Print your name and netid neatly in the space provided below; print your netid in the upper right corner of every page.

Name: ___________________________
Netid: _________________________

This is a closed book, closed notes examination. You may not use calculators or any other electronic devices. Any sort of cheating on the examination will result in a zero grade.

We cannot give any clarifications about the exam questions during the test. If you are unsure of the meaning of a specific question, write down your assumptions and proceed to answer the question on that basis.

Do all the problems in this booklet. Do your work inside this booklet, using the backs of pages if needed. The problems are of varying degrees of difficulty so please pace yourself carefully, and answer the questions in the order which best suits you. Answers to essay-type questions should be as brief as possible. If the grader cannot understand your handwriting you will get 0 points.

There are 10 questions on this exam and the maximum grade on this exam is $80 + 1$ (bonus) point.

<table>
<thead>
<tr>
<th>Page</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>
1. Testing

(a) Define and compare the terms fault and failure in the context of software testing.

**Solution:**

1. fault: A requirements, design or implementation flaw or deviation from a desired or intended state. Not just a syntactical bug in code.
2. failure: The inability of a system to perform its required functions within specified performance requirement. Not just a system “crashing”. A test returning false on asserts.
3. many failures can be caused by a fault.
4. failures help localize or find faults.

(b) Cyclomatic complexity is a software metric that gives a numerical value for complexity. How does it measure complexity? Explain in words without quoting the formula.

**Solution:**

The number of independent paths through the procedure, gives an upper bound on the number of tests necessary to execute every edge of a control graph.

(c) Number of tests can also be used as a software metric — does a large number of test directly translate to a better quality software? Why or why not? Give two reasons to support your answer.

**Solution:**

1. coverage
2. quality of tests
3. tests do not prove the absence of bugs/correctness of the system

(d) What are two advantages of writing test cases early in the software life cycle? What could be one potential disadvantage of writing tests early?

**Solution:**

Advantages

1. Clear understanding of requirements at hand
2. Faster feedback
3. Simpler/better quality design or code
4. EASIER refactoring
<table>
<thead>
<tr>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Less flexible</td>
</tr>
<tr>
<td>2. Design changes later in the process may render tests unusable</td>
</tr>
<tr>
<td>3. May not fully understand domain model</td>
</tr>
<tr>
<td>4. No codes to test in the beginning</td>
</tr>
</tbody>
</table>
(e) Given the following Java code fragment, what is the minimum number of test cases needed for structured basis testing as described in Code Complete 2? Explain your reasoning.

```java
public void put(URI uri, Map<String, List<String>> responseHeaders)
    throws IOException {
    List<String> setCookieList = responseHeaders.get("Set-Cookie");
    if (setCookieList != null) {
        for (String item : setCookieList) {
            Cookie cookie = new Cookie(uri, item);
            for (Cookie existingCookie : cookieJar) {
                if ((cookie.getURI().equals(existingCookie.getURI()))
                    && (cookie.getName().equals(existingCookie.getName()))) {
                    cookieJar.remove(existingCookie);
                    break;
                }
            }
            cookieJar.add(cookie);
        }
    }
}
```

**Solution:**
6 test cases

(f) What are regression errors? What testing practice is recommended to prevent regression errors?

**Solution:**
Re-emerging or new errors caused by improvements/addition of features. The improvements or changes may not directly change the codes/module that fails. “Program gets worse” is not a good enough answer. Test with automated regression testing, or unit tests everytime a new feature is added.

(g) Which type of changes tends to be more error-prone: small changes or large changes? Why?

**Solution:**
“More things/chances to go wrong” is not a good enough answer. Integration problems, parallel changes, non-localized planning, etc.

2. Development Models

In CS427, you have experienced the XP model for some of your MPs and the project, which is a well-known iterative software development process. There is also the waterfall model. First, briefly describe the waterfall model. Then, for each software development model, give one of its strengths and one of its weaknesses.
**Solution:**
The waterfall model is a sequential development process that follows the following phases: Requirement Specification - Design - Implementation - Testing/Verification - Installation - Maintenance.

**Waterfall:**
Strength: Simple, more disciplined, concrete/complete requirements before implementation - making sure that requirements and design are correct in the earlier phases saves time and effort in later phases.
Weakness: It’s nearly impossible/impractical for any non-trivial project to have one phase of its development lifecycle be perfected before proceeding to the next phase. Some details/limitations cannot be known before completion.

**XP**
Strength: More flexible/practical.
Weakness: Unstable requirements/Pair programming can be an expensive activity/User conflicts

3. User Stories
John is a developer in an XP team. He is responsible for helping the client write user stories. Here are a list of questions that he has prepared to ask the client. We believe that he hasn’t phrased the questions well. What is the overall problem with his questions?

1. Do you want your web application to be AJAX enabled?
2. How much data per user needs to be kept in a session?
3. How do you want to have your data persisted? On a database or using a third-party web service?

**Solution:**
The user stories should be described in the language that the client understands. So instead of asking the client: “How do you want to have your data persisted?” it’s better to ask him/her: “Do you want to be able to save your data? That is, do you want to re-enter your data each time you start a new session or you want the system to remember what you have entered before.”

4. Pair Programming
A manager claims that his team is practicing XP but not pair programming. He believes that pair programming is not a good use of developer time. Explain to him why he is wrong.
Solution:

**Real-Time Code Reviews** A second person makes the mistakes easier to spot.

**Avoiding Distractions** When pairing, you have a responsibility to the time and energy your partner is putting. So, you do your best to focus on your work.

**Managing for Two** When two programmers sit down to write some code, it’s usual that they start questioning some of each others’ practices. And, this questioning makes them discuss and learn those practices better.

**Knowledge and Information Migration** In the long run, people pick up their partners’ good programming habits by witnessing them for a while.

---

5. Planning Game

About a week into the iteration you find out that your estimates weren’t accurate. You have underestimated the time it takes to do the tasks. Mark the things that you think are reasonable to do in such a situation and explain your decisions.

1. I’ll meet with the group and reassess where we are.
2. I’ll spend as much time as I had estimated for the task and discuss the issue at the next iteration planning meeting.
3. I’ll let the client know about it immediately.
4. I’ll work faster by postponing some practices such as writing unit tests.

Solution:

You should make sure the client always has an accurate picture of the reality at all times. So, reasonable actions are (1), (2), and (3). Note that an incomplete user story shouldn’t make you extend the iteration. You should always declare the end of your iteration no matter how successfully you have done your planned tasks.
6. Software Configuration Management

(a) Two techniques of dealing with concurrent development in a version control system (e.g. CVS and Subversion) are merging and locking. Give one advantage for each technique.

Solution:
The advantage of merging is that all developers can contribute concurrently on the same artifact. The benefit of locking is that there is no need to resolve possible conflicts as no two developers can modify the same object.

(b) How does XP address change control? Who is the change control authority?

Solution:
The change control authority in XP is the customer and XP handles requested changes by adding user stories.

7. Reverse Engineering

(a) One of the reasons for reverse engineering a system is for maintenance especially if the original team that developed the system is no longer available. Give two other reasons why someone would be interested in reverse engineering a system.

Solution:
teaching the system to new developers, writing documentation
(b) Two of the reverse engineering patterns that we discussed in class and in MP3 are *Read all the Code in One Hour* and *Refactor To Understand*. For each pattern, provide one advantage for applying it.

**Solution:**
The *Read all the Code in One Hour* pattern lets you get an overall idea of the code without getting involved into the details of the system. The *Refactor To Understand* pattern helps one understand the code better by making the code cleaner. There is no better way to learn a system than playing with it and modifying it.
8. Metrics

A study by NASA measures the following two Chidamber & Kemerer metrics:

- Weighted Methods Per Class (WMC): number of methods defined in class
- Coupling between Object Classes (CBO): number of classes to which a class is coupled

The table below summarizes their report of three analyzed systems.

<table>
<thead>
<tr>
<th>System analyzed</th>
<th>Java</th>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>46</td>
<td>1000</td>
<td>1617</td>
</tr>
<tr>
<td>Lines</td>
<td>50,000</td>
<td>300,000</td>
<td>500,000</td>
</tr>
<tr>
<td>Quality</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>CBO</td>
<td>2.48</td>
<td>1.25</td>
<td>2.09</td>
</tr>
<tr>
<td>WMC</td>
<td>45.7</td>
<td>11.10</td>
<td>23.97</td>
</tr>
</tbody>
</table>

Adam is given a fourth system for which we have $CBO = 1.5$ and $WMC = 50.2$. He predicts that this new system must have a lower quality than the three analyzed systems. His reasoning is the following:

“In the three analyzed programs, higher values of $CBO$ and $WMC$ belong to projects of lower quality. So, I would say this new system that has higher values for these two metrics must have a lower quality than the three analyzed systems.”

What are two threats to the validity of his conclusion?

Solution:

1. More systems should be studied in order to be able to predict quality of code based on its metrics.
2. Studied systems have been developed in different languages which makes it hard to generalize the prediction.
3. The relation between the quality of the system and its metrics might depend on the size of the system. So, these metrics should be studied for several systems of variety of sizes. One cannot generalize the result for large systems to small systems.
9. Test Driven Development

A **Range** is an object with two integer values: a lower bound and an upper bound. Every integer between (and including) the lower and upper bounds is considered to be part of the range.

**User Story #1: Creating a Range**

You create a **Range** object by calling `new Range(LOWER_BOUND, UPPER_BOUND)` where LOWER_BOUND and UPPER_BOUND are both integers. If LOWER_BOUND > UPPER_BOUND, you should throw an **IllegalArgumentException**.

**User Story #2: Union of two Ranges**

You find the union of two **Ranges** by calling the `union(Range otherRange)` method. Calling `myRange1.union(myRange2)` returns a new **Range** object that represents a range between LOWER_BOUND = `minimum(myRange1, myRange2)` and UPPER_BOUND = `maximum(myRange1, myRange2)`.

Write two cases — in the JUnit 3 or 4 style — for each user story above i.e. you are going to have four test cases in total. Ensure that your tests cases exercise different scenarios for each user story. You may use the next page for your answers.

### Solution:

The purpose of this question is to test your understanding of writing automated tests in JUnit. So we were looking out for:

- Methods signatures of the form `public void test...` or `@Test`.
- Usage of the `assert*(expected, actual)` methods
- Other indications that demonstrate that you are familiar with writing JUnit tests.

For **Scenario #1**, we were looking for:

- Testing the creation of a valid **Range** object. You need to check that both the LOWER_BOUND and UPPER_BOUNDS have been set correctly.
- Testing the creation of an invalid **Range** object. In this case, you need to catch the exception that is thrown.

For **Scenario #2**, we were looking for (any two of these):

- Testing the union of two disjoint **Range** objects i.e. the ranges do not intersect.
- Testing the union of two **Range** objects with a proper intersection.
- Testing the union of a **Range** object with itself.

If you lost points on this question, you are encouraged to actually implement those test cases in Eclipse so that you can get a better sense of writing JUnit tests.
Extra space for question 9...
10. Code Smells and Refactorings

Now that you are taking CS427 and have read the *Refactoring* and *Code Complete 2* books, you have also developed an uncanny intolerance for code smells. Just to test out your new refactoring skills, you decide to look at some old code that you had written as a freshman. This is what you see . . .

```java
public void printLibraryInformation() {
    // print title of the library
    System.out.println("Start information for: " + title);

    // iterate through the list of books
    // a serves as the index
    for (int a = 0; a < books.size(); a++) {
        // each book contains an array of strings that holds its information
        String[] book = books.get(a);

        // Each index in the book array has a special meaning
        System.out.println("Book");
        System.out.println("---");
        System.out.println("Title: " + book[0]);
        System.out.println("Author: " + book[1]);
        System.out.println("Publication date:" + book[2]);

        if (book.length > 3)
            System.out.println("Price:" + book[3]);

        if (book.length > 4)
            System.out.println("Rating:" + book[4]);
    }
}
```

Listing 1: MP3 from CS 1xx freshman year

(a) Your refactoring senses go off the charts. You quickly identify a few significant code smells in the code above. Give three of those smells. Identify them in the listing above. It is acceptable to circle and use arrows to point to the offending code segments. For each code smell that you identify, suggest how you would refactor it to remove the offensive code smell. You do not need to implement the actual refactoring. Use the space on the next page for your suggestions.

(b) Tell us the story of how the term “code smell” was created. (*Hint: It was mentioned in the Refactoring book).*


Solution:

Here are the main code smells that we were looking for:

- **Primitive Obsession** — The String[] book array should be an actual class. Use the Extract Class refactoring to move it into a new class called Book.

- **Feature Envy/Inappropriate Intimacy** — Once we have moved the String[] book array into its class, the operation for printing its information should also be moved into the Book instead of leaving it in the printLibraryInformation method.

- **Magic Numbers** — The magic numbers 3 and 4 don’t really communicate the intent of the code. Moreover if we have already extracted the Book class, then we don’t need to use those magic numbers anymore. We will be using proper instance variables instead.

- **Comments** — The block comment for the printLibraryInformation method doesn’t actually tell us a whole lot. In fact, it duplicates a lot of the information that can be gleaned just from looking at the method signature. The other line comments don’t really add much value either.

- **Uncommunicative Variable Name** — The loop index variable a doesn’t really convey its purpose. In this case, it might be better to refactor to use a foreach loop construct. Or give the index variable a more communicative name.

**Bonus question** It was based on a comment from Kent Beck’s grandma about changing the baby’s diaper.