Review Score Aggregator 1.0 Component Specification

1. Design

1.1 Overview

This component is used to aggregate the various review scores for submissions in a component design or development contest. There are three steps in aggregating and ranking submissions:

1. **Aggregating**
2. **Calculate relative placements, using tie detection and tie breaking rules**
3. **Assign final placement ranks**

The algorithm that is used for aggregation of scores is pluggable, and a default implementation is provided. This implementation, as required, takes the arithmetic mean of the review scores of a submission to generate the aggregated score.

The relative placement of each submission in the content is also calculated by this component. Ties between submissions are *detected* using a pluggable algorithm; a default implementation is provided which uses an epsilon. If two score are relatively or absolutely within that epsilon of each other, a tie is detected. This functionality was not defined in the requirements specification, but was added to support various tie detection mechanisms.

After ties are detected, ties are broken using another pluggable algorithm, for which a default implementation is provided as described in the requirements specification. This implementation considers the submission with the most “wins” to be the highest ranked of all tied submissions. Then, the highest ranked is removed, and the remaining submissions are re-ranked. If ties remain, the submissions are considered ties.

Finally, to define the final placements, a pluggable algorithm is also available (with a default implementation) to further define tie-breaker rules. This algorithm takes the initial placement (e.g., 1, 2, 2, 3) and creates a final placement (e.g., 1, 2, 2, 4.) A pluggable algorithm was not mentioned in the requirements specification but was added in this design to support future expansion. The default implementation provides exactly the above functionality, which was required (in a post on the developer forums.)

1.2 Design Patterns

Strategy – The various interfaces and implementations thereof represent the Strategy Pattern. The four interfaces are: ***ScoreAggregationAlgorithm***, ***TieDetector***, ***TieBreaker*** and ***PlaceAssignmentAlgorithm***.

Façade – The ***ReviewScoreAggregator*** class acts as a façade to the various algorithms that it calls to aggregate and rank submissions. This effectively takes a complex API and makes it into a simple API.

1.3 Industry Standards

None

1.4 Required Algorithms

1.4.1 Calculate placements algorithm

This is the heart of the entire component. It is implemented in the ***calcPlacements*** method of the ***ReviewScoreAggregator*** class. This is a multi-part algorithm because the requirements are complex. There are four parts to this algorithm:

1. **Prepare**. In this step, the input array is sorted descending based on aggregated score
2. **Assign initial placements**. This is where ties are detected using the current ***TieDetector*** implementation, and broken using the current ***TieBreaker*** implementation. The initial placement for each submission is calculated. The output may look like [1, 2, 2, 3, 3, 4].
3. **Convert initial placements into final placements**. This utilizes the current ***PlaceAssignmentAlgorithm***. implementation. The placement array now may look like [1, 2, 2, 4, 4, 6]
4. **Build the output in the form of *RankedSubmission* objects**

1.4.1.1 Prepare

```
// Return an empty array if the input array is empty.
if (submissions.length == 0) {
  return new RankedSubmission[0];
}

// Make a copy of the original array.
// The next manipulations will be on this one.
AggregatedSubmission[] copy = (AggregatedSubmission[]) submissions.clone();

// Sort and reverse the array.
Arrays.sort(copy);
for (int i = 0; i < copy.length / 2; ++i) {
  AggregatedSubmission tmp = copy[i];
  copy[i] = copy[copy.length - 1 - i];
  copy[copy.length - 1 - i] = tmp;
}
```

1.4.1.2 Assign initial placements

```
// Initial placements array (values are 1-based).
int initialPlacements[] = new int[copy.length];

// Current placement (starts from 1).
int currentPlacement = 1;
initialPlacements[0] = 1;

// Index of the first submission that was tied.
int firstTied = 0;

// The current list of tied submission.
List tied = new ArrayList();
tied.add(copy[0]);

// Note the end condition is "i <= copy.length" not "i < copy.length".
// Since in the algorithm the designer provided, some tied submissions
// will be left un-processing.
for (int i = 1; i <= copy.length; ++i) {
  if (i != copy.length && tieDetector.tied(copy[i].getAggregatedScore(), copy[i -
1].getAggregatedScore())) {
    // Tied with the previous one. Add to the list.
    tied.add(copy[i]);
  } else {
    if (tied.size() > 1) {
      // Process the tied submissions.

      // Break ties among these submissions.
      // The array returned will have values starting at 1
     // which we will then add to each record.
     AggregatedSubmission[] subs = new AggregatedSubmission[tied.size()];
     for (int t = 0; t < tied.size(); ++t) {
       subs[t] = (AggregatedSubmission) tied.get(t);
     }
     int relativePlacements[] = tieBreaker.breakTies(subs);
```

```
      for (int j = 0; j < relativePlacements.length; ++j) {
        // Update all entries starting from where we "first tied"
        // (note, subtract 1 because both are 1-based).
        initialPlacements[j + firstTied] = currentPlacement + relativePlacements[j]
- 1;
      }

      // Set the currentPlacement value to the maximum
      // so we continue from the right number.
      for (int j = firstTied; j < i; ++j) {
        currentPlacement = Math.max(currentPlacement, initialPlacements[j]);
      }
    }

    // Clear the tied.
    tied.clear();

    // Assign the next placement value to this
     // (note, it may get overwritten in another iteration).
    if (i != copy.length) {
      // Increase currentPlacement first before assignment.
      initialPlacements[i] = ++currentPlacement;

      // Remember this location and submission in case subsequent
      // entries are also tied with this one.
      firstTied = i;
      tied.add(copy[i]);
     }
  }
}
```

1.4.1.3 Convert to final placements

```
// Convert initial placements into final placements.
int[] finalPlacements =
placeAssignmentAlgorithm.assignPlacements(initialPlacements);
```

1.4.1.4 Build output

```
// Build rankerSubmission objects from the AggregatedSubmissions.
RankedSubmission[] ret = new RankedSubmission[copy.length];
for (int i = 0; i < copy.length; ++i) {
  ret[i] = new RankedSubmission(copy[i], finalPlacements[i]);
}
```

1.4.2 Standard tie breaking algorithm

The standard tie-breaking algorithm takes into account the number of times each tied submission was “won” (according to its review score) by each reviewer. The submission(s) with the most “wins” (there may be several ones) is considered the highest ranked. These submissions are then removed from consideration, and the rest of the submissions are considered for the next highest ranking. This is the algorithm that is implemented by the ***breakTies*** method of the ***StandardTieBreaker*** class:

```
// Note : since the algorithm designer provided use sort every time
// the time complexity is O(nm^2logm)
// where n is the number of submissions and m is the number of reviewers.
// I use some extra memory to record whether a submission has been assigned rank.
// To get the winning submissions, I think a O(m) traverse is enough,
// there's no need to sort.
// So the time complexity is O(nm^2) and space complexity is O(n)
// which is a improvement to the algorithm.

// The rank of each submission.
int[] rank = new int[len];

// The number of scores for each submission.
int n = submissions[0].getScores().length;

// First initialize the nextrank to be 1.
```

```
int nextrank = 1;

// Representing whether a submission has been assigned a rank.
boolean[] visit = new boolean[len];

// The number of submission which has not been assigned a rank.
// Initialized to the number of submissions.
int left = submissions.length;

// These array stores the the score of the i-th submission, j-th reviewer.
// It is for fast access in the next step.
float[][] scores = new float[len][n];
for (int i = 0; i < len; ++i) {
  float[] ss = submissions[i].getScores();
  for (int j = 0; j < n; ++j) {
    scores[i][j] = ss[j];
  }
}

while (left > 0) {
  // The number of win's for each submission.
  // Initialized to be all 0.
  int[] wins = new int[len];

  // For each kind of score (e.g. from the same reviewer)
  // get the winning submissions (there may be several ones).
  for (int i = 0; i < n; ++i) {
    // Store the winning submissions.
    int[] win = new int[len];

    // The number of "EXACT" equal winning submissions.
    int num = 0;

    // The highest score for this specified reviewer score.
    float highest = -1;

    for (int j = 0; j < len; ++j) {
      // Has been assigned a rank, skip.
      if (visit[j]) {
        continue;
      }

      // Score for specified submission (j) and reviewer (i).
      float score = scores[j][i];
      if (score > highest) {
        highest = score;
        num = 0;
        win[num++] = j;
      } else if (score == highest) {
        // Here we use "==" to determine "EXACT" equal to winning submission.
        win[num++] = j;
      }
    }

    for (int j = 0; j < num; ++j) {
      ++wins[win[j]];
    }
  }

  // Find the most number of win's for all the non-rank-assigned submissions.
  int mostwin = 0;
  for (int i = 0; i < len; ++i) {
    // Has been assigned a rank, skip.
    if (visit[i]) {
      continue;
    }

    if (wins[i] > mostwin) {
      mostwin = wins[i];
    }
  }
  // Assign rank to the winning submissions.
  for (int i = 0; i < len; ++i) {
    // Has been assigned a rank, skip.
    if (visit[i]) {
      continue;
    }

    if (wins[i] == mostwin) {
      rank[i] = nextrank;
      --left;
      visit[i] = true;
    }
  }

  // Increase the rank number.
  ++nextrank;
}
return rank;
```

1.4.3 Standard place assignment algorithm

The “standard” place assignment algorithm takes an array of initial placements and creates a final placement array from it. This implementation skips intermediate rankings (e.g., 1,2,2,3 becomes 1,2,2,4). This algorithm is implemented in the ***assignPlacements*** method of the ***StandardPlaceAssignment*** class:

```
// Make a copy of the initialPlacements and sort it.
int[] copy = (int[]) initialPlacements.clone();
Arrays.sort(copy);

if (copy[0] != 1) {
  throw new IllegalArgumentException("The highest rank should be 1.");
}

// Maps from input (1-based) to output.
// The size is 'n+1' since the maximum number is n and the input ranks are 1-based.
int mapping[] = new int[n + 1];

// First place is always 1.
mapping[1] = 1;

// Skip the first one which is already mapped.
for (int i = 1; i < n; ++i) {
  if (copy[i] - copy[i - 1] > 1) {
    throw new IllegalArgumentException("Gaps exist in the numbers of the
initialPlacements");
  }
  if (copy[i] != copy[i - 1]) {
    // We have progressed a non-tied value, set it to the current index (1-based).
    mapping[copy[i]] = i + 1;
  }
}

// Remap inputs to their outputs
int[] ret = new int[n];
for (int i = 0; i < n; ++i) {
  ret[i] = mapping[initialPlacements[i]];
}
return ret;
```

1.5 Component Class Overview

class ReviewScoreAggregator

This is the main class of the Review Score Aggregator component. It provides pluggable implementations of the following four features:

1. Score aggregation (currently just averages)
2. Tie detector (using an epsilon)
3. Tie breaker (using "most wins" among tied submissions)
4. Final place assignment (e.g., to skip tied indices)

There are two main entry points in this class: ***aggregateScores***, which calculates aggregated scores for a set of submissions, and ***calcPlacements***, which takes aggregated submissions and determines their final relative ranking.

class Submission

This class represents a component submission and currently only stores the scores received from each of the reviewers as floats.

class AggregatedSubmission extends Submission implements Comparable

Represents a submission whose scores have been aggregated. This class implements Comparable so it can be used to sort submissions by their aggregated scores.

class RankedSubmission extends AggregatedSubmission

Represents a submission that has been assigned an aggregated score, and a relative rank to other submissions in this component contest. Stores the rank as an integer.

interface ScoreAggregationAlgorithm

This interface defines a score aggregation algorithm, whereby scores from multiple reviewers are aggregated into a single combined score. Different implementations may aggregate scores differently, e.g., arithmetic average, or by throwing out the highest and lowest, and averaging the rest.

class AveragingAggregationAlgorithm implements ScoreAggregationAlgorithm

This class defines the standard score aggregation algorithm, whereby the aggregated score is the arithmetic average of the individual score

interface TieDetector

This interface defines the "tie detector" contract. Two scores, represented as floats, are tested to determine if they are in fact tied. An implementation of this class is used by the ***ReviewScoreAggregator*** class when determining if two or more submissions are tied (and need to have their ranking tie broken by a ***TieBreaker*** implementation.)

class StandardTieDetector implements TieDetector

"Standard" implementation of the ***TieDetector*** interface, which uses an epsilon to detect relative or absolute differences between two scores. It uses an algorithm similar to the standard *TopCoder* comparison method for doubles.

interface TieBreaker

This interface determines the relative placement of a set of Submissions whose aggregated scores have already been determined to be 'tied'. An implementation of this interface is used by the ***ReviewScoreAggregator*** class to break ties between tied submissions. Implementations may use various methods to break ties (including not breaking ties at all).

class StandardTieBreaker implements TieBreaker

This class is the "standard" implementation of the ***TieBreaker*** interface. It implements the tie-breaking algorithm described in the Requirements

Specification, namely submissions with more 'wins' are ranked higher than other submissions with the same tied score.

interface PlaceAssignmentAlgorithm

This interface defines an algorithm for assigning the final place ranking for a set of submissions based on their initial placements. An implementation of this interface is used by the ***ReviewScoreAggregator*** class when calculating final rankings of submissions after aggregation, sorting by score, and tie-breaking is applied. Implementations of this interface can take the given array and apply various transformations on it to produce the final placements. For example, if the initial array is 1,2,2,3, one implementation might assign their final places to be 1,2,2,4. Another implementation might assign their final places as 1,3,3,4.

class StandardPlaceAssignment implements PlaceAssignmentAlgorithm

This is the “standard” implementation of the ***PlaceAssignmentAlgorithm*** interface. This implementation skips intermediate rankings (e.g, 1,2,2,3 becomes 1,2,2,4) after ties are applied.

class Util

Helper class for the Status Tracker Component which provides some useful functionality.

1.6 Component Exception Definitions

ReviewScoreAggregatorException extends BaseException

The generic exception that represents an error that occurred in any part of the Review Score Aggregator component. It can wrap another exception that occurred (e.g., ConfigManagerException)

InconsistentDataException extends ReviewScoreAggregatorException

This exception indicates that the data passed into a method is considered “inconsistent”. Usually it means that the number of review scores is not the same for all submissions provided. It is thrown by both the ***ReviewScoreAggregator*** class as well as implementations of the ***TieBreaker*** interface.

ReviewScoreAggregatorConfigException extends ReviewScoreAggregatorException

This exception indicates that there was a configuration problem with the component. It is thrown by the ***ReviewScoreAggregator*** class, in one of its constructors, when it is attempting to build objects using the Object Factory component.

1.7 Thread Safety

Each class in this component has been designed to be thread-safe. The various algorithm interface implementations are all immutable, as are the Submission class and its descendants. The only mutable class, ***ReviewScoreAggregator***, is explicitly synchronized in each of its methods so that it achieves thread safety.

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.4 is required for compilation and executing test cases.
- Java 1.4 or higher must be used for Java 1.4 built in logging.

2.2 TopCoder Software Components

- Base Exception 1.0: The base class for all custom exceptions in this component.
- Object Factory 2.0: Used by the **ReviewScoreAggregator** class' one-argument constructor to determine the implementation classes to instantiate
- Configuration Manager 2.1.4: Used by the dependent component Object Factory. Note that the Review Score Aggregator component does not rely directly on the Configuration Manager.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- None

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.review.scoreaggregator

3.2 Configuration Parameters

This component depends on a configuration file used by the Object Factory component for its configuration. This is used in the one-arg (String) constructor of the **ReviewScoreAggregator** class. The configuration file that is needed is:

```
<!-- This is the configuration file for ObjectFactory to use to build the
"standard" implementations of the four algorithms. -->
<Config name="com.topcoder.management.review.scoreaggregator">
  <Property name="ScoreAggregationAlgorithm:default">
    <Property name="type">

<Value>com.topcoder.management.review.scoreaggregator.impl.AveragingAggr
egationAlgorithm</Value>
    </Property>
  </Property>
  <Property name="TieDetector:default">
    <Property name="type">

<Value>com.topcoder.management.review.scoreaggregator.impl.StandardTieDe
tector</Value>
    </Property>
  </Property>
  <Property name="TieBreaker:default">
    <Property name="type">

<Value>com.topcoder.management.review.scoreaggregator.impl.StandardTieBr
eaker</Value>
    </Property>
  </Property>
  <Property name="PlaceAssignmentAlgorithm:default">
    <Property name="type">
```

```
<Value>com.topcoder.management.review.scoreaggregator.impl.StandardPlace
Assignment</Value>
    </Property>
  </Property>
</Config>
```

Parameter	Description	Typical Value
`ScoreAggregationAlgorithm:default/type`	Fully-qualified class name of the ScoreAggregationAlgorithm implementation to use	See above
`TieDetector:default/type`	Fully-qualified class name of the `TieDetector` implementation to use	See above
`TieBreaker:default/type`	Fully-qualified class name of the `TieBreaker` implementation to use	See above
`PlaceAssignmentAlgorithm:default/type`	Fully-qualified class name of the `PlaceAssignmentAlgorithm` implementation to use	See above

3.3 Dependencies Configuration

The Object Factory component must be configured prior to use. See above for Object Factory configuration. Setting up the Config Manager with this namespace is desirable so the user doesn't have to manually load the configuration file. In the default Config Manager configuration file (usually located at `conf/com/topcoder/util/config/ConfigManager.properties`), add this line

```
com.topcoder.management.review.scoreaggregator=<path to file described in 3.2>
```

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

- Configure the component by pointing the Configuration Manager at the configuration file described above, or manually managing the pluggable algorithms
- See Demo for examples of usage.

4.3 Demo

Here are the demo provided by the designer.

4.3.1 Manage pluggable algorithms

```
// instantiates the versions in the configuration file
ReviewScoreAggregator rsa = new ReviewScoreAggregator(
                                 ReviewScoreAggregator.DEFAULT_NAMESPACE);

// generates default implementations
rsa = new ReviewScoreAggregator();
```

```
// override the tie detector with a less sensitive one.
rsa.setTieDetector(new StandardTieDetector(0.1));

// should be the StandardTieBreaker
TieBreaker tb = rsa.getTieBreaker();

// should be the AveragingAggregationAlgorihtm
ScoreAggregationAlgorithm aggregator = rsa.getScoreAggregationAlgorithm();

// note, this class is not in the component, for examples only
rsa.setPlacementAlgorithm(new PassthroughPlacementAlgorithm());
```

4.3.2 Aggregate scores

```
// scores from the TCO finals:
float scores[][] = {{97.28, 97.44, 93.47},
                    {89.25, 98.47, 94.47},
                    {86.81, 96.75, 93.59},
                    {92.81, 90.94, 91.50},
                    {78.03, 86.63, 86.44},
                    {78.41, 82.50, 79.03},
                    {60.81, 71.22, 70.97}};

// should have 7 entries, with aggregated scores:
// 96.06333333
// 94.06333333
// 92.38333333
// 91.75
// 83.7
// 79.98
// 67.66666667
AggregatedSubmission aggregated[] = rsa.aggregateScores(scores);

// This should result in the same numbers, because the other version of
// this method takes an array of Submission objects, so it should aggregate
// them the same way.
AggregatedSubmission same[] = rsa.aggregateScores(aggregated);
```

4.3.3 Assign placements

```
// The rankings should be 1 through 7
RankedSubmission[] ranked = rsa.calcPlacements(aggregated);

// The results are aggregated first, then ranked, as above
RankedSubmission[] rawRanked = rsa.calcPlacements(scores);

// Override the tie detector with a MUCH less sensitive one.
rsa.setTieDetector(new StandardTieDetector(3));

// the order is now different - 2,1,3,4,5,6,7 because submission #2
// had more ‘wins’ than 1 or 3 (which are both within 2 points)
ranked = rsa.calcPlacements(aggregated);
```

Here are my demo tests.

/ Some variables used in tests. */**

```
/** Configuration file for ObjectFactory. */
private static final String CONFIG_FILE = "test_files" + File.separator + "Standard.xml";

/** Default namespace that external clients can use when configuring this component. */
private static final String NAMESPACE = ReviewScoreAggregator.DEFAULT_NAMESPACE;
```

```
/** Scores used for tests. */
private static final float[][] SCORES = new float[][] {
        {97.28f, 97.44f, 93.47f},
        {89.25f, 98.47f, 94.47f},
        {86.81f, 96.75f, 93.59f},
        {92.81f, 90.94f, 91.50f},
        {78.03f, 86.63f, 86.44f},
        {78.41f, 82.50f, 79.03f},
        {60.81f, 71.22f, 70.97f}
};

/** Array of <code>AggregatedSubmission</code> instances used for tests. */
private AggregatedSubmission[] aggregatedSubs = null;

/** Array of <code>Submission</code> instances used for tests. */
private Submission[] subs = null;

/** Set up the environment. */
aggregatedSubs = new AggregatedSubmission[SCORES.length];
subs = new Submission[SCORES.length];

for (int i = 0; i < SCORES.length; ++i) {
   aggregatedSubs[i] = new AggregatedSubmission(i + 1, SCORES[i],
                 (new AveragingAggregationAlgorithm()).calculateAggregateScore(SCORES[i]));

  subs[i] = new Submission(i + 1, SCORES[i]);
 }

 ConfigManager cm = ConfigManager.getInstance();
 File file = new File(CONFIG_FILE);
 if (cm.existsNamespace(NAMESPACE)) {
   cm.removeNamespace(NAMESPACE);
 }
 cm.add(file.getAbsolutePath());
```

4.3.4 Demo test for ScoreAggregationAlgorithm

```
ScoreAggregationAlgorithm algo = new AveragingAggregationAlgorithm();

for (int i = 0; i < SCORES.length; ++i) {
   System.out.println(algo.calculateAggregateScore(SCORES[i]));
}
```

4.3.5 Demo test for PlaceAssignmentAlgorithm

```
PlaceAssignmentAlgorithm algo = new StandardPlaceAssignment();
int[] original = new int[] {1, 2, 3, 2, 1, 7, 6, 4, 5};
int[] received = algo.assignPlacements(original);

for (int i = 0; i < received.length; ++i) {
   System.out.println(received[i]);
}
```

4.3.6 Demo test for TieBreaker

```
TieBreaker breaker = new StandardTieBreaker();
```

```
int[] received = breaker.breakTies(aggregatedSubs);

for (int i = 0; i < received.length; ++i) {
   System.out.println(received[i]);
}
```

4.3.7 Demo test for TieDetector

```
TieDetector detector = new StandardTieDetector();
detector = new StandardTieDetector(0.3f);
System.out.println(detector.tied(1, 25.1f));
System.out.println(detector.tied(0.99f, 3));
System.out.println(detector.tied(1.1f, 4));
System.out.println(detector.tied(1, 3.9f));
System.out.println(detector.tied(1, 1.1f));
```

4.3.8 Demo test for Submission

```
Submission sub = new Submission(1, SCORES[0]);
sub.getId();
sub.getScores();
```

4.3.9 Demo test for AggregatedSubmission

```
Submission sub = new Submission(1, SCORES[0]);
AggregatedSubmission aggregatedSub1 = new AggregatedSubmission(sub, 90.0f);
AggregatedSubmission aggregatedSub2 = new AggregatedSubmission(2, SCORES[0], 80.0f);
aggregatedSub1.getId();
aggregatedSub2.getScores();
aggregatedSub1.compareTo(aggregatedSub2);
```

4.3.10 Demo test for RankedSubmission

```
AggregatedSubmission aggregatedSub = new AggregatedSubmission(2, SCORES[0], 80.0f);
RankedSubmission rankedSub = new RankedSubmission(aggregatedSub, 1);
rankedSub.getRank();
```

4.3.11 Demo test for ReviewScoreAggregator

```
ReviewScoreAggregator aggregator = new ReviewScoreAggregator();
aggregator = new ReviewScoreAggregator(new AveragingAggregationAlgorithm(), new
StandardTieDetector(),
            new StandardTieBreaker(), new StandardPlaceAssignment());
      aggregator = new
ReviewScoreAggregator(ReviewScoreAggregator.DEFAULT_NAMESPACE);

aggregator.setScoreAggregationAlgorithm(new AveragingAggregationAlgorithm());
aggregator.getScoreAggregationAlgorithm();

aggregator.setPlaceAssignmentAlgorighm(new StandardPlaceAssignment());
aggregator.getPlaceAssignmentAlgorithm();

aggregator.setTieBreaker(new StandardTieBreaker());
aggregator.getTieBreaker();

aggregator.setTieDetector(new StandardTieDetector(0.3f));
aggregator.getTieDetector();
aggregator.aggregateScores(SCORES);
aggregator.aggregateScores(subs);

aggregator.calcPlacements(SCORES);
aggregator.calcPlacements(aggregatedSubs);
```

5. Future Enhancements

- Add additional tie-detector implementations, such as, a “relative” epsilon, or an “absolute” epsilon

- Add additional tie-breaker implementations, such as, “all the same” (i.e., returns 1,1,1,1), or “aggregated score value” (i.e., returns 1,2,3,4)

- Add additional score aggregation implementations, such as “throw out highest and lowest, and average the rest”

- Add additional final place assignment algorithms, such as one that would convert 1,2,2,3 into 1,3,3,4.

- Integrate with the Review Data Structure and Scorecard Data Structure Components