Added by Rolf Russell, last edited by David S Wood on Dec 11, 2006 (view change)
Labels: (None) EDIT

Quantity

Session Objectives

By the end of this session students should ...

- Understand the three relationships between objects and their appropriate usages
  - Inheritance - is a (includes interfaces)
  - Delegation - one-way relationship
  - Collaboration - two-way relationship
- Understand a good example of using the fewest classes possible
- Be able to effectively use slugs
- Look at 'else' and 'if' as suspicious statements
- Understand when to use checked and unchecked exceptions
- Be able to test exceptions
- Be able to create factory methods where appropriate

Session Overview

Activity
Object Interaction
Lab - Square
Lab - Length
Lab - Volume
Lab - Addition
Lab - Temperature

Session Notes

Object Interaction

Point out that we have talked about some ways that objects interact today. Elicit the 3 main ways:

Delegation (one-way relationship)

- Class A uses/contains B, but B doesn't know about A
- Also called *buying* another object
- Example: Addresses may contain a List but does not need to expose all of List's functionality

Collaboration (two-way relationship)

- Both objects know of each other and work together
- Symboitic relationships like a shark and a remora
- Example: ??????
- Example: the implementation of or(). Collaboration does not necessarily imply separate classes, only separate objects.
- In a large system you want to have many small tight relationships. you don't want all of your objects interacting, but rather want small groups to work closely together and then have well defined boundaries between the groups. (see wendy's notecard)

<u>Inheritance (is-a relationship)</u>

- Should be used only for *is a* relationships, not simply as a way to reuse code.
- No example now. Maybe we will find one during the class.
- Two tests for inheritance
    - Grandmother test - Would your grandmother (or some other non-programmer) agree that the subclass *is a* superclass?
    - Want most of the functionality from the super class

Ask students what interaction to prefer?

1. First delegation/buys relationship
2. then consider collaboration
3. and finally inheritance.

Explain students the fact that when two objects collaborate, there occassionally needs to be a contract between them (interfaces) to make sure that implementation changes can be isolated.

Lab - Square

Let's go back to rectangle for a minute. Ask the students to implement square.

<u>Things to look for</u>
2 to 3 solutions will appear

- Inheritance (class Square)
    - Where is the kick? doesn't have any methods - shouldn't
    - How is a square different from a rectangle?
    - Since the only thing that is different is how you construct it, couldn't you just use another constructor? Yes, but for readability purposes you may want to look at making the constructor private and public static creation methods (rectangle()/square()).
- Just use Rectangle: new Rectangle(3,3)
    - Does not reveal intention
    - Encodes domain logic in the test or wherever you create this "square"
- One arg constructor: new Rectangle(3)
    - The one-arg constructor should call the two-arg constructor
    - Encodes domain logic implicitly - does it actually make sense to have a "one-sided" Rectangle?
- Use a factory method - [Rectangle.CreateSquare](10) - this can call the Rectangle constructor.
    - This is clearer, has fewer classes. better expresses intent.

Lab - Length

Ask the students when does 3 != 3? When there are units involved. The Mars satellite was lost due to mismatching units - metric and imperial.

Tell the students they will implement this. Write '12in = 1ft' on the board. Give the students 5 minutes to start designing. Point out that the goal of design time is not to find the ultimate design, it is to find a good starting point. A good goal is to find your first test. Then you work from there.

Elicit designs and discuss.

Add '3ft = 1yd' and '1760yd = 1mi' to board. Ask the students to implement.

Things to look for

- A class encapsulating both amount and unit - Quantity
    - To help explain the Quantity pattern you can give an example: two random physical objects in room: meters and feet. Stand up and hold up a number and point to one of the objects. I am 5 feet. I am 3 meters.
- Jobs for classes and test classes.
- Representing unit as a String (or final int). This means that any value could be passed in. But there are a limited number of valid units.
- Only need one Unit class. Don't need separate Feet & Yard classes.
    - What is the job of Feet vs. job of Yard
    - Duplicated behavior between Yard and Feet
- Unit holding the conversionFactor but not converting to inches.
    - Violates encapsulation to let Quantity see the conversionFactor. Quantity trying to convert to inches has feature envy.
    - Classes need kick. A class has to do interesting work. It can't just be a data holder.
- Refactor to standard solution
    - One Unit class with a variable for conversion to a base unit (feet)
    - Unit has private constructor
    - Different implementations of this Unit for Feet, Yard, etc. are stored as slugs on the Unit class
    - No need to override Equals on Unit - want different units to be unequal
    - Quantity class stores an amount and a Unit

Discussion

- *Most suspicious statement lecture*. Now that goto is gone something else has stepped into its place as chief troublemaker: *else*. 'Else' is the most suspicious code statement, followed closely by 'if'. Often 'if' and especially 'if-else' statements are hiding valuable abstractions and can often be replaced by polymorphism. Later we will see some design patterns that are if-killers: strategy, state, etc. The more branches in your code the more complex it can be to understand. This can be measured by cyclomatic complexity, which counts the number of different paths you can follow and can help you target code for improvement.
- Point out that this is an example of delegation. Quantity calls Unit, but Unit does not know about Quantity.

Lab - Volume

Introduce volume - Write '1tbsp = 3tsp', '1oz = 2tbsp', '8oz = 1cup' on the board. Ask students to implement.

Things to look for

- *Slug* pattern - inheritance is not necessary.
    - "type" in unit
    - like the slugs that would fool the slot machines.
    - its value is in its identity which can be used to differentiate slugs with the == method (and thus with the default equals() method)
    - just use a plain [java.lang.Object]
- Parallel inheritance trees (ex. Length & Volume classes that are identical). This is a common code smell.

Lab - Addition

Write '2in + 2in = 4in', '2tbsp + 1oz = 12tsp' on the board. Ask students to implement.

If the students don't discover the need to handle adding incompatible units themselves, then add '2ft + 2tbsp' to the board.

Things to look for

- Throwing a RuntimeException in the add() method when trying to add quantities with incompatible units.
- Need a unit type - watch out for strings/int as unit type
    - Maybe new class for unit type? Show them the code - no behaviour => no class
    - Should unittype be on the Quantity? No - violates encapsulation.
    - Unit stores an object for unit type
    - The types are slugs on Unit - private static readonly object DISTANCE = new object();

Exceptions

- Contract of an unchecked (RuntimeException)
    - if a programmer could check for the issue before calling add() then it can be thrown as a RuntimeException. In other words if the programmer could have avoided the error then a RuntimeException can be thrown. If you ask me a complete nonsense question then all I can do is tell you it was a nonsense question - throw an exception. I can't fix the situation.
    - This has to be balanced against bleeding the interface. For example an Stock class interface should not show a SQLException. This can be done by wrapping exceptions either with checked or runtime exceptions. We are happy with the existing runtime exception though.
    - The only option to get rid of the runtime exception would be to introduce many subclasses for each type of Quantity (length, temperature, etc). This would be bad because it would be (1) an explosion of subclasses and (2) they wouldn't have any different behaviour.
- Use exceptions for exceptional cases only, when the program is in a state that it should not be in. Do not use them for flow control.
- Create custom exceptions - easier to catch the correct exception in the calling method
- Layered exceptions - different types of exceptions for database, domain objects, etc.

Lab - Temperature

Write '212F = 100C', '32F = 0C'. Ask students to implement.

Discuss progress so far.

Typically the students will not have thought about the impact of temperatures to on addition. Tell the students to implement '100C + 100C'. Ask them what the answer is? It isn't '200C'. If you add boiling water to boiling water do you get '200C'? Temperatures cannot be added. Convince the students using the *Addition of temperatures discussion* below. Ask the students what this means for temperature addition? Lead them to the conclusion that it should not be possible to add temperatures (ie. can't even call method). Ask students to implement.

Things to look for

- At first students may throw an exception from add() if it was adding temperature. Ask them to prevent adding of temperatures all together. This will result in subclassing Quantity.
- Inheritance on Quantity, not on Unit because add() is on Quantity.
- Introducing subclass is a refactoring and thus does not require failing tests. This is because it was not changing functionality, only implementation. Refactoring steps:
- It is possible to create a scalar inches or an arithmetic celcius. This is wrong. How can it be prevented?
    - create factory methods for quantities
    - once complete note that Unit no longer needs to be visible outside of quantity package. The class and its factory methods can be package protected.
    - Can inline variable declarations in QuantityTest at this point as they are now human readable.

Discuss refactoring steps
Below is a sample set of steps students took to create the inheritance. (it is not the only valid way):

1. use extractsuperclass & pullmembersup
2. make sure that superclass equals() method abstracts parameter and return value to be a ScalarQuantity

3. make sure that the variables in tests have the write type (most generic type possible)
4. throw away test that ensures add() throws exception for a temperature
5. update jobs on super and sub classes

Addition of temperatures discussion

Measurements come in four levels, including:

- *state* or *nominal*:
    - names are assigned to objects as labels
    - only equality comparisons can be made
    - examples: blood type or gender.
- *ordered* or *ordinal*
    - values are assigned that represent a rank order (first, second, third ...)
    - comparisons of greater or less than can be made in addition to equality
    - examples: social class or liver color going from A to C, with A being fine and C being bad
- *scaled* or *interval*:
    - equal differences between measurements represent equivalent intervals. in other words differences between two pairs of measurements can be meaningfully compared.
    - examples: temperature or year date
- *arithmetic* or *ratio*
    - operations like addition, multiplication and division are meaningful
    - examples: volume and distance
- See http://en.wikipedia.org/wiki/Levels_of_measurement for more details.

Note that each tier includes all the behavior of the previous tier and adds more behaviour.

Children

Hide Children | View in hierarchy

Lab - Addition (Training)
Lab - Length (Training)
Lab - Square (Training)
Lab - Temperature (Training)
Lab - Volume (Training)
Object Interaction (Training)

1 comment | Add Comment