(int x, int y) {
        . . .
        . . .
    }
};
```

Example #1

```
class XTest {
    . . .
    void testMax() {
        X x;
        int max = x.max(10,5);
        CPPUNIT_ASSERT(max == 10);
        max = x.max(-10,6);
        CPPUNIT_ASSERT(max == 6);
    }
    . . .
};
```

No parameters; no return value

Pass, if true

Fail: red light
Pass: green light

Unit Testing: Methodology

```
class X {
    . . .
    int Max(int a[], int size) {
        . . .
        . . .
    }
};
```

Example #2

```
class XTest {
    . . .
    void testMax() {
        X x;
        int a[5]={3,4,5,1,0};
        int max = x.max(a,5);
        CPPUNIT_ASSERT(max == 5);
    }
    . . .
};
```

Three steps:
(1) Prepare test data
(2) Invoke target method
(3) Assert correctness
Repeat (1)-(3)

Unit Testing: Methodology

```
class Account {
    ...
    double balance;
    void Deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
};
```

Example #3

```
class AccountTest {
    . . .
    void testDeposit() {
        Account a;
        a.balance = 100;
        a.Deposit(20);
        CPPUNIT_ASSERT(a.balance == 120);
        a.Deposit(-20);
        CPPUNIT_ASSERT(a.balance == 120);
        ...
    }
};
```

Let's assume balance and Deposit are public for now

Unit Testing: Methodology

- CppUnit ASSERT
  - **CPPUNIT_ASSERT(condition)**
    - Assertions that a condition is true
  - **CPPUNIT_ASSERT_EQUAL(expected, actual)**
    - Asserts that two values are equals
  - **CPPUNIT_ASSERT_THROW(expression, ExceptionType)**
    - Asserts that the given expression throws an exception of the specified type
  - **CPPUNIT_ASSERT_NO_THROW(expression)**
    - Asserts that the given expression does not throw any exceptions.

Unit Testing: Methodology

- JUnit Assert
  - **assertEquals** (long expected, long actual)
  - **assertEquals**(double expected, double actual, double delta)
  - **assertFalse**(boolean condition)
  - **assertTrue**(boolean condition)
  - **assertNotNull**(java.lang.Object object)
  - **assertNull**(java.lang.Object object)
  - **assertSame**(java.lang.Object expected, java.lang.Object actual)
  - **assertArrayEquals**(int[] expecteds, int[] actuals)

Test case preparation exercise

```
// GetRank
// Given an array a[] with n elements
// Assume x is a member of a[]
// Find the rank (the ith largest element) of x in a[]
// Example: let a[] = { 40, 70, 60, 60, 90}
//          n = 5;
//  GetRank(a, 5, 90) returns 1
//  GetRank(a, 5, 40) returns 5
//  GetRank(a, 5, 60) returns 3
//  ...

int GetRank(int a[], int n, int x) {
   ...
   ...
}
```


Unit Testing: Statement coverage (1)

```
class X {
   . . .
   void f() {
      . . .
      . . .
   }
};
```

```
class XTest {
   . . .
   void testf() {
      X x;
      x.f();
      CPPUNIT_ASSERT(…);
   }
   . . .
};
```

Statement coverage: are all statements of f() executed?

Unit Testing: Statement coverage (2)

```
class X {
   . . .
   void f(bool x) {
      . . .
      if (x) {
         . . .
      }
      . . .
   }
};
```

```
class XTest {
   . . .
   void testf() {
      X x;
      x.f(false);
      CPPUNIT_ASSERT(…);
   }
   . . .
};
```

There are unexecuted statements

Unit Testing: Statement coverage (3)

```
class X {
   . . .
   void f(bool x) {
      . . .
      if (x) {
         . . .
      }
      . . .
   }
};
```

```
class XTest {
   . . .
   void testf() {
      X x;
      x.f(true);
      CPPUNIT_ASSERT(…);
   }
   . . .
};
```

All statements are executed

Unit Testing: Branch coverage (1)

```
class X {
    . . .
    void f(bool x) {
        . . .
        if (x) {
            . . .
        }
        . . .
    }
};
```

Unexercised

```
class XTest {
    . . .
    void testf() {
        X x;
        x.f(true);
        CPPUNIT_ASSERT(…);
    }
    . . .
};
```

All statements are executed

Brach coverage: are all control transfer exercised?

Unit Testing: Branch coverage (2)

```
class X {
    . . .
    void f(bool x) {
        . . .
        if (x) {
            . . .
        }
        . . .
    }
};
```

```
class XTest {
    . . .
    void testf() {
        X x;
        x.f(true);
        CPPUNIT_ASSERT(…);
        . . .
        x.f(false);
        CPPUNIT_ASSERT(…);
    }
    . . .
};
```

Two test cases in the same test method

Is reusing x a good idea?

All control transfers are exercised

Unit Testing: Branch coverage (3)

```
class X {
    . . .
    void f(int x) {
        . . .
        do {
            x--;
            . . .
        }
        while (x>0);
        . . .
    }
};
```

Code coverage analysis tools
Eclipse
Java: EclEmma
C: gcov
Visual Studio 2010 (enterprise)
C#: built-in

```
class XTest {
    . . .
    void testf() {
        X x;
        x.f(0);
        CPPUNIT_ASSERT(…);
    }
    . . .
};
```

Statement coverage? Yes

Branch coverage? No

Path coverage?

Test case preparation

- How can I make sure that all of the important behaviors of the program are tested?
  - Usually, you can't!
- Test case preparation
  - Test specifications
    - Use case, module, class, method specifications
  - Test boundary conditions
    - First, Last, Empty, Full
  - Test exceptions
    - Out of range, Negative, Overflow
    - Exceptions that should be thrown
  - Test common mistakes
    - void append(List list1, List list2);
      - Does append work if list1==list2?
  - Test what is tricky to implement
    - E.g., Partition

Test case preparation exercise

```
// isPrime
//   A prime number is a positive integer that is
//   is divisible only by 1 and itself
//   For example, 2, 5, 7, 11, … are prime numbers
//
// The function isPrime(x) returns
//   true    if x is a prime number
//   false   if x is not a prime number

bool [illegible](int x)
{
  ..
}
```

Don't forget non-primes

Test case preparation exercise

```
// Select an element as pivot, located at a[p]
// Partition A[] so that a[l]..a[p-1] < a[p]
//                        a[p+1]..a[r] >= a[p]
// return the index of the pivot
int partition(int a[], int l, int r) {
    ...
    ...
};
```

< a[p]	a[p]	>= a[p]
l	p	r

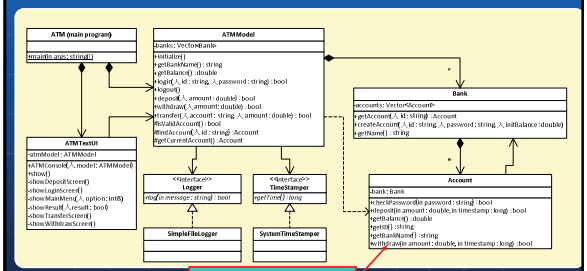
Exercise #1

Writing a simple test case

- White-box testing
- Android ATM app (next page)
- Target
  - testWithdraw()
  - 100% statement and branch coverage
  - Boundary conditions

Android ATM App

- Deposit, Withdraw, Transfer, Log



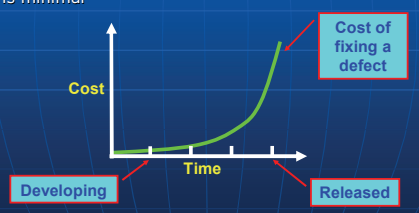
More Unit Testing

Advantages of Unit Testing

- Write test
  - Understand better what has to be done
  - Gain confidence in your code
- Provide a working specification of code
  - Tests are good documentation
- Silly bugs appear immediately
  - Less time spent on debugging → reduce overall cost (next page)

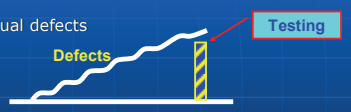
Testing: Early, Often, and Automated

- DCI (Defect Cost Increase)
  - The sooner a defect is found, the cheaper it is to fix.
  - Catch the defect the minute it is created → cost to fix it is minimal



Testing: Early, Often, and Automated

- Late testing (Long feedback loop)
  - Expensive
  - Many residual defects

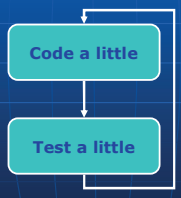


- Frequent testing (Short feedback loop)

- The Eclipse open source project has more than 21,000 unit tests

Best Practice

- Code a little, test a little



Best Practice

- **You don't need to write test cases for code that is too simple to break**
  - If it can't break on its own, it's too simple to break.
  - getX(): don't test getX(); cannot break unless the compiler is also broken.
  - setX(): if it does parameter validation, you likely need to test it.

```
class X {
    ...
    int age;
    int getAge() {
        return age;
    }
}
```

```
class X {
    ...
    int age;
    int setAge(int age) {
        if (...) {
            this->age = age;
        }
    }
}
```


Best Practice

- **All methods should be tested (except for methods that are too simple to break)**

```
class X {
    ...
    int Sum(int x, int y)
    {
        return x + y;
    }
}
```

```
class XTest {
    ...
    void testSum()
    {
        X x;
        CPPUNIT_ASSERT(x.Sum(1,2) == 3);
    }
}
```


Testing an entire Class

```
class X {
    . . .
    void f1() {...}
    void f2() {...}
    void f3() {...}
    ...
};
```

```
class XTest {
    . . .
    void testf1() {...}
    void testf2() {...}
    void testf3() {...}
    . . .
    void testX() {
        f1();
        f2();
        CPPUNIT_ASSERT(...);
        f1();
        f3();
        CPPUNIT_ASSERT(...);
        ...
    }
};
```

In addition to testing every method, test possible state changes of the class X, if necessary

Testing an entire Class: example

```
class Player {
    . . .
    void PowerOn() {...}
    void Play() {...}
    void Stop() {...}
    void Pause() {...}
    ...
};
```

```
class PlayerTest {
    . . .
    void testPowerOn() {...}
    void testPlay() {...}
    void testStop() {...}
    . . .
    void testPlayer() {
        Player p;
        p.PowerOn();
        p.Play();
        p.Pause();
        p.Stop();
        CPPUNIT_ASSERT(...);
        p.PowerOn();
        p.Stop();
        CPPUNIT_ASSERT(...);
        ...
    }
};
```

Testing possible user scenarios of using Player

Testing an entire Class: Test Fixture

```
class XTest {
    . . .
    void testf1() {X x; …; x.f1(…);…}
    void testf2() {X x; …; x.f2(…);…}
    void testf3() {X x; …; x.f3(…);…}
    . . .
};
```

C++

```
class XTest {
    . . .
    void testf1() {…; x.f1(…);…}
    void testf2() {…; x.f2(…);…}
    void testf3() {…; x.f3(…);…}
    ...
    X x;
};
```

C++: put commonly used variables in XTest

Testing an entire Class: Test Fixture

```
class XTest {
    . . .
    void testf1() {X x = new X(); …; x.f1(…);…}
    void testf2() {X x = new X(); …; x.f2(…);…}
    void testf3() {X x = new X(); …; x.f3(…);…}
    . . .
};
```

Java / C#

```
class XTest {
    . . .
    void setUp()     {x = new X(); …}
    void tearDown()  {// x.release() if necessary}
    void testf1()    {…; x.f1(…);…}
    void testf2()    {…; x.f2(…);…}
    void testf3()    {…; x.f3(…);…}
    . . .
    X x;
};
```

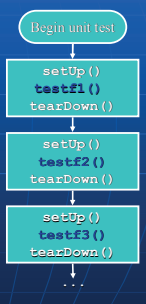
Java / C#: initialize commonly used variables in setUp()

Release resources in tearDown()

C++ also needs setUp() and tearDown()

Testing an entire Class: Test Fixture

```
class XTest {
    . . .
    void setUp() {...}
    void tearDown() {...}
    void testf1() {...}
    void testf2() {...}
    void testf3() {...}
    . . .
    void testX() {
        x.f1();
        x.f2();
        CPPUNIT_ASSERT(...);
        x.f1();
        x.f3();
        CPPUNIT_ASSERT(...);
        ...
    }
};
```



38

Best Practice

- **Isolated test**
  - Test case (method) execution order should be irrelevant
    - No good: test case #i creates a file for test case #j to use.
    - Every test case should be self-contained
  - A test method should be independent
    - Test only one method
      - Should not have test code that tests other methods
    - Ideally, should not rely use another method (class)

39

Best practice

- **Test Data**
  - An exception to "no magic numbers"
    - For small scopes; use constants if it's already there
  - One constant one meaning
    - Test 2+2 is bad; test 2 + 3 is better (order of argument)
  - Make the relationship of expected and actual results apparent

```
void testMax() {
    X x;
    CPPUNIT_ASSERT(x.max( 10, 5) == 10);
    CPPUNIT_ASSERT(x.max(-10, 6) ==  6);
    CPPUNIT_ASSERT(x.max( -3,-9) == -3);
}
```

Don't reuse 5

Actual

Excepted

Test Data

```
class Sudoku {
public:
    ...
    // check if a particular row is legal
    //    true   if data[row][0]..data[row][8] is legal, i.e.,
    //           data are within 0..9 and are different (except 0)
    //    false  otherwise
    bool is_legal_row(int row);
    ...
    //    1..9: number; 0: empty
    int data[9][9];
};
```

					9	7		
		8		3	5	1	4	
2		9	5	7		3		
5	2		9	8	4	7		
		3	6	1	7		2	5
		4		3	2	6		8
8	5	2	7			4		
	1	6						

41

Test Data

```
class SudokuTest … {
    …
    void test_is_legal_row()
    {
        Sudoku s;
        for (int i = 0; i < 9; i++)
            for (int j = 0; j < 9; j++)
                s.data[i][j] = j+1;
        s.data[0][0] = 0;
        CPPUNIT_ASSERT(s.is_legal_row(0) == true);
        s.data[1][0] = 2;
        CPPUNIT_ASSERT(s.is_legal_row(1) == false
        CPPUNIT_ASSERT(s.is_legal_row(2) == true
    }
};
```

What's wrong?

0	2	3	4	5	6	7	8	9
2	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

- All test data have the same pattern (too regular)
- data[0][0], data[1][0]
- Lack of other false conditions
- Lack of other true conditions
- Lack of rows 3-8

Test Data

```
class SudokuTest ... {
    ...
    void test_is_legal_row()
    {
        Sudoku s;
        int data[9][9] = {{1,2,3,4,5,6,7,8,9},
                          {0,9,5,3,2,1,6,7,8}, ...};
        for (int i = 0; i < 9; i++)
            for (int j = 0; j < 9; j++)
                s.data[i][j] = data[i][j];
        for (int i = 0; i < 4; i++)
            CPPUNIT_ASSERT(s.is_legal_row(i) == true);
        for (int i = 4; i < 9; i++)
            CPPUNIT_ASSERT(s.is_legal_row(i) == false);
    }
};
```

1	2	3	4	5	6	7	8	9
0	9	5	3	2	1	6	7	8
5	2	0	4	9	6	3	1	0
7	0	3	0	0	0	0	0	2
1	2	3	4	5	1	7	8	9
1	2	3	4	2	6	7	8	2
0	2	3	4	2	6	7	0	2
1	2	3	4	10	6	7	8	9
1	2	-3	4	5	6	7	8	9

Some other test methods can also use this table

Test Data and Assertion

```
class Sudoku {
...
    // find and return a legal number (between 1..9) for data[x][y]
    //   1..9  if such a legal number exists
    //   0     if such a number does not exist
    //
    int find_legal_number(int x, int y);
};
```

What's wrong?

```
void test_find_legal_number()
{
    Sudoku s;
    // Make s.data[][] the same as the picture
    ...
    int n = s.find_legal_number(0,0);
    CPPUNIT_ASSERT(n >=1 && n <= 9);
}
```

						9	7	
		8			3	5	1	4
2		9	5	7		3		
5	2		9	8	4	7		
		3	6	1	7		2	5
		4		3	2	6		8
8	5	2	7			4		
	1	6						

Bad test data; bad assertion

Exercise #2

Extending the simple test case

- Black-box testing
- Please implement testWithdraw() for a Withdraw that enforces a limitation of $30000/day
- Focus on what you should test
- Code coverage is not important for now

Exercise #3

Implementing production code

- Implement a withdraw that can pass the tests
- Check code coverage after implementation
- TDD (Test-Driven Development)
- How many times did you manipulate ATM UI to ensure that your implementation was correct ?

Difficult to test?

Doc, I am sick

NTUT SDRC

Unit Testing: Difficult to Test?

```
class X {
    . . .
    void foo() {
        string s;
        . . .
        s += ...;
        . . .
        cout << s;
    }
};
```

```
class X {
    . . .
    void foo() {
        cout << compute_s();
    }
    string compute_s() {
        string s;
        . . .
        s += ...;
        . . .
        return s;
    }
};
```

What if the output is console?

Better: separate console output and business logics.

Even better: separate View and Model (MVC pattern) and make your view as thin as possible.

NTUT SDRC

Best Practice

- Sometimes, you need to rewrite your code to make it easier to test

```
class X {
    . . .
    void LoadLogFile() {
        ifstream fin("SomeLogFileName");
        . . .
    }
};
```

What if the content of the log file is changing? (the log file increases whenever the application gets executed)

NTUT SDRC

Unit Testing: Difficult to Test?

```
class X {
    . . .
    void LoadLogFile(string fileName) {
        ifstream fin(fileName.c_str());
        . . .
    }
};
```

Better

```
class XTest {
    . . .
    void testLoadLogFile() {
        X x;
        x.LoadFromFile("a_log_file");
        // The file is now under
        //     control
    }
    . . .
};
```

Rewrite your code to get control of the inputs

NTUT SDRC

Unit Testing: Difficult to Test?

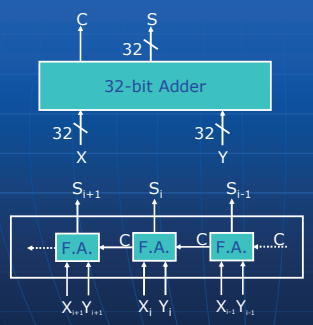
What's wrong?

```
void foo(int cardType, int mode) {
    for (...) {
        if (cardType == GOLD) {
            switch (mode) {
                case NORMAL_MODE:
                    while (...) {
                        ...
                    }
                    ...
                break;
            }
        }
        else if (cardType == PLATINUM) {
            ...
        }
    }
    ...
}
```

Too many levels of nesting; Difficult to cover all branches

NTUT SDRC

Unit Testing: Difficult to Test?



Black box testing

$2^{32} \times 2^{32} = 2^{64}$

White box testing

$2^3 \times 32 + 5 \times 32$

$= 416 < 2^9$

NTUT SDRC

Unit Testing: Difficult to Test?

```
double Discount(double amount, Customer c, Item items) {
    double ratio, customer_discount, item_discount;

    ...            // amount 10000-19999 0.9
    ratio = ...;   // amount 20000-39999 0.88
    ...            // amount 40000-79999 0.85

    ...                        // c is Member
    customer_discount = ...;   // c is VIP
    ...                        // c is Senior

    ...                   // Item #1 -$10
    item_discount = ...;  // Item #2 -$20
    ...                   // Item #3 80%

    return amount * ratio - customer_discount - item_discount;
}
```

Problems?

10 cases (5 ratios and 2 boundaries each

4 customer types

20 discount items

Total
10 x 4 x 20 = 800 cases

Unit Testing: Difficult to Test?

```
double Discount(double amount, Customer c, Item items) {
   double ratio, customer_discount, item_discount;
   ratio = Compute_ratio(amount);
   customer_discount = Compute_customer_discount(c);
   item_discount = Compute_item_discount(items);
   return amount * ratio - customer_discount - item_discount;
}
```

Logics that should be tested

```
void testDiscount() {
   // test logics in Discount(): maybe 3 cases
}
void testCompute_ratio() {
   // test 10 ratios and amount boundaries
}
void testCompute_customer_discount() {
   // test 4 customer types
}
void testCompute_item_discount() {
   // test 20 item discounts
}
```

Total
3 + 10 + 4 + 20 = 34 cases

Best Practice

- **Breaking a long method into smaller ones makes it easier to test**

```
class X {
   . . .
   void LoadFromFile(ifstream &fin) {
      . . .
      if (condition #1)
         LoadPart1();
      . . .
      LoadPart2();
   }
};
```

Long method

Testing LoadFromFile(): focus on the logics inside LoadFromFile().

```
class XTest {
   . . .
   void testLoadFromFile() {
   }
   void testLoadPart1() {
   }
   void testLoadPart2() {
   }
   . . .
};
```

Test each smaller function first.

Best Practice

- **Do I need to test private methods?**
  - Yes.
- **How to test a private method/attribute?** (next)
  - Language independent
    - Make it public, if being public is reasonable
    - Make the private member of X protected, and inherit XTest from X so that the protected member can be accessed by XTest
    - Make XTest an inner class of X
  - Language dependent
    - C++ friend class
    - C# privateobject (Visual Studio)
    - Java package private
    - Java PrivateAccessor

Unit Testing: Difficult to Test?

```
class X {
   . . .
   void Foo(double factor) {
      ...
      out = in * factor;
   }
   double out, in;
};
```

Use public SetXXX() and GetXXX() to access member variables

In addition to parameters, Foo() uses member variables

```
class XTest {
   . . .
   void testFoo() {
      X x;
      x.SetIn(100);
      x.Foo(0.5);
      CPPUNIT_ASSERT
         (x.GetOut() == 50);
   }
   . . .
};
```


Unit Testing: Difficult to Test?

```
class X {
   friend class XTest;
   . . .
   void Foo(double factor) {
      ...
      out = in * factor;
   }
   double out, in;
};
```

Make XTest a friend of X

In addition to parameters, Foo() uses member variables

```
class XTest {
   . . .
   void testFoo() {
      X x;
      x.in = 100;
      x.Foo(0.5);
      CPPUNIT_ASSERT
         (x.out == 50);
   }
   . . .
};
```


Design for Testability

- What is Testability?
  - Control
    - The ability to provide inputs and reach states in the software under test
  - Visibility
    - The ability to observe the states and outputs of the software under test
- Design for Testability
  - Rewrite (redesign) your code to get control and visibility

Design for Testability

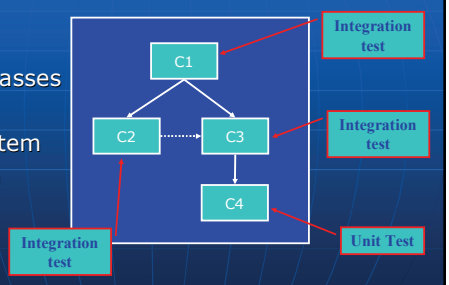
- How do you test the code that handles disk full?

```
class Document {
   ...
   void SaveDocument (string fileName) {
      // Create document file
      ...
      // write document to file
      ...
      // If disk full, handle the error
      if (disk full) {
         ...
      }
   }
};
```

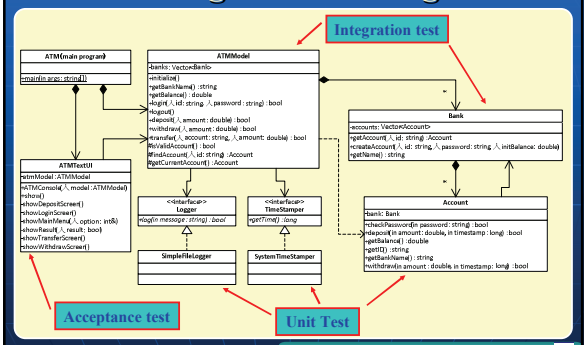
How to test the code?

Integration Testing

- Method
- Class
- Multi-classes
- Module
- Subsystem
- System



Integration Testing



Exercise #4

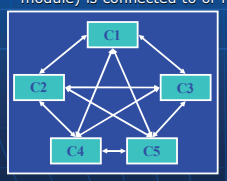
Integration testing

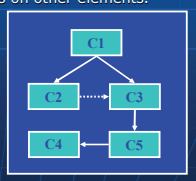
- Write the test case for ATMModel.transfer
- White-box testing
- 100% statement and branch coverage
- Ignore Log for now

Mock Objects

Unit Testing and Design

- Unit Testing and Design
  - When a class (method) is easy to test → high cohesion and low coupling
  - Easy to test → good design
  - Difficult to test → maybe a bad design
- Coupling
  - A measure of how strongly one element (method, class, module) is connected to or relies on other elements.





High Cohesion

- Cohesion
  - A measure of how strongly-related the responsibilities of one element (method, class, module) are.
- High cohesion class
  - The methods of the same class are similar in many aspects
  - A highly-cohesive system, code readability and reuse is increased, while complexity is kept manageable.
- Low cohesion class
  - The methods in the same class have little in common.
  - Methods carry out many varied activities, often using unrelated sets of data



Mock Objects

```
void CreateAccount() {
   ...
   if (isGoodAccount) {
      // Send an email to notify that the account is
      //    created successfully
      MailMessage msg = new MailMessage();
      msg.To.Add("someone@ntut.edu.tw");
      msg.Subject = "Subject line";
      msg.From = new MailAddress("From@a.b");
      msg.Body = "This is the message body";
      SmtpClient smtp = new SmtpClient("smpt.ntut.edu.tw");
      smtp.Send(msg);
   } else {
      ...
   }
}
```

External service
1. Difficult to assert
2. Slow
3. May not be available
4. Should not be used during testing

Mock Objects

```
class X {
   ...
   void CreateAccount() {
      ...
      ...
      smtp.Send(msg);
      ...
      ...
   }
};
```

```
// Method #1: skip testing
//   by extracting logics
//   into helper methods
class X {
   ...
   void CreateAccount() {
      msg = foo1();
      smtp.Send(msg);
      foo2();
   }
   message foo1() {
      ...
   }
   void foo2()
   {
      ...
   }
};
```

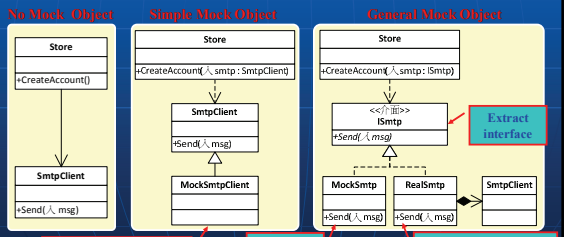
Unit Test

Unit Test

Disadvantages:
1. Works only if CreateAccount() can be made very very short.
2. foo() must be tested manually
3. Changing foo() is dangerous

Mock Objects

- Method #2: mock object
  - A mock object simulates the behavior of a real object
  - A mock object has the same interface as the real object



Mock Objects

- When is a mock object useful?
  - An object supplies non-deterministic results (e.g. the current time, random number, etc.);
  - An object has states that are difficult to create or reproduce (e.g. a network error, disk full);
  - An object that runs slowly (e.g. CD, a complete database)
  - An object that supports a service which may not always be available
  - An object that supports a service which cannot be used during testing
  - An object that is not available yet (TDD)



Mock Objects

```
void CreateAccount(ISmtp smtp) {
   ...
   if (isGoodAccount) {
      // Send an email
      //    created suc
      MailMessage msg =
      msg.To.Add("someo
      msg.Subject = "Su
      msg.From = new Ma
      msg.Body = "This
      smtp.Send(msg);
   } else {
      ...
   }
}
...
   CreateAccount(new RealSmtp());
```

Modify production code

```
interface ISmtp
{
   void Send(MailMessage msg);
}

class RealSmtp : ISmtp
{
   SmtpClient smtp = new
      SmtpClient("smtp.ntut.edu.tw");
   public void Send(MailMessage msg)
   {
      smtp.Send(msg);
   }
}
```

No logics

Mock Objects

```
class MockSmtp : ISmtp
{
    public MailMessage msg;
    public void Send(MailMessage msg)
    {
        this.msg = msg;
    }
}

void testCreateAccount()
{
    MockSmtp mockStmp = new MockSmtp();
    ...
    x.CreateAccount(mockSmtp);
    assert.AreEqual(mockSmtp.msg, ...);
    ...
}
```

Mock object

public

Store msg to enable assertion

Use mockSmtp during testing

Assert the state of the mock object

Exercise #5

Writing a Mock Object

- ATMModel.transfer revisited
- Use FakeTimeStamper
- Implement MockLogger
- Assert the correctness of log messages

```
Sample log:
2012-01-9 17:43:11 [Info] User (123-456-789) succeeded in transfer 100.0.
2012-01-9 17:43:21 [Warning] User (123-456-789) failed to transfer 4000.0.
```


Conclusions

Best Practice

- **Testing Idioms**
  - Code a little, test a little, code a little, test a little…
    - Unit testing helps improve the design along the way

I don't have time writing unit tests

帝問岳飛曰：…「臣有二馬，…，

良馬對

…初不甚急，比行百里，始奮迅，自午自酉，猶可兩百里，…不息不汗，若無事然。…，致遠之材也。

…今所乘者，…，攬轡未安，踴躍疾驅，甫百里，力竭汗喘，殆欲斃然。…，駑鈍之材也。」

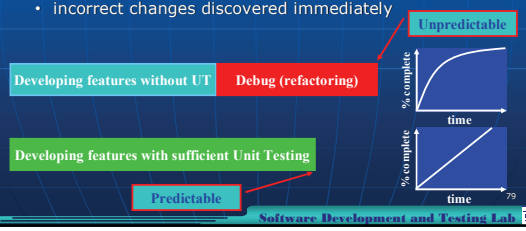
I don't have time writing unit tests

- Is quality important?
- Do you have time to fix your mistakes later?
- What happens when a developer rushes a feature?
  - Skip testing
  - Dirty code → bad design
    - long method, large class, …
  - No exception handling
  - Implement only partial (special) path
  - Return a fixed answer
  - Return without doing anything
- A moment's relief can bring endless suffering to come

Developers' secret weapon?

Advantages of Unit Testing

- Speed-up the development
  - Write test once, use test to find errors many times
  - Save the testing time of manipulating UI
- Regression tests
  - incorrect changes discovered immediately



Best Practice

- **How often should I run my tests?**
  - Run all your unit tests as often as possible
    - Ideally every time the code is changed.
    - Make sure all your unit tests always run at 100%.
  - For larger systems
    - Run specific test suites related to the code you're working on.
    - Run all the tests at least once per day (or night).
- **What do I do when a defect is reported?**
  - Write a failing test that exposes the defect
    - When the test passes, you know the defect is fixed!
  - You may need to be more aggressive about testing everything that could reasonably break.

Best Practice

- **My unit tests are slow.**
  - Unit testing should be fast
  - Test data should be fabricated → real-world data are for integration testing, which can be slow
- **Test File (resource)**
  - Create test file on the run
  - Put test file in the test project
- **Bad smells in test code**
  - Refactoring test code necessary
- **Production code : test code ratio?**
  - The best 2010 POSD HW was 4692: 6886 (1 : 1.46)
- **No more one red light at a time**

Best Practice

- **Why not just use print?**
  - Output must be scanned manually - inefficient
  - Makes code ugly
  - Generates too much information
  - Tests should retain its value over time
- **Why not just use a debugger?**
  - The same as that of using print.

If you find yourself testing using println() or debugger write a test case instead.

Why not just use print?

```
bool isPrime(int x)
{
   ...
}
```

```
void foo()
{
   ...
   if (isPrime(i))
        cout << i << "is prime" << endl;
   else
        cout << i << "not a prime" endl;
   ...
}
```

Using cout to make sure that isPrime() is correctly implemented

Why not just use print?

```
bool isPrime(int x)
{
   ...
}
```

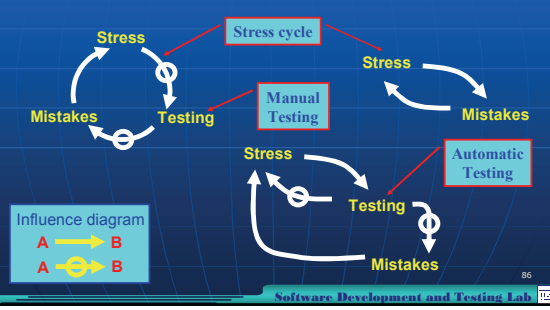
Use ASSERT instead of your eyes

```
void isPrimeTest()
{
   int prime[]={2, 3, 5, 7, 11, 97};
   for (int i = 0; i < 6; i++)
       CPPUNIT_ASSERT(isPrime(prime[i]));
   int nonprime[]={-1, 0, 1, 4, 9, 77};
   for (int i = 0; i < 6; i++)
       CPPUNIT_ASSERT(!isPrime(nonprime[i]));
}
```

Best Practice

- **Do I need to do unit testing for prototypes?**
  - No, if the prototype will be trashed
  - Yes, if the prototype will be turned into real products
- **Separate test code from production code**
  - Create a test project that tests the code in production code project
  - Eclipse: remember to refactor (e.g., rename) production code in the test project
- **I have a huge project (100,000 LOCs) without any unit tests. What do I do?**

Automated test breaks the stress cycle



Q & A