

Java Programming

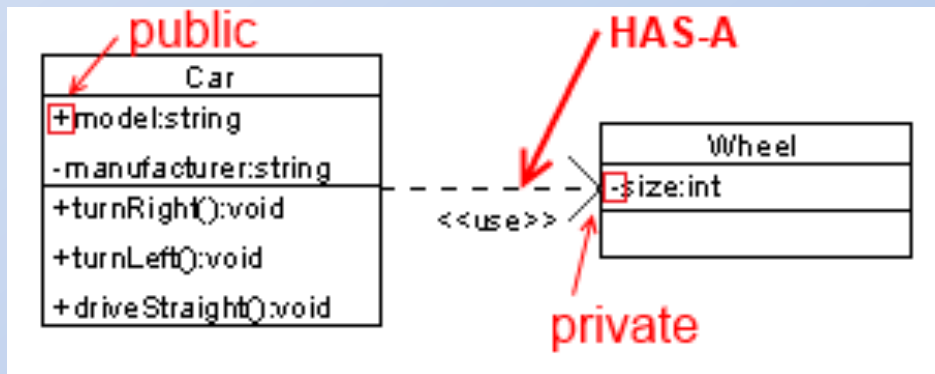
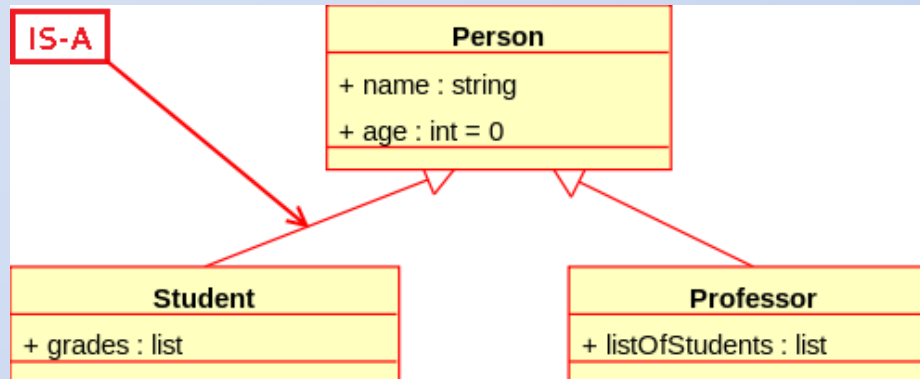
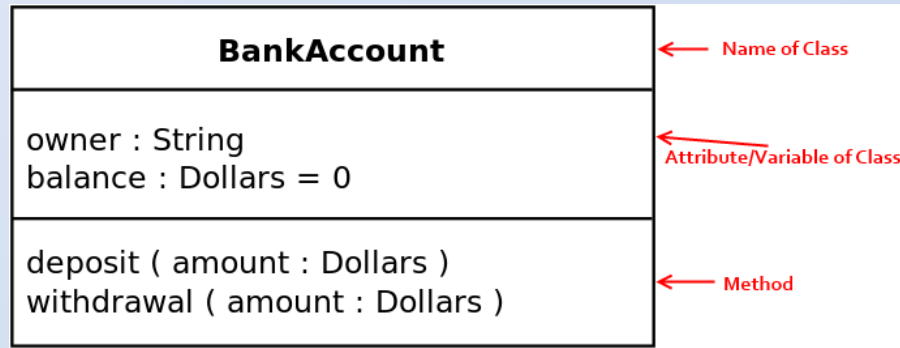
TA: Jimin Hsieh

Agenda

- Class Diagram
- Modular Programming
- Test your code
- Junit
- Basic Design Pattern

- Class Diagram
- Modular Programming
- Test your code
- Junit
- Basic Design Pattern

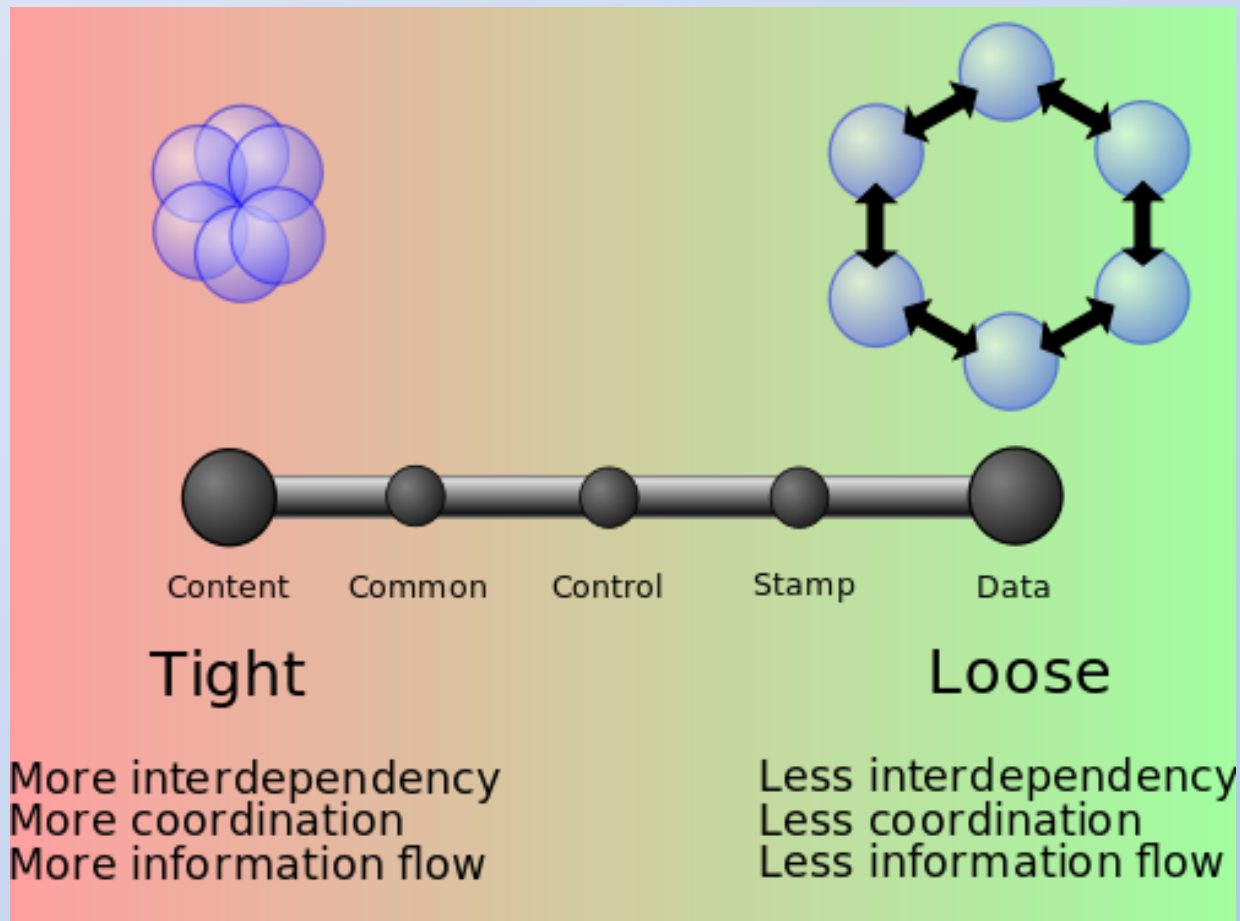
Class Diagram



- Class Diagram
- Modular Programming
- Unit Testing
- Junit
- Basic Design Pattern

Module Programming

- Definition: High Cohesion & Low Coupling



- Rule:
 - Break a large program into smaller independent modules
 - Work with modules of **reasonable size**, even in program involving a large amount of code
 - Share and **reuse** code without having to reimplement it
 - Easily substitute improved implementations
 - Develop **appropriate abstract models** for addressing programming problems
 - **Localize debugging**(Unit Testing)

How make code modular?

- Basic Skill: Encapsulation, Inheritance and Polymorphism
- Access Modifier: public vs. private
- Method: **static method** vs. instance(non-static) method
- Interface:
- Interface usage: unrelated classes implement a set of common methods

- Class Diagram
- Modular Programming
- Unit Testing
- Junit
- Basic Design Pattern

Unit Testing

- IEEE Standard Definition: Testing of individual hardware or software units or groups of related units
- In OOP, a unit is often an entire interface, such as class, but could be an individual method.

- Class Diagram
- Modular Programming
- Unit Testing
- Junit
- Basic Design Pattern

Junit

- What is this?
 - JUnit is a unit testing framework to write repeatable tests for Java.
- **JUnit 4** is already in Eclipse, you can just use it.
- JUnit already contain “static void main”.
- Similar Framework: TestNG

Annotations

- @Test – This method is a test method
- @Before & @After – This method is executed before/after **each test**.
- @BeforeClass & @AfterClass
 - The Method you annotate will get executed, **only once**, before/after all of your @Test methods.
- @Ignore – Ignore the test method

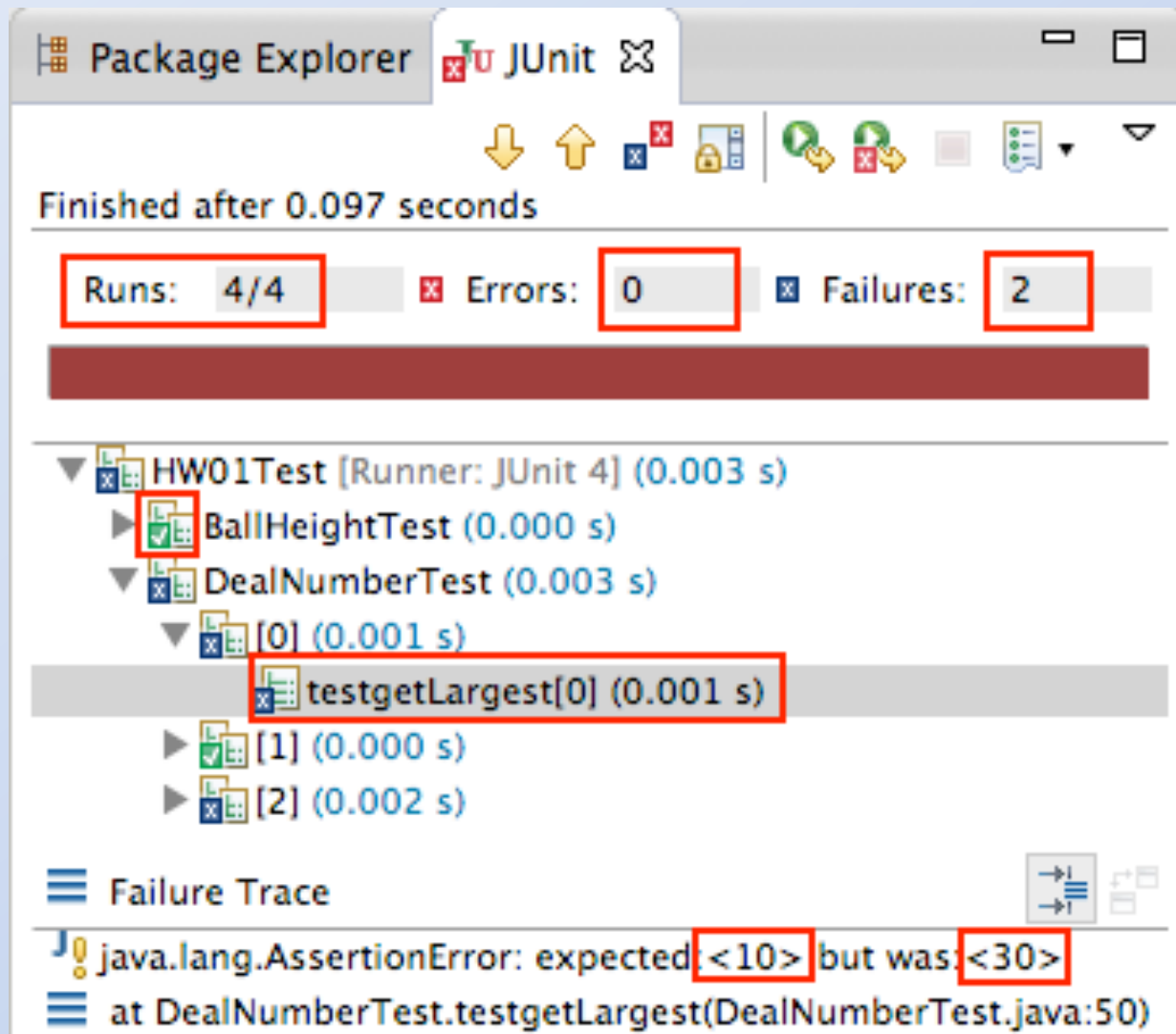
Test Methods

S.N.	Methods & Description
1	void assertEquals(boolean expected, boolean actual) Check that two primitives/Objects are equal
2	void assertTrue(boolean expected, boolean actual) Check that a condition is true
3	void assertFalse(boolean condition) Check that a condition is false
4	void assertNotNull(Object object) Check that an object isn't null.
5	void assertNull(Object object) Check that an object is null
6	void assertSame(boolean condition) The assertEquals() methods tests if two object references point to the same object
7	void assertNotSame(boolean condition) The assertEquals() methods tests if two object references not point to the same object
8	void assertEquals(expectedArray, resultArray); The assertEquals() method will test whether two arrays are equal to each other.

Sample Code

- DealNumberTest.java
- BallHeightTest.java

Result in Eclipse



Package Explorer JUnit

Finished after 0.097 seconds

Runs: 4/4 Errors: 0 Failures: 2

HW01Test [Runner: JUnit 4] (0.003 s)

- BallHeightTest (0.000 s)
- DealNumberTest (0.003 s)
 - [0] (0.001 s)
 - testgetLargest[0] (0.001 s)
 - [1] (0.000 s)
 - [2] (0.002 s)

Failure Trace

java.lang.AssertionError: expected: <10> but was: <30>
at DealNumberTest.testgetLargest(DealNumberTest.java:50)

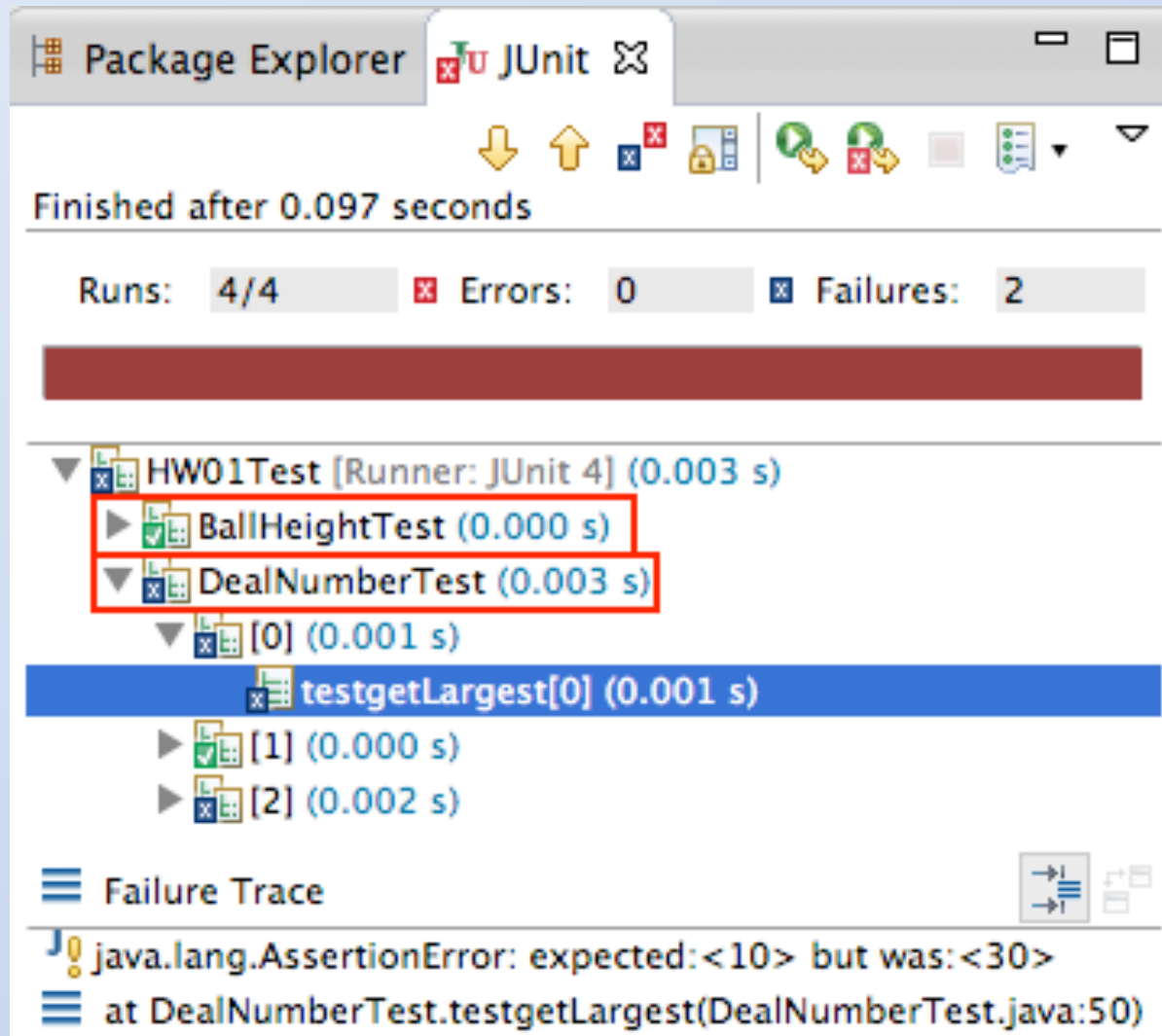
Test Suit

- Test suite means bundle a few unit test cases and run it together.
- What library we need?
 - `org.junit.runner.RunWith`
 - `org.junit.runners.Suite`
 - `org.junit.runners.Suite.SuiteClasses`

Test Suit Sample Code

- HW01Test.java

Result in Eclipse



The screenshot shows the Eclipse IDE's Package Explorer and JUnit test results. The Package Explorer on the left shows a project named 'JUnit'. The main area displays the test results for 'HW01Test [Runner: JUnit 4] (0.003 s)'. The test results are as follows:

- Runs: 4/4
- Errors: 0
- Failures: 2

The test results are displayed in a tree view:

- HW01Test [Runner: JUnit 4] (0.003 s)
 - BallHeightTest (0.000 s) [Pass]
 - DealNumberTest (0.003 s) [Fail]
 - [0] (0.001 s) [Fail]
 - testgetLargest[0] (0.001 s) [Fail]
 - [1] (0.000 s) [Pass]
 - [2] (0.002 s) [Pass]

The failure trace at the bottom shows:

```
java.lang.AssertionError: expected:<10> but was:<30>  
at DealNumberTest.testgetLargest(DealNumberTest.java:50)
```

Exercise

- Please write your own test for your previous homework.

- Class Diagram
- Modular Programming
- Unit Testing
- Junit
- Basic Design Pattern

Basic Design Pattern

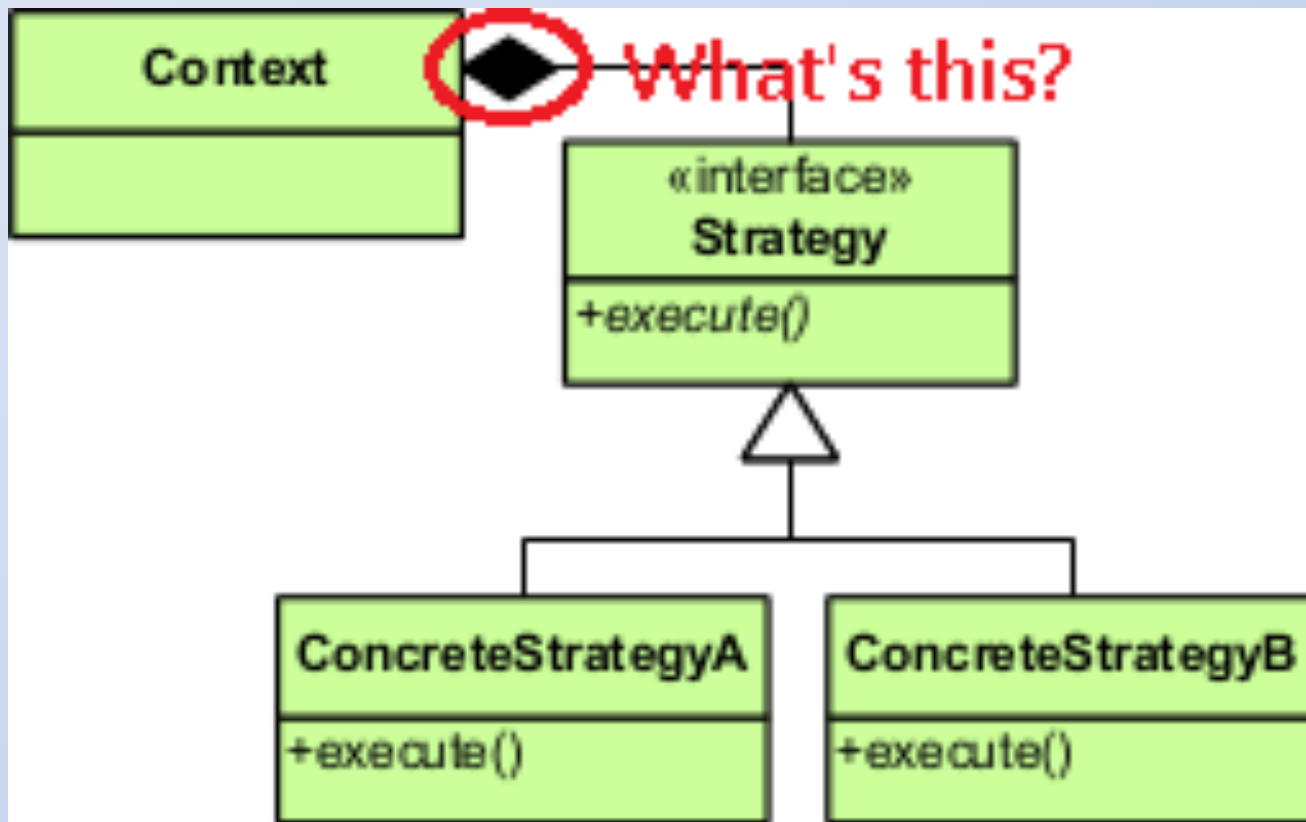
- What is Design Pattern?
- Strategy Pattern
- Observer Pattern

What is Design Pattern?

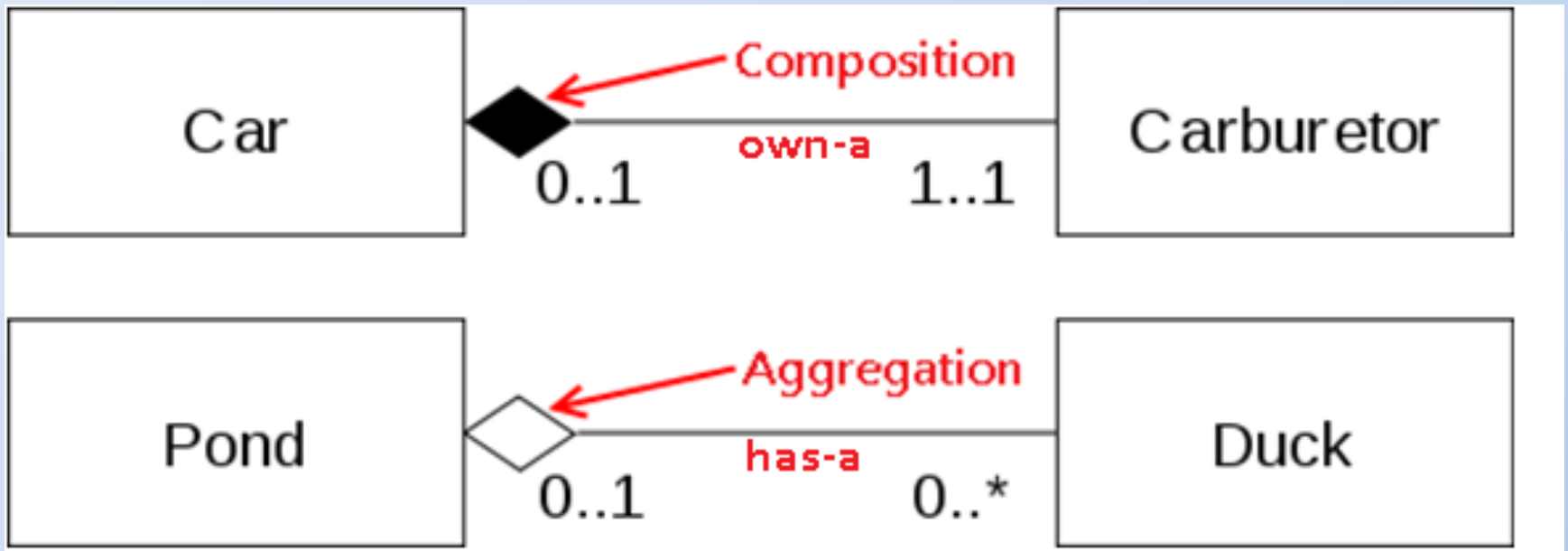
- Definition: A design pattern is a general **reusable solution** to a **commonly occurring problem** within a given context in software design.
- Why you should learn design pattern?

Strategy Pattern

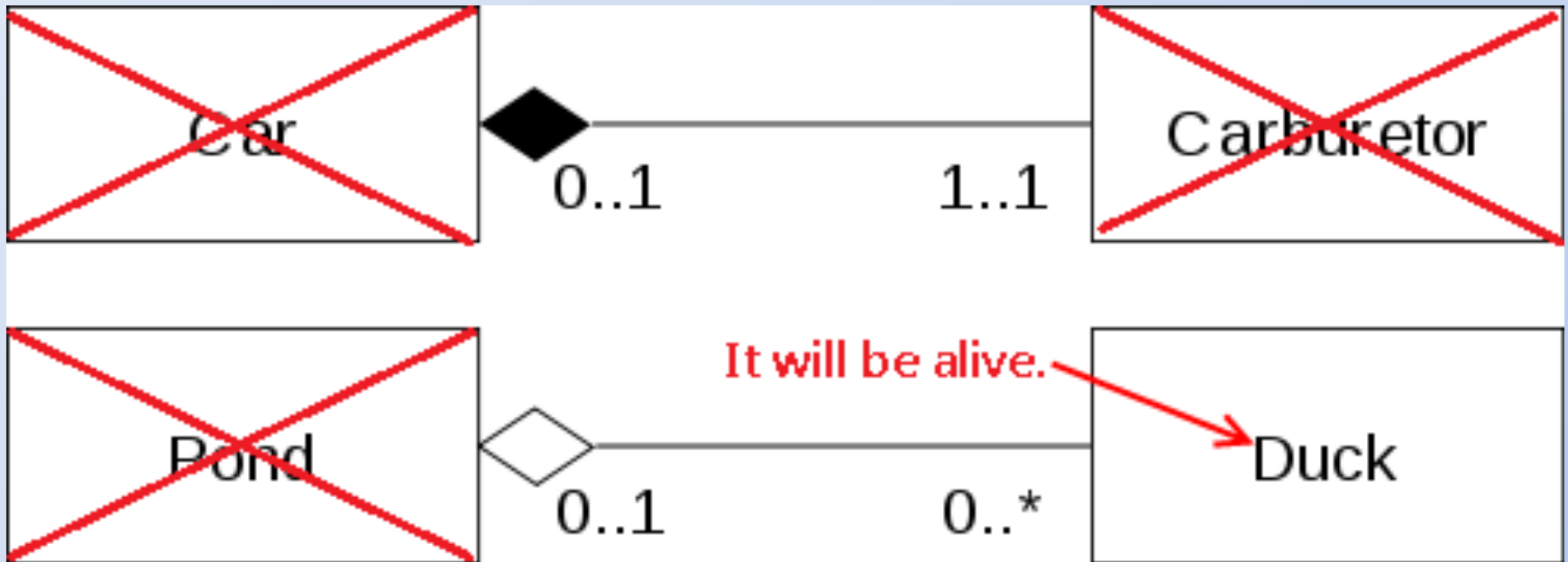
- Subclass decide how to implement steps in an algorithm.



Diamond Shape



Diamond Shape(Cont.)



Strategy Pattern - Interface

/** The classes that implement a concrete strategy should implement this.

* The Context class uses this to call the concrete strategy. */

```
interface Strategy {  
    int execute(int a, int b);  
};
```


Strategy Pattern – Real Strategy

```
class Add implements Strategy {  
    public int execute(int a, int b) {  
        System.out.println("Called Add's  
execute()");  
        return a + b; // Do an addition  
        with a and b  
    }  
};
```

```
class Subtract implements Strategy {  
    public int execute(int a, int b) {  
        System.out.println("Called  
Subtract's execute()");  
        return a - b; // Do a subtraction  
        with a and b  
    }  
};
```

Strategy Pattern – Setup Strategy

```
class Context {  
    private Strategy strategy;  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
    public int executeStrategy(int a, int b) {  
        return this.strategy.execute(a, b);  
    }  
};
```



The diagram illustrates the relationship between the `this` keyword and the `strategy` attribute in the `Context` class constructor. Two red arrows originate from the `this` keyword in the line `this.strategy = strategy;`. One arrow points to the `strategy` attribute of the `Context` class, and the other points to the `strategy` parameter of the `Context` constructor, indicating that the parameter is being assigned to the class attribute.

Strategy Pattern - TestClient

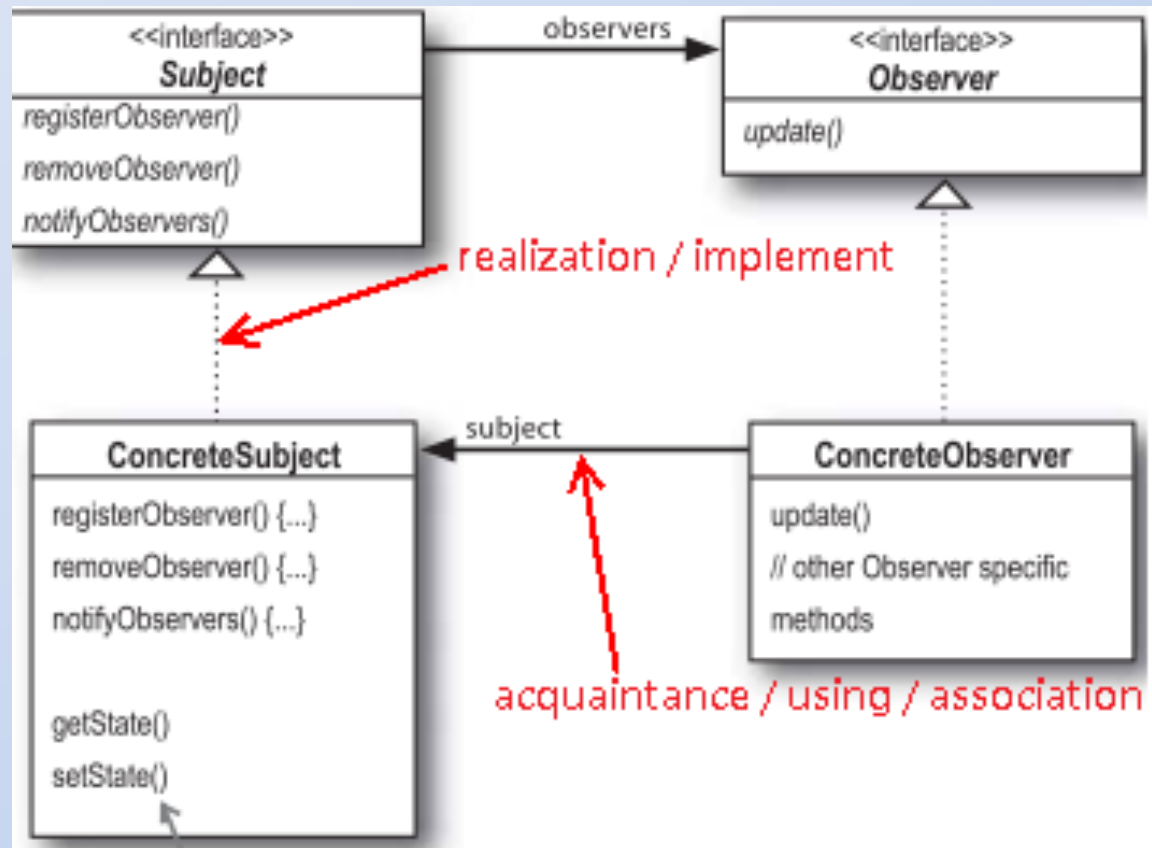
```
class StrategyExample {  
    public static void main(String[] args) {  
        Context context;  
  
        // Three contexts following different strategies  
        context = new Context(new Add());  
        int resultA = context.executeStrategy(3,4);  
  
        context = new Context(new Subtract());  
        int resultB = context.executeStrategy(3,4);  
  
        context = new Context(new Multiply());  
        int resultC = context.executeStrategy(3,4);  
  
        System.out.println("Result A : " + resultA );  
        System.out.println("Result B : " + resultB );  
        System.out.println("Result C : " + resultC );  
    }  
};
```

Exercise

- Try to create a player to play mp3, ALAC, and FLAC type music.
- Next week, I will put my source code to ftp server.

Observer Pattern

- Allows an object to change its behavior when some state change.



Sample Source Code

Please checkout from ftp.

Reference

- Head First Design Patterns
- Junit in Action
- Wikipedia
- <http://javapapers.com/>

Q&A

Thanks for you attentions!