CPSC 310, Practice Midterm Questions ("100 points")

True/False, **2 points each**.

1. (T/F, if false explain why) In McConnell's "Productive work analysis", he claimed that early attention to process could help reduce the amount of developer time devoted to process overhead to 0 by the end of a project.

2. (T/F, if false explain why) Agile methods are different from the Spiral model because development is organized as iterations instead of one single implementation phase.

3. (T/F, if false explain why) User stories should always be accompanied by automated tests (such as JUnit), so that testing can be performed efficiently.

4. (T/F, if false explain why) The Daily Scrum meeting allows developers to show a demo of their work, so the product owner can determine if progress is on track.

5. (T/F, if false explain why) Managers should attend Daily Scrum meetings so that they can give feedback at the meeting about whether a product is achieving a company's business objectives.

6. (T/F, if false explain why) In Extreme Programming, actual customers should be involved in the creation of acceptance criteria for user stories.

7. (T/F, if false explain why) User stories describe implementation tasks that developers must complete.

Short Answer, 1-3 sentences **(4 points each)**

8. Describe why a traditional waterfall process could be preferred over a spiral model, in cases where the project requirements and implementation were well understood and highly predictable?

9. In the area of Agile methods, what does the acronym INVEST stand for?

10. What is one of the major problems caused when user stories are too big?

11. What is the purpose of the Scrum Master in the Scrum methodology?

12. What is one problem that is addressed by Collective Ownership of code in Extreme Programming?

13.a. Which design pattern is used in the following code example, consisting of the below classes and interface? The names are intentionally vague for the purpose of the exam. **(6 points)**

13.b. For each class, what participant role does it play in the design pattern **(2 points each)**?

```
class Bar implements Foo {
        Baz baz = new Baz();

        public String pwr(String x, String y) {
                double i = Double.parseDouble(x);
                int j = Integer.parseInt(y);
                double result = baz.exp(i, j);
                return Double.toString(result);
        }
}

/*********************/
class Baz {
        double exp(double a, int b) {
                double result = 1;
                for(int i=0;i < b; i++)
                        result *= a;
                return result;
        }
}

/*********************/

interface Foo {
        String pwr(String x, String y);
}

/*********************/
class Top {
        public static void main(String[] args) {
                Foo foo = new Bar();
                System.out.println(foo.pwr("10", "2"));
        }
}
```

14. The following classes make up a simple program which follows the `Decorator` pattern. On the following pages, create a new version the program which has similar behavior but follows the `Command` pattern and `Chain of Responsibility` pattern. In the new code, the `inc` action is supported by both a method in the `Num`–typed classes and also dedicated `Command` classes. Logging functionality will be achieved by a handler class, it should print out a generic message, "Command executed". You will also need a handler which actually executes the command. Different from the decorator program, the new program will use an invoker which loops repeatedly.

```
/***Decorator-based program***/

class Client {
        public static void main(String[] args) {
                Num num = new LoggingDecorator(new PositiveNum());
                num.inc();
        }
}

interface Num {
        void inc();
}

class PositiveNum implements Num {
        int x = 0;

        public void inc() { x++; }
}

abstract class NumDecorator implements Num {
        Num delegate;

        NumDecorator(Num num) { delegate = num; }

        public void inc() { delegate.inc(); }
}

class LoggingDecorator extends NumDecorator {
        LoggingDecorator(Num num) { super(num); }

        public void inc() {
                super.inc();
                System.out.println("Incremented");
        }
}
```

The new program will consist of the following classes:

`Client` (different from the Client on the previous page, the class is provided for you)
`SimpleInvoker` (this class is provided for you)
`Num` (same as previous, no need to copy onto the next page)
`PositiveNum` (same as previous, no need to copy onto the next page)
`IncCommand` (a new class that needs to be created, **4 points**)
`NumCommand` (a new interface that needs to be created, **4 points**)
`NumHandler` (a new interface or abstract class that needs to be created, **4 points**)
`LoggingHandler` (a new class that needs to be created, **4 points**)
`NumCommandExecutorHandler` (a new class that needs to be created, **4 points**)

```
class Client {

        public static void main(String[] args) {
                SimpleInvoker invoker = new SimpleInvoker();
                new Thread(invoker).start();

                PositiveNum num = new PositiveNum();
                IncCommand command = new IncCommand(num);
                invoker.queue(command);
        }

}

class SimpleInvoker implements Runnable {
        NumHandler handler = new LoggingHandler(new NumCommandExecutorHandler(null));
        Queue<NumCommand> q = new ConcurrentLinkedQueue<NumCommand>();

        void queue(NumCommand c) { q.add(c); }

        public void run() {
                while(true) {
                        NumCommand c = q.poll();
                        if(c != null) { handler.handleRequest(c); }
                }
        }
}

//Add code for new classes/interfaces below
```

15. The following `main` function does not compile because there is no guarantee that the `max` function returns exactly the same type as the types of its arguments. Modify the method type signature of the `max` method to provide a guarantee that the return type is exactly the same as the type of the arguments, assuming the types of the two arguments are the same. Do not modify any other code other than the method signature of the `max` method. Write the new method signature in the space provided below. **(8 points)**

    (6 points) The new method signature also does not break any of the other existing code listed on the page.

```
public static void main(String[] args) {
       MyScalar fst = …; //details elided
       MyScalar snd = …; //details elided
       MyScalar result = max(fst, snd); //Compiler error on this line
}

/*******************************/

interface Scalar {
       boolean greaterThan(Scalar other);
}

/*******************************/

class MyScalar implements Scalar {
       //details elided ...
}

/*****************************/

public static Scalar max(Scalar x, Scalar y) {
        if(x.greaterThan(y))
          return x;
        else
          return y;
}

/*****************************/
```

16. Write a generic method which replaces an object at a given index of an array, with a given replacement object. The method must guarantee the replacement to be a compatible type with the given array. The index, the replacement object , and the array, are given as arguments to the method **(12 points)**

17. Will the following method compile? If not, why? **(6 points)**

```
public static void print(List<? extends Number> list) {
    for (Number n : list)
        System.out.print(n + " ");
    System.out.println();
}
```