Politecnico di Milano

2015-2016

Software Engineering 2

Code Inspection

version 1.0

Author: Nenad Petrovic

27th December 2015

Contents

1 Assigned classes and methods

In this assignment, a part of GlassFish Server source code is assigned and is going to be inspected according to the specified checklist. The following methods are assigned for code inspection:

Name: processMaybeNullOperation(List stack , StringBuffer result)

Start Line:

714

End Line:

789

Location:

appserver/persistence/cmp/support-sqlstore/src/main/java/com/sun/jdo/spi/persistence/support/sqlstore/sql/generator/Statement.java

Name: processNullOperation(int opCode , List stack , StringBuffer result)

Start Line:

791

End Line:

822

Location:

appserver/persistence/cmp/support-sqlstore/src/main/java/com/sun/jdo/spi/persistence/support/sqlstore/sql/generator/Statement.java

All of the assigned methods, as it can be seen, belong to the same .java file – „Statement.java“. This file contains one public abstract class:

public abstract class Statement

The corresponding package is:

package com.sun.jdo.spi.persistence.support.sqlstore.sql.generator;

A *package* is a named collection of classes (and possibly subpackages). Packages serve to group related classes and define a namespace for the classes they contain.

2 Functional role of assigned set of classes

This chapter is going to deal with functional role of assigned class (only one class is asigned), related classes and assigned methods.

The assigned class is public abstract class named „Statement“. This class is used to represent a SQL statement and belongs to part of GlassFish Server related to persistence.

Now, let’s take a look on the class structure and its members – methods and attributes.

2.1 Public Member Functions

void	addConstraint (LocalFieldDesc lf, Object value)
void	addQueryTable (QueryTable table)
void	addSecondaryTableStatement (Statement s)
void	appendTableText (StringBuffer text, QueryTable table)
void	bindInputValues (DBStatement s) throws SQLException
Object	clone ()
int	getAction ()
ArrayList	getColumnRefs ()
String	getFormattedSQLText ()
abstract QueryPlan	getQueryPlan ()
ArrayList	getQueryTables ()
ArrayList	getSecondaryTableStatements ()
String	getText ()
DBVendorType	getVendorType ()
StringBuffer	processConstraints ()
void	setAction (int action)
	Statement (DBVendorType vendorType)

2.2 Public Attributes

ArrayList	tableList

2.3 Protected Member Functions

void	addColumnRef (ColumnRef columnRef)
void	appendQuotedText (StringBuffer buffer, String text)
QueryTable	findQueryTable (TableElement tableElement)
void	generateColumnText (LocalFieldDesc desc, QueryPlan thePlan, StringBuffer sb)
void	generateInputValueForConstraintValueNode (ConstraintValue node)
abstract void	generateStatementText ()
ColumnRef	getColumnRef (ColumnElement columnElement)
QueryPlan	getOriginalPlan (ConstraintField fieldNode)
String	getWhereText (List stack)
String	infixOperator (int operation, int position)
int	operationFormat (int operation)
String	postfixOperator (int operation)
String	prefixOperator (int operation)
void	processConstraintParamIndex (ConstraintParamIndex node, StringBuffer result)
void	processConstraintValue (ConstraintValue node, StringBuffer result)
void	processIrregularOperation (ConstraintOperation opNode, int opCode, List stack, StringBuffer result)
void	processRootConstraint (ConstraintOperation opNode, List stack, StringBuffer whereText)

2.4 Static Protected Member Functions

static String	formatSqlText (String sqlText, Object[] input)

2.5 Protected Attributes

ArrayList	columns
InputDesc	inputDesc
ArrayList	secondaryTableStatements
StringBuffer	statementText
DBVendorType	vendorType

2.6 Static Protected Attributes

static final ResourceBundle	messages
static final int	OP_BINOP_MASK = 2 * OP_PARAM_MASK | OP_WHERE_MASK | OP_INFIX_MASK
static final int	OP_FUNC_MASK = OP_PARAM_MASK | OP_PREFIX_MASK | OP_PAREN_MASK | OP_WHERE_MASK
static final int	OP_INFIX_MASK = 0x002
static final int	OP_IRREGULAR_MASK = 0x040
static final int	OP_ORDERBY_MASK = 0x010
static final int	OP_OTHER_MASK = 0x080
static final int	OP_PARAM_MASK = 0x100
static final int	OP_PAREN_MASK = 0x008
static final int	OP_PCOUNT_MASK = 3 * OP_PARAM_MASK
static final int	OP_POSTFIX_MASK = 0x004
static final int	OP_PREFIX_MASK = 0x001
static final int	OP_WHERE_MASK = 0x020

2.7 Package Attributes

int	Action
Constraint	Constraint

2.8 Private Member Functions

Object[]	getInputValues ()
void	processConcatOperation (int opCode, List stack, StringBuffer result)
void	processConstraintField (ConstraintField fieldNode, StringBuffer result)
void	processConstraintOperation (ConstraintOperation opNode, List stack, StringBuffer result)
void	processFunctionOrBinaryOperation (int format, int opCode, List stack, StringBuffer result)
void	processInOperation (int opCode, List stack, StringBuffer result)
void	processMaybeNullOperation (List stack, StringBuffer result)
void	processNullOperation (int opCode, List stack, StringBuffer result)

2.9 Static Private Member Functions

static ColumnElement	getColumnElementForValueNode (ConstraintValue node)

2.10 Private Attributes

String	quoteCharEnd
String	quoteCharStart

2.11 Static Private Attributes

static final Integer	ONE = new Integer(1)

The assigned methods for code inspection are

void processMaybeNullOperation (List stack, StringBuffer result)

void processNullOperation (int opCode, List stack, StringBuffer result)

The first one is used like this:

switch(opCode)

---

case ActionDesc.OP_MAYBE_NULL:

processMaybeNullOperation(stack, result);

break;

Considering the ActionDesc.OP_MAYBE_NULL, we can see that inside the ActionDesc interface that this constant is used to identify queries on nullable columns mapped to primitive fields.

The second method is used like this:

switch (opCode) {

---

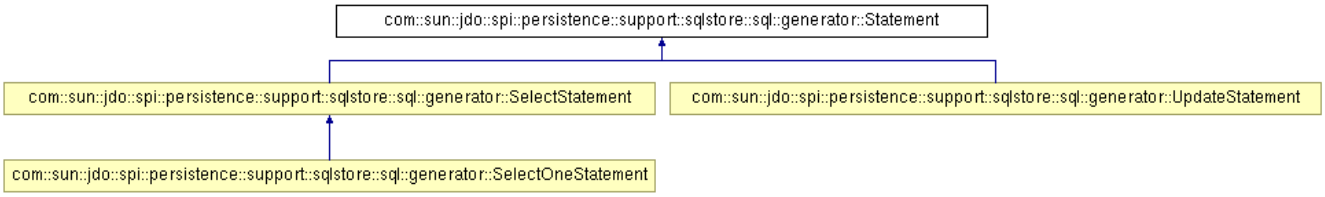
case ActionDesc.OP_NOTNULL:

processNullOperation(opCode, stack, result);

break;

ActionDesc.OP_NOTNULL is constant used to identify unary operator for checking non-null value.

Let us take a look at inheritance diagram displayed below:



As it can be seen, SelectStatement and Update Statement are classes which inherit the assigned class and are used to represent the corresponding types of SQL statements.

3 List of issues found by applying the checklist

In this chapter, the relevant checklist is going to be presented, together with the results of the inspection for each part of the checklist. It is going to be stated if the issue exists and the number that belongs to the corresponding code line where the issue is found. If issue exists, it is going to be discussed. If there is no issue, it is going to be stated that there is no issue, but also potential issues are going to be discussed even if they not clearly exist. Also, it is going to be stated if such case doesn't exist in assigned method/class.

Code inspection checklist and issues

3.1 Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.

   No issues.

2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.

   No real issues.

   There is only one temporary one-character variable in the assigned methods. It is not used as a loop counter or inside loop, but is throwaway variable that only „lives" inside one if statement, so it can't considered as a real issue.
   In what follows, a part of the code which includes this case is going to be extracted.

   Starting from line number 740, we have:

```
if (value instanceof String) {
        String v = (String) value;

        if (v.length() == 0) {
           stack.add(fieldNode);
           stack.add(new ConstraintOperation(ActionDesc.OP_NULL));
        } else {
           stack.add(valueNode);
           stack.add(fieldNode);
           stack.add(new ConstraintOperation(ActionDesc.OP_EQ));
        }
    }
```

3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;

   No issues.

   There is only one assigned class (public abstract class), and it is named „Statement“, so there are no issues with this.

4. Interface names should be capitalized like classes.

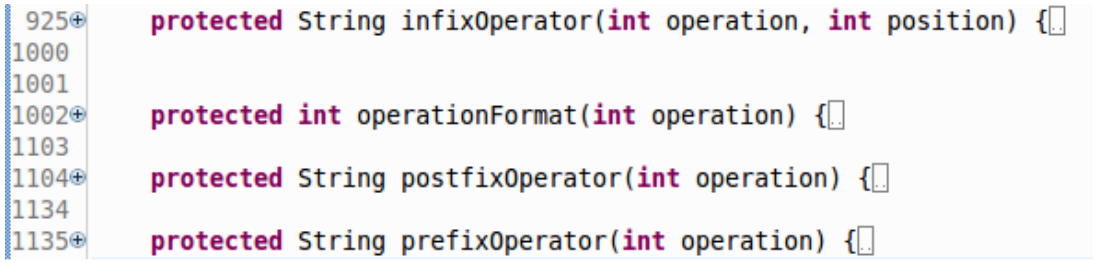
   No issues.

   There is no any interface defined in this file.

5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().

   Issues found.

   As it can be seen below, the methods starting from lines:
   925,1002,1104 and 1135 don’t start with verbs (they start with nouns and adjectives), so this could be an issue, according to the checklist.



6. Class variables, also called attributes, are mixed case, but might begin with an underscore (‘_’) followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.

   No issues.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH; MAX_HEIGHT;

   No issues.

   Constants are all upercase and spearated by an underscore, for example:

   protected static final int OP_PREFIX_MASK = 0x001;

3.2 Indention

8. Three or four spaces are used for indentation and done so consistently

No issues.

Four spaces are used consistently.

9. No tabs are used to indent

No issues.

Tabs are only used in some of the comments but not for indentation. /

3.3 Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

No issues.

Allman style (left) , Keringhan and Ritchie style (right) are illustrated below.

```
$ indent source.c
$ cat source.c
#include <stdio.h>

int
main ()
{
  if (3 == 3)
    {
      printf ("Yes\n");
    }
  else
    {
      printf ("Neah\n");
    }
  return 0;
}
$
```

```
$ indent -kr source.c
$ cat source.c
#include <stdio.h>

int main()
{
    if (3 == 3) {
        printf("Yes\n");
    } else {
        printf("Neah\n");
    }
    return 0;
}
$
```

"Kernighan and Ritchie" style is used in this code and is used consistently in this code, so there are no issues related to bracing.

11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example:

Avoid this:

```
if ( condition )        doThis();
```

Instead do this:

```
if ( condition )
{
  doThis();
}
```

No issues. The code satisfies this rule and is consistent.

3.4 File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

No real issues here, but comparing comments with comment starting from line number 756 with all other comments, it can be seen that it has a blank line before and after the rest of the code, while other comments don't follow this rule.

13. Where practical, line length does not exceed 80 characters.

No real issues related to this one.

There are lines that exceed 80 characters (lines numbered by 92,806, 815, 820, 824, for example), but it is not so practical to break these lines.
In fact, it could be done, but is not a real issue and depends only on coder's decision.
For example:
String str = (opCode == ActionDesc.OP_NULL) ? vendorType.getIsNull() : vendorType.getIsNotNull();
This line could be broken like:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

No issues.

There is no any line that exceeds 120 characters in this file.

3.5 Wrapping Lines

15. Line break occurs after a comma or an operator.

No issues.

Line breaks occur after comma or an operator (=, &&, for example) in the assigned code.

16. Higher-level breaks are used.

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

In what follows, two examples are going to be presented:

```
someMethod(longExpression1, longExpression2, longExpression3,
      longExpression4, longExpression5);

var = someMethod1(longExpression1,
           someMethod2(longExpression2,
                longExpression3));
```

The two examples apply breaking to an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

For example, in lines 800 and 801:

```
    ConstraintFieldName fieldNode =
            (ConstraintFieldName) nextNode;
```

This break is not after a comma, nor before the operator.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Taking a look at stated rules again:

- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

No issues.

Let's take a look at previous example:

```
    ConstraintFieldName fieldNode =
            (ConstraintFieldName) nextNode;
```

It can be seen that in lines 800 and 801 the second rule is used and line is aligned by 8 spaces, so there is no issue, because antoher rule is used.

3.6 Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

No issues.

There are comments that are used to make some things that are not so obvious more clear (constant values and some other values, as exception numbers, for example), so they are not only limited to what blocks are doing, but also could stand for some further explanations .

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

There is an issue related to previously mentioned part of the code (which is not a part of the assigned methods, by the way).

```
445         for (Iterator iter = tableList.iterator(); iter.hasNext() && table == null; ) {
446             QueryTable t = (QueryTable) iter.next();
447             if (t.getTableDesc().getTableElement() == tableElement) {
448             // if (t.getTableDesc().getTableElement().equals(tableElement)) {
449                 table = t;
450             }
451         }
```

Considering the line 448, we can see that there is no reason mentioned why it is commented, nor the date when it can be removed.

3.7 Java Source Files

20. Each Java source file contains a single public class or interface.

No issues.

There is only a single public class inside this java source file (Statement.java).

21. The public class is the first class or interface in the file.

No issues.

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

No issues.

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

No issues.

3.8 Package and Import Statements

24. If any package statements are needed, they should be the first noncomment statements. Import statements follow.

No issues.

As we can see below, the first noncomment statement is package statement, and import statement is next.

```
/*
 * Statement.java
 *
 * Created on March 3, 2000
 *
 */

package com.sun.jdo.spi.persistence.support.sqlstore.sql.generator;

import org.netbeans.modules.dbschema.ColumnElement;
```

3.9 Class and Interface Declarations

25. The class or interface declarations shall be in the following order:

A. class/interface documentation comment

No issues.

B. class or interface statement

C. class/interface implementation comment, if necessary

B. and C. are swapped in this case. First, the comment is written, and after that, class statement is written. According to rules mentioned above, the order is wrong in lines 65 and 68, it should be swapped.

```
65 /**
66  * This class is used to represent a SQL statement.
67  */
68 public abstract class Statement extends Object implements Cloneable {
69
```

D. class (static) variables

a. first public class variables

No issues, as there are no public static variables.

b. next protected class variables

There is an issue starting from line number 70.

There is also an issue at line 121:

protected final static ResourceBundle messages = I18NHelper.loadBundle(

"com.sun.jdo.spi.persistence.support.sqlstore.Bundle", // NOI18N

Statement.class.getClassLoader());

This one is after instance variables and it is not in correct order.

c. next package level (no access modifier)

No issues, because there are no such variables.

d. last private class variables

There is an issue, as it could be seen below:

```
68 public abstract class Statement extends Object implements Cloneable {
69
70     private static final Integer ONE = new Integer(1);
71
72     protected static final int OP_PREFIX_MASK =     0x001; // 1
```

First, we should have the code from line 72, and after all protected variables, we should have a private declaration from code 70.

E. instance variables

a. first public instance variables

There is an issue (line 112).

public ArrayList tableList;

is public variable but after protected and private. It should be before both of them.

e. next protected instance variables

There are issues.

f. next package level (no access modifier)

There are issues.

g. last private instance variables
There are issues.

Considering the code starting from line number 96:

```
96      protected StringBuffer statementText;
97
98      private String quoteCharStart;
99
100     private String quoteCharEnd;
101
102     /** array of ColumnRef */
103     protected ArrayList columns;
104
105     Constraint constraint;
106
107     protected InputDesc inputDesc;
108
109     int action;
110
111     /** array of QueryTable */
112     public ArrayList tableList;
113
114     protected DBVendorType vendorType;
115
116     protected ArrayList secondaryTableStatements;
```

we can see that there are many issues. Order is completely wrong and not according to the rules stated above.

Public: 112 should be first
Protected: 96, 103, 107, 114 and 116
Package level: 109
Private: 98, 100

So, the order above is the correct one.

F. constructors

No issues.

The only constructor is after variable declarations, located at line 126.

G. methods

No issues.

Methods are located after the constructor.

26. Methods are grouped by functionality rather than by scope or accessibility.

No serious issues here.

Methods are grouped rather by functionality than by accessibility or scope, but also they are ordered by dependency (i.e. if method a() calls method b(), put them as closely together as possible).

There are functional subsets – functions that process or generate, getters, etc., so it means that they are ordered by functionality, but these auxiliary getters are near the functions using them.
Another choice would be to put them after other subsets, or all of them before these functions that deal with processing and generating.

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

No issues.

3.10 Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)

No issues.

29. Check that variables are declared in the proper scope

No issues.

30. Check that constructors are called when a new object is desired

No issues.

31. Check that all object references are initialized before use

No issues.

32. Variables are initialized where they are declared, unless dependent upon a computation

No issues.

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.

Considering the lines 739 and 766, we can conclude that they are not decalred at the beginning of the block.

Line 766 should be declared at 752. For 739, there are many different choices (declaring it as null at the beginning of function defintion, for example, and after that assigning the value).

3.11 Method Calls

34. Check that parameters are presented in the correct order

No issues, parameters are called in correct order.

35. Check that the correct method is being called, or should it be a different method with a similar name

No issues.

36. Check that method returned values are used properly

No issues.

3.12 Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)

No issues.

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds

No issues.

39. Check that constructors are called when a new array item is desired

No issues.

3.13 Object Comparison

40. Check that all objects (including Strings) are compared with "equals" and not with "=="

No issues inside the assigned methods, but there could be an issue in this class, in line number 447.

```
445         for (Iterator iter = tableList.iterator(); iter.hasNext() && table == null; ) {
446             QueryTable t = (QueryTable) iter.next();
447             if (t.getTableDesc().getTableElement() == tableElement) {
448             // if (t.getTableDesc().getTableElement().equals(tableElement)) {
449                 table = t;
450             }
451         }
```

TableElements are compared using == and standard operators cannot be overloaded in java, so this could be an issue.

3.14 Output Format

41. Check that displayed output is free of spelling and grammatical errors
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem
43. Check that the output is formatted correctly in terms of line stepping and spacing

No issues related to 41,42 and 43.

3.15 Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming: (see http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html)

The implementation avoids brutish programming and there are no issues.

45. Check order of computation/evaluation, operator precedence and parenthesizing
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding

No issues related to 45,46,47 and 48, because the methods don't do such computations where some of the cases could happen.

49. Check that the comparison and Boolean operators are correct

No issues.

50. Check throw-catch expressions, and check that the error condition is actually legitimate

No direct issues, but take a look at 52 and 53. Error conditions are legitimate,.

51. Check that the code is free of any implicit type conversions

No issues.

The conversions are explicit.

3.16 Exceptions

52. Check that the relevant exceptions are caught
53. Check that the appropriate action are taken for each catch block

Considering 52 and 53, it can be said that there is a potential issue, because it can be concluded that the code inside this class doesn't catch the relevant exceptions in most cases. This is a consequence based on Java exceptions concepts. From different point of view, this code may not be wrong, because the controversial unchecked exceptions are used here.

There are two ways handling the exceptions in Java.

The first solution would be to put the code that could cause exception inside the try block, and handle the exception inside catch block. This is done in lines 1217-1219. Excpetion is handled inside the method itself, so there is no need to worry about the excpetion handling outside the method.

```
try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        //
        // shouldn't happen.
        //
        return null;
    }
```

There is also another way, where types of exceptions are written after method declaration, using „throws" keyword. After that, all the possible exception types are listed for this method. These excpetions are handled outside of the method.

```
double quotientValue (int bro, int im) throws Exception
{
  if (im==0) throw new Exception ("Division by zero!");
```

```
    else return bro/(double)im;

}

.
```

But, in Java, it is also possible to throw unchecked exceptions without having to declare them. Unchecked exceptions extend RuntimeException. Throwables that extend Error are also unchecked, but should only be used for really serious issues (such as invalid bytecode). This is used in most cases in the assigned code, in lines numbered:

719

724

732

815

So, this is possibly an issue according to the checklist, because not all the exceptions are handled and actions are not taken inside the code that corresponds to this class and methods.

3.17 Flow of Control

54. In a switch statement, check that all cases are addressed by break or return

    No issues.

    Assigned methods don't have switch statements, and other switch statements, outside these methods are properly addressed by break.

55. Check that all switch statements have a default branch

    No issues.

    Assigned methods don't have switch statements. Other switch statements, outside these methods satisfy this condition, so there are no issues.

56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

    No issues.

    All loops are correctly formed and there are no loops directly related to assigned methods (but there are in this class, in general).

3.18 Files

57. Check that all files are properly declared and opened

58. Check that all files are closed properly, even in the case of an error

59. Check that EOF conditions are detected and handled correctly

60. Check that all file exceptions are caught and dealt with accordingly

There are no issues related to 57,58,59 and 60, because the assigned methods don't deal with files directly (and this class, in general).

4 Appendix

4.1 Software and tools used

- Microsoft Office Word 2013: To create and redact this document
- Oracle VM Virtual Box: For virtual machine
- Eclipse
- Github (reporsitory: https://github.com/penenadpi/Software-Engineering-2-Project/Deliveries)

4.2 Hours of works

The time spent for constructing this document:

- Nenad Petrovic: ~ 12 hours.