Question 1. (22 points) Class specification. One of last summer's interns was working on a generic version of a graph that used an adjacency matrix representation. There's some existing code, but it's not specified or commented properly and it needs some errors fixed. In this problem we'll figure out what it does and clean it up.

Here is the code:

```
public class AdjacencyMatrixGraph<Edge> {
  private Edge[][] matrix;

  public AdjacencyMatrixGraph(int nodeCount) {
    matrix = new Edge[nodeCount][nodeCount];
  }

  public int getNodeCount() {
    return matrix.length;
  }

  public Edge getEdge(int start, int dest) {
    return matrix[start][dest];
  }

  public void addEdge(int start, int dest, Edge e) {
    matrix[start][dest] = e;
  }
}
```

(a) (3 points) The code for the constructor won't compile. What's wrong and how can it be fixed so it works as intended, but without any compiler errors or warnings. (Hints: an annotation – *@Something* – might be useful along with any other repairs needed. Also, remember that when a Java array is allocated with `new` the elements are initialized to `null`, so that's not the problem.)

The error is that we cannot create an array of a generic type (`Edge` in this case). Here is a version that does work, along with the annotation needed to eliminate the compiler warning.

```
  @SuppressWarnings("unchecked")
  public AdjacencyMatrixGraph(int nodeCount) {
    matrix = (Edge[][]) new Object[nodeCount][nodeCount];
  }
```

Question 1. (cont.) (b) (9 points) Write a suitable JavaDoc comment summarizing the class, and inside the class give a suitable *representation invariant (RI)* and *abstraction function (AF)*. The class declaration and instance variable declaration are repeated here, along with space for the JavaDoc comment above the class declaration. Write the RI and AF comments below the instance variable declaration.

```
/**
 *  An AdjacencyMatrixGraph represents a mutable directed
 *  graph with edges labeled with objects of type Edge.
 *  There is at most one directed edge from one particular
 *  node to another.
 *  This representation is particularly effective for dense
 *  matrices since it allows fast O(1) access to edge data
 *  at the cost of requiring O(|V|^2) storage.
 */
public class AdjacencyMatrixGraph<Edge> {
  private Edge[][] matrix;

  // write a suitable RI below

  // matrix is not null, and matrix is a square matrix
  // with size > 0 (i.e., for 0 <= i < matrix.length,
  // matrix[i].length = matrix.length).


  // write the corresponding AF below

  // matrix.length is the number of nodes in the graph.
  // For 0 <= i, j < matrix.length, if matrix[i][j] is
  // null, then there is no directed edge from node i to
  // node j, otherwise matrix[i][j] is the label of the
  // directed edge from i to j.
```

Grading notes:

- **Summary: we did not mark off points if the specification did not mention the efficiency, although that is probably something that would be appropriate in the public description of this class.**
- **Representation invariant: The constructor and `getNodeCount` method will actually work if the matrix size is allowed to be 0, but it will be impossible to meet the preconditions needed to call `addEdge` and `getEdge` successfully. So realistically the matrix size needs to be > 0. But we did not deduct anything if the size was required to be >= 0 instead.**

Question 1. (cont.) (c) (10 points) Give proper CSE331-style JavaDoc specification comments for the `getEdge` and `addEdge` methods of this class. The methods are repeated below for your convenience, with space for your JavaDoc comments.

```
/**
 *  Return the label of the directed edge from node
 *  start to dest, or return null if no such edge.
 *
 *  @param start The origin node of the edge
 *  @param dest  The destination node of the edge
 *  @requires    0 <= start, end < getNodeCount()
 *  @return the Edge label from start to dest if there is
 *          such an edge, otherwise null.
 */
public Edge getEdge(int start, int dest) {
  return matrix[start][dest];
}

/**
 *  Add or modify the edge from start to dest.
 *
 *  @param start The origin node of the edge
 *  @param dest  The destination node of the edge
 *  @param e     The label for the new edge
 *  @requires    0 <= start, end < getNodeCount()
 *  @modifies this
 *  @effects add or replace the edge from start to dest
 *           with one whose label is e.  If e is null,
 *           this effectively removes the edge start->dest
 *           from the graph.
 */
public void addEdge(int start, int dest, Edge e) {
  matrix[start][dest] = e;
}
```

Notes: There are two possible ways to handle a `null` value for `e` in `addEdge`. One is to disallow it by specifying `e!=null` as a precondition (adding it to the `@requires` clause). The other is the solution shown above, which is to allow a `null` value of `e` to be used to remove an edge if one is present. Either solution was acceptable. The `@requires` preconditions need to mention that `start` and `end` are valid node numbers. Strictly speaking this should use an abstract operation like `getNodeCount()`, but we were a bit relaxed in the grading and allowed `matrix.length` instead since this is easy to overlook.

A fairly common error in these specifications was to describe their effect on the matrix rather than to describe operations on graph edges.

Question 2. (16 points) A bit of design. Ima Hacker, a new Java programmer, was asked to create a small Java program for a text-based tic-tac-toe game, where the user plays against the computer. We eventually want to make the game work on a smartphone, but for now we just want a version that works using a keyboard and monitor. Omitting all the details, here's the outline of the program that Ima hacked up one afternoon.

```
/** A tic-tac-toe game */
public class TicTacToe {
  // instance variables omitted

  /** Initialize a new game object */
  public TicTacToe() { }

  /** Print the state of the game board on System.out */
  public void printGame() { }

  /** Read the player's next move from the keyboard
    * and update the game to reflect that move.      */
  public void getPlayerMove() { }

  /** Calculate the computer's next move and print it on
    * System.out */
  public void computerMove() { }

  /** Calculate the computer and user's current scores
    * and print them on System.out */
  public void printScores() { }

  /** Reset the game back to the same initial state it had
   *  when it was created */
  public void resetGame() { }
}
```

While this set of methods contains all the operations we want for the initial game, the design has problems.

(Question continued on the next page. You may remove this page for reference while working on it if convenient.)

Question 2. (cont.) (a) (4 points) What is (are) the major design flaw(s) in the above design? A brief couple of sentences should be enough to get the point across.

The major design flaw is that the game logic and user interface have been jumbled together in a single class. These need to be separated, particularly if we ever hope to be able to replace the user interface with a different one without a major rewrite.

(b) (12 points) Describe how you would refactor (change) the initial design to improve it. Briefly sketch the class(es) and methods in your design, and what each of them do. You do not need to provide very much detail, but there should be enough so we can tell how you've rearranged the design, what the major pieces are, and how they interact.

(There's additional space on the next page if you need it for your answer.)

The important thing that the application should be split using the model-view-controller pattern to separate the game logic from the interface.

Model: game logic. Operations would include the following from the original code:

- **Create a new game model**
- **Record a new player move**
- **Calculate a new move for the computer (probably in a separate module/object from the core ADT storing the game state)**
- **Return the current user and computer scores to the caller**
- **Return the current state of the game board to the caller**
- **Reset the game board to start a new game**

Viewer/Controller (likely a single module in a console text-based game):

- **Process initialize and reset commands from the user**
- **Read player moves and call the model to update the game state**
- **Display computer moves**
- **Display current contents of the game board and current scores**

The viewer/controller and the model would likely use some form of the Observer pattern to communicate.

There are obviously many possible answers to this question. The important thing we were looking for was a sensible overall organization and partitioning of the functions between modules.

Question 3. (8 points) A rather generic question. The following method performs a binary search for an integer value *x* in a sorted array of integers.

```
/** Search a sorted array ~~of integers*~~ for a given value
  * @param a Array to be searched
  * @param x Value to search for in array a
  * @return If x is found in a, return largest i such
  *         that a[i]==x. Return -1 if x not found in a
  * @requires a!=null & a is sorted in non-decreasing
  *          order (i.e., a[0]<=a[1]<=...<=a[a.length-1])
  */
public static <T extends Comparable<T>>
                          int bsearch(~~int~~ T[] a, ~~int~~ T x) {
  int L = -1;
  int R = a.length;
  // inv: a[0..L] <= x && a[R..a.length-1] > x &&
  //      a[L+1..R-1] not examined
  while (L+1 != R) {
    int mid = (L+R)/2;
    if (~~a[mid] <= x~~ a[mid].compareTo(x) <= 0)
      L = mid;
    else  // a[mid] > x
      R = mid;
  }
  if (L >= 0 && ~~a[L] == x~~ a[L].compareTo(x) == 0)
    return L;
  else
    return -1;
}
```

We would like to modify this code to change it into a generic method that works with any sorted array whose elements have type `Comparable<T>` (i.e., the elements can be ordered using the `compareTo` method). Mark the code above to show the changes needed to turn this into a generic method.

Changes shown in the code above. * = no deduction if the heading comment was left unmodified and still refers to an integer array (that's easy to miss and not the point). The test at the end could have been written as `a[L].equals(x)`, which is equivalent to checking that `compareTo` returns 0.

Question 4. (8 points) There's something fishy about this question. Suppose we have the following class hierarchy:

```
class Creature extends Object { }
class Fish extends Creature {
  /** Return the weight of this Fish */
  public float getWeight() { return ...; }
}
class Salmon  extends Fish { }
class Haddock extends Fish { }
class Tuna    extends Fish { }
```

Class `Creature` does not have a `getWeight` method. Class `Fish` implements that method and all of its subclasses inherit it.

Write a `static` method `collectionWeight` whose parameter can be any Java `Collection` containing `Fish` objects (including objects of any `Fish` subclass(es). Method `collectionWeight` should return the total weight of all the `Fish` in the `Collection`, using `getWeight` to determine the weight of each individual `Fish`.

Hints: Method `collectionWeight` will need a proper generic type for its parameter. Java `Collection`s are things like `List`s and `Set`s that implement the `Iterable` interface. They do not include things like `Map`s, which are not simple collections of objects.

```
public static
float collectionWeight(Collection<? extends Fish> f) {
  float totalWeight = 0;
  for (Fish fish: f) {
    totalWeight += fish.getWeight();
  }
  return totalWeight;
}
```

The heading of the method can also be written as

```
public static <T extends Fish>
float collectionWeight(Collection<T> f) ...
```

Question 5. (7 points) Testing. (a) (3 points) There are many metrics we can use to try to evaluate how well a test suite does its job. One common metric is *code coverage*. If a test suite achieves 100% code coverage that means that every statement in the code being tested was executed by at least one test.

True or false: 100% code coverage is sufficient to guarantee that if an error is present in the code then it will be detected. If your answer is *true* give a brief justification. If your answer is *false*, give an example that shows why 100% code coverage is not sufficient.

False. An example similar to one from the lecture slides is:

```
int min(int a, int b) {
   int ans = a;
   if (a <= b)
      ans = b;
   return a;
}
```

The test `min(1,2)` executes every statement but misses the bug.

(b) (2 points) Give one advantage that black box tests have compared to white (clear, glass) box tests. Be brief.

Some possibilities:

- **Test code less likely to be biased by implementation details of the code being tested.**
- **Tests can be used with different implementations.**
- **Allows for tests to be developed independently of, and possibly before, the implementation.**

(c) (2 points) Give one advantage that white (clear, glass) box tests have compared to black box tests. Be brief.

The main advantage is that we can test implementation details that are not part of the specification. Examples: caching behavior, automatic expansion of data structures when their initial capacity is exceeded.

Some shorter questions.

Question 6. (8 points) Subtyping. Suppose we have a class `A` with a method `m`:

```
class A {
  public T m(S x) { ... }
}
```

Now suppose we create a class `B` that is a subclass of `A` with its own method `m` that is supposed to override the one from class `A`:

```
class B extends A {
  public T1 m(S1 x) { ... }
}
```

(a) (4 points) If we want class `B` to be a true specification subtype of class `A`, what are the possible relationships between `m`'s types `T` and `S` in `A` and types `T1` and `S1` in `B`? Do the types have to match exactly or can one type be a specification (true) subtype or supertype of the other? Remember, this part of the question is asking about the typing rules for true specification subtypes, which might (or might not) be the same as Java's subclass rules.

Both of these must hold:

- **`T1` is the same as, or is a subtype of `T`.**
- **`S1` is the same as, or is a supertype of `S`.**

(b) (4 points) Are Java's rules for subclass and method subtypes the same as the specification subtyping rules in part (a)? If so, it's sufficient to just say that they are the same. If not, what's the difference, and why are the Java rules different? (Be brief)

Java allows result types to be covariant, i.e., `T1` can be a subtype of `T`. However, parameter types must be the same. If the parameter type in a subclass is different, then the subclass method overloads the one from the superclass instead of overriding it.

Question 7. (9 points) The implementation of `equals` in class `Object` returns true if two objects are exactly the same, i.e., `a.equals(b)` returns the result of the comparison `a==b`.

(a) Show that this definition of `equals` defines a proper equivalence relation (i.e., show that this implementation satisfies the properties required of an equivalence relation). A brief answer should be sufficient (and, yes, it's not tricky or complicated).

An equivalence relation must be reflexive, symmetric, and transitive. All of these are trivially true if equivalence is defined as object equality:

- **`a==a` always.**
- **If `a==b`, then `a` and `b` refer to the same object, so `b==a`.**
- **If `a==b` and `b==c`, then all three refer to the same object, so `a==c`.**

(b) The actual JavaDoc specification of `equals` in `Object` is fairly complex, enumerating many properties that an equivalence relation should have. Why didn't the specification for `a.equals(b)` in `Object` simply state that it returns the result `a==b`?

If `Object.equals` was specified to be object equality then no subclass could weaken that requirement. It would not be possible to treat two different objects as equal, even if they had the same abstract value.

(c) The actual implementation of `hashCode` in `Object` returns the memory address of the object. Explain why this implementation of `hashCode` satisfies the necessary properties required of a `hashCode` implementation.

The requirement on `hashCode` is that if `a.equals(b)` then it must be true that `a.hashCode()==b.hashCode()`. If `a.equals(b)` according to the `equals` method in `Object`, then we know that `a==b`. That means `a` and `b` have the same memory address, so they both have the same `hashCode()` value using `Object`'s implementation of `hashCode`.

Question 8. (7 points) A couple of questions on usability:

(a) (2 points) If several events occur fast enough, a person perceives them as being a single event. How fast is "fast enough"? Circle the longest time interval that can generally occur between two events for them to be perceived as happening at the same time:

10 sec. 5 sec. 2 sec. 1 sec. 0.5 sec. (0.1 sec.) 0.01 sec. 0.001 sec.

(b) (2 points) The complexity of a user interface should often be limited by the number of things that a typical person can hold at once in their short-term working memory. What is the maximum number of things that a typical person can hold in their short-term memory? (circle)

20 15 12 (7) 3 1

(c) (3 points) A strategy that initially seemed like a good idea was to present the user with a "confirmation dialog" requiring an additional approval before some irreversible action is performed. For example, the dialog might say "delete file?" or "launch missiles?" and require the user to click either yes or no. Why does this turn out not to be particularly effective in practice?

For common or frequently used operations, users tend to "chunk" acknowledging the confirmation dialog with the action that produces it. The sequence becomes a single action done without thinking separately about the confirmation step.

Question 9. (7 points) System design and implementation.

(a) (3 points) Why force everyone on a project to use a build tool like `make` or `ant`? IDEs like Eclipse have built-in tools to do this, why not let people use their IDE's build tools if they have them?

It is important that everyone on a project use the same build tools so everyone gets the same, reproducible build results.

A large project can be built either bottom-up starting with modules that do not depend on others and that provide infrastructure for modules higher up, or top-down, starting at the top layer and adding lower-level modules as the project progresses.

(b) (2 points) Give one advantage of a top-down development strategy over a bottom-up one. (But be brief about it!)

Problems with global design and implementation decisions can be found earlier in the project, when they are (much) less expensive and easier to fix.

There is usually better visibility. Clients can see that progress is being made and partially-built systems are easier to demonstrate and try earlier.

(c) (2 points) Give one advantage of a bottom-up development strategy over a top-down one. (And be brief about this, too!)

Efficiency and feasibility problems tend to get uncovered earlier. For example, it is easier to discover that the hardware or infrastructure won't be able to provide the required performance or satisfy other required constraints.

Bottom-up can require building less non-deliverable scaffolding code, although this is not necessarily a major advantage.

Question 10. (8 points) Design patterns. For each of the following design patterns, give a brief explanation of the design problem that it solves and an example of a situation where it would be appropriate to use the pattern. For example,

Singleton: we want to insure that only one instance of a class is created. Examples would be to ensure there is only one random number generator shared throughout a program, or there is only one instance of a class that controls a particular printer or other device.

(a) Observer

Reduce the coupling between objects that produce information and objects that need to examine them. The observed object does not need to know the actual identity of the observers. Examples are views in MVC architectures.

(b) Interning

Ensures that there is only one copy of each abstract value of a class stored in the system. Avoids the costs of constructing multiple copies and the storage and garbage collection expenses of managing them. Also can allow faster comparison of abstract values using == identity tests instead of `equals()`. Examples are interning strings or Boolean values to avoid multiple copies.

(c) Visitor

Provides a standard way to traverse a hierarchical data structure with the correct method being used to process each node in the structure depending on its actual type. There are many examples such as traversals of abstract syntax trees in compilers, processing structured documents, and so on.

(d) Factory

Hides details of object creation from client code. Examples are code that can operate on specialized representations such as sparse or dense matrices. We would like to isolate the choice of representation to Factory methods or objects so the rest of the code is independent of these decisions.

Have a great winter break! See you in January!!