CS 520
Theory and Practice of Software Engineering
Fall 2022

Test driven development

October 20, 2022
Software development process: Two alternatives

Traditional software development process:
• First the software requirements are used to implement the software system
• Then they are used to write the test cases for that system

Test driven development process:
• First the software requirements are used to write the test cases for the software system
• Then they are used to implement that system
Example: Row game app

git clone -b hw1 https://github.com/LASER-UMASS/cs520
Row game app (Version 1)

- src/
  - TicTacToeGame
  - TicTacToeBlock
- test/
  - TestExample (2)

**Issues:**
- Simple architecture
- Poor design
- Violates best programming practices
Satisfied best practices
Satisfied best practices

- **Understandability**
  - Naming conventions (e.g., blocksData, player)
  - Documentation: README file, build file, comments, version control

- **Modularity**: TicTacToeGame and TicTacToeBlock classes

- **Information hiding (with encapsulation)**: TicTacToeBlock class has private fields and their methods

- **Type safety**: Some TicTacToeBlock class methods (e.g., constructor) have input validation

- **Testability**: TestExample class contains a test suite

- ...
Violated best practices
Violated best practices

• **Understandability**
  - Some key methods (e.g., move) lack comments.

• **Modularity:**
  - The TicTacToeGame class could apply the MVC architecture pattern.
  - The move method should have helper methods (e.g., isWin).

• **Information hiding:** The TicTacToeGame class declares public fields (e.g., blocksData, blocks).

• **Testability:** TestExample class is not adequate because it declares only 2 test cases.

• ...
Apply the MVC (Model View Controller) architecture pattern

Separates data representation (Model), visualization (View), and client interaction (Controller)
Row game app (Version 2)

- src/
  - controller/
    - RowGameController
    - RowGameRulesStrategy
  - model/
    - RowBlockModel
    - RowGameModel
  - view/
    - RowGameGUI
    - RowGameBoardView
    - RowGameStatusView
    - RowGameView
- test/
  - TestExample (10)

Goals:
- MVC architecture
- OO design: OO design principles and patterns
- Satisfies best programming practices
- More extensive testing
Test driven development process

1. Add new test case(s) for a new feature and run all test cases
   ○ The new test case(s) should fail
2. Implement that new feature and run all test cases
   ○ All tests should pass
3. Refactor the implementation as needed to improve its quality and run all test cases
   ○ All tests should pass
4. Repeat

https://en.wikipedia.org/wiki/Test-driven_development
1. **Add new test case(s) for a new feature**

A commonly applied test case template is:

1. setup       // In JUnit, @Before and check pre-conditions
2. execution   // In JUnit, call the constructor or method
3. validation  // In JUnit, check post-conditions
4. cleanup     // In JUnit, @After

**NOTE** The pre- and post-conditions are commonly implemented using assertions.
1. **Add new test case(s) for a new feature (cont.)**

Partially implement that new feature:

- Default usually no-op or return default value
- Stub may add some simplistic logic to a Default
Example: Add new RowGameModel class

1. Create a RowGameModel class. In that class, add two default methods:
   - String getFinalResult()
   - void setFinalResult(String finalResult)

2. Write 2 new test cases for these methods covering null inputs and non-null inputs

3. Run all of the test cases // One test case should fail
public class RowGameModel {
    public RowGameModel() {
        //TODO
    }

    public String getFinalResult() {
        //TODO
        return null;
    }

    public void setFinalResult(String finalResult) {
        //TODO
    }
}
New RowGameModel:
Two test cases for setFinalResult

```java
@Test
public void testSetFinalResultNull() {
    RowGameModel model = new RowGameModel();
    String finalResultNull = null;
    model.setFinalResult(finalResultNull);
    assertEquals(finalResultNull, model.getFinalResult());
}

@Test
public void testSetFinalResultNonNull() {
    RowGameModel model = new RowGameModel();
    String finalResultNonNull = "Player 2 wins!";
    model.setFinalResult(finalResultNonNull);
    assertEquals(finalResultNonNull, model.getFinalResult());
}
```
New RowGameModel: Failing test results

compile.tests:
  [javac] Compiling 1 source file to /Users/hconboy/Desktop/Teaching/2021Spring-CS520/Assignments/Homeworks/hw2/practice1/cs520-Spring2020/threeinarow/bin

test:
  [echo] Running unit tests ...
  [junit] Running TestExample
  [junit] Testsuite: TestExample
  [junit]
  [junit] Testcase: testSetFinalResultNull took 12.818 sec
  [junit] Testcase: testSetFinalResultNonNull took 0.014 sec
  [junit] FAILED
  [junit] expected:<Player 2 wins!> but was:<null>
  [junit] junit.framework.AssertionFailedError: expected:<Player 2 wins!> but was:<null>
  [junit] at TestExample.testSetFinalResultNonNull(TestExample.java:46)
  [junit]
  [junit] Testcase: testNewGame took 0.008 sec
  [junit] Testcase: testNewBlockViolatesPrecondition took 0.006 sec
  [junit] Test TestExample FAILED

BUILD SUCCESSFUL
Total time: 16 seconds
HeathernboysMBP:threeinarow hconboy$
2. Implement the new feature

- Fully implement the new feature in the simplest way possible to pass all test cases

- Often may need to add pre- and post-conditions
  - Explicit exception handling (e.g., Java if statement throws exception)
  - Run-time assertions (e.g., Java run-time assertions: https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html)
Example: Implement new RowGameModel class

1. In the RowGameModel class, add a finalResult field and use that field in the corresponding getter and setter methods
2. Re-run all of the test cases // They should now all pass

NOTE) If any test case fails, make additional changes and re-run the test cases
public class RowGameModel
{
    private String finalResult;

    public RowGameModel() {
        this.setFinalResult(null);
    }

    public String getFinalResult() {
        return this.finalResult;
    }

    public void setFinalResult(String finalResult) {
        this.finalResult = finalResult;
    }
}

New RowGameModel: Passing test results

ant-1.10.//bin/ant test

init:

compile:
    [javadoc] Compiling 1 source file to /Users/hconboy/Desktop/Teaching/2021Spring-CS520/Assignments/Homeworks/hw2/practice1/cs520-Spring2020/threeinarow/bin

compile.tests:

test:
    [echo] Running unit tests ...
    [junit] Running TestExample
    [junit] Testsuite: TestExample
    [junit]
    [junit] Testcase: testSetFinalResultNull took 14.294 sec
    [junit] Testcase: testSetFinalResultNonNull took 0.014 sec
    [junit] Testcase: testNewGame took 0.01 sec
    [junit] Testcase: testNewBlockViolatesPrecondition took 0.008 sec

BUILD SUCCESSFUL
Total time: 30 seconds
HeathernboysMBP:threeinarow hconboy$  []
3. **Refactor the implementation**

- Want a given software system to adhere to design principles and best programming practices
- Restructure code elements (e.g., packages, classes, methods, fields) of that system:
  - to better satisfy the non-functional requirements
  - while still satisfying the functional requirements

Some common refactoring patterns

- move class/field/method
- rename class/field/method
- extract class/field/method
- encapsulate field/method
- ...

e.g., https://refactoring.com/catalog/
Automated support for those patterns
Example: Move the RowGameModel class

- Create a new model package. Move the RowGameModel class to that package.
  - Will need to update all the uses of this class (including in the test cases)

- Run all of the test cases
package model;

public class RowGameModel
{
    private String finalResult;

    public RowGameModel()
    {
        this.setFinalResult(null);
    }

    public String getFinalResult()
    {
        return this.finalResult;
    }

    public void setFinalResult(String finalResult)
    {
        this.finalResult = finalResult;
    }
}
Move RowGameModel: Updated test cases

```java
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import model.RowGameModel;

/**
 * An example test class, which merely shows how to write JUnit tests.
 */
public class TestExample {
    ...
    // All exactly the same code
}
```
New RowGameModel: Passing test results

2/practice1/cs520-Spring2020/threeinarrow/build.xml

init:

compile:
   [javac] Compiling 1 source file to /Users/hconboy/Desktop/Teaching/2021Spring-CS520/Assignments/Homeworks/hw2/practice1/cs520-Spring2020/threeinarrow/bin

compile.tests:
   [javac] Compiling 1 source file to /Users/hconboy/Desktop/Teaching/2021Spring-CS520/Assignments/Homeworks/hw2/practice1/cs520-Spring2020/threeinarrow/bin

test:
   [echo] Running unit tests ...
   [junit] Running TestExample
   [junit] Testsuite: TestExample
   [junit] Testcase: testSetFinalResultNull took 16.904 sec
   [junit] Testcase: testSetFinalResultNonNull took 0.01 sec
   [junit] Testcase: testNewGame took 0.008 sec
   [junit] Testcase: testNewBlockViolatesPrecondition took 0.009 sec

BUILD SUCCESSFUL
Total time: 24 seconds
HeathernboysMBP:threeinarrow hconboy$
Example: Extract the player field

1. Extract the player field of the TicTacToeGame class to the RowGameModel class
   - Update the other classes as needed

2. Run all of the test cases
   - All of the test cases should pass
Extract player field: Java classes

1. Cut the “public String player;” field from the TicTacToeGame class and then paste that field into the RowGameModel class

2. Update the TicTacToeGame class

   ○ Import the model.RowGameModel

   ○ Added a new field “public RowGameModel model = new RowGameModel();”

   ○ In the constructor/methods, replace “player.” with “model.player” as well as “player = …” with “model.player = …”
@Test
    public void testNewGame() {
        assertEquals("1", game.model.player);
        assertEquals(9, game.movesLeft);
    }
Example: Encapsulate the player field

1. Encapsulate the player field of the RowGameModel class
   - Update the other classes as needed

2. Run all of the test cases
   - All of the test cases should pass
Potential benefits
Potential benefits

- First design a given component interface and then implement it
- Often the component better satisfies its non-functional requirements
- When test cases fail unexpectedly, can easily revert to the latest version that passed all test cases
- Usually better test adequacy (e.g., code coverage)
Potential drawbacks

- If a developer misunderstands the software requirements, that misunderstanding will be reflected in both the test cases and implementation.
- Generally must write a larger set of small test cases and then maintain them
- Usually focuses on unit testing but not integration or system testing
Class invariant: Overview

In computer programming, specifically object-oriented programming, a class invariant (or type invariant) is an invariant used for constraining objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object.

Class invariants are established during construction and constantly maintained between calls to public methods. Code within functions may break invariants as long as the invariants are restored before a public function ends. With concurrency, maintaining the invariant in methods typically requires a critical section to be established by locking the state using a mutex.

An object invariant, or representation invariant, is a computer programming construct consisting of a set of invariant properties that remain uncompromised regardless of the state of the object. This ensures that the object will always meet predefined conditions, and that methods may, therefore, always reference the object without the risk of making inaccurate presumptions. Defining class invariants can help programmers and testers to catch more bugs during software testing.

https://en.wikipedia.org/wiki/Class_invariant
Class invariant: Some model and tool support

<table>
<thead>
<tr>
<th>Development phase</th>
<th>Class invariant support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>e.g., Natural Language, FSA</td>
</tr>
<tr>
<td>Architecture &amp; design</td>
<td>e.g., javadoc</td>
</tr>
<tr>
<td>Implementation</td>
<td>e.g., Explicit exception handling, Run-time assertions</td>
</tr>
<tr>
<td>Unit testing</td>
<td>e.g., JUnit</td>
</tr>
</tbody>
</table>
Software Engineering: Applied skills

- Architecture pattern: MVC
- OO design principles
- Implementation: Refactoring (e.g., Eclipse)
- Verification & Validation: Code review, unit testing framework (e.g., JUnit)
- Documentation: README, doc tool (e.g., javadoc), internal comments
- Build tool: ant
- Version control system: git