70094 16

Creation

Submitters

zz3823	Zhang Zhang
sf23	Shihan Fu

```
Final Tests: Summary for zz3823 of v5
-------------------------------------

  Public Tests:
    Compiles:                   1 / 1
    Tests Pass:                 1 / 1
    Coverage and Style Checks:  1 / 1

Git Repo: git@gitlab.doc.ic.ac.uk:lab2324_spring/SE_Design_Ex6_zz3823.git
Commit ID: 4df6d
```

Formatting doesn’t actually pass Google’s guidelines? Should use 2 spaces indentation.

17/20

Good implementation!

Since you’re controlling what `searchFor` returns, you only need to checks that the queries are correct (using expectations). You also need to check (once) that the correct results is propagated from `searchFor`.
`catalogue` must be a mandatory parameter for the builder: the query without an underlying library doesn’t make sense and is useless.
When implementing singleton, make the constructor private.

See comments below.

```java
package ic.doc;

import ic.doc.catalogues.LibraryCatalogue;
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.junit.Rule;
import org.junit.Test;

import java.util.Arrays;
import java.util.List;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;

public class BookSearchQueryTest {

  @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

  private final LibraryCatalogue catalogue = context.mock(LibraryCatalogue.class);

  @Test
  public void searchesForBooksInLibraryCatalogueByAuthorSurname() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(
                    Arrays.asList(
                        new Book("A Tale of Two Cities", "Charles Dickens", 1859),
                        new Book("Oliver Twist", "Charles Dickens", 1838))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder().withLastName("dickens").libraryCatalogue(catalogue).build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(2));
    assertTrue(books.get(0).matchesAuthor("dickens"));
  }
```

Better to check that `books` coincides with the expected list of books.

```java
  @Test
  public void searchesForBooksInLibraryCatalogueByAuthorFirstname() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(
                    Arrays.asList(
                        new Book("Pride and Prejudice", "Jane Austen", 1813),
                        new Book("Sense and Sensibility", "Jane Austen", 1811))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder().withFirstName("Jane").libraryCatalogue(catalogue).build();
    List<Book> books = query.execute();

```

```java
    assertThat(books.size(), is(2));
    assertTrue(books.get(0).matchesAuthor("Austen"));
  }

  @Test
  public void searchesForBooksInLibraryCatalogueByTitle() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(
                        List.of(new Book("A Tale of Two Cities", "Charles Dickens", 1859))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder().withTitle("Two Cities").libraryCatalogue(catalogue).build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(1));
    assertTrue(books.get(0).matchesAuthor("dickens"));
  }

  @Test
  public void searchesForBooksInLibraryCatalogueBeforeGivenPublicationYear() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(
                    Arrays.asList(
                        new Book("Hamlet", "William Shakespeare", 1603),
                        new Book("The Tempest", "William Shakespeare", 1611))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder().publishedBefore(1700).libraryCatalogue(catalogue).build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(2));
    assertTrue(books.get(0).matchesAuthor("Shakespeare"));
  }

  @Test
  public void searchesForBooksInLibraryCatalogueAfterGivenPublicationYear() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(List.of(new Book("Lord of the Flies", "William Golding", 1954))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder().publishedAfter(1950).libraryCatalogue(catalogue).build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(1));
    assertTrue(books.get(0).matchesAuthor("Golding"));
  }

  @Test
  public void searchesForBooksInLibraryCatalogueWithCombinationOfParameters() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(returnValue(List.of(new Book("Oliver Twist", "Charles Dickens", 1838))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder()
            .withLastName("dickens")
            .publishedBefore(1840)
            .libraryCatalogue(catalogue)
            .build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(1));
    assertTrue(books.get(0).matchesAuthor("charles dickens"));
  }

  @Test
  public void searchesForBooksInLibraryCatalogueWithCombinationOfTitleAndOtherParameters() {
    context.checking(
        new Expectations() {
          {
            oneOf(catalogue).searchFor(with(any(String.class)));
            will(
                returnValue(
                    Arrays.asList(
                        new Book("Great Expectations", "Charles Dickens", 1861),
                        new Book("The Mystery of Edwin Drood", "Charles Dickens", 1870),
                        new Book("The Old Curiosity Shop", "Charles Dickens", 1841))));
          }
        });

    BookSearchQuery query =
        new BookSearchQueryBuilder()
            .withTitle("of")
            .publishedAfter(1800)
            .publishedBefore(2000)
            .libraryCatalogue(catalogue)
            .build();
    List<Book> books = query.execute();

    assertThat(books.size(), is(3));
    assertTrue(books.get(0).matchesAuthor("charles dickens"));
  }
}
```

```java
package ic.doc.catalogues;

import ic.doc.Book;

import java.util.Collection;
import java.util.List;

public interface LibraryCatalogue {
  List<Book> searchFor(String query);

  Collection<Book> allTheBooks();
}
```

`allTheBooks()` method was initially private, so shouldn't be extracted into the interface.

```java
package ic.doc.catalogues;

import static ic.doc.catalogues.QueryParser.firstNameFrom;
import static ic.doc.catalogues.QueryParser.lastNameFrom;
import static ic.doc.catalogues.QueryParser.publishedAfterFrom;
import static ic.doc.catalogues.QueryParser.publishedBeforeFrom;
import static ic.doc.catalogues.QueryParser.titleFrom;

import ic.doc.Book;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

public class BritishLibraryCatalogue implements LibraryCatalogue {

  private static BritishLibraryCatalogue instance;

  // imagine that each new instance of this object uses more than 500MB of RAM

  private final Collection<Book> catalogue = allTheBooks();

  // Private constructor to prevent instantiation.
  private BritishLibraryCatalogue() {
    System.out.println("Memory Usage: 500MB...");
  }

  // Static method to get the instance of the class.
  public static synchronized BritishLibraryCatalogue getInstance() {
    if (instance == null) {
      instance = new BritishLibraryCatalogue();
    }
    return instance;
  }

  @Override
  public List<Book> searchFor(String query) {
    return catalogue.stream()
        .filter(book -> book.matchesAuthor(lastNameFrom(query)))
        .filter(book -> book.matchesAuthor(firstNameFrom(query)))
        .filter(book -> book.matchesTitle(titleFrom(query)))
        .filter(book -> book.publishedSince(publishedAfterFrom(query)))
        .filter(book -> book.publishedBefore(publishedBeforeFrom(query)))
        .collect(Collectors.toList());
  }

  @Override
  public Collection<Book> allTheBooks() {

    return Arrays.asList(
        new Book("A Tale of Two Cities", "Charles Dickens", 1859),
        new Book("Pride and Prejudice", "Jane Austen", 1813),
        new Book("Pride and Prejudice", "Jane Austen", 1813),
        new Book("The Picture of Dorian Gray", "Oscar Wilde", 1890),
        new Book("Oliver Twist", "Charles Dickens", 1838),
        new Book("Frankenstein", "Mary Shelley", 1817),
        new Book("Brave New World", "Aldous Huxley", 1932),
        new Book("Lord of the Flies", "William Golding", 1954),
        new Book("Hamlet", "William Shakespeare", 1603),
        new Book("The Life and Opinions of Tristram Shandy, Gentleman", "Laurence Sterne", 1759));

    // and so on... Imagine that this list is very large and therefore uses a lot of memory.

  }
```

It’s still possible to construct your class
without using `getInstance`.
Also need to make the constructor private.

```java
}
```

```java
package ic.doc;

import ic.doc.catalogues.BritishLibraryCatalogue;
import ic.doc.catalogues.LibraryCatalogue;

public class BookSearchQueryBuilder {

  private String name1;
  private String name2;
  private String title;
  private Integer date1;
  private Integer date2;
  private LibraryCatalogue catalogue = ~~BritishLibraryCatalogue.getInstance()~~;

  public BookSearchQueryBuilder withFirstName(String name1) {
    this.name1 = name1;
    return this;
  }

  public BookSearchQueryBuilder withLastName(String name2) {
    this.name2 = name2;
    return this;
  }

  public BookSearchQueryBuilder withTitle(String title) {
    this.title = title;
    return this;
  }

  public BookSearchQueryBuilder publishedAfter(Integer date1) {
    this.date1 = date1;
    return this;
  }

  public BookSearchQueryBuilder publishedBefore(Integer date2) {
    this.date2 = date2;
    return this;
  }

  public BookSearchQueryBuilder libraryCatalogue(LibraryCatalogue catalogue) {
    this.catalogue = catalogue;
    return this;
  }

  public BookSearchQuery build() {
    return new BookSearchQuery(name1, name2, title, date1, date2, catalogue);
  }
}
```

Unnecessary coupling and a dangerous default.

Make `catalogue` a mandatory parameter in the constructor (and maybe add a factory method as well).

```java
package ic.doc;

import ic.doc.catalogues.LibraryCatalogue;
import java.util.List;

public class BookSearchQuery {

  private final String name1;
  private final String name2;
  private final String title;
  private final Integer date1;
  private final Integer date2;
  private final LibraryCatalogue catalogue;

  public BookSearchQuery(
      String p1, String p2, String p3, Integer p4, Integer p5, LibraryCatalogue catalogue) {
    this.name1 = p1;
    this.name2 = p2;
    this.title = p3;
    this.date1 = p4;
    this.date2 = p5;
    this.catalogue = catalogue;
  }

  public List<Book> execute() {
    StringBuilder query = new StringBuilder();
    if (name1 != null) {
      query.append("FIRSTNAME='").append(name1).append("' ");
    }
    if (name2 != null) {
      query.append("LASTNAME='").append(name2).append("' ");
    }
    if (title != null) {
      query.append("TITLECONTAINS(").append(title).append(") ");
    }
    if (date1 != null) {
      query.append("PUBLISHEDAFTER(").append(date1).append(") ");
    }
    if (date2 != null) {
      query.append("PUBLISHEDBEFORE(").append(date2).append(") ");
    }
    return catalogue.searchFor(query.toString());
  }
}
```

```
-------- Test Output --------
Running LabTS build... (Wed 21 Feb 23:08:07 UTC 2024)

Submission summary...
You made 6 commits
  - 6908581 feat: Introduce builder to improve query construction [3 files changed, 60 insertions, 8 deletions]
  - 806bda8 feat: Introduce Singleton to ensure only one instance of BritishLibraryCatalogue is created [3 files changed, 17 insertions, 7 deletions]
  - 2b8a182 feat: Introduce Dependence Inversion to reduce coupling between BookSearchQuery and BritishLibraryCatalogue [5 files changed, 29 insertions, 6 deletions]
  - 1d23dfa feat: Introduce mock setup to test BookSearchQuery in isolation [2 files changed, 113 insertions, 24 deletions]
  - 5c1adcd style: minor changes [2 files changed, 3 insertions, 4 deletions]
  - 4df6db6 refactor: Put allTheBooks into the interface LibraryCatalogue [2 files changed, 6 insertions, 1 deletion]

Preparing...

BUILD SUCCESSFUL in 622ms

Compiling...
BUILD SUCCESSFUL in 4s

Running tests...

ic.doc.BookTest > supportsPublicationDataQuery PASSED

ic.doc.BookTest > supportsCaseInsensitiveTitleQuery PASSED

ic.doc.BookTest > convertsToFormattedStringOfTitleAndAuthor PASSED

ic.doc.BookTest > supportsCaseInsensitiveAuthorQuery PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueWithCombinationOfParameters PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueByAuthorFirstname PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueByTitle PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueAfterGivenPublicationYear PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueWithCombinationOfTitleAndOtherParameters PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueBeforeGivenPublicationYear PASSED

ic.doc.BookSearchQueryTest > searchesForBooksInLibraryCatalogueByAuthorSurname PASSED

BUILD SUCCESSFUL in 1s

Checking test coverage and code style...
BUILD SUCCESSFUL in 3s
Finished auto test. (Wed 21 Feb 23:08:25 UTC 2024)

-------- Test Errors --------

```