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, then 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 may get 0 points.

There are 17 questions on this exam and the maximum grade on this exam is 90 points.

<table>
<thead>
<tr>
<th>Page</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>
1. **Golden Rule of Version Control software**

   (a) According to Prof. Johnson, the *golden rule* of version control is to check-in files quickly. Give **two** reasons why this is true.

   **Solution:**
   Refer to Lecture 2, slide 24
   - The longer you hold on to your changes, the more you interfere with others since you might have critical changes that will change the design of the system and this should be propagated to the mainline quickly so that everyone else will use this new design.
   - The longer you have code checked out the harder it is to merge (for you and other developers) since multiple changes might have occurred.

   (b) Does this rule apply to editing the wiki too? If so, describe how. If not, explain why not.

   **Solution:**
   Yes. The longer a person delays committing his/her changes, the more likely that other people would make changes to the same page on the wiki. In the end, the person who holds the page longest without committing would have the hardest task of merging all the commits that have occurred.

2. **Branch mania** refers to the situation where branches are created often and for no apparent reason. It is an *anti-pattern* and should be avoided. However, branching, when done properly, can also be beneficial.

   (a) What are two benefits of branching?

   **Solution:**
   - Branching *increases productivity* by allowing parallel development.
   - Branching helps *reduce risk* by isolating experimental changes from breaking the mainline.

   (b) Give **two** situations when a branch should be created.

   **Solution:**
   - To try out a new experimental version using new tools/ technology.
   - To add a new component/ task that might interfere with the mainline.
   - To split a separate release that no longer meets the vision and goals of the original developer.
1. (c) What is the main cost of branching?

**Solution:**
Merging. You would eventually have to merge some branches back into the mainline and depending on how much that branch has deviated the merge process could be very complicated.

2. According to Brad Appleton, before committing to the repository, you should always do a *Private System Build* to avoid breaking the build. List **two** things that you need to run during a Private System Build to ensure that your code will not break the build.

**Solution:**
Refer to Lecture 3, slide 37
- Run tests (smoke tests, unit tests, regression tests)
- Run build system (ant, make).
- Run integrated system like cruise control.

3. One of the best practices of XP is *test first*. In fact, the authors of *Extreme Software Engineering* claim that testing first helps you write better code.

   (a) Give **two** reasons for their claim.

**Solution:**
Refer to *Extreme Software Engineering* p. 57 - 61
- Testing forces simplicity
- Testing clarifies the task at hand
- Testing frees you from on-the-fly editing

(b) Must you write a test for *every* method or function? Explain your answer.

**Solution:**
No, you are not required to write a test for every method. You must write explicit test to ensure that every aspect of the requirements are satisfied. (i.e. black box tests). It is a best practice to write (white box) tests such that as much up of the code is covered as is possible. However, this does not mean that a test for every method is required. Some methods (such as accessors) are so trivial they do no necessarily require their own dedicated tests.
5. Automated testing

(a) Give one reason why *automated* tests are important in XP.

**Solution:**
You can easily run automated tests to check if your refactoring has broken some functionality. This fast feedback – compared to manual tests – allows you to make changes more confidently and quickly.

(b) Give two features that an automated testing framework like JUnit offers to the developer to make testing more *effective*.

**Solution:**
Any reasonable feature is acceptable.

- Visual feedback (red/green bar) that lets you know quickly if something is broken.
- Ability to identify and jump quickly to the source of error.
- Tests are decoupled from your classes – you can easily release a product without including the tests since the classes make no reference to those tests.

(c) Why does Brian Marrick says that attempting to automate all tests is a bad idea.

**Solution:**
Some tests will be only be run infrequently such that benefit gained does not commensurate with the cost of automation.

6. List three types of tools for testing that we have mentioned in class. (*Hint: We used two of them in the MPs*)

**Solution:**

- Unit Testing Framework – JUnit
- Code Coverage Tool – EclEmma
- Functional GUI Testing – FEST
7. Name **two** tools that you could use for debugging.

**Solution:**

- Debugger
- Logging system e.g. `log4j`

8. Configuration Management for projects

(a) We mentioned **three** kinds of tools for configuration management. What are they?

**Solution:**

- Version Control
- Build System
- Change Control

(b) Eclipse does **two** of these things automatically for Java programs. What are they?

**Solution:**

- CVS
- Build System
- Mylyn (task list) in Eclipse 3.3

9. Companies sometimes distribute beta programs over the internet as a way to test their software. According to Brian Marrick, an **over-reliance** on beta testing is a classical testing mistake. Give **two** reasons for his claim.

**Solution:**

Refer to the **over-reliance on beta testing** section in *Classical Testing Mistakes*

- Beta users are not committed to testing it thoroughly.
- Beta users might not report all problems (even serious ones).
- Beta users are not a good representation of the majority of end-users who will be using the product since they have different priorities.
10. *Pair programming* is another best practice principles of XP. What are **two advantages** of pair programming? What is **one disadvantage** of pair programming?

**Solution:**

**Advantages**
- Constant code reviews. Errors are caught more quickly - two sets of eye on the code.
- Each partner learns and teaches more.
- Mutual code ownership enables a culture of code sharing.

**Disadvantages**
- Required both people to be available at once.
- More expensive to allocation two programmers per machine (customer my not like it).

11. Code coverage

Listing 1: Branch coverage

```java
if(x > 5) {
    print('a');
    if(x == 8) {
        print('b');
    }
    if(x < 10) {
        print('c');
    } else {
        print('d');
    }
} else {
    print('e');
}
```

(a) Analyze the code fragment in Listing 1. Then, fill in the following table with the possible combinations of output (via the `print(...)` call) and one possible value of `x` that will cause it to happen. The first row has been filled in for you as an example.

<table>
<thead>
<tr>
<th>Value of x</th>
<th>Printed output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>abce</td>
</tr>
<tr>
<td>5 &lt; x &lt; 10; x ≠ 8</td>
<td>ace</td>
</tr>
<tr>
<td>x &gt; 10</td>
<td>ade</td>
</tr>
</tbody>
</table>
12. How would using a metrics tool like the one that we used in MP5 help you reverse engineer a system? Pick one of the reverse engineering patterns that we cover in class and describe how using a metrics tool would help.

**Solution:**
Refer to the relevant sections in *A Pattern Language for Reverse Engineering*

- **Read all the code in one hour** – To help focus on the important parts can use metrics to identify *suspicious* code that score above the tolerated metrics threshold.
- **Identify the Largest** — The metrics tool reports the size of each entity and from this you can infer the larger ones and concentrate your efforts on those.

13. *Software metrics are more useful as indicators of what’s wrong rather than what’s right about a system.* Explain.

**Solution:**
Software metrics should be used to determine which parts of your program might require attention and fixing. If a part of the system scores a measurement beyond the recommended threshold then you should investigate why that is and whether it needs to be fixed. On the other hand, just because some part of the system is within the threshold of tolerance does not mean that is right. It might have hidden problems that the metrics suite cannot detect since metrics are calculated based on heuristics.

14. Depth of Inheritance Tree (DofIT)

(a) *Every class that you create in Java starts with a DofIT of 1.* Answer true or false and explain your answer.

**Solution:**
True. Every class in Java automatically inherits from *Object.* Since the interpretation of the phrase *starts with a DofIT of 1* might vary, other answers with proper explanations were accepted as well.

(b) Give a reason why is it desirable to keep the DofIT of a class low.

**Solution:**
The higher the DofIT of a class, the more methods it inherits. This makes it more complex to reason about its behavior.
2 (c) What is an example of a refactoring that could *increase* the DoIIT of an existing class?

**Solution:**
Extract superclass (or a sentence describing the same procedure that this refactoring does).

15. Kent Beck’s principles of simplest design

1. The system should have the fewest possible methods
2. The system (code and tests) must communicate everything you want to communicate
3. The system should have the fewest possible classes
4. The system must contain no duplicate code

2 (a) Order the principles listed above in order of importance – from most important to least important

(a) __________

**Solution:**
2, 4, 3, 1

4 (b) Explain your decision for your choice of the most important principle and your choice for the least important i.e. why did you list the principles in that order?

**Solution:**
2 as first: If the software doesn’t communicate what it doesn’t it is hard to know how it works, and whether or not it works correctly.
1 as last: While it is important to keep the code simple and minimize the number of methods, it is far more important to add methods whenever necessary to increase the clarity and communicativeness of the code.

![Figure 1: Abstraction vs Instability Graph](image)

(a) According to Martin, what makes a design rigid? What kind of code smell does a rigid design suffer from?

**Solution:**

A rigid design is one that is hard to change because making a single change would entail making multiple changes in the related modules. A rigid design suffers from the *Shotgun Surgery* or *Non-localized Plan* code smell.

(b) Explain – in words without referring to Martin’s formulas – what makes a class stable.

**Solution:**

A stable class is one that does not depend on any other classes but has other classes depending on it. Martin calls the former *independence* and the latter *responsible*.

(c) Describe the *category* at position (0,0) in Figure 1 in terms of stability and abstraction. Why is such a category not desired?

**Solution:**

The category at (0,0) is very stable but not abstract. It is not desired because its stability makes it hard to change. And its concreteness makes it hard to extend. The (0,0) location marks a design that is *rigid*. 
(d) Why is it desirable for a category to be as close as possible to the main sequence shown in Figure 1?

**Solution:**

A category that is close to the main sequence is neither too abstract nor too instable. Thus, it has the appropriate number of classes that depend on it (and an appropriate number of classes that it depends on) to make maintenance easier. This is the kind of design that object-oriented system should aim for.
17. Refactoring

**Listing 2: Code with code smells**

```java
public int readStuff(String fn) throws IOException {
    // Open a file for read/write ("rw") access
    RandomAccessFile f = new RandomAccessFile(fn, "rw");
    f.seek(100); // Move to byte 100 of file
    byte[] bytes = new byte[100]; // Create a buffer to hold the data
    f.read(bytes); // Read 100 bytes from file
    int i = f.readInt(); // Read a 4-byte integer from file
    f.seek(100); // Move back to byte 100
    f.writeInt(i); // Write the integer first
    f.write(bytes); // Then write the 100 bytes

    if (i > 10000) {
        f.close();
        return -1; // Return error code
    }

    if (i < 1234) {
        f.seek(200); // Move back to byte 200
        f.writeInt(i); // Write the integer first
        f.write(bytes); // Then write 100 bytes
    }

    f.close(); // Close the file
    return 0; // Return no error
}
```

(a) List three code smells that you detect in the Listing 2. It is acceptable to circle and use arrows to point to the offending code segments. Use the name of the code smell. If you do not remember the name simply do your best to describe the type of code smell.

**Solution:**

- Non-descriptive method name `readStuff`. The method both reads and writes.
- Non-descriptive variable names `fn`, `f`, `bytes`, `i`.
- Excessive comments. The comments are necessary because the code doesn’t speak for itself.
- Magic numbers on lines 5, 6, 9, 13, 15, 18, 19 and 26.

Non-descriptive conditional statements on lines 13 and 18.

Feature envy. The method send almost all of its method calls to the temporary variable f.

(b) Provide the refactored version of the code in Listing 2 that addresses the code smells that you listed in part a).

Listing 3: Sample refactoring solution

```java
public int rearrangeFile(String fileName) throws IOException {
    MidtermFile file = new MidtermFile(fileName, "rw");
    return file.rearrange();
}

public class MidtermFile extends RandomAccessFile {
    private static final int BUFFER_SIZE = 100;
    private static final int START_OFFSET = 100;
    private static final int SECOND_OFFSET = START_OFFSET * 2;
    private static final int LOWER_VALUE = 1234;
    private static final int MAXIMUM_VALUE = 10000;
    private static final int SUCCESS_CODE = 0;
    private static final int ERROR_CODE = -1;

    public MidtermFile(String fileName, String mode)
    throws FileNotFoundException {
        super(fileName, mode);
    }

    public int rearrange() throws IOException {
        seek(START_OFFSET);
        byte[] buffer = new byte[BUFFER_SIZE];
        read(buffer);
        int numberToMove = readInt();
        seekAndWrite(START_OFFSET, buffer, numberToMove);
        if (needsDuplication(numberToMove)) {
            seekAndWrite(SECOND_OFFSET, buffer, numberToMove);
        }
        close();
        if (isAboveMax(numberToMove)) return ERROR_CODE;
        return SUCCESS_CODE;
    }

    private boolean needsDuplication(int value) {
        return value < LOWER_VALUE;
    }
```
private boolean isAboveMax(int value) {
    return value > MAXIMUM_VALUE;
}

private void seekAndWrite(int offset, byte[] buffer, int number) throws IOException {
    seek(offset);
    writeInt(number);
    write(buffer);
}