Component Competition Predictor 1.0 Component Specification

1. **Design**

This component provides a concrete implementation of the Predictor Framework API, specialized to predict the outcome of a component competition.

It introduces a situation - *ComponentCompetitionSituation* - that encompasses information about a competition, a prediction - *ComponentCompetitionFulfillmentPrediction* - that states the likelihood of submissions being provided, and a predictor - *ComponentCompetitionPredictor* - that makes that prediction from that situation.

An additional predictor is available that averages the results of predictors that also use this situation and prediction. It is the *ComponentCompetitionAveragePredictor*. One important issue with this predictor is its compatibility with Object Factory so it can be used with the *PredictorManager*. This is why the constructor uses an array of predictors to instantiate itself. *Object Factory* can work with constructor array parameters, but it is not compatible with Collection parameters.

There are three situation generators available. The *ComponentCompetitionSituationTimelineGenerator* generates a series of *ComponentCompetitionSituations* that span a duration range. The *ComponentCompetitionSituationPrizeGenerator* generates a series of *ComponentCompetitionSituations* that span a prize range. Finally, the *CompetitionSituationTimelineAndPrizeGenerator* generates a series of *ComponentCompetitionSituations* that span a duration and prize range.

There are two comparators available. The *ComponentCompetitionFulfillmentPredictionTimelineComparator* is used to sort the predictions according to their expected submission count, their prize target, and timeline duration, in that order of importance. The *ComponentCompetitionFulfillmentPredictionPrizeComparator* is used to sort the predictions according to their expected submission count, timeline duration, and their prize target, in that order of importance.

1. **Design Patterns**
    1. *Strategy*

All the classes are **strategy** realizations for the interfaces found in the Predictor Framework component.

    2. *Iterator*

The generator classes implement the **iterator** pattern.

    3. *Composite*

The ComponentCompetitionAveragePredictor is a **composite** of Predictor<ComponentCompetitionSituation, ComponentCompetitionFulfillmentPrediction> types, as it is itself that type of predictor.

2. **Industry Standards**

None

3. **Required Algorithms**

None

4. **Component Class Overview**

    1. *com.topcoder.predictor.impl.componentcompetition*

ComponentCompetitionSituation
An implementation of the Situation interface that completely describes a component competition, including the technologies and participants in the component competition, the prize awarded for first place, the timeline, and various other pieces of information.

Technology
A bean that describes a technology used in a component competition situation. It has a name and description.

Participant
A bean that describes a participant in a component competition situation, including the participant's scores given as the component goes through the review stages and various other pieces of information.

ComponentCompetitionFulfillmentPrediction
A Prediction implementation that states the expected number of submissions that will pass review.

ComponentCompetitionPredictor
A simple predictor implementation that uses a ComponentCompetitionSituation to predict how many submissions will pass review, which it will provide in the form of a ComponentCompetitionFulfillmentPrediction. It allows for no training, and has no capabilities. Specifically, any attempt to add capabilities will be ignored. The capability methods are basically disabled, where setting has no effect, and retrieving will result in no results.

ComponentCompetitionAveragePredictor
A simple predictor that aggregates other predictors that use ComponentCompetitionSituation and ComponentCompetitionFulfillmentPrediction. It delegates all training to the constituent predictors, and has no capabilities. Its predict method will obtain the predictions from each constituent predictor, and average them in a new prediction. Please note that when there are three or more predictors, then the average will ignore the highest and lowest prediction. It has no capabilities. Specifically, any

attempt to add capabilities will be ignored. The capability methods are basically disabled, where setting has no effect, and retrieving will result in no results.

2. *com.topcoder.predictor.impl.componentcompetition.analysis*

BaseComponentCompetitionSituationGenerator
A base class for generators of ComponentCompetitionSituation. It provides the place holder for the generated situations, and the access to the iterator. It implements the SituationGenerator interface.

This base class is available to future generators of ComponentCompetitionSituations.

ComponentCompetitionSituationTimelineGenerator
A generator of ComponentCompetitionSituation based on a timeline range, and an increment factor for the step between range values. It extends BaseComponentCompetitionSituationGenerator, to which is passes the generated situations.

ComponentCompetitionSituationPrizeGenerator
A generator of ComponentCompetitionSituation based on a prize range, and an increment factor for the step between range values. It extends BaseComponentCompetitionSituationGenerator, to which is passes the generated situations.

CompetitionSituationTimelineAndPrizeGenerator
A generator of ComponentCompetitionSituation based on timeline and prize ranges, and respective increment factors for the step between these range values. It extends BaseComponentCompetitionSituationGenerator, to which is passes the generated situations.

ComponentCompetitionFulfillmentPredictionTimelineComparator
This PredictionComparator implementation is used to sort the predictions according to their expected submission count, their prize target, and timeline duration, in that order of importance.

ComponentCompetitionFulfillmentPredictionPrizeComparator
This PredictionComparator implementation is used to sort the predictions according to their expected submission count, timeline duration, and their prize target, in that order of importance.

5. **Component Exception Definitions**

This component defines no new exceptions.

6. **Thread Safety**

This component requires that all implemented classes (save for the predictor implementations) are mandated by the framework to be thread-safe.

The ComponentCompetitionSituation is a simple bean where values are managed via their respective getter and setter methods, and locking mechanisms must be used to ensure that they properly manage the backing variable. Since all the values are non-primitive, the easiest and cleanest choice is to use the AtomicReference class to back each value, and thus ensure non-locking, atomic, thread-safe access.

The exceptions to this are the three collection fields – keywords, technologies, and participants. Since these use defensive copying (and thus ensure their continuing thread-safety), all are set to unsynchronized lists and lock when performing the copies, thus obviating the need for an AtomicReference for them. Access to these lists will be synchronized.

The Technology and Participant classes follow the same approach as the ComponentCompetitionSituation to ensure thread-safety. The ComponentCompetitionFulfillmentPrediction is thread-safe because it is immutable.

The remaining classes that are required to be thread-safe are so because they are immutable. This encompasses the prediction, generators and comparators.

Although the predictors are not required to be thread-safe, the ComponentCompetitionPredictor is thread-safe as it is stateless. ComponentCompetitionAveragePredictor is mutable and not thread-safe. One would need to synchronize its getter/setter methods for the predictors to make it thread-safe, but even then this does not guarantee that the predictors it uses are thread-safe. That can only be done by these predictors.

2. **Environment Requirements**
    1. **Environment**

- Development language: Java 1.5+
- Compile target: Java 1.5

    1. **TopCoder Software Components**

- Predictor Framework 1.0
    - Provides the interfaces implemented by this component, and the exceptions they throw.

    3. **Third Party Components**

None

3. **Installation and Configuration**
    1. **Package Names**

com.topcoder.predictor.impl.componentcompetition

com.topcoder.predictor.impl.componentcompetition.analysis

2. **Configuration Parameters**

None

3. **Dependencies Configuration**

The reader/user should look into the supporting components, shown in 2.2, to see how they can be configured.

4. **Usage Notes**

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.

2. **Required steps to use the component**

None

2. **Demo**

The demo will proceed with several scenarios of usage of the implementations. This demo will not rehash any parts that are tangential to it, such as configuration of the PredictorManager from the framework component.

1. *Working with ComponentCompetitionPredictor*

We begin with a demonstration of the ComponentCompetitionPredictor by assembling a ComponentCompetitionSituation with information about one competition that has 4 competitors.

```
// Create a component competition situation (with just the relevant fields)
ComponentCompetitionSituation situation = new ComponentCompetitionSituation();
situation.setPostingDate(// October 1, 2008, 9 am);
situation.setEndDate(// October 5, 2008, 9 am);
situation.setPrize(new Double(800.00));
List<Participant> participants = new ArrayList<Participant>()
Participant participant1 = new Participant();
participant1.setReliability(new Double(0.0));
participants.add(participant1);
Participant participant2 = new Participant();
Participant2.setReliability(new Double(1.0));
participants.add(participant2);
Participant participant3 = new Participant();
Participant3.setReliability(new Double(0.8));
participants.add(participant3);
Participant participant4 = new Participant();
Participant4.setReliability(new Double(0.6));
participants.add(participant4);
```

```
situation.setParticipants(participants);

        // Create a component competition predictor and predict the expected number of
        submissions to pass review
ComponentCompetitionPredictor predictor = new ComponentCompetitionPredictor();
        ComponentCompetitionFulfillmentPrediction prediction = predictor.predict(situation);
        // The prediction's expectedPassedReviewSubmissionCount will be 2.4, as it predicts that
        there will be 2.4 submissions for this competition.
```

2. *Working with ComponentCompetitionAveragePredictor*

This demonstration will show the average predictor in action. We will reuse the predictor created above, and add 4 more predictors. For the sake of brevity, we simply define them as predictors that return predictions of successful passed review submission counts of 3.0, 1.0, 7.1, and 4.0. (For developers, you can simply extend the existing ComponentCompetitionPredictor to return a hardcoded prediction to accomplish the coding of this demo.)

```
        // Create a component competition average predictor and predict the expected number of
        submissions
        Predictor<ComponentCompetitionSituation, ComponentCompetitionFulfillmentPrediction>[]
        predictors = new Predictor<ComponentCompetitionSituation,
        ComponentCompetitionFulfillmentPrediction>[5]
        predictors[0] = predictor; // from 4.2.1. It predicts 2.4
        predictors[1] = // another predictor that predicts 3.0
        predictors[2] = // another predictor that predicts 1.0
        predictors[3] = // another predictor that predicts 7.1
        predictors[4] = // another predictor that predicts 4.0
        ComponentCompetitionAveragePredictor predictor = new
        ComponentCompetitionAveragePredictor(predictors);
        ComponentCompetitionFulfillmentPrediction prediction = predictor.predict(situation);
        // The prediction's expectedPassedReviewSubmissionCount will be 3.133 (rounded off)
        // Here's the reason for this result:
        // There are five predictions, and our algorithm is tasked to ignore the highest and
        lowest if we have 3 or more predictions, and we have 5. So the 7.1 and 1.0 are ignored.
        That leaves 2.4, 3.0, and 4.0, which sum up to 9.4. Divided by 3, it is 3.133 (rounded
        off).
```

3. *Performing analysis with the custom generators and comparators*

This demonstration will show the usage of our generators and comparators in an actual analysis. The analysis will be done by the framework's SimpleAnalyzer class. The goal is to see how these generators and comparators work, not how the SimpleAnalyzer works. We will focus on the CompetitionSituationTimelineAndPrizeGenerator for this demo only because it encompasses the functionality of the other two, and makes the analysis

more interesting. The simpler generators will still be shown to illustrate the expected output, but will not be part of the analysis.

```
// Generate situations based on a template and a range of prizes.
ComponentCompetitionSituationPrizeGenerator prizeGenerator = new
ComponentCompetitionSituationPrizeGenerator(situation, 400.0, 1000.00, 100.0);
Iterator<ComponentCompetitionSituation> = prizeGenerator.iterator();
// This iterator will contain 7 situations, where the prizes will be, in the given order:
// 400.00, 500.00, 600.00, 700.00, 800.00, 900.00, 1000.00

// Generate situations based on a template and a range of timeline durations.
ComponentCompetitionSituationTimelineGenerator timelineGenerator = new
ComponentCompetitionSituationTimelineGenerator(situation, // Oct 3, 9 am, // Oct 5, 9 am,
// one day);
Iterator<ComponentCompetitionSituation> = timelineGenerator.iterator();
// This iterator will contain 3 situations, where the end dates will be, in the given
order:
// Oct 3 at 9 am, Oct 4 at 9 am, Oct 5 at 9 am
```

Now we show an analysis scenario with the CompetitionSituationTimelineAndPrizeGenerator and each comparator.

```
// Create a SimpleAnalyzer and the generator for our component competition
        SimpleAnalyzer<ComponentCompetitionPredictor, ComponentCompetitionSituation,
        ComponentCompetitionFulfillmentPrediction> analyzer = new
        SimpleAnalyzer<ComponentCompetitionPredictor, ComponentCompetitionSituation,
        ComponentCompetitionFulfillmentPrediction>();

        CompetitionSituationTimelineAndPrizeGenerator generator = new
        CompetitionSituationTimelineAndPrizeGenerator(situation, 700.0, 900.00, 100.0, // Oct 3,
        9 am, // Oct 5, 9 am, // one day););
        // This generator will provide 9 situations, in the given order:
        // situation 1: prize = 700.0, end date = Oct 3 at 9 am (duration of 2 days)
        // situation 2: prize = 800.0, end date = Oct 3 at 9 am (duration of 2 days)
        // situation 3: prize = 900.0, end date = Oct 3 at 9 am (duration of 2 days)
        // situation 4: prize = 700.0, end date = Oct 4 at 9 am (duration of 3 days)
        // situation 5: prize = 800.0, end date = Oct 4 at 9 am (duration of 3 days)
        // situation 6: prize = 900.0, end date = Oct 4 at 9 am (duration of 3 days)
        // situation 7: prize = 700.0, end date = Oct 5 at 9 am (duration of 4 days)
        // situation 8: prize = 800.0, end date = Oct 5 at 9 am (duration of 4 days)
        // situation 9: prize = 900.0, end date = Oct 5 at 9 am (duration of 4 days)
```

```
// Perform analysis using timeline comparator. As a reminder, this comparator will favor
situations that will be within the provided range, then closest to the prize target
without being over, then the one with the lowest duration
ComponentCompetitionFulfillmentPredictionTimelineComparator timelineComparator = new
ComponentCompetitionFulfillmentPredictionTimelineComparator(2.0, 3.0, 800);

ComponentCompetitionFulfillmentPrediction prediction = analyzer.analyze(generator,
timelineComparator);
// This will select either situation 1 or situation 2
// Here's why:
// 1. All situations have the same expected submission count, so none are favored
// 2. Situations 1, 2, 4, 5, 7, 8 are favored because they are not above the target prize
of 800.0
// 3. Situations 1, 2 are favored because they have the shortest duration time.

// Perform analysis using prize comparator. As a reminder, this comparator will favor
situations that will be within the provided range, then closest to the timeline target
without being over, then the one with the lowest prize
ComponentCompetitionFulfillmentPredictionPrizeComparator prizeComparator = new
ComponentCompetitionFulfillmentPredictionPrizeComparator(2.0, 3.0, // 3 days);

ComponentCompetitionFulfillmentPrediction prediction = analyzer.analyze(generator,
prizeComparator);
// This will select either situation 1 or situation 4
// Here's why:
// 1. All situations have the same expected submission count, so none are favored
// 2. Situations 1, 2, 3, 4, 5, 6 are favored because they are not above the target
duration of 3 days
// 3. Situations 1, 4 are favored because they have the lowest prize

// The reason why in both cases we state possibility of two predictions even if the
analyzer only returns one is that we illustrate what the power of the comparator is in
each case. It is the case that the analyzer's sorting algorithm, not the comparator, will
determine which situation of the two will be returned.
```

5. Future Enhancements

More complex Predictor implementations have been developed separately, and they will be integrated with this component in a separate competition or set of competitions.