Added by Rolf Russell, last edited by David S Wood on Jan 08, 2007 (view change)
Labels: (None) EDIT

Chance

Session Objectives

By the end of this session students should ...

- Be able to pair program
- Understand and use the test-code-refactor cycle
- Think about design before they code
- Be able to proficiently use the IDE
- Understand the value of custom classes over primitives
- Understand the concepts of objects churning on each other
- Understand logical equality vs. object identity
- Understand the use of guard clauses
- Follow the general contract when overriding equals(Object)
- Know the 4 principles of simple design
- Understand and apply YAGNI (You Ain't Gonna Need It)
- Understand and apply DRY (Don't Repeat Yourself)

Session Overview

Activity
Lab - Designing Chance
Lab - Finding the First Test for not()
Lab - Implementing the First Test for not()
Lab - Implementing not()
Mutable vs Immutable
Four Principles of Simple Design
Lab - and() & or()
Lab - equals()

Session Notes

Lab - Designing Chance

Ask the students to model the chance of something happening. Flip a coin and ask the class what the probability of getting heads is. Roll a die and ask what the chance of getting a '4' is.

Remind the students that design is an important part of coding. Do 5 minutes of design discussion as a class. Tell the students they will implement a Chance class. Elicit:

- What the job of Chance is: Understands the likelihood of something happening
- What properties Chance would have:
  - a value representing the mathematical probability
  - for example how would you represent a coin. (0.50)
  - what constraints does that property have? (between 0 and 1 inclusive)
- If you roll a die you have a probability of getting a '6'. You can also have a probability of not getting a '6'. Is that something that our Chance class might do? yes
- What other behaviours might Chance have:
  - two things both occuring (and)
  - one of two things occuring (or)

If the students struggle, make sure they are separating the chance concept from particular events. Tell them to leave the event out and just focus on the mathematical concept of 'chance'.

Ask the students what they might name the behaviour of something not occuring?

Lab - Finding the First Test for not()

Tell the students they will implement not(). Give them 5 minutes to write the first test. Suggest writing the assert statement first in a test method. If you can't come up with an assert statement, it means that you have more design to do.

Common mistakes

- Return a double out of not() - What can you do with a double? Its a magic number that doesn't mean anything. Isn't it the probability of something NOT occurring? What should not() return? - a Chance
- Create a method that returns the probability of the Chance as a double - breaks encapsulation
- Don't test constructor. you can assume the java language works.

Discussion

- Ask what they tested first? They need to test equals() first because any test of not() requires assertEquals(). So equals() is the first small chunk.
- Tell students this is how the class will work. We will be showing peoples' code on the projector and discussing it. So don't be shy,

Lab - Implementing the First Test for not()

Give the students 15 minutes to get to a green bar. Be firm on the time.

Common mistakes

- Not chosing the simplest test first. The students should be asserting identity equals. If it takes them more than 15 minutes to get to a green bar then they probably have bitten off more than they can chew.
- Missing job for Chance
- Missing job for ChanceTest (but maybe tests don't need jobs)
- Using an unclear name for the parameter in the constructor (for example: 'value'). What is the 'value'? Its a ratio or fraction representing the probability. 'valueAsFraction' is better.
- Using an unclear name for the parameter in equals(). For example 'probability' is not a meaningful name. 'other' reads much more clearly.

Lab - Implementing not()

Ask the students to implement not(). Since the object of the lesson is not how well you already know probability, instructors may provide the formula

Formula

P(!A) = 1 - P(A)

Common mistakes

- Magic number for '1' in not() --> GUARANTEED_VALUE

Mutable vs Immutable

This should come up when a student implements not() by modifying an existing Chance object. If not then bring it up before finishing Chance.

> Mutable objects are objects that can be changed after they are created. Immutable objects are objects that cannot be changed and must be entirely set up in constructor.
> Modifying the value of the Chance object is problematic because others might be depending on it. What will happen to them?
> Use immutable when:
>
> - Creation is cheap - it is easy to create a new Chance, harder to create a new Lease
> - Object is intrinsically tied to its value - changing the chance value changes the whole object
>
> In general immutable objects are safer - they can be passed around and will never change. It is a good idea to make small, cheap objects immutable.
> To make objects immutable you declare the fields final.

Value Object

> Modern OO systems (Java, C#) have the concept of object equality build into them, based on the objects place in memory. One object reference is equal to another object reference if and only if they occupy the same space in memory. In general, this makes sense for objects that are mutable, in that changes in their state do not change the fundamental nature of the object. For instance, a person is still the same person if their name changes, and a Person class should reflect this fact. However, there are certain concepts in the world where the concept of equals is slightly different. Object representations of Money, Probability, and Complex Numbers don't follow the same rules for equality. There is some internal value in each of these concepts that should be used to determine equality, because $1 is always equal to $1, even if it is being represented by two objects that have different locations in memory. Changing this internal value fundamentally changes what the object represents, in contrast to changing a person's name not doing so. These types of object, in which the internal value is used to determine equality, and changing the value changes the identity of the object, are fundamentally immutable, and can be referred to as Value Objects.

Four Principles of Simple Design

- It works
  - At the most basic level the code must do what it is supposed to do. green bar - tests pass.
- It communicates
  - You should be able to read the code directly. The code itself should express its intent. The best world is where the code reads like english. people have dozens of years of practice understanding english and the closer your code gets to english the easier will be to read. That is why we use full parameter names ('probability' instead of 'p') and we use method names that clearly express the intent of the method. If your code requires a lot comments then the code itself is not communicating. Seeing comments in your code likely points to an opportunity to improve the way the code communicates and then delete the comments.
  - Short method names are prefered to long method names.
    - Long method names often indicate the method is doing too much and could be broken into simpler methods. for example if your method name has an 'and' in it then it is probably doing too much
  - Code should be easily understandable. if there are too many words it becomes harder to read and understand.
- No duplicate code
  - Duplicate code creates barriers to maintanence and leads to the strong possibility of eventual contradiction. to work with duplicated code you have to know that it is duplicated and make modifications in all locations
  - DRY (don't repeat yourself)
- No extra code
  - Cost of ongoing maintanence

- Having the extra code may inhibit good factoring of your code and hide valuable abstractions
- You can't predict the future: your understanding of the system/design changes, requirements change, etc
- YAGNI (you ain't gonna need it)

Lab - and() & or()

Ask the students to implement and() and then or(). Instructors may provide the following

> **Formula (events that are not mutually exclusive)**
>
> P(A && B) = P(A) * P(B)
> P(A || B) = P(A) + P(B) - P(A) * P(B)

Common mistakes

- Using an unclear name for the parameter in and() or or(). For example 'probability' is not a meaningful name. 'other' reads much more clearly.
- Duplication (new Chance(0) many times in tests). This indicates the students were not conducting the refactoring part of the cycle.
- Don't need to use setUp() in tests. JUnit creates a new instance of the class for each test anyway. For complex setups the method is useful but for creating simple test objects it wastes space.
- Duplicating the and() and not() method code in or() instead of calling them.

Discussion

- DeMorgan's law: P(A||B) = !(!P(A) && !P(B))
- Churn
  - Ask class how many objects are used in or()? 6
  - This is a great example of *churn*: objects collaborating with other objects to perform a task. Churn is a good thing in this sense. you will see plenty of it in this class.
  - Why is all this object creation OK? After the objects churn only 3 survive, others are garbage collected. Creation of these short lived objects was a problem in some of the first OO systems, but analysis has led to the realization that there are many short lived objects (like the ones above) and many fewer long lived objects. This has led to a GC technique called "Generational Garbage Collection" that optimizes memory allocation and reclamation for systems that exhibit this type of behavior.
- toString()
  - Usually it will be difficult to debug failing tests so we will need to implement toString().
  - toString() should not be used in production code. It is valuable to help debug.
    - As such it does not require tests.
    - This prevents it from being used as a workaround to break encapsulation.
- When in Chance might a comment be valid? // demorgan's law

Lab - equals()

Ask the students to fully implement equals() for Chance.

Discussion
The 4 steps of an equals() method:

1. Identity check - optimisation because high percentage of calls to equals() are on the object itself
2. Null check - "if (o == null) return false;"

3. Type check - "getClass() != [other.getClass]()"
    - Introduce guard clauses - simple condition that protects a later statement. For example the null check protects the type check.
    - getClass() is better than instanceof because:
        - Instinctively a subclass instance should not equal a super class instance
        - You are not required to override equals() in subclasses so if you have Animal, Goat and Pig and only Pig overrides the equals() from Animal, then [goat.equals](pig) does not call the same method as [pig.equals](goat), which violates the symmetric property of the equals() contract.
4. Actual check - "return (ratio == [other.ratio])"

If logic of the actual check is complex, you should extract out "private equals(Foo clazz)"

Test things you think could fail (ie. not java language or getters/setters)

- assertNotEqual(chance, null)
- assertNotEqual(chance, new Object())
- assertNotEqual(chance, new Chance(0.25)
- assertEquals(chance, new Chance(0.75))

The equals() contract:

- hashCode() must be implemented as well: Two objects that are equal must have the same hash code. Other than that, the details of implementation are not important.
- It cannot throw exceptions
- Reflexive: [a.Equals](a) = true
- Symmetric: [a.Equals](b) = true means [b.Equals](a)
- Transitive: [a.Equals](b) and [b.Equals](c) means [a.Equals](c)
- Repeatable: if [a.Equals](b) true now, and the objects don't change, then [a.Equals](b) = true later

Children Hide Children | View in hierarchy

Four Principles of Simple Design (Training)
Lab - and() & or() (Training)
Lab - Designing Chance (Training)
Lab - equals() (Training)
Lab - Finding the First Test for not() (Training)
Lab - Implementing not() (Training)
Lab - Implementing the First Test for not() (Training)