CSU213 Exam 2 – Fall 2007

Name: _____

Student Id (last 4 digits): _____

- Write down the answers in the space provided.
- You may use any valid Java code. We provide the APIs for any library classes and their methods that you may need. If you need a method and you don't know whether it is provided, define it.
- Remember that the phrase "design a class" or "design 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		/24
2		/13
3		/18
Total		/55

Reference page

- For all classes assume that the `tester` package is available.
- Write all tests as invocation of the `test` method in the `Examples` class:

```
test("name of the test", actual, expected)
```

- If the problem consumes a `Traversal` and it is not known how the `Traversal` is implemented, define the example data as:

```
Traversal<Data> tr = '(data1, data2, ...);
```

- In all problems you can omit the visibility modifiers, unless it is
  - required by the interface you are implementing
  - required by the the class you are extending
  - the problem explicitly asks you to do so
- Throughout the exam you may use the clasess `AList<T>`, `ConsList<T>` and `MTList<T>` to represent a collection of data. You may also assume that the methods that implement the `Traversal<T>` interface have been defined for this class hierarchy.
- Recall that the `Traversal<T>` interface is defined as follows:

```
public Traversal<T> getRest()
    throws IllegalUseOfTraversalException;
public T getFirst()
    throws IllegalUseOfTraversalException;
public boolean isEmpty();
```

Methods for ArrayList:

```
ArrayList<E>()    // the constructor

// append the specified element to the end of this list
boolean add(E e)

// inserts the specified element at the specified position in this list
void add(int index, E element)

// returns true if this list contains the specified element
boolean contains(E e)

// returns the element at the specified position in this list
E get(int index)

// tests if this list has no elements
boolean isEmpty()

// removes the element at the specified position in this list
// moves all elements at higher indices one position to the left
E remove(int index)

// replaces the element at the specified position in this list
// with the specified element - returns ths original element
E set(int index, E e)

// returns the number of elements in this list
int size()

// returns the index of the first occurrence of the given element
int indexOf(E e)
```

24 Points

Problem 1

The system analysts designed the following classes and interfaces to keep track of inventory information (names of items and their cost:

- `class Pair` that has two fields, a `String name` and a `int cost`.
- `interface PairData` that represents a dataset of `Pair`s and includes the following methods:

```
public void add(Pair p);

public int find(String name);

public void remove(String name, int value);

public int totalPrice();
```

Your job is to implement the `PairData` interface and the class `Pair`. Here is some more information about the design of the interface:

- The `add` method works as follows:

  If a `Pair` with the same name as the given `String` is already in the `PairData` dataset, then it adds the new cost to the cost already recorded in the dataset. Otherwise, it adds a new entry to the dataset with the given cost.

- The `find` method produces `0` if a `Pair` with the same name as the given `String` does not appear in the dataset. Otherwise it returns the cost associated with the given name.

- The `remove` method throws `ItemNotFoundException` (a RuntimeException) if a `Pair` with the same name as the given `String` does not appear in the dataset. It throws `InsufficientFundsException` (a RuntimeException) if a `Pair` with the same name as the given `String` appears in the dataset, but the cost recorded in the dataset is smaller than the given cost. Otherwise, it decreases the recorded cost of that item (or removes it completely if the remaining cost is zero.

- The `totalPrice` method computes the total cost of all `Pair`s in the dataset.
- The cost for any new `Pair` is always a positive number, though this is not enforced.

1. Design the class `Pair` and make examples of five instances of this class that represent inventory in a shop of your choice.

Solution

[POINTS 3: 1 point class definition, 1 point purpose statement, 1 point examples]

```
// to represent an inventory item and its cost
public class Pair{
  String name;
  int cost;

  Pair(String name, int cost){
    this.name = name;
    this.cost = cost;
  }

  /* sample data
  Pair shoes = new Pair("shoes", 300);
  Pair shirts = new Pair("shirts", 100);
  Pair shirtsremoved = new Pair("shirts", 50);
  Pair hats = new Pair("hats", 50);
  Pair socks = new Pair("socks", 20);
  Pair jackets = new Pair("jackets", 200);
  */
}
```

2. Design the class or classes that implement the `PairData` interface.

   - Class header, field declarations, constructor(s), examples of data.

     Solution

     [POINTS 4: 1 point class name and purpose, 1 point field declarations, 1 point constructor, 1 point examples of data]

```
// to represent a collection of Pair data
public class PairDataList implements PairData{

  ArrayList<Pair> data;

  PairDataList(){
    data = new ArrayList<Pair>();
  }

  // add the given pair - or just its cost - to the collection
  public void add(Pair p){
    if (this.find(p.name) > 0)
      this.update(p);
    else
      data.add(p);
  }


// in the Examples class:
  PairData plist1;
  PairData plist2;
  PairData plist3;

  public PairData makeData(Pair one, Pair two, Pair three){
    PairData newlist = new PairDataList();
    newlist.add(one);
    newlist.add(two);
    newlist.add(three);
    return newlist;
  }

// later, inside of testSomething(Tester t){...
//-------------------------------------------------
```

```
plist1 = makeData(shoes, shirts, socks);
plist2 = makeData(shoes, hats, socks);
plist3 = makeData(hats, jackets, shoes);
```

- Methods `add` and `find`

Solution

```
[POINTS 10:  1 point purpose/header for add, 3 points examples/tests,
3 points - body (possible helper method); 1 point purpose/header
for find, 1 point examples for find, 1 point body of method
find]

//--- in the class PairData:
//--------------------------

  // add the given pair - or just its cost - to the collection
  public void add(Pair p){
    if (this.find(p.name) > 0)
      this.update(p);
    else
      data.add(p);
  }

  // helper method: update the given item by the given cost
  public void update(Pair p){
    int i = 0;
    while(i < data.size() && !p.name.equals(data.get(i).name))
      i++;
    data.set(i, new Pair(p.name, data.get(i).cost + p.cost));
  }

  // find the cost of the item with the given name
  // produce 0 if not found
  public int find(String name){
    int i = 0;
    while(i < data.size() && !name.equals(data.get(i).name))
      i++;
    if (i == data.size())
      return 0;
    else
      return data.get(i).cost;
  }

//--- in the Examples class:
//--------------------------
```

```
// Test the method add for the class PairDataList
public void testAdd(Tester t){

  // reset the data before a new test
  plist1 = makeData(shoes, shirts, hats);
  plist = new PairDataList();
  plist.add(shoes);
  plist.add(shirts);
  plist.add(hats);
  t.test("add data", plist, plist1);
}

// Test the method find for the class PairDataList
public void testFind(Tester t){

  // reset the data before a new test
  plist = makeData(shoes, shirts, hats);
  t.test("find shirts", plist.find("shirts"), 100);
  t.test("find socks", plist.find("socks"), 0);
}
```

- Methods remove and totalPrice

Solution

[POINTS 8:1 point purpose/header for remove, 2 points examples/tests, 2 points - body (possible helper method); 1 point purpose/header for find, 1 point examples for find, 1 point body of method find]

```
  // remove the given value from the cost of the item with the given name
  // throw ItemNotFoundException
  //       - if item with the given name is not in the collection
  // throw InsufficientFundsException
  //       - if the cost of the item with the given name
  //         is less that the given value
  // if the remaining value is 0, remove the item from the collection
  public void remove(String name, int value){
    int i = 0;
    while(i < data.size() && !name.equals(data.get(i).name))
      i++;
    if (i == data.size())
      throw new ItemNotFoundException("Item not found");
    else if (data.get(i).cost < value)
      throw new InsufficientFundsException("Insufficient funds");
    else if (data.get(i).cost == value)
      data.remove(i);
    else data.get(i).cost = data.get(i).cost - value;
  }

  // compute the total cost of the entire inventory
  public int totalPrice(){
    int total = 0;
    for (Pair p : data){
      total = total + p.cost;
    }
    return total;
  }

//--- in the Examples class:
//-------------------------

 public void testRemove1(Tester t){
```

```
  // reset the data before a new test
  plist = makeData(shoes, shirts, socks);
  PairData plistrem = makeData(shoes, shirtsremoved, socks);
  plist.remove("shirts", 50);
  t.test("remove - OK", plist, plistrem);
}

public void testRemove2(Tester t){
  // reset the data before a new test
  plist = makeData(shoes, shirts, socks);
  PairData plistrem = new PairDataList();
  plistrem.add(shoes);
  plistrem.add(socks);
  plist.remove("shirts", 50);
  t.test("remove - out", plist, plistrem);
}

public void testRemove3(Tester t){
  // reset the data before a new test
  plist = makeData(shoes, new Pair("shirts", 100), socks);
  t.test("remove - not found", plist,
      "remove",
      new Object[]{"shirts", ((int)150)},
      new InsufficientFundsException("Insufficient funds"));
}

public void testRemove4(Tester t){
  plist = makeData(shoes, new Pair("shirts", 100), hats);
  t.test("remove - not found", plist,
      "remove",
      new Object[]{"socks", ((int)50)},
      new ItemNotFoundException("Item not found"));
}
```

13 Points

Problem 2

You are given two `Traversal<T>` traversals that generate sorted collections of data, with the data sorted according to the given `Comparator<T>`.

1. Design the method `merge` in the `Algorithms` class that consumes the two `Traversal<T>` objects and an object of the type `Comparator<T>` and produces an `ArrayList<T>` of all items from both collections, sorted according to the given `Comparator<T>`. However, there is a catch. Some objects appear in both lists. If that is the case, then only one of them appears in the resulting list.

   For example, if:

   `tr1` traverses over: `"Good Day" "Hello" "So Long"`
   `tr2` traverses over: `"Bye" "Ciao" "Hello" "Hi"`

   the resulting `ArrayList<T>` will contain `"Bye" "Ciao" "Good Day" "Hello" "Hi" "So Long"`

   Solution

   [POINTS 8: 1 point purpose statement, 2 points merge body, 2 points helper methods, 1 point throwing exception, 2 points examples of use with lists of Strings]

```
// Algorithms class: ------------------------------------------------
// Note: methods could be static.
 // merge two sorted lists using the given Comparator
  // eliminate duplicates
  // produce the result as an ArrayList
  public <T> ArrayList<T> merge(Traversal<T> tr1, Traversal<T> tr2, Comparator<T
    ArrayList<T> arrlist = new ArrayList<T>();

    while (!tr1.isEmpty() &&
           !tr2.isEmpty()){
      if (comp.compare(tr1.getFirst(), tr2.getFirst()) < 0){
        arrlist.add(tr1.getFirst());
        tr1 = tr1.getRest();
```

```
    }
    else if (comp.compare(tr1.getFirst(), tr2.getFirst()) > 0){
      arrlist.add(tr2.getFirst());
      tr2 = tr2.getRest();
    }
    else if (comp.compare(tr1.getFirst(), tr2.getFirst()) == 0){
      arrlist.add(tr2.getFirst());
      tr1 = tr1.getRest();
      tr2 = tr2.getRest();
    }
  }
  if (tr1.isEmpty()){
    while (!tr2.isEmpty()){
      arrlist.add(tr2.getFirst());
      tr2 = tr2.getRest();
    }
  }
  else if (tr2.isEmpty()){
    while (!tr1.isEmpty()){
      arrlist.add(tr2.getFirst());
      tr1 = tr1.getRest();
    }
  }
  return arrlist;
}
```

2. Define the data `tr1` and `tr2` where each represents a list of `String`s ordered by their length. The class `String` defines the method `int length()` that computes the length of the `String`.

   Use the `merge` method you designed to merge any two lists of `String`s sorted by their length. Use the data above in your tests.

   Solution

   [POINTS 3: 1 point purpose statement, 1 point for using the method correctly, 1 point for examples for the merge method]

```
// Examples class: ------------------------------------------
  Algorithms algo = new Algorithms();

  // Test the method merge for the class Algorithms
  // The first list to merge
  AList<String> arr1 =
    new ConsList<String>("Hello",
        new ConsList<String>("Servus",
            new ConsList<String>("So Long",
                new ConsList<String>("Good Day", new MTList<String>()))));

  // The second list to merge
  AList<String> arr2 =
    new ConsList<String>("Hi",
        new ConsList<String>("Bye",
            new ConsList<String>("Ciao",
                new ConsList<String>("Good Day", new MTList<String>()))));

  // test the method merge in the class Algorithms
  public void testMerge(Tester t){
    ArrayList<String> result = new ArrayList<String>();
    result.add("Hi");
    result.add("Bye");
    result.add("Ciao");
    result.add("Hello");
    result.add("Servus");
    result.add("So Long");
```

```
  result.add("Good Day");

  t.test("test merge", algo.merge(arr1, arr2, cmp), result);
}

// The Comparator for the merge method test
Comparator<String> cmp = new Comparator<String>(){
  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
};

// test the length comparator for String-s
public void testLengthComparator(Tester t){
  t.test("test length Comparator", cmp.compare("Hello", "Bye") > 0, true);
  t.test("test length Comparator", cmp.compare("Hi", "Bye") < 0, true);
  t.test("test length Comparator", cmp.compare("Ciao", "Ahoy") == 0, true);
}
```

18 Points

Problem 3

The administrator of an online role-playing game manages the game players according to the following rules. The game consists of clans of players that compete or collaborate with each other. Each player can belong to only one group. A player can request to change clans, to leave the game, or to join the game. The administrator keeps track of all clans and of all players.

For example we may have the following current state of the game:

Clan Eagles has Jan, Ted, Ann
Clan Beavers has Bob, Eli, Jim
Clan Falcons has Tom, Jen, Pat

1. Design the classes that represent this data. For each `Player`, there should be a field that represents the name, and the clan the player belongs to. For each `Clan` there should be a name and a list of `Player`s who are its members. The `Administrator` keeps both a lists of players and a list of clans.

   Write the actual class definitions — class diagrams are not sufficient.

   Note: In a real game we would keep more information about each player and about each clan.

(This page is intentionally left blank.)

Solution

[POINTS 11: 3 points for the class Player, 3 points for the class Clan, 2 points for the class Administrator, 3 points for the method that adds a person to the clans list of members.]

```
/**
 * A class to represent a player in a role playing game
 * @author vkp
 */
public class Player{
  String name;
  String cname;

  Player(String name){
    this.name = name;
    this.cname = "none";
  }

  /**
   * move this player to the clan with the given name
   * @param cname the name of the clan to join
   */
  public void moveToClan(String cname){
    this.cname = cname;
  }
}

import java.util.*;

/**
 * A class to represent a warring clan in a role playing game
 * @author vkp
 */
public class Clan{
  String name;
```

```
  ArrayList<Player> players;

  Clan(String name){
    this.name = name;
    this.players = new ArrayList<Player>();
  }

  /**
   * Add a new player to this clan
   * @param p the player to join this clan
   */
  public void addPlayer(Player p){
    p.moveToClan(this.name);
    players.add(p);
  }

  /**
   * Remove the given player from this clan
   * @param p
   */
  public void removePlayer(Player p){
    players.remove(p);
  }
}

import java.util.*;

public class Administrator{
  ArrayList<Player> players;
  ArrayList<Clan> clans;

  Administrator(){
    players = new ArrayList<Player>();
    clans = new ArrayList<Clan>();
  }

  // A player with the given name
  // joins the clan with the given name
  public void moveTo(String pname, String cname){
    // if the player is not in our list, make a new player
```

```
  // else find the player in our list
  Player p = this.getPlayer(pname);

  // if the clan is not in our list, make a new clan
  // else find the clan in our list
  Clan c = this.getClan(cname);

  // find the player's current clan
  // remove the player from the current clan - if not "none"
  if (!(p.cname.equals("none"))){
    this.getClan(p.cname).players.remove(p);
  }

  // and move to the new clan - notifying both the player and the clan
  c.addPlayer(p);
}

// if the player is not in our list, make a new player
// else find the player in our list
protected Player getPlayer(String pname){
  // look for the player in out list and return if found
  for (Player p : this.players){
    if (p.name.equals(pname))
      return p;
  }

  // new player - add to our list and return
  Player p = new Player(pname);
  this.players.add(p);
  return p;
}

// if the clan is not in our list, make a new clan
// else find the clan in our list
protected Clan getClan(String cname){
  // look for the clan in out list and return if found
  for (Clan c : this.clans){
    if (c.name.equals(cname))
      return c;
  }
```

```
    // new clan - add to our list and return
    Clan c = new Clan(cname);
    this.clans.add(c);
    return c;
  }
}
```

2. Show how you would represent the information shown at the beginning of this problem as data in your classes. Assume that these objects are defined in the `Examples class`.

Solution

```
[POINTS 3:  Just examples; circularly referential data - cannot
be instantiated in the constructor.]


//in the Examples class:
  Administrator boss = new Administrator();

  public void makeGame(){
    boss.moveTo("Jan", "Eagles");
    boss.moveTo("Ted", "Eagles");
    boss.moveTo("Ann", "Eagles");
    boss.moveTo("Bob", "Beavers");
    boss.moveTo("Eli", "Beavers");
    boss.moveTo("Jim", "Beavers");
    boss.moveTo("Tom", "Falcons");
    boss.moveTo("Jen", "Falcons");
    boss.moveTo("Pat", "Falcons");
  }
  // test the method getPlayer in the class Administrator
  public void testGetPlayer(Tester t){
    t.test("test getPlayer - exists", boss.getPlayer("Ted"), ted);
    t.test("test getPlayer - not there",
        boss.getPlayer("Ned"),
        new Player("Ned"));
  }

  // test the method getClan in the class Administrator
  public void testGetClan(Tester t){
    Clan result = new Clan("Eagles");
    result.addPlayer(jan);
    result.addPlayer(ted);
    result.addPlayer(ann);
    t.test("test getClan - exists", boss.getClan("Eagles"), result);
    t.test("test getClan - not there",
        boss.getClan("Ravens"),
        new Clan("Ravens"));
```

```
  }


// Produced by makeString(boss):

   new Administrator(
    this.players =
     new java.util.ArrayList(){
       new Player(
        this.name = "Jan"
        this.cname = "Eagles"),
       new Player(
        this.name = "Ted"
        this.cname = "Eagles"),
       new Player(
        this.name = "Ann"
        this.cname = "Eagles"),
       new Player(
        this.name = "Bob"
        this.cname = "Beavers"),
       new Player(
        this.name = "Eli"
        this.cname = "Beavers"),
       new Player(
        this.name = "Jim"
        this.cname = "Beavers"),
       new Player(
        this.name = "Tom"
        this.cname = "Falcons"),
       new Player(
        this.name = "Jen"
        this.cname = "Falcons"),
       new Player(
        this.name = "Pat"
        this.cname = "Falcons")}
    this.clans =
     new java.util.ArrayList(){
       new Clan(
        this.name = "Eagles"
        this.players =
```

```
  new java.util.ArrayList(){
    new Player(
     this.name = "Jan"
     this.cname = "Eagles"),
    new Player(
     this.name = "Ted"
     this.cname = "Eagles"),
    new Player(
     this.name = "Ann"
     this.cname = "Eagles")}),
new Clan(
 this.name = "Beavers"
 this.players =
  new java.util.ArrayList(){
    new Player(
     this.name = "Bob"
     this.cname = "Beavers"),
    new Player(
     this.name = "Eli"
     this.cname = "Beavers"),
    new Player(
     this.name = "Jim"
     this.cname = "Beavers")}),
new Clan(
 this.name = "Falcons"
 this.players =
  new java.util.ArrayList(){
    new Player(
     this.name = "Tom"
     this.cname = "Falcons"),
    new Player(
     this.name = "Jen"
     this.cname = "Falcons"),
    new Player(
     this.name = "Pat"
     this.cname = "Falcons")})})
```

3. Design the method moveTo that updates the Administrator information when a Player joins a new Clan.

   Note: The method should allow a new player to join a clan, an existing player join an existing clan and leave the one he is currently in, or even start a new clan.

   Warning: Follow the Design Recipe carefully.

   Solution

   [POINTS 5: 1 point purpose, 1 point for checking if player already in the list and for checking whether this is a new clan, 1 point body, 2 points tests for all possibilities]

   TBD