COMP 401 Final: Question Part

Saturday, Dec 10, 2016 12pm-3pm

Instructions

1. Please spread out and try and sit in alternate seats.
2. This is a closed book exam.
3. You will not be penalized for errors in Java syntax.
4. Do not write on this question document. A separate answer document is also being distributed.
5. The question document has:
    - 6 numbered pages including this one and any marked blank pages.
    - 5 questions.
6. The exam is worth 150 possible points. Point values appear in brackets next to each question. **Points have been allocated for writing the onyen, name and initials clearly in capitals.**
7. You are not required to comment or annotate any code you write, but may get partial credit if you write appropriate comments/annotations but incorrect code.
8. If you need to make any assumptions to clarify a problem, *write your assumptions down.* Only reasonable assumptions get full credit.
9. Please inform the proctor of anything in the exam that you think is a mistake.
10. Your code will be evaluated not only for correctness, but also for time and space efficiency and *style*. Certain relevant stylistic rules, covered in class, are not explicitly stated. *A major part of your grade will depend on how well you apply these rules*. In other words, you must adhere to the same requirements that you would if this was a class assignment (with a perfect human grader!), except where explicitly stated otherwise. **This implies, for instance, if you are asked to implement a class or subclass, you must define any additional types (classes and interfaces), if necessary, to follow the design principles, pattern and other concepts covered in class. Similarly, if you are asked to implement certain methods, you should write other methods, if necessary to follow stylistic rules**.
11. You cannot use any Java capabilities not covered in class.
12. If you do not understand some English word, do not hesitate to ask the proctor. Naturally, you are expected to know the computer science terms defined in class.
13. Write clearly using a pen/pencil with a dark color – we will be scanning your exams. You will lose points if we cannot understand what you wrote. Please try and write in the allocated space on the answer document, as that will reduce mistakes in grading of scanned exams. **We strongly recommend that you answer on scratch paper before committing your solution to the answer sheet.**
14. Put your initials on the front page of the answer document and on each page of the document, in case the pages are separated.

1. [31 pts.] UNDERSTANDING CODE

Study and understand the code below. It is used in all five questions.

```
public interface Expression {
      int evaluate();
}
```

```
public interface DigitExpression extends Expression{
      public char getDigit() ;
      public void setDigit(char aDigit);
}
```

```
public class ADigitExpression implements DigitExpression{
      char digit;
      public ADigitExpression (char aDigit) {
            setDigit(aDigit);
      }
      public char getDigit() {
            return digit;
      }
      public void setDigit(char aDigit) {
            digit = aDigit;
      }
      /*
       * Converts a digit character to the corresponding numeric value.
       * For example parseInt ('5') returns the int value 5.
       */
      public static int parseInt(char aChar) {
            return aChar - '0'; // characters are ordered
      }
      protected String toErrorMessage(char anIllegalChar) {
            return anIllegalChar + " is not a digit";
      }
      public int evaluate() {
            if (!Character.isDigit(digit)) {
                  System.out.println(toErrorMessage(digit));
                  return Integer.MAX_VALUE;
            }
            return parseInt(digit);
      }
}
```

```
public class AMinusExpression implements Expression{
      Expression leftOperand;
      Expression rightOperand;
      public AMinusExpression(Expression aLeftOperand, Expression aRightOperand) {
            leftOperand = aLeftOperand;
            rightOperand = aRightOperand;
      }
      public int evaluate() {
            return leftOperand.evaluate() - rightOperand.evaluate();
      }
}
```

```java
public class AnExpressionTester implements ExpressionTester{
      public static final char DIGIT1 = '5';
      public static final char DIGIT2 = '4';
      public static final char DIGIT3 = '3';
      public static final char NON_DIGIT = 'a';
      protected Expression createDigitExpression(char aDigit) {
            return new ADigitExpression(aDigit);
      }
      protected Expression createLegalExpression() {
        return new AMinusExpression (
            createDigitExpression(DIGIT1),
            new AMinusExpression(
              createDigitExpression(DIGIT2),
              createDigitExpression(DIGIT3)));
      }
      protected void print(Expression anExpression) {
            int aValue = anExpression.evaluate();
            if (aValue != Integer.MAX_VALUE) {
                  System.out.println(aValue);
            }
      }
      public void testExpressions() {
            Expression aLegalExpression = createLegalExpression();
            print (aLegalExpression);
            Expression anIllegalExpression = createDigitExpression(NON_DIGIT);
            print (anIllegalExpression);
      }
}
```

```java
public class ExpressionDriver {
      public static void main (String[] anArgs) {
            (new AnExpressionTester()).testExpressions();
      }
}
```

(a) [4 pts] Give the header of a factory method in this code.

(b) [4 pts] Give a precondition tested by a method in this code.

(c) [13 pts] Draw the physical structure of the object (based on instance variables, not properties) assigned to `aLegalExpression` in `AnExpressionTester`. Label the root, an internal node, and a leaf node.

(d)[10 pts] Give the output of the main method in `ExpressionDriver`.

2. [27 pts.] EXCEPTIONS

This question builds on the code in Q 1 and code given below. The exception below is unchecked. The argument passed to the constructor can be accessed by calling the getMessage() instance method.

```java
public class AnIllegalCharacterException extends RuntimeException{
      public AnIllegalCharacterException(String aMessage) {
            super(aMessage);
      }
}
```

(a) [12 pts] Create a subclass of `ADigitExpression` (Q. 1), called `AnExceptionBasedDigitExpression(char aDigit)` that is like the superclass except that it throws `AnIllegalCharacterException` if the instance variable is not a digit.

The header and constructor of the class are given to you.

```
public class AnExceptionBasedDigitExpression extends ADigitExpression {
      public AnExceptionBasedDigitExpression(char aDigit) {
            super(aDigit);
      }
...
}
```

(b) [15 pts] Create a subclass of `AnExpressionTester`, called `AnExceptionBasedDigitExpression` that is like the superclass except that it creates instances of `AnExceptionBasedDigitExpression` instead of instances of `ADigitExpression`. The code below shows how this class is to be used.

```
public class ExceptionBasedExpressionDriver  {
      public static void main (String[] anArgs) {
            (new AnExceptionBasedExpressionTester()).testExpressions();
      }
}
```

The output of `testExpressions()` executed on instances of `AnExpressionTester` and `AnExceptionBasedExpressionTester` should be identical and of course, as always, your goal should be to avoid code duplication.

The header of the class is given to you.

```
public class AnExceptionBasedExpressionTester extends AnExpressionTester{
      …
}
```

3. [24 pts.] Observable Bean and Generics

This question builds on the code in Q 1 and code given below.

```
public interface PropertyListenerSupport {
      public void add(PropertyChangeListener l);
      public void notifyAllListeners(PropertyChangeEvent event);
}
```

```
public class APropertyListenerSupport implements PropertyListenerSupport {
      List<PropertyChangeListener> observers = new ArrayList<PropertyChangeListener>();
      public void add(PropertyChangeListener l) {
            observers.add(l);
      }
      public void notifyAllListeners(PropertyChangeEvent event) {
            for (int index = 0; index < observers.size(); index++) {
                  observers.get(index).propertyChange(event);
            }
      }
```

```
public interface PropertyListenerRegisterer {
      public void addPropertyChangeListener(PropertyChangeListener aListener);
}
```

(a) [4 pts] Give (reproduce) an *elaboration* of a generic type (class or interface) used in the code above.
(b) [20 pts] Create a subclass of `ADigitExpression` (Q. 1), called `AnObservableDigitExpression` that is like the superclass except that it can be used to instantiate an observable bean that notifies its observers about changes to its Digit property.

The header of the constructor of `PropertyChangeEvent` is given below.

```
public PropertyChangeEvent(Object source, String propertyName,
                           Object oldValue, Object newValue);
```

4. [17 pts.] THREADS

This question also builds on the code in Q 1. Write a command-object class and a main class that, together, create an instance of `AnExpressionTester` and execute its `testExpressions()` method in a separate thread (that is, a thread that is not the main thread).

5. [46 pts.] GRAMMARS AND RECURSIVE DESCENT PARSING

Again, this question builds on the code in Q 1. The goal here is to convert certain strings representing expressions into instances of the `Expression` type defined in in Q 1.

```
<Expression> --> digit
<Expression> --> digit - <Expression>
```

Here `digit` is a one-character token in the range 0..9. Thus, a legal expression string consists only of digit and minus characters. The first rule says that a string with just a digit is a legal expression. The other rules is recursive. It says that a digit followed by a minus character followed by a string that is a legal expression is also a legal expression.

(a) [2pts] Identify a non-terminal defined by the grammar.
(b) [2 pts] Identify a terminal defined by the grammar.
(c) [3 pts] Identify the root defined by the grammar.
(d) [40 pts]

Write a recursive descent parser class, called `AnExpressionParser`, that implements the interface `ExpressionParser` given below and provides a constructor taking a String argument.

```
public interface ExpressionParser {
    public Expression parseExpression();
}
```

The method `parseExpression()` converts the string passed as the constructor argument to an instance of `Expression` that points to the parse tree defined by the grammar above. The following code shows how `AnExpressionParser` is to be used.

```
public class ExpressionParserDriver {
      public static void main (String[] args) {
            Expression anExpression1 = (new AnExpressionParser("5")).parseExpression();
            Expression anExpression2 = (new AnExpressionParser("5-4")).parseExpression();
            Expression anExpression3 = (new AnExpressionParser("5-4-3")).parseExpression();
            System.out.println (anExpression1.evaluate());
            System.out.println (anExpression2.evaluate());
            System.out.println (anExpression3.evaluate());
      }
}
```

`anExpression1` should be an instance of `ADigitExpression`, `anExpression2` should be an instance `AMinusExpression` (Q.1) whose two instance variables should point to instances of `ADigitExpression`(Q,1), and `anExpression3` should be equivalent to the object returned by `createLegalExpression` of `AnExpressionTester` (Q. 1) . Of course, the parser should handle arbitrary legal expression strings. You can assume no illegal string is passed to the constructor of the parser – thus no error handling should be done.

You will get partial credit based *on the legal cases you handle correctly. So do handle some cases even if you cannot handle all.* You will get some points for getting the method headers completely correct. You should be able to detect the three examples above, even if you cannot identify the recursive step. Once you do these examples, you may be able to identify the recursive step. If you give a partial solution, please document the cases you can and cannot handle. It will be a good idea to write comments for yourself and the grader though you will not lose points for not commenting if you have a *clear* and *correct* solution. You should probably do the work on a scratch paper before committing it to the exam.

The grammar rules above *are not necessarily consistent with standard arithmetic or Java* – so do not make any assumptions based on your knowledge of expressions in other domains.

The header of the class and its constructor are given to you.

```
public class AnExpressionParser implements ExpressionParser {


      public AnExpressionParser(String aString) {//fill code in constructor


      }
```