Problem Set 3 | Problem Set 3 Beta | Contenu du cours 6.005.1x

Problem Set 3: The Librarians

Overview

The purpose of this problem set is to practice designing, testing, and implementing abstract data types. This problem set includes both immutable and mutable types. For one type (`SmallLibrary`), the representation of the type is specified, and you should implement its methods to match that rep. For other types, the representation is up to you to choose. The problem set also includes an example of two different implementations of the same Java interface. Finally, you'll be expected to implement equality appropriately for all the types.

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

1. Study the specifications of the type's operations carefully.
2. Write JUnit tests for the operations according to the spec.
3. Write the type's rep (its fields), document its rep invariant and abstraction function, and implement the rep invariant in a `checkRep()` method.
4. Implement the type's methods according to the spec.
5. Make an argument about why your rep is safe from rep exposure, and write it down in a comment.
6. Revise your implementation and improve your test cases until your implementation passes all your tests.

As in Problem Set 2, part of the point of this problem set is to learn how to write good tests for abstract data types, so the same expectations apply:

- Your test cases should be chosen using the input/output-space partitioning approach.
- Your test cases should be small and well-chosen.
- Your tests should find bugs. The Final grading test suite will include buggy implementations of the types, so your tests need to find those bugs.
- Your tests must be legal clients of the spec.

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 class names:** the classes `Book`, `BookCopy`, `Library`, `SmallLibrary`, `BigLibrary`, `BookTest`, `BookCopyTest`, `LibraryTest`, and `BigLibraryTest` must use those names and remain in the `library` package.
- **Don't change the method signatures and specifications:** The public methods provided for you to implement in `Book`, `BookCopy`, and `Library` must use the method signatures and the specifications that we provided.
- **Don't include illegal test cases:** The tests you implement in `BookTest`, `BookCopyTest`, and `LibraryTest` must respect the specifications that we provided for the methods you are testing.
- **Don't change the rep for `SmallLibrary`:** One type, `SmallLibrary`, has a required rep that you should not change. For the other types (`Book`, `BookCopy`, and `BigLibrary`) the rep is up to you.

Aside from these requirements, however, you are free to add new public and private methods and new public or private classes if you wish.

Problem 1: Book and BookCopy

The overall theme of this problem set is implementing a book catalog for a lending library. In this problem, you will test and implement the abstract data types `Book` and `BookCopy`. `Book` is an immutable type representing a published book (uniquely identified by its title, author list, and publication year). Since a library may have more than one copy of the same book, we also have a mutable type `BookCopy` that represents one copy of a book. The operations and specs for `Book` and `BookCopy` are given in their source files, and should not be changed.

You'll find `Book.java` and `BookCopy.java` in the `src` folder, and their corresponding JUnit test classes `BookTest.java` and `BookCopyTest.java` in the `test` folder. as we did in previous problem sets.

1. Devise, document, and implement test cases for the operations of `Book`, and put them in `BookTest.java`.
2. Choose a representation for `Book`, and write down the rep invariant and abstraction function in a comment. These types are basically warmups, so your rep invariant and abstraction function will be very simple. Implement the rep invariant by writing assertions in the `checkRep()` method.
3. Implement the operations of `Book` using your rep. Make sure to call `checkRep()` at appropriate points, i.e. at the end of every creator, producer, and mutator operation. Also make sure to implement `equals()` and `hashCode()` as appropriate for an immutable type.
4. Convince yourself that your type is safe from rep exposure, and write your argument down in a comment after the abstraction function.
5. Finally, run your tests, and revise until your `Book` implementation passes your tests.

Repeat these same steps for `BookCopy`, taking care to note that it is a mutable type where `Book` is immutable:

1. Devise, document, and implement test cases for the operations of `BookCopy`, and put them in `BookCopyTest.java`.
2. Choose a representation for `BookCopy`, and write down the rep invariant and abstraction function in a comment. Again, these will be very simple for this class. Implement the rep invariant by writing assertions in the `checkRep()` method.
3. Implement the operations of `BookCopy`, including calls to `checkRep()` at appropriate points. Also make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
4. Convince yourself that your type is safe from rep exposure, and write your argument down in a comment after the abstraction function.
5. Finally, run your tests, and revise until your `BookCopy` implementation passes your tests.

Problem 2: LibraryTest

The library catalog itself is represented by the `Library` type, which is a Java interface. In later problems on this problem set, you will implement two different implementations of this type:

- `SmallLibrary`, a simple brute-force representation that works well enough for small libraries, like a few hundred books owned by a single person.
- `BigLibrary`, a representation that performs better on bigger libraries.

In this problem, we'll just write tests for the `Library` interface itself, without caring which kind of library the tests are run on. The `LibraryTest` JUnit class has been designed so that it automatically runs twice, once using `SmallLibrary` and again using `BigLibrary`.

Devise, document, and implement test cases for the operations of `Library`, and put them in `LibraryTest.java`.

Problem 3: SmallLibrary

Now that we have some test cases, we're ready to implement `SmallLibrary`. The representation for `SmallLibrary` is already given in `SmallLibrary.java`, including its rep invariant and abstraction function.

1. Implement the rep invariant of `SmallLibrary` by writing assertions in the `checkRep()` method.
2. Implement the operations of `SmallLibrary`, including calls to `checkRep()` at appropriate points. Also make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
3. Write down an argument that your type is safe from rep exposure.
4. Finally, run your `LibraryTest` tests, and revise until your `SmallLibrary` implementation passes your `LibraryTest` tests. Note that you won't get a full green light yet from JUnit, because it is also running `LibraryTest` against `BigLibrary`, which you haven't implemented yet.

Notes:

- **Don't change the rep**. You may find yourself tempted to add new fields to the `SmallLibrary` rep, but don't. The rep has been chosen to be as simple as possible, to make some Library operations very easy to implement (like `isAvailable()`) even though others are harder (like `find()`). Bear with that, and make it work. Starting with a simple brute-force rep allows you to debug your tests and your understanding of the spec. You'll invent a better rep in the next problem.
- **Ignore performance**. SmallLibrary is designed for very small book collections, so its operations can be very slow or use extra space. It's brute force. Just don't worry about performance, and focus on simplicity and correctness. Save your performance-optimization ideas for the next problem.
- **Simplest possible functionality.** In particular, the spec for `find()` is underdetermined. Just do the minimum required for `SmallLibrary`. Save your ideas for making it better for the last problem.

Problem 4: BigLibrary

Now that we've implemented a simple version of the library, let's make it better by implementing `BigLibrary`. You will choose your own representation for `BigLibrary`. You can borrow ideas and code from your `SmallLibrary`, but your goal should be to make the operations of `BigLibrary` work efficiently even when there are millions of books in the library.

1. Choose a rep for `BigLibrary` and write down its rep invariant and abstraction function. You may want to use `Map` or `SortedSet` or other data structures in your rep. You may want to store information redundantly to save time or space when you're implementing the operations.
2. Implement the rep invariant of `BigLibrary` by writing assertions in the `checkRep()` method.
3. Implement the operations of `BigLibrary`, including calls to `checkRep()` at appropriate points. As usual, make sure to implement `equals()` and `hashCode()` as appropriate for a mutable type.
4. Write down an argument that your type is safe from rep exposure.
5. Finally, run your `LibraryTest` tests, and revise until your `BigLibrary` implementation passes your `LibraryTest` tests.

Notes:

- **Aim for constant-time or logarithmic-time performance**. Since BigLibrary is designed for big book collections, try to make its operations run in O(1) or O(log N) time for a library with N books. Don't count the cost of `checkRep()`.
- **Simplest possible functionality.** Again, start out by implementing only minimum required behavior for `find()`. Save your ideas for making it better for the final problem on this problem set.

Problem 5: Improving find()

Now improve your `BigLibrary` so that `find()` behaves more like a user would expect a library catalog interface to behave. Examples of reasonable behavior that is allowed by the find() spec include:

- Matching words in the `keywords` argument to words in title or author names.
- Ranking the resulting list of books so that books that match more keywords appear earlier in the list.
- Ranking books that match multiple contiguous keywords higher in the list.
- Ranking older books or checked-out books lower in the list.
- Supporting quotation marks in the `keywords` argument, so that (for example) `"\"David Foster Wallace\" \"Infinite Jest\""` finds books whose title or author contains David Foster Wallace or Infinite Jest as contiguous words.

When you add this new behavior to `BigLibrary.find()`, you should strengthen its spec accordingly, so that clients of `BigLibrary` can expect the behavior. Don't change Library's spec, however, and leave your `LibraryTest` tests and `SmallLibrary` implementation unchanged. Instead, to test your new stronger behavior, you should put the new tests in `BigLibraryTest.java`.

1. Write down the spec for your new behavior for `BigLibrary.find()` as a Javadoc comment above the method, including preconditions and postconditions as appropriate.
2. Devise, document, and implement test cases for your stronger `find()` spec in `BigLibraryTest`.
3. Change the rep of `BigLibrary` as needed to handle your new behavior, update the rep invariant and abstraction function comments, and update the `checkRep()` method.
4. Implement your new `BigLibrary.find()`.

5. Finally, run all your tests, including the original `LibraryTest` tests and your new `BigLibraryTest` tests, and revise until your `BigLibrary` implementation passes your tests.
