Java Driving Test

Tuesday 21st March 2017

12:30 – 15:30

THREE HOURS

(including 10 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time, log in using your username as **both** your username and password.

There are **TWO** sections: Section A and Section B, each worth 50 marks.

Credit will be awarded throughout for code that successfully compiles, which is clear, concise, usefully commented, and has pre-conditions expressed with appropriate assertions.

Important note:

- In each section, the tasks are in increasing order of difficulty. Manage your time so that you attempt both sections. You may wish to solve the easier tasks in both sections before moving on to the harder tasks
- It is critical that your solution compiles for automated testing to be effective. **TEN MARKS** will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave.
- You can use the terminal or an IDE like IDEA to compile and run your code. **Do not ask an invigilator for help on how to use an IDE.**
- Before you log out at the end of the test, you **must** ensure that your source code is in the correct directory otherwise your marks can suffer heavy penalties. Only code in the original directories provided will be checked.

Section A

Problem Description

You will write a number of classes that implement a `Cell` interface. A `Cell` is a wrapper for an object reference, and allows the reference to be retrieved, and optionally to be set to a different reference. The `Cell` interface is *generic*, so that a `Cell` can be a wrapper for references of any class or interface type.

For brevity, we refer to the object reference stored in a cell as the *value* of the cell, but you should remember that this "value" is in fact a reference to an object.

You will first implement two basic kinds of cell: a *mutable* cell, whose value can be changed, and an *immutable* cell, whose value is fixed on construction and cannot be subsequently changed.

You will then implement a more sophisticated kind of mutable cell that supports *backup*: it is capable of saving and restoring previously-stored values.

The final part of Section A is to implement a *comparator* for backed-up cells, described in detail below.

Getting Started

The skeleton files are located in the `cells` package. This is located in your Lexis home 'ccd' directory, under:

```
~/ccd/SectionA/src/cells
```

During the test you will need to make use of (though should **NOT** modify) the following:

- `Cell.java`: a generic interface describing operations on a cell
- `BackedUpCell.java`: an interface extending `Cell` to describe the extra operations associated with a backed-up cell
- `IntegerComparator.java`: an implementation of the `Comparator<T>` interface for the `Integer` class
- `StringComparator.java`: an implementation of the `Comparator<T>` interface for the `String` class

You should not modify these files in any way: auto-testing of your solution depends on the files having exactly their original contents.

During the test, you will create a number of classes of your own as described in the instructions.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed in the `cells` package.

Testing

The `Tests.java` class, which will not be marked, contains four JUnit tests to help you gauge your progress during Section A. The tests in this file are commented out initially. **As you progress through the exercise you should un-comment the test associated with each question in order to test your work.**

You can also add extra tests in this file as you see fit, to help you develop your solution.

What to do

1. **Implementing a `MutableCell` class.**

   The `Cell` interface, in `Cell.java`, describes operations that should be supported by a cell storing values of some arbitrary reference type `T`.

   Add a new class, `MutableCell`, implementing the `Cell` interface. This cell implementation should allow the value of a cell to be modified. Like the `Cell` interface, `MutableCell` should be generic.

   You should maximise encapsulation in the design of this class.

   - `MutableCell` should have a field of type `T` representing the value stored in the cell.
   - `MutableCell` should have a constructor that takes no parameters and sets the value to `null`.
   - `MutableCell` should also have a constructor that takes one parameter, with type `T`. If the parameter is `null`, an `IllegalArgumentException` (which is an *unchecked* exception) should be thrown. Otherwise the parameter should be used to initialise the cell value.
   - The `set` method of `Cell` should be implemented. This method should throw an `IllegalArgumentException` if its parameter is `null`. Otherwise, the value of the cell should be updated using the given parameter.
   - The `get` method of `Cell` should be implemented, returning the cell value.
   - The `isSet` method `Cell` should be implemented, returning *false* if and only if the cell value is `null`.

   Un-comment the `testQuestion1()` test in `Tests.java` and and debug your solution until this test passes.

   [**8 marks**]

2. **Implementing an `ImmutableCell` class.**

   Add another generic class, `ImmutableCell`, implementing the `Cell` interface. This cell implementation requires that a cell is initialised with a non-null value, which cannot subsequently be changed.

   You should maximise encapsulation in the design of this class.

   - `ImmutableCell` should have a field of type `T` representing the value stored in the cell.

- `ImmutableCell` should have a single constructor that takes one parameter, with type `T`. If the parameter is `null`, an `IllegalArgumentException` should be thrown. Otherwise the parameter should be used to initialise the cell value.
- The `set` method of `Cell` needs to be implemented to satisfy the requirements of the interface. It is illegal to invoke `set` on an `ImmutableCell`, therefore this method should simply throw an `UnsupportedOperationException` (which is an *unchecked* exception).
- The `get` method of `Cell` should be implemented, returning the cell value.
- The `isSet` method `Cell` should be implemented, returning *false* if and only if the cell value is `null`.
- Override the `equals` method from the `Object` class to provide a notion of equality between `ImmutableCell`s: two `ImmutableCell`s are regarded as equal if and only if their associated values are regarded as equal by the `equals` method from `Object`.

Un-comment the `testQuestion2()` test in `Tests.java` and debug your solution until this test also passes.

[**8 marks**]

3. **Implementing a `BackedUpMutableCell` class.**

Your task is now to write a generic subclass of `MutableCell` that supports backing up of previously-stored references. Your class should provide two modes of operation: *bounded backup*, in which a bounded number of previous cell values are tracked, up to a given limit, with older values being lost; and *unbounded backup*, in which all previous values for the cell are tracked.

You should maximise encapsulation in the design of this class, and should maintain encapsulation of the `MutableCell` super-class as much as possible.

- As well as extending the `MutableCell` class, your `BackedUpMutableCell` class should implement the `BackedUpCell` interface, in `BackedUpCell.java`, which specifies two additional operations that backed-up cells must support.
- `BackedUpMutableCell` should maintain an ordered sequence of values that have been previously stored by the cell.
- `BackedUpMutableCell` should provide a constructor taking no parameters. When constructed in this way, the cell value should be `null`, the sequence of previously-stored values should be empty, and the cell should use *unbounded backup* mode.
- `BackedUpMutableCell` should also provide a constructor taking one `int` parameter, `limit`, representing the backup limit. An `IllegalArgumentException` should be thrown if `limit` is negative (a limit of 0 is acceptable, but means that no backups will be recorded). When constructed in this way, the cell value should be `null`, the list of previously-stored values should be empty, and the cell should use *bounded backup* mode with `limit` as the backup limit.
- Override the `set` method from `MutableCell` so that if the cell is already set, its current value is added to the sequence of backed-up values before its value

is updated. In *bounded backup* mode, the *oldest* backed-up value should be discarded if the backup limit is reached.

- The hasBackup method of BackedUpCell should be implemented, returning *false* if and only if the sequence of backups is empty.
- The revertToPrevious method of BackedUpCell should be implemented. If the sequence of backups is empty then an UnsupportedOperationException should be thrown. Otherwise, the cell value should be changed to the most recently backed-up value, which should itself be removed from the sequence of backups.

Un-comment testQuestion3() in Tests.java and debug your solution until this test also passes.

[**14 marks**]

4. **Implementing a comparator for BackedUpCell.**

The Comparator<T> interface, from the java.util package, specifies a compare method with the following signature:

```
int compare(T first, T second);
```

The method is intended to return a negative value if first is "less than" second, a positive value if second is "less than" first, and 0 otherwise (indicating that first and second are equal from the point of view of the comparator). An implementation of Comparator<T> defines what "less than" means for a particular type T.

For reference, you are provided with two example Comparator implementations:

- IntegerComparator (in IntegerComparator.java), and
- StringComparator (in StringComparator.java).

Each of these classes implements Compare using the compareTo methods that are available for Integers and Strings.

Your task is to provide a comparator implementation for BackedUpCell. Specifically, you should implement a comparator that can compare two objects implementing the BackedUpCell interface, itself using a given comparator in order to compare the contents of cells.

Note: you should not override the equals method of Object in your solution to this question.

- Write a generic class, BackedUpCellComparator<U> that implements the Comparator<T> interface, substituting T for the type BackedUpCell<U>.
- Your BackedUpCellComparator<U> class should have a single field, valueComparator with type Comparator<U>, and should have a constructor that accepts a parameter of this type in order to initialise the field.
- You should implement the method:

  ```
  public int compare(BackedUpCell<U> a, BackedUpCell<U> b)
  ```

  to define what it means for one BackedUpCell<U> to be less than another. The rules for this are as follows.

  - A cell that is not set is *less than* a cell that is set.
  - Otherwise, cells are ordered based on the values they store. If the cells store equal values, the cells are ordered based on their most recently backed-up values. If these are also equal, the cells are ordered based on their second-most recently backed-up values, etc. Comparison of stored and backed up values should be made using the `valueComparator` comparator.
  - If cells `c` and `d` match on stored values and backups, except that `d` has *further* backed-up values beyond those held by `c`, then `c` is *less than* `d`.
  - Two cells are equal if neither cell is set, or if both cells are set to matching values and have an equal number of matching backups.

- Your `compare` method should *not* assume that it is working specifically with `BackedUpMutableCell`s; the comparator should work with any class implementing the `BackedUpCell` interface, and thus should only rely on operations available via this interface.
- It is acceptable for your `compare` method to invoke operations that might modify `first` and/or `second` while the comparison is being made, but `first` and `second` should be restored to their original states before `compare` returns.

Un-comment the `testQuestion4()` test in `Tests.java` and debug your solution until this test also passes.

[**20 marks**]

Total for Section A: 50 marks

Useful commands

```
cd ~/ccd/SectionA

mkdir out

javac -g -d out -cp /usr/share/java/junit4.jar -sourcepath src:test
src/cells/*.java test/cells/Tests.java

java -cp /usr/share/java/junit4.jar:out org.junit.runner.JUnitCore cells.Tests
```

Section B

Problem Description

Collision detection is an essential part of most video games. It consists of determining whether a given set of objects contains pairs of objects that may collide. A naïve implementation of collision detection can be computationally very expensive. For instance, in the case of 100 objects, to check each pair of objects for collision would require about 5000 operations. In the case of 2D-objects, an abstract data type called `QuadTree`, can be used to speed up considerably the execution of a collision detection task.

Informally, a `QuadTree` is a tree structure in which each non-leaf node has exactly four children. Each node represents a 2D **region**, and each of its four children represents a quadrant of its region. A 2D-object is said to be covered by a node if its center is within the region of that node (i.e., the region covers the center). However, only leaf nodes can actually store 2D-objects, and they have the same **nodeCapacity**. When a leaf node covers more than the **nodeCapacity** number of 2D-objects, it spawns four children, referred as NorthWest (**NW**), NorthEast (**NE**), SouthEast (**SE**) and SouthWest (**SW**). Each child represents a quadrant of its parent´s region. In addition, the parent´s 2D-objects are distributed to its children according to their regions. This process goes on until each leaf node contains at most the **nodeCapacity** number of 2D-objects.

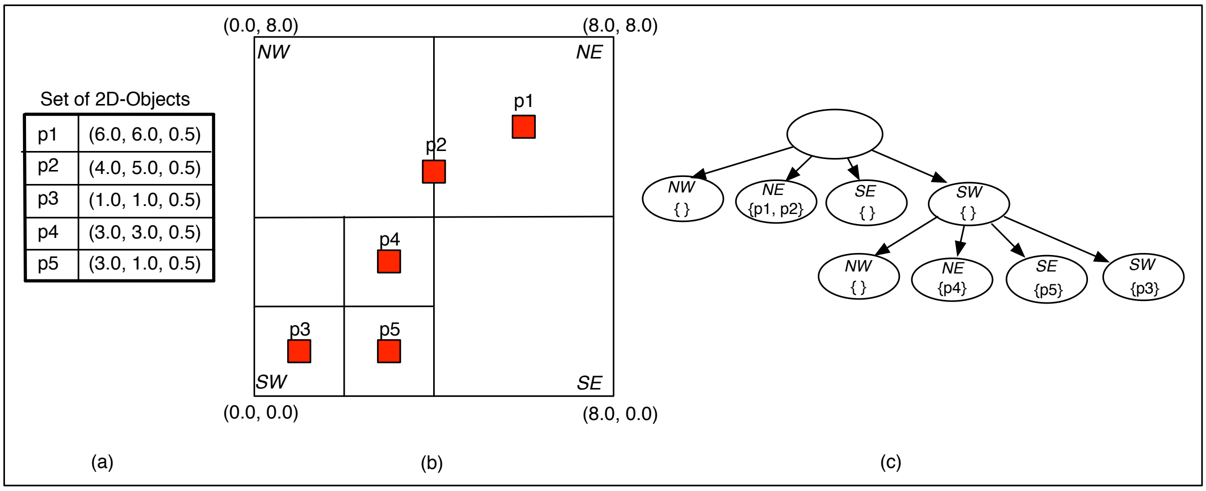

Figure 1: Example of QuadTree representation of set of 2D-Objects in the area 8.0 x 8.0

Figure 1: Example of `QuadTree` representation of set of 2D-Objects in the area 8.0 x 8.0

Note that if a 2D-objects center lays on the boundary between two children´s regions, then the child with higher **rank** contains it. The ranking of the children are **NE** >**NW** >**SE** >**SW**. For instance, Figure 1(a) above shows a set of five 2D-objects (p1-p5), each with coordinates x and y, given by the first two elements in the tuple, as its centre, and size equals to 0.5, represented by the third element of the tuple. Figure 1(b) shows a 8.0 x 8.0 square region, and a **nodeCapacity** equals to 2. The five objects cannot be assigned all to the main given region (since 5 >2). The region is therefore divided into four quadrants. 2D-object p1 falls within the **NE** quadrant. 2D-object p2 also falls within the **NE** quadrant, because its centre is on the boundary between **NE** and **NW**, but **NE** has higher rank than **NW**. The 2D-objects p3 to p5 fall within the **SW** quadrant. But in this case the **SW** quadrant is also divided into four quadrants, as it cannot contain 3 objects (3 >2), and the three objects (p3-p5) are then distributed to their respective sub-quadrants of the **SW** region that cover them. The `QuadTree` representation is given in Figure 1(c). To query what 2D-objects in a `QuadTree` are covered by a given region (i.e. the region covers their centers), it can be done efficiently by searching down the `QuadTree` and considering, for each non-leaf node, only the children whose regions overlap with the queried region.

Collision detection for a given set of 2D-objects can then be done efficiently using a `QuadTree` data structure. For each 2D-object a **safetyRegion** is considered. This is a square centred at the object and with width equals to twice the size of the object. For instance, the **safetyRegion** of a 2D-object (7.0, 7.0, 0.5) would be the region bounded by the fours 2D coordinates (6.5, 6.5)-(7.5, 6.5)-(7.5, 7.5)-(6.5, 7.5). So to check for collisions, the 2D-objects from the given set are considered, one by one, in the order of increasing sizes. For each 2D-Object, we query the `QuadTree` for stored 2D-objects that are covered by its **safetyRegion**. If none is found, then this 2D-object must not collide with any of the stored objects, and hence it can be added to the tree. If at least one 2D-object is found then a collision is detected. If all the given 2D-objects are successfully added to the `QuadTree`, then these objects are said to be collision-free.

QuadTree Data Structure

The `QuadTree` data structure includes a **root** and a **nodeCapacity**. The **root** is a reference variable of type `QuadTreeNode`. A `QuadTreeNode` includes the **region** it covers, the **values**, a (possibly empty) list of 2D-objects covered by that region, and four reference variables of type `QuadTreeNode` for the four quadrants the node´s region can be divided into (see Figure 2). Note that non-leaf nodes (Figure 2(a)) must have exactly four (not null) children and an empty list of 2D-objects. Leaf nodes (Figure 2(b)) have all four children being null and a list of no more than **nodeCapacity** stored 2D-objects.

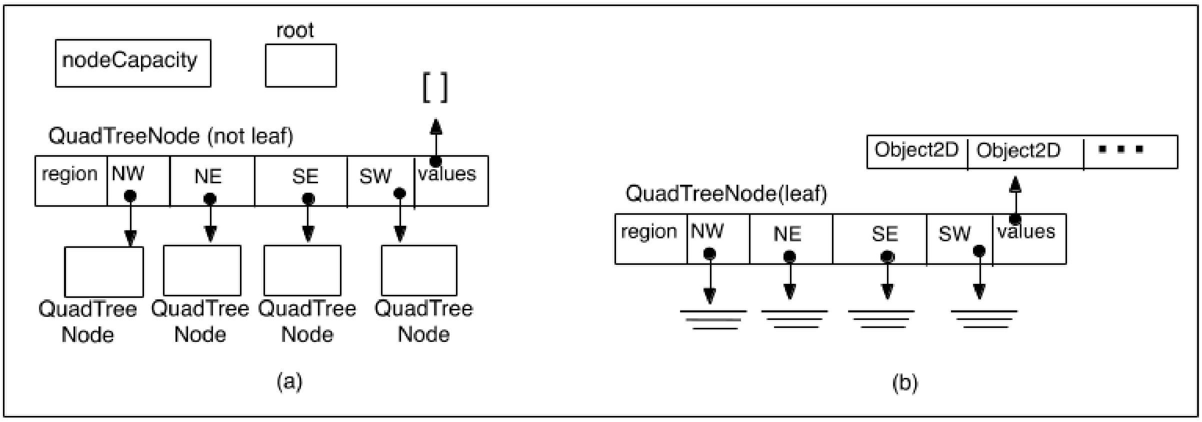

Figure 2: QuadTree Data Strutures

The following operations are supported by the `QuadTree` data structure:

- `add` takes a 2D-object and adds it to the tree. The method starts from the **root**. If the current node is not a leaf, then the 2D-object is passed to the child whose region covers the object (considering also the ranking). If the current node is a leaf and it has less than **nodeCapacity** stored objects, the 2D-object is simply added to the current node´s **values** list. Otherwise, the current leaf node `subdivides` into four children, and all the 2D-objects in the current node´s values list, plus the new one, are passed to these four children accordingly. Finally, the current node (no longer being a leaf) empties its **values** list.

- `contains` takes a 2D-object and searches for the object in the tree. If it is found, it returns true, otherwise it returns false (see code provided).
- `queryRegion` takes an `AABB` region and returns a list of all the 2D-objects in the tree that are covered by that region. This is done by recursively going down from the **root** of the tree through all the children whose regions overlap with the given region. A region A overlaps with another region B if A `covers` any of the corners of B (i.e., top-left, bottom-left, bottom-right and top-right). When a leaf node is reached (i.e., its region overlaps with the given region), then only the 2D-objects stored at this leaf node that are covered by the given region are collected.

Your task is to complete the provided implementation of the `CollisionDetection` in both a sequential and parallel way.

Getting Started

The files used in Section B can be found in the `SectionB` subdirectory in your Lexis home 'ccd' directory:

`~/ccd/SectionB/src/`

Also a UML class diagram describing the architecture of the system is provided (See Figure 3, page 12).

During Section B, you will be working on the following files:

- `CollisionDetection.java`: the class which reads in input a file of 2D-objects and performs the collision detection task. It first orders the 2D-objects in ascending order with respect to their size, using the given `PriorityQueue`, then checks the 2D-objects for collision, as described above.
- `ParallelCollisionDetection.java`: same structure of CollisionDetection but requiring a parallel implementation where three threads perform the checks concurrently.
- `PriorityQueue.java`: the generic class `PriorityQueue<T>` that is used to store the 2D-objects given in the input file.
- `QuadTree.java`: the class that implements the given `QuadTreeInterface`. This class uses auxiliary methods for supporting the recursive implementation of its three methods.

You may also need to make use of (though any change to their interface, if needed, must be back-compatible! i.e., you can add methods or change the modifiers for existing methods and fields only if the change is compatible with the provided implementation) the following:

- `Point2D.java`: the class that stores the x and y coordinates of a 2D-object's center.
- `Object2D.java`: the class that includes as attribute **center**, of type `Point2D`, and the attribute size that defines the **size** of the object.

- `AABB.java`: the class that defines a region, aligned with the axis. It includes four attributes **left**, **right**, **top** and **bottom**. The coordinates of the four corners of a region are then given by pairwise combinations of these attributes. For instance, the coordinate of the top left corner of a region is given by the pair (**left**, **top**), etc.
- `QuadTreeNode.java`: the class that includes an attribute **region**, of type `AABB`, defining the region represented by the node, **values**, a list of 2D-objects covered by the region and stored in the node, and four `QuadTreeNode` reference attributes, **NW**, **NE**, **SE**, **SW**.

What to do

As mentioned before, your task is to complete the provided implementation of the `CollisionDetection`. In addition to the specification, you should carefully read the commented source files. You should write the following methods:

1. The `remove()` method for the class `PriorityQueue<T>` together with its auxiliary method `PQRebuild(int root)`. The `PriorityQueue<T>` is assumed to be implemented as a minHeap ADT.

   [**13 marks**]

2. The `add(Object2D elem)` method for the class `QuadTree` that adds a given 2D-object `elem` to a `QuadTree`. Provide also the implementation of the auxiliary method `addHelper(QuadTreeNode node, Object2D elem)`.

   [**9 marks**]

3. The `queryRegion(AABB region)` method for the class `QuadTree` that returns a list of all the 2D-objects in the `QuadTree` that are covered by the region given in input. Provide also the implementation of the auxiliary method
   `queryRegionHelper(QuadTreeNode node, AABB region, ListInterface<Object2D> bucket)`.

   [**9 marks**]

4. The static method `checkObjects` for the class `CollisionDetection`. This method takes as input a priority queue of 2D-objects, which are already sorted in ascending order with respect to their size, and a region and returns *true* if and only if all the given 2D-objects can be added to the `QuadTree` without any collision. This method creates a `QuadTree` with a **nodeCapacity** of 4 and follows the collision detection algorithm described above.

   [**7 marks**]

5. The static method `checkObjects` for the class `ParallelCollisionDetection`.

   This method has to be functionally equivalent to `CollisionDetection.checkObjects` but it has to spawn three threads that check for collisions in parallel. To be functionally equivalent to its sequential implementation means that the two methods *systematically* return the same value for the same input.

   If you think it is necessary to modify the implementation of PriorityQueue, AABB and QuadTree, you can do that, but the new versions have to be compatible with the current skeleton not to fail the tests (i.e., you can add methods or change the modifiers of existing fields and methods only if your changes keep the compatibility with the skeleton declarations).

[8 marks]

6. Discussion of alternative parallelization strategies. Add a brief comment at the end of the method `ParallelCollisionDetection.checkObjects` to discuss alternative, valid parallelization strategies and compare them with yours. Your comment must be at most 500 words and it is not expected to describe all the possible alternatives but a careful selection of them for the purpose of highlighting the advantages and disadvantages of your choices. (*Hint: You can use a text editor to count the words, or copy the comment in a text file and use wc -w filename to obtain a word count*).

   [4 marks]

   Total for Section B: 50 marks

Useful commands

To help you test your implementation, a few input files for the `CollisionDetection` and `ParallelCollisionDetection` classes containing 2D-objects are provided in the directory tests in the `SectionB` directory in your Lexis home directory. Each file name in the tests subfolder has a suffix of either with-collision or no-collision, which indicates whether the given set of 2D-objects has a collision or not. Examples follow.[1]

```
cd ~/ccd/SectionB/src

javac *.java

java CollisionDetection ../tests/points100-no-collision.txt
java CollisionDetection ../tests/points100-with-collision.txt
java CollisionDetection ../tests/points-no-collision.txt
java CollisionDetection ../tests/points-with-collision.txt
java CollisionDetection ../tests/simple-no-collision.txt
java CollisionDetection ../tests/simple-with-collision.txt

java ParallelCollisionDetection ../tests/points100-no-collision.txt
java ParallelCollisionDetection ../tests/points100-with-collision.txt
java ParallelCollisionDetection ../tests/points-no-collision.txt
java ParallelCollisionDetection ../tests/points-with-collision.txt
java ParallelCollisionDetection ../tests/simple-no-collision.txt
java ParallelCollisionDetection ../tests/simple-with-collision.txt
```

[1] You can also create your own input files for testing purposes. A valid input file should contain N lines of N 2D-objects. Each line is a comma separated triple x,y,z where (x,y) is the 2D coordinates of the 2D object and z is the size of the object.

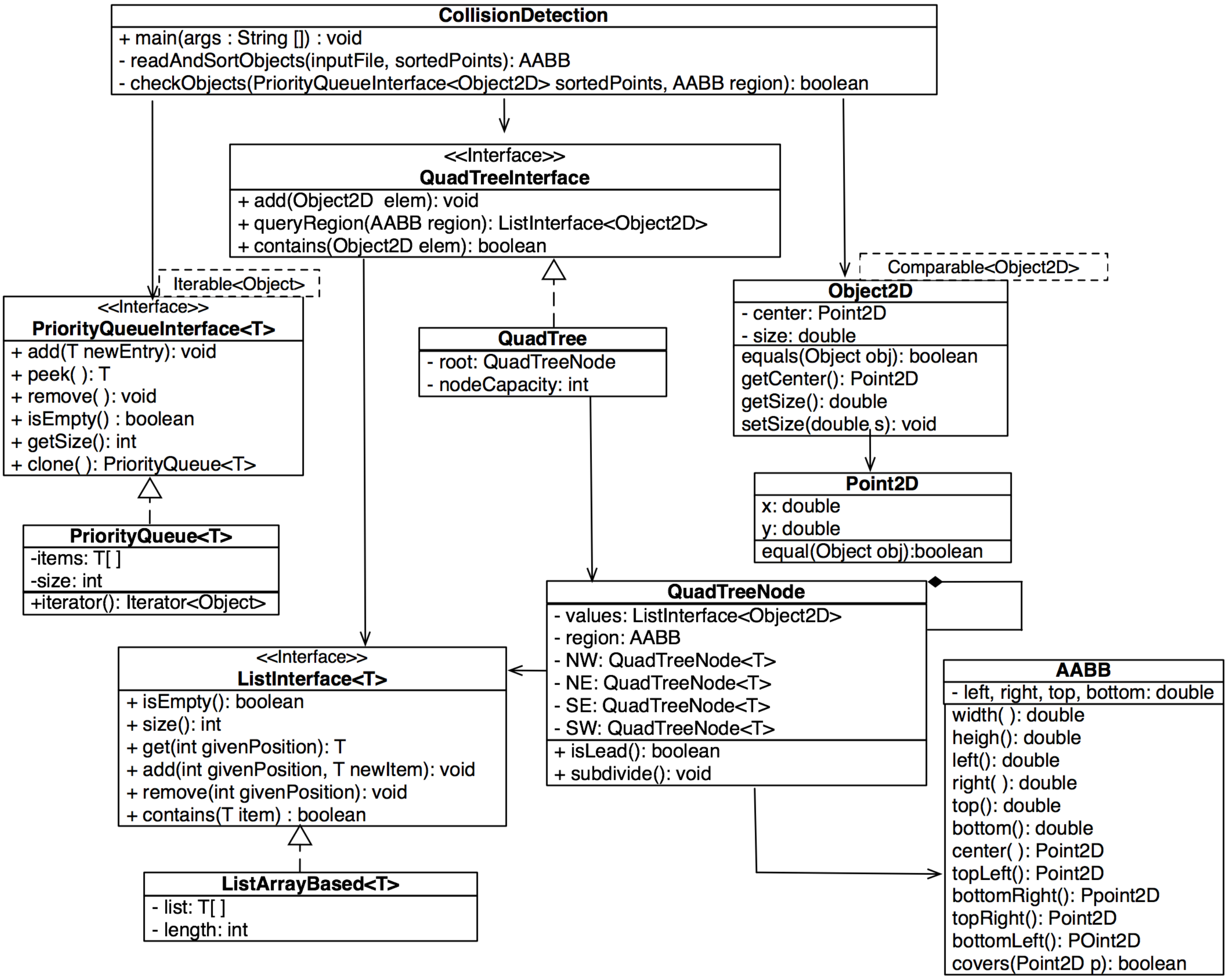

Figure 3: System Architecture (*ParallelCollisionDetection is omitted for brevity*)

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning