```
/*
 * Created on 13-okt-2008
 */
package be.SIRAPRISE.typeimplementations;

import java.nio.ByteBuffer;
import java.util.*;

import be.SIRAPRISE.util.NotFoundException;

/**
 * A NonScalarTypeDeclaration is a TypeDeclaration that accompanies the name of an attribute in a Heading if that attribute is relation-typed or tuple-typed.
 *
 * @author Erwin Smout
 */
public final class NonScalarTypeDeclaration extends TypeDeclaration {

	/**
	 * @param heading
	 * @return -
	 */
	private static HashSet<KeyDef> getSingleFullKeySpec (Heading heading) {
		final HashSet<KeyDef> s = new HashSet<KeyDef>();
		s.add(new KeyDef(heading.getAttributeNameIdentifiers(), false));
		return s;
	}

	/**
	 * Gets the type declaration that is the declaration for the type that is the least specific subtype for both types given
	 *
	 * @param t1
	 *            type 1
	 * @param t2
	 *            type 2
	 * @return the type declaration that is the declaration for the type that is the least specific subtype for both types given
	 * @throws NotFoundException
	 *             If no such supertype exists.
	 */
	public static TypeDeclaration getCommonNonScalarSubTypeDeclaration (NonScalarTypeDeclaration t1, NonScalarTypeDeclaration t2) throws NotFoundException {
		final NameIdentifier typeName = t1.getTypeNameIdentifier();
		if (typeName.equalsNameIdentifier(t2.getTypeNameIdentifier())) {
			return TypeDeclaration.getTypeDeclaration(Heading.getCommonSubTypesHHeading(t1.getHeading(), t2.getHeading()), typeName);
		}

		throw new NotFoundException(t1.getTypeNameIdentifier().getString() + '/' + t2.getTypeNameIdentifier().getString());
```

```
	}

	/**
	 * Gets the type declaration that is the declaration for the type that is the most specific supertype for both types given
	 *
	 * @param t1
	 *            type 1
	 * @param t2
	 *            type 2
	 * @return the type declaration that is the declaration for the type that is the most specific supertype for both types given
	 * @throws NotFoundException
	 *             If no such supertype exists. This is the case if one type is a relation type and the other is a tuple type, or if the involved headings' attribute name sets are unequal, or if an attribute in the involved headings is of respective types that do not have a common supertype.
	 */
	public static TypeDeclaration getCommonNonScalarSuperTypeDeclaration (NonScalarTypeDeclaration t1, NonScalarTypeDeclaration t2) throws NotFoundException {
		final NameIdentifier typeName = t1.getTypeNameIdentifier();
		if (typeName.equalsNameIdentifier(t2.getTypeNameIdentifier())) {
			return TypeDeclaration.getTypeDeclaration(Heading.getCommonSuperTypesHHeading(t1.getHeading(), t2.getHeading()), typeName);
		}

		throw new NotFoundException(t1.getTypeNameIdentifier().getString() + '/' + t2.getTypeNameIdentifier().getString());
	}

	/**
	 *
	 */
	private int hashCode = Integer.MIN_VALUE;

	/**
	 * The heading
	 */
	private final Heading heading;

	/**
	 * The set of key specifications that relations corresponding to this type declaration are known to satisfy. Only maintained for RELATION type declarations
	 */
	private Collection<KeyDef> keySpecifications;

	/**
	 * The predicate corresponding to the nonscalar value held in the container that this typedeclaration is for
	 */
	private String relationPredicate;
```

```
	/**
	 * Creates the typedeclaration
	 *
	 * @param heading
	 *		The heading defining the details of the nonscalar type the declaration si for
	 * @param typeName
	 *		The name of the nonscalar type that the declaration is for
	 */
	public NonScalarTypeDeclaration (Heading heading, NameIdentifier typeName) {
		this(heading, typeName, getSingleFullKeySpec(heading), ""); //$NON-NLS-1$
	}

	/**
	 * Creates the NonScalarTypeDeclaration
	 *
	 * @param heading
	 *		The heading defining the details of the nonscalar type the declaration si for
	 * @param typeName
	 *		The name of the nonscalar type that the declaration is for
	 * @param keySpecifications
	 *		The set of key specifications that relations corresponding to this type declaration
are known to satisfy. Only maintained for RELATION type declarations
	 * @param relationPredicate
	 *		The predicate corresponding to the nonscalar value held in the container that this
typedeclaration is for
	 */
	public NonScalarTypeDeclaration (Heading heading, NameIdentifier typeName,
Collection<KeyDef> keySpecifications, String relationPredicate) {
		super(typeName);
		if (!(typeName.getString().equalsIgnoreCase(TYPENAMES.RELATION) ||
typeName.getString().equalsIgnoreCase(TYPENAMES.TUPLE))) {
			throw new IllegalArgumentException(typeName.getString());
		}
		this.heading = heading;
		this.keySpecifications =
typeName.getString().equalsIgnoreCase(TYPENAMES.RELATION) ? keySpecifications : null;
		this.relationPredicate = relationPredicate.intern();
	}

	/**
	 * Creates the typedeclaration
	 *
	 * @param heading
	 *		The heading defining the details of the nonscalar type the declaration si for
	 * @param typeName
	 *		The name of the nonscalar type that the declaration is for
	 * @deprecated deprecated
	 */
	@Deprecated
	public NonScalarTypeDeclaration (Heading heading, String typeName) {
		this(heading, NameIdentifier.get(typeName));
	}
```

```
	/**
	 * Creates the NonScalarTypeDeclaration
	 *
	 * @param heading
	 *		The heading defining the details of the nonscalar type the declaration si for
	 * @param typeName
	 *		The name of the nonscalar type that the declaration is for
	 * @param keySpecifications
	 *		The set of key specifications that relations corresponding to this type declaration
are known to satisfy. Only maintained for RELATION type declarations
	 * @param relationPredicate
	 *		The predicate corresponding to the nonscalar value held in the container that this
typedeclaration is for
	 * @deprecated deprecated
	 */
	@Deprecated
	public NonScalarTypeDeclaration (Heading heading, String typeName,
Collection<KeyDef> keySpecifications, String relationPredicate) {
		this(heading, NameIdentifier.get(typeName), keySpecifications, relationPredicate);
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see java.lang.Object#equals(java.lang.Object)
	 */
	@Override
	public boolean equals (Object obj) {
		return obj instanceof NonScalarTypeDeclaration &&
equalsNonScalarTypeDeclaration((NonScalarTypeDeclaration) obj);
	}

	/**
	 * Compares this NonScalarTypeDeclaration to another one for equality. A
NonScalarTypeDeclaration is equal to another one if and only if both the type names and the
Headings are equal to one another.
	 *
	 * @param obj
	 *		Another NonScalarTypeDeclaration to compare this one to for equality
	 * @return true if this NonScalarTypeDeclaration is the same type and the same Heading as
the other one, false otherwise.
	 */
	public boolean equalsNonScalarTypeDeclaration (NonScalarTypeDeclaration obj) {
		return this.hashCode() == obj.hashCode() &&
getTypeNameIdentifier().getString().equalsIgnoreCase(obj.getTypeNameIdentifier().getString())
&& heading.equalsHeading(obj.heading);
	}

	/**
	 * Gets the heading defining the concerned nonscalar type
	 *
```

```
	 * @return the heading defining the concerned nonscalar type
	 */
	public final Heading getHeading ( ) {
		return heading;
	}

	/**
	 * Gets keySpecifications
	 *
	 * @return keySpecifications.
	 */
	public final Collection<KeyDef> getKeySpecifications ( ) {
		return keySpecifications;
	}

	/**
	 * Gets The predicate corresponding to the nonscalar value held in the container that this
typedeclaration is for
	 *
	 * @return The predicate corresponding to the nonscalar value held in the container that this
typedeclaration is for
	 */
	public final String getRelationPredicate ( ) {
		return relationPredicate;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
be.SIRAPRISE.typeimplementations.TypeDeclaration#getValueBuffer(java.nio.ByteBuffer)
	 */
	@Override
	public NonScalarValueBuffer getValueBuffer (ByteBuffer attributeValueEncoding) {
		switch (getTypeNameIdentifier().getString()) {
			case TYPENAMES.TUPLE: {
				return new TupleBuffer(heading, attributeValueEncoding);
			}
			default: {
				return new RelationBuffer(heading, attributeValueEncoding);
			}
		}
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see java.lang.Object#hashCode()
	 */
	@Override
	public int hashCode ( ) {
```

```
		return hashCode == Integer.MIN_VALUE ? (hashCode = heading.hashCode()) :
hashCode;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see be.SIRAPRISE.typeimplementations.TypeDeclaration#isIntervalTyped()
	 */
	@Override
	public boolean isIntervalTyped ( ) {
		return false;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see be.SIRAPRISE.typeimplementations.TypeDeclaration#isScalar()
	 */
	@Override
	public final boolean isScalar ( ) {
		return false;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see
be.SIRAPRISE.typeimplementations.TypeDeclaration#matchesExpectedTypeDeclaration(be.SIRA
PRISE.typeimplementations.TypeDeclaration)
	 */
	@Override
	public boolean matchesExpectedTypeDeclaration (TypeDeclaration
expectedTypeDeclaration) {
		return
getTypeNameIdentifier().getString().equalsIgnoreCase(expectedTypeDeclaration.getTypeNameIden
tifier().getString()) && getHeading().isNonScalarSubTypeOf(((NonScalarTypeDeclaration)
expectedTypeDeclaration).getHeading());
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see java.lang.Object#toString()
	 */
	@Override
	public String toString ( ) {
		return getTypeNameIdentifier().getString() + getHeading().toString();
	}
}
```