Name:

NetID:

Final Exam

CS 427 Software Engineering I (Fall 2016)

TIME LIMIT = 3 hours
20 PAGES

Upon receiving your exam, print your name and netid neatly in the space provided above; print your netid in the upper right corner of every page.

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 on the exam and reporting you to the UIUC office in charge of cheating. We cannot give any clarifications about the questions during the exam. 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 empty pages at the end if needed. Nothing except your exam and pencils/pens/erasers should be on your desk, not even additional paper sheets or your phone. Put all your books, notebooks, electronic devices in your bags; turn off your phones before you put them in your bag. 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.

Part One: ________**/50**

Part Two:

Testing ________**/16** **Total Score** ________**/100**

Class Invariants ________**/4**

Design Patterns ________**/20**

Smells & Refactoring ________**/10**

Name ________________________________ **NetID** __________________

Draw an X in the box correspond-ing to the prob-lem and answer.

#	a	b	c	d	e
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

#	T	F
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		

Part One (2 points each)

The first section of the test contains multiple choice questions. Once you determine an answer, draw an X or fill in the box on the answer sheet (on the first page). The five columns (a-e) for the first 20 questions represent the multiple-choice answers. For 21-30, check the T (true) or F (false) column.

1. Which of the following statements is NOT true about tests:
   a. Tests can document faults and code
   b. Tests can be used to demonstrate the lack of faults
   c. Tests can be used to improve the code quality
   d. Tests can be used to determine if software is ready to be released
   e. Tests can be used to make software easier to change

2. We have a simple project with 3 modules A, B, and C. A depends on B and C is independent. Test suites Ta, Tb, and Tc run separately on the three modules A, B, and C, respectively. We make changes to code in modules A and C. Which of the test suites Ta, Tb, and Tc we should run:
   a. Ta and Tc
   b. Tb and Tc
   c. Ta and Tb
   d. All of them
   e. Not enough details

3. Which of the following statements is NOT true about system documentation:
   a. Although a software project does not proceed in a rational way, it is acceptable to fake the ideal rational process when producing the documentation
   b. Documentation can help to improve design and plan further construction
   c. Good documentation explains the mistakes the team made along the way
   d. The system documentation can make it look like you followed a waterfall development process
   e. Documentation also needs to be tested

4. To record sequences of actions that can be later playbacked programmatically (e.g., editor keyboard macros), you should use which of the following design patterns:decision coverage
   a. Command
   b. Observer
   c. Strategy
   d. Interpreter
   e. Visitor

5. When writing a unit test for a class that depends on an external database, you should:
   a. Use a connection to a real database in production
   b. Use a connection to a test database in staging
   c. Create a simple test database with dummy entries
   d. Mock the database connection
   e. Avoid writing tests for methods interacting with the database

6. When using git, to update remote repository with the changes you have made on the local repository you should use:
   a. git add
   b. git commit
   c. git push
   d. git update
   e. git pull

7. Which of the following is NOT a reverse engineering pattern/activity:
    a. Interview users/developers during demo
    b. Read the codebase in one hour
    c. Write tests to understand
    d. Skim through the documentation
    e. Refactor to understand

8. Buying insurance is ____ ; Data redundancy is ___ ; Opting not to develop a high risk product is ___.
    a. Risk protection; Risk protection; Risk protection
    b. Risk protection; Risk protection; Risk avoidance
    c. Risk protection; Risk avoidance; Risk protection
    d. Risk avoidance; Risk protection; Risk avoidance
    e. Risk avoidance; Risk avoidance; Risk protection

9. Which of the following is true:
    a. Methods with low cyclomatic complexity should be rewritten/refactored
    b. Methods with high cyclomatic complexity should be rewritten/refactored
    c. Cyclomatic complexity measures the number of dependent paths through the procedure
    d. Cyclomatic complexity is not a software metric
    e. Cyclomatic complexity measures the number of eyes in the control flow graph

10. Which of the following is NOT a code smell:
    a. Refused bequest
    b. Duplicated code
    c. Large parameter list
    d. Many classes
    e. Feature Envy

11. Which of the following is NOT an essential requirement of Agile practice:
    a. Having frequent interactions with customers and stakeholders
    b. Work iteratively, frequent delivery of software to customers
    c. Get requirements in the form of features
    d. Working in a sequential, phase by phase design process
    e. All of the above are Agile practices

12. Which of the following is an aspect that software configuration management (SCM) is NOT concerned with:
    a. Tracking and meeting release deadlines
    b. Keeping track of code changes
    c. Keeping track of reported errors and their status
    d. Managing external library dependencies needed to build your system
    e. None of the above

13. Which one of the following is NOT a key element in bug reporting:
    a. Check reproducibility of the failure as part of writing a bug report
    b. Document a crisp sequence of actions that will reproduce the failure
    c. Explain what implications the bug has
    d. Look for related failures in SUT
    e. Test it differently to isolate the bug

14. You have a big system with lots of components and it takes minutes or hours to run all the tests. You want one or few tests that you can run quickly to make sure that the system still at least runs after you make a change to it (even if some of the tests may fail). Such tests are called:
    a. Unit Tests

b. Integration Tests
c. Regression Tests
d. Smoke tests
e. None of the above

15. Which of the following is NOT a testing mistake:
    a. Test plans should be biased towards functional testing
    b. All tests should be automated
    c. Purpose of testing is to find bugs
    d. Documentation needs to be tested as well
    e. Using testing as a job for new programmers

16. Given a chat application, a team following the XP process decides to write user stories for their next iteration themselves, on behalf of their customer who is not responding to their requests for user stories. They have come up with the following five user stories:

User stories for a chat application

User Story	Priority	Point
a) Send audio chat message	1	1
b) Migrate the Database from MySQL to MongoDB	1	2
c) Add read receipt	2	2
d) Add block contact feature	3	1
e) Add profile picture	3	1

Which of the stories above is the least likely to be proposed by an actual customer?

17. Which of the following is NOT a usual step in the process of debugging a failure:
    a. Write automated test to reproduce the failure
    b. Find and understand the cause
    c. Write a bug report
    d. Fix the bug
    e. Add test case to regression suite

18. The following requirement is a __________: "The K framework is implemented only in Java or Scala."
    a. Functional requirement
    b. Non-functional requirement
    c. Constraint (or pseudo requirement)
    d. Formal requirement
    e. None of the above

19. Which of the following is NOT an Object-Oriented Design Quality Metric:
    a. Weighted Methods Per Class
    b. Depth of Inheritance Tree
    c. Number of Children
    d. Response for a Class

e. Lack of Coupling of Modules

20. Which of the following is a well-known benefit of pair programming:
    a. The pair finishes the task in half the time
    b. Company only needs half as many computers as programmers
    c. One person can write functional code while the other writes the corresponding tests, this way leading to better code coverage of tests
    d. A novice can be paired with an expert
    e. Increased discipline

True/False (1 points each)

For each of the sentences below, mark an X in the appropriate box on the answer sheet. Naturally, mark an X in the T box if the statement is true and an X in the F box if the statement is false.

21. When testing object-oriented software it is important to test each class method separately as part of the unit testing process

T 22. Frequent changes can be used as one of the metrics for reverse engineering.

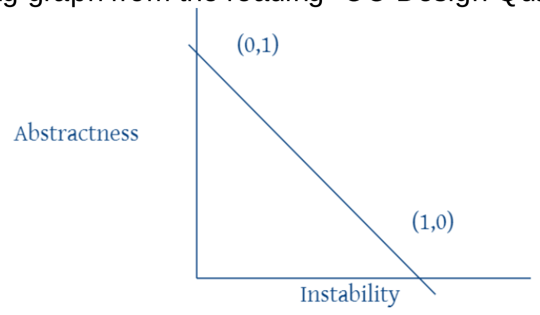
F 23. Suppose that you have to test a Java function that takes an integer between 1 and 10 as input and returns a Boolean. Let's say that the function has no loops and that your tests have 100% statement, branch, condition, and path coverage. You can therefore conclude that this function will never throw any uncaught exception.

F 24. Coupling measures dependencies within modules, so well designed systems strive to achieve high coupling

F 25. A large number of tests directly translate into a better quality software

F 26. Branch coverage requires all combinations of atomic Boolean expressions inside the branch condition are covered.

F 27. Consider the following graph from the reading "OO Design Quality Metrics":



The line in the graph symbolizes the category of systems that have maximal rigidity, in the sense that they are hard to extend and thus should be avoided.

F 28. Reengineering is the process of extracting the design and requirements from the code.

T 29. Software metrics are more useful as indicators of what's wrong rather than what's right about a system.

F 30. Acceptance testing is generally done by developers.

Testing [16 Points]

You are part of an **XP team** and your task is to develop a function that tells whether three given integers can be the sides of a triangle. Specifically, your team is supposed to eventually implement a function

```
public int isTriangle(int a, int b, int c) {
  …
}
```

which returns:

- 0 when a, b, c cannot be the sides of a triangle
- 1 when a, b, c can be the sides of an arbitrary triangle (we assume an arbitrary triangle has all the sides positive, *distinct*, and satisfying the triangle property that each side is smaller than the sum of the other two)
- 2 when a, b, c can be the sides of an isosceles triangle (we assume an isosceles triangle is a triangle with two sides equal and the third of *different* length)
- 3 when a, b, c can be sides of an equilateral triangle (a triangle with all its sides equal)

1. (0.5 points) Which of the following are you supposed to do first? (circle the correct answer)
(A) Implement a first version of the function
(B) [✓ Write white-box tests]
(C) Write black box tests

2. (2.5 points) Orthogonally to your answer to 1 above, here you are asked to write ten black box tests (each worth 0.25 points) to test the functionality of such a triangle checking function. Write these test cases by completing the black box test case template below. Mark the strategy for each test case as being either an equivalence class (EC), a boundary value analysis (BVA), or a diabolical (DT) test case.

Test ID	Description (a,b,c)	Expected Result	Actual Result	Strategy
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	
			Nothing here	

3. (2 points) Write a first iteration of the code for this function below, without worrying about overflows of arithmetic operations (that is, assume that integers are arbitrarily large):

```
public int isTriangle(int a, int b, int c) {







}
```

4. (4 points) If you were to implement your black-box tests (from part 2) using jUnit, would they give you, respectively, **full statement**, **full branch**, **full path**, and **full condition coverage** of your code (from part 3)? Explain why / why not. Write three JUnit-style asserts for three test cases, each covering something different and preferably different from what your black-box tests in part 2 cover. You can also assume that no overflows of arithmetic operations will happen in this part.

5. (1 point) Now it is time to worry about arithmetic operation overflows. Write a new test that should succeed but fails on your implementation above because of arithmetic overflows.

6. (2 points) By now, you probably realize that it is not trivial to correctly implement this function if we take overflows into account, too. Based on your code at 3 above, write a complete set of tests (as Junit-style assertions) that cover all possible paths to overflows that can take place in your code.

7. (2 points) Write a correct implementation of the triangle checking function and argue that it passes all your tests above (possible hint: “a + b > c” is mathematically equivalent to “a > c - b”).

```
    public int isTriangle(int a, int b, int c) {









    }
```

8. (1 point) Based on the example discussed here, explain why black box testing is needed after white box testing is conducted.

9. (1 point) Based on the example discussed here, explain why white box testing is needed after black box testing is conducted.

Class Invariants [4 Points]

Definition: “A logically implies B” means that if A is true then B is also true. For example, “x == x * y implies x == 0 or y ==1” and “y == x * x implies y >= 0”. Assume ideal mathematical integers, that is, do not worry about overflow of arithmetic operations.

Consider the class

```
class Sum {
    private int n, sum;

    public Sum(int n) {
        this.n = n<0 ? 0 : n;
        sum = 0;
        for(int i = 0; i <= n; i++) sum += i;
    }

    public void increment() {
        n++;
        sum += n;
    }

    public int getN() { return n; }

    public int getSum() { return sum; }

}
```

1. (2 points) Recall that “true” is always a class invariant, because it holds before and after each method is called, but a very imprecise one. We say that an invariant A is more precise than an invariant B if A logically implies B. What is the most precise invariant of the class above for `n` and `sum`?

2. (2 points) Give three class invariants Inv1, Inv2 and Inv3 for `n` and `sum` such that:

Inv1 logically implies Inv3
Inv2 logically implies Inv3
Inv1 does not logically imply Inv2
Inv2 does not logically imply Inv1

Design Patterns [20 Points]

1. (2 points) In the lecture notes you were asked to study at least one design pattern that was not covered in class. Describe such a design pattern briefly, including its name, intent, participants, and a short example (intuition only for the example is acceptable if a lot of code would be required).

The rest of the design patterns questions are related to the following Java code:

```
abstract class Element {}  // can also be an interface

class Book extends Element {
    private String title;
    private String author;
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle() { return this.title; }
    public String getAuthor() { return this.author; }
}

class Collection extends Element {
    private String name;
    private List<Element> elements;
    public Collection(String name){
        this.name = name; elements = new ArrayList<Element>();
    }
    public String getName() { return this.name; }
    public List<Element> getElements() { return this.elements; }
    public void addElement(Element element) { elements.add(element); }
}
```

2. (1 point) Do you recognize any design pattern in the code above? If yes, which one?

For the remaining questions, consider the collection `c1` created with the following code snippet:

```
Book b1 = new Book("Title1", "Author1");
Book b2 = new Book("Title2", "Author2");
Book b3 = new Book("Title3", "Author3");
Book b4 = new Book("Title4", "Author4");
Collection c1 = new Collection("Collection1");
Collection c2 = new Collection("Collection2");
Collection c3 = new Collection("Collection3");
Collection c4 = new Collection("Collection4");
c1.addElement(c2);
c1.addElement(c4);
c2.addElement(b1);
c2.addElement(c3);
c3.addElement(b2);
c3.addElement(b3);
c4.addElement(b4);
```

3. (4 points) Using the Interpreter Pattern, add a method (or more) `int totalInterpreter()` which calculate the total number of elements (books and collections) in a given element; or, regarding an element as a tree, the total number of nodes in the tree. For example, `c1.totalInterpreter()` returns 8. If you need to add more methods, specify for each method to which class you add it.

4. (4 points) Add a method (or more) `String prettyPrintInterpreter(String indent)` using also the <u>Interpreter Pattern,</u> which pretty prints a given element by printing each of its sub-elements on a new line, indented 4 spaces to reflect nesting. Moreover, each line should start with `String indent` (method parameter). For example, `c1.prettyPrintInterpreter("---- ")` produces the string

```
---- Collection1 {
----     Collection2 {
----         Author1 : Title1
----         Collection3 {
----             Author2 : Title2
----             Author3 : Title3
----         }
----     }
----     Collection4 {
----         Author4 : Title4
----     }
---- }
```

If you need to add more methods, specify for each method to which class you add it.

5. (2 points) What is the main disadvantage of the Interpreter Pattern that Visitor Pattern eliminates?

6. (7 points) Rewrite the code in 3 and 4 above using the Visitor Pattern instead of the Interpreter Pattern. Specifically, modify/extend the `Element` class (and possibly its sub-classes) generically to accept visitors, and then define two visitor classes, `TotalVisitor` and `PrettyPrintVisitor`, that achieve the same functionality as in 3 and 4 above. For example, the code

```
System.out.println(c1.accept(new TotalVisitor()));
System.out.println(c1.accept(new PrettyPrintVisitor("--- ")));
```

should print

```
8
--- Collection1 {
---     Collection2 {
---         Author1 : Title1
---         Collection3 {
---             Author2 : Title2
---             Author3 : Title3
---         }
---     }
---     Collection4 {
---         Author4 : Title4
---     }
--- }
```

Smells & Refactoring [10 Points]

The following is a variant of a refactoring exercise proposed by Martin Fowler, one of the most prominent experts in refactoring. The program prints out a statement of a customer's charges at a video store.

There are several classes that represent various video elements. DomainObject is a general class that holds a name:

```
class DomainObject {
    protected String _name = "no name";

    public String name() {
        return _name;
    };
}
```

Movie represents the notion of a film, and Tape represents any physical support on which a movie can be stored. A video store might have several tapes in stock of the same movie:

```
class Movie extends DomainObject {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private int _priceCode;

    public Movie(String name, int priceCode) {
        _name = name;
        _priceCode = priceCode;
    }

    public int priceCode() {
        return _priceCode;
    }
}

class Tape extends DomainObject {
    private String _serialNumber;
    private Movie _movie;

    public Tape(String serialNumber, Movie movie) {
        _serialNumber = serialNumber;
        _movie = movie;
    }

    public Movie movie() {
        return _movie;
    }
}
```

The Rental class represents a customer renting a movie:

```
class Rental extends DomainObject {
    private Tape _tape;
    private int _daysRented;

    public Rental(Tape tape, int daysRented) {
        _tape = tape;
        _daysRented = daysRented;
    }

    public Tape tape() {
        return _tape;
    }

    public int daysRented() {
        return _daysRented;
    }
}
```

The Customer class represents the customer. So far all the classes have been dumb encapsulated data. Customer holds all the behavior for producing a statement in its `statement()` method:

```
class Customer extends DomainObject {
    private Vector _rentals = new Vector();

    public Customer(String name) {
        _name = name;
    }

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }

    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            switch (each.tape().movie().priceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.daysRented() > 2)
                        thisAmount += (each.daysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.daysRented() * 3;
                    break;
                case Movie.CHILDREN:
                    thisAmount += 1.5;
                    if (each.daysRented() > 3)
                        thisAmount += (each.daysRented() - 3) * 1.5;
                    break;

            }
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1)
                frequentRenterPoints ++;
            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.val-
ueOf(thisAmount) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent
renter points";
        return result;
    }
}
```

There are more than 10 meaningful refactorings possible in the code above, going even as far as replacing the switch statement by using polymorphism. You are required to only **identify and apply 5 refactorings**. You will be graded based on how relevant your refactorings are in terms of improving code readability and maintenance, but not the performance of the resulting code. Some refactorings actually decrease the code performance (typically in places where it does not matter), relying on compilers to regain the performance through their optimizations or on future refactorings specifically targeted to improve performance. If it is not obvious why a refactoring is relevant, explain in a few words why you believe it is relevant. For each refactoring, state the code smell(s) that trigger it, show where it appears in the code (mark the code using your pen), and then sketch the relevant resulting code below. It could be that some refactorings apply to already refactored code, so write the new code carefully.