CSU213 Exam 1 – Spring 2005

Name: _____

Student Id (last 4 digits) : _____

Instructor's Name: _____

High School/State: _____

- Write down the answers in the space provided.
- You may use all forms that you know from *ProfessorJ (Beginner)*. If you need a method and you don't know whether it is provided, define it.
- Remember that the phrase "develop a class" or "develop a method" means more than just providing a definition. It means to design them according to the design recipe. You are *not* required to provide a method template unless the problem specifically asks for one. However, be prepared to struggle if you choose to skip the template step.
- We will not answer *any* questions during the exam.

Good luck.

Problem	Points	/
1		/ 8
2		/ 15
3		/ 10
4		/ 15
5		/ 6
6		/5 extra
Total		/ 54

Problem 1 8 Points

Take a look at these structure and data definitions:

```
abstract class AMessage {}

class SymbolM extends AMessage {
 String s;
 SymbolM(String s) {
  this.s = s;
 }
}

class StringM extends AMessage {
 String s;
 StringM(String s) {
  this.s = s;
 }
}

class Message extends AMessage {
 int slot;
 String cont;
 Message(int slot, String cont) {
  this.slot = slot;
  this.cont = cont;
 }
}
```

Answer the following questions in this context:

1. Write down three distinct data examples, one per clause:

   Solution [POINTS 3]

   ```
   new SymbolM("hello")
   new StringM("world")
   new Message(10, "hello")
   ```

2. Write down the template for a method on `AMessage`:

Solution

[POINTS 3: 1 for abstract, 1 for three concrete, 1 for selector expressions]

```
abstract class AMessage {
 abstract ??? method();
}

class SymbolM extends AMessage {
 String s;
 SymbolM(String s) {
  this.s = s;
 }
 ??? method() {
   ... this.s ...
 }
}

class StringM extends AMessage {
 String s;
 StringM(String s) {
  this.s = s;
 }
 ??? method() {
   ... this.s ...
 }
}

class Message extends AMessage {
 int slot;
 String cont;
 Message(int slot, String cont) {
  this.slot = slot;
  this.cont = cont;
 }
 ??? method() {
   ... this.slot ... this.cont ...
 }
```

```
}
```

3. Write down a purpose statement for the program that translates a `AMessage` into a `String`:

   Solution [POINTS 1]

```
abstract class AMessage {
 // convert this message into a string
 abstract ??? method();
}
```

4. Write down three method examples (one per variant) for the method `countMessage`, which counts how many characters are contained in all strings associated with a given `AMessage`.

   Solution [POINTS 1]

```
new SymbolM("hello").countMessage() // should be
5

new StringM("world") // should be
5

new Message(10, "hello") // should be
5
```

Problem 2 15 Points

The main purpose of this exercise is to develop the method `listOOS`. It consumes a list of basic event strings (`"on"`, `"off"`, `"steady"`); its result is an `OOS` that represents how many times `"on"`, `"steady"`, and `"off"`, respectively, occur in the given list.

Here are the relevant class definitions:

```
// represent a sequence of events as strings
abstract class AEvents {}

class ConsE extends AEvents {
 String first;
 AEvents rest;
 ConsE(String first, AEvents rest) {
  this.first = first;
  this.rest = rest;
 }
}

class MtE extends AEvents {
 MtE() { }
}

// represent a count of "on", "off", "steady" events
class OOS {
 int on;
 int off;
 int steady;
 OOS(int on, int off, int steady) {
  this.on = on;
  this.off = off;
  this.steady = steady;
 }
}
```

Answer the following questions in this context.

1. Provide a template for a method that consumes a String and processes an `OOS` structure.

Solution

```
;; [POINTS 1, for all selector expressions]
// represent a count of "on", "off", "steady" events
class OOS {
 int on;
 int off;
 int steady;
 OOS(int on, int off, int steady) {
  this.on = on;
  this.off = off;
  this.steady = steady;
 }

 OOS method(String e) {
  ... this.on ... this.off ... this.steady ...
 }
}
```

2. Define the OOS method `update`, which consumes one of three strings: "on", "off", "steady". Its result is an OOS structure with the matching counter bumped by one; the other two counters remain the same.

Solution

```
;; [POINTS 3, for all three branches correct]
 OOS update(String e) {
  if (e.equals("on")
    new OOS(this.on+1,this.off,this.steady);
  else if (e.equals("off")
    new OOS(this.on,this.off+1,this.steady);
  else (e.equals("steady")
    new OOS(this.on,this.off,this.steady+1);
 }
```

3. Formulate a contract and a purpose statement for `listOOS`:

Solution

```
abstract class AEvents {
 // count how many times "on", "steady", "off"
 // occur on this list [POINTS 1: this]
 abstract OOS listOOS();  //  [POINTS 1]
}
```

4. Provide two distinct method examples:

Solution [POINTS 2]

Given:	Wanted:
`new MtE()`	`new OOS(0,0,0)`
`new ConsE("on",new ConsE("on",new MtE()))`	`new OOS(2,0,0)`

5. Define the method `listOOS`. Develop auxiliary methods as needed.

Solution [POINTS 5]

```
// represent a sequence of events as strings
abstract class AEvents {
 // count how many times "on", "steady", "off"
 // occur on this list
 abstract OOS listOOS();
}

class ConsE extends AEvents {
 String first;
 AEvents rest;
 ConsE(String first, AEvents rest) {
  this.first = first;
  this.rest = rest;
 }
 OOS listOOS() {
  return this.rest.listOOS().update(this.first);
 }
}

class MtE extends AEvents {
 MtE() {
  return new OOS(0,0,0);
 }
}
```

6. Translate your examples for `listOOS` into tests.

Solution

```
;; TESTS: [POINTS 2, each one correctly; 0 if not examples]

--- I am stumped because I don't know how you teach tests ---
```

Problem 3 10 Points

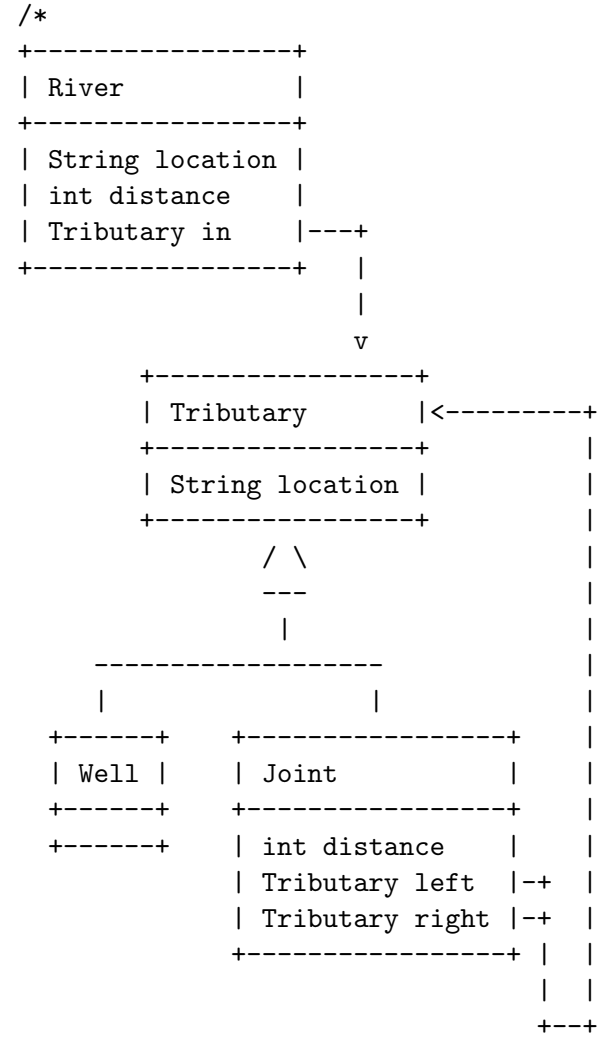
Here are data definitions from an environmental surveillance program for rivers:

```
// a system of river tributaries (composite pattern)
class River {
  String location;
  int distance;
  Tributary in;
  River(String location, int distance, Tributary in) {
    this.location = location;
    this.distance = distance;
    this.in = in;
  }
}

abstract class Tributary {
  String location;
}

class Well extends Tributary {
  Well(String location) {
    this.location = location;
  }
}

class Joint extends Tributary {
  int distance;
  Tributary left;
  Tributary right;

  Joint(String location, int distance,
        Tributary left, Tributary right) {
    this.location = location;
    this.distance = distance;
    this.left = left;
    this.right = right;
  }
}
```

• Draw a class diagram of these definitions.

Solution

[POINTS 2]



• Provide a data example of a `River` that involves all three classes.

Solution

[POINTS 1: data example]

```
Tributary c = new Well("C");
Tributary d = new Well("D");
Tributary b = new Joint("B",20,c,d);
River e = new River("A",10,b)
```

• Develop a template for procesing river systems.

Solution

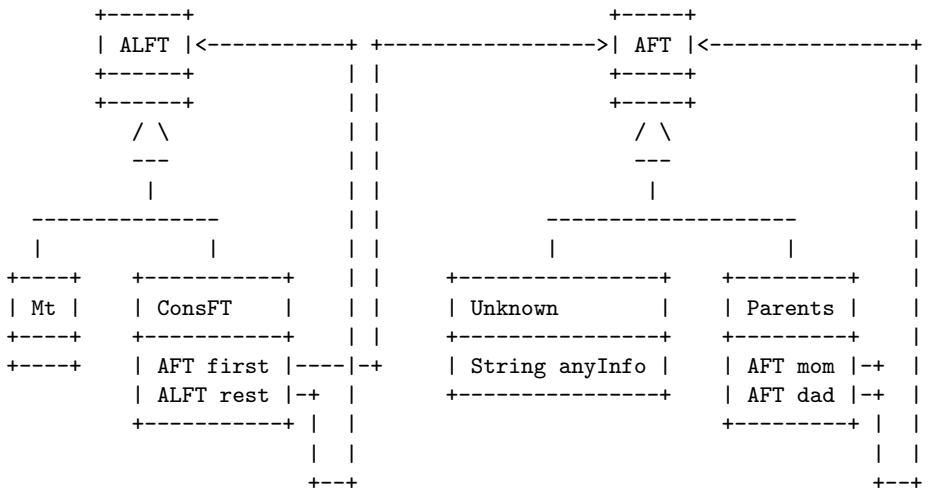
[POINTS 7]

```
class River {
  String location;
  int distance;
  Tributary in;
  ...
  ??? method() {
   ... this.location ... this.distance ... this.in.method() ...
  }
}

// a system of river tributaries
abstract class Tributary {
  String location;

  abstract ??? method();
}

class Well extends Tributary {

 ??? method() {
   ...
 }
}

class Joint extends Tributary {
  int distance;
  Tributary left;
  Tributary right;
  ...
  ??? method() {
    ... this.location ... this.distance ...
    ... this.left.method() ... this.right.method()
  }
}
```

Problem 4 15 Points

Develop a program that counts the number of unknowns in a list of family trees.

Here is the data definition in the form of a class diagram:



• Translate this diagram into a class hierarchy

Solution

[POINTS: 6]

```
// list of family trees
abstract class ALFT {}

class Mt extends ALFT {
  Mt() { }
}

class ConsFT extends ALFT {
  AFT first;
  ALFT rest;
  ConsFT(AFT first, ALFT rest) {
    this.first = first;
    this.rest = rest;
  }
}

// a family tree
abstract class AFT {}

// the unknown ancestor
class Unknown extends AFT {
  String anyInfo;
  Unknown(String anyInfo) {
    this.anyInfo = anyInfo;
  }
}

class Parents extends AFT {
  AFT mom;
  AFT dad;
  Parents(AFT mom, AFT dad) {
    this.mom = mom;
    this.dad = dad;
  }
}
```

• Add the method for counting the number of `Unknown`s. You may employ the following style if you are comfortable with in:

```
// in Unknown:
int countUnknown() {
 return 1;
}
```

Solution

// [POINTS: 9, 5 for each and 4 for the four recursions]

```
// in AFT
// count the number of Unknowns in this AFT
abstract int countUnknown();

// in Parents
int countUnknown() {
 return this.mom.countUnknown() + this.dad.countUnknown();
}

// in ALFT
// count the number of Unknowns in this list of AFTs
abstract int countUnknown();

// in Mt
int countUnknown() {
 return 0;
}

// in ConsFT
int countUnknown() {
 return this.first.countUnknown() + this.rest.countUnknown();
}
```

Problem 5 6 Points

Your boss, an electrical engineer who thinks he can program, wrote these two classes:

```
class UP {
  int x;

  UP(int x) {
   this.x = x; }

  boolean zero() {
   return 0 == this.x; }

  UP count() {
   return
    new UP(this.x+1); }
}
```

```
class DOWN {
  int x;

  DOWN(int x) {
   this.x = x; }

  boolean zero() {
   return 0 == this.x; }

  DOWN count() {
   if (this.zero())
    return
     new DOWN(this.x-1);
   else
    return this; }
}
```

First you naturally add purpose statements to each class and method. You know that someone will have to read these things at some point and a purpose statement is always a good idea.

Solution

[POINTS: 3]
UP: a class that counts up
zero: is this counter 0?
count: create a counter from x+1
DOWN: a class that counts down to 0
zero: is this counter 0?
count: create a counter from x-1, if it isn't 0 yet

Second, you abstract the two classes as much as possible, because you know that they both represent counters (and will need many more common methods):

Solution

[POINTS: 3]

```
class COUNTER {
  int x;
  COUNTER(int x) {
    this.x = x;
  }

  boolean zero() {
   return 0 == this.x;
  }
}

class UP extends COUNTER {
  UP(int x) { super(x); }

  UP count() {
   return
    new UP(this.x+1);
  }
}

class DOWN extends COUNTER {
  DOWN(int x) { super(x); }

  DOWN count() {
   if (this.zero())
    return
     new DOWN(this.x-1);
   else
    return this;
  }
}
```

Problem 6 5 Points

Your boss struck again. This time he should have it right, it's all about boolean expressions, which is what these signals are all about. He is trying to evaluate Boolean expressions like this:

```
(true /\ x) \/ false
```

where `x` is a variable that stands for booleans.

Here is how far he got:

```
// a representation for Boolean expressions
abstract class ABool {
  // evaluate this expression as if it were a Java boolean expression
  abstract boolean evaluate(String y);
}

class And extends ABool {
  ABool left;
  ABool right;

  And(ABool left, ABool right) {
    this.left = left;
    this.right = right;
  }

  boolean evaluate(String y) {
    return _________
  }
}

class Or extends ABool {
  ABool left;
  ABool right;

  Or(ABool left, ABool right) {
    this.left = left;
    this.right = right;
  }
```

```
  boolean evaluate(String y) {
    return _________
  }
}

class Not extends ABool {
  ABool op;

  Not(ABool op) {
    this.op = op;
  }

  boolean evaluate(String y) {
    return !(this.op.evaluate(y));
  }
}

class Const extends ABool {
  boolean b;

  Const(boolean b) {
    this.b = b;
  }

  boolean evaluate(String y) {
    return this.b;
  }
}

class Var extends ABool {
  String x;
  Var(String x) {
    this.x = x;
  }

  boolean evaluate(String y) {
   _________
  }
}
```

His idea is that the string he passes to `evaluate` is the name of the variable that is `true`. But now he doesn't know how to finish the method. Can you help?

Naturally you supplement the examples that he forgot to make:

```
class Ex {
  ABool b1 = new Const(true);
  ABool b2 = new Var("x");
  ABool b3 = new And(this.b1,this.b2);
  ABool b4 = new Or(this.b1,this.b2);
  ABool b5 = new Not(this.b4);

  boolean r1 = this.b3.evaluate("x"); // should be true
  boolean r2 = this.b3.evaluate("y"); // should be false

  Ex() {}
}
```

Solution [POINTS: 3 for the last one (it needs equals), 1 each for the recursions]