CS3343 Software Engineering Practice - Review

Project Management

Project Management (Lecture 3)

4 functions of Project Manager (PM): Plan, Schedule, Track, Measure

5 success factors of a project: schedule, budget, product, quality, and customer satisfaction

4 Project Dimensions:

- **People**: productivity
- **Development Process**: follow a software development process
- **Product Focus**: manage code and documentation
- **Technology**: use tools and design patterns

McConnell's Anti-Patterns (examples):

- People related: weak personnel, heroics, adding people to a late project, unrealistic expectations, wishful thinking
- Process related: optimistic schedules, wasted time, insufficient risk management, contractor failure, shortchanged quality
- Product related: requirements gold-plating, feature creep, developer gold-plating
- Technology related: silver-bullet syndrome, overestimated savings from new tools, switching tools in the middle of a project

Organizational Structure

- **Functional**: each department is responsible for a specific function
    - Pros: clear career paths, specialization, eliminates duplication
    - Cons: not enough communication, slow decision making, resource conflicts
- **Project-based**: teams are formed for a specific project
    - Pros: unity of command, fast inter-project communication
    - Cons: duplication of resources, unclear career paths
- **Matrix**: a combination of functional and project-based
    - Pros: project integration across functional lines, resource utilization, retains functional specialization
    - Cons: two bosses, complexity, resource conflicts

Schedule (Lecture 4)

Objectives of scheduling: best time, least cost, least risk

How to schedule:

- What needs to be done
- Size of the task
- Dependency between tasks

- Estimate total duration

Work Breakdown Structure (WBS): a hierarchical decomposition of the total scope of work to be carried out by the project team to accomplish the project objectives and create the required deliverables. Can be shown as a table (Gantt Chart) or a tree diagram

Gantt Chart: a bar chart that shows the start and finish dates of elements of a project. Hierarchical

4 ways to produce WBS: top-down, bottom-up, analogy, brainstorming

Critical Path Method (CPM): a project management technique used to determine the critical path and calculate the minimum project duration. CPM = Longest full path

Earlist Event Time (EET) = Max(EET of previous event) + time to complete

Latest Event Time (LET) = Min(LET of next event) - time to complete

critical path = a chain of tasks that EET = LET

Testing

Test Automation (Lecture 2)

What is software testing: executing input data to a program to check if the output is as expected

Objectives of testing: find as many bugs as possible, at the lowest cost, within least time

Procedure of testing: defining test objectives, test design, test execution, test analysis

- Smoke Testing: a preliminary test to reveal simple failures severe enough to reject a prospective software release. Used in nightly builds and continuous integration

Test stubs: a piece of code that mimics the behavior of a component that is not yet implemented. Used in top-down testing that tests higher-level modules first

Junit 4

```
class TestClass {
    @BeforeClass
    public static void setUpBeforeClass() {
        // code to run before all tests
    }
    @Before
    public void setUp() {
        // code to run before each test
    }
    @After
    public void tearDown() {
        // code to run after each test
    }
    @Test
    public void testMethod() {
        // test code
        assertEquals(expected, actual);
    }
}
```

JUnit 5

```
class TestClass {
    @BeforeAll
    public static void setUpBeforeClass() { }
    @BeforeEach
    public void setUp() { }
    @AfterEach
    public void tearDown() { }
    @Test
    public void testMethod() {
        assertEquals("message", expected, actual); // you can add a message to
display when the test fails
    }
}
```

P.S. How to use string I/O in Java:

```
System.setIn(new ByteArrayInputStream("input".getBytes()));
ByteArrayOutputStream outContent = new ByteArrayOutputStream();
System.setOut(new PrintStream(outContent));
```

Test Coverage (Lecture 8, 9)

Exhaustive Testing: testing all possible inputs (infeasible)

Code coverage: the percentage of code that has been executed by the test suite

Type	Description
Statement Coverage	Each statement is executed at least once
Branch Coverage	Each branch is executed at least once
Loop Coverage	Each loop is executed 0, 1, and more than 1 times
Path Coverage	Each path is executed at least once
CC	Each condition is evaluated to true and false
DC	Each decision is evaluated to true and false
C/DC	CC + DC
MC/DC	C/DC + each condition affects the decision outcome independently

Path: a possible and complete sequence of statements, from an entry point to an exit point

Statement coverage <= Branch coverage <= Path coverage

Branch coverage is weak but low cost; bug detection capability increases generally, but it won't be useful until high coverage is achieved

Basis path (independent path): a complete path that introduces at least a new condition or a new statement compared to other paths

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

Test Organization (Lecture 10)

Methodology: select **the combinations** of classes to test, and identify the focus of the test; test systematically by **changing the levels of detail**

Test Level	Description
Unit Testing	Test individual classes
Integration Testing	Test the integration of classes to a subsystem
System Testing	Test the entire system
Acceptance Testing	Test the system with the customer

- **White-box Testing**: test the system against the code
  - effective to detect implementation bugs
- **Black-box Testing**: test the system against the use case scenarios
  - effective to detect unimplemented requirements

Test types:

- **Top-down Testing**: High-level unit testing first
- **Bottom-up Testing**: Low-level unit testing first
- **Modified Top-down Testing**: Top-down testing with each class unit-tested
- **Sandwich Testing**: Top-down and bottom-up testing meet in the middle

Debugging

Bug Reporting (Lecture 7)

Bug report should include:

- Basic information: title, severity, priority, status, version / build, component, platform / environment (OS, browser, etc.)
- Description
  - Steps to reproduce
  - Expected result
  - Actual result (stack trace)

A good bug report should be:

- Reproducible: clear steps to reproduce the bug
- Specific: look into the code to locate the bug (e.g. what part of the webpage causes the crash), and provide it as a HTML snippet

Debugging (Lecture 11)

3 steps to debug: locate bug, repair program, retest program

Failure: an incorrect output or behavior of a program

Bug (fault): an incorrect implementation that causes a failure

Error: difference between the expected program state and an infected program state

Propagation from Bug to Failure: buggy statement propagates the error to the outputting statement; all statements after buggy statement transform infected state to a sequence of infected states (observed as a failure)

Hypothesis Testing: develop hypotheses to explain the failure, and test them to find the bug

Empirical observations -> Hypothesis -> Prediction -> Experiment -> Observation & Conclusion

Code Refactoring

Testing-Debugging-Refactoring Cycle (Lecture 5)

Objectives of code refactoring:

- improve the design of software incrementally
- make software more comprehensible (maintainable, extensible)
- write programs faster

Do code refactoring when:

- code is reused (copy-paste)
- before adding new features
- before fixing bugs
- during code review

Code Refactoring Techniques (Lecture 6)

Code smells: bad code that indicates a deeper problem

- duplicated code
- long method
- large class
- switch statement
- temporary field
- nested conditional

Refactoring techniques:

- Duplicate Code
    - Extract Method
    - Pull Up Field
    - Form Template Method
    - Substitute Algorithm
    - Extract Class
- Long Method
    - Replace Temp with Query

    - Introduce Parameter Object
    - Replace Method with Method Object
    - Decompose Conditional
- Others
    - Encapsulate Field
    - Introduce Null Object
    - Parameterize Method
    - Extract Subclass
    - Extract Interface
    - Pull Up Method
    - Replace Switch with State/Strategy
    - Replace Inheritance with Delegation
    - Hide Delegate

Past Paper

[1] Project Management

(1) What are the 4 most important Project Manager's Functions in a software project?

Planning, Scheduling, Tracking, Measuring

(2) What are the 3 key aspects to be considered as a Successful Project?

Budget, In-time delivery, Quality

(3) Please identify at least 5 reasons for Project Failures.

- Every-changing project scope
- Poor requirement gathering
- Lack of management support
- Unrealistic planning and scheduling
- Lack of resources
- Lack of required technical skills
- Ineffective team members

(4) What are the 4 Project Dimensions?

People, Development Process, Product Focus, Technology

(5) What are the 3 types of Organizational Structures? Choose one and explain its Pros and Cons.

- Functional
    - Pro: clear career paths, specialization, eliminates duplication
    - Con: not enough communication, slow decision making, resource conflicts
- Project-based
- Matrix

(6) Once tasks from the WBS (Work Breakdown Structure) and size and effort from project estimation are known, then we will perform scheduling. Then what are the Primary Objectives and Secondary Objectives for the project scheduling?

- Primary objectives: best time, least cost, least risk
- Secondary objectives: resource utilization, communication, evaluation of schedule alternatives

[2] Testing & Code Refactoring

```
package pokerExam;
public class FiveCards {
    Card[] cards;
    int length;
    PreCondition cond;

    public FiveCards(String[] strCards, PreCondition cond) {
        this.length = strCards.length;
        this.cards = new Card[length];
        for (int i = 0; i < length; i++) this.cards[i] = new
Card(strCards[i].charAt(0), strCards[i].charAt(1));
        this.cond = cond;
    }
    public int getLength();
    public Card getCard(int i);
    public boolean isFlush() {
        char suit = cards[0].getSuit();
        for (int i = 1; i < length; i++) if (cards[i].getSuit() != suit) return
false;
        return true;
    }
    public Category getCategory() {
        if (length != 5) return Category.INVALID;
        if (!cond.isSorted(cards)) // statement S1
            return Category.NOSORTED;
        if (isFlush()) return Category.FLUSH;
        return Category.NOTIMPLEMENTED;
    }
}
class Card {
    char suit; // S, H, D, C
    char rank; // 2-9, X, J, Q, K, A
    public Card(char suit, char rank);
    public char getSuit();
    public char getRank();
}
class Precondition {
    public boolean isSorted(Card[] fivrCards) {
        for (int i = 0; i < fiveCards.length - 1; i++) {
            Card cur = fiveCards[i];
            Card next = fiveCards[i + 1];
            if (cur.rank == 'A' && next.rank != 'A') return false; // statement
L1
            if (cur.rank == 'K' && (next.rank != 'A' && next.rank != 'K'))
return false;
            if (cur.rank == 'Q' && (next.rank != 'A' && next.rank != 'K' &&
next.rank != 'Q')) return false;
            if (cur.rank == 'J' && (next.rank != 'A' && next.rank != 'K' &&
next.rank != 'Q' && next.rank != 'J')) return false;
            if (cur.rank == 'X' && next.rank <= '9') return false;
            if (cur.rank <= '9' && next.rank < cur.rank) return false;
        }
        return true;
```

```
    }
}
```

(1) Write Four (4) unit test cases (in Java JUnit code) for FiveCards.getCategory() such that each test case will check one unique category, and one of these test cases must go through the statement Labelled as L1.

```
public class TestPoker {
    @Test
    public void testInvalid() {
        String[] input = {"S2"};
        PreCondition cond = new PreCondition();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.INVALID, actual);
    }

    @Test
    public void testNotSorted() {
        String[] input = {"HA", "S2", "D3", "C4", "H5"}; // goes through L1
        PreCondition cond = new PreCondition();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.NOSORTED, actual);
    }

    @Test
    public void testFlush() {
        String[] input = {"HX", "HJ", "HQ", "HK", "HA"};
        PreCondition cond = new PreCondition();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.FLUSH, actual);
    }

    @Test
    public void testNotImplemented() {
        String[] input = {"S2", "H3", "D4", "C5", "H6"};
        PreCondition cond = new PreCondition();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.NOTIMPLEMENTED, actual);
    }
```

(2) Write Two (2) unit test cases for FiveCards.getCategory() such that each test case goes through a unique path with executing the statement cond.isSorted(cards) Labelled as S1. Your test case invokes your test stub of the PreCondition.isSorted(Card []). (test stub is required)

```
public class TestPoker {
    @Test
    public void testSorted() {
        class PreConditionStub extends PreCondition {
            @Override
            public boolean isSorted(Card[] fiveCards) {
```

```
                for (int i = 0; i < fiveCards.length - 1; i++) {
                    if (fiveCards[i].getRank() > fiveCards[i + 1].getRank())
return false;
                }
                return true;
            }
        }
        String[] input = {"S2", "S3", "S4", "S5", "S6"};
        PreCondition cond = new PreConditionStub();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.FLUSH, actual);
    }

    @Test
    public void testNoSorted() {
        class PreConditionStub extends PreCondition {
            @Override
            public boolean isSorted(Card[] fiveCards) {
                for (int i = 0; i < fiveCards.length - 1; i++) {
                    if (fiveCards[i].getRank() > fiveCards[i + 1].getRank())
return false;
                }
                return true;
            }
        }
        String[] input = {"S6", "S5", "S4", "S3", "S2"};
        PreCondition cond = new PreConditionStub();
        FiveCards obj = new FiveCards(input, cond);
        Category actual = obj.getCategory();
        assertEquals(Category.NOSORTED, actual);
    }
}
```

(3) Code Smell:
In the function isSorted(Card []), class PreCondition directly operate on the attributes of class Card to determine its execution control flow. This design will propagate the change of class Card to the behaviour of class PreCondition. We expect to keep good encapsulation between different classes.

Remove the above code smell via the “Extract a method” code refactoring strategy. Conditions that Your Submitted Code Must Satisfy:

1. Only the object of class Card itself can access its attribute directly.
2. The function isSorted(Card []) keeps the same interface .

```
class Card {
    public boolean isGreater(Card other) {
        if (this.rank == 'A' && other.rank != 'A') return true;
        if (this.rank == 'K' && (other.rank != 'A' && other.rank != 'K')) return
true;
        if (this.rank == 'Q' && (other.rank != 'A' && other.rank != 'K' &&
other.rank != 'Q')) return true;
        if (this.rank == 'J' && (other.rank != 'A' && other.rank != 'K' &&
other.rank != 'Q' && other.rank != 'J')) return true;
```

```
        if (this.rank == 'X' && other.rank <= '9') return true;
        if (this.rank <= '9' && other.rank < this.rank) return true;
        return false;
    }
}

class PreCondition {
    public boolean isSorted(Card[] fiveCards) {
        for (int i = 0; i < fiveCards.length - 1; i++) {
            if (fiveCards[i].isGreater(fiveCards[i + 1])) return false;
        }
        return true;
    }
}
```

[3] Bug Report

```
Bug 1063894
Summary: After Flash crash, reload page does not work, and other Flash tabs are
dead.
Product: [Components] Core
Component: Plug-ins
Status: UNCONFIRMED ---
Priority: --
Severity: normal
Reporter: Jake <jake_hotson>
Assignee: Nobody; OK to take it and work on it <nobody>
Version: 31 Branch
Hardware: x86_64
OS: Windows 7

User Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101
Firefox/31.0 Build ID:
20140716183446
Steps to reproduce:
Used a web page with a buggy Flash app that 'crashed' the Flash plug-in.
Actual results:
Was told Flash had crashed, invited to send bug report, and to reload the page to
resolve.
Upon reloading page, cursor flickered between APPLOADING and ARROW several times,
but page did not reload.  Other tabs using Flash were dead.
Expected results:
Pages and Flash should have reloaded.  According to the instructions on the
'crash page' it should not be necessary to restart Firefox itself to resolve.
A Flash crash on one tab should not have affected Flash apps on other tabs.
This has been an issue for several years and it's about time it's fixed.
```

(1) Is the reported case reproducible?

Answer: Yes. It provides a clear set of steps to reproduce the bug, that is, after the Flash plug-in crashes, the page does not reload and other Flash tabs are dead. It can be replicated without knowing the specific Flash app that caused the crash.

(2) Is the reported case specific enough?

Answer: Yes. The bug report provides a detailed description of the issue, including the steps to reproduce, the actual results, and the expected results. However, it could have included traces or logs from the browser console to provide more technical details.

(3) What makes a good bug report?

- Clear and concise description of the issue
- Steps to reproduce the bug
- Expected and actual results
- Environment details (browser version, OS, hardware)
- Error messages or logs
- Impact explanation (severity and priority)

[4] Project Management

(1) In your recent CS3343 software project, which integration testing approach was employed in efforts to expose software bugs?

Project Title: Hong Kong Journey Planner

Description: This is a console-based application that helps users plan their journey in Hong Kong. User inputs the starting and ending date, area, budget, and preferences, and the application will provide a list of recommended places for the user to choose. After choosing the places, the application will generate a detailed itinerary for the user, minimizing the travel time and cost.

Approach: Bottom-up Testing

Justification: There are many components in the project, such as model/dataloader, recommender, route generator, and I/O handler. As such, there are many individual leaf-level classes, which are tested first before moving up to the higher-level components. This approach ensures that the individual components are working correctly before integrating them into the system, and also does not involve the use of stubs or drivers.

(2) There are two types of WBS: Process-oriented and Product-oriented. With your CS3343 group project in mind, choose a WBS type, draw a sample WBS diagram with specific tasks

Type: Process-oriented

#	Task	Predecessor
A	Decide on project topic	-
B	Gather requirements	A
C	Scrape data	B
D	Prototyping the software	B
E	Implement restaurant recommender	C, D
F	Implement scenic spot recommender	C, D
G	Implement route generator	C, D
H	Merge components	E, F, G
I	Unit Testing	E, F, G

#	Task	Predecessor
J	Integration Testing	H, I
K	System Testing	J
L	Release	K

More formal example:

#	Task	Predecessor
A	Define the features of the application	-
B	Sketch user stories	A
C	Relevant research on existing applications	A
D	Formally define functional requirements	B
E	Implement the backend	D
F	Write test cases	C
G	Implement the entire application with integration testing	E, F
H	Form a user evaluation (testing) group	C
I	Perform user acceptance testing	G, H

(3) Use Problem-Solution-Report (PSR) to describe a problem you encountered in your CS3343 project, one for each of the four dimensions of project management.

(Report: How could you avoid each of them when you are given a chance to redo the same project? Suggest one report that your team has prepared, or one tool that your team has used that may help to solve the problem.)

- People
    - Problem: Team members have different schedules and availability, making it challenging to coordinate meetings and work sessions.
    - Solution: Implement a shared scheduling tool to coordinate team meetings and deadlines. Use communication channels like Discord to keep everyone informed and engaged.
    - Report: Weekly progress reports to track individual contributions and identify potential scheduling conflicts.
- Process
    - Problem: Lack of bug report guidelines and tracking system, leading to unstructured feedback and difficulty in prioritizing and resolving issues.
    - Solution: Use Bugzilla to track bugs and feature requests, assign tasks, and monitor progress.
    - Report: The documented bug guidelines and bug report template to standardize the reporting process.
- Product

    - Problem: No user manuals made it difficult for testers and users to understand the application's features and functionalities.
    - Solution: Generate a JavaDoc for the codebase and create user manuals with detailed instructions and screenshots.
    - Report: JavaDoc reference stored in webpages for easy access, and user manuals distributed to testers and users.
- Technology
    - Problem: Manual testing is error-prone and time-consuming, leading to missed bugs and delays in the testing process.
    - Solution: Implement automated testing tools like JUnit and Selenium to streamline the testing process and improve test coverage.
    - Report: Test reports generated by automated tools to track test results and identify areas for improvement.

[5] Testing & Debugging

```
public class Poker {
    public boolean isFullHouse(String cards[], int n) {
        return isThreeOfaKind(cards, n) && isTwoPairs(cards, n);
    }
    public boolean isThreeOfaKind(String cards[], int n) {
        for (int i = 0; i < n - 2; i++)
            if (cards[i].charAt(1) == cards[i + 1].charAt(1) && cards[i +
1].charAt(1) == cards[i + 2].charAt(1))
                return true; // statement A
        return false; // statement B
    }
    public boolean isTwoPairs(String cards[], int n) {
        int count = 0;
        for (int i = 0; i < n - 1; i++)
            if (cards[i].charAt(1) == cards[i + 1].charAt(1)) count++;
        if (count == 2) return true; // statement C
        return false; // statement D
    }
}
```

(1) Achieve loop coverage for the isThreeOfaKind() method.

```
public class PokerTest {
    @Test
    public void testIsThreeOfaKind1() {
        Poker poker = new Poker();
        String[] cards = {"S2", "H2", "D2", "C3", "H4"}; // loop 3 times
        boolean actual = poker.isThreeOfaKind(cards, 5);
        assertEquals(true, actual); // goes through statement A
    }

    @Test
    public void testIsThreeOfaKind2() {
        Poker poker = new Poker();
        String[] cards = {"S2", "H2", "D3", "C3", "H4"}; // loop 1 time
        boolean actual = poker.isThreeOfaKind(cards, 5); // true on 1st
predicate, false on 2nd
```

```
        assertEquals(false, actual); // goes through statement B
    }

    @Test
    public void testIsThreeOfaKind3() {
        Poker poker = new Poker();
        String[] cards = {"S2", "H3", "D4", "C5", "H6"}; // loop 1 time
        boolean actual = poker.isThreeOfaKind(cards, 5); // false on 1st
predicate
        assertEquals(false, actual); // goes through statement B
    }

    // above achieves branch coverage

    @Test
    public void testIsThreeOfaKind4() {
        Poker poker = new Poker();
        String[] cards = {"S2"}; // loop 0 times
        boolean actual = poker.isThreeOfaKind(cards, 1);
        assertEquals(false, actual); // goes through statement B
    }
}
```

(2) isFullHouse() is bugged because it will return true if the hand is four of a kind. Explain how to systmatically locate the bug.

a) Initial Hypothesis

- Hypothesis: IsFullHouse() runs correctly
- Prediction: IsFullHouse({"S2", "H2", "D2", "C2", "H4"}, 5) returns false
- Experiment: Run the test case
- Observation: Result is true
- Conclusion: Hypothesis is rejected

b) Hypothesis on IsThreeOfaKind()

- Hypothesis: The infection takes place in IsThreeOfaKind()
- Prediction: the above test case will fail
- Experiment: Run the test case
- Observation: All test cases pass
- Conclusion: Hypothesis is rejected

c) Hypothesis on IsTwoPairs() (similar)

d) Hypothesis on IsFullHouse()

- Hypothesis: The infection takes place when combining the results of IsThreeOfaKind() and IsTwoPairs()
- Prediction: IsFullHouse({"S2", "H2", "D2", "C2"}, 4) returns true
- Experiment: Run the test case
- Observation: Result is true
- Conclusion: Hypothesis is accepted, bug is located

(3) Fix the bug in the isFullHouse() method.

```
public boolean hasPair(String cards[], int n) { // has at least a pair, but not
three-of-a-kind
    for (int i = 0; i < n - 1; i++)
        if (cards[i].charAt(1) == cards[i + 1].charAt(1)) {
            if (i < n - 2 && cards[i].charAt(1) == cards[i + 2].charAt(1))
continue;
            return true;
        }
    return false;
}

public boolean isFullHouse(String cards[], int n) {
    return isThreeOfaKind(cards, n) && hasPair(cards, n);
}
```

Why it fixes the bug: now isFullHouse() won't consider the pair in a three-of-a-kind as a separate pair, ensuring that the hand is a full house and not a four of a kind.

(4) Write a specific Java code example to demonstrate your understanding of test stubs.

```
public class Calculator {
    private Logger logger;
    public Calculator(Logger logger) { this.logger = logger; }
    public int add(int a, int b) {
        int result = a + b;
        logger.log(a + " + " + b + " = " + result);
        return result;
    }
}
public class Logger {
    public void log(String message) { System.out.println(message); }
}

public class CalculatorTest {
    @Test
    public void testAdd {
        class LoggerStub extends Logger {
            private String message;
            @Override
            public void log(String message) { this.message = message; }
            public String getMessage() { return message; }
        }
        LoggerStub logger = new LoggerStub();
        Calculator calculator = new Calculator(logger);
        int result = calculator.add(2, 3);
        assertEquals(5, result);
        assertEquals("2 + 3 = 5", logger.getMessage());
    }
}
```

Another textbook example is to determine if the given year is a leap year, and compute the days in a month.

[6] Bug Report

```
Submitted:  15 Sep 2014 13:32
Reporter:   Viacheslav Romanov
Status: Open
Category:   MySQL Server: InnoDB storage engine
Version:    5.5.34
Severity:   S2 (Serious)
OS: Linux (Ubuntu)
CPU Architecture:   Any

[15 Sep 2014 13:32] Viacheslav Romanov
Description:
I am running a mysql server for almost a year, and a month ago it started to
freeze once in 2-3 days with bunch of semaphores waiting for the locks. I think
it is a bug in mysql because I do not use transactions - only single line queries
(selects or updates or inserts).

I have mysql server monitored by zabbix - and it has no clues what is going on.
I've tried to tweak several things - like now I switched off adaptive hash index,
but it have not helped yet.

How it can be I have deadlock without having transactions?

How to repeat:
I don't know, it occurs under the normal or high load. But recently it hangs only
at evening times - at the highest load level.

I am having server that runs a lot of simple request like this:
1) select * from users where userId = 222;
2) update users set name='dfdf' where userId = 333;
3) insert into users values (444, 'hello')

Number of rows inserted 4734264766, updated 27816634675, deleted 1054032407, read
801227142950
12.42 inserts/s, 83.50 updates/s, 0.03 deletes/s, 658.40 reads/

[log omitted]
```

(1) Is the reported case reproducible?

Answer: Not reproducible.

The reporter mentions that the issue occurs under "normal or high load" but does not provide specific steps or a consistent environment to reproduce the freeze. The fact that it only hangs during "evening times" adds variability, suggesting that external factors (like user activity) could influence the occurrence of the bug. Thus, without a clear and consistent method to replicate the issue, it remains uncertain if the case is reproducible.

(2) Is the reported case specific enough?

Answer: Yes.

he report provides some helpful context, such as the server's operational history, types of queries executed, and monitoring attempts (with Zabbix). It also includes the environment, MySQL version, logs, and the reporter's attempts to resolve the issue. However, the lack of specific steps to reproduce the bug and the absence of detailed error messages or logs could make it challenging for developers to pinpoint the root cause.

[7] Refactoring

```
public class Manager {
    public Manager() { }
    private static Manager instance = new Manager();
    public Contact lastAddedContact;
    public boolean addContact(Contact contact) {
        this.lastAddedContact = contact;
        boolean isNull = (null == contact);
        if (isNull) { System.out.println("{Manager} Contact is null"); return
false; }
        System.out.println("{Manager} Contact information:\n" + "Name\t:" +
contact.name + "\nSex\t:" + contact.sex + "\nAge\t:" + contact.age);
        return true;
    }
    public static Manager getManager() { return instance; }
}

public class Contact {
    public String name;
    public int age;
    public String sex;
    public Contact(String name, int age, String sex) { this.name = name;
this.age = age; this.sex = sex; }
    public static Contact createContact(String inStream) {
        Scanner sc = new Scanner(inStream);
        try { return new Contact(sc.nextLine(), Integer.parseInt(sc.nextLine()),
sc.nextLine()); }
        catch (Exception e) { return null; }
    }
}

public class Window {
    public void saveContact (String conInfo) {
        Manager myManager = Manager.getManager();
        boolean result = myManager.addContact(Contact.createContact(conInfo));
        if (result == false) { System.out.println("{Window} Failed to save
contact"); }
        else { System.out.println("{Window} Contact saved successfully"); }
    }
}
```

Code smells:

- `System.out.println` in `Manager` and `Window` should be refactored via "Extract Method"
- non-local access to public fields in `Contact` should be refactored via "Encapsulate Field"
- call to `Contact.CreateContact` in `Window` should be refactored. JUnit test class `TestWindow` should be used to test the `Window` class before and after refactoring (i.e. the

system should not change its behavior after refactoring)

Requirements:

- There should be one and only one `System.out.println` over the entire project
- No class other than `Contact` should access the fields of `Contact`
- No statement of the class `Window` should directly call any method of the class `Contact`

Refactored code:

```
public class Logger {
    public void log(String message) { System.out.println(message); }
}

public class Manager {
    private static Manager instance = new Manager(new Logger());
    private Contact lastAddedContact;
    private Logger logger;

    private Manager(Logger logger) { this.logger = logger; }
    public boolean addContact(String conInfo) {
        Contact contact = Contact.createContact(conInfo);
        this.lastAddedContact = contact;
        boolean isNull = (null == contact);
        if (isNull) { logger.log("{Manager} Contact is null"); return false; }
        logger.log("{Manager} Contact information:\n" + "Name\t:" +
contact.getName() + "\nSex\t:" + contact.getSex() + "\nAge\t:" +
contact.getAge());
        return true;
    }
    public static Manager getManager() { return instance; }
}

public class Contact {
    private String name; { getName }
    private int age; { getAge }
    private String sex; { getSex }
    public Contact(String name, int age, String sex) { this.name = name;
this.age = age; this.sex = sex; }
    public static Contact createContact(String inStream) {
        Scanner sc = new Scanner(inStream);
        try { return new Contact(sc.nextLine(), Integer.parseInt(sc.nextLine()),
sc.nextLine()); }
        catch (Exception e) { return null; }
    }
}

public class Window {
    private Logger logger;

    public Window(Logger logger) { this.logger = logger; }
    public void saveContact (String conInfo) {
        Manager myManager = Manager.getManager();
        boolean result = myManager.addContact(conInfo);
        if (result == false) { logger.log("{Window} Failed to save contact"); }
        else { logger.log("{Window} Contact saved successfully"); }
```

```
    }
}
```

Explanation:

1. By extracting the `System.out.println` to a `Logger` class, we centralize the logging functionality and reduce code duplication
2. By encapsulating the fields of `Contact`, we ensure that only `Contact` can access its own fields, promoting encapsulation and information hiding
3. In `Window.saveContact`, we pass `string conInfo` to `Manager.addContact`, preventing the unrelated class `Window` from having knowledge of the internal structure of `Contact`, adhering to the Law of Demeter (LoD, you should only talk to your immediate friends)

[8] Testing & Debugging

(1) Explain and illustrate with an example on why integration testing is still needed after the completion of unit testing.

Explanation:

- Unit testing focuses on testing individual components or modules in isolation to ensure they work correctly
- Integration testing verifies that the interactions between these components are functioning as expected

They test on different levels of detail, and thus, passing unit tests does not guarantee that the subsystem as a whole will work correctly. Integration testing is essential to validate the integration points and ensure that the components work together seamlessly.

Example:

- Scenario: Consider a scenario where the UserService is designed to register users and send them a welcome email using the EmailService.
- Unit Testing Phase: During unit testing, we verify that the UserService correctly registers a user and calls the sendEmail method on the EmailService. We might use a mock for EmailService, ensuring that our unit tests do not depend on the actual email-sending logic.
- Integration Testing Phase: After unit tests are complete, we conduct integration testing. In this phase, we test the complete workflow from user registration to email sending. We use the real EmailService to ensure that the integration is seamless and that the UserService correctly interacts with it.

(2) Write 3 JUnit test cases to achieve 100% statement coverage for the following code snippet.

```
public class Course {
    int code;
    String level;
    String name;
    public boolean SetCourse(int c, String n) {
        if ((c > 1000 && c <= 8999) && (n!=null)) {
            code = c; name = n;
        } else return false;
        int x = 1000;
        int lvl = 1;
        while (x <= 6999) {
            if (c > x && c <= x + 999) { level = "B" + lvl; return true; }
```

```
            x += 1000; lvl++;
        }
        level = "B8";
        return true;
    }
}
```

```
public class CourseTest {
    @Test
    public void testSetCourse1() {
        Course course = new Course();
        boolean actual = course.SetCourse(1234, "name");
        assertEquals(true, actual);
        assertEquals("B1", course.level);
    }

    @Test
    public void testSetCourse2() {
        Course course = new Course();
        boolean actual = course.SetCourse(8999, "name");
        assertEquals(true, actual);
        assertEquals("B8", course.level);
    }

    @Test
    public void testSetCourse3() {
        Course course = new Course();
        boolean actual = course.SetCourse(9000, "name");
        assertEquals(false, actual);
    }
}
```

(3) Does these test cases achieve 100% loop coverage? If not, write additional test cases to achieve 100% loop coverage.

Answer: The loop has 2 paths: one is loop for 1 time, another is loop for >1 times. It is infeasible to loop for 0 times since `x = 1000` makes `while (x <= 6999)` true at least once. Therefore, loop coverage is achieved with the existing test cases. Test 1 covers the loop for 1 time, and Test 2 covers the loop for >1 times. Test 3 does not enter the loop, which is irrelevant to loop coverage.

(4) Modify the code below so that each class can be tested independently at the unit test level.

```
public class Course {
    int code; String level; String name;
    public Course() { code = 0; level = null; name = null; }
    public boolean setCourse(int c, String n) {
        if ((c > 1000 && c <= 8999) && (n!=null)) {
            code = c; name = n;
        } else return false;
        Rule myRule = new Rule(c);
        level = myRule.computeLevel();
        return true;
    }
}
```

```
public class Rule {
    int code;
    public Rule(int c) { code = c; }
    public String computeLevel() {
        int x = 1000; int lvl = 1; String level = "B1";
        while (x <= 8999) {
            if (code > x && code <= x + 999) level = "B" + lvl;
            x += 1000; lvl++;
        }
        return level;
    }
}
```

`Rule` can be tested independently, so the question is to make `Course` can be initialized with a `Rule` object, so that we can write a `RuleStub` for testing `Course`.

```
public class Course {
    int code; String level; String name; Rule rule;
    public Course(Rule rule) { code = 0; level = null; name = null; this.rule =
rule; }
    public boolean setCourse(int c, String n) {
        if ((c > 1000 && c <= 8999) && (n!=null)) {
            code = c; name = n;
        } else return false;
        level = rule.computeLevel();
        return true;
    }
}
```

Explanation:

- By delegating the level computation to a `Rule` object, we decouple the logic from the `Course` class, making it feasible to test `Course` independently.
- The `Course` class now accepts a `Rule` object in its constructor, allowing us to inject a `RuleStub` during testing to isolate the behavior of the `Course` class.

(5) `Rule.computeLevel()` has a bug. Explain how to systematically locate the bug if the test input `code = 0` returns `B1`.

a) Initial Hypothesis

- Hypothesis: Rule.computeLevel() runs correctly
- Prediction: Rule.computeLevel(0) returns null
- Experiment: Run the test case
- Observation: Result is "B1"
- Conclusion: Hypothesis is rejected

b) Hypothesis on the Loop

- Hypothesis: The infection happens in the while loop
- Prediction: Remove the while loop and the method should return null
- Experiment: Run the test case
- Observation: Result is "B1"
- Conclusion: Hypothesis is rejected

c) Hypothesis on the Default Value

- Hypothesis: The default value of "B1" is causing the issue
- Prediction: Change the default value to null and the method should return null
- Experiment: Run the test case
- Observation: Result is null
- Conclusion: Hypothesis is accepted, bug is located

(6) Fix the bug in the `Rule.computeLevel()` method.

```
public String computeLevel() {
    int x = 1000; int lvl = 1; String level = null;
    while (x <= 8999) {
        if (code > x && code <= x + 999) level = "B" + lvl;
        x += 1000; lvl++;
    }
    return level;
}
```

[9] Bug Report

Explain four desirable properties of a good bug report, and illustrate with the following bug report.

```
Bug #50675  Passwords are not converted to same character set before comparing
Submitted:  27 Jan 2010 20:34
Reporter:   Lawrenty Novitsky
Status: Verified
Category:   MySQL Server
Version:    4.1, 5.0, 5,1, 5.5 bzr
Severity:   S2 (Serious)
OS: Any
CPU Architecture:   Any

[27 Jan 2010 20:34] Lawrenty Novitsky
Description:
I'm not sure is this bug report or feature request. But it looks more like a bug
for me.
In fact synopsis should be something like: compared passwords are not converted
to same charset representation before applying password() function.

The problem is looks like this: if one adds user and providing password in one
charset, and then some client sends(passes to mysql_real_connect) the same
string's representation in other charset(and i won't disguise that i'm talk here
about utf8 mainly), and if this representation differs from original(non-latin
characters is usual reason), - client will see "access denied".

How to repeat:
In the Windows command line(to ensure it won't be utf8) run CLI and add user
like
GRANT ALL ON *.* TO 'someuser'@'%' identified by '6'; # password contains
cyrillic character

Then try to connect using some linux shell(utf8) try to run smth like
mysql -h<your_dbserver_host> -u someuser -p6 test
```

```
Another option could be c/odbc 5.1 DSN configuration dialog on windows. myodbc5.1
converts all login credentials to utf8. And dialog has "Test" connection button.

"Access denied" you supposed to get at this point.

Now if you do

update user set password=password(0xD0B1) where User='someuser'; # 0xD0B1 is
bytes for "б" in utf8
flush privileges;

Both connections will be successful.

And one more little thing: --default-character-set doesn't come in play here from
my observations.

Suggested fix:
Convert password to utf8 before saving in mysql.user and do the same during
authentication process with supplied password. I'm not sure, but possibly the
latter should be done on client side in c/c or in other connectors, that have own
protocol implementation.
```

- Clarity: a good bug report should clearly describe the issue, including the problem, steps to reproduce, and expected results.
    - In this example, author clearly explains the problem with password comparison in different character sets and provides detailed steps to reproduce the issue.
- Reproducibility: the bug report should include clear and concise steps to reproduce the issue, allowing developers to replicate the problem.
    - The author provides specific commands to run in different environments to reproduce the access denied error.
- Environment Details: the report should include information about the environment, such as OS, MySQL version, and affected components.
    - The author specifies the MySQL versions affected by the bug and mentions the use of different character sets (utf8).
- Suggested Fix: a good bug report should offer suggestions or solutions to resolve the issue, providing insights for developers.
    - The author suggests converting passwords to utf8 before saving in mysql.user and during the authentication process to address the problem.

[10] Refactoring

```java
class AirTicket {
    String pax_Name;
    int ticket_fare;
    int ticket_class;
    int ticket_tax;

    public static final int ECONOMY = 0;
    public static final int ECONOMY_PLUS = 1;
    public static final int BUSINESS = 2;
    public static final int FIRST = 3;
```

```
}
// in another file
    double calculateGroupFare(AirTicket tickets[]) {
        double totalFare = 0;
        AirTicket ticket;
        for (int i = 0; i < tickets.length; i++) {
            ticket = tickets[i];
            if (ticket.ticket_class == AirTicket.ECONOMY_PLUS ||
ticket.ticket_class == AirTicket.BUSINESS || ticket.ticket_class ==
AirTicket.FIRST) {
                totalFare += 0.035 * ticket.ticket_fare + ticket.ticket_tax;
            } else {
                totalFare += 0.028 * ticket.ticket_fare + ticket.ticket_tax;
            }
        }
        return totalFare;
    }
```

The service fee is 2.8% for economy class and 3.5% for economy plus, business, and first class, and only the ticket fare is subject to the service fee.

(1) Identify 3 code smells in the given code snippet.

- Incorrect Implementation: The service fee should be added to the ticket fare. So, the multiplier should be 1.035 and 1.028 for the respective classes, instead of 0.035 and 0.028.
- Field exposure: The fields of the AirTicket class are public, violating encapsulation principles.
- Accessing object fields directly: The calculateGroupFare method directly accesses the ticket_class, ticket_fare, and ticket_tax fields of the AirTicket object, which is a violation of encapsulation.
- Code duplication: The calculation logic for different ticket classes is repeated in the if-else block, leading to code duplication.
  (2) Refactor the code.

```
abstract class TicketClass {
    abstract double getServiceFeeRatio();
}

class Economy extends TicketClass {
    @Override
    double getServiceFeeRatio() { return 0.028; }
}

class EconomyPlus extends TicketClass {
    @Override
    double getServiceFeeRatio() { return 0.035; }
}

class Business extends TicketClass {
    @Override
    double getServiceFeeRatio() { return 0.035; }
}

class First extends TicketClass {
    @Override
    double getServiceFeeRatio() { return 0.035; }
```

```
}

class AirTicket {
    private String paxName; { getPaxName }
    private int ticketFare; { getTicketFare }
    private TicketClass ticketClass; { getTicketClass }
    private int ticketTax; { getTicketTax }

    public AirTicket(String paxName, int ticketFare, TicketClass ticketClass,
int ticketTax) {
        this.paxName = paxName;
        this.ticketFare = ticketFare;
        this.ticketClass = ticketClass;
        this.ticketTax = ticketTax;
    }

    public double calculateTotalFare() {
        return ticketFare * (1 + ticketClass.getServiceFeeRatio()) + ticketTax;
    }
}

// Usage
    public double calculateGroupFare(AirTicket tickets[]) {
        double totalFare = 0;
        for (AirTicket ticket : tickets) {
            totalFare += ticket.calculateTotalFare();
        }
        return totalFare;
    }
```

(3) Explain 3 refactoring techniques used in the refactored code.

- Extract method: the total fare calculation logic is extracted into the AirTicket.calculateTotalFare method, encapsulating the calculation within the AirTicket class.
- Encapsulation: the fields of the AirTicket class are encapsulated, making them private and providing getter methods to access them.
- Replace Switch with State/Strategy: the polymorphic behavior of the ticket classes is implemented using the Strategy pattern, providing a single method signature for calculating the service fee ratio.
- Form Template Method: the TicketClass abstract class defines a template method getServiceFeeRatio, allowing subclasses to implement the specific service fee ratio calculation.
- Extract class: instead of integers, the ticket class is represented by a TicketClass object, encapsulating the behavior of different ticket classes.

[11] Project Management

The project is for creating a family therapy game for low-income families. UP (Unified Process) is used in this project.

(1) Provide specific deliverables for each of the represented activities in the UP.

- I1: Establish project scope
    - Activity: Define the kinds of family therapy games to develop

    - Deliverable: Project scope document on the types of games, including target audience, objectives, and features
- I2: Agree use case
    - Activity: Specify important use cases with therapy effects and standard game operations
    - Deliverable: Use case "Summarzing a Round" with detailed steps and expected outcomes
- E1: Demonstrate basic game operation
    - Activity: Model how a game can be operated to support features like user turns, self-esteemed calculation
    - Deliverable: Initial game operation model in flowchart showcasing how players interact with the game
- E2: Show ACT/SAY functionality
    - Activity: Design an application prototype that can prolong the period of inter-player interactions via ACT/SAY
    - Deliverable: UI Prototype with ACT/SAY buttons as the major feature
- C1: Complete the basic game operation
    - Activity: Implement the basic game play use cases
    - Deliverable: Functional game prototype with basic operations like dice rolling, player movement, and score calculation
- C2: Deliver family therapy features
    - Activity: Implement all essential and family therapy use cases and integrate all code as integrated application
    - Deliverable: Game prototype with family therapy features like conflict resolution, communication exercises, and emotional regulation
- C3: Improve code structure
    - Activity: Conduct code refactoring to enhance code quality
    - Deliverable: Refactored codebase with improved readability, maintainability, and performance
- T1: Support automated tests
    - Activity: Fix all bugs identified in the system test for the basic game plays
    - Deliverable: Automated test suite for the basic game operations
- T2: Conduct user acceptance tests
    - Activity: Log the user reported bugs and clear all outstanding issues
    - Deliverable: User acceptance test report with resolved issues and feedback
- T3: Release the software
    - Activity: Compose the configuration items for the release
    - Deliverable: Software release package with installation instructions, user manual, and release notes

(2) Identify the dependencies between the activities.

Activity	Dependencies	Remarks
I1	-	
I2	I1	
E1	I2	This focus on the basic game operation

Activity	Dependencies	Remarks
E2	I2	This focus on the speciality of the game, i.e., ACT/SAY functionality
C1	E1	This iteration only focuses on the basic game operation
C2	C1, E2	This iteration introduces family therapy features
C3	E1	Code refactoring happens during the coding phase
T1	C2, C3	
T2	T1	
T3	T2	

[12] Testing

Use one example to show that the bottom-up integration testing is not the reverse of top-down integration testing.

Consider the following hierarchy (A -> B means A calls B):

```
A -> B -> C, D
D -> E
```

Bottom-up:

- Unit testing:
    1. E
    2. C
- Integration testing:
    1. D + E
    2. B + C + D + E
- System Testing: A + B + C + D + E

Test stubs are not required for bottom-up testing because the lower-level modules are tested first.

Top-down:

- Unit testing:
    1. A (stub: B)
- Intergation testing:
    1. A + B (stub: C, D)
    2. A + B + C + D (stub: E)
- System Testing: A + B + C + D + E

In top-down testing, stubs are required for the lower-level modules that are not integrated yet.

[13] Debugging

(1) State and explain two merits and two drawbacks of the following bug report.

```
Bug #64737  Can not remove User Account
Submitted:  22 Mar 2012 15:10
Reporter:   Jeff Gao
Status: Open
Category:   MySQL Workbench: Administration
Version:    5.2.38 CE (Revision 8753)
Severity:   S3 (Non-critical)
OS: Windows (Windows 7 64)
CPU Architecture:   Any

[22 Mar 2012 15:10] Jeff Gao
Description:
In Server Management/Security/Users and Privileges, tab Server Access Management,
an user is listed in User Accounts list:
  User=bjeff, From Host=bctdw7jeffewc

1. Select the user bjeff, click Button Remove, click Remove on the confirmation
dialog, the item bjeff is gone from the User Accounts list.
  The button Apply is grayed, so there is no way to "apply" the change. I assume
the change is applied automatically by the system.
  But click on Refresh, the user comes back to the list again. Or click away from
Users and Privileges tab, and comes back to the tab, the user comes back again to
the list.

2. If I select the user bjeff, and click Reovoke All Privileges button, click
Revoke to confirm, I get the error:
  Unhandled exception: Error executing 'REMOVE ALL PRIVILIGES, GRANT OPTION FROM
'bjeff'@'bctdw7jeffewc'
  There is no such grant defined for user 'bjeff' on host 'bctdw7jeffewc'.
  SQL Error: 1141.
Click OK on the error. The use still in the list.

So I have no way to remove this user.
```

- Merits
    - Clarity. The report provides a clear description of the issue, including the steps to reproduce: after removing a user, the user reappears in the list; the expected result: the user should be removed; and the actual result: the user reappears. It also includes the error message when trying to revoke privileges.
    - Environment Details. The report specifies the MySQL Workbench version, OS, and CPU architecture, which helps in identifying the context of the issue.
- Drawbacks
    - Lack of Reproducibility. The report does not provide a consistent method to reproduce the issue. The user mentions that the user reappears after removal, but the steps to reproduce are not detailed enough to replicate the problem reliably.
    - Incomplete Information. The report lacks details on the system configuration, database setup, and any recent changes that might have triggered the issue. Without this information, developers may struggle to identify the root cause of the problem.

    - No Suggested Fix. The report does not offer any suggestions or solutions to resolve the issue. Providing insights or potential fixes can help developers in addressing the bug effectively.

(2) This is bugged because players are expected to only share 50% of the difference in their scores. Explain how to systematically locate the bug.

```
public class ShareSE {
    public shareSE() {}
    // share point between player p and player h by chance
    public int share(int c, int p, int h) {
        int cards = c, se_score1 = p, se_score2 = h;
        if ((se_score1 * cards - se_score2) % 2 == 1) return (se_score1 -
se_score2);
        else return 0;
    }
}
```

a) Initial Hypothesis

- Hypothesis: ShareSE.share() runs correctly
- Prediction: ShareSE.share(3, 15, 10) returns 2, i.e. half of the difference
- Experiment: Run the test case
- Observation: Result is 5
- Conclusion: Hypothesis is rejected

b) Hypothesis on the Condition

- Hypothesis: The infection takes place in the if condition
- Prediction: By reversing the condition, the method should return 2
- Experiment: Run the test case
- Observation: Result is 0
- Conclusion: Hypothesis is rejected

c) Hypothesis on the Calculation

- Hypothesis: The calculation is incorrect
- Prediction: Modify the calculation to return half of the difference
- Experiment: Run the test case
- Observation: Result is 2
- Conclusion: Hypothesis is accepted, bug is located

[14] Refactoring

Explain how the process of code refactoring is related to the process of testing.

In the testing-debugging-refactoring cycle:

- Testing
    - Define test objectives, write tests, execute tests, and analyze results
- Debugging
    - set hypothesis, make prediction about results, conduct experiment, record observations, and draw conclusions
- Code refactoring

- Identify code smells, apply refactoring templates, and re-test the code

In this cycle, if all test cases T are not passed, the code is debugged to locate the bug. After the bug is fixed, the code is refactored to improve its quality. The refactored code is then re-tested to ensure that the changes did not introduce new bugs. This iterative process of testing, debugging, and refactoring helps in improving the code quality and maintainability.

It is only possible, with the help of testing, to ensure that the refactored code behaves as expected and does not introduce new bugs. Testing provides a safety net that allows developers to make changes confidently, knowing that any regressions will be caught by the test suite. Further, refactoring helps in improving the testability of the code by making it more modular, readable, and maintainable, which in turn facilitates the testing process.