Test driven development of a Stack

Pieter van den Hombergh

September 21, 2015

Test Driven Develop Two Stacks.

Derive the test from the requirements of the stack. Use the design from the diagram. When implementing, you create an

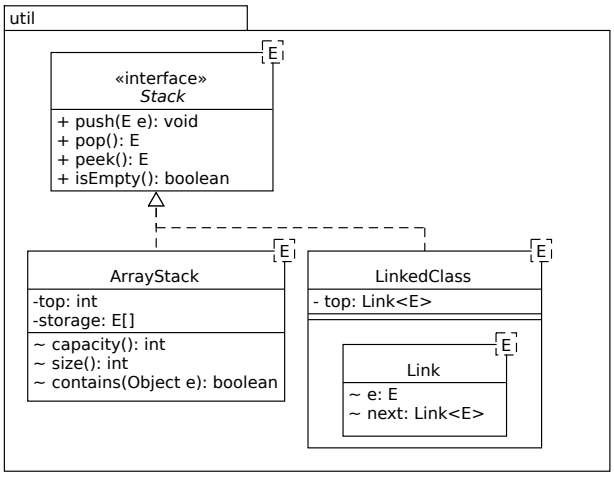

Figure 1: Stack design and alternative implementation classes

array based stack first, called `ArrayStack<E>`. After that is complete, you create a `LinkedStack<E>`, starting from the same requirements and tests. Do you see a way to reuse code (or test code) here? Hint: Think of a class hierarchy in the test classes too. But first, start simple, then refactor.

TDD: Write your tests first. Derive them from the requirements given below.

Requirements of a Stack

- The Stack should be generified, that is accept a Type parameter on instantiation as in `Stack<E>`.
- An empty Stack reports `isEmpty()` true.
- When you `push(x)` an element onto a stack, `peek()` should return the *same* object `x`.
- When you `push(x)` an element onto a stack, `pop()` should return the *same* element `x` and that element should no longer be on or in the stack.
- When you push the elements from a `list(a,b,c,d...)` in this order onto a stack, and then `pop()` the elements of that stack, the elements should come off in the reverse order.
- For the implementations:

  ArrayStack An `ArrayStack` should have the following additional package private methods. They are to be used in tests:
  - An ArrayStack should be able to report it's current size (how many elements are stored).
  - An ArrayStack should be able to report it's current capacity (how many elements can be stored).
  - An ArrayStack has a method to check if an Object is contained, to verify that the requirement to actually erase the element on pop is implemented.

  LinkedStack A linked stack has no extra requirements beyond the methods defined in the interface.

Implementation hints

Work test driven. Derive a test method (and its name) from the requirements and code it in the same package as the Stack interface.

For instance, name your first test method `public void a_new_stack_is_empty()`.

Let the IDE do the work

If your start with declaring the complete interface, you may be best of by first implementing a `DummyStackAdapter` that implements all methods to satisfy the compiler and runtime. Simply accept what NetBeans IDE generates for you, thrown exceptions and all. Then *extend* this `DummyStackAdapter` with the class that you want to implement, implementing the methods in "natural" order. When you are done with all methods in both implementation, throw the `DummyStackAdapter` away (it is useless anyway, since all its methods are overwritten, right) and modify the `extends DummyStackAdapter<E>` to `implements Stack<E>`.

Hints to ArrayStack

- Internally use an array of Object.
- Design the stack on paper using pen or pencil (pencil is best). Good names and types for the members are `E [] storage;` for the array and `int top;` for the member that knows where the top-most element is. Initialize the members properly, either in a constructor or initialize at declaration.
- Do not move or copy the elements around on push or pop. Just update your `top` to do the administration.
- An initial capacity of 4 should do.
- This stack may overflow its array, so before you push, make sure you have sufficient capacity. Use a separate method that does so, call it `ensureCapacity()`.
- There is no need to shrink the stack back again when elements are popped off.
- When you pop an element off the stack, make sure it's place in the storage is overwritten with `null` to prevent leaking memory.

LinkedStack hints

Imagine the structure of the linked stack as a chain with links and each link carrying an element.

Make a drawing with paper and pencil. Draw a stack with three elements stored in it. Scan it in or make a photo with your smart phone or tablet and add the result to the repository. A nice touch would be to trim the scan file down to size and include it in the javadoc for the linked stack.

- Use a static inner class called `Link`. Give it a final field of type E, named e and a field next, which can hold a reference to another link.
- The stack is empty when `top == null`.
- The chain "hangs" by its top.
- Push creates a new link, adds the current top as next and assigns this new link to top.
- Pop takes the element from the top element and assigns top.next to top.
- Peek simply returns top.element.
- The LinkedStack does not need a `size()`, a `capacity()` or `contains(Object o)` method.