Problem Set 1: Tweet Tweet

Beta due	Tuesday, February 23, 2016, 10:00 pm
Code reviews due	Thursday, February 25, 2016, 10:00 pm
Final due	Tuesday, March 1, 2016, 10:00 pm

The purpose of this problem set is to give you practice with test-first programming. Given a set of specifications, you will write unit tests that check for compliance with the specifications, and then implement code that meets the specifications.

Get the code

To get started,

1. Ask Didit to create a remote `psets/ps1` repository (https://didit.csail.mit.edu/psets/ps1) for you on Athena.
2. Pull the repo from Athena using Git:

```
git clone ssh://[username]@athena.dialup.mit.edu/mit/6.005/git/sp16/psets/ps1/[username].git ps1
```

If you need a refresher on how to create, clone, and import your repository, see Problem Set 0 (../ps0/#clone).

Overview

The theme of this problem set is to build a toolbox of methods that can extract information from a set of tweets downloaded from Twitter.

Since we are doing test-first programming, your workflow for each method should be (*in this order*).

1. Study the specification of the method carefully.
2. Write JUnit tests for the method according to the spec.
3. Implement the method according to the spec.
4. Revise your implementation and improve your test cases until your implementation passes all your tests.

On PS0, we graded only your method implementations. On this problem set, we will also grade the tests you write. In particular:

- **Your test cases should be chosen using the input/output-space partitioning approach**. This approach is explained in the reading about testing (../../classes/03-testing/).
- **Include a comment at the top of each test suite class describing your *testing strategy*** – how you partitioned the input/output space of each method, and then how you decided which test cases to choose for each partition.
- **Your test cases should be small and well-chosen.** Don’t use a large set of tweets from Twitter for each test. Instead, create your own artificial tweets, carefully chosen to test the partition you’re trying to test.
- **Your tests should find bugs.** We will grade your test cases in part by running them against buggy implementations and seeing if your tests catch the bugs. So consider ways an implementation might inadvertently fail to meet the spec, and choose tests that will expose those bugs.

- **Your tests must be legal clients of the spec.** We will also run your test cases against legal, variant implementations that still strictly satisfy the specs, and your test cases should not complain for these good implementations. That means that your test cases can't make extra assumptions that are only true for your own implementation.
- **Put each test case in its own JUnit method.** This will be far more useful than a single large test method, since it pinpoints where the problem areas lie in the implementation.
- Again, keep your tests small. Don't use unreasonable amounts of resources (such as `MAX_INT` size lists). We won't expect your test suite to catch bugs related to running out of resources; *every* program fails when it runs out of resources.

You should also keep in mind these facts from the readings and classes about specifications (part 1 (../../classes/06-specifications), part 2 (../../classes/07-designing-specs)):

- **Preconditions.** Some of the specs have preconditions, e.g. "this value must be positive" or "this list must be nonempty". When preconditions are violated, the behavior of the method is *completely unspecified*. It may return a reasonable value, return an unreasonable value, throw an unchecked exception, display a picture of a cat, crash your computer, etc., etc., etc. In the tests you write, do not use inputs that don't meet the method's preconditions. In the implementations you write, you may do whatever you like if a precondition is violated. Note that if the specification indicates a particular exception should be thrown for some class of invalid inputs, that is a *postcondition*, not a precondition, and you *do* need to implement and test that behavior.
- **Underdetermined postconditions.** Some of the specs have underdetermined postconditions, allowing a range of behavior. When you're implementing such a method, the exact behavior of your method within that range is up to you to decide. When you're writing a test case for the method, you must allow the implementation you're testing to have the full range of variation, because otherwise your test case is not a legal client of the spec as required above.

Finally, in order for your overall program to meet the specification of this problem set, you are required to keep some things unchanged:

- **Don't change these classes at all:** the classes `Tweet` , `Timespan` , and `TweetReader` should not be modified *at all*.
- **Don't change these class names:** the classes `Extract` , `Filter` , `SocialNetwork` , `ExtractTest` , `FilterTest` , and `SocialNetworkTest` must use those names and remain in the `twitter` package.
- **Don't change the method signatures and specifications:** The public methods provided for you to implement in `Extract` , `Filter` , and `SocialNetwork` must use the method signatures and the specifications that we provided.
- **Don't include illegal test cases:** The tests you implement in `ExtractTest` , `FilterTest` , and `SocialNetworkTest` must respect the specifications that we provided for the methods you are testing.

Aside from these requirements, however, you are free to add new public and private methods and new public or private classes if you wish. In particular, if you wish to write test cases that test a stronger spec than we provide, you should put those tests in a separate JUnit test class, so that we don't try to run them on staff implementations that only satisfy the weaker spec. We suggest naming those test classes `MyExtractTest` , `MyFilterTest` , `MySocialNetworkTest` , and we suggest putting them in the `twitter` package in the `test` folder alongside the other JUnit test classes.

Problem 1: Extracting data from tweets

In this problem, you will test and implement the methods in `Extract.java` .

You'll find `Extract.java` in the `src` folder, and a JUnit test class `ExtractTest.java` in the `test` folder. Separating implementation code from test code is a common practice in development projects. It makes the implementation code easier to understand, uncluttered by tests, and easier to package up for release.

a. Devise, document, and implement test cases for `getTimespan()` and `getMentionedUsers()` , and put them in `ExtractTest.java` .

b. Implement `getTimespan()` and `getMentionedUsers()` , and make sure your tests pass.

If you want to see your code work on a live sample of tweets, you can run `Main.java` . (`Main.java` will not be used in grading, and you are free to edit it as you wish.)

Hints:

- Note that we use the class `Instant` (http://docs.oracle.com/javase/8/docs/api/?java/time/Instant.html) to represent the date and time of tweets. You can check this article on Java 8 dates and times (http://java.dzone.com/articles/deeper-look-java-8-date-and) to learn how to use `Instant` .
- You may wonder what to do about lowercase and uppercase in the return value of `getMentionedUsers()` . This spec has an underdetermined postcondition, so read the spec carefully and think about what that means for your implementation and your test cases.
- `getTimespan()` *also* has an underdetermined postcondition in some circumstances, which gives the implementor (you) more freedom and the client (also you, when you're writing tests) less certainty about what it will return.
- Read the spec for the `Timespan` class carefully, because it may answer many of the questions you have about `getTimespan()` .

Commit to Git. Once you're happy with your solution to this problem, commit and push! Committing frequently – whenever you've fixed a bug or added a working and tested feature – is a good way to use version control, and will be a good habit to have for your team projects.

Problem 2: Filtering lists of tweets

In this problem, you will test and implement the methods in `Filter.java` .

a. Devise, document, and implement test cases for `writtenBy()` , `inTimespan()` , and `containing()` , and put them in `FilterTest.java` .

b. Implement `writtenBy()` , `inTimespan()` , and `containing()` , and make sure your tests pass.

Hints:

- For questions about lowercase/uppercase and how to interpret timespans, reread the hints in the previous question.
- For all problems on this problem set, you are free to rewrite or replace the provided example tests and their assertions.

Commit to Git. Once you're happy with your solution to this problem, commit and push!

Problem 3: Inferring a social network

In this problem, you will test and implement the methods in `SocialNetwork.java` . The `guessFollowsGraph()` method creates a social network over the people who are mentioned in a list of tweets. The social network is an approximation to who is following whom on Twitter, based only on the evidence found in the tweets. The `influencers()` method returns a list of people sorted by their influence (total number of followers).

a. Devise, document, and implement test cases for `guessFollowsGraph()` and `influencers()` , and put them in `SocialNetworkTest.java` . Be careful that your test cases for `guessFollowsGraph()` respect its underdetermined postcondition.

b. Implement `guessFollowsGraph()` and `influencers()` , and make sure your tests pass. For now, implement only the minimum required behavior for `guessFollowsGraph()` , which infers that Ernie follows Bert if Ernie @-mentions Bert.

If you want to see your code work on a live sample of tweets, run `Main.java` . It will print the top 10 most-followed people according to the social network you generated. You can search for them on Twitter to see if their actual number of followers has a similar order.

Problem 4: Get smarter

In this problem, you will implement one additional kind of evidence in `guessFollowsGraph()` . Note that we are taking a broad view of "influence" here, and even Twitter-following is not a ground truth for influence, only an approximation. It's possible to read Twitter without explicitly following anybody. It's also possible to be influenced by somebody through other media (email, chat, real life) while producing evidence of the influence on twitter.

Here are some ideas for evidence of following. Feel free to experiment with your own.

- **Common hashtags.** People who use the same hashtags in their tweets (e.g. `#mit`) may mutually influence each other. People who share a hashtag that isn't otherwise popular in the dataset, or people who share multiple hashtags, may be even stronger evidence.
- **Triadic closure (http://en.wikipedia.org/wiki/Triadic_closure).** In this context, triadic closure means that if a strong tie (mutual following relationship) exists between a pair A,B and a pair B,C, then some kind of tie probably exists between A and C – either A follows C, or C follows A, or both.
- **Awareness**. If A follows B and B follows C, and B retweets a tweet made by C, then A sees the retweet and is influenced by C.

Keep in mind that whatever additional evidence you implement, your `guessFollowsGraph()` must still obey the spec. To test your specific implementation, make sure you put test cases in your own `MySocialNetworkTest` class rather than the `SocialNetworkTest` class that we will run against staff implementations. Your work on this problem will be judged by the clarity of the code you wrote to implement it and the test cases you wrote to test it.

Submitting

Make sure you commit AND push your work to your repository on Athena. We will use the state of your repository on Athena as of 10:00pm on the deadline date. When you `git push` , the continuous build system attempts to compile your code and run a **subset** of the autograder tests. You can always review your build results at didit.csail.mit.edu (https://didit.csail.mit.edu).

Didit feedback is provided on a best-effort basis:

- There is no guarantee that Didit tests will run within any particular timeframe, or at all. If you push code close to the deadline, the large number of submissions will slow the turnaround time before your code is examined.
- If you commit and push right before the deadline, the Didit build does not have to complete in order for that commit to be graded.
- Passing some or all of the **public tests** on Didit is no guarantee that you will pass the full battery of autograding tests — but failing them is almost sure to mean lost points on the problem set.