CS 520

Theory and Practice of Software Engineering
Spring 2021

Test driven development

March 18, 2021

Software development process: Two alternatives

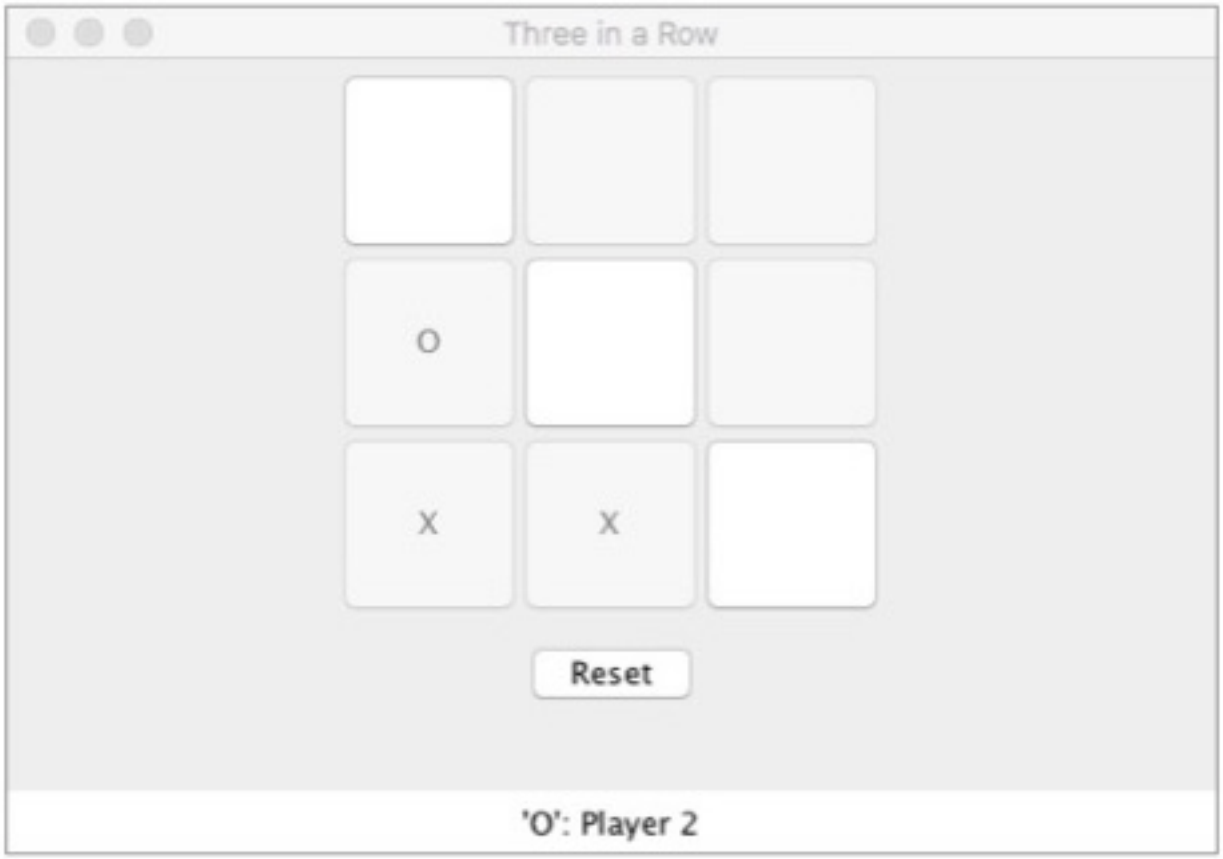
T**raditional software development process**:

- First the software requirements are used to implement the software system
- Then they are used to write the test cases for that system

T**est 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



From homework 1 and 2

Row game app (Version 1)

- src/
  - ThreeInARowGame
  - ThreeInARowBlock
- test/
  - TestExample (2)

Issues:

- Simple architecture
- Poor design
- Violates best programming practices
- Minimal unit testing

Homework 1:
git clone -b initial-version https://github.com/LASERUMASS/cs520-Spring2020

Row game app (Version 2)

- src/
  - controller/
    - RowGameController
    - RowGameRulesStrategy
  - model/
    - RowBlockModel
    - RowGameModel
  - view/
    - RowGameGUI
    - RowGameBoardView
    - RowGameStatusView
    - RowGameView
- test/
  - TestExample (14)

Goals:

- MVC architecture
- OO design: OO design principles and OO design patterns
- Satisfies programming best practices
- More extensive testing

Homework 2:
git clone https://github.com/LASERUMASS/cs520-Spring2020

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 should still 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:

- Dummy usually no-op or return default value
- Stub may add some simplistic logic to a Dummy

Example: Add new RowGameModel class

1. Create a RowGameModel class. In that class, add two dummy 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

New RowGameModel: Java class (partial)

```
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

```
@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-CS
520/Assignments/Homeworks/hw2/practice1/cs520-Spring2020/threeinarow/bin

test:
     [echo] Running unit tests ...
    [junit] Running TestExample
    [junit] Testsuite: TestExample
    [junit] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 12.872 s
ec
    [junit] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 12.872 s
ec
    [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

New RowGameModel: Java class (full)

```
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-1.10.7/bin/ant test
Buildfile: /Users/hconboy/Desktop/Teaching/2021Spring-CS520/Assignments/Homeworks/hw
2/practice1/cs520-Spring2020/threeinarow/build.xml

init:

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

compile.tests:

test:
     [echo] Running unit tests ...
    [junit] Running TestExample
    [junit] Testsuite: TestExample
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.378 s
ec
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.378 s
ec
    [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

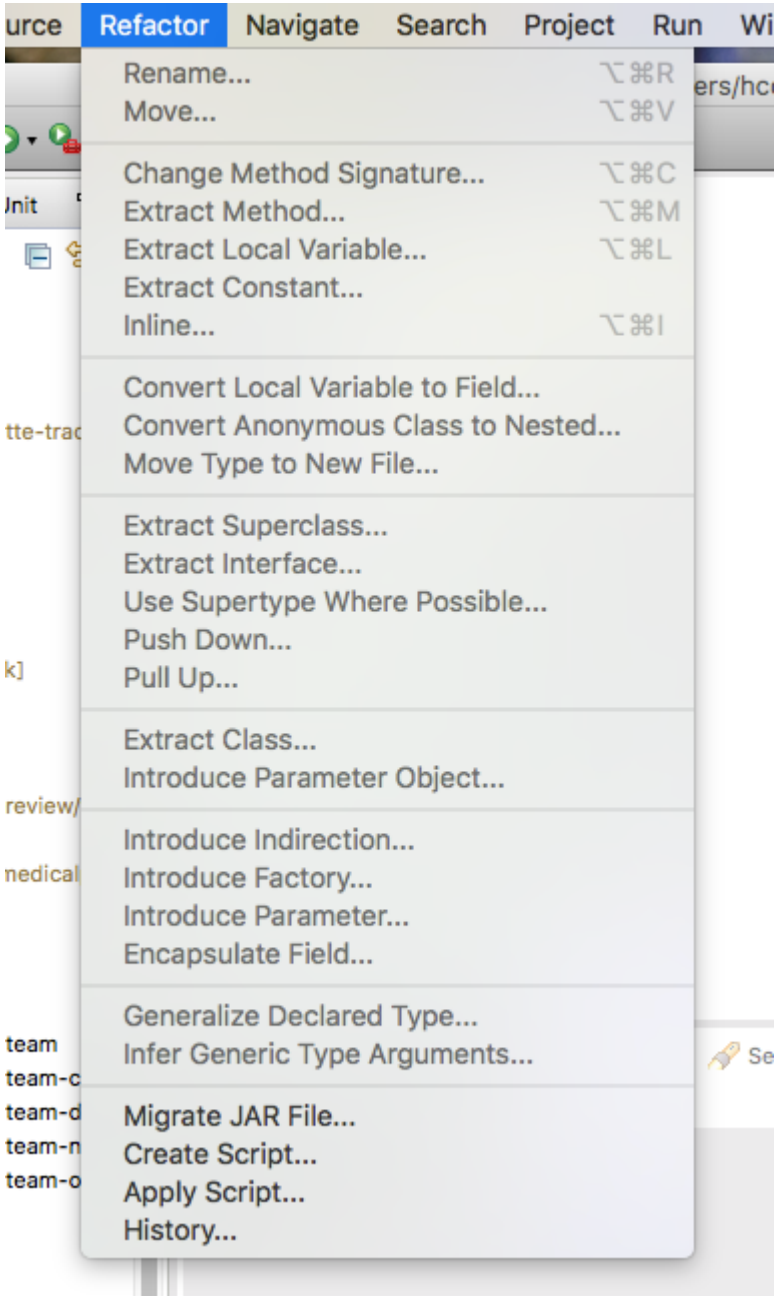
https://en.wikipedia.org/wiki/Code_refactoring

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

Move RowGameModel: Java class (full)

```
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;
    }
}
```

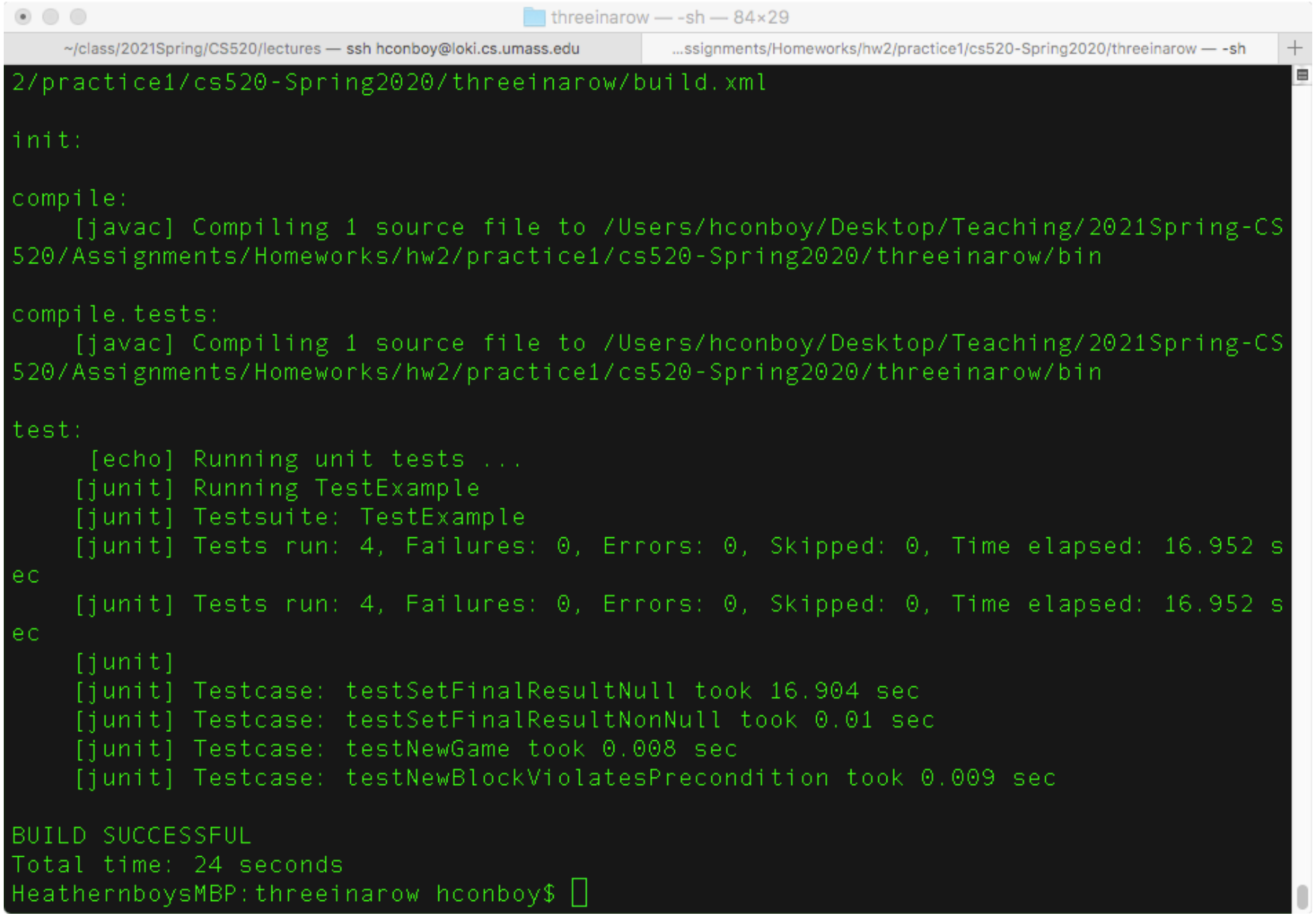
Move RowGameModel: Updated test cases

```
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



Example: Extract the player field

1. Extract the player field of the ThreeInARowGame 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 ThreeInARowGame class and then paste that field into the RowGameModel class
2. Update the ThreeInARowGame 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 = …”

NOTE) See the Homework 2 source code

Extract player field: Update test cases

```
@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

Development phase	Class invariant support
Requirements	e.g., Natural Language, FSA
Architecture & design	e.g., javadoc
Implementation	e.g., Explicit exception handling, Run-time assertions
Unit testing	e.g., JUnit

Homework 2

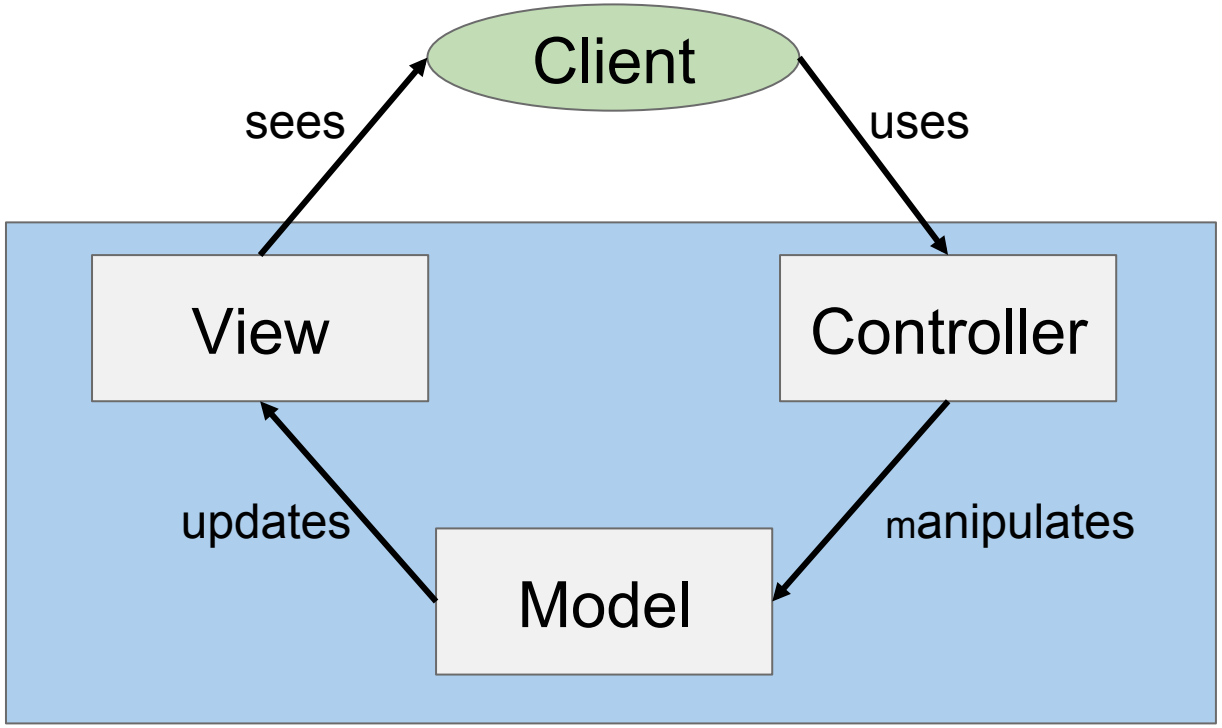
- Goal: Re-design, re-implement, and test the Row game app
- Deliverables: MVC architecture pattern, Observer design pattern, Strategy (or template method) design pattern, Code review proposed fixes, Git log messages
- Due: Thursday April 1, 2021, 9 AM EDT

https://people.cs.umass.edu/~hconboy/class/2021Spring/CS520/hw2.pdf

Code review: Proposed fixes

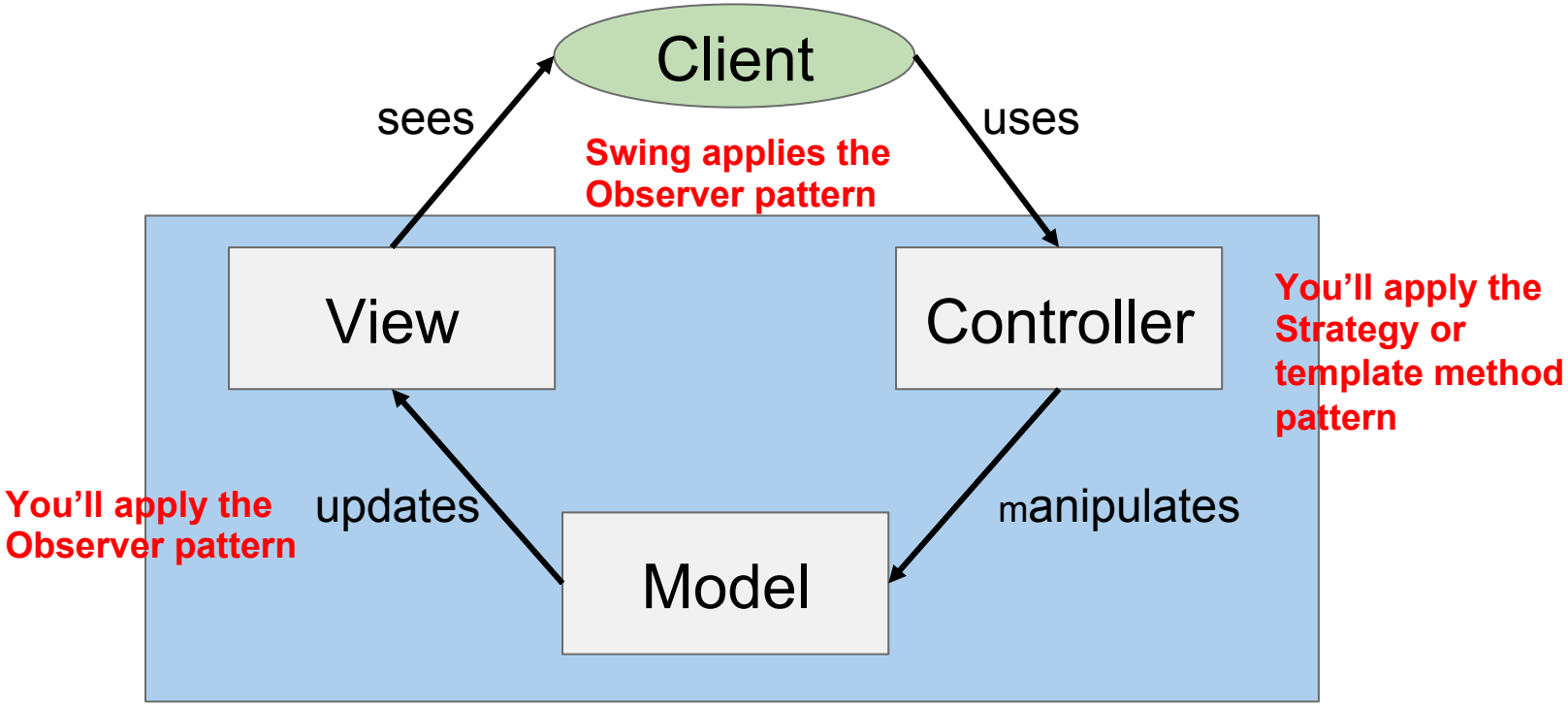
- See the Homework 1 possible answers
- Your own

Architecture pattern:
MVC (Model View Controller)



Separates data representation (Model),
visualization (View), and client interaction (Controller)

Design patterns:
Observer and Strategy

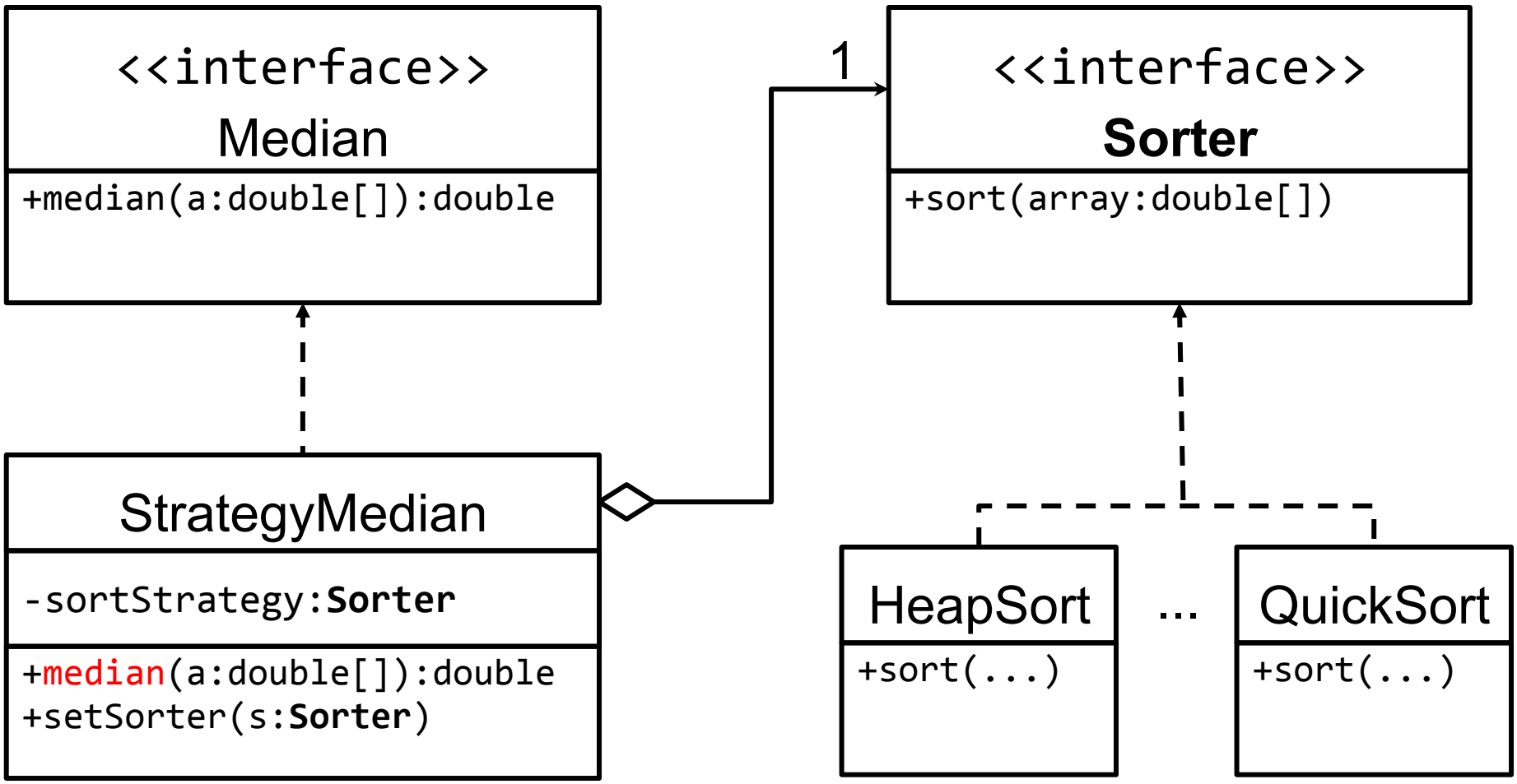


Separates data representation (Model), visualization (View), and client interaction (Controller)

RowGameController:
Supporting different row game rules

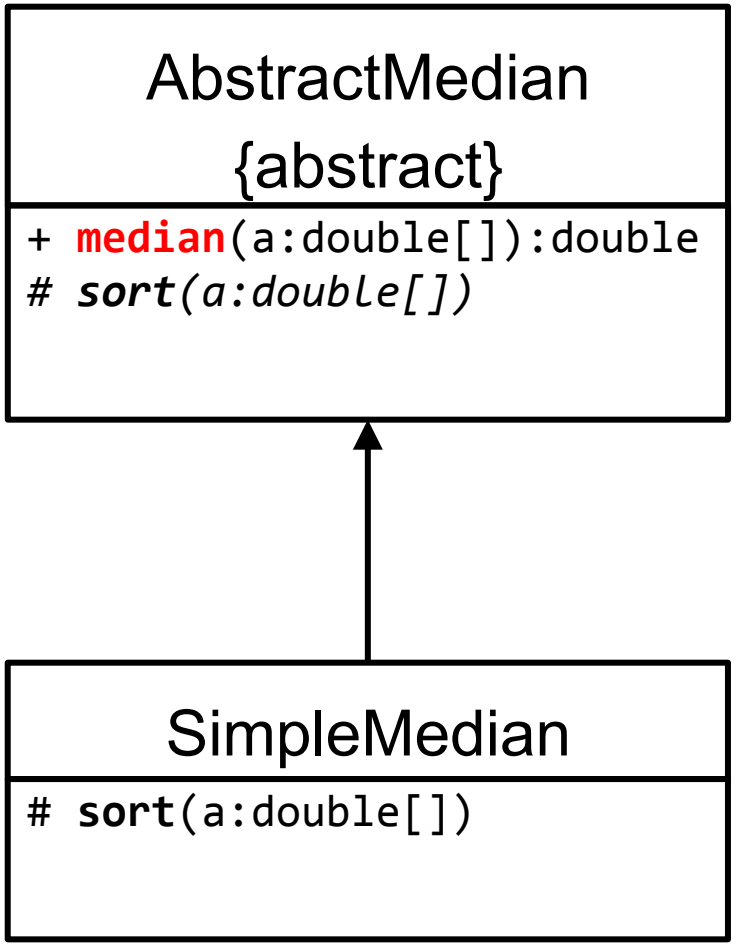
- Different row game rules:
  - Three in a row
  - Tic tac toe
- Two possible solutions:
  - Strategy design pattern
  - Template method design pattern

Strategy design pattern:
RowGameController



"median" delegates the sorting of the array to a "sortStrategy"

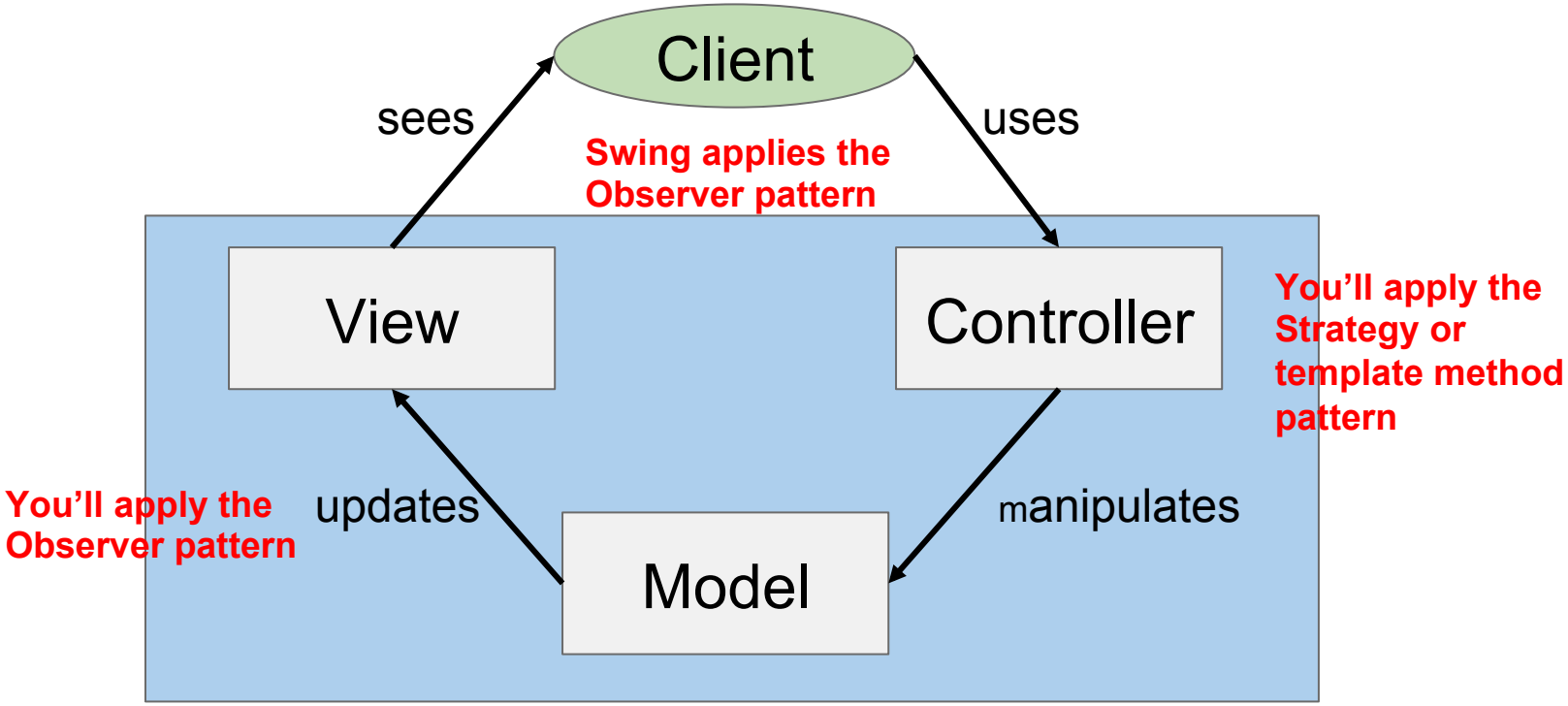
Template method design pattern:
RowGameController



Should the median method be final?

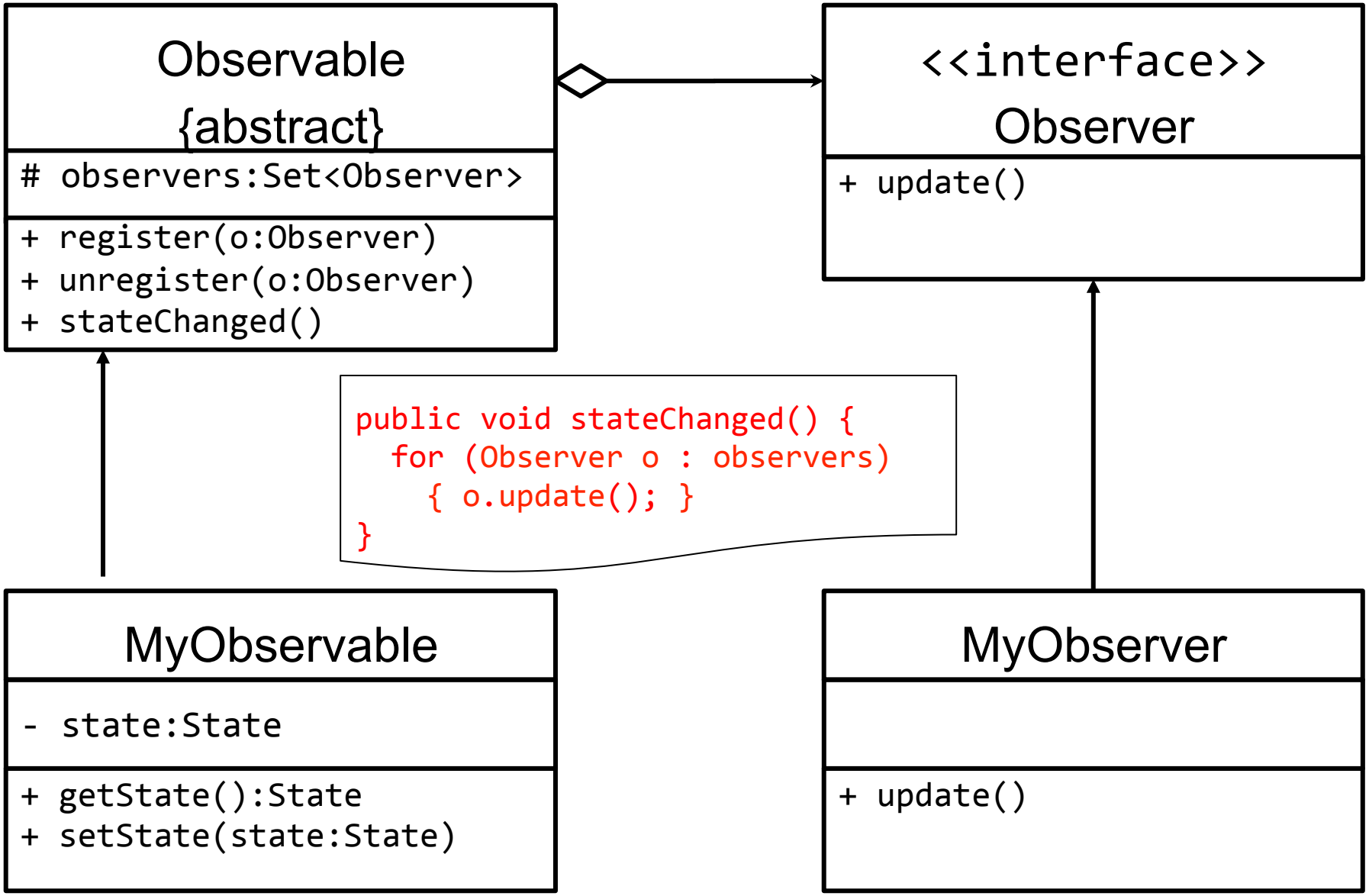
- The template method (**median**) implements the algorithm but leaves the **sorting** of the array undefined.
- The concrete subclass only needs to implement the actual **sorting**.

Design patterns:
Observer and Strategy



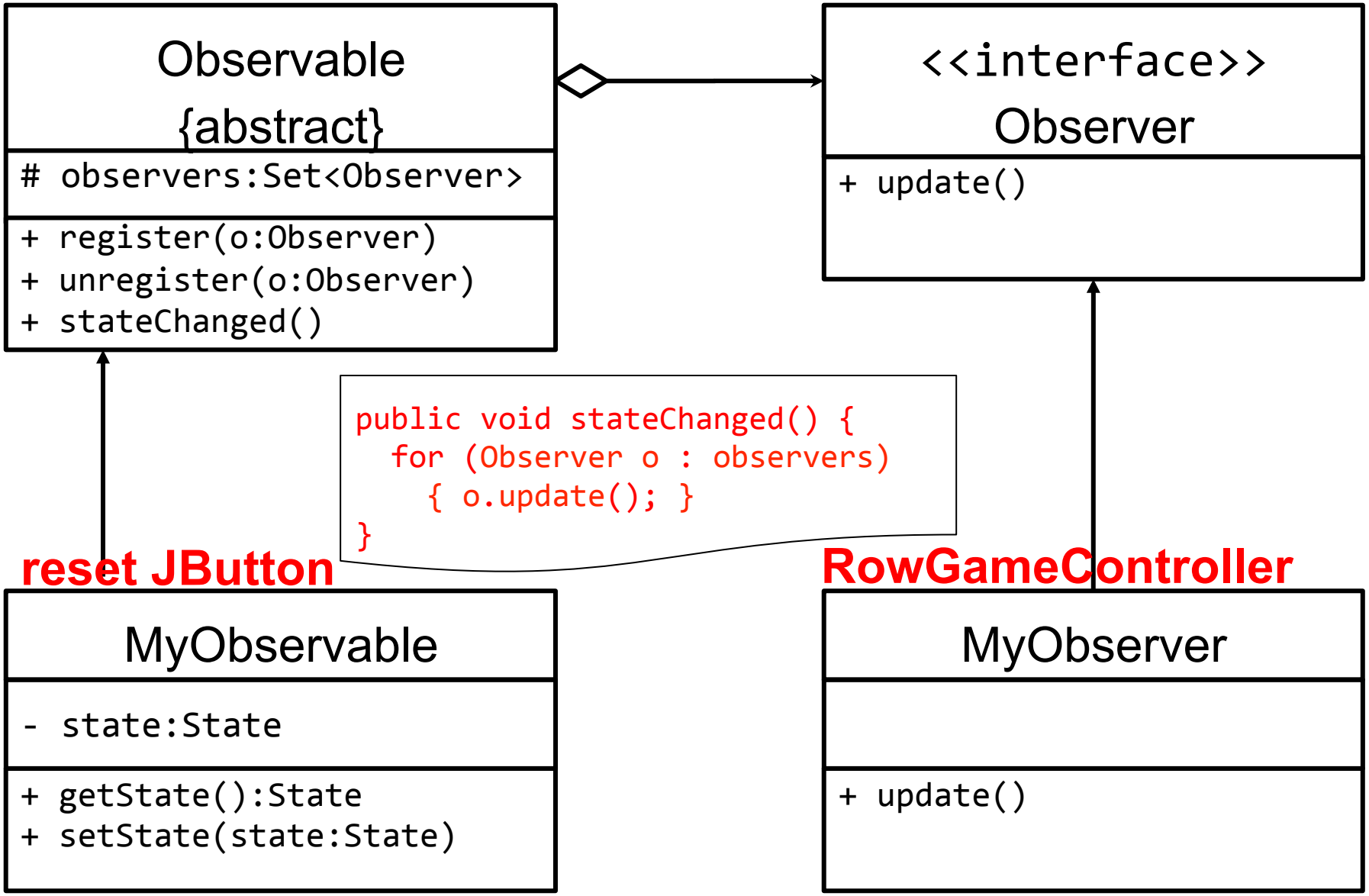
Separates data representation (Model), visualization (View), and client interaction (Controller)

Observer pattern



// For the setState method, use the stateChanged method

Observer pattern: Reset JButton and RowGameController

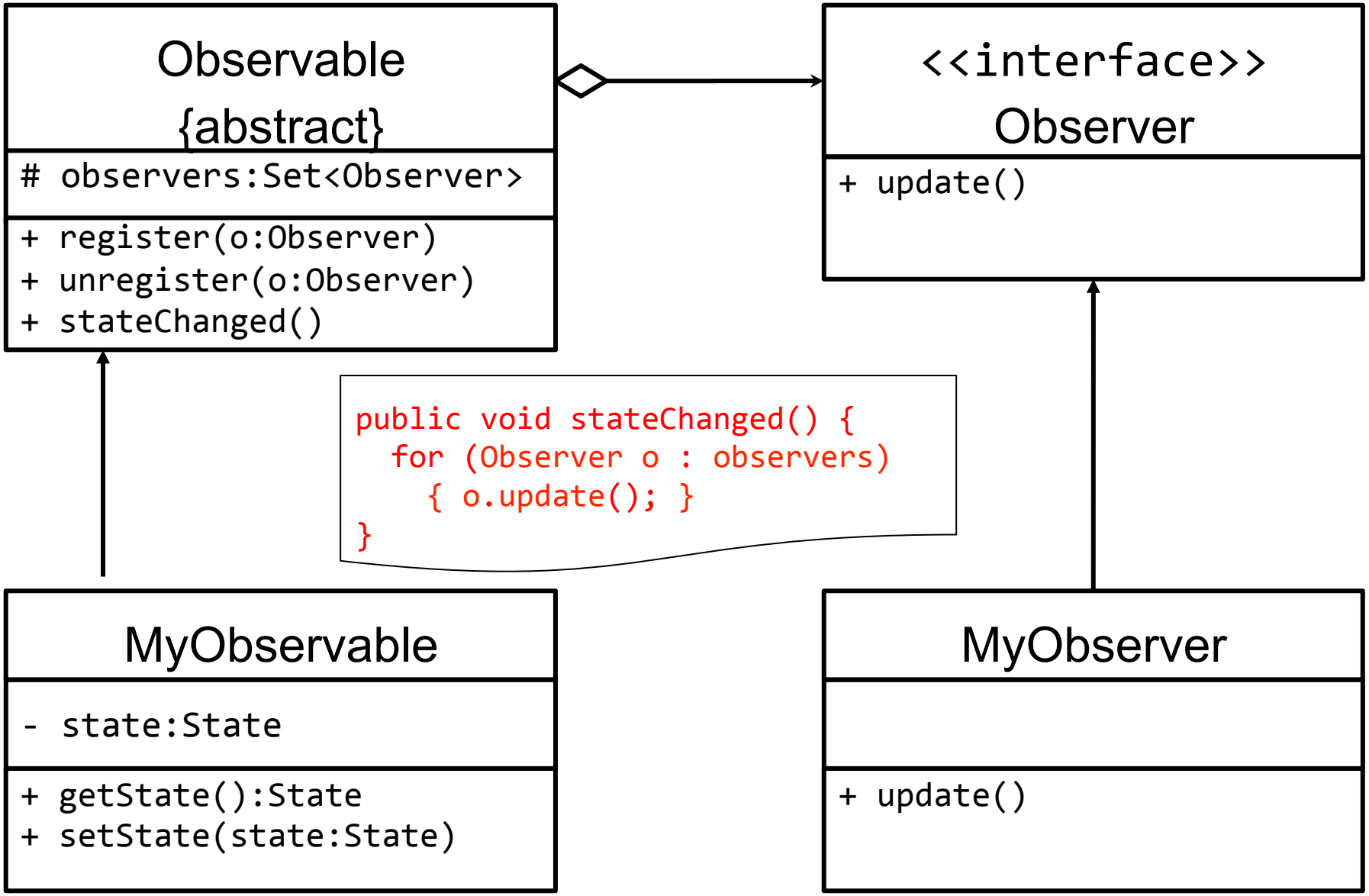


// For the setState method, use the stateChanged method

Learn by example: Swing View/Controller pair

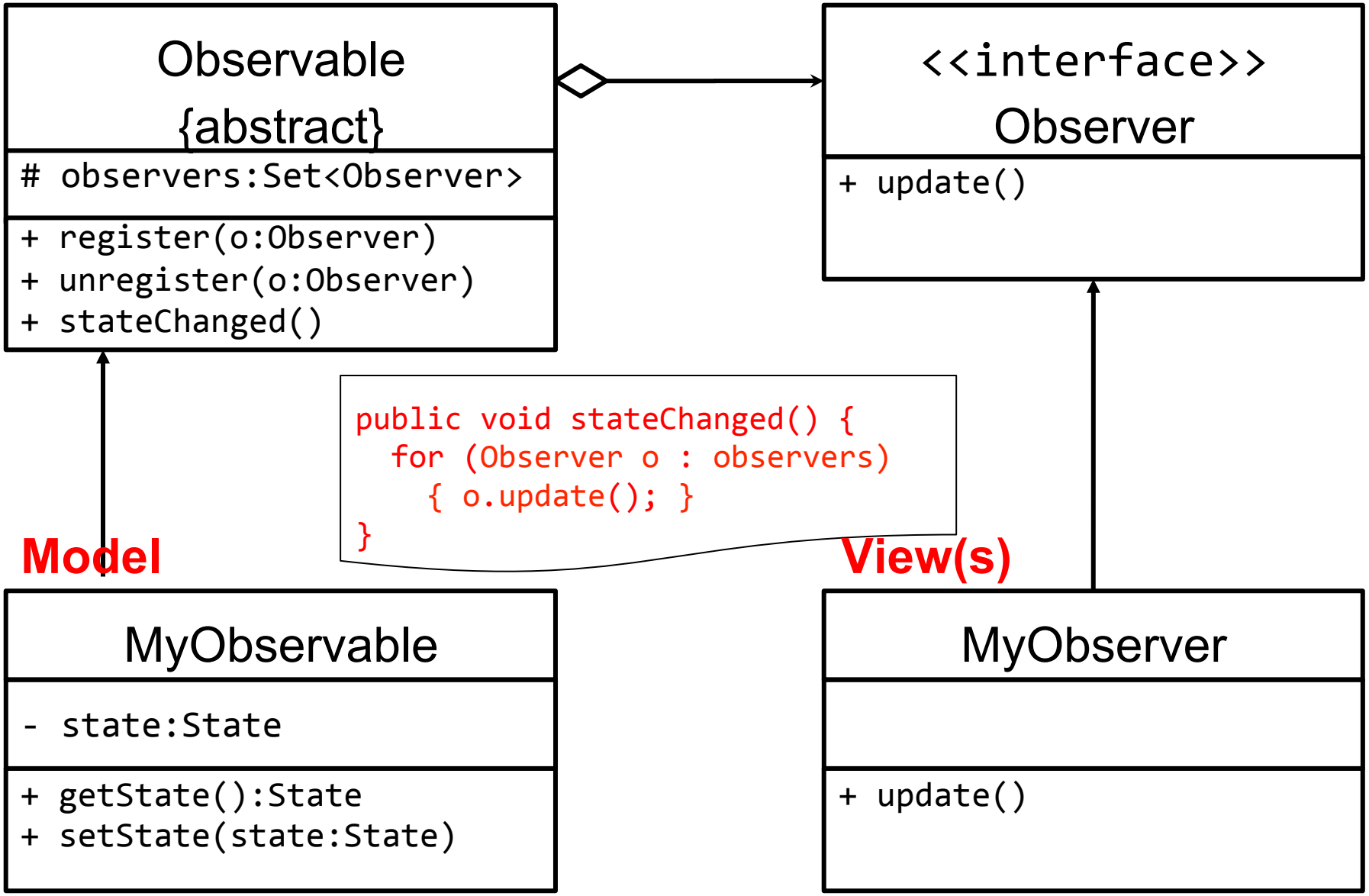
- https://docs.oracle.com/en/java/javase/14/docs/api/java.desktop/javax/swing/JButton.html
- https://docs.oracle.com/en/java/javase/14/docs/api/java.desktop/javax/swing/AbstractButton.html
- https://docs.oracle.com/en/java/javase/14/docs/api/java.desktop/java/awt/event/ActionListener.html

Observer pattern



// For the setState method, use the stateChanged method

Observer pattern: Model and its View(s)



// For the setState method, use the stateChanged method

Learn by example: Game Model/View pair

- https://docs.oracle.com/en/java/javase/15/docs/api/java.desktop/java/beans/PropertyChangeSupport.html

- https://docs.oracle.com/en/java/javase/15/docs/api/java.desktop/java/beans/PropertyChangeListener.html

Automated testing: JUnit test suite

- At least 14 test cases (including the two test cases given to you)
- For each package model, view, and controller, at least one test cases
- For each of the row game rules (Three in a Row and Tic Tac Toe), at least the following test cases:
  - Illegal move (should not change the game board)
  - Legal move (should change the game board)
  - One of the players win
  - The two players tie
  - Reset

Software Engineering: Applied skills

- Architecture pattern: MVC
- OO design principles and patterns: Observer, Strategy (or Template method)
- 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