CS3343 Software Engineering Practice

2. Test Automation

Concepts

Software testing:

- Executing **input** data to a program and checking the program **output** against the expected output.
- Testing can only show the presence of errors, not their absence.

Why testing: to find as many bugs as possible in the software.

- Criteria for good testing:
  - **Effectiveness**: the ability of a test suite to detect faults in a program.
  - **Efficiency**: the ability of a test suite to detect faults in a program with minimum time and effort.

4 steps in software testing:

1. **Defining test objectives**
2. **Test design**
3. **Test execution**:
   - **Smoke test**: a preliminary test that reveals simple failures severe enough to reject a prospective software release.
   - **Regression test**: a test that is performed after the software has been changed to ensure that the changes have not affected the existing functionality.
4. **Test analysis**

JUnit

```
public class CalcTest {
    private Calculator calc;

    @Before
    public void setUp() {
        // Initialization code
        calc = new Calculator();
    }

    @After
    public void tearDown() {
        // Cleanup code
    }

    @Test // Each test will be executed in a separate instance of the test class
    public void testAdd() {
        assertEquals(5, calc.add(2, 3));
```

```
    }
}
```

3. Project Management

Mini trade-off triangle: **Scope**, **Time**, **Cost**

4 project dimensions:

- **People**: developer productivity, right people doing the right job
- Development **process**: following a software process
- **Product** focus: manage **code and documentation** with respect to the requirements
- **Technology**: use right **tools and design patterns**

Test-driven development (TDD):

- Every function developed must be tested.
- Write test cases before writing the code.
- **Schedule-oriented** practice.

Organizational Structures

- **Functional**: organized by functions performed.
    - e.g. Engineering Dept, Marketing Dept, etc.
    - Pros:
        - Clear definition of roles and responsibilities.
        - Clear career paths, encourages specialization.
    - Cons:
        - Communication between departments can be difficult.
        - Slow decision-making process.
- **Project-based**: organized by projects.
    - e.g. Project A, Project B, etc.
    - Pros:
        - Fast decision-making process.
        - Encourages teamwork.
    - Cons:
        - Duplication of facilities and resources.
- **Matrix**: a combination of functional and project-based structures.
    - e.g. Project A, Project B, etc. with Engineering Dept, Marketing Dept, etc.
    - Pros:
        - Both project integration and functional specialization.
        - Efficient use of resources.
    - Cons:
        - Complexities in managing the matrix.
        - Resource and priority conflicts.

Work Breakdown Structure (WBS)

Can be represented as a tree structure.

Uses a decimal numbering system to represent the levels of the hierarchy.

3 levels is usually sufficient. Example:

- 0.0 Whole Web Application (root)
- 1.0 Project Management
- 2.0 Requirement Analysis
- 3.0 Design
- 4.0 Implementation
    - 4.1 Frontend
        - 4.1.1 Framework, Libraries
        - 4.1.2 UI Design
        - 4.1.3 UX Design
    - 4.2 Backend
        - 4.2.1 Database
        - 4.2.2 Server
- 5.0 Testing and Deployment

4 techniques for creating WBS:

- Top-down approach
- Bottom-up approach
- Analogy (using a similar project as a reference)
- Brainstorming

Critical Path Method (CPM)

Once tasks are identified, they need to be scheduled.

Primary objectives of scheduling: least **time, cost, and risk**.

Critical path: the longest path through the network.

- EET (Earliest Event Time): the earliest time at which an event can occur (after all dependencies are satisfied).
- LET (Latest Event Time): the latest time at which an event can occur without delaying the project.
- Critical path: the path where all activities are critical, i.e. EET = LET.

4. Software Development Process

Refactoring

Code smells: symptoms of poor design. Part of the code that potentially indicates a deeper problem.

An example of code smell: **long method**.

```
    int processPlayerTurn (
        int a, int[] b, int c) {
        // manipulate GUI
        ...
        // compute score
        score = a;
        for (int i=0; i<b.length; i++)
            score += b[i] * c;
        ...
        return 1;
    }
```

Issues with above code:

- **Frontend and backend logic mixed**.
    - Manipulating GUI and computing score should be separated. Therefore, we can perform unit testing on the backend logic.
- **Variable names are not descriptive**.

Refactoring the code:

```
int processPlayerTurn (int curPt, int[] groupScore, int bonus) {
    updateGUI();
    ...
    score = computeBonus(curPt, groupScore, bonus);
    ...
    return 1;
}
int computeBonus(int initialScore, int[] groupScore, int bonus) {
    int score = initialScore;
    for (int i=0; i<groupScore.length; i++)
        score += groupScore[i] * bonus;
    return score;
}
```

Importance of Refactoring

- **Improves the design of software** incrementally.
    - Refactor dirty fixes into clean, maintainable code.
    - Remove unnecessary code.
- **Improves readability**.
    - Easier to understand, maintain, and extend.
- **Write program faster**.
    - Make modification on a clean codebase is easier.

Code Smell: Duplicate Code

Solution:

- **Extract method**: create a new method and move the duplicated code into it.
- **Pull up field**: move a field that is common to multiple classes to a superclass.
- **Form template method**: create a superclass that defines a template method.
- Substitute algorithm: choose the best algorithm for the job.

- Extract class: for unrelated responsibilities in a class.

Example of form template method:

```
abstract class Site {}
class ResidentialSite extends Site {
    void getBillableAmount() {
        // compute bill for residential site
    }
}
class LifelineSite extends Site {
    void getBillableAmount() {
        // compute bill for lifeline site
    }
}
```

After refactoring:

```
abstract class Site {
    void getBillableAmount() { return getBaseAmount() + getTax(); }
    abstract int getBaseAmount();
    abstract int getTax();
}
class ResidentialSite extends Site {
    int getBaseAmount() { }
    int getTax() { }
}
class LifelineSite extends Site {

    int getBaseAmount() { }
    int getTax() { }
}
```

Code Smell: Long Method

Solution:

- **Replace temp with query**: extract the expression into a method.

Before refactoring:

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

After refactoring:

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

- **Introduce Parameter Object**: group related parameters into a single object.

Before refactoring:

```
function amountInvoicedIn(start: Date, end: Date) {}
function amountReceivedIn(start: Date, end: Date) {}
function amountOverdueIn(start: Date, end: Date) {}
```

After refactoring:

```
interface DateRange {
    start: Date;
    end: Date;
}
function amountInvoicedIn(dateRange: DateRange) {}
function amountReceivedIn(dateRange: DateRange) {}
function amountOverdueIn(dateRange: DateRange) {}
```

- **Replace Method with Method Object**: when a method is too complex, create a new class to encapsulate the method.

Before refactoring:

```
double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long computation
    ...
    return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
}
```

After refactoring:

```
class PriceCalculator {
    PriceCalculator(Order order) { }
    double compute() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation
        ...
        return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
    }
}
// Usage
return new PriceCalculator(order).compute();
```

- **Decompose Conditional**: when a conditional statement is too complex, extract the condition into a method.

Before refactoring:

```
if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    return 29;
else
    return 28;
```

After refactoring:

```
if (isLeapYear(year))
    return 29;
else
    return 28;
boolean isLeapYear(int year) { return year % 4 == 0 && (year % 100 != 0 || year
% 400 == 0); }
```

When to Refactor

Refactor when:

- Large amount of code duplication.
- Before adding new features.
- Before fixing bugs.
- During code review.

DO NOT refactor when:

- During adding new features.
- During fixing bugs.
- No unit tests available.
- Code does not work.
- Deadline is near.

In these cases, refactor after the task is completed.

Steps to Refactor

1. **Write tests**: ensure that the code works before refactoring.
    - If not, fix the code first.
2. Look for **code smells**.
3. Refactor, test, and repeat.
4. Debug if there is any failure.

Example of refactoring:

```
enum MovieType {
    REGULAR, NEW_RELEASE, CHILDREN
}
public class Movie {
    private String _title; { get; }
    private MovieType _priceCode; { get; }
    Movie(String title, MovieType priceCode) { ... }
}
public class Rental {
    private Movie _movie; { get; }
    private int _daysRented; { get; }
```

```
    Rental(Movie movie, int daysRented) { ... }
}
public class Customer {
    private String _name; { get; }
    private ArrayList<Rental> _rentals; { get; }
    Customer(String name) { ... }
    public void addRental(Rental arg) { _rentals.add(arg); }
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        String result = "Rental Record for " + getName() + "\n";

        for (Rental each : _rentals) {
            double thisAmount = 0;
            switch (each.getMovie().getPriceCode()) {
                case REGULAR:
                    thisAmount += 2;
                    if (each.getDaysRented() > 2)
                        thisAmount += (each.getDaysRented() - 2) * 1.5;
                    break;
                case NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case CHILDREN:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    break;
            }
            frequentRenterPoints++;
            if (each.getMovie().getPriceCode() == NEW_RELEASE &&
each.getDaysRented() > 1)
                frequentRenterPoints++;
            result += "\t" + each.getMovie().getTitle() + "\t" +
String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }

        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + "
frequent renter points";
        return result;
    }
}
```

Steps to refactor:

1. Long method. Extract `amountFor(Rental)` method from `statement()`.
2. Since this method is only related to `Rental`, move it to `Rental` class.
3. Additionally, `frequentRenterPoints` can be moved to `Rental` class.

After refactoring:

```
public class Rental {
    ...
```

```
    double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            case REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                    result += (getDaysRented() - 2) * 1.5;
                break;
            case NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
            case CHILDREN:
                result += 1.5;
                if (getDaysRented() > 3)
                    result += (getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
    int getFrequentRenterPoints() {
        if (getMovie().getPriceCode() == NEW_RELEASE && getDaysRented() > 1)
            return 2;
        return 1;
    }
}
public class Customer {
    ...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        String result = "Rental Record for " + getName() + "\n";

        for (Rental each : _rentals) {
            frequentRenterPoints += each.getFrequentRenterPoints();
            result += "\t" + each.getMovie().getTitle() + "\t" +
String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + "
frequent renter points";
        return result;
    }
}
```

More Refactoring Techniques

1. **Encapsulate fields**: make fields private and provide getter and setter methods.
2. **Introduce null objects**: avoid null checks.

Before refactoring:

```
class Customer {
    private Plan _plan;
    Plan getPlan() { return _plan; }
}

// Usage
if (customer != null && customer.getPlan() != null)
    return customer.getPlan().getDiscount();
else
    return 0;
```

After refactoring:

```
class Customer {
    private Plan _plan;
    Plan getPlan() { return _plan; }
}
class NullPlan extends Plan {
    double getDiscount() { return 0; }
}
class NullCustomer extends Customer {
    Plan getPlan() { return new NullPlan(); }
}
```

3. **Parameterize method**: when two methods have similar code, extract the common code into a new method.
4. **Extract subclass**: when a class has two responsibilities, extract one of them into a subclass.

Before refactoring:

```
class JobItem {
    double getUnitPrice();
    double getTotalPrice();
    ArrayList<Employee> getEmployee();
}
```

After refactoring:

```
class JobItem {
    double getUnitPrice();
    double getTotalPrice();
}
class LaborItem extends JobItem {
    double getUnitPrice(); // if getUnitPrice() is different
    ArrayList<Employee> getEmployee();
}
```

5. **Extract interface**: when several classes have similar methods, extract the common methods into an interface.

Before refactoring:

```
class PhoneBill {
    double getCharge();
    double getDiscount();
}
class WaterBill {
    double getCharge();
    double getDiscount();
}
class HouseHoldBill {
    double getCharge();
    double getDiscount();
    double getTax();
}
```

After refactoring:

```
abstract class Billable {
    double getCharge();
    double getDiscount();
}
class PhoneBill extends Billable { }
class WaterBill extends Billable { }
class HouseHoldBill extends Billable {
    double getTax();
}
```

6. **Pull up method**: when two subclasses have similar methods, move the method to the superclass. Similar to extract interface.
7. **Replace type code with state/strategy**: when a class has a type code that affects its behavior, replace the type code with a state or strategy pattern.

Before refactoring:

```
class Customer {
    CustomerType _type;
    double getDiscount() {
        switch (_type) {
            case GREEN:
                return 0.02;
            case SILVER:
                return 0.03;
            case GOLD:
                return 0.05;
        }
    }
}
```

After refactoring:

```
class Customer {
    CustomerType _type;
    double getDiscount() { return _type.getDiscount(); }
}
```

```
abstract class CustomerType {
    abstract double getDiscount();
}
class GreenCustomer extends CustomerType {
    double getDiscount() { return 0.02; }
}
class SilverCustomer extends CustomerType {
    double getDiscount() { return 0.03; }
}
class GoldCustomer extends CustomerType {
    double getDiscount() { return 0.05; }
}
```

It is advisable to use inheritance to replace all `switch` statements.

8. **Replace inheritance with delegation**: when a subclass only uses a part of the superclass, replace inheritance with delegation.
9. **Hide delegate**: when a class delegates a lot of methods to another class, hide the delegate.

Before refactoring:

```
class Person {
    Department _department; { get; }
}
class Department {
    Person getManager() { return _manager; }
}
class Client {
    Person getManager() { return _department.getManager(); }
}
```

After refactoring:

```
class Person {
    Department _department; { get; }
    Person getManager() { return _department.getManager(); }
}
class Client {
    Person getManager() { return _person.getManager(); }
}
```

Bug Report

A bug report should contain the following information:

- **Situation**: (OS, component, platform, etc.)
- **Bug Description**:
    - Bug report title
    - Severity
    - Steps to reproduce
    - Expected result
    - Actual result

What quantify a good bug report:

- **Reproducibility**: the bug can be reproduced.
- **Specific**: a minimal set of steps to reproduce the bug.

How to fix a bug:

1. **Reproduce the bug**. Find simplest input that will always show the bug.
    - Start with data that first revealed the bug.
    - Binary search to find the smallest input that still shows the bug.
2. **Understand the bug**.
    - Try simple changes to see if they affect the bug.
    - Keep a record of what you have tried.
    - **Predict and display intermediate results**.
3. **Fix the bug**.
    - **Write a test** that fails because of the bug.
    - **Version control**: keep track of changes.

Five desirable properties of a bug report:

- **Reproducible**: The bug report includes a minimal set of steps to reproduce the bug, with specific resources mentioned or attached. (e.g. web page, minimal code, etc.)
- **Specific**: The bug report includes a clear description of the bug, with expected and actual results.
- **Importance**: The bug report includes the severity of the bug.
- **Actual Result**: The bug report includes the actual result of the bug, by providing screenshots, logs, etc. (rather than just saying "it doesn't work").
- **Location**: The bug report includes the location of the bug, such as the file, line number, etc.

Test Comprehensiveness

Test coverage: the percentage of code that is executed by the test suite.

Path: a **possible and complete** sequence of statements from an input to an output in a program.

- **Exhaustive testing**: testing all possible paths.
- **Random testing**

They are not sensible for functional testing. To test programs, we need to achieve **higher coverage**, and apply a **stronger coverage criterion** if necessary.

Control Flow Testing

To achieve 100% test coverage, after running a set of tests,

- **statement coverage**: every statement in the code is executed at least once.
- **branch coverage**: every branch in the code is executed at least once.
- **loop coverage**: every loop in the code is executed 0, 1, and more than 1 times, ignoring infeasible cases.
- **path coverage**: every path in the code is executed at least once.

From statement to path coverage, the cost increases. Path coverage is practically impossible to achieve.

Example of statement coverage but not branch coverage:

```
if (a >= 0 && a <= 9)
    sum = list1[a];
if (b >= 0 && b <= 9)
    sum = sum + list2[b];
```

Here, `a = 9, b = 3` will achieve 100% statement coverage but not branch coverage.

For branch coverage, both branches must have at least 1 test case that condition is true and false.

For path coverage, all of `True-True`, `True-False`, `False-True`, and `False-False` must be tested.

Branch coverage is weak but **low cost** to achieve. Bug detection capability increases generally, but it won't be useful until high coverage is achieved.

If low-level coverage is not good enough, we strengthen our test by using a stronger testing technique.

Predicate Coverage

To achieve 100% predicate coverage, every predicate in the code must be evaluated to both true and false.

- **Condition coverage** (CC): Each condition has been evaluated to both true and false.
- **Decision coverage** (DC): Each decision has been evaluated to both true and false.
- **C/DC**: Both CC and DC are satisfied.
- **MC/DC**: (1) Both CC and DC are satisfied, (2) each conditional is shown to independently affect the decision.

Since there is short-circuit evaluation in most programming languages, it is important to test all conditions.

Example:

```
while (PlayerHealth > 1 || !isCombat)
```

Truth Table:

Test Case	PlayerHealth	isCombat	`PlayerHealth > 1`	`!isCombat`	Result
t1	10	T	T	F	T
t2	20	F	T	T	T
t3	0	T	F	F	F
t4	1	F	F	T	T

For CC:

- we need `PlayerHealth > 1` to be true and false, at least once, and same for `!isCombat`.
- Therefore, `t1, t4` or `t2, t3` is sufficient.

For DC:

- we need result to be true and false, at least once.

- Therefore, `t1, t3`, `t2, t3`, or `t3, t4` is sufficient.

For C/DC:

- CC and DC must be achieved on the same test case.
- Therefore, `t2, t3` or `t1, t3, t4` is sufficient.

For MC/DC:

- Upon changing `PlayerHealth > 1` or `!isCombat`, with other conditions fixed, the result must change.
- For `t3, t4`, `PlayerHealth > 1` is false, and result changes with `!isCombat`.
- For `t1, t3`, `!isCombat` is false, and result changes with `PlayerHealth > 1`.
- Therefore, `t1, t3, t4` is sufficient.
- MC/DC is a subset of C/DC. i.e. MC/DC is always C/DC, but not vice versa.
- Here, `t2, t3` is C/DC but not MC/DC; `t1, t3, t4` is C/DC and MC/DC.

An **independent path** (**basis path**) is a complete path that introduces at least a new condition or a new statement relative to previous paths, i.e. at least one branch that has not been covered by other paths.

Number of basis paths: $V(G) = E - N + 2$, where E is the number of edges, N is the number of nodes.

Example (Tutorial 9)

```
bool isFullHouse(const string cards[], int n) {
    return isThreeOfaKind(cards, n) && isTwoPairs(cards, n);
}

bool isThreeOfaKind(const string cards[], int n) {
    for (int i=0; i<n-2; i++) {
        if (cards[i][1] == cards[i+1][1] && cards[i+1][1] == cards[i+2][1])
                return true;
    }

    return false;
}

bool isTwoPairs(const string cards[], int n) {
    int count = 0;
    for (int i=0; i<n-1; i++) {  // s1
        if (cards[i][1] == cards[i+1][1]) { // s2
            count++; // s3
            i++;
        }
    }
    if (count == 2) // s4
        return true; // s5
    else
        return false; // s6
}
```

Testing `isFullHouse`:

Test Case	Input	Expected Output	Path
1	`{}`	F	F-/(F)
2	`{"C2", "D2", "H2", "S3", "S4"}`	F	T-F
3	`{"DJ", "SJ", "CK", "DK", "HK"}`	T	T-T
4	`{"C3", "D3", "S3", "HX", "SX"}`	T	T-T
5	`{"C2", "D2", "C3", "D3", "C4"}`	F	F-/(T)
6	`{"CA", "C2", "C3", "C4", "C5"}`	F	F-/(F)
7	`{"C6", "D6", "H6", "S6", "S7"}`	F	T-T

`/` indicates a predicate that is not covered due to short-circuit evaluation.

Here are test case combinations:

```
{1, 2}
{1, 3}
{4, 6}
{2, 5, 6}
{2, 4, 5, 6}
{1, 2, 3, 4, 5, 6, 7}
```

1. For CC
   - `{1, 2}` is insufficient because `isTwoPairs` is either `/` or `F`, that is, never `T`.
   - `{1, 3}` is insufficient because `isTwoPairs` is either `/` or `T`, that is, never `F`.
   - `{4, 6}` is insufficient because `isTwoPairs` is either `/` or `T`, that is, never `F`.
   - `{2, 5, 6}` is insufficient because `isTwoPairs` is either `/` or `F`, that is, never `T`.
   - `{2, 4, 5, 6}` is sufficient.
   - `{1, 2, 3, 4, 5, 6, 7}` is sufficient.
2. For DC
   - `{1, 2}` is insufficient because `isFullHouse` is always `F`, never `T`.
   - `{1, 3}` is insufficient because `cards[i][1] == cards[i+1][1]` in `isTwoPairs` is always `T` (or unexecuted), never `F`. (Note that `i++` skips the next card, if the current and next cards form a pair.)
   - `{4, 6}` is insufficient because `count == 2` in `isTwoPairs` is always `T` (or unexecuted), never `F`.
   - `{2, 5, 6}` is insufficient because `isFullHouse` is always `F`, never `T`.
   - `{2, 4, 5, 6}` is sufficient.
   - `{1, 2, 3, 4, 5, 6, 7}` is sufficient.
3. For C/DC
   - `{2, 4, 5, 6}` and `{1, 2, 3, 4, 5, 6, 7}`.
   - C/DC test cases are the common subset of CC and DC test cases.
4. For MC/DC
   - Same as C/DC.
   - **In language with short-circuit evaluation, MC/DC is equivalent to C/DC.** (Why? changing the value of a condition will immediately change the result.)

5. For statement coverage
    - Same as C/DC.
    - `{2, 4, 5, 6}` covers all statements, and so does its superset.
6. For branch coverage
    - Same as C/DC.
    - Branch coverage is a superset of statement coverage. All branch coverage test cases are also statement coverage test cases.
7. For path coverage
    - None of the test cases achieve path coverage, because there are more than 7 paths in the code.

Paths in `isTwoPairs`:

- Path 1: `s1, s4, s6`
- Path 2: `s1, s2, s1, s4, s6`
- Path 3: `s1, s2, s3, s1, s4, s5`

Example 2

Part of Sudoku solver:

```
bool isValid(int board[9][9], int row, int col, int num) {
    if (!isRowValid(board, row, num) || !isColValid(board, col, num))
        return false;
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            if (board[i+startRow][j+startCol] == num)
                return false;
    return true;
}
bool isRowValid(int board[9][9], int row, int num) {
    for (int i=0; i<9; i++)
        if (board[row][i] == num)
            return false;
    return true;
}
bool isColValid(int board[9][9], int col, int num) {
    for (int i=0; i<9; i++)
        if (board[i][col] == num)
            return false;
    return true;
}
```

Construct test cases to achieve 100% branch coverage. Assume we are testing `row = 0, col = 0, num = 1`.

Test Case	Input	Expected Output	Path
T1	Row 1 = .12345678	F	isRowValid = F

Test Case	Input	Expected Output	Path
T2	Row 1 = .23456789, Col 1 = .12345678	F	isRowValid = T, isColValid = F
T3	Row 1 = .23456789, Col 1 = .23456789, Box 1 = .12345678	F	isRowValid = T, isColValid = T, isValid = F
T4	Row 1 = .23456789, Col 1 = .23456789, Box 1 = .23456789	T	isRowValid = T, isColValid = T, isValid = T

Example 3

```
public void CreditCheck(int Purchase, int LastPayment, int Balance, boolean
badRecord) {

    boolean GoodRecord = (LastPayment < 30) && (!badRecord);

    if (GoodRecord) {
        if (Purchase < 500) {
            if (Balance < 2000)
                Print("Credit OK.");
            else
                Print("Refer to Credit Dept (high balance).");
        } else {
            if (Balance < 500)
                Print("Refer to Credit Dept (high purchase).");
            else
                Print("Credit Denied (high balance).");
        }
    } else {
        Print("Credit Denied.");
    }
}
```

Test Case	Input	Expected Output	Path
1	2000, 35, 2000, F	Credit Denied	`(LastPayment < 30)` = F, `GoodRecord` = F
2	2000, 25, 2000, T	Credit Denied	`(LastPayment < 30)` = T, `!badRecord` = F, `GoodRecord` = F
3	100, 25, 1000, F	Credit OK.	... `GoodRecord` = T, `Purchase < 500` = T, `Balance < 2000` = T
4	100, 25, 2000, F	Refer to Credit Dept (high balance).	... `GoodRecord` = T, `Purchase < 500` = T, `Balance < 2000` = F
5	600, 25, 400, F	Refer to Credit Dept (high purchase).	... `GoodRecord` = T, `Purchase < 500` = F, `Balance < 500` = T

Test Case	Input	Expected Output	Path
6	600, 25, 2000, F	Credit Denied (high balance).	... `GoodRecord` = T, `Purchase < 500` = F, `Balance < 500` = F

Test Organization

An OOP program being tested can be viewed as a listing of classes $\{c_1, c_2, \ldots, c_n\}$.

Methodology of test organization:

- Select **the combinations of classes** to be tested, and **identify the focus** of the test.
- Test these combinations systematically by **varying the levels of detail**.

Testing Level	Description
Unit Test	Test individual **classes**.
Integration Test	Test the interaction among a subset of classes, **components**. Test the interaction among a subset of components, **subsystems**.
System Test	Test the entire **system**.
Acceptance Test	Test whether the product meets the customer's requirements.

White-box and black-box testing:

- **White-box testing**: test the internal structure of the program.
  - Developers can see and use the source code to create and check test cases.
  - Infeasible to test large programs comprehensively.
  - Effective to detect **implementation bugs**.
  - Ineffective to detect implementation discrepancies.
- **Black-box testing**: test the program against use case scenarios.
  - Testers can only see the input and output of the program, and reply on specifications to design test cases.
  - Scalable to large programs.
  - Less effective, but can detect whether use case scenarios are met.

We usually use white-box testing at the unit and integration levels, and black-box testing at the system and acceptance levels.

Integration Testing

- **Big Bang**
  - Unit Testing: test all classes individually.
  - No Integration Testing
  - System Testing
- **Top Down**
  - Unit Testing: test main class first.
  - Integration Testing: test classes from top to bottom, layer by layer.
  - System Testing

- **Bottom Up**
  - Unit Testing: test leaf classes first.
  - Integration Testing: test classes from bottom to top, layer by layer. No Stub needed.
  - System Testing
- **Modified Top Down**
  - Unit Testing: test all classes individually.
  - Integration Testing: test classes from top to bottom, layer by layer.
  - System Testing

Example

```
public class Main {
    public static void main(String[] args) {
        D.doSomething();
    }
}
class D {
    public static void doSomething() {
        A.doSomething();
        G.doSomething();
    }
}
class A {
    public static void doSomething() {
        J.doSomething();
    }
}
```

1. Big Bang Approach
   - Unit Testing 1: Main (stub: D)
   - Unit Testing 2: D (stub: A, G)
   - Unit Testing 3: A (stub: J)
   - Unit Testing 4: G
   - Unit Testing 5: J
   - Integration Testing: N/A
   - System Testing: Main + D + A + J + G
2. Top Down Approach
   - Unit Testing 1: Main (stub: D)
   - Integration Testing 1: Main + D (stub: A, G)
   - Integration Testing 2: Main + D + A + G (stub: J)
   - System Testing: Main + D + A + J + G
3. Bottom Up Approach
   - Unit Testing 1: J
   - Unit Testing 2: G
   - Integration Testing 1: A + J
   - Integration Testing 2: D + A + J + G
   - System Testing: Main + D + A + J + G
4. Modified Top Down Approach
   - Unit Testing 1: Main (stub: D)
   - Unit Testing 2: D (stub: A, G)

- Unit Testing 3: G
- Unit Testing 4: A (stub: J)
- Unit Testing 5: J
- Integration Testing 1: Main + D (stub: A, G)
- Integration Testing 2: Main + D + A + G (stub: J)
- System Testing: Main + D + A + J + G