RFC - Properties

Metacosm Development Team

Revision History

Revision M1July, 15th 2001
Conversion to DocBook SGML. Field definition and interface. StringKey replaced by String.
Revision 0.1 April, 30th 2001Revised by: Ruffy
Based on a meeting report

This document describes Properties as used in Entity-RFC.

1. License

2. Metacosm

For more information on Metacosm, check our site: Metacosm (http://metacosm.org).

3. Definition

Properties describe a particular aspect of an Entity. A Property is defined by a name and a value, which can be limited to some field. There are two main types of Properties: numeric ones and non-numeric ones.

we haven’t found some useful example where a ’null’ value could be useful, so ’null’ value is tested against.

Implementation:

```
abstract class Property
{
  Property( String name) throws IllegalArgument;
  // IllegalArgument if null value

  final String getName();
  Object getValue();
}
```

4. Numeric Properties

A Numeric Properties contains a numeric value.

Idea 1: we mainly need numeric values only defined in some field (for example, [3,18] for the sum of 3 dices).

Idea 2: the field could be defined with intervals, sets or regular expressions.

The value of a Numeric Property is always limited to some field. We couldn't find useful examples for using sets or regular expressions (moreover the last ones are complex to implement and to use). So we'll only use intervals. We couldn't find useful examples for using intersections, unions or exclusions of intervals to define the field. So we'll only use one interval. The interval should contain one or more numbers.

If no interval is given, the interval with the min and max values for this value type are used.

If a interval is given (a min and a max values), both values should be legal (not 'NaN' for a real, for example), and correctly put in order.

We'll need at least some Properties both for integers and real values. To start, we'll defined Properties with 'long' and 'double' values.

Implementation:

```
class LongProperty extends Property
{
  LongProperty( String name, long value) throws IllegalArgument;
  // IllegalArgument if null name
  // field defaulted to [Long.MIN_VALUE, Long.MAX_VALUE]

  LongProperty( String name, long value, long min, long max)
  throws IllegalArgument;
  // IllegalArgument if null name or max < min

  long getLongValue();
  long getMinValue();
  long getMaxValue();

  void applyModifier( AbsoluteModifier modifier);
  void applyModifier( RelativeModifier modifier);
  void removeModifier( AbsoluteModifier modifier);
  void removeModifier( RelativeModifier modifier);
}

class DoubleProperty extends Property
{
```

```
  DoubleProperty( String name, dou-
ble value) throws IllegalArgument;
  // IllegalArgument if null name or NaN value
  // field defaulted to [Double.MIN_VALUE, Double.MAX_VALUE]

  DoubleProperty( String name, double value, double min, dou-
ble max)
  throws IllegalArgument
  // IllegalArgu-
ment if null name or NaN value, min or max, or max < min

  double getDoubleValue();
  double getMinValue();
  double getMaxValue();

  void applyModifier( AbsoluteModifier modifier);
  void applyModifier( RelativeModifier modifier);
  void removeModifier( AbsoluteModifier modifier);
  void removeModifier( RelativeModifier modifier);
}
```

for implementation, min and max values can be stored in a memory-friendly way. They aren't stored as two long or double values but as a special immutable Interval object (in fact a reference to this object). A hashtable links (min, max) couples to IntervalManager objects, so for a given interval, only one Interval exists.

Old solution's memory requirements: two values * number of such Properties.

New solution's memory requirements: a reference * number of such Properties + Interval object with two values (+ hashtable)

Implementation:

```
final class LongInterval
{
  LongInterval( long min, long max); // no control here
  long getMin();
  long getMax();
  boolean isInInterval( long value);
  // return (min <= value && value <= max)
  long boundedValue( long value);
  // min if value < min, max if value > max, otherwise value
}

final class DoubleInterval
{
  DoubleInterval( double min, double max); // no control here
  double getMin();
  double getMax();
  boolean isInInterval( double value);
  // return (min <= value && value <= max)
  double boundedValue( long value);
  // min if value < min, max if value > max, otherwise value
}
```

5. Non-numeric Properties

A Non-numeric Properties contains a non-numeric value (i.e. an object).

Idea 1: the value can be in a field or not If it has a field, the Property can't have a value outside the field (i.e. must have a value from the field).

Idea 2: the field could be defined with sets or regular expressions.

A Non-Numeric Property could or not be limited to some field. We couldn't find useful examples for using regular expressions. So we'll only use sets.

We couldn't find useful examples for using intersections, unions or exclusions of sets to define the field. So we'll only use one set (or none). The set should contain one or more values (i.e. a field is a non-null and non-empty set of distinct non-null Objects).

Implementation:

```
class NonNumericProperty extends Property
{
  NonNumericProperty( String name, Ob-
ject value) throws IllegalArgument;
  // IllegalArgument if name or value are null
  // no field

  NonNumericProperty( String name, Object value, Field field);
  // IllegalArgument if name or value or field are null

  void setValue( Object value) throws IllegalArgument;
  // IllegalArgument if null value or not in field

  void noField();

  void addToField( Object value);
  // IllegalArgument if null value
  // no error if value already in the field

  void removeFromField( Object value) throws IllegalArgument;
  // IllegalArgument if null value or cur-
rent value equals to value
  // no error if value is not in the field
}

// immutable
class Field
{
  Field( Object value) throws IllegalArgument;
  // IllegalArgument if value is null

  Field( Object[] values) throws IllegalArgument;
```

```
  // IllegalArgument if values is null or one value is null
  // if multiple occurrences, stores only one

  static Field add( Field field, Ob-
ject value) throws IllegalArgument;
  // IllegalArgument if value is null
  // no error if value already in the field
  // returns a new Field

  static Field add( Field field, Object[] values);
  // IllegalArgument if values or one value is null
  // no error if a value is already in the field
  // if multiple occurrences, stores only one
  // returns a new Field

  static Field remove( Field field, Object value);
  // IllegalArgument if value is null
  // no error if value is not in the field
  // returns a new Field or null if empty

  static Field remove( Field field, Object[] values);
  // IllegalArgument if values or one value is null
  // no error if value is not in the field
  // returns a new Field or null if empty
}
```