



CHAPTER 13

Test Driven Development

13

Implementing a user story on student enrollment

Suppose that we are working on a student management system and that currently we are working on a user story on student enrollments. A student enrolls in a certain course by paying the course fee specified in the course. The details of the enrollment must be recorded (e.g., student id, course code, date, handled by whom, etc.). The details of the payment must be recorded (e.g., amount, cash or credit card, etc.). The student and the course must have been registered in the system. He can enroll only if there is at least one free seat in the course. The number of seats in a course is specified in the course. A seat is taken when someone enrolls in it. A seat is reserved if someone makes a reservation. The reservation is usually maintained for 24 hours but can be changed by the users. A course may consist of a number of "modules". For example, a Java programming course may consist of a "Java Programming Language" module and a "JDBC" module. Each module is just like a course. However, some modules can not be enrolled in isolation. That is, you must enroll in the whole course in order to enroll in that module. A student can enroll in the whole course only if all the modules have at least one free seat.

Suppose that in the system there are some useful classes:

```
class Student {
    ...
}
class Course {
    ...
    Course[] getModules() { ... }
}
class Reservation {
    Date reserveDate;
    int daysReserved;
    ...
}
class StudentSet {
    ...
}
class CourseSet {
    ...
}
class ReservationSet {
    Reservation[] getReservationsFor(String courseCode) { ... }
    ...
}
```

We figure that to implement this story, we should create an Enrollment class and an

EnrollmentSet class. The various checking can be done before an Enrollment is added to the database:

```
class Enrollment {
    ...
}
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertValid(enrollment);
        addToDB(enrollment);
    }
    void assertValid(Enrollment enrollment) {
        ...
    }
    void addToDB(Enrollment enrollment) {
        ...
    }
}
```

So, we will continue to work on them:

```
class Enrollment {
    String studentId;
    String courseCode;
    Date enrolDate;
    Payment payment;
}
class Payment {
    int amount;
    Payment(int amount) {
        ...
    }
}
class CashPayment extends Payment {
    CashPayment(int amount) {
        ...
    }
}
class CreditCardPayment extends Payment {
    String cardNo;
    String nameOnCard;
    Date expiryDate;
    CreditCardPayment(String cardNo, String nameOnCard, Date expiryDate, int
amount) {
        ...
    }
}
class EnrollmentSet {
    StudentSet studentSet;
    CourseSet courseSet;
    ReservationSet reservationSet;

    EnrollmentSet(StudentSet studentSet, CourseSet courseSet, ReservationSet
reservationSet) {
```

```

        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    void add(Enrollment enrollment) {
        assertValid(enrollment);
        addToDB(enrollment);
    }
    void assertValid(Enrollment enrollment) {
        assertStudentExists(enrollment.getStudentId());
        assertCourseExists(enrollment.getCourseCode());
        assertPaymentCorrect(enrollment);
        assertHasFreeSeat(enrollment);
        assertCanEnrollInIsolation(enrollment);
    }
}

```

It is not finished yet. We still have to write the `assertStudentExists`, `assertCourseExists`, `assertPaymentCorrect`, `assertHasFreeSeat`, `assertCanEnrollInIsolation` and `addToDB` methods. Let's implement them one by one. Implement `assertStudentExists` first:

```

class EnrollmentSet {
    ...
    void assertStudentExists(String studentId) {
        studentSet.assertStudentExists(studentId);
    }
}

```

This is easy. Let's assume that the `StudentSet` class already has such an `assertStudentExists` method. So this method is done. Similarly, implement `assertCourseExists`:

```

class EnrollmentSet {
    ...
    void assertCourseExists(String courseCode) {
        courseSet.assertCourseExists(courseCode);
    }
}

```

Next, implement `assertPaymentCorrect`:

```

class EnrollmentSet {
    ...
    void assertPaymentCorrect(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (enrollment.getPayment().getAmount() != course.getFee()) {
            throw new IncorrectPaymentException();
        }
    }
}

```

This is not hard. Next, implement `assertHasFreeSeat`:

```

class EnrollmentSet {
    ...
    void assertHasFreeSeat(Enrollment enrollment) {

```

```

        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (course.getNoSeats() <= getNoEnrollments(course)+getNoActiveReservations
(course)) {
            throw new CourseFullException();
        }
    }
}

```

This is getting difficult: This method requires us to write two other methods: `getNoEnrollments` and `getNoActiveReservations`. So we have to write them one by one:

```

class EnrollmentSet {
    Connection conn;
    ...
    EnrollmentSet(
        Connection conn,
        StudentSet studentSet,
        CourseSet courseSet,
        ReservationSet reservationSet) {
        this.conn = conn;
        this.studentSet = studentSet;
        this.courseSet = courseSet;
        this.reservationSet = reservationSet;
    }
    int getNoEnrollments(Course course) {
        PreparedStatement st =
            conn.prepareStatement("select count(*) from enrollments where
courseCode=?");
        try {
            st.setString(1, course.getCode());
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        } finally {
            st.close();
        }
    }
}

```

Then:

```

class EnrollmentSet {
    ...
    int getNoActiveReservations(Course course) {
        return reservationSet.getNoActiveReservations(course.getCode());
    }
}

```

Suppose that there is no `getNoActiveReservations` method in the `ReservationSet` class. It means we need to add it:

```

class ReservationSet {
    Reservation[] getReservationsFor(String courseCode) { ... }
    int getNoActiveReservations(String courseCode) {
        int activeCount = 0;
    }
}

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

        Reservation reservations[] = getReservationsFor(courseCode);
        for (int i = 0; i < reservations.length; i++) {
            if (reservations[i].isActive()) {
                activeCount++;
            }
        }
        return activeCount;
    }
}

```

In turn we have to add an isActive method to the Reservation class:

```

class Reservation {
    Date reserveDate;
    int daysReserved;
    boolean isActive() {
        Date today = new Date();
        return getLastReservedDate().before(today);
    }
    Date getLastReservedDate() {
        GregorianCalendar lastReservedDate = new GregorianCalendar();
        lastReservedDate.setTime(reserveDate);
        lastReservedDate.add(GregorianCalendar.DATE, daysReserved);
        return lastReservedDate.getTime();
    }
}

```

Next, implement assertCanEnrollInIsolation:

```

class EnrollmentSet {
    ...
    void assertCanEnrollInIsolation(Enrollment enrollment) {
        Course course = courseSet.getByCode(enrollment.getCourseCode());
        if (course.isModule() && !course.canEnrollInIsolation()) {
            throw new CannotEnrollInIsolation();
        }
    }
}

```

Let's assume the Course class already has the isModule and canEnrollInIsolation methods:

```

class Course {
    boolean isModule() { ... }
    boolean canEnrollInIsolation() { ... }
}

```

Finally, implement the addToDB method. Let's assume that the enrollments table is like this:

```

CREATE TABLE enrollments (
    courseCode VARCHAR(50),
    studentId VARCHAR(50),
    enrolDate DATE,
    amount INT,
    paymentType VARCHAR(20),
    cardNo VARCHAR(20),

```

```

        expiryDate DATE,
        nameOnCard VARCHAR(50)
    );

```

The addToDB method is:

```

class EnrollmentSet {
    private void addToDB(Enrollment enrollment) {
        PreparedStatement st =
            conn.prepareStatement(
                "insert into enrollments values (?, ?, ?, ?, ?, ?, ?, ?)");
        try {
            st.setString(1, enrollment.getCourseCode());
            st.setString(2, enrollment.getStudentId());
            st.setDate(3, new Date(enrollment.getEnrolDate().getTime()));
            Payment payment = enrollment.getPayment();
            st.setInt(4, payment.getAmount());
            if (payment instanceof CashPayment) {
                st.setString(5, "Cash");
                st.setNull(6, Types.VARCHAR);
                st.setNull(7, Types.DATE);
                st.setNull(8, Types.VARCHAR);
            } else if (payment instanceof CreditCardPayment) {
                CreditCardPayment creditCardPayment =
                    (CreditCardPayment) payment;
                st.setString(5, "CreditCard");
                st.setString(6, creditCardPayment.getCardNo());
                st.setDate(
                    7,
                    new Date(creditCardPayment.getExpiryDate().getTime()));
                st.setString(8, creditCardPayment.getNameOnCard());
            }
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

We have finished writing all the code. We have written about 150 lines of code without running any of it. At this moment, the biggest question on our mind is: Is this code correct? In fact, when we were writing this code, this worry had been accumulating. The more complicated the code was (e.g., addToDB, getNoEnrollments, isActive), the more worried we were. The more code we wrote, the more worried we were. For example, when we found that we needed to write two other methods in order to implement one method, we got more worried. Our accumulated worry can only be addressed by test running the code. How to test the code?

How to test the code just written

If we had a GUI making use of this code, we would be much tempted to just test run the

system, input some data manually then inspect the database to see if it is working. However, this is no good because the test is not automated and will not be run frequently. For example, if someone changes the code and introduces a bug by accident, if he doesn't test run the application again or just test it in a very simple way, he may not notice anything wrong. Therefore, a much better way is to have automated unit tests. How to unit test say the add method of EnrollmentSet? We may test adding a valid enrollment first:

```
class EnrollmentSetTest extends TestCase {
    testAddValidEnrollment() {
        Connection conn = ...;
        try {
            EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
            enrollmentSet.deleteAll();
            Enrollment enrollment =
                new Enrollment(
                    "s001",
                    "c001",
                    new Date(),
                    new CashPayment(100));
            enrollmentSet.add(enrollment);
            assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
        } finally {
            conn.close();
        }
    }
}
```

This test is not complete yet. Several things are missing: the database connection must be setup, we must add the student "s001" to the database, we must add the course "c001" to the database. Let's create the database connection first. Let's assume we are using a postgresSQL test database named "testdb" on our own computer:

```
class EnrollmentSetTest extends TestCase {
    testAddValidEnrollment() {
        Class.forName("org.postgresql.Driver");
        Connection conn =
            DriverManager.getConnection(
                "jdbc:postgresql://localhost/testdb",
                "username",
                "password");
        try {
            EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
            enrollmentSet.deleteAll();
            Enrollment enrollment =
                new Enrollment(
                    "s001",
                    "c001",
                    new Date(),
                    new CashPayment(100));
            enrollmentSet.add(enrollment);
            assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
        } finally {
            conn.close();
        }
    }
}
```

```
}

```

Next, add the student "s001" to the database. To do that, we need to create such a Student object first. This is not as easy as it seems because the Student class has a constructor that takes a lot of parameters:

```
class Student {
    String studentId;
    String idCardNo;
    String name;
    boolean isMale;
    Date dateOfBirth;
    ContactInfo address;
    Employment employment;

    Student(
        String studentId,
        String idCardNo,
        String name,
        boolean isMale,
        Date dateOfBirth,
        ContactInfo address,
        Employment employment) {
        ...
    }
}

class ContactInfo {
    String email;
    String telNo;
    String faxNo;
    String postalAddress;

    public ContactInfo(String email, String telNo, String faxNo, String
postalAddress) {
        ...
    }
}

public class Employment {
    String companyName;
    String jobTitle;
    int yearsOfService;
    ContactInfo contactInfo;
    public Employment(
        String companyName,
        String jobTitle,
        int yearsOfService,
        ContactInfo contactInfo) {
        ...
    }
}

```

Therefore, the create such a student, the code will be like:

```
class EnrollmentSetTest extends TestCase {
    testAddValidEnrollment() {

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

Class.forName("org.postgresql.Driver");
Connection conn =
    DriverManager.getConnection(
        "jdbc:postgresql://localhost/testdb",
        "username",
        "password");
try {
    StudentSet studentSet = new StudentSet(conn);
    studentSet.deleteAll();
    Student student =
        new Student(
            "s001",
            "9/29741/8",
            "Paul Chan",
            true,
            new Date(),
            new ContactInfo(
                "paul@yahoo.com",
                "123456",
                "123457",
                "postal address"),
            new Employment(
                "ABC Ltd.",
                "Manager",
                2,
                new ContactInfo(
                    "info@abc.com",
                    "111000",
                    "111001",
                    "ABC postal address")));
    studentSet.add(student);
    EnrollmentSet enrollmentSet = new EnrollmentSet(conn, ...);
    enrollmentSet.deleteAll();
    Enrollment enrollment =
        new Enrollment(
            "s001",
            "c001",
            new Date(),
            new CashPayment(100));
    enrollmentSet.add(enrollment);
    assertEquals(enrollmentSet.get("s001", "c001"), enrollment);
} finally {
    conn.close();
}
}
}

```

This is terrible. All we need is just his student id and his presence. We don't need all the other information such his name, id card number, date of birth, contact information, employment information and etc. Can we set them to null?

```

Student student = new Student("s001", null, null, true, null, null, null);
studentSet.add(student);

```

No. Because when the student is added to the database, various validity checking will be performed and the data in say the contact information will also be saved to the database. If

you set the reference to null, the program will crash. So, we have to bite the bullet to input all the data.

However, the horror doesn't stop here. The test is still incomplete. We still need to add the course "c001" to the database. We will face a similar difficulty: All we need is the course code, its fee, number of seats. We don't need other information such as course name, course content outline, instructor, schedule, and lots of other information.

This is getting out of hand. We must find a way to solve these problems.

Solve these problems by writing tests first

Let's throw away all this code and implement the story in a totally different way. First, let's write the unit test (yes, we don't write any other code yet!). The test is simple: we need to have an enrollment and an enrollment set, then add the enrollment to the enrollment set:

```
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.add(enrollment);
    }
}
```

Note that at the moment we have no Enrollment class nor EnrollmentSet class yet. In addition, the constructor for the enrollment object takes two parameters only: the student id and course code. Why these parameters? Simply because they are the parameters that come to me at the moment and I don't bother to think of other parameters. Similarly, I can't think of any parameters for the constructor for the enrollment set, so we will just let it take no parameter.

But how to test if the enrollment is really added to the database? Should we inspect the database:

```
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.add(enrollment);
        Class.forName("org.postgresql.Driver");
        Connection conn =
            DriverManager.getConnection(
                "jdbc:postgresql://localhost/testdb",
                "username",
                "password");
        ...
    }
}
```

This is too much code and it is too troublesome to check the database. We must come up with some way to replace the database. For example, all the EnrollmentSet needs is to write the Enrollment to somewhere. That somewhere is not necessarily a database. Let's call this EnrollmentStorage. We will create an EnrollmentStorage in the test, let the EnrollmentSet use it and finally check if it has really added the Enrollment to that EnrollmentStorage:

```
interface EnrollmentStorage {
    void add(Enrollment enrollment);
}
class EnrollmentStorageForTest implements EnrollmentStorage {
    Enrollment enrollmentStored;
    void add(Enrollment enrollment) {
        enrollmentStored = enrollment;
    }
};

class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        EnrollmentStorageForTest storage = new EnrollmentStorageForTest();
        enrollmentSet.setStorage(storage);
        enrollmentSet.add(enrollment);
        assertTrue(storage.enrollmentStored==enrollment);
    }
}
```

Of course, at the moment the unit test will not even compile. Let's create the Enrollment class and EnrollmentSet class. However, we will just write enough code that is necessary to make the unit test compile, so that we can run the unit test ASAP:

```
class Enrollment {
    Enrollment(String studentId, String courseCode) {
    }
}
class EnrollmentSet {
    void setStorage(EnrollmentStorage storage) {
    }
    void add(Enrollment enrollment) {
    }
}
```

Yes, there is no code in the methods! As we said, we would like to run the unit test ASAP. Now, run the unit test. It fails and we see a red bar in JUnit (as expected). Why run it if we know it is going to fail? We will explain that later. For the moment, simply remember to run a test and see it fail before writing the implementation code. Now, we can go ahead to write the code required to make the test pass:

```
class EnrollmentSet {
    EnrollmentStorage storage;

    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
}
```

```

    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
}

```

This is just three lines of code. We can do it in a minute and now we can run the test again. Now it passes (green bar). We have made a progress. Next, we should write another unit test. For example, test if it can check the student is registered. However, before that, we can simplify the first unit test using an anonymous class to get rid of the `EnrollmentStorageForTest` whose name is not telling anything and therefore is bothering me:

```

class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        final Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
                assertTrue(enrollmentToStore == enrollment);
            }
        });
        enrollmentSet.add(enrollment);
    }
}

```

As we have changed the code, run the unit test again to see if the bar is still green. Yes, it is. So, we are safe to go ahead.

Testing if the student is registered

Now, write the test to see if it really checks the student is registered. The test will try to enroll a student who is really registered. But how to tell the `EnrollmentSet` that this student is registered? Adding to `StudentSet` is a lot of work as we have seen. Just like replacing `EnrollmentStorage` for the database, we can let it use a `StudentRegistryChecker` instead of a `StudentSet`:

```

interface StudentRegistryChecker {
    boolean isRegistered(String studentId);
}
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        ...
    }
    void testStudentRegistered() {
        final Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return studentId.equals("s001");
            }
        });
    }
}

```

```

    });
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.add(enrollment);
}
}

```

However, it is very similar to `testAddEnrollment`. In particular, the code to test if the enrollment is added to the storage. Usually it makes little sense to test the same behavior twice. `testStudentRegistered` should just test the checking behavior before the enrollment is added to the storage. This suggests that `EnrollmentSet` should have an `assertValid` method so that we can call it instead of the `add` method:

```

class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        ...
    }
    void testStudentRegistered() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return studentId.equals("s001");
            }
        });
        enrollmentSet.assertValid(enrollment);
    }
}

```

Now the test is much cleaner. Of course, we need to test if `add` really calls `assertValid` before sending the enrollment to the storage. But for the moment, let's put it onto our todo list:

TODO
Test if <code>add</code> really calls <code>assertValid</code> .

We will make the current test (`testStudentRegistered`) pass first. So, create the necessary methods so that the test will compile:

```

class EnrollmentSet {
    EnrollmentStorage storage;

    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
    }
    void assertValid(Enrollment enrollment) {

```

```
    }
}
```

Now run the two tests. Oops! We get a green bar! This is because at the moment it will treat any enrollment as valid. So our really valid enrollment is also considered valid without any additional effort. Of course, we know that this `assertValid` should really do something like using the `StudentRegistryChecker` to lookup the student. However, before we do it, we must have a failing test first. So, let's try to check an unregistered student. We expect that it should throw a `StudentNotFoundException`:

```
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        ...
    }
    void testStudentRegistered() {
        ...
    }
    void testStudentUnregistered() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return false;
            }
        });
        try {
            enrollmentSet.assertValid(enrollment);
            fail();
        } catch (StudentNotFoundException e) {
        }
    }
}
```

It won't compile because `StudentNotFoundException` is undefined yet. Create an empty class so that the test will compile:

```
class StudentNotFoundException extends RuntimeException {
}
```

Now run all the tests. It fails. This is expected. Let's write the implementation code to make it pass:

```
class EnrollmentSet {
    EnrollmentStorage storage;
    StudentRegistryChecker studentRegistryChecker;

    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
    void add(Enrollment enrollment) {
        storage.add(enrollment);
    }
    void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
        this.studentRegistryChecker = registryChecker;
    }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
    }  
    void assertValid(Enrollment enrollment) {  
        if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {  
            throw new StudentNotFoundException();  
        }  
    }  
}
```

However, we need to have a `getStudentId` method in `Enrollment`. Usually we need to write a failing test for this method before we write the method, but it is such a simple getter method (see the code below) that we believe that there is no need to test it. So we go ahead to write the method:

```
class Enrollment {  
    String studentId;  
  
    Enrollment(String studentId, String courseCode) {  
        this.studentId = studentId;  
    }  
    String getStudentId() {  
        return studentId;  
    }  
}
```

As a side note, now it is the first time that we make use of the student id passed to the constructor of the enrollment. It means it has been useless until now because the tests didn't require the student id.

Now run the tests. Surprisingly, both `testStudentRegistered` and `testStudentUnregistered` fail! The bug is very likely to be in the code (about 10 lines) we just added. The most likely place is the `assertValid` method:

```
void assertValid(Enrollment enrollment) {  
    if (studentRegistryChecker.isRegistered(enrollment.getStudentId())) {  
        throw new StudentNotFoundException();  
    }  
}
```

Obviously this is throwing an exception when the student is registered. This is wrong. So, let's fix it:

```
void assertValid(Enrollment enrollment) {  
    if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {  
        throw new StudentNotFoundException();  
    }  
}
```

Now run the tests again. We see a green bar. From this incident we can see the benefit of "test a little, code a little": if we introduce a bug, it is detected right away and it can be easily located. Compare this with our original approach where we wrote about 150 lines of code that remained untested for hours.

Testing if there is a free seat

Now, we will pick another thing to test. For example, we may test if it checks if the course exists. However, it should be very similar to the checking of the student, so there is little that we can learn by implementing it. In general, we should pick something that is not too hard to do but we can still learn from it. In this case, we may pick the free seat checking. However, it is still too large. We can ignore the part about reservations for now and concentrate on the seats that are really taken. First, add the part about reservations to our todo list:

<i>TODO</i>
Test if add really calls <code>assertValid</code> .
Test if reservations are considered.

Now, let's test if it is considering the seats taken by enrollments. In the test we will set the total number of seats to say two, make the enrollment set believe that two students have enrolled, then try to validate a new enrollment. If it works, it should throw a `ClassFullException`:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        //How to make enrollmentSet believe two students have enrolled in c001?
        try {
            enrollmentSet.assertValid(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
}
```

The question is: How to make `enrollmentSet` believe two students have enrolled in course `c001`? We may add two enrollments there, but it is quite troublesome because we will have to create two enrollments, make sure they are valid and then implement reading enrollments from the storage. This is too much work. So, let's use a `EnrollmentCounter` instead:

```
interface EnrollmentCounter {
    int getNoEnrollments(String courseCode);
}
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        try {
```

```

        enrollmentSet.assertValid(enrollment);
        fail();
    } catch (ClassFullException e) {
    }
}
}

```

But we will need to make enrollmentSet believe that the total number of seats for course c001 is two. To do that, we may let it use a CourseLookup to find the Course object and then get the class size in it. But having to create a Course object is still too much work. A simpler way is to let it use a ClassSizeGetter:

```

public interface ClassSizeGetter {
    public int getClassSize(String courseCode);
}
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        try {
            enrollmentSet.assertValid(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
}
}

```

The test won't compile yet. Create ClassFullException, the setClassSizeGetter and the setEnrollmentCounter methods so that the test compiles:

```

class ClassFullException extends RuntimeException {
}
class EnrollmentSet {
    EnrollmentStorage storage;
    StudentRegistryChecker studentRegistryChecker;

    void assertValid(Enrollment enrollment) {
        ...
    }
    void setClassSizeGetter(ClassSizeGetter sizeGetter) {
    }
    void setEnrollmentCounter(EnrollmentCounter counter) {
    }
}

```

Now run the tests. It fails. But the error is unusual: It is a `NullPointerException`. By further inspection, it is because the `studentRegistryChecker` in the `EnrollmentSet` is null. Right, it is checking if the student `s001` is registered, but we haven't setup a `StudentRegistryChecker`. To solve this problem, we could copy the code to setup a `StudentRegistryChecker`:

```
void testAllSeatsTaken() {
    Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
        boolean isRegistered(String studentId) {
            return false;
        }
    });
    enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
        int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
        }
    });
    enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
        int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
        }
    });
    try {
        enrollmentSet.assertValid(enrollment);
        fail();
    } catch (ClassFullException e) {
    }
}
```

But this is too much work. In fact, we are testing the student registration checking again. As we have said before, it is generally not good to test the same thing again. This is suggesting that `EnrollmentSet` should have an `assertHasSeat` method so that we can test this method alone. Of course, we will also need to test that the `add` method calls `assertHasSeat` before adding the enrollment to the storage. Accordingly, the `assertValid` method should be called `assertStudentRegistered` instead:

```
class EnrollmentSet {
    ...
    void assertStudentRegistered(Enrollment enrollment) {
        ...
    }
    void assertHasSeat(Enrollment enrollment) {
    }
}
class EnrollmentSetTest extends TestCase {
    void testAddEnrollment() {
        ...
    }
    void testStudentRegistered() {
        ...
        enrollmentSet.assertStudentRegistered(enrollment);
        ...
    }
    void testStudentUnregistered() {
        ...
    }
}
```

```

        enrollmentSet.assertStudentRegistered(enrollment);
        ...
    }
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
}

```

Now run the tests again. The current test fails. This is expected. Let's write the implementation code to make the test pass:

```

class EnrollmentSet {
    ...
    void assertStudentRegistered(Enrollment enrollment) {
        ...
    }
    void assertHasSeat(Enrollment enrollment) {
        String courseCode = enrollment.getCourseCode();
        if (enrollmentCounter.getNoEnrollments(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
}

```

It uses a `getCourseCode` method of the `Enrollment` class. As it is just a simple getter, we will implement it without writing a failing test for it:

```

class Enrollment {
    String studentId;
    String courseCode;

    Enrollment(String studentId, String courseCode) {
        this.studentId = studentId;
        this.courseCode = courseCode;
    }
    String getStudentId() { ... }
    String getCourseCode() {
        return courseCode;
    }
}

```

This is the first time that we use the `courseCode` passed to the `Enrollment`'s constructor.

Now, run the tests. They pass. To be sure that it is really working, we should test the case when there are indeed free seats. We can basically copy the code of `testAllSeatsTaken`. We simply change the number of enrollments from two to one and change to expect that it will not throw any exception:

```
public void testHasAFreeSeat () {
    Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
        int getClassSize(String courseCode) {
            return courseCode.equals("c001") ? 2 : 0;
        }
    });
    enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
        int getNoEnrollments(String courseCode) {
            return courseCode.equals("c001") ? 1 : 0;
        }
    });
    enrollmentSet.assertHasSeat(enrollment);
}
```

Now run the tests. They pass.

Testing if enrollments in the parent course are considered

But we haven't finished testing it yet. We still need to test a very interesting behavior: When looking for a free seat, it should not only consider the course itself, but also its parent course if it is a module. For example, if `c002` is a module of `c001` and there are three enrollments for `c001` and two enrollments for `c002`, if the class size of `c002` is five, it is already full. At the moment, we are using the `getNoEnrollments` method of `EnrollmentCounter` to find the number of enrollments. For `c002` in the above example, will it return two or five? If it returns two, `EnrollmentSet` must check the parent of `c002`. If it returns five, there is nothing `EnrollmentSet` needs to do. To take the easier route, we will assume that it will return five. To make it more explicit, let rename `getNoEnrollments` to `getNoSeatsTaken` (`Enrollment` is just one of the possible reasons for a seat in a module being taken):

```
interface EnrollmentCounter {
    int getNoSeatsTaken(String courseCode);
}
```

Just to be safe, run the tests again and they should pass.

Testing if reservations are considered

Next, pick a task from our todo list:

<i>TODO</i>
Test if add really calls <code>assertStudentRegistered</code> and <code>assertHasSeat</code> .
Test if reservations are considered.

Let's do the second task. We will setup a `ReservationCounter` to return the reservations for a given course (including its parent if it is a module). We will setup the context so that some seats are taken and the rest are reserved. This is to test that `assertHasSeat` will consider both enrollments and reservations. It is very similar to `testAllSeatsTaken`:

```
public interface ReservationCounter {
    public int getNoSeatsReserved(String courseCode);
}
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsReservedOrTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 3 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
                return courseCode.equals("c001") ? 1 : 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
}
```

Create an empty `setReservationCounter` method to make test compile:

```
class EnrollmentSet {
    ...
    void setReservationCounter(ReservationCounter counter) {
```

```
    }
}
```

Run the tests and the current one fails. This is expected. Write the implementation code so that the test passes:

```
class EnrollmentSet {
    ...
    ReservationCounter reservationCounter;
    ...
    public void assertHasSeat(Enrollment enrollment) {
        String courseCode = enrollment.getCourseCode();
        if (enrollmentCounter.getNoSeatsTaken(courseCode)
            + reservationCounter.getNoSeatsReserved(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
    void setReservationCounter(ReservationCounter counter) {
        this.reservationCounter = counter;
    }
}
```

Now run the tests. The current one passes but two previous tests fail: `testAllSeatsTaken` and `testHasAFreeSeat`. They both trigger a `NullPointerException`. This is because in these tests we didn't setup a `ReservationCounter` for the `EnrollmentSet` to use. A quick way to fix it is to setup a `ReservationCounter` in these tests. For example:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001");
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoEnrollments(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat(enrollment);
            fail();
        } catch (ClassFullException e) {
        }
    }
    public void testHasAFreeSeat() {
        ...
    }
}
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
        ...
    }
}
```

Now run the tests again and they should pass.

Testing if add performs validation

Next we will pick the last task on our todo list:

TODO
Test if add really calls assertStudentRegistered and assertHasSeat.

We can do it like this:

```
class EnrollmentSetTest extends TestCase {
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
            void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment("s001", "c001");
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}
```

This test assumes that add will first call assertStudentRegistered, then assertHasSeat and finally add the enrollment to the storage. The StringBuffer callLog is used to log the calls. For it to be accessible to the inner classes, it has to be final. We cannot use a String instead because a String object cannot be changed.

Now run the tests and the current test fails. This is expected because at the moment the add

method is not validating the enrollment. Add the code so that it does:

```
class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        storage.add(enrollment);
    }
    ...
}
```

The current test passes, but the first test fails with a NullPointerException:

```
void testAddEnrollment() {
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.add(enrollment);
}
```

This is because now add will validate the enrollment but we have not setup the needed objects yet (e.g., StudentRegistryChecker, ClassSizeGetter, etc.). We could setup all these objects but it is too much work. As always we should separate the code that is being tested into a new method so that our tests don't test the same thing. Let's call the new method addToStorage:

```
void testAddEnrollment() {
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.addToStorage(enrollment);
}
```

Create an empty addToStorage method:

```
class EnrollmentSet {
    ...
    void addToStorage(Enrollment enrollment) {
    }
}
```

As it is now testing addToStorage instead of testing add, let's rename it from testAddEnrollment to testAddToStorage:

```
void testAddToStorage() {
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

        void add(Enrollment enrollmentToStore) {
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.addToStorage(enrollment);
}

```

Run the test and expect it to fail. But it doesn't! At the moment `addToStorage` is not really adding the enrollment to the storage. But the `add` method of our storage object is never called at all. So the `assertTrue` inside is simply not executed. This means that this test is not working. That is, it is useless. This is why we need to see a failing test before writing implementation code. If the test indeed fails, it means the test should be working. If it doesn't fail, it means the test is broken. That is, we are testing the test itself with this procedure. Now back to the case at hand. How to correct the test? We can use a call log again:

```

void testAddToStorage() {
    final StringBuffer callLog = new StringBuffer();
    final Enrollment enrollment = new Enrollment("s001", "c001");
    EnrollmentSet enrollmentSet = new EnrollmentSet();
    enrollmentSet.setStorage(new EnrollmentStorage() {
        void add(Enrollment enrollmentToStore) {
            callLog.append("x");
            assertTrue(enrollmentToStore == enrollment);
        }
    });
    enrollmentSet.addToStorage(enrollment);
    assertEquals(callLog.toString(), "x");
}

```

Now run it again and it should fail. This is good. Write the implementation code so that the test passes:

```

class EnrollmentSet {
    ...
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    void addToStorage(Enrollment enrollment) {
        storage.add(enrollment);
    }
}

```

Now run the tests and they should pass.

Changing Enrollment's constructor breaks all existing tests

Now, our todo list is empty, but it doesn't mean that we have nothing to do. From the user story it is clear that we still need to do many things such as checking the payment, saving the

enrollment data to the database and etc. Now, let's test to see if it will check the payment. We will store a payment in an enrollment whose amount is equal to the course fee, call `assertPaymentCorrect` and expect it to pass:

```
interface CourseFeeLookup {
    int getFee(String courseCode);
}
class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentCorrect() {
        Enrollment enrollment = new Enrollment("s001", "c001", new CashPayment(100));
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
            int getFee(String courseCode) {
                return courseCode.equals("c001") ? 100 : 0;
            }
        });
        enrollmentSet.assertPaymentCorrect(enrollment);
    }
}
```

Note that we are passing a payment object as the third parameter of the constructor for `Enrollment`. This sounds fine. After all, an enrollment must have a payment. To make the test compile, we need to add this parameter to the constructor:

```
class Enrollment {
    Enrollment(String studentId, String courseCode, Payment payment) {
        ...
    }
}
```

However, this will break all the existing tests:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testAddEnrollment() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
        ...
    }
    void testStudentRegistered() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
        ...
        enrollmentSet.assertStudentRegistered(enrollment);
    }
    void testStudentUnregistered() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
        ...
        enrollmentSet.assertStudentRegistered(enrollment);
        ...
    }
    void testAllSeatsTaken() {
        Enrollment enrollment = new Enrollment("s001", "c001"); //No such constructor
        ...
        enrollmentSet.assertHasSeat(enrollment);
        ...
    }
}
```

```
}
```

But for all these tests such as `testStudentRegistered` and `testAllSeatsTaken`, they don't care about the payment at all. This suggests that they shouldn't work with an `Enrollment`. They should only work on what they really care. For example, `testStudentRegistered` should only work with the student id. Therefore, we should fix them. However, we do not like having more than one errors in the code. So, comment out the `testPaymentCorrect` method and keep both constructors for `Enrollment` for the moment:

```
class Enrollment {
    ...
    Enrollment(String studentId, String courseCode) {
        ...
    }
    Enrollment(String studentId, String courseCode, Payment payment) {
    }
}
```

Note that the new constructor is empty. This looks weird. If it is empty then it is definitely not going to work. There are several options here. We may write the code without a failing test:

```
class Enrollment {
    ...
    Enrollment(String studentId, String courseCode, Payment payment) {
        this.studentId = studentId;
        this.courseCode = courseCode;
        this.payment = payment;
    }
}
```

We may create a new class `EnrollmentTest` to test it first:

```
class EnrollmentTest extends TestCase {
    void testCreation() {
        Payment payment = ...;
        Enrollment enrollment = new Enrollment("a", "b", payment);
        assertEquals(enrollment.getStudentId(), "a");
        assertEquals(enrollment.getCourseCode(), "b");
        assertTrue(enrollment.getPayment() == payment);
    }
}
```

We may create a test in `EnrollmentSetTest` that fails until we implement the constructor. Any of these options should be fine. Suppose that in this case we decide to take the last option. However, we can't do it right away because we're in the process of migrating to the new constructor. So, add the task to our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.

Then, change the tests one by one to make sure they don't use the old `Enrollment` constructor.

Let's consider the first test: `testAddToStorage`. It needs to pass an `Enrollment` to `addToStorage` because `addToStorage` is required to pass it to the `EnrollmentStorage`, but the `Enrollment` object doesn't have to contain meaningful data. So, let's update the test to use the new constructor using "null" data:

```
class EnrollmentSetTest extends TestCase {
    public void testAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        final Enrollment enrollment = new Enrollment(null, null, null);
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("x");
                assertTrue(enrollmentToStore == enrollment);
            }
        });
        enrollmentSet.addToStorage(enrollment);
        assertEquals(callLog.toString(), "x");
    }
    ...
}
```

Run the test to make sure it passes. Then consider `testStudentRegistered`. It should only need the student id instead of an enrollment object:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testStudentRegistered() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return studentId.equals("s001");
            }
        });
        enrollmentSet.assertStudentRegistered("s001");
    }
}
```

This will not compile because currently `assertStudentRegistered` takes an `Enrollment` as parameter. We could change it to take the student id as parameter, but this may break other tests. Because we are in the process of migrating from the two-parameter `Enrollment` constructor to the three-parameter version, we'd like to avoid changing `assertStudentRegistered` at the same time. Instead, let's copy `assertStudentRegistered` to create an overloaded version that takes the student id as parameter:

```
class EnrollmentSet {
    ...
    void assertStudentRegistered(Enrollment enrollment) {
        if (!studentRegistryChecker.isRegistered(enrollment.getStudentId())) {
            throw new StudentNotFoundException();
        }
    }
    void assertStudentRegistered(String studentId) {
        if (!studentRegistryChecker.isRegistered(studentId)) {
            throw new StudentNotFoundException();
        }
    }
}
```

```

    }
}

```

After migrating to the new Enrollment constructor, we will get rid of the `assertStudentRegistered` that takes an enrollment as parameter. So, add this task to our todo list:

<i>TODO</i>
Write test to force us to write code in new Enrollment constructor.
Get rid of <code>assertStudentRegistered</code> that takes an enrollment.

Now run the test again and it should pass. Next consider `testStudentUnregistered`. It is similar to `testStudentRegistered`:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testStudentUnregistered() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setStudentRegistryChecker(new StudentRegistryChecker() {
            boolean isRegistered(String studentId) {
                return false;
            }
        });
        try {
            enrollmentSet.assertStudentRegistered("s001");
            fail();
        } catch (StudentNotFoundException e) {
        }
    }
}

```

Now consider `testAllSeatsTaken`. It should only need the course id instead of an enrollment object:

```

class EnrollmentSetTest extends TestCase {
    ...
    public void testAllSeatsTaken() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            public int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            public int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            public int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
    }
}

```

```

    });
    try {
        enrollmentSet.assertHasSeat("c001");
        fail();
    } catch (ClassFullException e) {
    }
}
}

```

We need to create an overloaded version of `assertHasSeat`:

```

class EnrollmentSet {
    ...
    void assertHasSeat(Enrollment enrollment) {
        ...
    }
    void assertHasSeat(String courseCode) {
        if (enrollmentCounter.getNoSeatsTaken(courseCode)
            + reservationCounter.getNoSeatsReserved(courseCode)
            >= classSizeGetter.getClassSize(courseCode)) {
            throw new ClassFullException();
        }
    }
}

```

Add a task to delete the old `assertHasSeat` to our todo list:

<i>TODO</i>
Write test to force us to write code in new Enrollment constructor.
Get rid of <code>assertStudentRegistered</code> that takes an enrollment.
Get rid of <code>assertHasSeat</code> that takes an enrollment.

Run the tests and they should still pass. Similarly, update `testHasAFreeSeat` to use a course id:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testHasAFreeSeat() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 1 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            int getNoSeatsReserved(String courseCode) {
                return 0;
            }
        });
    }
}

```

```

    });
    enrollmentSet.assertHasSeat("c001");
}
}

```

Run the tests and they should still pass. Similarly, update `testAllSeatsReservedOrTaken` to use a course id:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testAllSeatsReservedOrTaken() {
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setClassSizeGetter(new ClassSizeGetter() {
            public int getClassSize(String courseCode) {
                return courseCode.equals("c001") ? 3 : 0;
            }
        });
        enrollmentSet.setEnrollmentCounter(new EnrollmentCounter() {
            public int getNoSeatsTaken(String courseCode) {
                return courseCode.equals("c001") ? 2 : 0;
            }
        });
        enrollmentSet.setReservationCounter(new ReservationCounter() {
            public int getNoSeatsReserved(String courseCode) {
                return courseCode.equals("c001") ? 1 : 0;
            }
        });
        try {
            enrollmentSet.assertHasSeat("c001");
            fail();
        } catch (ClassFullException e) {
        }
    }
}

```

Run the tests and they should still pass. Now, consider `testValidateBeforeAddToStorage`. It is only testing the calling sequence. It doesn't need an enrollment object at all. It can happily use a null:

```

class EnrollmentSetTest extends TestCase {
    ...
    public void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            public void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
            public void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        enrollmentSet.add(null);
    }
}

```

```

        assertEquals(callLog.toString(), "xyz");
    }
}

```

Run the tests and they should still pass.

By now we should have eliminated all the calls to the two-parameter version of the Enrollment constructor. Let's delete it. Then run all the tests and they should still pass.

Now, check our todo list:

<i>TODO</i>
Write test to force us to write code in new Enrollment constructor.
Get rid of assertStudentRegistered that takes an enrollment.
Get rid of assertHasSeat that takes an enrollment.

Let's do the second task. First, check if the old version of assertStudentRegistered is still being called by any code at all (some IDEs can do this for you). We find it is still being used at one place:

```

class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment);
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    ...
}

```

That's easy: just change it to use the new version:

```

class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment);
        addToStorage(enrollment);
    }
    ...
}

```

Run the tests but testValidateBeforeAddToStorage fails with a NullPointerException:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(Enrollment enrollment) {
                callLog.append("x");
            }
        };
        void assertHasSeat(Enrollment enrollment) {

```

```

        callLog.append("y");
    }
};
enrollmentSet.setStorage(new EnrollmentStorage() {
    public void add(Enrollment enrollmentToStore) {
        callLog.append("z");
    }
});
enrollmentSet.add(null);
assertEquals(callLog.toString(), "xyz");
}
}

```

This is because in the add method, we are trying to get the student id. In that case in the unit test we should no longer pass null as the enrollment object. In addition, we find that we are overriding the assertStudentRegistered method in this test. We should change it to use the new version:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                callLog.append("x");
            }
            void assertHasSeat(Enrollment enrollment) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}

```

As there should be no more code using the original assertStudentRegistered, let's delete it. Observe that there is no compile error. Run the tests again. Yes, they continue to pass.

Now, check our todo list:

<i>TODO</i>
Write test to force us to write code in new Enrollment constructor.
Get rid of assertHasSeat that takes an enrollment.

Let's do the last task. This is very similar to how we got rid of assertStudentRegistered. Therefore, we will only show the final changes:

```

class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment.getCourseCode());
        addToStorage(enrollment);
    }
    ...
}
class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}

```

Now, check our todo list:

TODO
Write test to force us to write code in new Enrollment constructor.

This is interesting. We are using the new Enrollment constructor. Even though it is empty (i.e., it does nothing at all), our tests still pass. It probably means our tests are missing something. Let's check the code of EnrollmentSet to see where it gets the student id or course code from an enrollment. It is in the add method:

```

class EnrollmentSet {
    ...
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment.getCourseCode());
        addToStorage(enrollment);
    }
}

```

It means these parts of the code are not being tested. At the moment we are testing the call sequence inside the add method, but are not checking the arguments passed to those methods:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}

```

Therefore, we should also check the arguments actually passed:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {
            void assertStudentRegistered(String studentId) {
                assertTrue(studentId==null);
                callLog.append("x");
            }
            void assertHasSeat(String courseCode) {
                assertTrue(courseCode==null);
                callLog.append("y");
            }
        };
        enrollmentSet.setStorage(new EnrollmentStorage() {
            public void add(Enrollment enrollmentToStore) {
                callLog.append("z");
            }
        });
        Enrollment enrollment = new Enrollment(null, null, null);
        enrollmentSet.add(enrollment);
        assertEquals(callLog.toString(), "xyz");
    }
}

```

The arguments should be null because we are using null as the student id and course code. However, this is not a good test. A better test is have non-null student id and course code:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        EnrollmentSet enrollmentSet = new EnrollmentSet() {

```

```

    void assertStudentRegistered(String studentId) {
        assertEquals(studentId, "a");
        callLog.append("x");
    }
    void assertHasSeat(String courseCode) {
        assertEquals(courseCode, "b");
        callLog.append("y");
    }
};
enrollmentSet.setStorage(new EnrollmentStorage() {
    public void add(Enrollment enrollmentToStore) {
        callLog.append("z");
    }
});
Enrollment enrollment = new Enrollment("a", "b", null);
enrollmentSet.add(enrollment);
assertEquals(callLog.toString(), "xyz");
}
}

```

Now run the test again and it should fail. This is exactly our purpose. We'd like to force ourselves to create some code in the Enrollment constructor. So, let's do it:

```

class Enrollment {
    Enrollment(String studentId, String courseCode, Payment payment) {
        this.studentId = studentId;
        this.courseCode = courseCode;
    }
    ...
}

```

Note that the payment parameter is still not used because our test doesn't require it. So, update our todo list:

TODO
Write test to force us to store the payment in the Enrollment constructor.

Now run the tests and now they should all pass. Finally we are ready to test the payment. Uncomment the testPaymentCorrect method:

```

class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentCorrect() {
        Payment payment = new CashPayment(100);
        Enrollment enrollment = new Enrollment("s001", "c001", payment);
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
            int getFee(String courseCode) {
                return courseCode.equals("c001") ? 100 : 0;
            }
        });
        enrollmentSet.assertPaymentCorrect(enrollment);
    }
}
}

```

Learning from the past mistakes we know that it doesn't really need an enrollment object. It only needs a course code (to find course fee) and the payment:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentCorrect() {
        Payment payment = new CashPayment(100);
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
            int getFee(String courseCode) {
                return courseCode.equals("c001") ? 100 : 0;
            }
        });
        enrollmentSet.assertPaymentCorrect("c001", payment);
    }
}
```

To make it compile, create a CashPayment class and an empty constructor, an empty setCourseFeeLookup method and an empty assertPaymentCorrect method:

```
class CashPayment extends Payment {
    CashPayment(int amount) {
    }
}
class EnrollmentSet {
    ...
    void setCourseFeeLookup(CourseFeeLookup lookup) {
    }
    void assertPaymentCorrect(String string, Payment payment) {
    }
}
```

Now run the test. It passes! It means this test is not enough. This is because our assertPaymentCorrect method does nothing and thus any payment is considered correct. So, write another test that should fail: validate an incorrect payment. We can do it by changing the course fee from 100 to say 101 and expect that an IncorrectPaymentException:

```
class EnrollmentSetTest extends TestCase {
    ...
    void testPaymentIncorrect() {
        Payment payment = new CashPayment(100);
        EnrollmentSet enrollmentSet = new EnrollmentSet();
        enrollmentSet.setCourseFeeLookup(new CourseFeeLookup() {
            public int getFee(String courseCode) {
                return courseCode.equals("c001") ? 101 : 0;
            }
        });
        try {
            enrollmentSet.assertPaymentCorrect("c001", payment);
            fail();
        } catch (IncorrectPaymentException e) {
        }
    }
}
```

To make it compile, create an `IncorrectPaymentException` class. Then run the test and it should fail. Now, write the implementation code so that the test passes:

```
class EnrollmentSet {
    CourseFeeLookup courseFeeLookup;
    ...
    void setCourseFeeLookup(CourseFeeLookup lookup) {
        this.courseFeeLookup = lookup;
    }
    void assertPaymentCorrect(String courseCode, Payment payment) {
        if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
            throw new IncorrectPaymentException();
        }
    }
}
```

This will force us to create the `getAmount` method and quite some related code:

```
abstract class Payment {
    abstract int getAmount();
}
class CashPayment extends Payment {
    int amount;
    public CashPayment(int amount) {
        this.amount = amount;
    }
    int getAmount() {
        return amount;
    }
}
```

Now run the tests again and they should pass. By now it seems that we're done with the payment checking. But we are not.

Maintaining an integrated mental picture of EnrollmentSet

There is a bug in `EnrollmentSet`. If we run the acceptance tests for this story, we will find that it doesn't check the payment before saving the enrollment to the database. This bug may also be found by inspecting the code of `EnrollmentSet` from start to end:

```
class EnrollmentSet {
    EnrollmentStorage storage;
    StudentRegistryChecker studentRegistryChecker;
    EnrollmentCounter enrollmentCounter;
    ClassSizeGetter classSizeGetter;
    ReservationCounter reservationCounter;
    CourseFeeLookup courseFeeLookup;

    void setStorage(EnrollmentStorage storage) {
        this.storage = storage;
    }
}
```

```

void add(Enrollment enrollment) {
    assertStudentRegistered(enrollment.getStudentId());
    assertHasSeat(enrollment.getCourseCode());
    //Not calling assertPaymentCorrect!
    addToStorage(enrollment);
}
void assertStudentRegistered(String studentId) {
    if (!studentRegistryChecker.isRegistered(studentId)) {
        throw new StudentNotFoundException();
    }
}
void assertHasSeat(String courseCode) {
    if (enrollmentCounter.getNoSeatsTaken(courseCode)
        + reservationCounter.getNoSeatsReserved(courseCode)
        >= classSizeGetter.getClassSize(courseCode)) {
        throw new ClassFullException();
    }
}
void setStudentRegistryChecker(StudentRegistryChecker registryChecker) {
    this.studentRegistryChecker = registryChecker;
}
void setClassSizeGetter(ClassSizeGetter sizeGetter) {
    this.classSizeGetter = sizeGetter;
}
void setEnrollmentCounter(EnrollmentCounter counter) {
    this.enrollmentCounter = counter;
}
void setReservationCounter(ReservationCounter counter) {
    this.reservationCounter = counter;
}
void addToStorage(Enrollment enrollment) {
    storage.add(enrollment);
}
void setCourseFeeLookup(CourseFeeLookup lookup) {
    this.courseFeeLookup = lookup;
}
void assertPaymentCorrect(String courseCode, Payment payment) {
    if (courseFeeLookup.getFee(courseCode) != payment.getAmount()) {
        throw new IncorrectPaymentException();
    }
}
}
}

```

We did test the behavior of `assertPaymentCorrect` but we forgot to test that it is indeed called by the `add` method. In fact, in the whole development process, our idea on the actual code of the whole `EnrollmentSet` class is moot. Because each behavior is tested in isolation, it is easy to lose sight of the whole picture. Then integration bugs may occur. Therefore, we should regularly review the whole class to maintain an integrated mental picture.

Returning to the bug, as always, write a failing test to show the bug. We do this by amending the `testValidateBeforeAddToStorage` method:

```

class EnrollmentSetTest extends TestCase {
    void testValidateBeforeAddToStorage() {
        final StringBuffer callLog = new StringBuffer();
        final Payment payment = new CashPayment(5);
        EnrollmentSet enrollmentSet = new EnrollmentSet() {

```

```

    void assertStudentRegistered(String studentId) {
        assertEquals(studentId, "a");
        callLog.append("x");
    }
    void assertHasSeat(String courseCode) {
        assertEquals(courseCode, "b");
        callLog.append("y");
    }
    void assertPaymentCorrect(
        String courseCode,
        Payment payment2) {
        assertEquals(courseCode, "b");
        assertTrue(payment2 == payment);
        callLog.append("t");
    }
};
enrollmentSet.setStorage(new EnrollmentStorage() {
    void add(Enrollment enrollmentToStore) {
        callLog.append("z");
    }
});
Enrollment enrollment = new Enrollment("a", "b", payment);
enrollmentSet.add(enrollment);
assertEquals(callLog.toString(), "xytz");
}
}

```

Run it and it fails. This is good. Write the required implementation code:

```

class EnrollmentSet {
    void add(Enrollment enrollment) {
        assertStudentRegistered(enrollment.getStudentId());
        assertHasSeat(enrollment.getCourseCode());
        assertPaymentCorrect(enrollment.getCourseCode(), enrollment.getPayment());
        addToStorage(enrollment);
    }
}

```

This will in turn force us to write the `getPayment` method in `Enrollment` and complete its constructor:

```

class Enrollment {
    String studentId;
    String courseCode;
    Payment payment;

    Enrollment(String studentId, String courseCode, Payment payment) {
        this.studentId = studentId;
        this.courseCode = courseCode;
        this.payment = payment;
    }
    Payment getPayment() {
        return payment;
    }
}

```

Now run the test again and it should pass.

TDD and its benefits

The way we developed the code as shown above is called "Test Driven Development (TDD)" because we always write a failing test before we code the real thing. TDD has the following benefits over writing tests last:

- In order to easily write a unit test, we make extensive use of interfaces (e.g., StudentRegistryChecker and etc.). This makes unit tests very easy to write and read because there is no unneeded data in the tests. If we were not using TDD and just coded the implementation directly, we would easily make use of existing classes (e.g., StudentSet) and would have to pack a lot of unneeded data in the tests (e.g., creating huge objects like Student or Course).
- Due to the extensive use of interfaces, our classes are separate from one another (e.g., EnrollmentSet doesn't know about StudentSet at all) and thus are much more reusable.
- When writing unit tests, it is easy to write a test for one behavior, make it pass, then write another test for another behavior. That is, the whole task is divided into many small pieces to be individually conquered. If we were not using TDD and just coded the implementation directly, we would easily try to implement all the behaviors at the same time. This would take a lot of time and the code would remain untested for quite a long time. In contrast, with TDD, we only implement the code for the behavior being tested. Therefore it only takes a very little time (minutes) to do and we can test the code right away.

Do's and Dont's

- Do not include unneeded data in a test (e.g., if you only need a student id, don't create an enrollment object; if you only need to check the existence of a student, use a StudentRegistryCheck interface instead of the StudentSet class). To do that, you will mostly use interfaces and anonymous classes.
- Do not bear the pain of writing a lot of code in a test to setup the context (e.g., setup a database connection). If it happens, use an interface (e.g., EnrollmentStorage).
- Do not test the same thing in two tests (e.g., when testing validation, do not test the writing to the storage).

- Do not write any code without a failing test, unless that code is very simple.
- Do not (at least try to avoid) break more than one tests at a time (e.g., if you need to add a parameter to Enrollment's constructor, add a new constructor along with the existing one, migrate all clients to the new one, run all tests and finally delete the old one).
- Do not try to do many things at the same time or do something that takes hours to do. Try to break it down into smaller behaviors. Use a todo list.
- Do write a test and make it pass in a few minutes.
- Do run all the tests after making one or a few small changes.
- Do pick a task that can teach you something before those that are boring.
- Do use a call log to test the calling sequence.
- Do strive to maintain a whole picture at any time.

References

- Kent Beck, Test Driven Development: By Example, Addison-Wesley, 2002.
- David Astels, Test-Driven Development, A Practical Guide by, Prentice Hall, 2003.
- <http://www.objectmentor.com/writeUps/TestDrivenDevelopment>.
- <http://www.objectmentor.com/resources/articles/xpepisode.htm>.
- <http://www.mockobjects.com>.
- <http://www.testdriven.com>.

Chapter exercises

Introduction

- You must use TDD to develop the solutions.
- You must use a todo list.

Problems

1. Develop the code to count the number of files directly in a given directory whose sizes are at least 2GB. Sub-directories are ignored and not considered as files.
2. Develop the code to delete a directory and all the files inside. To delete a directory, you must empty it first.
3. Develop the code to generate a unique code for each fax. The code consists of three parts and is like 1/2004/HR. The first part is a sequence number that starts from 1. The next time it will be 2. The second part is the current year. The third part is the id of the department that issues the fax. Each department will have its own sequence number.
4. Develop the code to check if two employees are equal. An employee has an employee id, a list of qualifications, a superior (also an employee). A qualification has a text description and indicates the year when the qualification was achieved.
5. Develop a `EnrollmentDBStorage` class implementing the `EnrollmentStorage` interface. It should be able to write an `Enrollment` object into a database. You can use the schema shown in the text or make up your own. You can assume the name of the database, the driver used and etc.
6. Develop the code to find the number of seats taken for a given course (due to enrollments in it or in its parent). As part of it, you should have a class implementing the `EnrollmentCounter` shown in the text. When you need to access the database, use an interface instead.
7. Develop the code to find the number of seats reserved for a given course (due to reservations for it or its parent). As part of it, you should have a class implementing the `ReservationCounter` shown in the text. When you need to access the database, use an interface instead. You should only count the reservations that are s



Hints

1. Consider using the following interfaces to replace the file system:

```
public interface FileLookup {
    public String[] getFilesIn(String dirPath);
}
public interface DirChecker {
    public boolean isDir(String path);
}
public interface FileSizeGetter {
    public long getSize(String path);
}
```

2. It is similar to the previous one.
3. Consider using the following interfaces to replace the system clock:

```
public interface YearGetter {
    public int getFullYear();
}
```

To avoid testing the same thing in different tests, you may imagine that the `FaxCodeGenerator` is like:

```
public class FaxCodeGenerator {
    public String getSequenceNo() {
        ...
    }
    public String generate() {
        return getSequenceNo() + ...
    }
}
```

and then test the `getSequenceNo` method and the `generate` method separately. Alternatively, you may separate the behavior of `getSequenceNo` into a separate class:

```
public class SequenceNumberGenerator {
    public String getSequenceNo() {
        ...
    }
}
public class FaxCodeGenerator {
    public String generate() {
        ...
    }
}
```

and then test each class separately.

4. In the tests you will need to create some employee objects, qualification list objects and etc. Try to use as little data as possible as it will turn out to be useless. Try using nulls as long as appropriate.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agilekills.org

5. As this code must access the database, you shouldn't use an interface to replace the database. So, you must create a database and a table in order to run these tests.

6. The following interfaces should make it easier to write the tests:

```
public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}
public interface EnrollmentSource {
    public List getEnrollmentsFor(String courseCode);
}
```

In addition, you can use the following interface to replace the database:

```
public interface CourseSource {
    public Course getCourse(String courseCode);
}
```

7. It is similar to the previous one. However, the code developed would be similar to the code in the previous problem. It'd be great if you'd remove the duplication (both the tests and the implementation code).

Sample solutions

1. Develop the code to count the number of files directly in a given directory whose sizes are at least 2GB. Sub-directories are ignored and not considered as files.

The major problem here is that to setup the test context, we need to have a directory to put some files into. In particular, some of the files in it must be \geq 2GB. We must also ensure the directory is initially empty, otherwise the test has to clean up its contents first. We must also ensure that the directory doesn't contain useful files. Therefore, setting up the context will take a lot of care, code, execution time (to create large files) and disk space. To solve this problem, we can use some interfaces to replace the system file:

```
public interface FileLookup {
    public String[] getFilesIn(String dirPath);
}
public interface DirChecker {
    public boolean isDir(String path);
}
public interface FileSizeGetter {
    public long getSize(String path);
}
```

The tests are:

```
public class FileCounterTest extends TestCase {
    public void testCount() {
        final long KB = 1024;
        final long MB = 1024 * KB;
        final long GB = 1024 * MB;
        FileCounter fileCounter = new FileCounter();
        fileCounter.setFileSizeGetter(new FileSizeGetter() {
            public long getSize(String path) {
                if (path.equals("dir/a")) {
                    return 2 * GB;
                }
                if (path.equals("dir/b")) {
                    return 2 * GB - 1;
                }
                if (path.equals("dir/c")) {
                    return 2 * GB + 1;
                }
                fail();
                return 0;
            }
        });
        fileCounter.setDirChecker(new DirChecker() {
            public boolean isDir(String path) {
                return false;
            }
        });
        fileCounter.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                if (dirPath.equals("dir")) {
                    return new String[] { "a", "b", "c" };
                }
            }
        });
    }
}
```

```

        return null;
    }
    });
    assertEquals(fileCounter.countLargeFilesIn("dir"), 2);
}
public void testIgnoreDirectories() {
    FileCounter fileCounter = new FileCounter();
    fileCounter.setFileSizeGetter(new FileSizeGetter() {
        public long getSize(String path) {
            fail();
            return 0;
        }
    });
    fileCounter.setDirChecker(new DirChecker() {
        public boolean isDir(String path) {
            return true;
        }
    });
    fileCounter.setFileLookup(new FileLookup() {
        public String[] getFilesIn(String dirPath) {
            if (dirPath.equals("dir")) {
                return new String[] { "dl" };
            }
            return null;
        }
    });
    assertEquals(fileCounter.countLargeFilesIn("dir"), 0);
}
}

```

The implementation code is:

```

public class FileCounter {
    private FileLookup fileLookup;
    private FileSizeGetter fileSizeGetter;
    private DirChecker dirChecker;

    public void setFileLookup(FileLookup lookup) {
        this.fileLookup = lookup;
    }
    public void setDirChecker(DirChecker checker) {
        this.dirChecker = checker;
    }
    public void setFileSizeGetter(FileSizeGetter getter) {
        this.fileSizeGetter = getter;
    }
    public int countLargeFilesIn(String pathToDir) {
        final long KB = 1024;
        final long MB = 1024 * KB;
        final long GB = 1024 * MB;
        String files[] = fileLookup.getFilesIn(pathToDir);
        int noLargeFiles = 0;
        for (int i = 0; i < files.length; i++) {
            if (!dirChecker.isDir(files[i])) {
                if (fileSizeGetter.getSize(pathToDir + "/" + files[i])
                    >= 2 * GB) {
                    noLargeFiles++;
                }
            }
        }
    }
}

```

```

    }
    return noLargeFiles;
}
}

```

Ultimately we have to use the file system to implement these interfaces. This code is not tested by the unit tests so it'd better be as thin as possible:

```

public class FileLookupInJava implements FileLookup {
    public String[] getFilesIn(String dirPath) {
        return new File(dirPath).list();
    }
}
public class FileSizeGetterInJava implements FileSizeGetter {
    public long getSize(String path) {
        return new File(path).length();
    }
}
public class DirCheckerInJava implements DirChecker {
    public boolean isDir(String path) {
        return new File(path).isDirectory();
    }
}

```

The FileCounter should use these implementations by default:

```

public class FileCounter {
    public FileCounter() {
        setFileLookup(new FileLookupInJava());
        setFileSizeGetter(new FileSizeGetterInJava());
        setDirChecker(new DirCheckerInJava());
    }
    ...
}

```

2. Develop the code to delete a directory and all the files inside. To delete a directory, you must empty it first.

Like the previous one, we'd use some interfaces to replace the system file:

```

public interface FileLookup {
    public String[] getFilesIn(String path);
}
public interface FileRemover {
    public void delete(String path);
}

```

The tests are:

```

public class RecursiveFileRemoverTest extends TestCase {
    public void testEmpty() {
        final StringBuffer deleteLog = new StringBuffer();
        RecursiveFileRemover remover = new RecursiveFileRemover();
        remover.setFileLookup(new FileLookup() {
            public String[] getFilesIn(String dirPath) {
                return null;
            }
        });
    }
}

```

```

    }
  });
  remover.setFileRemover(new FileRemover() {
    public void delete(String path) {
      deleteLog.append("<" + path + ">");
    }
  });
  remover.delete("dir");
  assertEquals(deleteLog.toString(), "<dir>");
}
public void testDeleteFilesFirst() {
  final StringBuffer deleteLog = new StringBuffer();
  RecursiveFileRemover remover = new RecursiveFileRemover();
  remover.setFileLookup(new FileLookup() {
    public String[] getFilesIn(String dirPath) {
      if (dirPath.equals("dir")) {
        return new String[] { "a", "b" };
      }
      return null;
    }
  });
  remover.setFileRemover(new FileRemover() {
    public void delete(String path) {
      deleteLog.append("<" + path + ">");
    }
  });
  remover.delete("dir");
  assertEquals(deleteLog.toString(), "<dir/a><dir/b><dir>");
}
public void testRecursion() {
  final StringBuffer deleteLog = new StringBuffer();
  RecursiveFileRemover remover = new RecursiveFileRemover();
  remover.setFileLookup(new FileLookup() {
    public String[] getFilesIn(String dirPath) {
      if (dirPath.equals("d1")) {
        return new String[] { "d2" };
      }
      if (dirPath.equals("d1/d2")) {
        return new String[] { "f1" };
      }
      return null;
    }
  });
  remover.setFileRemover(new FileRemover() {
    public void delete(String path) {
      deleteLog.append("<" + path + ">");
    }
  });
  remover.delete("d1");
  assertEquals(deleteLog.toString(), "<d1/d2/f1><d1/d2><d1>");
}
}

```

The implementation code is:

```

public class RecursiveFileRemover {
  private FileLookup fileLookup;
  private FileRemover fileRemover;
  public void setFileLookup(FileLookup lookup) {

```

```

        this.fileLookup = lookup;
    }
    public void setFileRemover(FileRemover remover) {
        this.fileRemover = remover;
    }
    public void delete(String path) {
        String filesInDir[] = fileLookup.GetFilesIn(path);
        if (filesInDir != null) {
            for (int i = 0; i < filesInDir.length; i++) {
                String pathToChild = path + "/" + filesInDir[i];
                delete(pathToChild);
            }
        }
        fileRemover.delete(path);
    }
}

```

The RecursiveFileRemover should by default use implementations of FileLookup and FileRemover that rely on the file system:

```

public class FileLookupInJava implements FileLookup {
    public String[] getFilesIn(String path) {
        return new File(path).list();
    }
}
public class FileRemoverInJava implements FileRemover {
    public void delete(String path) {
        new File(path).delete();
    }
}
public class RecursiveFileRemover {
    public RecursiveFileRemover() {
        setFileLookup(new FileLookupInJava());
        setFileRemover(new FileRemoverInJava());
    }
    ...
}

```

3. Develop the code to generate a unique code for each fax. The code consists of three parts and is like 1/2004/HR. The first part is a sequence number that starts from 1. The next time it will be 2. The second part is the current year. The third part is the id of the department that issues the fax. Each department will have its own sequence number.

The most obvious problem here is that the code needs to find out the current year. To setup the context so that it generates the code 1/2004/HR, we seem to have to set the system clock first to year 2004, otherwise the test will break when it is run in say 2005. Setting the system clock is a nasty thing to do. A much better way is to use an interface to replace the system clock:

```

public interface YearGetter {
    public int getFullYear();
}

```

Then we can write the tests more easily. However, when writing the tests, we will find

that we are testing two separate behaviors: one is that the fax code consists of the sequence number, the current year and the department id; the other one is that the sequence number is incremented. So it is easier to test and implement them separately. Here are the tests and implementation code for the fax code behavior:

```
public interface NumberGenerator {
    public void setSeqNo(int seqNo);
    public int generate();
}

public class FaxCodeGeneratorTest extends TestCase {
    public void testCombineFormat() {
        FaxCodeGenerator generator = new FaxCodeGenerator("a");
        generator.setNumberGenerator(new NumberGenerator() {
            public int generate() {
                return 3;
            }
        });
        generator.setYearGetter(new YearGetter() {
            public int getCurrentYear() {
                return 2004;
            }
        });
        assertEquals(generator.generate(), "3/2004/a");
    }

    public void testInitialSeqNo() {
        final StringBuffer callLog = new StringBuffer();
        FaxCodeGenerator generator = new FaxCodeGenerator("a");
        generator.setNumberGenerator(new NumberGenerator() {
            public void setSeqNo(int seqNo) {
                assertEquals(seqNo, 1);
                callLog.append("x");
            }

            public int generate() {
                fail();
                return 0;
            }
        });
        assertEquals(callLog.toString(), "x");
    }
}

public class FaxCodeGenerator {
    private final static int INITIAL_SEQ_NO=1;
    private NumberGenerator numberGenerator;
    private String departId;
    private YearGetter yearGetter;

    public FaxCodeGenerator(String departId) {
        this.departId = departId;
    }

    public void setYearGetter(YearGetter getter) {
        this.yearGetter = getter;
    }

    public String generate() {
        return numberGenerator.generate()
            + "/"
            + yearGetter.getCurrentYear()
            + "/"
            + departId;
    }
}
```

```

public void setNumberGenerator(NumberGenerator generator) {
    this.numberGenerator = generator;
    this.numberGenerator.setSeqNo(INITIAL_SEQ_NO);
}
}

```

Here are the tests and implementation code for the sequence number behavior:

```

public class SeqNoGeneratorTest extends TestCase {
    public void testInitialValue() {
        SeqNoGenerator generator = new SeqNoGenerator(10);
        assertEquals(generator.generate(), 10);
    }
    public void testIncrementAfterGenerate() {
        SeqNoGenerator generator = new SeqNoGenerator(10);
        generator.generate();
        assertEquals(generator.generate(), 11);
    }
}
public class SeqNoGenerator implements NumberGenerator {
    private int seqNo;
    public SeqNoGenerator(int initialSeqNo) {
        this.seqNo = initialSeqNo;
    }
    public int generate() {
        return seqNo++;
    }
    public void setSeqNo(int seqNo) {
        this.seqNo = seqNo;
    }
}

```

Ultimately we have to use the system clock to implement the YearGetter interface. To make this code as thin as possible, we further divide this into two behaviors: to get the current time from the system clock and to get the year from a given time. The former cannot be tested but the latter can. The code for the latter is:

```

public class YearGetterFromCalendarTest extends TestCase {
    public void testGetCurrentYear() {
        GregorianCalendar calendar = new GregorianCalendar(2003, 0, 23);
        YearGetterFromCalendar getter = new YearGetterFromCalendar(calendar);
        assertEquals(getter.getCurrentYear(), 2003);
    }
}
public class YearGetterFromCalendar implements YearGetter {
    private GregorianCalendar calendar;
    public YearGetterFromCalendar(GregorianCalendar calendar) {
        this.calendar = calendar;
    }
    public int getCurrentYear() {
        return calendar.get(Calendar.YEAR);
    }
}

```

The FaxCodeGenerator should by default use this implementation:

```

public class FaxCodeGenerator {

```

```

public FaxCodeGenerator(String departId) {
    this.departId = departId;
    setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
}
...
}

```

It should also use the SeqNoGenerator as the NumberGenerator by default:

```

public class FaxCodeGenerator {
    public FaxCodeGenerator(String departId) {
        this.departId = departId;
        setYearGetter(new YearGetterFromCalendar(new GregorianCalendar()));
        setNumberGenerator(new SeqNoGenerator(INITIAL_SEQ_NO));
    }
    ...
}

```

4. Develop the code to check if two employees are equal. An employee has an employee id, a list of qualifications, a superior (also an employee). A qualification has a text description and indicates the year when the qualification was achieved.

The difficulty here is that we need to setup a lot of data (two employee objects) but most turn out to be unused. The real behavior in the Employee class here is that when determining whether it is equal to another Employee object, it checks if its three elements (id, qualification list, superior) are equal to their counterparts in that other Employee object. The actual contents of its id, qualification list and superior are totally unimportant to it.

The tests are:

```

public interface EqualityChecker {
    public boolean eachEquals(Object objList1[], Object objList2[]);
}

public class EmployeeTest extends TestCase {
    public void testEquals() {
        final StringBuffer callLog = new StringBuffer();
        final QualificationList qualiList1 = new QualificationList();
        final QualificationList qualiList2 = new QualificationList();
        final Employee superior1 = new Employee(null, null, null);
        final Employee superior2 = new Employee(null, null, null);
        Employee employee1 = new Employee("id1", qualiList1, superior1);
        Employee employee2 = new Employee("id2", qualiList2, superior2);
        employee1.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                callLog.append("x");
                assertEquals(objList1.length, 3);
                assertEquals(objList1[0], "id1");
                assertSame(objList1[1], qualiList1);
                //assertSame(X, Y) means assertTrue(X==Y). It is provided by JUnit.
                assertSame(objList1[2], superior1);
                assertEquals(objList2.length, 3);
                assertEquals(objList2[0], "id2");
                assertSame(objList2[1], qualiList2);
                assertSame(objList2[2], superior2);
            }
        });
    }
}

```

```

        return true;
    }
    });
    assertTrue(employee1.equals(employee2));
    assertEquals(callLog.toString(), "x");
}
public void testNonEmployee() {
    Employee employee = new Employee("id", null, null);
    employee.setEqualityChecker(new EqualityChecker() {
        public boolean eachEquals(Object[] objList1, Object[] objList2) {
            fail();
            return false;
        }
    });
    assertFalse(employee.equals("non-employee"));
    //assertFalse(X) means assertTrue(!X). It is provided by JUnit.
}
}

```

The implementation code is:

```

public class Employee {
    private String id;
    private QualificationList qualiList;
    private Employee superior;
    private EqualityChecker equalityChecker;

    public Employee(
        String id,
        QualificationList qualiList,
        Employee superior) {
        this.id = id;
        this.qualiList = qualiList;
        this.superior = superior;
    }
    public boolean equals(Object obj) {
        return obj instanceof Employee ? equals((Employee) obj) : false;
    }
    private boolean equals(Employee employee) {
        return equalityChecker.eachEquals(
            getElementsForEqualityCheck(),
            employee.getElementsForEqualityCheck());
    }
    private Object[] getElementsForEqualityCheck() {
        return new Object[] { id, qualiList, superior };
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
}

```

We will need to provide an implementation for EqualityChecker:

```

public class ShortCircuitEqualityCheckerTest extends TestCase {
    public void testOneElement() {
        ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
        assertTrue(
            checker.eachEquals(new Object[] { "a" }, new Object[] { "a" }));
    }
}

```

```

    assertFalse(
        checker.eachEquals(new Object[] { "a" }, new Object[] { "b" }));
}
public void testNotSameLength() {
    ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
    assertFalse(checker.eachEquals(new Object[] { "a" }, new Object[0]));
}
public void testFirstElementNotEqual() {
    ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
    assertFalse(
        checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "c", "b" }));
}
public void testFirstElementEqual() {
    ShortCircuitEqualityChecker checker = new ShortCircuitEqualityChecker();
    assertTrue(
        checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "a", "b" }));
    assertFalse(
        checker.eachEquals(
            new Object[] { "a", "b" },
            new Object[] { "a", "c" }));
}
}

```

The Employee class should by default use this implementation:

```

public class Employee {
    public Employee(
        String id,
        QualificationList qualiList,
        Employee superior) {
        ...
        equalityChecker = new ShortCircuitEqualityChecker();
    }
    ...
}

```

After implementing Employee, we also need to implement the equality check for the QualificationList class. It is very similar to Employee:

```

public class QualificationListTest extends TestCase {
    public void testEquals() {
        final StringBuffer callLog = new StringBuffer();
        QualificationList list1 = new QualificationList() {
            public List getQualifications() {
                List qualifications = new ArrayList();
                qualifications.add("a");
                qualifications.add("b");
                return qualifications;
            }
        };
        QualificationList list2 = new QualificationList() {
            public List getQualifications() {
                List qualifications = new ArrayList();
                qualifications.add("c");
            }
        };
    }
}

```

```

        qualifications.add("d");
        return qualifications;
    }
};
list1.setEqualityChecker(new EqualityChecker() {
    public boolean eachEquals(Object[] objList1, Object[] objList2) {
        callLog.append("x");
        assertEquals(objList1.length, 2);
        assertEquals(objList1[0], "a");
        assertEquals(objList1[1], "b");
        assertEquals(objList2.length, 2);
        assertEquals(objList2[0], "c");
        assertEquals(objList2[1], "d");
        return true;
    }
});
assertTrue(list1.equals(list2));
assertEquals(callLog.toString(), "x");
}
public void testNonQualificationList() {
    QualificationList list = new QualificationList();
    list.setEqualityChecker(new EqualityChecker() {
        public boolean eachEquals(Object[] objList1, Object[] objList2) {
            fail();
            return true;
        }
    });
    assertFalse(list.equals("non-qualification-list"));
}
}
public class QualificationList {
    private EqualityChecker equalityChecker;

    public QualificationList() {
        setEqualityChecker(new ShortCircuitEqualityChecker());
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
    public boolean equals(Object obj) {
        return obj instanceof QualificationList
            && equals((QualificationList) obj);
    }
    public List getQualifications() {
        return null; //TODO: Implement when required.
    }
    private boolean equals(QualificationList list) {
        return equalityChecker.eachEquals(
            getQualifications().toArray(),
            list.getQualifications().toArray());
    }
}
}

```

Next, we also need to implement the equality check for the Qualification class in a similar way:

```

public class QualificationTest extends TestCase {
    public void testEquals() {
        Qualification qualification1 = new Qualification("desc1", 2003);
    }
}

```

```

        Qualification qualification2 = new Qualification("desc2", 2004);
        qualification1.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                assertEquals(objList1.length, 2);
                assertEquals(objList1[0], "desc1");
                assertEquals(objList1[1], new Integer(2003));
                assertEquals(objList2.length, 2);
                assertEquals(objList2[0], "desc2");
                assertEquals(objList2[1], new Integer(2004));
                return true;
            }
        });
        assertTrue(qualification1.equals(qualification2));
    }
    public void testNotQualification() {
        Qualification qualification = new Qualification("desc", 2003);
        qualification.setEqualityChecker(new EqualityChecker() {
            public boolean eachEquals(Object[] objList1, Object[] objList2) {
                fail();
                return false;
            }
        });
        assertFalse(qualification.equals("non-qualification"));
    }
}
public class Qualification {
    private String desc;
    private int yearWhenAchieved;
    private EqualityChecker equalityChecker;

    public Qualification(String desc, int yearWhenAchieved) {
        this.desc = desc;
        this.yearWhenAchieved = yearWhenAchieved;
        setEqualityChecker(new ShortCircuitEqualityChecker());
    }
    public boolean equals(Object obj) {
        return (obj instanceof Qualification) && equals((Qualification) obj);
    }
    public boolean equals(Qualification qualification) {
        return equalityChecker.eachEquals(
            makeElementsForEqualityCheck(),
            qualification.makeElementsForEqualityCheck());
    }
    private Object[] makeElementsForEqualityCheck() {
        return new Object[] { desc, new Integer(yearWhenAchieved)};
    }
    public void setEqualityChecker(EqualityChecker checker) {
        this.equalityChecker = checker;
    }
}

```

5. Develop a `EnrollmentDBStorage` class implementing the `EnrollmentStorage` interface. It should be able to write an `Enrollment` object into a database. You only need to handle cash payments only. You can use the schema shown in the text or make up your own. You can assume the name of the database, the driver used and etc.

```

public class EnrollmentDBStorageTest extends TestCase {
    private Connection conn;

```

```

protected void setUp() throws Exception {
    Class.forName("org.postgresql.Driver");
    conn =
        DriverManager.getConnection(
            "jdbc:postgresql://localhost/testdb",
            "testuser",
            "testpassword");
    deleteAllEnrollments();
}
protected void tearDown() throws Exception {
    conn.close();
}
private void deleteAllEnrollments() throws SQLException {
    PreparedStatement st =
        conn.prepareStatement("delete from enrollments");
    try {
        st.executeUpdate();
    } finally {
        st.close();
    }
}
public void testAdd() throws SQLException {
    EnrollmentDBStorage storage = new EnrollmentDBStorage(conn);
    Date enrolDate = new GregorianCalendar(2004, 0, 28).getTime();
    Payment payment = new CashPayment(200);
    Enrollment enrollment =
        new Enrollment("s001", "c001", enrolDate, payment);
    storage.add(enrollment);
    PreparedStatement st =
        conn.prepareStatement(
            "select * from enrollments where studentId=? and courseCode=?");
    try {
        st.setString(1, "s001");
        st.setString(2, "c001");
        ResultSet rs = st.executeQuery();
        assertTrue(rs.next());
        assertEquals(rs.getDate("enrolDate"), enrolDate);
        assertEquals(rs.getInt("amount"), 200);
        assertEquals(rs.getString("paymentType"), "Cash");
        assertNull(rs.getObject("cardNo"));
        //assertNull(X) means assertTrue(X == null). It is provided by JUnit.
        assertNull(rs.getObject("expiryDate"));
        assertNull(rs.getObject("nameOnCard"));
        assertFalse(rs.next());
    } finally {
        st.close();
    }
}
}
public class EnrollmentDBStorage implements EnrollmentStorage {
    private Connection conn;

    public EnrollmentDBStorage(Connection conn) {
        this.conn = conn;
    }
    public void add(Enrollment enrollment) {
        try {
            PreparedStatement st =
                conn.prepareStatement(

```

```

        "insert into enrollments values(?,?,?,?,?,?,?,?)");
    try {
        st.setString(1, enrollment.getCourseCode());
        st.setString(2, enrollment.getStudentId());
        st.setDate(
            3,
            new java.sql.Date(enrollment.getEnrolDate().getTime()));
        CashPayment payment = (CashPayment) enrollment.getPayment();
        st.setInt(4, payment.getAmount());
        st.setString(5, "Cash");
        st.setNull(6, Types.VARCHAR);
        st.setNull(7, Types.DATE);
        st.setNull(8, Types.VARCHAR);
        st.executeUpdate();
    } finally {
        st.close();
    }
} catch (SQLException e) {
    throw new RuntimeException(e);
}
}
}

```

6. Develop the code to find the number of seats taken for a given course (due to enrollments in it or in its parent). As part of it, you should have a class implementing the EnrollmentCounter shown in the text. When you need to access the database, use an interface instead.

The tests are:

```

public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}

public interface EnrollmentSource {
    public List getEnrollmentsFor(String courseCode);
}

public class EnrollmentCounterFromSrcTest extends TestCase {
    public void testConsiderAncestor() {
        EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc() {
            public int getNoEnollmentsForSingleCourse(String courseCode) {
                if (courseCode.equals("c001")) {
                    return 4;
                }
                if (courseCode.equals("c002")) {
                    return 2;
                }
                if (courseCode.equals("c003")) {
                    return 1;
                }
                fail();
                return 0;
            }
        };
        counter.setCourseParentGetter(new CourseParentGetter() {
            public String getParentCode(String courseCode) {
                if (courseCode.equals("c003")) {
                    return "c002";
                }
            }
        });
    }
}

```

```

        if (courseCode.equals("c002")) {
            return "c001";
        }
        if (courseCode.equals("c001")) {
            return null;
        }
        fail();
        return null;
    }
});
assertEquals(counter.getNoSeatsTaken("c003"), 7);
}
public void testSingleCourse() {
    EnrollmentCounterFromSrc counter = new EnrollmentCounterFromSrc();
    counter.setEnrollmentSource(new EnrollmentSource() {
        public List getEnrollmentsFor(String courseCode) {
            List enrollments = new ArrayList();
            enrollments.add("e1");
            enrollments.add("e2");
            return enrollments;
        }
    });
    assertEquals(counter.getNoEnrollmentsForSingleCourse("c001"), 2);
}
}

```

The implementation code is:

```

public class EnrollmentCounterFromSrc implements EnrollmentCounter {
    private EnrollmentSource enrollmentSource;
    private CourseParentGetter courseParentGetter;

    public int getNoSeatsTaken(String courseCode) {
        int noSeatsTaken = 0;
        for (;;) {
            noSeatsTaken += getNoEnrollmentsForSingleCourse(courseCode);
            courseCode = courseParentGetter.getParentCode(courseCode);
            if (courseCode == null) {
                return noSeatsTaken;
            }
        }
    }

    public int getNoEnrollmentsForSingleCourse(String courseCode) {
        return enrollmentSource.getEnrollmentsFor(courseCode).size();
    }

    public void setCourseParentGetter(CourseParentGetter getter) {
        this.courseParentGetter = getter;
    }

    public void setEnrollmentSource(EnrollmentSource source) {
        this.enrollmentSource = source;
    }
}

```

We also need to provide an implementation for the CourseParentGetter that gets its information from a CourseSource:

```

public interface CourseSource {
    public Course getCourse(String courseCode);
}

```

```

}
public class CourseParentGetterFromSrcTest extends TestCase {
    public void testHasParent() {
        CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
        getter.setCourseSource(new CourseSource() {
            public Course getCourse(String courseCode) {
                Course c002 = new Course() {
                    public String getParentCode() {
                        return "c001";
                    }
                };
                return courseCode.equals("c002") ? c002 : null;
            }
        });
        assertEquals(getter.getParentCode("c002"), "c001");
    }
    public void testHasNoParent() {
        CourseParentGetterFromSrc getter = new CourseParentGetterFromSrc();
        getter.setCourseSource(new CourseSource() {
            public Course getCourse(String courseCode) {
                Course c002 = new Course() {
                    public String getParentCode() {
                        return null;
                    }
                };
                return courseCode.equals("c002") ? c002 : null;
            }
        });
        assertNull(getter.getParentCode("c002"));
    }
}

```

The implementation code is:

```

public class CourseParentGetterFromSrc implements CourseParentGetter {
    private CourseSource courseSource;

    public String getParentCode(String courseCode) {
        return courseSource.getCourse(courseCode).getParentCode();
    }
    public void setCourseSource(CourseSource source) {
        this.courseSource = source;
    }
}

```

The `CourseParentGetterFromSrc` should by default use an implementation for `CourseSource` that gets the information from the database:

```

public class CourseParentGetterFromSrc implements CourseParentGetter {
    public CourseParentGetterFromSrc() {
        setCourseSource(new CourseDBSource());
    }
    ...
}

```

The `EnrollmentCounterFromSrc` should by default use the implementations for the `EnrollmentSource` and `CourseParentGetter` interfaces that get the information from the

database:

```
public class EnrollmentCounterFromSrc implements EnrollmentCounter {
    public EnrollmentCounterFromSrc() {
        setEnrollmentSource(new EnrollmentDBSource());
        setCourseParentGetter(new CourseParentGetterFromSrc());
    }
    ...
}
```

- Develop the code to find the number of seats reserved for a given course (due to reservations for it or its parent). As part of it, you should have a class implementing the ReservationCounter shown in the text. When you need to access the database, use an interface instead. You should only count the reservations that are still active.

When we are working on this, we will find that the tests are very similar to those for the previous problem. It suggests that they share some common logic: They are both trying to calculate the sum of an integer for a course, for its parent, for its grand-parent and etc. So, we extract that common logic and test it in isolation:

```
public interface CourseIntGetter {
    public int getInt(String courseCode);
}

public interface CourseParentGetter {
    public String getParentCode(String courseCode);
}

public class CourseAncestorsTraverserTest extends TestCase {
    public void testSum() {
        CourseAncestorsTraverser traverser = new CourseAncestorsTraverser();
        traverser.setCourseParentGetter(new CourseParentGetter() {
            public String getParentCode(String courseCode) {
                if (courseCode.equals("c003")) {
                    return "c002";
                }
                if (courseCode.equals("c002")) {
                    return "c001";
                }
                if (courseCode.equals("c001")) {
                    return null;
                }
                fail();
                return null;
            }
        });
        CourseIntGetter intGetter = new CourseIntGetter() {
            public int getInt(String courseCode) {
                if (courseCode.equals("c001")) {
                    return 1;
                }
                if (courseCode.equals("c002")) {
                    return 2;
                }
                if (courseCode.equals("c003")) {
                    return 5;
                }
                fail();
            }
        };
    }
}
```

```

        return 0;
    }
};
assertEquals(traverser.sum("c003", intGetter), 8);
}
}

```

The implementation code is:

```

public class CourseAncestorsTraverser {
    private CourseParentGetter courseParentGetter;

    public void setCourseParentGetter(CourseParentGetter getter) {
        this.courseParentGetter = getter;
    }
    public int sum(String courseCode, CourseIntGetter intGetter) {
        int sum = 0;
        for (;;) {
            sum += intGetter.getInt(courseCode);
            courseCode = courseParentGetter.getParentCode(courseCode);
            if (courseCode == null) {
                return sum;
            }
        }
    }
}

```

The `CourseAncestorsTraverser` should by default use an implementation for `CourseParentGetter` that gets the information from the database:

```

public class CourseParentGetterFromSrc implements CourseParentGetter {
    ...
}
public class CourseAncestorsTraverser {
    public CourseAncestorsTraverser() {
        setCourseParentGetter(new CourseParentGetterFromSrc());
    }
    ...
}

```

To get the number of enrollments and number of reservations, we just need to provide a different `CourseIntGetter` implementation for each. For enrollments:

```

public class CourseNoEnrollmentsGetterTest extends TestCase {
    public void testCount() {
        CourseNoEnrollmentsGetter getter = new CourseNoEnrollmentsGetter();
        getter.setEnrollmentSource(new EnrollmentSource() {
            public List getEnrollmentsFor(String courseCode) {
                if (courseCode.equals("c001")) {
                    List enrollments = new ArrayList();
                    enrollments.add("e1");
                    enrollments.add("e2");
                    return enrollments;
                }
            }
        });
        fail();
        return null;
    }
}

```

```

    });
    assertEquals(getter.getInt("c001"), 2);
  }
}
public class CourseNoEnrollmentsGetter implements CourseIntGetter {
    private EnrollmentSource enrollmentSource;

    public void setEnrollmentSource(EnrollmentSource source) {
        this.enrollmentSource = source;
    }
    public int getInt(String courseCode) {
        return enrollmentSource.getEnrollmentsFor(courseCode).size();
    }
}
}

```

The `CourseNoEnrollmentsGetter` should by default use an implementation for `EnrollmentSource` that gets the information from the database:

```

public class CourseNoEnrollmentsGetter implements CourseIntGetter {
    public CourseNoEnrollmentsGetter() {
        setEnrollmentSource(new EnrollmentDBSource());
    }
    ...
}

```

Finally the `CourseAncestorsTraverser` can act as an `EnrollmentCounter`:

```

public class CourseAncestorsTraverser implements EnrollmentCounter {
    ...
    public int getNoSeatsTaken(String courseCode) {
        return sum(courseCode, new CourseNoEnrollmentsGetter());
    }
}

```

For reservations, it is similar:

```

public class CourseNoReservationsGetterTest extends TestCase {
    public void testCountOnlyActive() {
        CourseNoReservationsGetter getter = new CourseNoReservationsGetter();
        getter.setReservationSource(new ReservationSource() {
            public List getReservationsFor(String courseCode) {
                if (courseCode.equals("c001")) {
                    List reservations = new ArrayList();
                    reservations.add(new Reservation() {
                        public boolean isActive() {
                            return true;
                        }
                    });
                    reservations.add(new Reservation() {
                        public boolean isActive() {
                            return false;
                        }
                    });
                    reservations.add(new Reservation() {
                        public boolean isActive() {
                            return true;
                        }
                    });
                }
            }
        });
    }
}

```

```

        });
        return reservations;
    }
    fail();
    return null;
}
});
assertEquals(getter.getInt("c001"), 2);
}
}

public class CourseNoReservationsGetter implements CourseIntGetter {
    private ReservationSource reservationSource;

    public CourseNoReservationsGetter() {
        setReservationSource(new ReservationDBSource());
    }

    public int getInt(String courseCode) {
        int result = 0;
        List reservations = reservationSource.getReservationsFor(courseCode);
        for (Iterator iter = reservations.iterator(); iter.hasNext();) {
            Reservation reservation = (Reservation) iter.next();
            if (reservation.isActive()) {
                result++;
            }
        }
        return result;
    }

    public void setReservationSource(ReservationSource source) {
        this.reservationSource = source;
    }
}

public class CourseAncestorsTraverser
implements EnrollmentCounter, ReservationCounter {
    ...
    public int getNoSeatsReserved(String courseCode) {
        return sum(courseCode, new CourseNoReservationsGetter());
    }
}

```

The Reservation class needs to check if it's active or not. To test it, it is easier to use an interface to replace the system clock:

```

public interface Clock {
    public Date getCurrentDate();
}

public class ReservationTest extends TestCase {
    public void testActive() {
        Reservation reservation = new Reservation();
        reservation.setClock(new Clock() {
            public Date getCurrentDate() {
                return new GregorianCalendar(2004, 0, 22).getTime();
            }
        });
        reservation.setReserveDate(
            new GregorianCalendar(2004, 0, 20).getTime());
        reservation.setDaysReserved(3);
        assertTrue(reservation.isActive());
    }

    public void testInactive() {

```

```

Reservation reservation = new Reservation();
reservation.setClock(new Clock() {
    public Date getCurrentDate() {
        return new GregorianCalendar(2004, 0, 22).getTime();
    }
});
reservation.setReserveDate(
    new GregorianCalendar(2004, 0, 20).getTime());
reservation.setDaysReserved(2);
assertFalse(reservation.isActive());
}
}

public class Reservation {
    private Clock clock;
    private Date reserveDate;
    private int daysReserved;

    public boolean isActive() {
        GregorianCalendar lastEffectiveDate = new GregorianCalendar();
        lastEffectiveDate.setTime(reserveDate);
        lastEffectiveDate.add(Calendar.DAY_OF_MONTH, daysReserved - 1);
        return !clock.getCurrentDate().after(lastEffectiveDate.getTime());
    }
    public void setDaysReserved(int daysReserved) {
        this.daysReserved = daysReserved;
    }
    public void setReserveDate(Date reserveDate) {
        this.reserveDate = reserveDate;
    }
    public void setClock(Clock clock) {
        this.clock = clock;
    }
}
}

```

The Reservation class should by default use an implementation for Clock that gets the data from the system clock:

```

public class ClockInJava implements Clock {
    public Date getCurrentDate() {
        return new Date();
    }
}

public class Reservation {
    public Reservation() {
        setClock(new ClockInJava());
    }
    ...
}

```