CSE 331, Spring 2011
Section 4 cheat sheet

JUnit, OO design patterns

JUnit 4

Method annotations:

tag	description

`@Test`
`@Test(timeout = time)`
`@Test(expected = exception.class)`
	Turns a public method into a JUnit test case. Adding a timeout will cause the test case to fail after **time** milliseconds. Adding an expected exception will cause the test case to fail if **exception** is not thrown.

`@Before`	Method to run before every test case
`@After`	Method to run after every test case
`@BeforeClass`	Method to run once, before any test cases have run
`@AfterClass`	Method to run once, after all test cases have run

Assertion methods:

method	description
`assertTrue(`**test**`)`	fails if the Boolean test is `false`
`assertFalse(`**test**`)`	fails if the Boolean test is `true`
`assertEquals(`**expected, actual**`)`	fails if the values are not equal
`assertSame(`**expected, actual**`)`	fails if the values are *not* the same (by ==)
`assertNotSame(`**expected, actual**`)`	fails if the values are the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

Each method can also be passed a string to display if it fails, e.g.

```
assertEquals("message", expected, actual)
```

Unit testing tips:

- The entire goal is ***FAILURE ATOMICITY*** – the ability to know exactly what failed when a test case did not pass
- Tests should be self-contained and not care about each other
- You cannot test everything! Instead, think about:
  - boundary cases,
  - empty cases,
  - error cases
  - behavior in combination (but not to excess)
- Each test case should test ONE THING
  - 10 small tests are better than 1 test 10x as large
  - Rule of thumb: 1 assert statement per test case
  - Try to avoid complicated logic
- Torture tests are ok, but only *in addition* to simple tests

JUnit best practices:

- Use descriptive test names
- Add a default timeout to every test
- Use private methods to get rid of redundant test code

- Create test suites using `@RunWith` and `@Suite.SuiteClasses` to run tests for several classes at once
- Build quick arrays and collections using array literals
    - `int[] quick = new int[] {1, 2, 3, 4};`
    - `List<Integer> list = Arrays.asList(7, 4, -3, 18);`
    - `Set<Integer> set = new HashSet<Integer>( Arrays.asList(5, 6, 10) );`

OO design patterns

"A standard solution to a common software problem in a context."

Iterator

Problem: To access all members of a collection, must perform a specialized traversal for each data structure.

- Introduces undesirable dependences.
- Does not generalize to other collections.

Solution: Provide a standard iterator object supplied by all data structures.

- The implementation performs traversals, does bookkeeping.
- The implementation has knowledge about the representation.
- Results are communicated to clients via a standard interface.

Disadvantages:

- Iteration order is fixed by the implementation, not the client.
- Missing various potentially useful operations (add, set, etc.).

Adapter

Problem: We have an object that contains the functionality we need, but not in the way we want to use it.

- Cumbersome / unpleasant to use. Prone to bugs.

Solution: Create an adapter object that bridges the provided and desired functionality.

Singleton

Goal: Create and interact with an object that is the only object of its type. Includes:

- Ensuring that a class has at most one instance.
- Providing a global access point to that instance.
    - e.g. Provide an accessor method that allows users to see the instance.

Benefits:

- Takes responsibility of managing that instance away from the programmer (illegal to construct more instances).
- Saves memory.
- Avoids bugs arising from multiple instances.

Implementation in Java:

- Make constructor(s) private so that they cannot be called from outside by clients.
- Declare a single private static instance of the class.
- Write a public `getInstance()` or similar method that allows access to the single instance.
    - May need to protect / synchronize this method to ensure that it will work in a multi-threaded program.

Flyweight / Interning

Problem: Redundant objects can bog down the system.

- Many objects have the same state.

Solution: An assurance that no more than one instance of a class will have identical state.

- Achieved by caching identical instances of objects.
- Similar to Singleton, but one instance for each unique object state.
- Useful when there are many instances, but many are equivalent.
- Can be used in conjunction with the Factory Method pattern to create a very efficient object-builder.
- Examples in Java: `String`, `Image`

Implementation in Java:

- Works best on immutable objects
- Class pseudo-code sketch:

```
public class Name {
    ✓ static collection of instances
    ✓ private constructor
    ✓ static method to get an instance:
        if (we have created this kind of instance before):
            get it from the collection and return it.
        else:
            create a new instance, store it in the collection and return it.
```

Prototype

Problem: Client wants another object similar to an existing one, but doesn't care about the details of the state of that object.

- Sometimes client doesn't even care about the object's exact type.

Solution: Clone or copy the object's state into a new object, modify as needed, then use it.

- Often closely related to Java's clone method.
- Sometimes done with producer methods that return new objects.

Factory

Problem: Object creation is cumbersome or heavily coupled for a given client. Client needs to create but doesn't want the details.

Factory Method pattern

- A helper method that creates and returns the object(s).
- Can return subclass objects if so desired (hidden from client).

Abstract Factory pattern

- A hierarchy of classes/objects, each of which is a factory for a type.
- Allows hot-swappable factory to be used by a given client.

OO design pattern exercise

You were not required to use object oriented design patterns in the first three homework assignments (besides Iterator and possibly the interning of Strings, thanks to the creators of Java).

Now that you have learned several design patterns, consider their use in those assignments. Try answering questions such as: What classes would have made good Singletons? Where would you have liked to use a Factory? How could you have applied the Flyweight pattern? Discuss with your classmates.

As a reminder, the classes used in each homework assignment were:

Homework 1: Shopping

- `Item`, a single item that can be purchased
- `DiscountedItem`, a single item with a bulk discount for high-quantity purchases
- `Catalog`, a set of all items that are available in the store
- `Purchase`, a single item to be purchased in a given quantity
- `ShoppingCart`, the list of all purchases that the user currently wants to make

Homework 2: Scheduler

- `Weekday`, an enumeration (enum) representing weekdays from Monday-Friday
- `Time`, a class representing times of day such as 12:30 PM or 9:00 AM
- `Course`, a class representing individual courses
- `Schedule`, a class representing a student's current collection of courses
- `ScheduleConflictException`, an exception class to be thrown when course conflicts occur
- `CourseNameComparator`, a class for sorting a schedule by course name
- `CourseCreditComparator`, a class for sorting a schedule by number of credits for each course
- `CourseTimeComparator`, a class for sorting a schedule by days and times

Homework 3: Restaurant

- `Restaurant`, the overall restaurant and its associated data
- `Party`, a group of customers that can eat at the restaurant
- `Table`, a table at which parties of customers may sit to eat
- `Servers`, a manager for servers and their allocation to tables in the restaurant