org.jax.mgi.shr.bucketizer Design Document

Author: M Walker

Created: March 15, 2005

Last Modified: June 21, 2005 12:19

1 Purpose of Document

This document will describe the process known as bucketization and will provide an overview of the classes used in it's implementation and provide some examples on how to utilize this framework within an application.

2 Introduction

The process known as bucketization involves a comparison between two sets of data, discovering the relationships between members of the sets and distinguishing the cardinality of these relationships. For example, a member from set A may be related to one or more members from set B and any of these members from set B may be related to other members from set A. What the bucketizer process does is discover these relationships and "bucketize", or categorize, these data in sets or "buckets" according to the cardinalities of the relationships. The cardinalities we are interested in are 1:1. 1:N, 1:0, 0:1, N:1, N:N, where N is any whole number great than 1.

Note that a relationship between any two objects discovered through the bucketizer is not definitive. A call to the application is required to determine if two set members are truly related. The **Decider** interface (described in the next section) is used for this purpose.

The bucketization process uses an abstraction called the set-valued attribute (SVA) to determine the relationships. As the name implies, an SVA is an attribute of some object with a value represented as a set of one or more unique values. The **SVASet** class (see org.jax.mgi.shr.sva from the lib_java_core product) is a class which contains a set of SVAs, all of which are named. The bucketizer algorithm can determine a relationship between two objects by comparing the corresponding **SVASets** from each. For two objects to be associated the following rule is applied: they must share at least one value from at least one SVA from their **SVASets** and that value cannot belong to any corresponding SVA from any other object (that is the sharing of a common value within a common SVA must be exclusive between the two objects). So, in order to utilize the bucketization algorithm, the data attributes from members of the data sets must be represented as **SVASets**. This constraint is represented in the **Bucketizable** interface. A **Bucketizable** is an interface for any object which can represent it's data contents as an **SVASet.** All application objects that are to be processed through bucketization must implement the **Bucketizable** interface (see section 6 for more information).

One example of an **SVASet** would be marker to sequence relationships grouped by sequence category (RefSeq, GenBank, etc). A set valued attribute would represent a one to many relationship between the marker and the sequences from a particular sequence category. So the **SVASet** would be a set of named sequence categories, each containing sequences belonging to the particular category. The Entrez Gene load implements the bucketizer frameworks in this way, defining the MGI markers and Entrez Gene genes as entities containing sequence associations represented as **SVASets.** The Entrez Gene load could be used as an example implementation of the bucketization frameworks.

3 Class Overview

The **Bucketizable** interface has an id, a provider id (for example MGI or EntrezGene) and an accessor method for obtaining its attributes as an **SVASet** (the getSVASet() method). Application objects which are to be compared and bucketized are intended to implement this interface.

The **SimpleBucketizable** is simple implementation of the **Bucketizable** interface. It has an id, provider and a SVASet. Additionally, there are methods for adding values to the **SVASet**. This class can be used by the application as a base class when implementing the **Bucketizable** interface.

The **Decider** interface provides a decide method which determines whether two given **Bucketizables** are related and returns an Object if true or null if false. This interface is implemented at the application level. Three objects are passed as parameters to the decide() method call for this interface. Parameter one and two are the **Bucketizables** that are associated and the third is an **SVASet** which contains only the common values between the two **Bucketizables**. The return object of the decide() method may represent anything the application wants. It is stored internally along with the two related objects and is made available to the application level during bucket item processing. A default implementation is provided at the framework level that always returns true. The object returned is the same object passed in as the third parameter to the call, i.e. the SVASet object containing the common values between the associated objects.

The **AbstractBucketizer** contains two iterators for the two sets of **Bucketizables**, a **Decider**, and a list of SVA names to be used in finding common values between **Bucketizable** objects. To match **Bucketizables**, each must share at least one value from at least one of their SVAs and that value must not be shared by any **Bucketizable** outside the pair. This pair is passed to the **Decider** along with a **SVASet** that contain only values meeting the match criteria. The **Decider** determines whether they are definitively associated. The **AbstractBucketizer** also iterates through all the associated elements and creates **BucketItems** (see below) which are subsequently processed by the abstract methods of the **AbstractBucketizer**, process_one_to_one(**BucketItem**), process_one_to_many(**BucketItem**), etc. A default **Decider** is used by the **AbstractBucketizer** if none is provided, which always returns true.

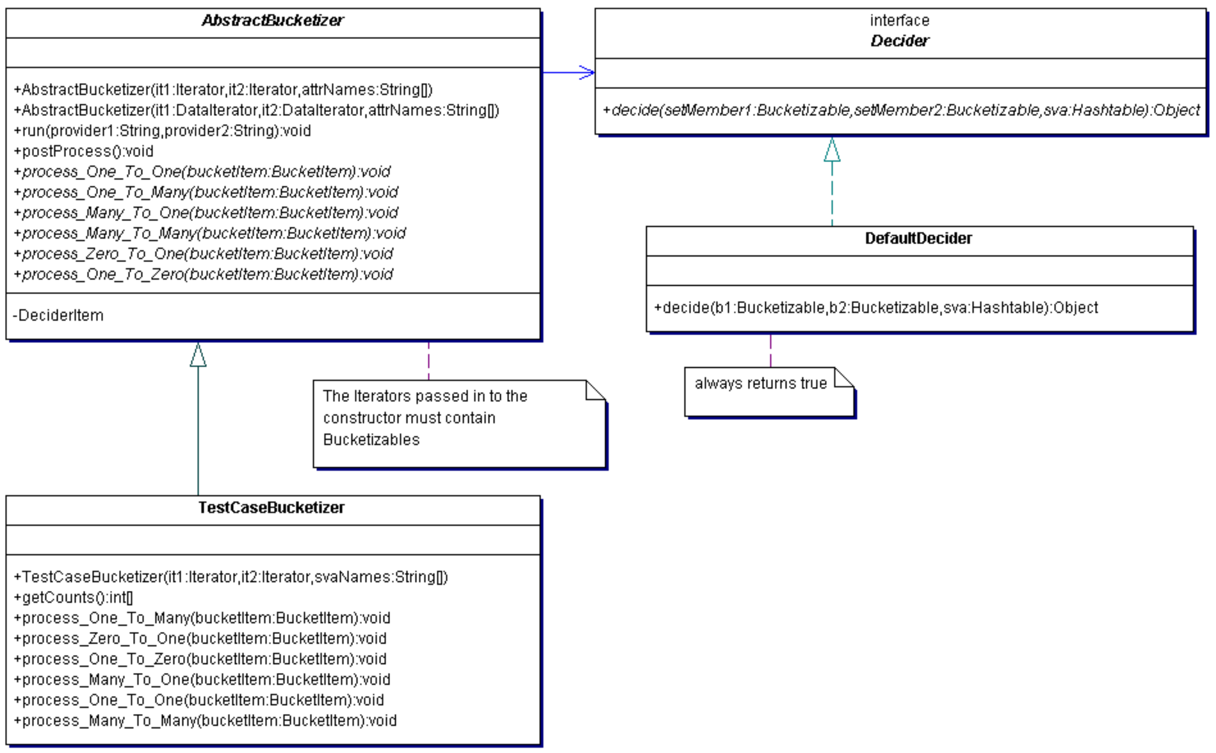
The **BucketItems** stores a set of associated objects and determines the cardinality of the associations (i.e., 1:1, 1:0, 1:N, etc). Accessor methods are provided for accessing the associated objects, their cardinality classification and a label for the association. This label is application dependant. It is the return object from the decide() method in the **Decider** interface. If the default **Decider** is used, then this object represents the reason the two objects were associated (represented as a **SVASet** containing the common values between the two objects).

The **BucketItemProcessor** interface provides a method to process **BucketItems**. It can be used by classes which extend **AbstractBucketizer** and implement the abstract methods for processing **BucketItems.**

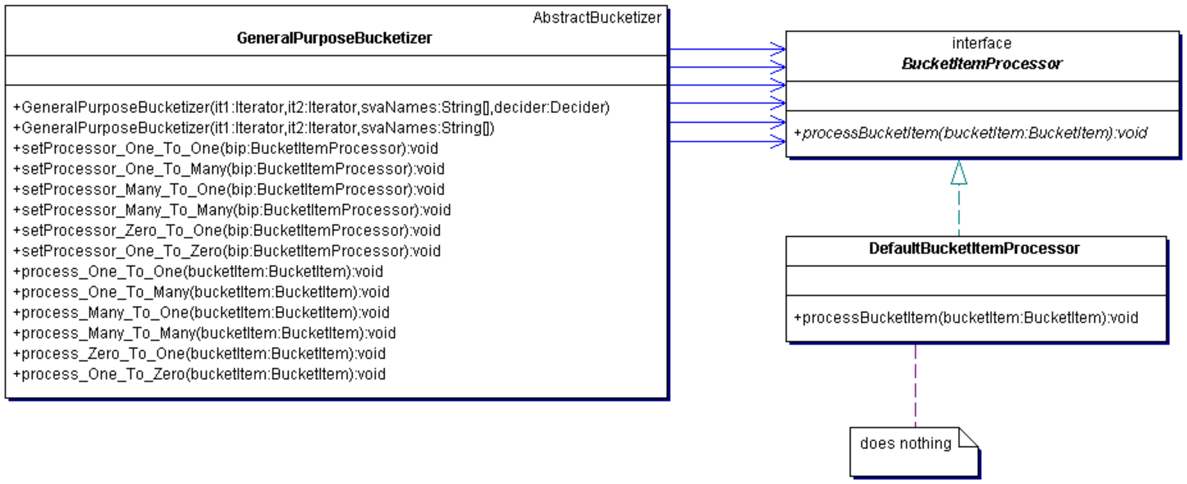
The **GeneralPurposeBucketizer** extends **AbstractBucketizer** and implements the abstract methods, process_one_to_one(**BucketItem**), process_one_to_many(**BucketItem**), etc. It implements them by calling a **BucketItemProcessor** for the corresponding bucket. There is one **BucketItemProcessor** for each bucket. The client of this class provides the **BucketItemProcessors** for each bucket, the two data sets of **Bucketizables**, a **Decider** and a list of SVA names to match on. A default **BucketItemProcessor** which implements the process method by doing nothing is used for any bucket where no **BucketItemProcessor** was provided.

4 Class Diagrams

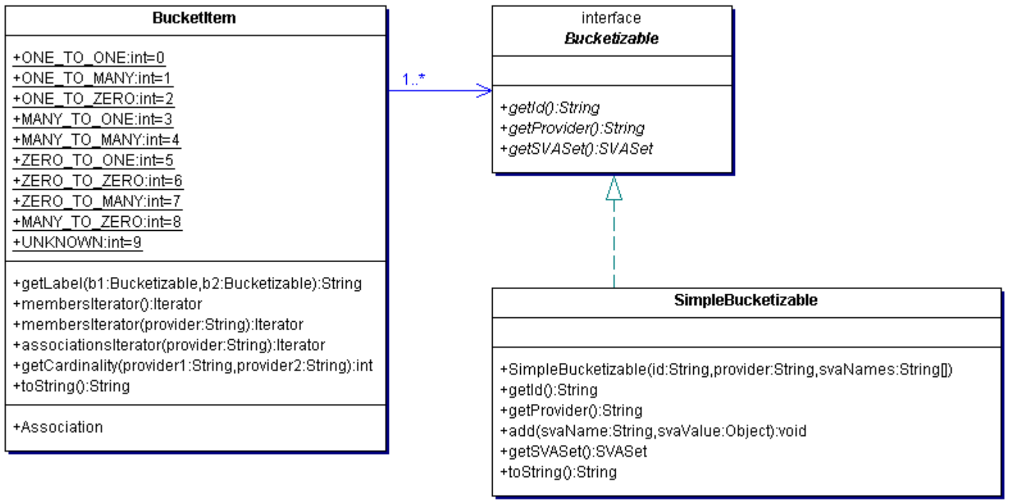
4.1 AbstractBucketizer


AbstractBucketizer
+AbstractBucketizer(it1:Iterator,it2:Iterator,attrNames:String[])
+AbstractBucketizer(it1:DataIterator,it2:DataIterator,attrNames:String[])
+run(provider1:String,provider2:String):void
+postProcess():void
+*process_One_To_One(bucketItem:BucketItem):void*
+*process_One_To_Many(bucketItem:BucketItem):void*
+*process_Many_To_One(bucketItem:BucketItem):void*
+*process_Many_To_Many(bucketItem:BucketItem):void*
+*process_Zero_To_One(bucketItem:BucketItem):void*
+*process_One_To_Zero(bucketItem:BucketItem):void*
-DeciderItem
interface
Decider
+*decide(setMember1:Bucketizable,setMember2:Bucketizable,sva:Hashtable):Object*
DefaultDecider
+decide(b1:Bucketizable,b2:Bucketizable,sva:Hashtable):Object
always returns true
The Iterators passed in to the constructor must contain Bucketizables
TestCaseBucketizer
+TestCaseBucketizer(it1:Iterator,it2:Iterator,svaNames:String[])
+getCounts():int[]
+process_One_To_Many(bucketItem:BucketItem):void
+process_Zero_To_One(bucketItem:BucketItem):void
+process_One_To_Zero(bucketItem:BucketItem):void
+process_Many_To_One(bucketItem:BucketItem):void
+process_One_To_One(bucketItem:BucketItem):void
+process_Many_To_Many(bucketItem:BucketItem):void

4.2 GeneralPurposeBucketizer


AbstractBucketizer
GeneralPurposeBucketizer
+GeneralPurposeBucketizer(it1:Iterator,it2:Iterator,svaNames:String[],decider:Decider)
+GeneralPurposeBucketizer(it1:Iterator,it2:Iterator,svaNames:String[])
+setProcessor_One_To_One(bip:BucketItemProcessor):void
+setProcessor_One_To_Many(bip:BucketItemProcessor):void
+setProcessor_Many_To_One(bip:BucketItemProcessor):void
+setProcessor_Many_To_Many(bip:BucketItemProcessor):void
+setProcessor_Zero_To_One(bip:BucketItemProcessor):void
+setProcessor_One_To_Zero(bip:BucketItemProcessor):void
+process_One_To_One(bucketItem:BucketItem):void
+process_One_To_Many(bucketItem:BucketItem):void
+process_Many_To_One(bucketItem:BucketItem):void
+process_Many_To_Many(bucketItem:BucketItem):void
+process_Zero_To_One(bucketItem:BucketItem):void
+process_One_To_Zero(bucketItem:BucketItem):void
interface
BucketItemProcessor
+processBucketItem(bucketItem:BucketItem):void
DefaultBucketItemProcessor
+processBucketItem(bucketItem:BucketItem):void
does nothing

4.3 BucketItem



5 Implementation Details

The bucketization process is mostly implemented within the **AbstractBucketizer** class and it involves three major steps. First, the given **Bucketizables** are parsed and stored internally as nodes within a graph (the graphs package from lib_java_core is used). These nodes will eventually get connected to one another by creating edges within the graph between the nodes. These edges will represent associations between elements. As these **Bucketizables** are being parsed, internal hashtables are being created from their **SVASets** where the **Bucketizables** are being indexed by the values within the **SVASets**. That is, for each set-valued attribute from each **Bucketizable**, a hashtable is built where the keys are the values from the SVAs and the hashtable values are the set of **Bucketizables** which share that value. For example, in the Entrez Gene load, internal hashtables are created for each sequence category (RefSeq, GenBank, etc). The sequences are the keys to these hashtables and the values are the sets of MGI markers and Entrez Gene genes which share that sequence. This is how the exclusivity rule from the definition of an association between two objects is implemented. The internal hashtables are parsed, looking for sequences which have only one member from one set and one member from the other. The **SVAIndex** class from the org.jax.mgi.shr.sva package is used for this. Its purpose is to perform this indexing of objects based upon the values from their corresponding **SVASets**.

The second step involves iterating through all the internal hashtables and finding all the associated pairs. These objects are passed to the application level through the **Decider** interface so that the application can determine definitively whether or not they are associated. If the application determines an association, then an edge is added to the internal graph between the two **Bucketizables** (represented as nodes in the graph). If the application does not implement a **Decider** then all associations are provided with edges.

The third step involves utilizing the org.jax.mgi.shr.graphs package from lib_java_core to traverse the internal graph and find all the connected components. These connected components represent a cluster of elements from each data set that are all associated to each other. This cluster can then be bucketized according to the cardinality between the two data sets (see Introduction section above). An instance of the **BucketItem** class is created for each cluster and handed to the application level for processing. Processing is handled on the bases of cardinality. So there is a method for processing one to ones, one to manys, etc. These methods are implemented at the application level. The **AbstractBucketizer** defines these processing methods as abstract and it is up to the application to implement them.

6 Application Level Usage

An application may directly extend the class **AbstractBucketizer** and implement the abstract process methods used for processing the various cardinality classifications or the application may use the **GeneralPurposeBucketizer** and provide various **BucketItemProcessors** to handle processing. With either approach the application needs to provide two data sets of Bucketizables, a **Decider**, and a list of SVA names to match on, although a default **Decider** which always returns true is used if one is not provided. One typical approach involves implementing the **Bucketizable** interface through the application objects themselves. For example, if the application objects represent MGI markers and Entrez Gene genes, then one approach would entail creating these objects or extending existing objects to implement the **Bucketizable** interface. That way, when a call is made to the **Decider**, these **Bucketizable** objects are passed back to the application which can be cast to the application objects for further interrogation. Alternatively, a class called the **SimpleBucketizable** is provided which would allow the application level class to extend and inherit the **SVASet** functionality without having to implement it.

The **Decider** interface is implemented at the application level unless the default version is being used. The **Decider** interface defines one method with the following signature

Object decide(**Bucketizable**, **Bucketizable, SVASet**)

The application class that implements this interface is called by the **AbstractBucketizer** and the two associated **Bucketizables** are passed in as parameters as well as the intersection of the two **SVASets** (that is the common values from the two **SVASets**). The intersection is represented as a **SVASet**. The object returned by this call is set to null by the application if the given **Bucketizables** are decided to not be associated. If they are associated, then the object would be set to a not null value. This object can represent anything the application wants. It is stored internally (as a label to the graph edge) along with the pair of matching **Bucketizables** and is made available again to the application level at a later time during **BucketItem** processing. One possible use of this object is to store the reason why the **Decider** associated the pair. This is provided by the **AbstractBucketizer** in the third parameter to the decide() method call. The default **Decider** class provided by this package does just that. It returns the third parameter to the decide() method as the return object to the decide() method call. If the label object is not really needed then simply return a new instance of Java Object to signify an association.