Java Programming

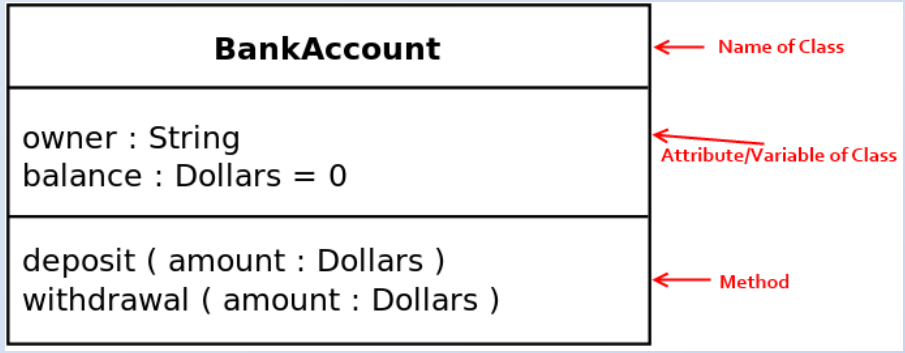
TA: Jimin Hsieh

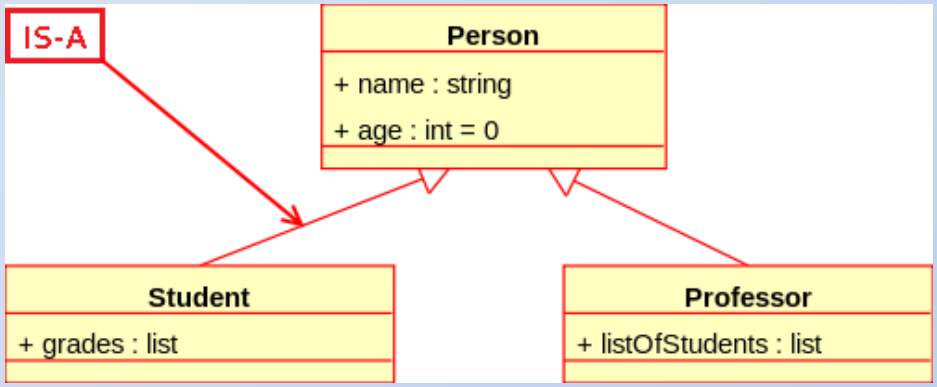
Agenda

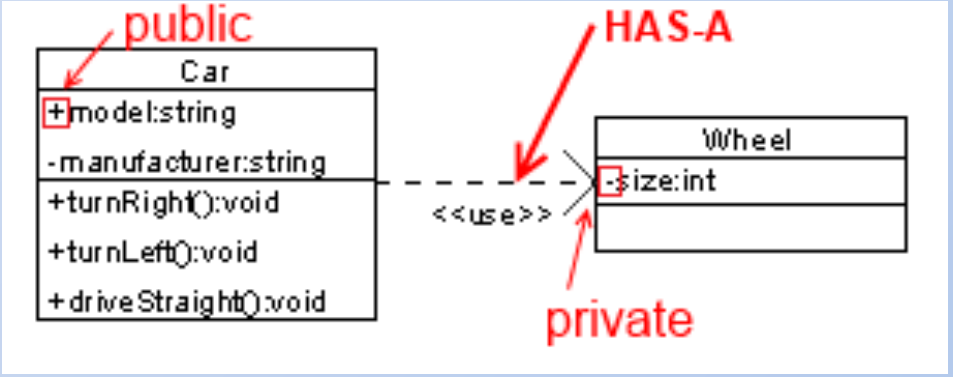
- Class Diagram
- Modular Programming
- Test your code
- Junit
- Basic Design Pattern

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

Class Diagram



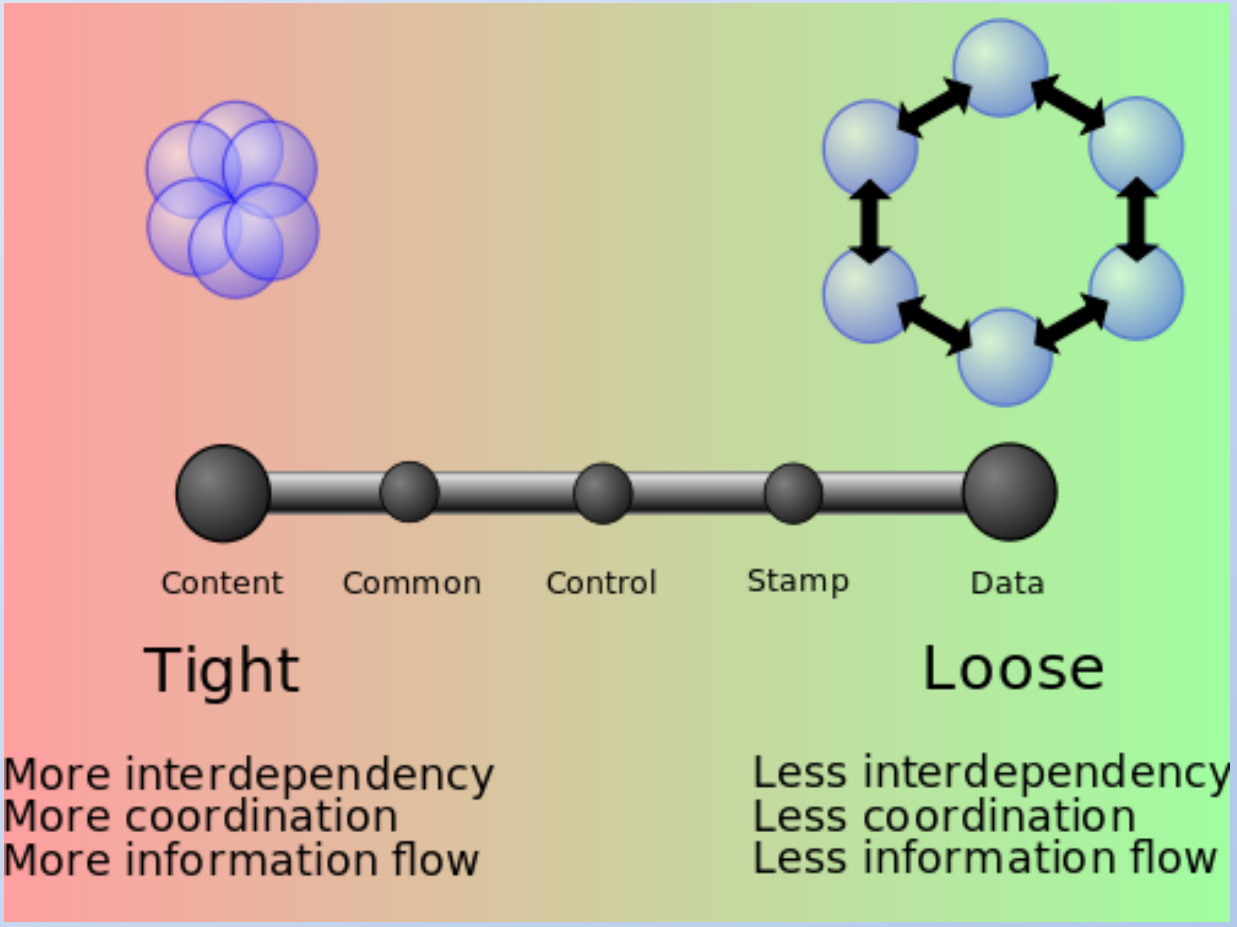




- Class Diagram
- Modular Programming
- Unit Testing
- Junit
- Basic Design Pattern

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 mothod.

- 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

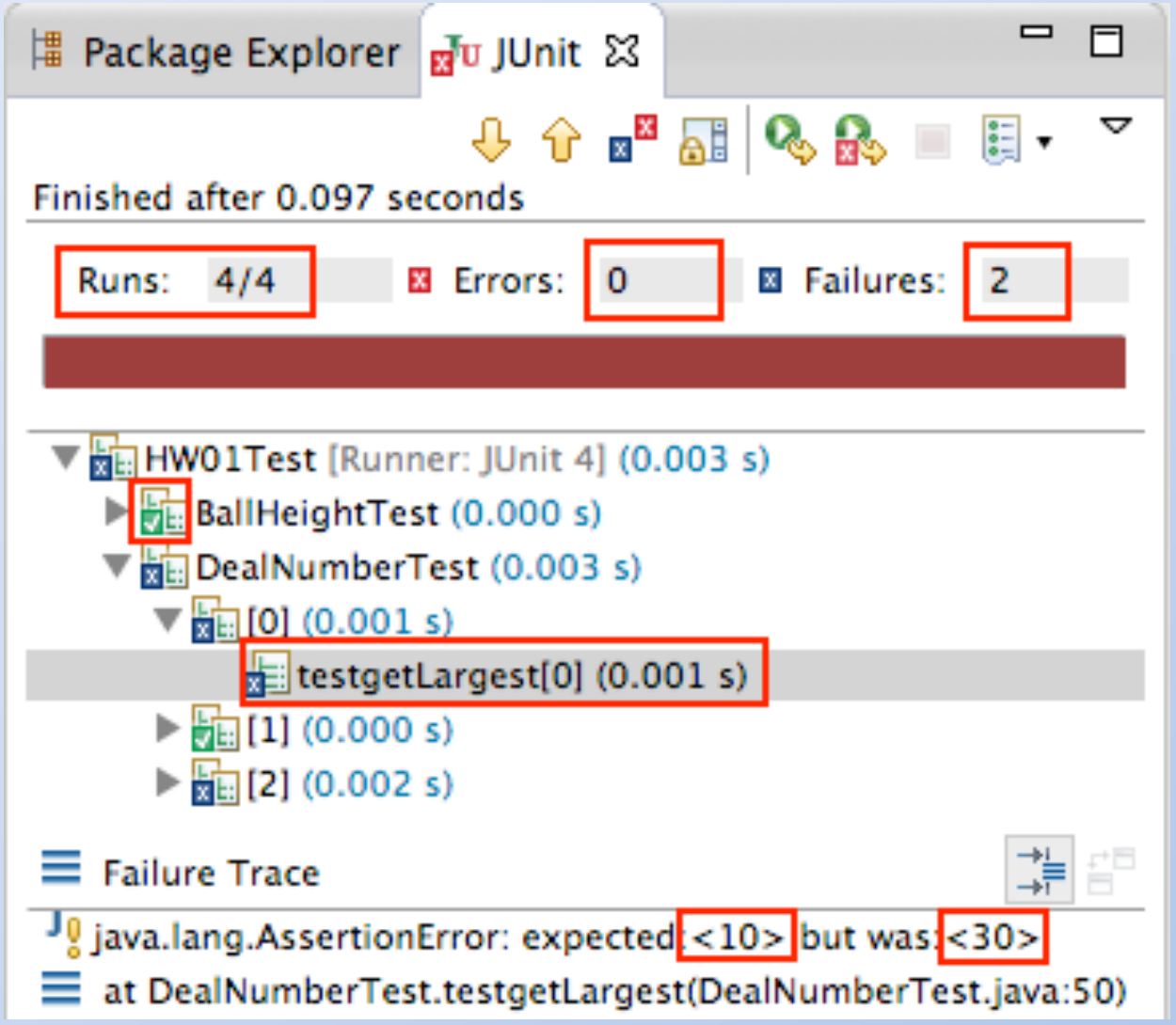
Test Methods

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 assertSame() methods tests if two object references point to the same object
7	**void assertNotSame(boolean condition)** The assertNotSame() methods tests if two object references not point to the same object
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

Sample Code

- DealNumberTest.java
- BallHeightTest.java

Result in Eclipse



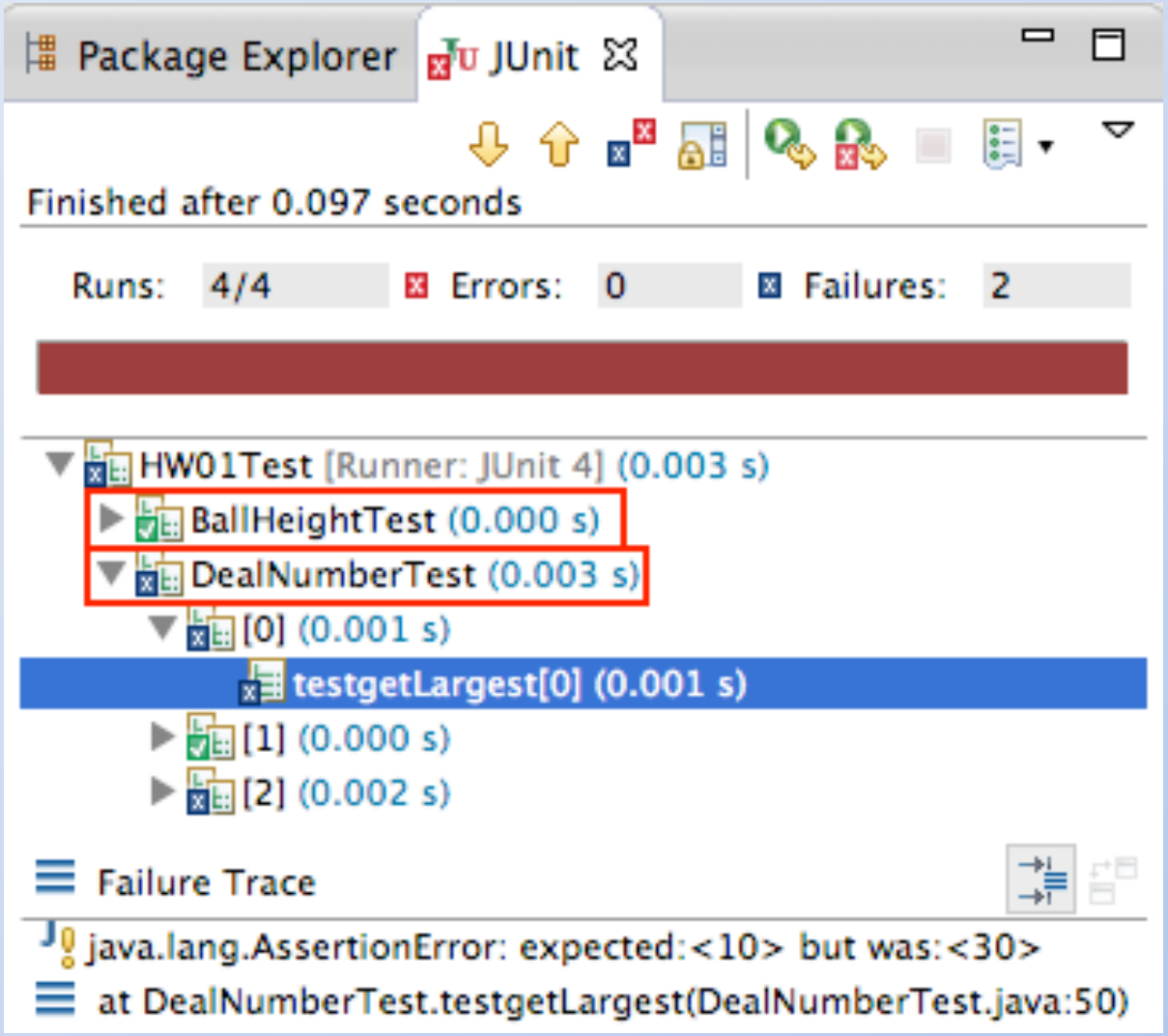
Test Suit

- 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



Exercise

- Please write your own test for your previous homework.

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

Basic Design Pattern

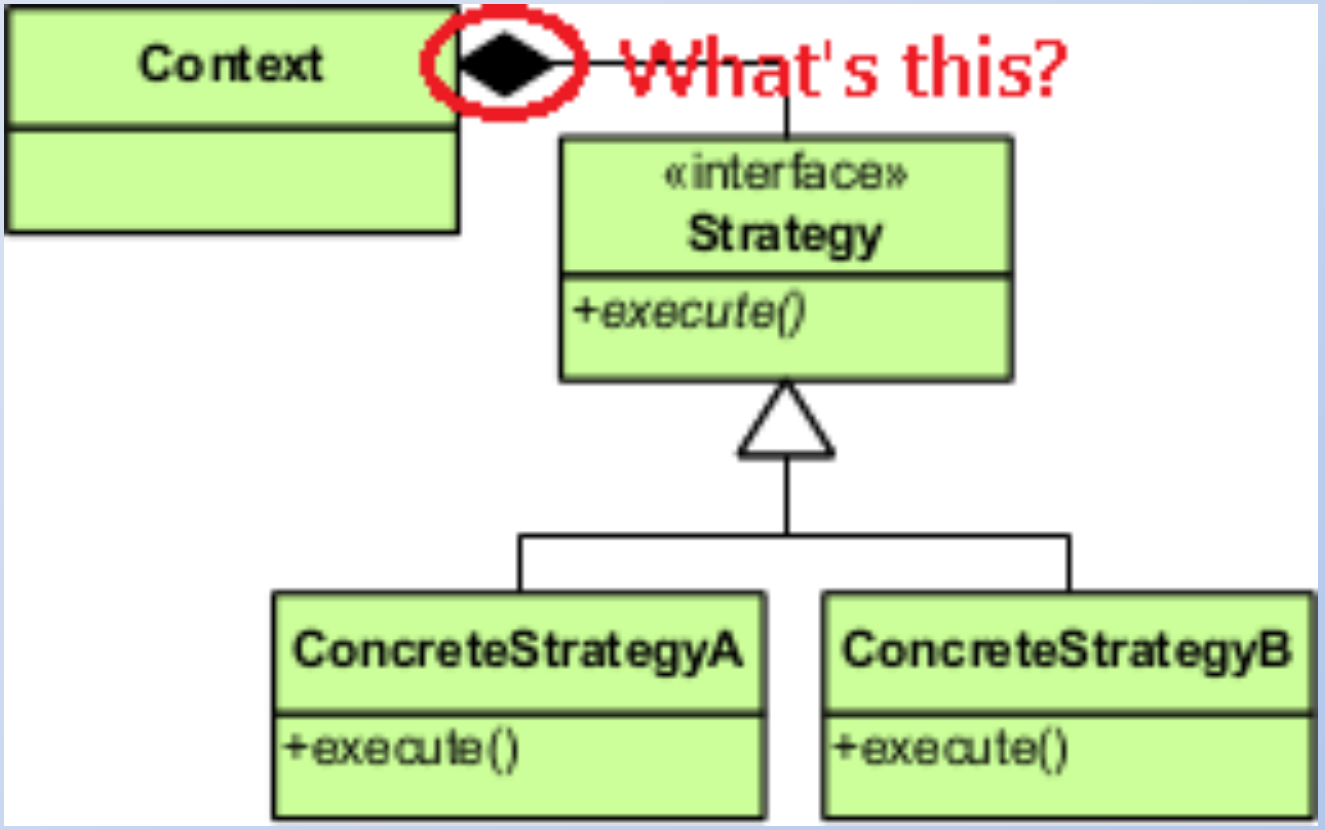
- What is Design Pattern?
- Strategy Pattern
- Observer Pattern

What is Design Pattern?

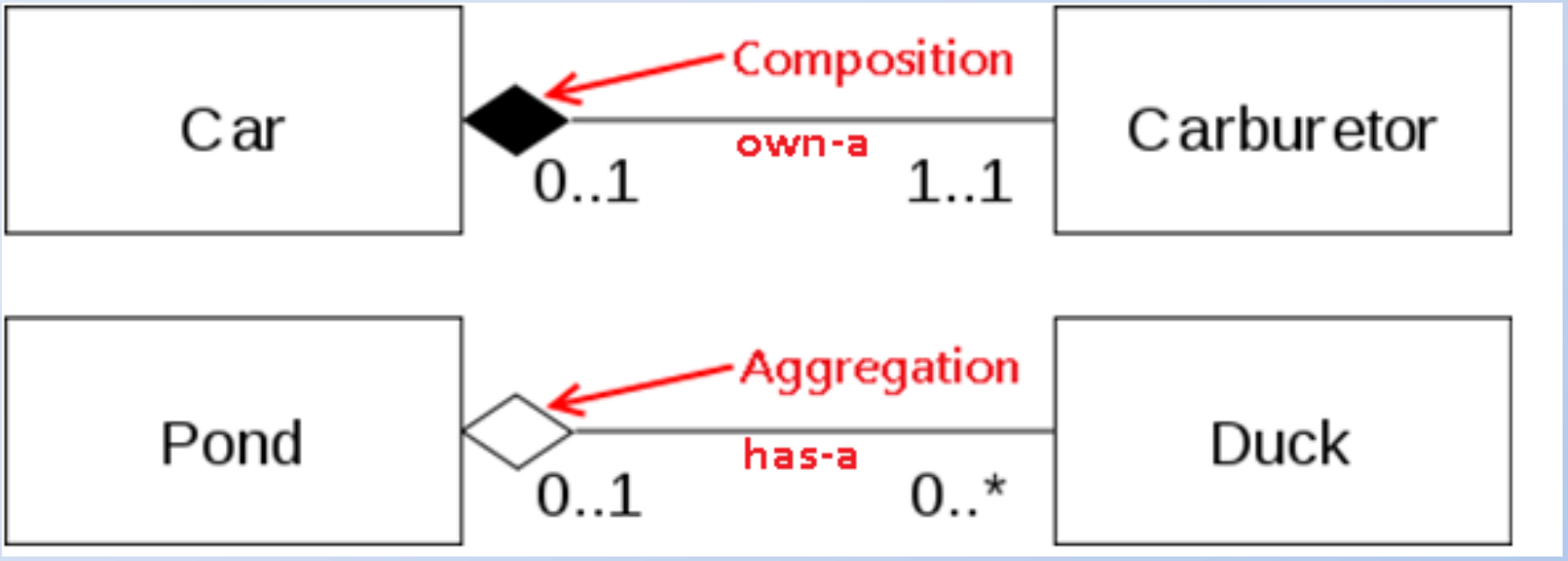
- Definition: A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
- Why you should learn design pattern?

Strategy Pattern

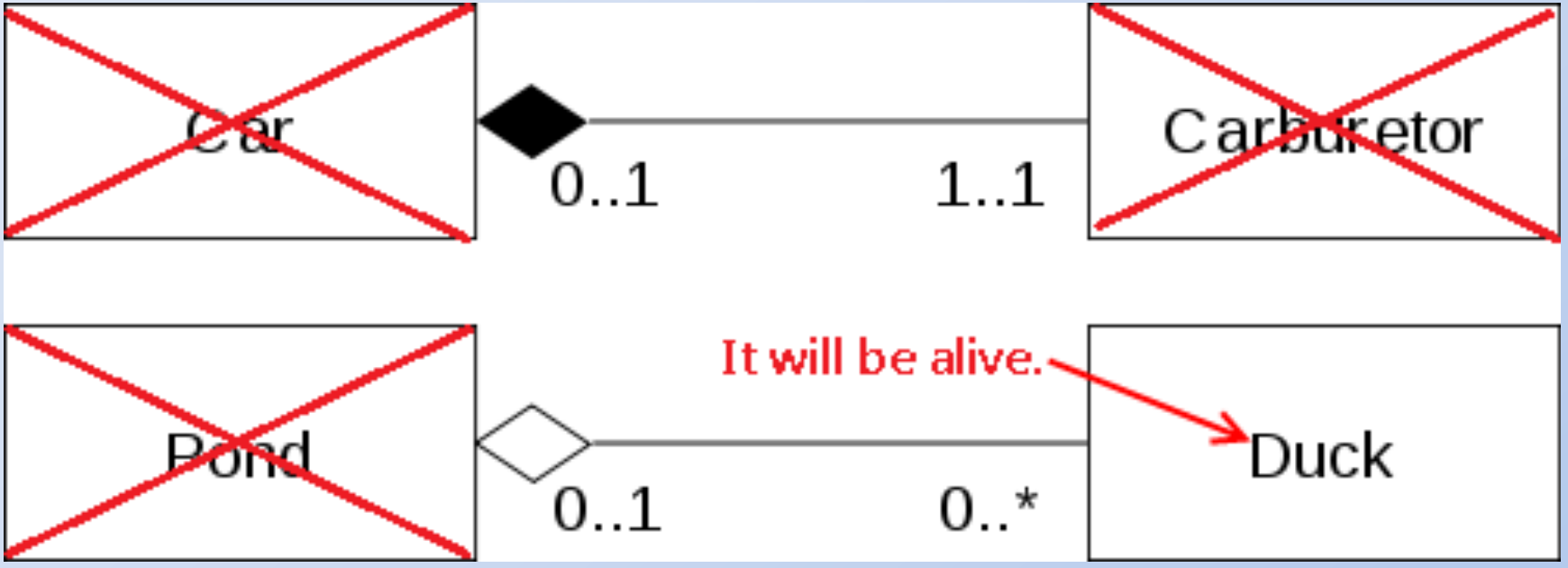
- Subclass decide how to implement steps in an algorithm.



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);
   }
};
```

Strategy Pattern - TestClient

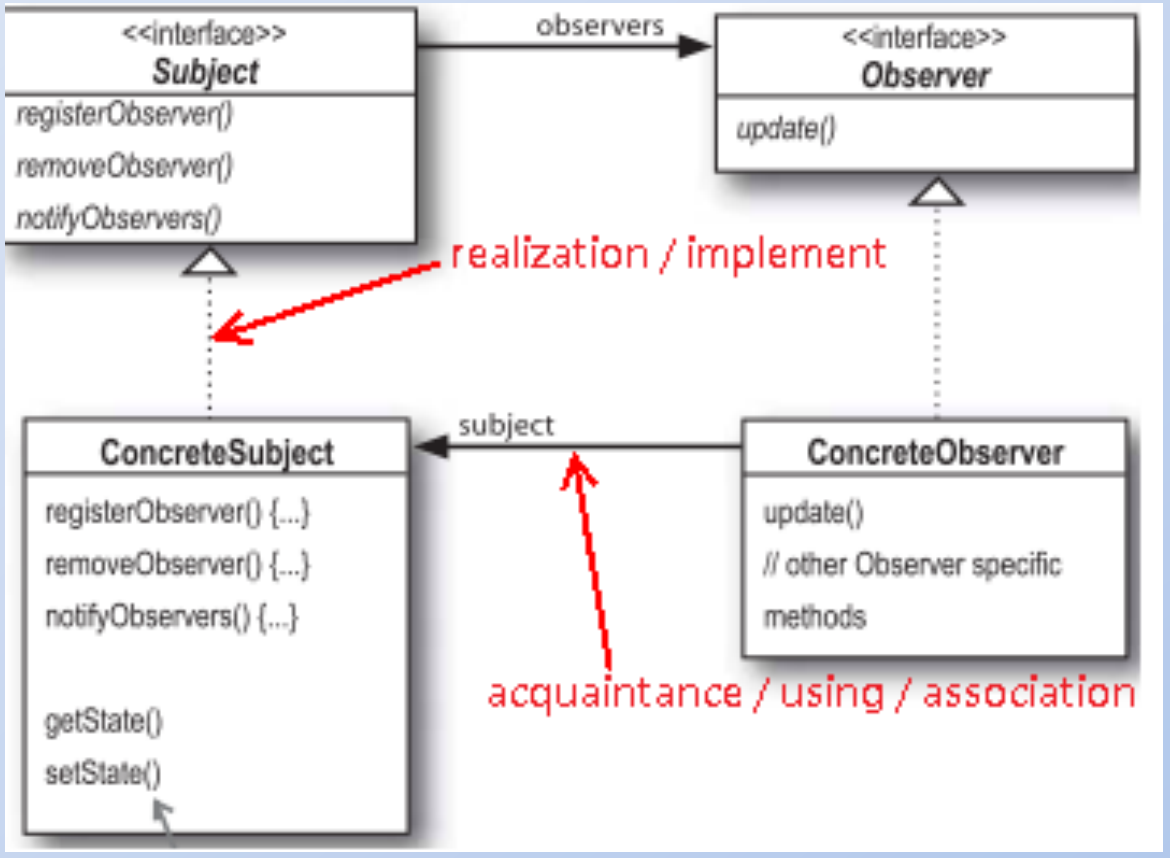
```
class StrategyExample {
   public static void main(String[] args) {
      Context context;

      // 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!