Scorecard Data Structure 1.0 Component Specification

1. Design

A scorecard is a template to produce reviews. The scorecard hierarchy consists of groups, sections and questions. Weight can be assigned on each level to control the contribution to the overall score of the scorecard. Different question types are supported and each question will allow a document, test log for instance, to be uploaded in addition. A scorecard has minimum and maximum passing score, as well as type, status and category attributes. The component defines the scorecard data structure. Separate component will be developed to provide management functionality.

All of the classes that make up this model (Scorecard, Group, Section, Question, QuestionType, ScorecardType, and ScorecardStatus) possess a Java Bean-like interface. That is, they all have a no-argument constructor, and all properties have both getter and setter methods. Convenience constructors for setting some fields are also provided.

Two abstract base classes, NamedScorecardStructure and WeightedScorecardStructure are used to abstract identical methods from multiple classes. This is done to prevent a duplication of code across classes. Choosing this design causes the name field (and its get/set methods) to be added to the Question class. While it is likely this change will not be of great use in the current custom component (because the database persistence backend has already been created), this addition does not present a problem. Adding the name field also makes the component “more symmetrical” in that every class now has a name property.

1.1 Design Patterns

The ScorecardEditor class is similar to a **Proxy**. However, it does not proxy the get methods, but instead allows direct access to the scorecard. Nor is this a **Decorator** pattern, as no common interface/base class exists.

1.2 Industry Standards

The scorecard modeling classes (Scorecard, Group, Section, Question, QuestionType, ScorecardType, and ScorecardStatus) have a Java Bean-like interface. (Although they do not broadcast property changes, which is an integral part of normal Java Beans.)

1.3 Required Algorithms

There are no complicated algorithms in this class, so this section will simply review a few features in the design which may be useful for developers to see condensed here.

1.3.1 *ScorecardEditor*

All set methods in the ScorecardEditor class need to make the following sequence of calls:

- Call the corresponding set/add/remove/clear method on the scorecard
- Call setModificationUser on the scorecard
- Call setModificationDate on the scorecard with the current date/time

1.3.2 *set and reset methods*

These methods provide all of the functionality to manipulate the main classes of this component. There are essentially two types of pairs of set and reset methods provided:

- When the underlying field is a primitive long value, the set method will not permit the user to set the field to its “unassigned” value (which is -1). The reset method must be used in this case.
- When the underlying field is a reference type, the set method can be used both for a normal set and to set the field to its “unassigned value” (which is null). The reset method is provided as a convenience in this case.

The difference in the set/reset behavior is based on the reasoning that when the field is a primitive long type, it is expected to always be in a valid state once initialization of the object (by loading data from a database or through another method) is complete. Resetting it is not considered a normal action and therefore warrants a special method to enact it. On the other hand, for fields for which an “unassigned” (i.e. null) value is expected to occur in normal use, the set method is allowed to set the field to the “unassigned” value, as this is an expected normal state for these fields. The reset method is provided in these cases as a convenience and to maintain API consistency.

1.3.3 *Checking weight sums*

Weights are stored in this component as floats. Because floats do not have an infinite precision, a tolerance value is needed when checking that the weights sum to 100. For this component, an absolute tolerance is used. This simply means that to check for the sum being 100, instead of using:

```
sum == 100
```

The following is used:

```
abs(sum – 100) < tolerance
```

1.4 Component Class Overview

Scorecard:

The Scorecard class is the class at the root of the scorecard modeling hierarchy. It represents an entire scorecard, which is composed primarily of a list of groups. Each scorecard also possesses a few simple data fields. The Scorecard class is simply a container for these basic data fields (along with the groups list). All data fields are mutable, and each data field has a 3 method get/set/reset combination for that data field. The groups list is manipulated through 8 methods (2 adds, 2 removes, a clear and 3 getters).

The only thing to take note of when developing this class is the difference between the set/reset pairs for longs and the set/reset pairs for non-primitive fields. The differences are well documented in the method doc, and the reasoning for this is explained in the component specification.

This class is highly mutable. All fields can be changed.

Group:

The Group class is the second level in the scorecard model hierarchy. A scorecard consists of an ordered list of groups, each of which is represented by an instance of this class. There are two types of data fields for this class: the simple data fields and the sections list. As the simple fields are handled by the base class, all that remains to code in this class is the sections list manipulation methods. The sections list is manipulated through 8 methods (2 adds, 2 removes, a clear, and 3 getters).

This class is mutable (due to its base class being mutable).

Section:

The Section class is the third level in the scorecard model hierarchy. A group consists of an ordered list of sections, each of which is represented by an instance of this class. There are two types of data fields for this class: the simple data fields and the questions list. As the simple fields are handled by the base class, all that remains to code in this class is the questions list manipulation methods. The questions list is manipulated through 8 methods (2 adds, 2 removes, a clear, and 3 getters).

This class is mutable (due to its base class being mutable).

Question:

The Question class is the fourth and final level in the scorecard model hierarchy. A section consists of an ordered list of questions, each of which is represented by an instance of this class. This class is simply a storage container for several data fields, and each is manipulated through a set of 3 methods (get/set/reset). There is no development challenge to this class.

This class is mutable (due to its base class being mutable and its fields being mutable).

NamedScorecardStructure:

The NamedScorecardStructure class is a base class from which other classes in the scorecard structure hierarchy inherit. The purpose of this class is to prevent repetition of code for the id and name properties, and this class simply serves as a container for these properties in addition to having the standard 3 methods for manipulating them (get/set/reset).

The only thing to take note of when developing this class is the difference between the set/reset methods for the id field and those for the name field.

This class is mutable, and all fields can be changed.

WeightedScorecardStructure:
The WeightedScorecardStructure class is a base class from which other classes in the scorecard structure hierarchy inherit. All classes that have a weight property (Group, Section, and Question) inherit from this class. The purpose of this class is to prevent repetition of code for the weight property, and this class simply serves as a container for this field in addition to having the standard 3 methods for manipulating it (get/set/reset).

This class should prove very simple to implement, as all methods should consist of a single line.

This class is mutable.

ScorecardType:
The ScorecardType class supports the Scorecard class, and allows a scorecard to be tagged as belonging to a certain type. Unlike the other classes in this component, which are likely to be created dynamically and frequently, only a few ScorecardType instances are likely to be used in any application. For this component, this consideration really has no impact on the design or development of this class. Like the other classes in this component, it consists of simple data fields, each of which has a 3 method get/set/reset combination for manipulating that field. In fact, all of these fields are in the base class, so there is nothing to code in this class besides the constructors.

This class is mutable (due to its base class being mutable).

ScorecardStatus:
The ScorecardStatus class supports the Scorecard class, and allows a scorecard to be tagged as being in a certain status state. Unlike the other classes in this component, which are likely to be created dynamically and frequently, only a few ScorecardStatus instances are likely to be used in any application. For this component, this consideration really has no impact on the design or development of this class. Like the other classes in this component, it consists of simple data fields, each of which has a 3 method get/set/reset combination for manipulating that field. In fact, all of these fields are in the base class, so there is nothing to code in this class besides the constructors.

This class is mutable (due to its base class being mutable).

QuestionType:

The QuestionType class supports the Question class, and allows a question to be tagged as being of a certain type. Unlike the other classes in this component, which are likely to be created dynamically and frequently, only a few QuestionType instances are likely to be used in any application. For this component, this consideration really has no impact on the design or development of this class. Like the other classes in this component, it consists of simple data fields, each of which has a 3 method get/set/reset combination for manipulating that field. In fact, all of these fields are in the base class, so there is nothing to code in this class besides the constructors.

This class is mutable (due to its base class being mutable).

ScorecardEditor:

The ScorecardEditor class provides the ability to edit the properties of a scorecard while automatically updating the modification user and modification date whenever a property is edited. This class allows the client using this component to associate a user with an editing session and then avoid the hassles of having to manually call the setModificationUser and setModificationDate whenever changes are made. This class simply parallels the set/reset/add/remove/clear methods of the Scorecard class. Each set/reset/add/remove/clear call simply calls the same method on the Scorecard instance and then invokes the setModificationUser and setModificationDate methods.

This class is immutable.

1.5 Component Exception Definitions

No custom exceptions are defined in this component. The Java standard exceptions IllegalArgumentException and IllegalStateException are sufficient to cover all exceptional situations that occur.

1.6 Thread Safety

This component is not thread safe. This decision was made because the majority of the classes in this component are mutable. Combined with there being no business oriented requirement to make the component thread safe, making this component thread safe would only increase development work and needed testing while decreasing runtime performance (because synchronization would be needed for get/set methods). These tradeoffs can not be justified given that there is no current business need for this component to be thread-safe.

This does not mean that making this component thread-safe would be particularly hard. Making modeling classes (Scorecard, Group, Section, Question, QuestionType, ScorecardType, and ScorecardStatus) thread safe can be done by simply adding the synchronized keyword to the various set and access methods. This would leave only the ScorecardEditor class as possibly non thread-safe. The only thread safety issue that would remain with this class is that multiple ScorecardEditor instances that are editing the same underlying Scorecard from multiple threads could conflict in the following way:

- ScorecardEditor #1 sets attribute
- ScorecardEditor #2 sets attribute
- ScorecardEditor #2 sets modification date/time
- ScorecardEditor #1 sets modification date/time

Notice that this causes the modification date/time (last set by #1) to not reflect the last modification (made by #2). Note that even in this case, the synchronization in the Scorecard class would prevent any internal data structure corruption. This race condition could be eliminated by having ScorecardEditor lock on the scorecard field in each set/reset/add/remove/clear method.

2. Environment Requirements

2.1 Environment

Java 1.4 or greater is required for compilation, testing, or use of this component.

2.2 TopCoder Software Components

None.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.scorecard.data

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

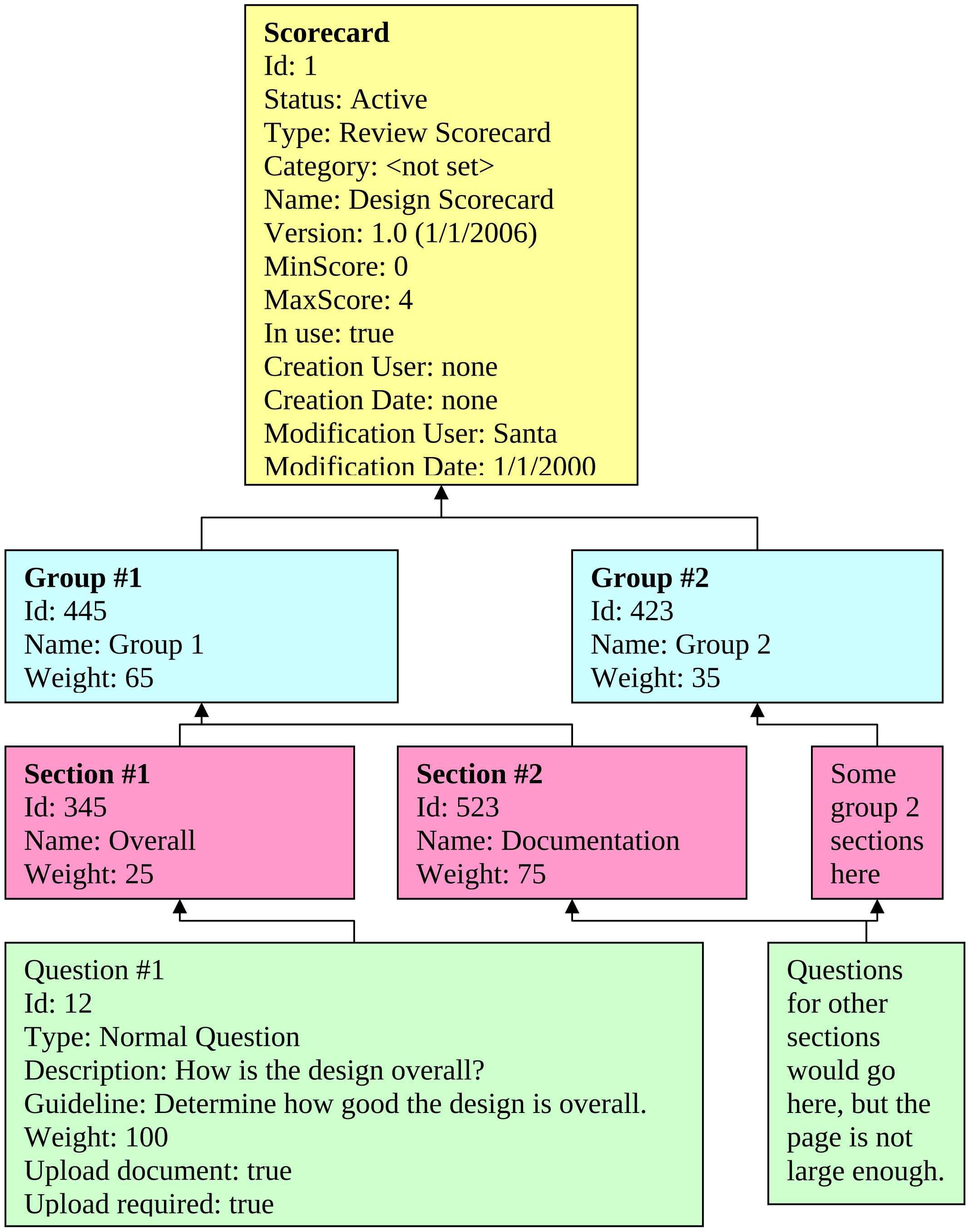
- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Install as per section 3 and follow demo below.

4.3 Demo

As there is only really one thing that can be done to with this component - build up a scorecard hierarchy (and query into an existing scorecard hierarchy) - this demo will simply show how to set up a scorecard object hierarchy that a customer might use. The hierarchy will be the one in the following diagram. (The status and type classes have been condensed into single lines in the classes that use them. This is done simply to make the diagram fit.)



4.3.1 *Creating a Scorecard hierarchy*

```
// Create the scorecard
Scorecard scorecard = new Scorecard();

// Create scorecard status and type objects
// In normal use, these would probably come from some persistence
// source.
ScorecardType scorecardType =
      new ScorecardType(7, “Review Scorecard”);
ScorecardStatus scorecardStatus =
      new ScoreStatus(12, “Active”);

// Set scorecard data values
scorecard.setId(1);
// Note that although this demo will show the ids being set on
// scorecard structurs, the component in no way requires this
// to be done.  Ids can be assigned later.
scorecard.setScorecardStatus(scorecardStatus);
scorecard.setType(scorecardType);
scorecard.setName(“Design Scorecard”);
scorecard.setVersion(“1.0 (1/1/2006)”);
scorecard.setMinScore(0);
scorecard.setMaxScore(4);
scorecard.setInUse(true);
scorecard.setModificationUser(“Santa”);
scorecard.setModificationDate(
      DateFormat.getDateInstance.parse(“1/1/2000”));

// Set the category and then demonstrate how to reset it
// to the unassigned value
scorecard.setCategory(17);
scorecard.resetCategory();

// Create groups
Group group1 = new Group(445, “Group 1”, 65);
Group group2 = new Group(423, “Group 2”, 35);

// Add groups to scorecard
scorecard.addGroups(new Group[] {group1, group2});

// Create sections
Section section1 = new Section(345, “Overall”, 25);
Section section2 = new Section(523, “Documentation”, 75);

// Add sections to first group
group1.addSection(section1);
group1.addSection(section2);

<< Repeat for sections in second group >>

// Create question type.  As for scorecard type/status, this
// would almost certainly be loaded from some persistence.
QuestionType questionType =
      new QuestionType(12, “Normal Question”);
```

```
// Create questions
Question question = new Question();
question.setId(12);
question.setQuestionType(questionType);
question.setDescription("How is the design overall?");
question.setGuideline(
      "Determine how good the design is overall.");
question.setWeight(100);
// Note that this call is not really necessary, as the
// setUploadRequired call would automatically set it to true
question.setUploadDocument(true);
question.setUploadRequired(true);

// Add the question to the section
section1.addQuestion(question);

<< Repeat for questions in other sections >>
```

4.3.2 *Using ScorecardEditor*

```
// Create a ScorecardEditor for editing the properties of the
// scorecard while automatically updating modification
// user and date/time
ScorecardEditor editor =
      new ScorecardEditor(scorecard, "The Grinch");

// Update a property of the scorecard, automatically updating the
// modification user and date
editor.setInUse(false);

// Retrieve the modification date and user.  These will now
// be the "The Grinch" and the current date/time
String modificationUser =
      editor.getScorecard().getModificationUser();
Date modificationDate =
      editor.getScorecard().getModificationTimestamp();
```

4.3.3 *Retrieving data from a Scorecard hierarchy*

Typically, to inspect the data in a scorecard, the scorecard would be loaded from some persistence source, probably by using the Scorecard Management component. This demo will demonstrate the retrieval functionality using the scorecard created above.

```
// In reality, this sort of a line would be used to get a
// scorecard for inspection.
// scorecard = scorecardManager.getScorecard(12345);

// Retrieve data from the scorecard
long id = scorecard.getId();        // will be 1
String version = scorecard.getVersion();// will be 1.0 (1/1/2006)
// … Continue in like manner for other data fields

// Retrieve groups
Group[] groups = scorecard.getAllGroups();
// And iterate over groups
```

```
for (int i = 0; i < groups.length; i++) {
      Group group = groups[i];
      long groupId = group.getId();
      // … Continue in like manner for other data fields

      // Retrieve sections
      Section[] sections = group.getAllSections();
      // And iterate over sections
      for (int j = 0; j < sections.length; j++) {
            Section section = sections[j];
            long sectionId = section.getId();
            // … Continue in like manner for other data fields

            // Retrieve questions
            Question[] questions = section.getAllQuestions();
            // And iterate over questions
            for (int k = 0; k < questions.length; k++) {
                  Question question = questions[k];
                  long questionId = question.getId();
                  // … Continue in like manner for other data
                  // fields
            }
      }
}
```

5. Future Enhancements

As this is a custom component with a well defined (and quite limited) scope, no enhancements are expected.