Refactoring

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring.

Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring."

- Martin Fowler
Refactoring

- Put simply, **refactoring** is changing the **implementation** without changing the **interface**.
- One example we mentioned previously involved our DAOs. We may change the implementation to load data from a **database** rather than a **file**.
- In this case the **interface remained** the same. How we interacted with these objects did not change. We called the same methods.
- The **implementation** details **changed**, but that wasn’t important to using the class itself.

Refactoring – Unit Testing

- A key component to refactoring is **testing**.
- Since we are **not** changing the **interface**, we expect to get the same results from **after** our refactoring.
- Develop good unit test cases.
  - Run them **before** the refactoring to get baseline results.
  - Run them **after** the refactoring to ensure they match the baseline.
  - Also, test any **new** cases if necessary.
Let's take a look at an example with playing cards.

Currently there is only a default constructor.

Suppose we have a need to add an additional constructor:

```java
public StandardDeck(IPlayingCard[] cards)
```

Add the code for this new constructor.

Was this easy?

How did you accomplish the new constructor?

Perhaps the simplest approach is:

```java
public StandardDeck(IPlayingCard[] cards){
    myCards = new ArrayList<IPlayingCard>();
    this.add(cards);
}
```

Wait!

- Does this introduce any problems?
- Let's do a UnitTest.
Adding a new constructor, means we should add additional test cases.

(Note: in this demo I am only testing Reset, but in the real world you would test all methods)

```java
@Test
public void testArrayConstructorReset() {
    IPlayingCard[] myCards = new IPlayingCard[4];
    myCards[0] = new StandardPlayingCard(Suit.CLUB, Rank.ACE);
    myCards[1] = new StandardPlayingCard(Suit.DIAMOND, Rank.ACE);
    myCards[2] = new StandardPlayingCard(Suit.HEART, Rank.ACE);
    myCards[3] = new StandardPlayingCard(Suit.SPADE, Rank.ACE);

    myDeck = new StandardDeck(myCards);
    String initialDeck = myDeck.toString();
    myDeck.drawCard();
    myDeck.reset();
    String resetDeck = myDeck.toString();
    assertEquals(initialDeck, resetDeck);
}
```

You will need to **refactor** the `reset()` method. (And maybe other implementation details as well).

Notice, we are changing the **implementation**, but not the **interface**. This means we want the class to function as before, but with a new implementation.
Why?

- The original reset method has two loops that populated the deck with cards.
- Then in the default constructor we simply called the reset() method.
- This won’t work now, because depending on the constructor used, the reset will have to reset the deck differently.

What’s the solution?

- Refactor your reset() method so it correctly resets the deck, regardless of what constructor was used initially.
Like everything else there are options.
- A simple approach:
  - Create a second copy during of the original deck and use it to reset.

Does it work now?
- Try re-running the unit test.
  - It fails? Why?

This line is the cause:
- originalDeck = myCards;

To solve this problem we need to change our code to create new copies, instead of simply assigning them the same value.

- For example, instead of:
  originalDeck = myCards;
- Use:
  originalDeck.addAll(myCards);
More information on Refactoring

- Martin Fowler is a famous OO designer.
- He started a company named ThoughtWorks that does OO development.
- You can find information and examples on refactoring at:
  - http://refactoring.com/
- There is a particularly nice list of refactoring examples here:

Agile Programming

- For large projects it is often beneficial to use an agile programming approach.
- It was originally described as Extreme Programming, but this name scared a lot of business people off.
The basic idea of agile programming is to develop the system with an *incremental, iterative* approach.

Rather than using the traditional workflow approach where all of the analysis and design is done *before* any coding begins, we create smaller functional systems.

It is through the design we *add a small set* of new functionality.

This process allows for a *quick* working prototype that has some of the functionality we need.

In each *iteration* we have a working system, with increased functionality, instead of waiting for until everything is done at once.

This also allows for developers to more timely respond to *requirement changes*.