Cascianelli Silvia

Baldi Roberto

CODE INSPECTION

V 1.0

1. Methods assigned to the group

From: appserver/persistence/cmp/model/src/main/java/com/sun/jdo/api/persistence/model/mapping/

1. Name: stripSchemaName( )
   Start Line: 427
   Location: impl/MappingTableElementImpl.java

2. Name: firePropertyChange(String name , Object o , Object n)
   Start Line: 189
   Location: impl/MappingClassElementImpl.java

3. Name: fireVetoableChange(String name , Object o , Object n)
   Start Line: 210
   Location: impl/MappingClassElementImpl.java

4. Name: addTable(TableElement table)
   Start Line: 384
   Location: impl/MappingClassElementImpl.java

5. Name: setPrimaryTable(TableElement table)
   Start Line: 469
   Location: impl/MappingClassElementImpl.java

2. Functional role of assigned set of classes

2.1. impl/MappingTableElementImpl.java

This class allows us to create a representation of a database table; it is a mapping of an element from the class *TableElement*, which is in the database model.
The methods of the class are useful to manage (add, remove, set and get) the columns that compose the primary key and the foreign keys. In addition, there are method to set and to get the table to map, to strip the names (transform the absolute names in relative ones), to make comparison (*isEqual*) and to manage the name of the mapping table.
The primary key and the foreign keys can be set in a different way from the original table; so, it is possible to set “fake” primary and foreign keys.

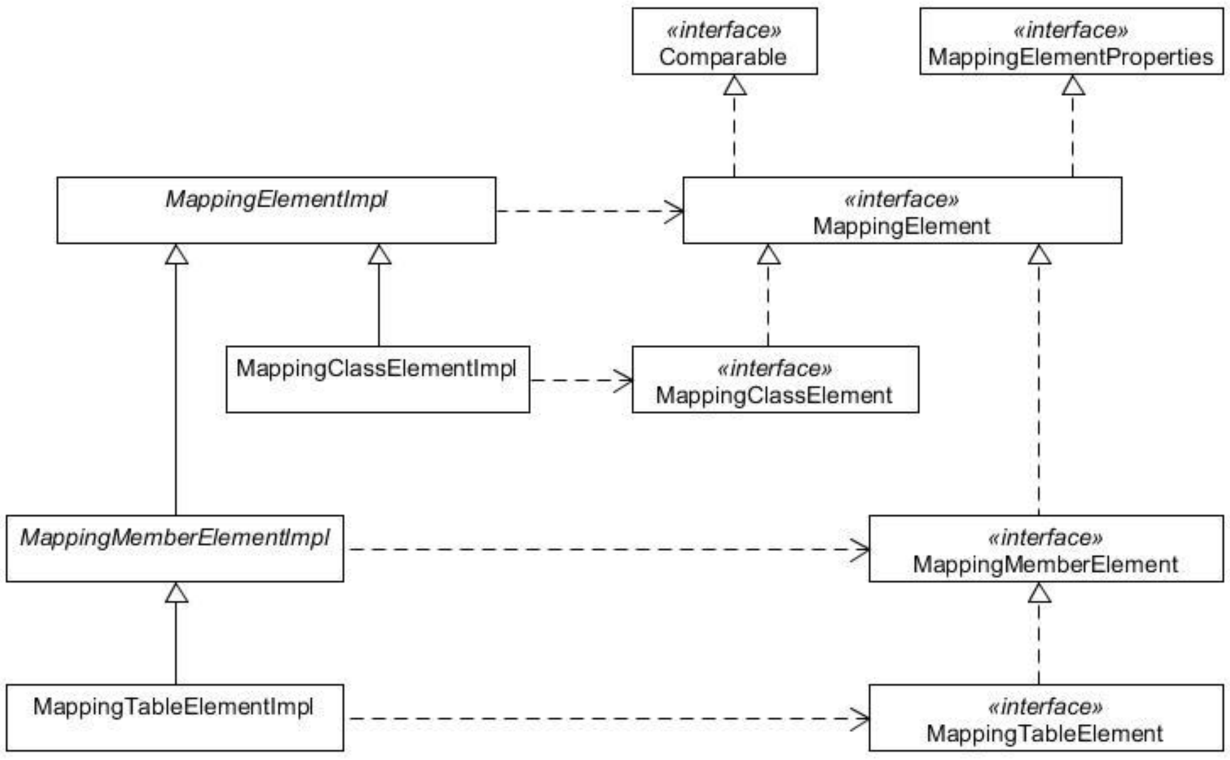
MappingTableElementImpl implements the *MappingTableElement* interface and in addition, it extends the *MappingMemberElementImpl*; that is an implementation of the *MappingMemberElement* interface and extends the abstract class *MappingElementImpl*. This last class implements the method of the *MappingElement* class, which implements two interfaces: *MappingElementProperties* and Comparable.
This was the hierarchy of the class and if we analysis it we can see that all the method are correctly defined for the *MappingTableElementImpl* class; so, there are not errors in the definition and in the realization of the class.

2.2. impl/MappingClassElementImpl.java

This class permits to map an entity to multiple table representations; indeed, the Java Persistence API grants the opportunity of distributing the state of an entity in different parts associated with different tables, as long as these tables are conveniently linked. Therefore, this class offers many methods to obtain, search, add or remove tables (actually MappingTableElement or TableElement objects) associated with the mapping class. Moreover, it has also methods to access or modify the fields of the mapping table and a set/get couple to manage the persistence class element for this mapping class; finally, it deals with other methods that handle and signal property change events.

MappingClassElementImpl implements the *MappingClassElement* interface and moreover, it extends the *MappingElementImp abstract class*; this class implements the method of the *MappingElement* interface, that inherits and extends two interfaces, which are respectively the *MappingElementProperties* and the *Comparable* interfaces. Since all the inherited methods and also all the methods exposed by the interfaces, and to be implemented, are properly defined, the realization of this class is correct and complete.

The UML Diagram below illustrates clearly all the meaningful hierarchical dependencies and implementation links among the classes and the interfaces of interest with respect with both the assigned classes.



3. Functional role of assigned set of method

3.1. stripSchemaName( )

The role of this method is to convert the absolute names of the database elements to the correspondent relative names. This procedure affects the name of the *MappingTableElement* itself and the column names stored in _*k*eys. The method is recursively called for all *MappingReferenceKeyElements* attached to this *MappingTableElement*.
The algorithm acts with the class *NameUtil* (from *org.netbeans.modules.dbschema.util*), which has a static method *getRelativeMemberName* that takes a string and returns a new string. This method splits the string when there is a semicolon and then calls a new method (*getRelativeMemberNameInternal*); it searches a dot, then it creates a substring with the piece of string that comes after the dot and returns this substring. The *getRelativeMemberName* method join all this new piece of string with a semicolon and return it. In this way, the method cancels every information about the absolute name and only the relative part remains.

3.2. firePropertyChange(String name , Object o , Object n)

This method allows the user to report a property change to listeners that are registered; the changed property is identified by the *name* parameter, the parameters *o* and *n* identify the old and the new value of this property.
The method calls its superclass' method, which calls the *firePropertyChange* method of a non-null object of the class *PropertyChangeSupport* (from *java.beans*). This last method will perform the real operations of notification only if the old and the new values are non-null and non-equal.
If the two values are equal, the method will assign to a Boolean variable a true value; if this variable is false and the parameter *name* is not equal to "*modified*", the method *setModified* is called with a true value for the parameter. This is going to modify a class variable (_*isModified*), if the value of this variable really changes, the *firePropertyChange* method will be recall with the "modified" string and the old and new values of the variable _*isModified*.

3.3. fireVetoableChange(String name , Object o , Object n)

This method, like the previous one, is going to report a property update to listeners that are registered; the name of the changed property is defined in the *name* parameter of the method, the *o* and *n* parameters identify the old and the new values of this property.
The method calls its superclass' method, which calls the *fireVetoableChange* method of a non-null object of the class *VetoableChangeSupport* (from *java.beans*). This last method will perform the real operations of notification only if the old and the new values are non-null and non-equal. This method can also throws an exception (*PropertyVetoException*) if one of the listener want to veto; in this case, all the updates are reversed and the exception is thrown again.
If the two values are equal, the method will assign to a Boolean variable a true value; if this variable is false and the parameter *name* is not equal to "*modified*", there will be a recursive call of this method. This call is made with the parameters "*modified*" for the property *name*, false for the old value and true for the new value.
If the method receive the *PropertyVetoException*, it will do nothing but propagates the exception.

3.4. addTable(TableElement table)

This method tries to add a table element as a primary or secondary table for the mapping class of interest, depending on the already existing list of tables and the foreign keys for the table. Indeed, the Java Persistence API grants the opportunity to map the state of a single entity to multiple tables, as long as only one of them is defined as primary table. To be more precise, this method uses only one parameter, which is exactly the table element to be added as either a primary or a secondary table. Moreover, this method can raise exceptions of kind *ModelException*. In fact, first of all it controls that the *TableElement* object passed as parameter is not null; whether the table it attempts to add is null, the method ends immediately throwing a *ModelException* exception with a specific message. In any other case, the method proceeds calling *getTables()* to obtain the list of tables (of kind *MappingTableElements*) already associated with that mapping class. Hence, if this ArrayList is empty the table is certainly set as primary Table for the mapping class, by calling the convenient method *SetPrimaryTable(TableElement table)*, that we will discuss in the next point. Otherwise, the method prepares a HashMap and creates an iterator to scan the ArrayList of tables; through it, the method controls that the table passed as parameter has not been added yet, in such a way that, if the table is already present in the list, the method ends immediately. Otherwise, the method goes on scanning again the ArrayList of tables and each time it extracts the current table representation and gets its foreign keys, conveniently stored in an array of support. At this point, the method starts a loop, depending on the number of foreign keys of the current table. The goal of each turn of this cycle is comparing the table referred by each foreign key with the one passed as parameter; notice that doing this the method *getReferencedTable()* of the class *org.netbeans.modules.dbschema.ForeignKeyElement* is used. In this way, every time a foreign key, belonging to a table already existing in the list of tables, refers to the table passed as parameter at the beginning, a new pair containing the foreign key and the corresponding mappingTable of the list is put into the HashMap created previously. Then, if no foreign keys have been found the method ends throwing a Model Exception, whereas if at least a foreign key has been found, an iterator is created to scan the collection of mappingTable elements contained in the HashMap and returned by the method *keySet()*. During this operation the method *addSecondaryTable(…)* associates the current table, taken into account through the iterator, with the supplied table, creating respectively a primary/secondary table pair. In particular, this method call sets the supplied table as a secondary table for the mapping class and returns an object of kind *MappingReferenceKeyElement*, called refKey, which represents the relationship between the tables of the considered primary/secondary table pair. Actually, this element can be thought as a "fake foreign key", because it designates the column pairs used to join the primary table with the secondary table. At this point, the code considers each *ForeignKeyElement* instance associated with the current *MappingTable* table in the HashMap, gets all its column pairs and finally adds them as column pairs for the refKey.

3.5. setPrimaryTable(TableElement table)

This method sets the primary table for this mapping class to the supplied table, passed as parameter. It may also terminate throwing an exception of kind *ModelException,* whether it does not succeed in achieving its purpose. First of all, it controls the list of tables associated with the mapping class is effectively empty, and whether it is not the method ends immediately throwing a *ModelException* which specifies the primary table is already existing. In any other case, the method proceeds extracting the primary key of the supplied table and creating a new *MappingTableElement* instance called mappingTable (actually a *MappingTableElementImpl* concrete object) with the supplied table as corresponding table and with the mapping class into consideration as declaring class. Then it saves in a proper variable, of kind *SchemaElement*, the schema represented in memory that owns this table element. Furthermore, it obtains the current database root, by calling the method *getDatabaseRoot()*; indeed, this latter returns the name of the *SchemaElement* object which represents the database used by the tables mapped to this mapping class. Since the primary table of the mapping class should not have been defined yet, the returned value for the current root should be null and ready to be set equal to the schema corresponding to the supplied table. Whether instead it is not null, it must match with the schema of the supplied table in order not to raise a *ModelException* caused by the mismatch. Anyway, supposing this step of setting the database root concludes successfully, the method goes on with a try –catch block, responsible for adding the supplied table in the ArrayList of tables for the mapping class. Actually, the block calls also two methods *fireVetoableChange(…)* and *firePropertyChange(…)* which are dedicated to report the update and change of a property, as discussed above. However, these methods work on the static final property PROP_TABLES and have null parameters as both old and new values; hence, the real operations of notification is not performed. Anyway, the block ends catching the potential *PropertyVetoException*, related to the veto of a registered listener, and possibly throwing a *ModelVetoException*, which is a sub-class of the generic *ModelException* class.
At this point, when the method continues, the code controls that the variable *key* taken out at the beginning is not null. In this latter case, the method extracts all the unique keys of the supplied table and, as long as the obtained array has at least one *UniqueKeyElement* object, the variable *key* is set equal to the first unique key of this array. Then, whether the key is still null the method ends but none update operations can be performed on this table, though it is associated with the mapping class, until the key is not defined. In any other case instead, all the columns of the key of interest are added as columns to the primary key of columns in the mappingTable created at the beginning.

4. List of issues found by applying the checklist

4.1. Classes

4.1.1. impl/MappingTableElementImpl.java

Naming Conventions	The name of this method is coherent with its usage; indeed, it is a mapping of *TableElement* and it is an implementation of the *MappingTableElement* interface. The class variables have not meaningful names; they represent the meaning in a bad way, longer name with more detail can be more useful. This can be seen like a violation of the first point.
Indention	The indention is always made with tabs and never with spaces. It does not respect the norms, which are define at the eighth and the ninth points.
File Organization	The different section of the java file are divided by blank lines; the comments are used to describe elements of the code but also to divide in different block the methods, this division is made by actions performed.
Comments	The comments are more important because the names of the variables are not enough to understand the meaning. They do not exceed the eighty character of the specification but are able to describe in a good way all the elements.
Java Source Files	The *MappingTableElementImpl* is the only class present in the java file, it is a public class and it is declared in the right form. All the Javadoc comments are complete and they cover every elements.
Package and Import Statements	At the top of the file, there is a comment that gives some information about the file, the version and the copyright. Then there is the package definition and after a blank line, there are all the import instruction for all the needed elements.
Class and Interface Declaration	All the variables are private and for this reason, it is sure that they respect the specify order for them. The method are well organized in different block that are divided by scope and it is another point of the specification. The method never are too long and the class is made to be very cohesive; in facts, it is define to manage a very specific element.
Initialization and Declaration	All the variable of the class are declared in the first statements and the visibility is coherent with the usage that they have.
Arrays	All the arrays (*ArrayList*) of the class are not initialized in an explicit way; but before the action to get, add or remove them there is always (unless for the get-method) a call to the get method. The get-method performs a control to verify that the array is not null but if it is, it is created a new empty array; in this way it is always sure that the array is not null but it exists. All of this is allowed by the specification in the thirty-ninth point.

4.1.2. impl/MappingClassElementImpl.java

Naming Conventions	The name of this class is appropriate with respect to its functionality; actually it is the concrete implementation of a mapping of a persistent entity, that can be distributed among multiple table representations. The class and method variables have names which are coherent with respect to their roles and usages, though, in certain case, these names are not enough self-explanatory. Anyway, all the attribute or method names are properly formed following the naming conventions, as specified in the points 5 and 6 of the checklist.
Indention	For the indention, tabs are always used, in opposition to the rules that define to use three or four spaces, as specified at points 8 and 9 of the checklist.
Braces	The Allman style for braces is predominantly used in this class; anyway there are also some methods that use the Kernighan and Ritchie style, such as for instance getVersionNumber () [line 154]. Hence, the bracing style adopted is not always the same throughout the code of this class and this disregards the point 10 of the checklist. Moreover, also the point 11 of the checklist is not respected at all, as we can see clearly from the following extract [line 199]: *if (!(PROP_MODIFIED.equals(name)) && !noChange)* *setModified(true);…* In fact, this if-statement, that has only one consecutive instruction to be executed, does not surround this latter with curly braces. The same happens also for all the analogous cases throughout the code of this class.
File Organization	The code of this class is conveniently divided in different sections through blank lines and optional comments; this latter, together with the javaDoc comments are used to better explain the roles of methods and attributes or to clarify some parts of the implementation itself. Anyway, each line of comment does not exceed 80 characters, according with what is specified in the point 13 of the checklist.
Wrapping Lines	All the statements are correctly aligned with the beginning of the expression at the same level at the previous line.
Comments	The comments are widely used because the names of variables and methods are often not sufficient to understand the meaning and the behaviour of certain blocks of code.
Java Source Files	The Java source file taken into account contains only the *MappingClassElementImp.* It is a public class correctly declared and which defines conveniently all the methods coming from the implemented *MappingClassElement* interface (and its super-interfaces). Finally,all the Javadoc comments are complete and they cover every parts of the code of this class.
Package and Import Statements	At the beginning of the file, there are many lines of comment concerning the file version, the copyright and the contributors. Anyway, the package definition statement is correctly the first non-comment statement and it is followed by a blank line and then by all the import statements. Hence, the point 24 of the checklist is appropriately respected.

Class and Interface Declaration	The declaration of the class follows directly the initial comment, mentioned above, and the documentation comment about authors and version; the declaration of the class, in its turn, is followed by the declaration of the static variables, which are all public in this case. Then the code proceeds with the instance variables, which are all private in this case. Finally we can see in order the constructors and then all the other methods, grouped by functionality and lacking of useless redundancies. Hence, all is properly arranged according with the corresponding points in the checklist (25-27)
Initialization and Declaration	All the variables of the class are declared among the first statements and each visibility is suitably thought to be appropriate with respect to the scope. Furthermore the variables are also properly initialized either when they are declared, if they are static final attributes, or with appropriate setting methods, called before their use.

4.2. Methods

4.2.1. stripSchemaName( )

Naming Conventions	The name of the method is coherent with its functionality; in facts, it has to remove some parts of the absolute name to convert it in a relative one and we can say that it strips the name of some information. The names of the variables used are in the right form; the variables declared in this method are two and both have the same name that identifies their nature of iterators.
Indention	The indention is made with the tabs and it does not respect the norms that are defined in the points eight and nine.
Braces	The if-statements and the first while-loop are written in the right form with the Allman style; but the last while-loop has the following statement that is not surrounded by curly braces.
Wrapping Line	Between the lines 444 and 445, there is a line break and the code wraps in the middle of an instruction; however, it is not a problem because this line break occurs after an equality operator.
Comments	The comments respect the default size; they are very useful to explain the actions of the code and they do it in a very easy way.
Initialization and Declaration	All the internal variables are declared in the right way; in facts, the declaration occurs always at the beginning of a new block of instructions.
Method Calls	All the method are called in the right way; the calls have the right number, type and position of the parameters.
Exceptions	The only issue can be found in the method *substring* that is called by the *getRelativeMemberNameInternal*. This call can throw an exception that is names *StringIndexOutOfBoundsException*. However, this scenario cannot become true because the caller method always passes right parameters.
Flow of Control	There are two loops but they do not create any issues because they explore iterators. The exploration is well done, it uses the commands *hasNext* and *next* that guarantee an end at the cycle.

4.2.2. firePropertyChange(String name , Object o , Object n)

Naming Conventions	The name of this method is coherent, it has to alarm about a property change and the name allows us to understand it. The name of the only variable declared in the method has an explicit meaning, it has to represent if the change is really occurred or not.
Indention	The indention is made in the wrong way; the code always has tabs and not the spaces. It does not respect the norm, which specifies this aspect at the eighth and the ninth points.
Braces	The if-statement is written in the wrong form because the curly braces do not follow it. There is only an instruction but it does not respect the norm at the eleventh point.
Comments	All the comments respects the constraint, they have less of eighty characters and they are useful to explain some fragments of the code.
Initialization and Declaration	The only variable declared is define at the beginning of the method, so it respects the specification because the declaration occurs at the beginning of a block of instructions.
Method Calls	The calls of other method is made in correctly, all the types of the parameters and the positions are correct.
Object Comparison	In this method, there is a comparison between objects and it is made in the right way, with an *isEquals* method.
Computations, Comparisons and Assignments	The parenthesis are not always used to avoid precedence problems; in facts, sometimes are useless because are used to specify a precedence where it is already present. This is not coherent with the forty-sixth point, which says that the parenthesis must use to avoid precedence problems and it does not mention others usages.

4.2.3. fireVetoableChange(String name , Object o , Object n)

Naming Conventions	The name of this method is coherent, it has to alarm about a change of a vetoable property and the name allows us to understand it. The name of the only variable declared in the method has an explicit meaning, it has to represent if the change is really occurred or not.
Indention	The indention is wrong; the code always uses tabs and never the spaces. It does not respect the specification, which is define at the eighth and the ninth points.
Braces	The if-statement has a wrong form because the following statement is not contain between curly braces. There is only an instruction but it does not respect the specification at the eleventh point.
Comments	All the comments respects the constraint, they have less of eighty characters and they are useful to explain some fragments of the code.
Initialization and Declaration	The only variable declared is define at the beginning of the method, so it respects the norms because the declaration occurs at the beginning of a block of instructions.
Method Calls	The calls of other method is always correct, all the types of the parameters are correct and they are in the correct position. It is correct also when the method calls recursively itself.
Object Comparison	In this method, there is a comparison between objects and it respects the specification, it is done with an *isEquals* method.
Computations, Comparisons and Assignments	The parenthesis are not always used to avoid precedence problems; in facts, sometimes are useless because are used to specify a precedence where it is already present. This is not coherent with the forty-sixth point, which says that the parenthesis must use to avoid precedence problems and it does not mention others usages.
Exceptions	This method calls its superclass in the code and its superclass' method can throw a *PropertyVetoException*. This method does not manage this exception, but it propagates it; so, this method throws the same exception of its superclass' method. It is a correct management of an exceptional situation because the exception is thrown when a listener vetoes the update of the property; in this case is coherent that the exception return to who calls this method and it is allowed to manage the problem.

4.2.4. addTable(TableElement table)

Naming Conventions	The name of this method is appropriate with respect to its role; moreover also all the local variables have meaningful names related to their own utilization and respect the naming conventions. The only one-character variable *i* is conveniently used uniquely to handle a loop [from line 415], according with the point 2 of the checklist. A local variable with a name which is not enough self-explanatory is *fk*, belonging to the class ForeignKeyElement, [line 420]. Anyway this variable is used to extract, one after the other, the foreign keys of the table taken into consideration at that moment; hence its name, although is not exhaustive, represents an abbreviation for foreign key. Notice that a similar kind of abbreviation regards also the variable r*efKey* of kind MappingReferenceKeyElement [line 441].
Indention	The indention is always realized using tabs instead of three or four spaces. Hence, the points 8 and 9 of the checklist are violated.
Braces	This method adopts the Allman style for braces coherently throughout all the lines of code. But as far as the if-statements are concerned, when they are followed only by a single instruction to be executed, this latter is never surrounded by curly braces, as we can see for instance in this extract [lines 391-392] *if (tables.isEmpty())* *setPrimaryTable(table);* Hence, the point 11 of the checklist is not respected at all.
File Organization	The comments and the blank lines are used to separate sections performing different alternatives or different steps of the behaviour of this method. Anyway, each line of comment does not exceed 80 characters, according with what is specified in the point 13 of the checklist.
Wrapping Lines	All the statements are correctly aligned with the beginning of the expression at the same level at the previous line.
Comments	The comments are widely used to better explain the scope of this method and to clarify some parts of the implementation itself, as for instance at line 390: *// If the table list is empty, this should be the primary table* or at lines 424-5-6: *// store it so it can be added after we finish* *// iterating the array (can't now because of* *// concurrent modification restrictions)*
Java Source Files	The method of interest belongs to the Java source file containing only the *MappingClassElementImp.* In particular, it represents the concrete implementation of a method that the class *MappingClassElementImp* acquires from the implemented *MappingClassElement* interface. Finally, the Javadoc comments are complete, since they specify the role, the needed parameter and the possible exception the method could throw.
Initialization and Declaration	All the local variables of this method are correctly inizialized as soon as they are declared and also the only needed parameter, which is of kind TableElement is properly controlled, in such a way that if it is null, the method ends immediately.
Method Calls	All the method calls occur in this piece of code are correct, since the parameters always belong to the suitable classes and appear in the right order in the invocation. Furthermore, every returned value which is meaningful for the implementation of the method is properly assigned to a variable of the same kind, as for instance in the extract below [lines 411-2]:

	String absoluteTableName = NameUtil.getAbsoluteTableName(*_databaseRoot, mappingTable.getTable());* As we have said, the static method of the *NameUtil* class *getAbsoluteTableName*(…), with the correct list of parameters, returns a string representing the absolute name of a table, and this returned value is conveniently assigned to a variable of kind String properly called *absoluteTableName.*
Arrays	The method scans the array *foreignKeys* of kind *ForeignKeyElement,* through a for-loop. This array is obtained at line 413 as result of a method call and it could be even null. However, the number of consecutive cycles for the for-loop is conveniently set according with the exact length of the array foreignKeys; therefore there is no loop if the array is null whereas in any other case the indexes used for scanning it are certainly prevented from going out of bounds, as required by the point 38 of the checklist.
Object Comparison	Not every object comparison in the method code takes place with "equals", and therefore the point 40 of the checklist is not respected. For instance, at line 422 there is an if statement in which the condition is expressed by a comparison using "==": *if (table == fk.getReferencedTable())*
Output Format	The method does not provide displayed output except for error messages in the event that it throws an exception. In that case, the exception is created specifying as parameter a message which is comprehensive and suitable to clarify the problem has occurred, as we can notice in the following extracts: [lines 452-3-4] *throw new ModelException(I18NHelper.getMessage(* *getMessages(),* *"mapping.table.foreign_key_not_found", table));* [lines 460-1] *throw new ModelException(I18NHelper.getMessage(getMessages(),* *"mapping.table.null_argument"));*
Computation, Comparisons and Assignments	The implementation offered for this method does not use brutish programming; moreover, the use of parenthesis is suitable to specify all the necessary precedencies or explicit cast. The use of comparisons is correct and also the code of throwing exception is correctly performed.
Exceptions	The method does not provide catch block to handle the exception eventually raised. Anyway, this method can throw exceptions of kind ModelException in case the parameter is null or whether the foreign key referring to the parameter table is not found.
Flow of Control	The method contains two different while loops, correctly formed since both are related to an iterator involved in scanning an ArrayList, which is accurately controlled to be not-empty. Furthermore, the second while loop contains also an inner for-loop, which defines properly the initialization and increasing of the counter variable *i* as well as the designation of a suitable termination condition, related to the length of the array to scan. It is shown clearly in the following extract [from line 415]: *int i, count =* *((foreignKeys != null) ? foreignKeys.length : 0);* *for (i = 0; i < count; i++)* *{…*

4.2.5. setPrimaryTable(TableElement table)

Naming Conventions	The name of this method is suitable considering the aim it pursues; furthermore, also all the names adopted for the local variables are coherent with their roles and enough self-explanatory and they respect the naming conventions. The only one-character variable *i* is conveniently used uniquely to handle a loop [from line 525], according with the point 2 of the checklist.
Indention	The indention always uses tabs, in opposition to the rules specified at points 8 and 9 of the checklist, that require to indent using three or four spaces.
Braces	This method adopts the Allman style for braces coherently throughout all the lines of code. But, as far as the if-statements are concerned, when they are followed only by a single instruction to be executed, this latter is never surrounded by curly braces, as we can see for instance in these extracts: [lines 486-487] *if (currentRoot == null)* *setDatabaseRoot(schema); …* [lines 512-513] *if ((uniqueKeys != null) && (uniqueKeys.length > 0))* *key = uniqueKeys[0];…* Hence the point 11 of the checklist is not respected at all.
File Organization	Once again, the comments and the blank lines are used to separate sections performing different alternatives or different steps of the behaviour of the method. Anyway, the length of each line of comment never exceeds 80 characters, according with the point 13 of the checklist.
Wrapping Lines	All the statements are correctly aligned with the beginning of the expression at the same level at the previous line.
Comments	The comments are intensively used to specify some aspects of the method all over all the code; in particular some of them are really meaningful and useful to better explain the scope of a certain part of code, as for instance at line 490: *// if database root was set before, it must match* or at lines 518-19-20: *// This is a warning -- we can still use the table but we* *// cannot perform update operations on it. Also the user* *// may define the key later.*
Java Source Files	The method of interest belongs to the Java source file containing only the *MappingClassElementImp.* In particular, it represents the concrete implementation of a method that the class *MappingClassElementImp* acquires from the implemented *MappingClassElement* interface. Finally, the Javadoc comments are complete, since they specify the role, the needed parameter and the possible exception the method could throw.

Initialization and Declaration	All the local variables of this method are correctly inizialized as soon as they are declared, by calling proper methods that return objects of the desired kind; furthermore, every time a returned value could be null, this one is properly controlled before any possible utilization, as we can see in the following examples:
	[lines 484-5-6]
	String currentRoot = getDatabaseRoot();
	
	if (currentRoot == null) …
	[lines 480 and 508]
	UniqueKeyElement key = table.getPrimaryKey();

	…
	if (key== null) …
	[lines 510-1-2]
	UniqueKeyElement[] uniqueKeys = table.getUniqueKeys();
	
	if ((uniqueKeys != null) && (uniqueKeys.length > 0)) …
	Even the variable *i*, which is used only to scan the array *columns* in the for loop starting at line 527, is initialized before the cycle, together with the other variable *count*; nevertheless at the beginning of the for loop its value is modified and set equal to zero, as we can notice in the following extract:
	[from line 525]
	int i, count = ((columns != null) ? columns.length : 0);
	
	for (i = 0; i < count; i++)…

Method Calls	The method calls that occur in this piece of code are always correctly performed; indeed all the needed parameter belong to proper classes and are presented in the right order. Moreover, the returned value obtained from each method call is accurately assigned to a variable of the same kind, in such a way that it can be properly used later on in the method implementation. For instance, this aspect appears clearly in the extract below: [lines 480-4]
	UniqueKeyElement key = table.getPrimaryKey();
	MappingTableElement mappingTable =
	    new MappingTableElementImpl(table, this);
	SchemaElement schema = table.getDeclaringSchema();
	String currentRoot = getDatabaseRoot();

Arrays	The method considers first of all the ArrayList *tables* which is obtained through the method *getTables()* and that should be empty to properly carry on with the desired implementation. Furthermore, the method involves also two different arrays, actually *uniqueKeys* of kind *UniqueKeyElement* and *columns* of kind *ColumnElement*. Both are obtained as result of two different method calls, respectively *table.getUniqueKeys()* and *key.getColumns()*. Hence, since these arrays could be also null, this aspect is properly controlled in the code before performing any kind of action over them. Finally, as far as the array *columns* is concerned, its exact length is evaluated in order to set the number of consecutive cycles of a for-loop equal to this length. In this way, as long as *columns* is not null, the method scans this array through the mentioned for loop assuring that indexes used for scanning are prevented from going out of bounds.
Object Comparison	Only one object comparison in the method code takes place with "equals", actually at line 488: *else if (!currentRoot.equals(schema.getName().getFullName()))…* All the other comparisons are used to compare an object with *null,* in a positive or negative sense, and they are realized through a condition expressed using "==" or "!=". Hence, the point 40 of the checklist is not respected.
Output Format	The method does not provide displayed output except for the error messages in the event that it throws an exception related to specific causes. In fact, in the case that the exception derives from the already existence of a primary table or from a mismatch of the schemas for the database root, an exception of kind *ModelException* is created specifying as parameter a message, suitable to clarify the problem has occurred. It appears clearly in the following extracts:

	[lines 475-6] *throw new ModelException(I18NHelper.getMessage(getMessages(),* *"mapping.table.null_argument"));…* [lines 491-2-3] *throw new ModelException(I18NHelper.getMessage(* *getMessages(), "mapping.table.schema_mismatch",* *table.toString(), currentRoot)); …*
Computation, Comparisons and Assignments	The implementation offered for this method does not use brutish programming and the use of parenthesis is not redundant but suitable to specify all the necessary precedencies or explicit cast. The use of comparisons is efficient and also the code for throwing and catching exception is correctly performed
Exceptions	The method provides only one try-catch block to handle the *PropertyVetoException* that the method *fireVetoableChange*(…) can eventually raise. Anyway, the method limits itself to catch this kind of exception and then to launch a new kind of exception to be propagated, actually a *ModelVetoException*, which is a sub-class of the *ModelException* class. Hence, though the behaviour performed to manage this kind of exception is not meaningful, it is at least appropriate. Furthermore, as we have already discussed, this method can throw also other exceptions of kind *ModelException*, respectively in case the mapping class has already a primary key or in case the schema of the supplied table does not match with the database root.

Flow of Control	The method contains only a for-loop, which defines properly the initialization and increasing of the counter variable *i* as well as the designation of a suitable termination condition, related to the length of the array to scan. It is shown clearly in the following extract [from line 525]:
	int i, count = ((columns != null) ? columns.length : 0);
	
	for (i = 0; i < count; i++)
	mappingTable.addKeyColumn(columns[i]);
	Hence, the point 56 of the checklist is properly respected.