Politecnico di Milano

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Computer Science and Engineering

Software Engineering 2 project

Glassfish 4.1.1
Code Inspection Document

Version 1

Prof. Elisabetta Di Nitto

Andrea Autelitano	Matr. 849869
Marco De Cobelli	Matr. 858360
Matthew Rossi	Matr. 858880

Academic year 2015-2016

TABLE OF CONTENTS

1 Classes that were assigned to the group

- **ClusterHandler**

  appserver/admingui/common/src/main/java/org/glassfish/admingui/common/handlers/ClusterHandler.java

- **JdbcTempHandler**

  appserver/admingui/common/src/main/java/org/glassfish/admingui/common/handlers/JdbcTempHandler.java

2 Introduction

2.1 GlassFish Server Clusters

A *cluster* is a collection of GlassFish Server instances that work together as one logical entity. A cluster provides a runtime environment for one or more Java Platform, Enterprise Edition (Java EE) applications. A cluster provides high availability through failure protection, scalability, and load balancing.

https://docs.oracle.com/cd/E26576_01/doc.312/e24934/clusters.htm#GSHAG00005

2.2 GlassFish Server Instances

A GlassFish Server *instance* is a single Virtual Machine for the Java platform (Java Virtual Machine or JVM machine) on a single node in which GlassFish Server is running. A node defines the host where the GlassFish Server instance resides. The JVM machine must be compatible with the Java Platform, Enterprise Edition (Java EE).

https://docs.oracle.com/cd/E26576_01/doc.312/e24934/instances.htm#GSHAG00006

2.3 GlassFish Server Nodes

A *node* represents a host on which the GlassFish Server software is installed. A node must exist for every host on which GlassFish Server instances reside. A node's configuration contains information about the host such as the name of the host and the location where the GlassFish Server is installed on the host.

Each GlassFish Server node is one of the following types of node:

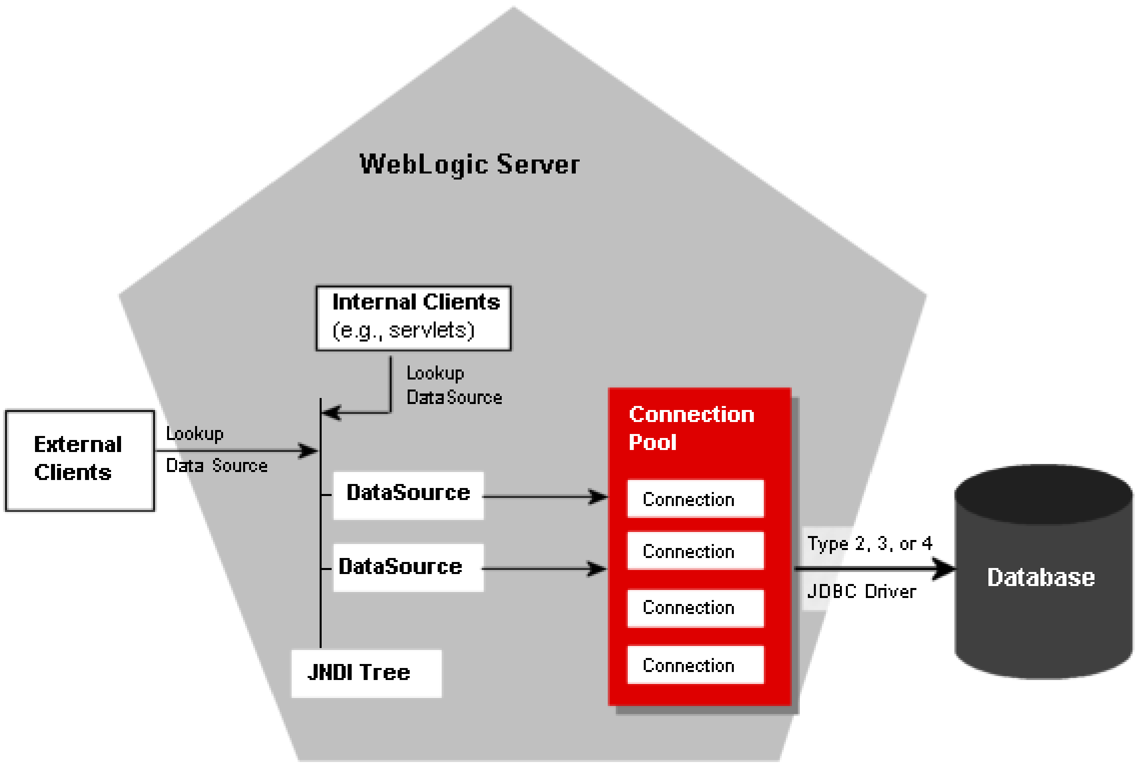
- A *DCOM* node supports communication over the Distributed Component Object Model (DCOM) remote protocol. The DCOM remote protocol is available only on Windows systems.
- An *SSH* node supports communication over secure shell (SSH).
- A *CONFIG* node does not support remote communication.

If neither DCOM nor SSH is set up and you plan to administer your instances locally, the instances can reside on CONFIG nodes. You cannot use CONFIG nodes for instances that you plan to administer centrally.

https://docs.oracle.com/cd/E26576_01/doc.312/e24934/nodes.htm#GSHAG00004

2.4 JDBC Connection Pools

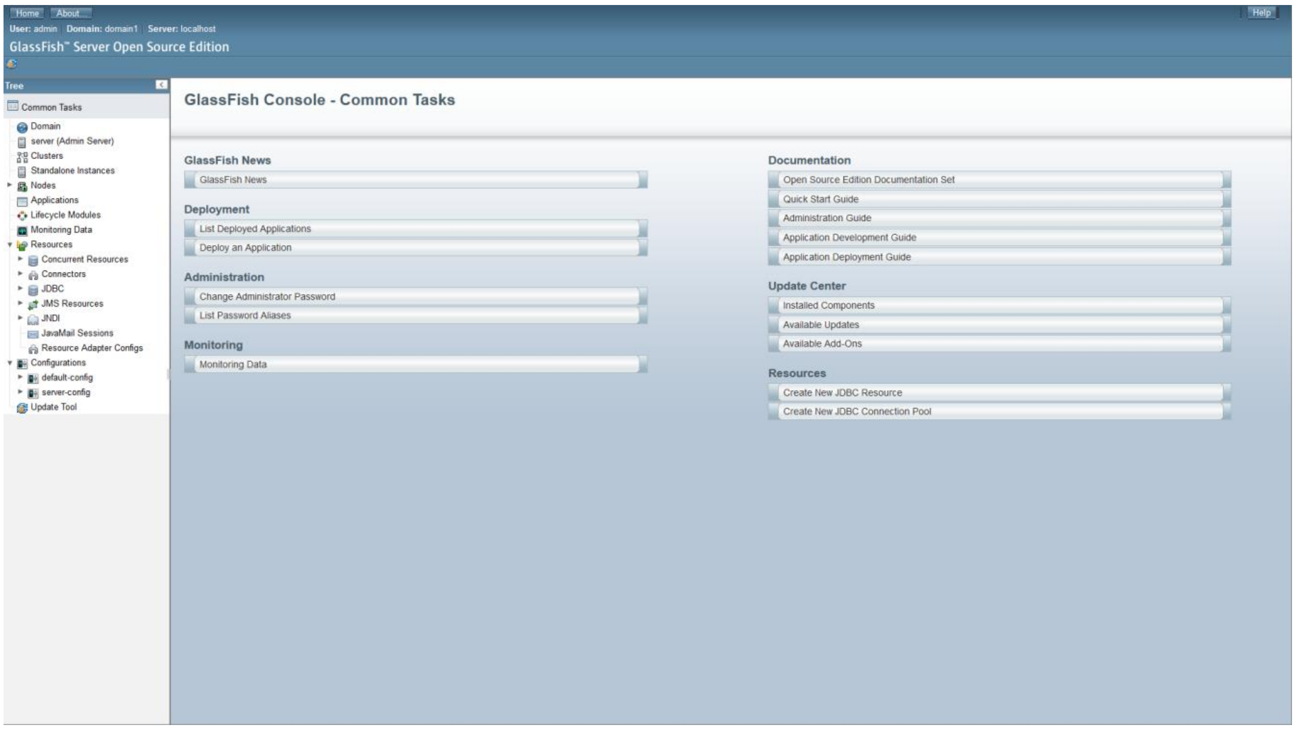
A *Connection Pool* operates by performing the work of creating connections to simplify and optimize the connection of a client application to the server side. In the case of a *JDBC Connection Pool*, a pool of *Connection* objects is created at the time the application server (or some other server) starts. These objects are then managed by a *pool manager* that disperses connections as they are requested by clients and returns them to the pool when it determines the client is finished with the Connection object.



https://docs.oracle.com/cd/E13222_01/wls/docs81/ConsoleHelp/jdbc_connection_pools.html

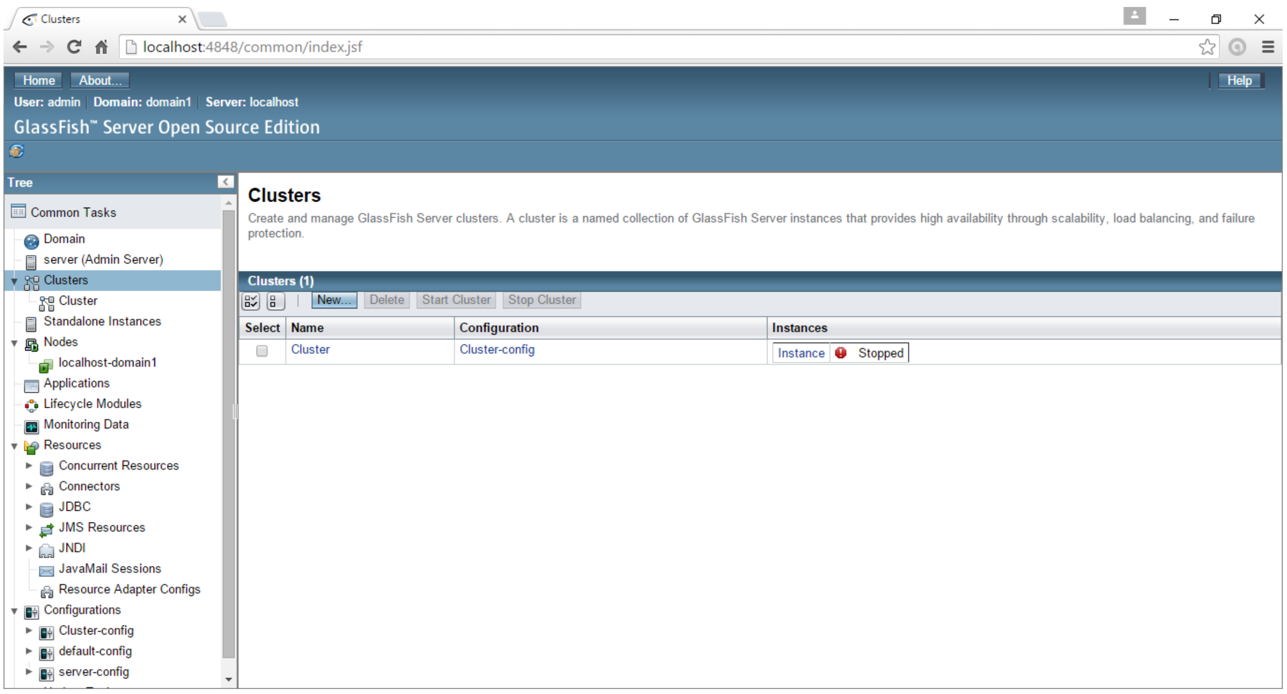
3 Functional role of assigned set of classes

The assigned set of classes is part of the *admingui* package and is used in the administration panel to manage, configure and monitor the access of an application to a database.



3.1 public class ClusterHandler

The class *ClusterHandler* gets requests by the admin console and handles them on clusters, instances and nodes that take part in a cluster.



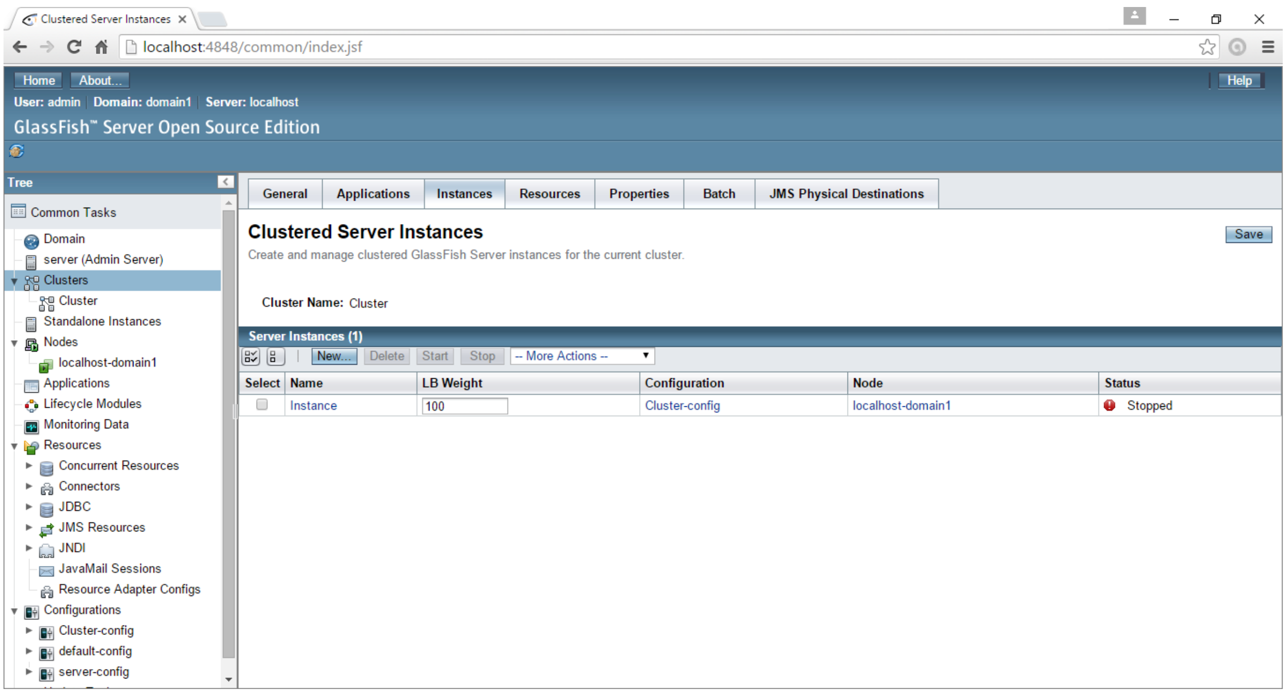
Clusters

Create and manage GlassFish Server clusters. A cluster is a named collection of GlassFish Server instances that provides high availability through scalability, load balancing, and failure protection.

Clusters (1)

Select	Name	Configuration	Instances
☐	Cluster	Cluster-config	Instance ● Stopped

Using the above interface, the application server administrator can create a new cluster and delete, start and stop an existing cluster by selecting it.



Clustered Server Instances

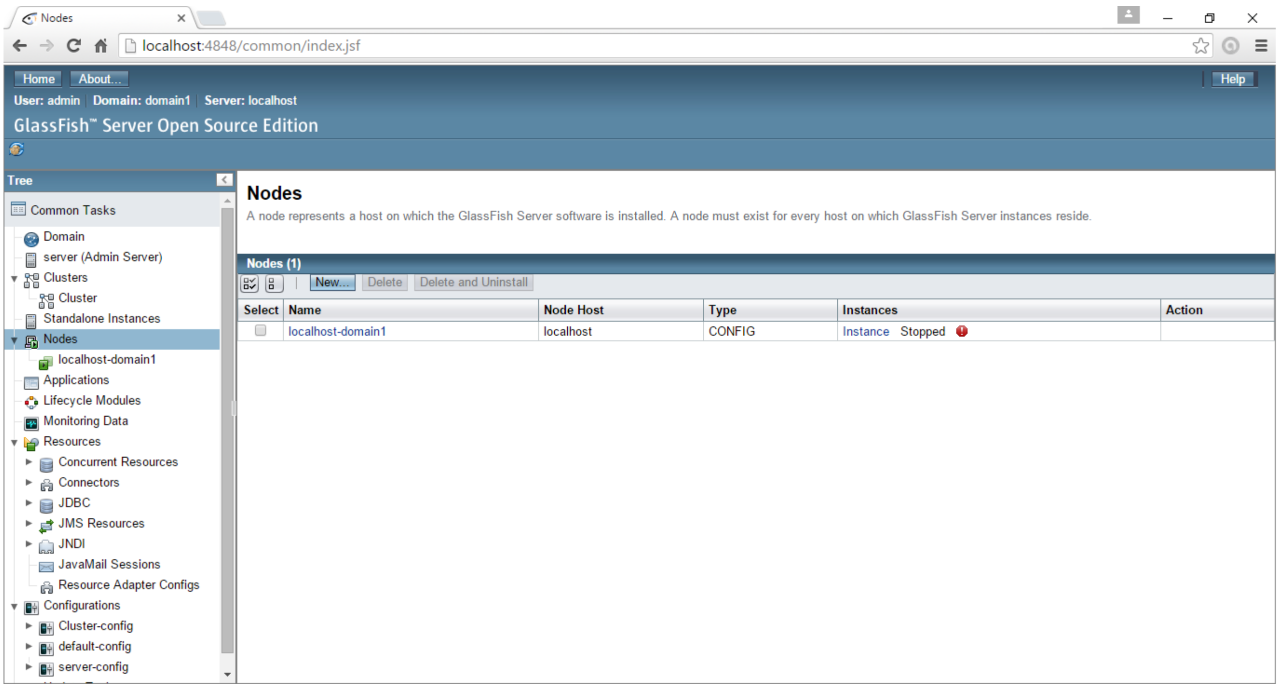
Create and manage clustered GlassFish Server instances for the current cluster.

Cluster Name: Cluster

Server Instances (1)

Select	Name	LB Weight	Configuration	Node	Status
☐	Instance	100	Cluster-config	localhost-domain1	● Stopped

Using the above interface, an admin can create a new cluster instance and delete, start and stop an existing cluster instance. It is also possible to change the load balancing weight for each instance.


Nodes

localhost:4848/common/index.jsf

Home About... Help

User: admin | Domain: domain1 | Server: localhost

GlassFish™ Server Open Source Edition

Tree

- Common Tasks
- Domain
- server (Admin Server)
- Clusters
  - Cluster
- Standalone Instances
- Nodes
  - localhost-domain1
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations
  - Cluster-config
  - default-config
  - server-config

Nodes

A node represents a host on which the GlassFish Server software is installed. A node must exist for every host on which GlassFish Server instances reside.

Nodes (1)

New... Delete Delete and Uninstall

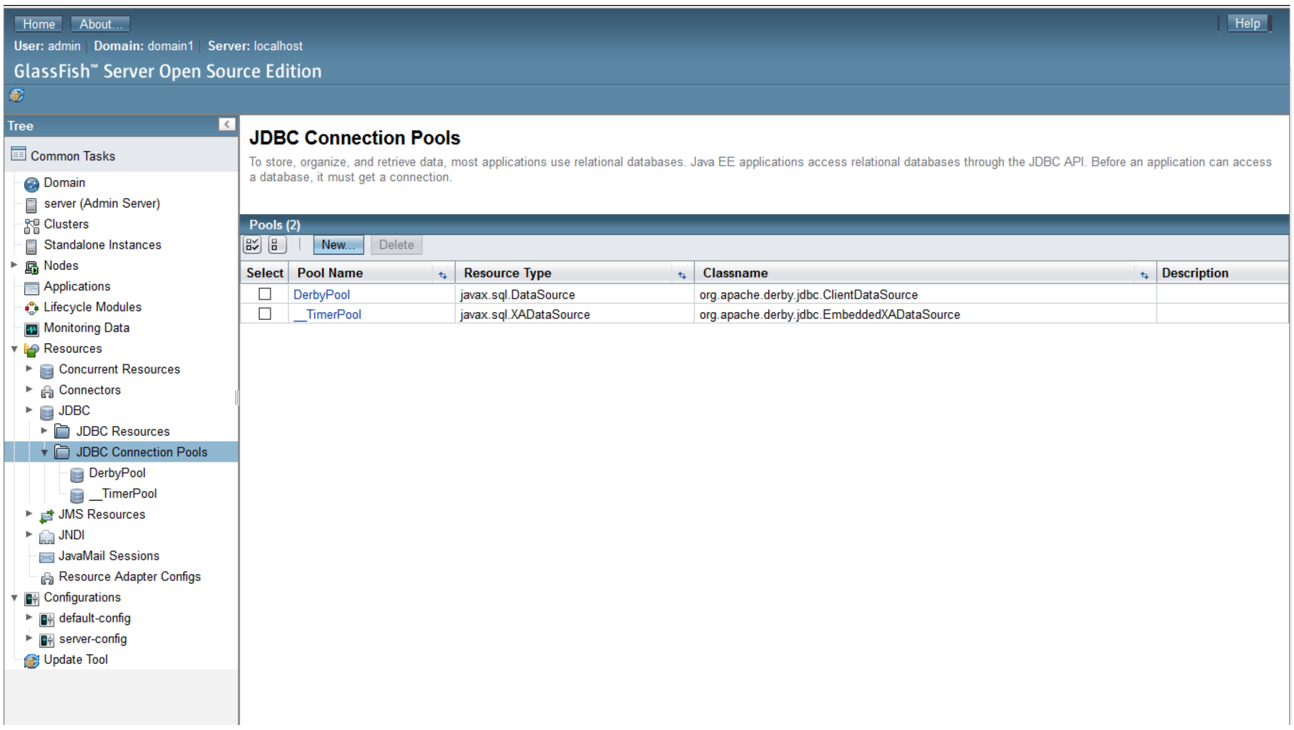
Select	Name	Node Host	Type	Instances	Action
☐	localhost-domain1	localhost	CONFIG	Instance Stopped ●	

Using the above interface, the application server administrator can delete nodes or delete and uninstall CONFIGs, SSHs or DCOMs nodes: in order to do that it has to do a REST request to the node. This cannot be done if the node is the one used by the administrator or the node has a GlassFish Server instance associated to it: in these cases the method terminates.

3.2 public class JdbcTempHandler

This class allows the creation of JDBC Connection Pools. All of the settings made with the Administration Console are static, meaning that they persist even after the Application Server is reset. Dynamic connection pools can be created using the command line or programmatically using the API.

To create a JDBC Connection Pool by means of the GlassFish console, click on the buttons "Resources→JDBC→JDBC Connection Pool→New".



JDBC Connection Pools

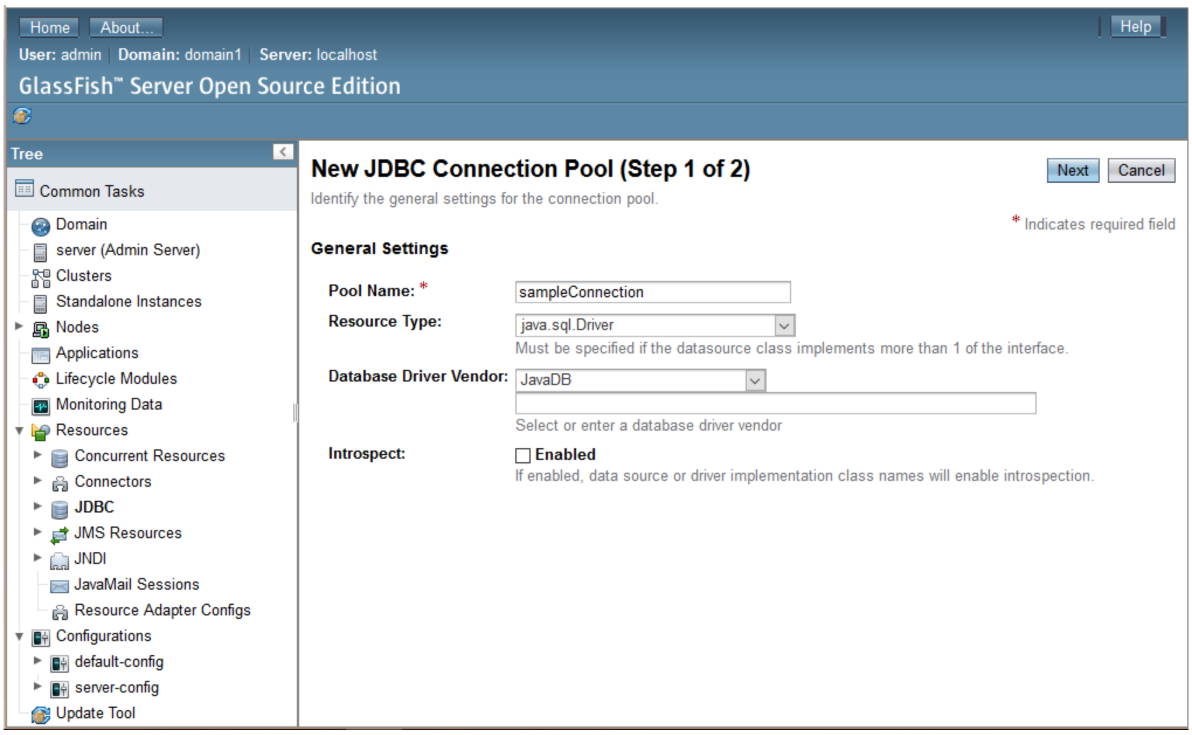
To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC API. Before an application can access a database, it must get a connection.

Select	Pool Name	Resource Type	Classname	Description
☐	DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource	
☐	__TimerPool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource	

The creation of a JDBC Connection Pool is divided in two steps.

Step 1 allows to choose the main properties of the connector:

- Pool Name: the name of the Connection Pool
- Resource Type: the type of the resource that will connect to the Database (DataSource, Driver…)
- Database Driver Vendor: the DBMS to which the application will connect
- Introspect: if enabled, allows introspection for the data source



New JDBC Connection Pool (Step 1 of 2)

Identify the general settings for the connection pool.

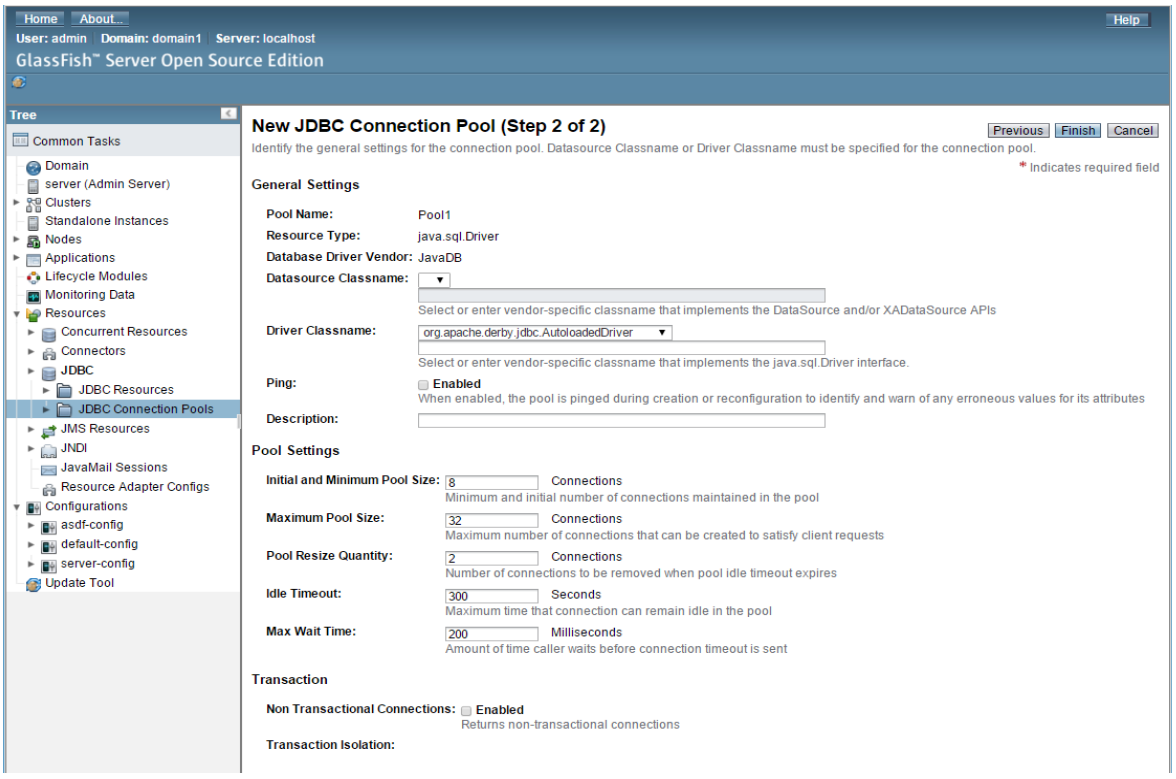
* Indicates required field

General Settings

Pool Name: *	sampleConnection
Resource Type:	java.sql.Driver Must be specified if the datasource class implements more than 1 of the interface.
Database Driver Vendor:	JavaDB Select or enter a database driver vendor
Introspect:	☐ Enabled If enabled, data source or driver implementation class names will enable introspection.

Step 2 allows to set additional properties. The main ones are:

- General settings: depending on the choice done at the Step 1, they can be relative to Datasource Classname or Driver Classname; in addition to that, it allows to test the Ping during the creation of the pool.
- Pool settings: options relative to the number of connections and limitations of time in the pool that is going to be created.
- Properties for the transactions: for instance the isolation level of the transactions.



New JDBC Connection Pool (Step 2 of 2)

Identify the general settings for the connection pool. Datasource Classname or Driver Classname must be specified for the connection pool.

* Indicates required field

General Settings

Pool Name:	Pool1
Resource Type:	java.sql.Driver
Database Driver Vendor:	JavaDB
Datasource Classname:	Select or enter vendor-specific classname that implements the DataSource and/or XADataSource APIs
Driver Classname:	org.apache.derby.jdbc.AutoloadedDriver Select or enter vendor-specific classname that implements the java.sql.Driver interface.
Ping:	Enabled When enabled, the pool is pinged during creation or reconfiguration to identify and warn of any erroneous values for its attributes
Description:	

Pool Settings

Initial and Minimum Pool Size:	8 Connections Minimum and initial number of connections maintained in the pool
Maximum Pool Size:	32 Connections Maximum number of connections that can be created to satisfy client requests
Pool Resize Quantity:	2 Connections Number of connections to be removed when pool idle timeout expires
Idle Timeout:	300 Seconds Maximum time that connection can remain idle in the pool
Max Wait Time:	200 Milliseconds Amount of time caller waits before connection timeout is sent

Transaction

Non Transactional Connections:	Enabled Returns non-transactional connections
Transaction Isolation:	

The methods related to Step 2 analyzed more in detail are:

- public static void updateJdbcConnectionPoolPropertiesTable(HandlerContext handlerCtx)

  It updates the properties table in the Step 2 of the wizard. It is the logger of the properties of the JDBC connection pool, where all requests are stored.
- public static void updateJdbcConnectionPoolWizardStep2(HandlerContext handlerCtx)

  It updates the transaction isolation level if it is deleted by the user and updates the Datasource Classname/Driver Classname depending on the type of the used resource.
- public static void addClassNameColumn(HandlerContext handlerCtx)

  It adds the Classname column to the pool depending on the type of the used resource (Driver/Datasource).

4 List of issues found by applying the checklist

4.1 public class ClusterHandler

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.

Method variables “rows” and “oneRow” does not have a meaningful name. For example in the “nodeAction” method those names should be “nodes” and “oneNode”.

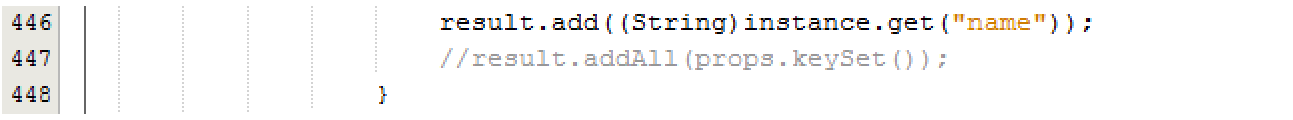
Comments

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

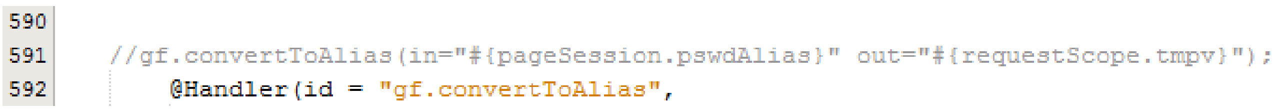
Comments cover only the 6% of the whole class code, and so cannot adequately explain what the code does.

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.

Lines 447 and 591 are commented out and do not contain neither an explanation nor a date.



```
446                         result.add((String)instance.get("name"));
447                         //result.addAll(props.keySet());
448                     }
```



```
590
591     //gf.convertToAlias(in="#{pageSession.pswdAlias}" out="#{requestScope.tmpv}");
592         @Handler(id = "gf.convertToAlias",
```

Java source files

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

There is almost no Javadoc for the class ClusterHandler and the only method that has a Javadoc associated to it (getClusterStatusSummary) in addition to what is said in the Javadoc also calculates the number of instances that need a restart and have an unknown state.

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

ClusterHandler’s Javadoc is not complete at all since no parameters and almost all the methods have no Javadoc associated to them.

Class and interface declarations

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

A. class/interface documentation comment
B. class or interface statement
C. class/interface implementation comment, if necessary
D. class (static) variables

a. first public class variables
b. next protected class variables
c. next package level (no access modifier)
d. last private class variables

E. instance variables

a. first public instance variables
b. next protected instance variables
c. next package level (no access modifier)
d. last private instance variables

F. constructors

G. methods

A. A comment about the class implementation should be there in order to explain what the class does.

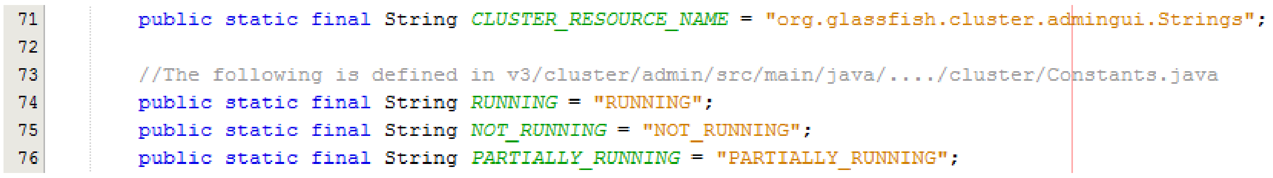
26. *Methods are grouped by functionality rather than by scope or accessibility.*

getClusterNameForInstance and getClusterForConfig should be grouped since both of them find a cluster name the former starting by the instance name and the latter by its configuration.

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).

All class members are public, but they should be at least protected (Lines 71, 74, 75 and 76).



```
71        public static final String CLUSTER_RESOURCE_NAME = "org.glassfish.cluster.admingui.Strings";
72
73        //The following is defined in v3/cluster/admin/src/main/java/..../cluster/Constants.java
74        public static final String RUNNING = "RUNNING";
75        public static final String NOT_RUNNING = "NOT_RUNNING";
76        public static final String PARTIALLY_RUNNING = "PARTIALLY_RUNNING";
```

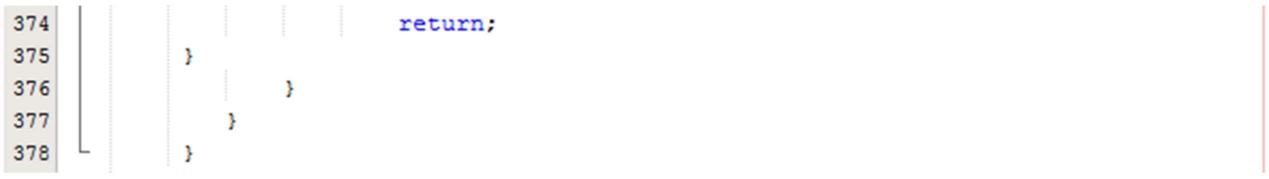
4.1.1 for(Map oneRow : rows)

Lines 319-377

Indentation

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

Line 375 has five indenting spaces while in the whole piece of code four are used.

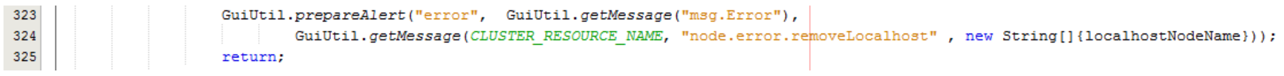


```
374                         return;
375          }
376                   }
377              }
378          }
```

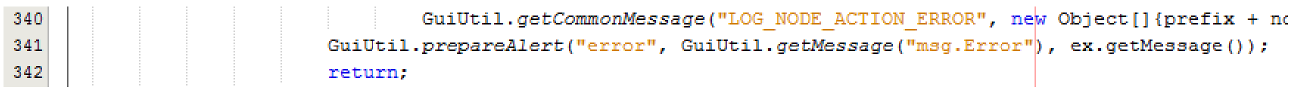
File organization

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

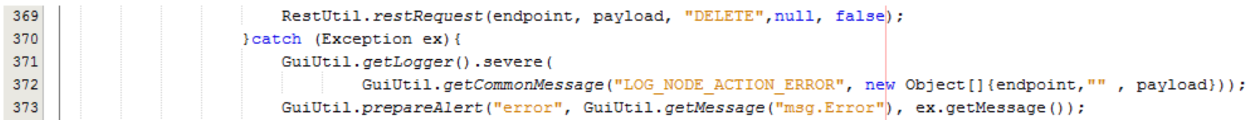
Lines 324, 341, 369, 372 and 373 exceed 80 characters and could be written in 80 characters using correct line breaks.



```
323                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
324                         GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "node.error.removeLocalhost" , new String[]{localhostNodeName}));
325                 return;
```



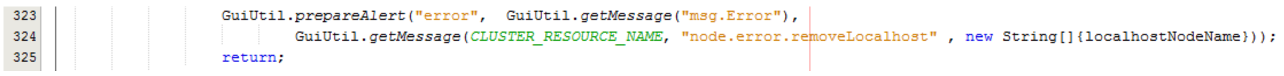
```
340                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{prefix + nc
341                     GuiUtil.prepareAlert("error", GuiUtil.getMessage("msg.Error"), ex.getMessage());
342                     return;
```



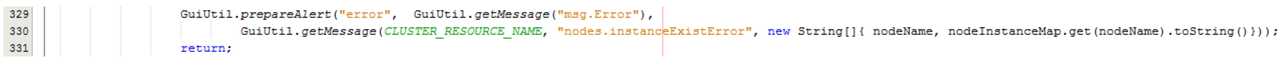
```
369                 RestUtil.restRequest(endpoint, payload, "DELETE",null, false);
370             }catch (Exception ex){
371                 GuiUtil.getLogger().severe(
372                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{endpoint,"" , payload}));
373                 GuiUtil.prepareAlert("error", GuiUtil.getMessage("msg.Error"), ex.getMessage());
```

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

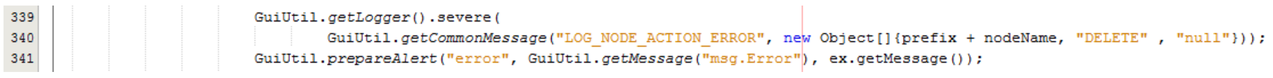
Lines 324, 330 and 340 exceed 120 characters.



```
323                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
324                         GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "node.error.removeLocalhost" , new String[]{localhostNodeName}));
325                 return;
```



```
329             GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
330                     GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "nodes.instanceExistError", new String[]{ nodeName, nodeInstanceMap.get(nodeName).toString()}));
331             return;
```

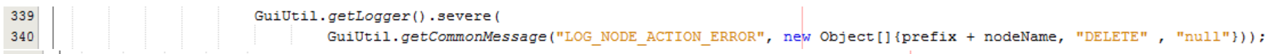


```
339                 GuiUtil.getLogger().severe(
340                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{prefix + nodeName, "DELETE" , "null"}));
341                 GuiUtil.prepareAlert("error", GuiUtil.getMessage("msg.Error"), ex.getMessage());
```

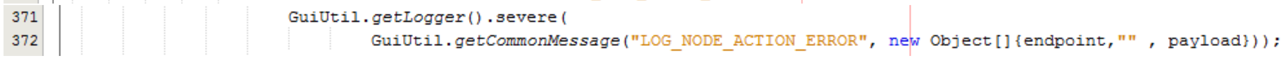
Wrapping lines

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

Lines 339 and 371 have line brakes after the opening of a round bracket.

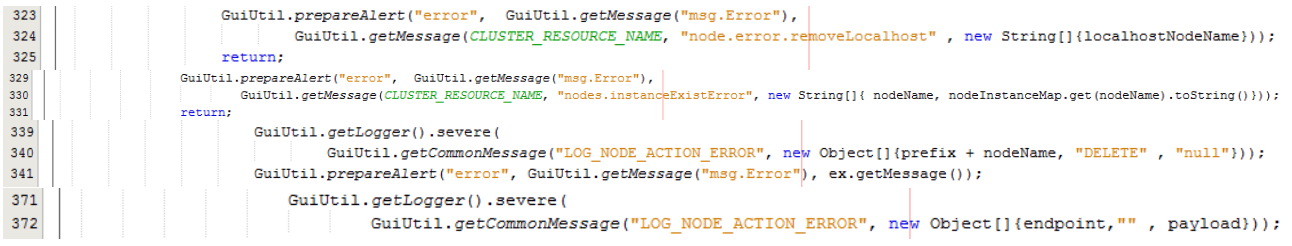


```
339                 GuiUtil.getLogger().severe(
340                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{prefix + nodeName, "DELETE" , "null"}));
```



```
371                 GuiUtil.getLogger().severe(
372                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{endpoint,"" , payload}));
```

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.
Lines 324, 330, 340 and 372 do not respect this constraint, but they are correct (as reported in the Oracle coding conventions for Java:
http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html).

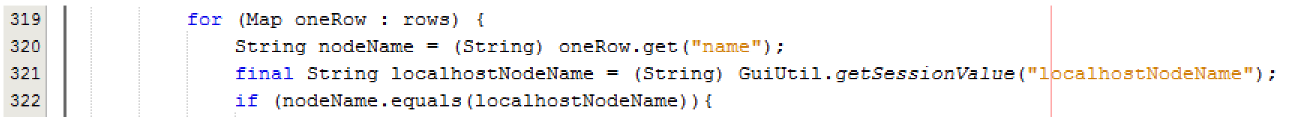


```
323                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
324                         GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "node.error.removeLocalhost" , new String[]{localhostNodeName}));
325                 return;
329             GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
330                     GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "nodes.instanceExistError", new String[]{ nodeName, nodeInstanceMap.get(nodeName).toString()}));
331             return;
339                 GuiUtil.getLogger().severe(
340                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{prefix + nodeName, "DELETE" , "null"}));
341                 GuiUtil.prepareAlert("error", GuiUtil.getMessage("msg.Error"), ex.getMessage());
371                 GuiUtil.getLogger().severe(
372                         GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{endpoint,"" , payload}));
```

Initialization and declarations

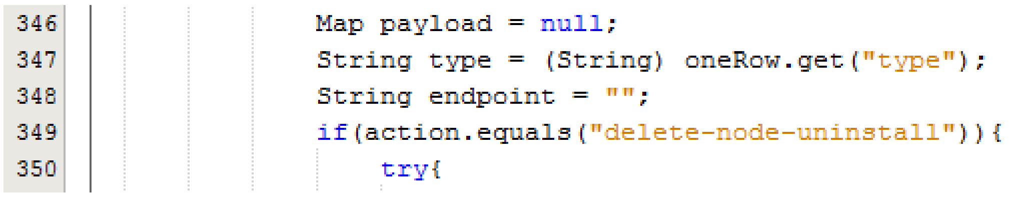
29. Check that variables are declared in the proper scope.

"localhostNodeName" at line 321 should be defined outside the for cycle (starting at line 319), since it is used for every iteration but its value does not change.



```
319         for (Map oneRow : rows) {
320             String nodeName = (String) oneRow.get("name");
321             final String localhostNodeName = (String) GuiUtil.getSessionValue("localhostNodeName");
322             if (nodeName.equals(localhostNodeName)){
```

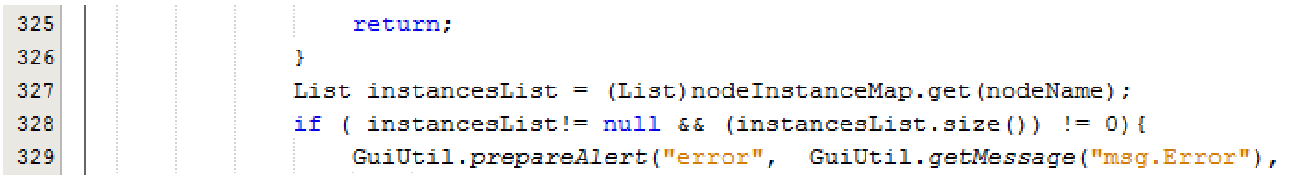
"payload", "type" and "endpoint" respectively at lines 346, 347 and 348 should be defined in the if that follows starting from line 349.



```
346                 Map payload = null;
347                 String type = (String) oneRow.get("type");
348                 String endpoint = "";
349                 if(action.equals("delete-node-uninstall")){
350                     try{
```

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.

"instancesList" at line 327 should be declared at the beginning of the block.

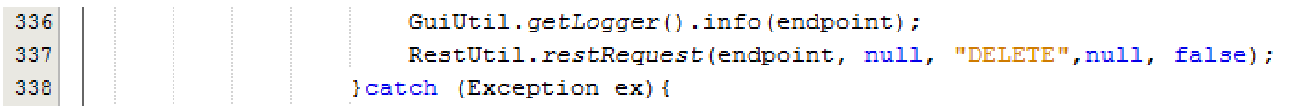


```
325                     return;
326                 }
327                 List instancesList = (List)nodeInstanceMap.get(nodeName);
328                 if ( instancesList!= null && (instancesList.size()) != 0){
329                     GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
```

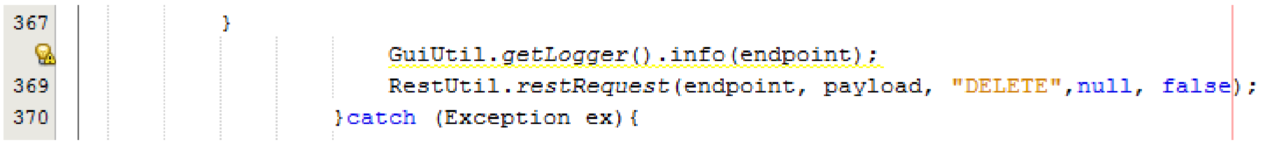
Method calls

36. Check that method returned values are used properly

The values returned by RestUtil.restRequest(…) are not checked at lines 337 and 369, but this does not introduce any bug since if any problem with the REST request occurs the method throws an exception, and so no checking on the returned value is needed.



```
336                         GuiUtil.getLogger().info(endpoint);
337                         RestUtil.restRequest(endpoint, null, "DELETE",null, false);
338                     }catch (Exception ex){
```



```
367                }
                        GuiUtil.getLogger().info(endpoint);
369                     RestUtil.restRequest(endpoint, payload, "DELETE",null, false);
370                 }catch (Exception ex){
```

Output format

41. Check that displayed output is free of spelling and grammatical errors.

The output of the "nodeAction" method execution (alert or writings on the log file) are due to exception situations, and are the following:

1. **The node to be removed is the localhost**
   Msg.Error = An error has occurred
   Node.error.removeLocalhost = Cannot remove Node <localhostNodeName>.
2. **The node to be removed is in the instanceList**
   Msg.Error = An error has occurred
   Nodes.instanceExistError = Node "<nodeName>" referenced in GlassFish Server instaces: <instancesList.toString()>. Remove the instaces before deleting the node.
3. **A problem has occurred in the REST request**
   LOG_NODE_ACTION_ERROR = Error in nodeAction; endpoint = "<prefix+nodeName>/DELETE.json attrsMap=null
   Msg.Error = An error has occurred
   ex.getMessage()
4. **A problem has occurred in the REST request**
   LOG_NODE_ACTION_ERROR = Error in nodeAction; endpoint = "<endpoint>/.json attrsMap=< payload>
   Msg.Error = An error has occurred
   ex.getMessage()

When the third and fourth errors occur, in the log file quotation marks are not closed.

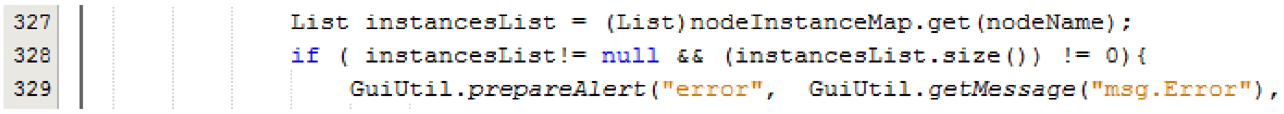
42. *Check that error messages are comprehensive and provide guidance as to how to correct the problem.*

In the last two errors, what it is written in the log could be more comprehensible by writing in both cases *endpoint = "<endpoint>/DELETE.json* ...

Computation, Comparisons and Assignments

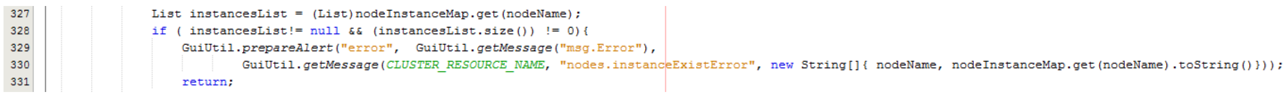
44. Check that the implementation avoids "brutish programming: (see http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html).

At line 328 "instancesList.size() != 0" should be "!instancesList.isEmpty()".



```
327             List instancesList = (List)nodeInstanceMap.get(nodeName);
328             if ( instancesList!= null && (instancesList.size()) != 0){
329                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
```

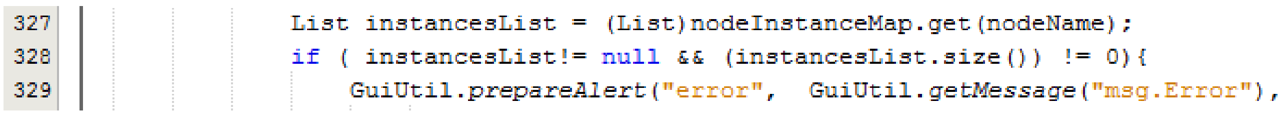
Line 330 uses "nodeInstanceMap.get(nodeName).toString()", but "nodeInstanceMap.get(nodeName)" is already stored into "instancesList". So it should use "instancesList.toString()".



```
327             List instancesList = (List)nodeInstanceMap.get(nodeName);
328             if ( instancesList!= null && (instancesList.size()) != 0){
329                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
330                         GuiUtil.getMessage(CLUSTER_RESOURCE_NAME, "nodes.instanceExistError", new String[]{ nodeName, nodeInstanceMap.get(nodeName).toString()}));
331                 return;
```

46. Check the liberal use of parenthesis is used to avoid operator precedence problems.

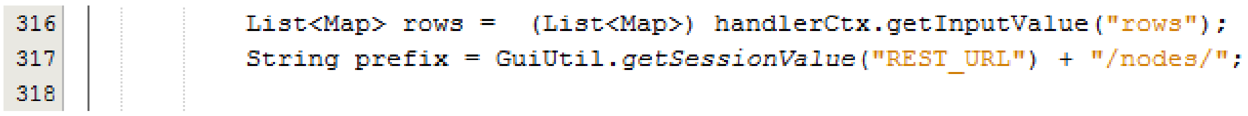
At line 328 “instancesList.size()” is within brackets, but there is no need of them.



```
327             List instancesList = (List)nodeInstanceMap.get(nodeName);
328             if ( instancesList!= null && (instancesList.size()) != 0){
329                 GuiUtil.prepareAlert("error",  GuiUtil.getMessage("msg.Error"),
```

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

Line 317 “GuiUtil.getSessionValue(…)” returns an Object, but we deal with it like it is a String implicitly.



```
316         List<Map> rows =  (List<Map>) handlerCtx.getInputValue("rows");
317         String prefix = GuiUtil.getSessionValue("REST_URL") + "/nodes/";
318
```

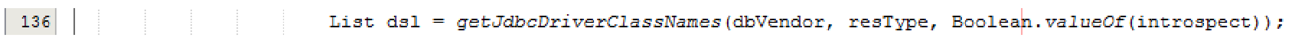
4.2 public class JdbcTempHandler

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.

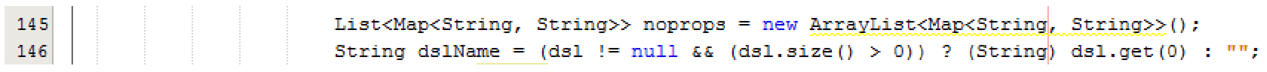
Some variables do not have meaningful names:

- **dsl**



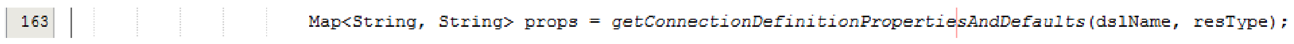
```
List dsl = getJdbcDriverClassNames(dbVendor, resType, Boolean.valueOf(introspect));
```

- **noprops** and **dslName**



```
List<Map<String, String>> noprops = new ArrayList<Map<String, String>>();
String dslName = (dsl != null && (dsl.size() > 0)) ? (String) dsl.get(0) : "";
```

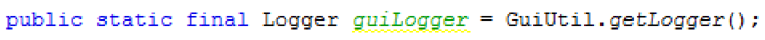
- **props**



```
Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(dslName, resType);
```

7. Constants are declared using all uppercase with words separated by an underscore.

At line 350 a constant is declared not using all upper case and not separating words with the underscore.



```
public static final Logger guiLogger = GuiUtil.getLogger();
```

File Organization

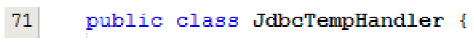
12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

Blank lines are correctly used; however, no comments are used to separate one section from another in any part of the code. In addition to that, comments for the description of what the class does and related to the attribute declaration are missing. Moreover, there aren't comments that point out what the role of the class in the package is and what is imported.

Comments

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

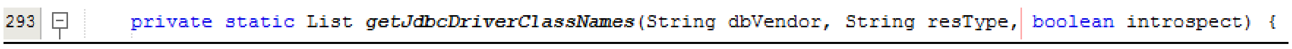
Comments are the main problem for the selected class: in fact, only 14% of the source file is composed of comments. However, most of the comments do not explain adequately what the section does: they are either redundant (e.g. they just repeat the name of the method) or not present. This is particularly evident , for example, at line 71:



```
71   public class JdbcTempHandler {
```

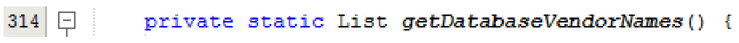
where there is no explanation for what the class does, or for the following private methods at lines:

- 293:



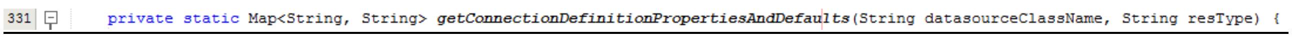
```
293   private static List getJdbcDriverClassNames(String dbVendor, String resType, boolean introspect) {
```

- 314:



```
314   private static List getDatabaseVendorNames() {
```

- 331:

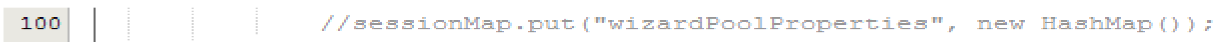


```
331   private static Map<String, String> getConnectionDefinitionPropertiesAndDefaults(String datasourceClassName, String resType) {
```

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.

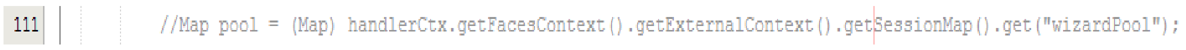
Pieces of commented out code are present, but no explanation is given at lines:

- 100:



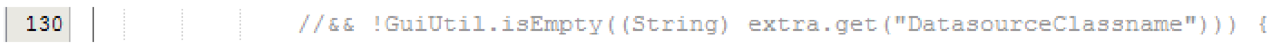
```
100    //sessionMap.put("wizardPoolProperties", new HashMap());
```

- 111:



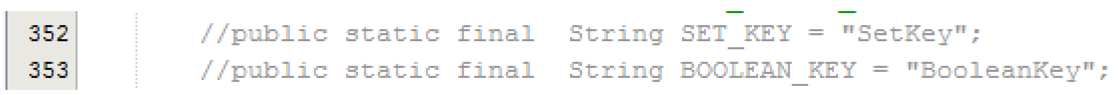
```
111    //Map pool = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPool");
```

- 130:



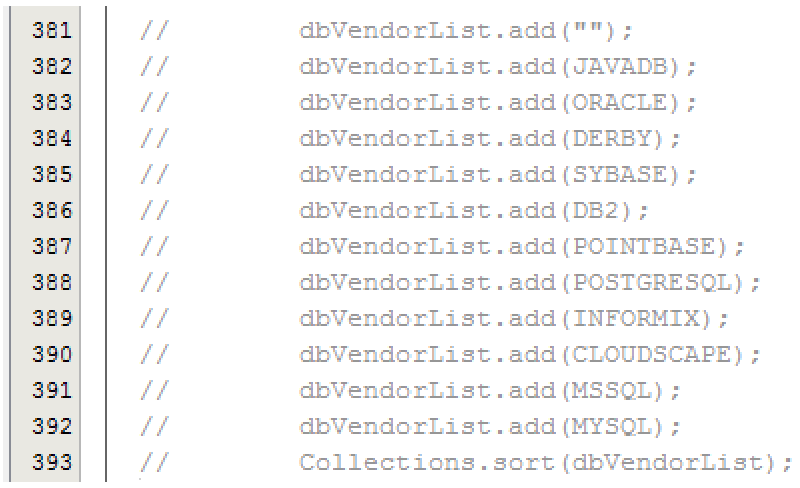
```
130    //&& !GuiUtil.isEmpty((String) extra.get("DatasourceClassname"))) {
```

- 352-353:



```
352    //public static final  String SET_KEY = "SetKey";
353    //public static final  String BOOLEAN_KEY = "BooleanKey";
```

- 381-393:



```
381  //        dbVendorList.add("");
382  //        dbVendorList.add(JAVADB);
383  //        dbVendorList.add(ORACLE);
384  //        dbVendorList.add(DERBY);
385  //        dbVendorList.add(SYBASE);
386  //        dbVendorList.add(DB2);
387  //        dbVendorList.add(POINTBASE);
388  //        dbVendorList.add(POSTGRESQL);
389  //        dbVendorList.add(INFORMIX);
390  //        dbVendorList.add(CLOUDSCAPE);
391  //        dbVendorList.add(MSSQL);
392  //        dbVendorList.add(MYSQL);
393  //        Collections.sort(dbVendorList);
```

Java Source Files

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

The Javadoc is not thorough, but the class described is implemented in a way that’s consistent with it.
In addition to that, the reason why the Javadoc for methods updateJdbcConnectionPoolWizardStep2 and updateJdbcConnectionPoolPropertiesTable is the same is not clear, since they perform two different actions.

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

There is no Javadoc for the following methods:

- private static List getJdbcDriverClassNames(String dbVendor, String resType, boolean introspect)
- private static List getDatabaseVendorNames()
- private static Map<String, String> getConnectionDefinitionPropertiesAndDefaults(String datasourceClassName, String resType)

Class and Interface Declarations

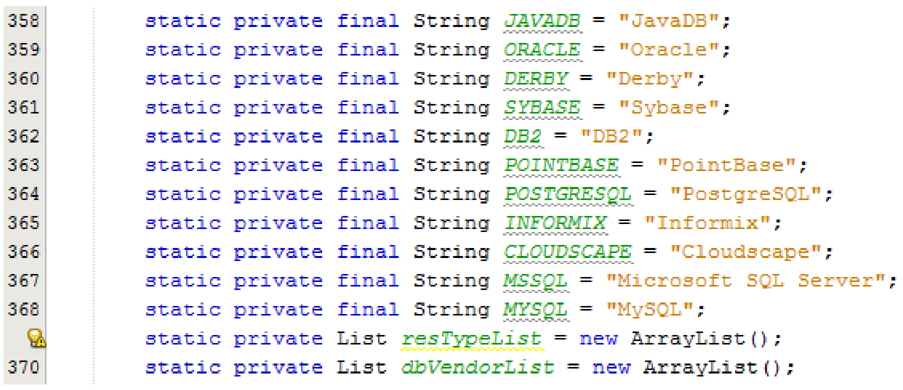
25. The class or interface declarations shall be in the following order:

- *A. class/interface documentation comment*
- *B. class or interface statement*
- *C. class/interface implementation comment, if necessary*
- *D. class (static) variables*
  - *a. first public class variables*
  - *b. next protected class variables*
  - *c. next package level (no access modifier)*
  - *d. last private class variables*
- *E. instance variables*
  - *a. first public instance variables*
  - *b. next protected instance variables*
  - *c. next package level (no access modifier)*
  - *d. last private instance variables*
- *F. constructors*
- *G. methods*

A. The class documentation comment is missing.
C. The class implementation comment is missing.
D. The static variables are not declared after the class implementation comment but at the end of the methods of the class.

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

The variables declared from line 358 to line 368 are useless because are used only by lines of code of the class that are currently commented.



```
358        static private final String JAVADB = "JavaDB";
359        static private final String ORACLE = "Oracle";
360        static private final String DERBY = "Derby";
361        static private final String SYBASE = "Sybase";
362        static private final String DB2 = "DB2";
363        static private final String POINTBASE = "PointBase";
364        static private final String POSTGRESQL = "PostgreSQL";
365        static private final String INFORMIX = "Informix";
366        static private final String CLOUDSCAPE = "Cloudscape";
367        static private final String MSSQL = "Microsoft SQL Server";
368        static private final String MYSQL = "MySQL";
           static private List resTypeList = new ArrayList();
370        static private List dbVendorList = new ArrayList();
```

4.2.1 updateJDBCPoolWizardStep1(HandlerContext handlerCtx)

Lines 110-198

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).

The preferred "Allman" style as bracing style is not used; however, it is okay because the "Kernighan and Ritchie" style is used consistently.

File Organization

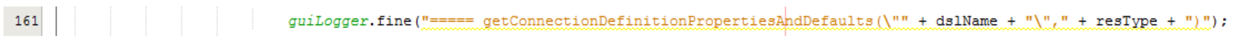
13. Where practical, line length does not exceed 80 characters.

Few lines exceed the limit of 80 characters, but it is necessary in order to guarantee the readability of the code.

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

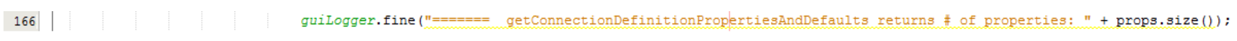
There are many examples of lines that exceed both the limits of 80 and 120 characters (in brackets, the number of characters) at lines:

- 161 (131):



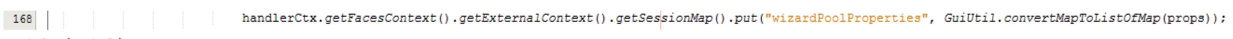
```
161                         guiLogger.fine("===== getConnectionDefinitionPropertiesAndDefaults(\"" + dslName + "\"," + resType + ")");
```

- 166 (142):



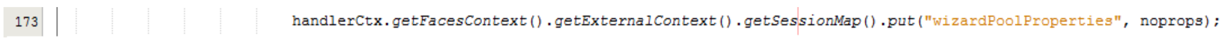
```
166                         guiLogger.fine("=======  getConnectionDefinitionPropertiesAndDefaults returns # of properties: " + props.size());
```

- 168 (157):



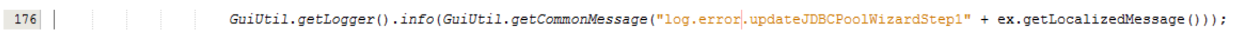
```
168                     handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", GuiUtil.convertMapToListOfMap(props));
```

- 168 (128):



```
173                     handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", noprops);
```

- 176 (138):

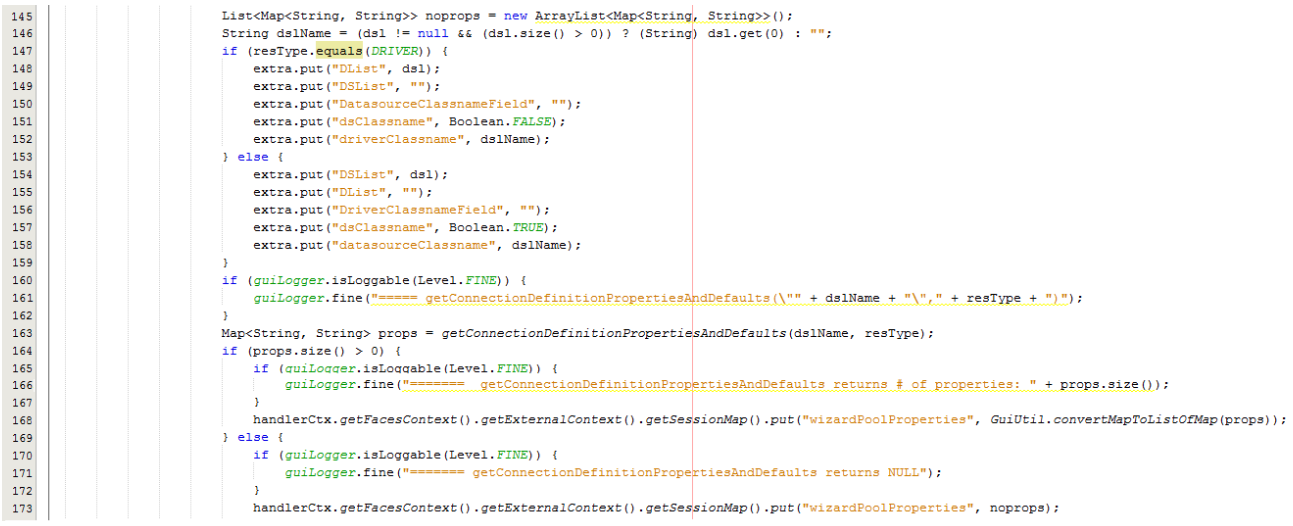


```
176                 GuiUtil.getLogger().info(GuiUtil.getCommonMessage("log.error.updateJDBCPoolWizardStep1" + ex.getLocalizedMessage()));
```

Initialization and Declarations

29. Check that variables are declared in the proper scope

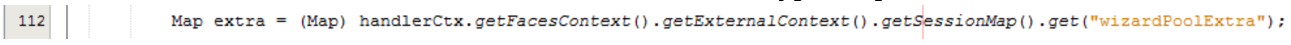
Variable **noprops** is declared at line 145; however, it is not used until line 173 in an "else" statement.



```
145  List<Map<String, String>> noprops = new ArrayList<Map<String, String>>();
146  String dslName = (dsl != null && (dsl.size() > 0)) ? (String) dsl.get(0) : "";
147  if (resType.equals(DRIVER)) {
148      extra.put("DList", dsl);
149      extra.put("DSList", "");
150      extra.put("DatasourceClassnameField", "");
151      extra.put("dsClassname", Boolean.FALSE);
152      extra.put("driverClassname", dslName);
153  } else {
154      extra.put("DSList", dsl);
155      extra.put("DList", "");
156      extra.put("DriverClassnameField", "");
157      extra.put("dsClassname", Boolean.TRUE);
158      extra.put("datasourceClassname", dslName);
159  }
160  if (guiLogger.isLoggable(Level.FINE)) {
161      guiLogger.fine("===== getConnectionDefinitionPropertiesAndDefaults(\"" + dslName + "\"," + resType + ")");
162  }
163  Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(dslName, resType);
164  if (props.size() > 0) {
165      if (guiLogger.isLoggable(Level.FINE)) {
166          guiLogger.fine("=======  getConnectionDefinitionPropertiesAndDefaults returns # of properties: " + props.size());
167      }
168      handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", GuiUtil.convertMapToListOfMap(props));
169  } else {
170      if (guiLogger.isLoggable(Level.FINE)) {
171          guiLogger.fine("======= getConnectionDefinitionPropertiesAndDefaults returns NULL");
172      }
173      handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", noprops);
```

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

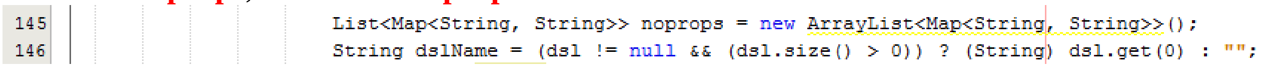
At line 112 the constructor is not used for variable **extra** of type Map



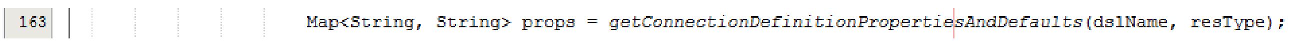
```
112  Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
```

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.

Variables **noprops**, **dslName** and **props** are declared in the middle of a block.



```
145  List<Map<String, String>> noprops = new ArrayList<Map<String, String>>();
146  String dslName = (dsl != null && (dsl.size() > 0)) ? (String) dsl.get(0) : "";
```



```
163  Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(dslName, resType);
```

4.2.2 updateJdbcConnectionPoolPropertiesTable(HandlerContext handlerCtx)

Lines 203-224

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).

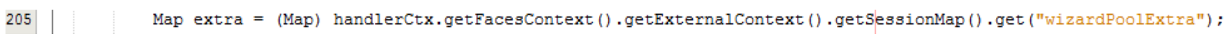
The preferred "Allman" style as bracing style is not used; however, it is okay because the "Kernighan and Ritchie" style is used consistently.

File Organization

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

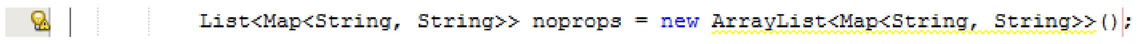
The following lines have a length greater than 80 characters and is practical (according to the principles explained at the chapter 4.2 Wrapping lines of the Oracle documentation http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html) to insert a line break:

- 205:



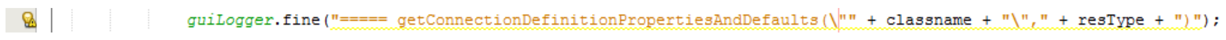
```
205      Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
```

- 208:



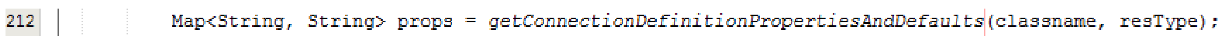
```
             List<Map<String, String>> noprops = new ArrayList<Map<String, String>>();
```

- 210:



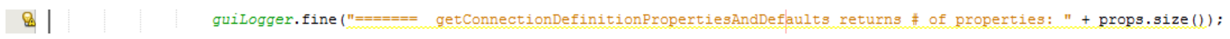
```
             guiLogger.fine("===== getConnectionDefinitionPropertiesAndDefaults(\"" + classname + "\"," + resType + ")");
```

- 212:



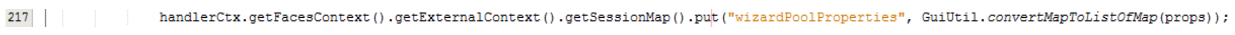
```
212          Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(classname, resType);
```

- 215:



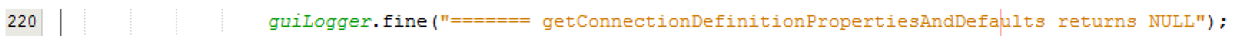
```
                 guiLogger.fine("=======  getConnectionDefinitionPropertiesAndDefaults returns # of properties: " + props.size());
```

- 217:



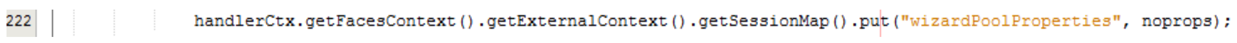
```
217          handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", GuiUtil.convertMapToListOfMap(props));
```

- 220:



```
220                  guiLogger.fine("======= getConnectionDefinitionPropertiesAndDefaults returns NULL");
```

- 222:

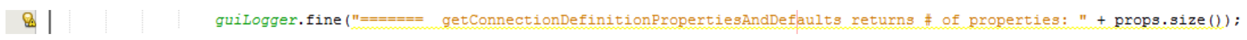


```
222              handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", noprops);
```

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

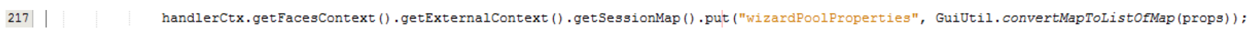
The following lines have a length greater than 120 characters:

- 215:



```
                 guiLogger.fine("=======  getConnectionDefinitionPropertiesAndDefaults returns # of properties: " + props.size());
```

- 217:

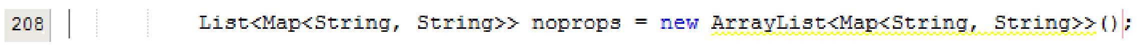


```
217          handlerCtx.getFacesContext().getExternalContext().getSessionMap().put("wizardPoolProperties", GuiUtil.convertMapToListOfMap(props));
```

Initialization and Declarations

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

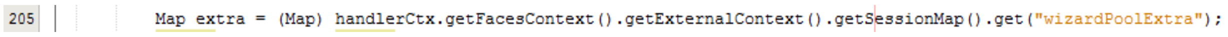
At the line 208 an ArrayList is declared with redundant type arguments in the new expression which can be canceled.



```
208          List<Map<String, String>> noprops = new ArrayList<Map<String, String>>();
```

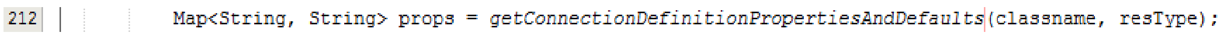
At the following lines, the variables are declared without using the constructor:

- 205: constructor not used for variable **extra** of type Map



```
205            Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
```

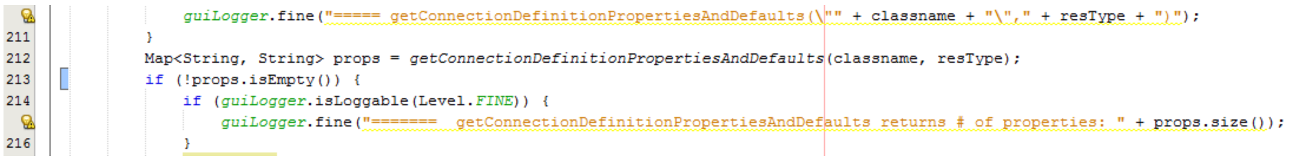
- 212: constructor not used for variable **props** of type Map<String, String>



```
212              Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(classname, resType);
```

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.

The variable at the line 212 should be defined at the beginning of the method.



```
                guiLogger.fine("===== getConnectionDefinitionPropertiesAndDefaults(\"" + classname + "\"," + resType + ")");
211         }
212         Map<String, String> props = getConnectionDefinitionPropertiesAndDefaults(classname, resType);
213         if (!props.isEmpty()) {
214             if (guiLogger.isLoggable(Level.FINE)) {
                    guiLogger.fine("=======  getConnectionDefinitionPropertiesAndDefaults returns # of properties: " + props.size());
216             }
```

4.2.3 updateJdbcConnectionPoolWizardStep2(HandlerContext handlerCtx)

Lines 229-256

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).

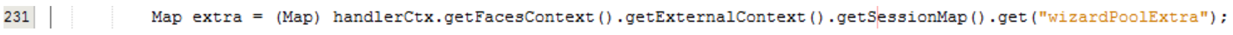
The preferred "Allman" style as bracing style is not used; however, it is okay because the "Kernighan and Ritchie" style is used consistently.

File Organization

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

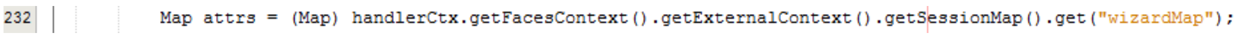
The following lines have a length greater than 80 characters and is practical (according to the principles explained at the chapter 4.2 Wrapping lines of the Oracle documentation http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html) to insert a line break:

- 231:



```
231         Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
```

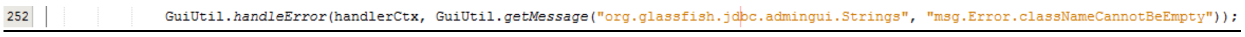
- 232:



```
232         Map attrs = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardMap");
```

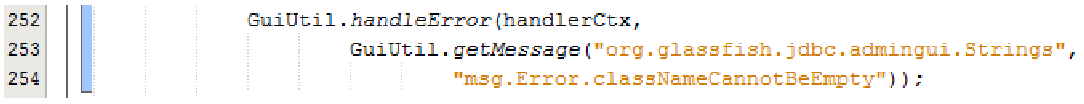
For instance, in the following line is practical to insert line breaks:

- 252:



```
252             GuiUtil.handleError(handlerCtx, GuiUtil.getMessage("org.glassfish.jdbc.admingui.Strings", "msg.Error.classNameCannotBeEmpty"));
```

While it should be

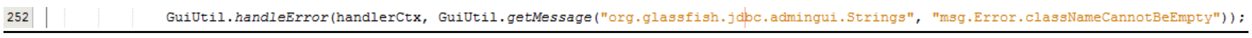


```
252             GuiUtil.handleError(handlerCtx,
253                     GuiUtil.getMessage("org.glassfish.jdbc.admingui.Strings",
254                             "msg.Error.classNameCannotBeEmpty"));
```

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

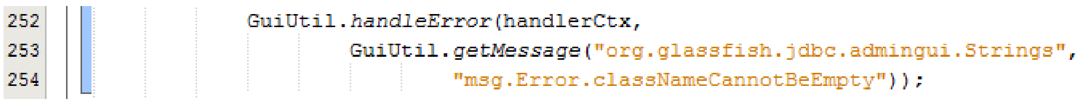
The following line has a length greater than 120 characters:

- 252:



```
252             GuiUtil.handleError(handlerCtx, GuiUtil.getMessage("org.glassfish.jdbc.admingui.Strings", "msg.Error.classNameCannotBeEmpty"));
```

While it should be

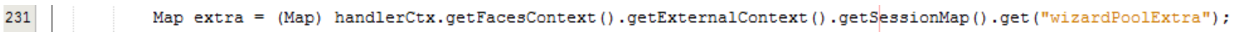


```
252             GuiUtil.handleError(handlerCtx,
253                     GuiUtil.getMessage("org.glassfish.jdbc.admingui.Strings",
254                             "msg.Error.classNameCannotBeEmpty"));
```

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

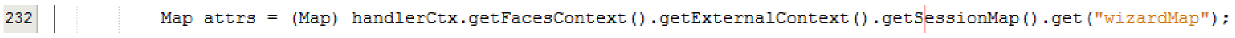
At the following lines, the variables are declared without using the constructor:

- 231: constructor not used for variable **extra** of type Map



```
231         Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
```

- 232: constructor not used for variable **attrs** of type Map

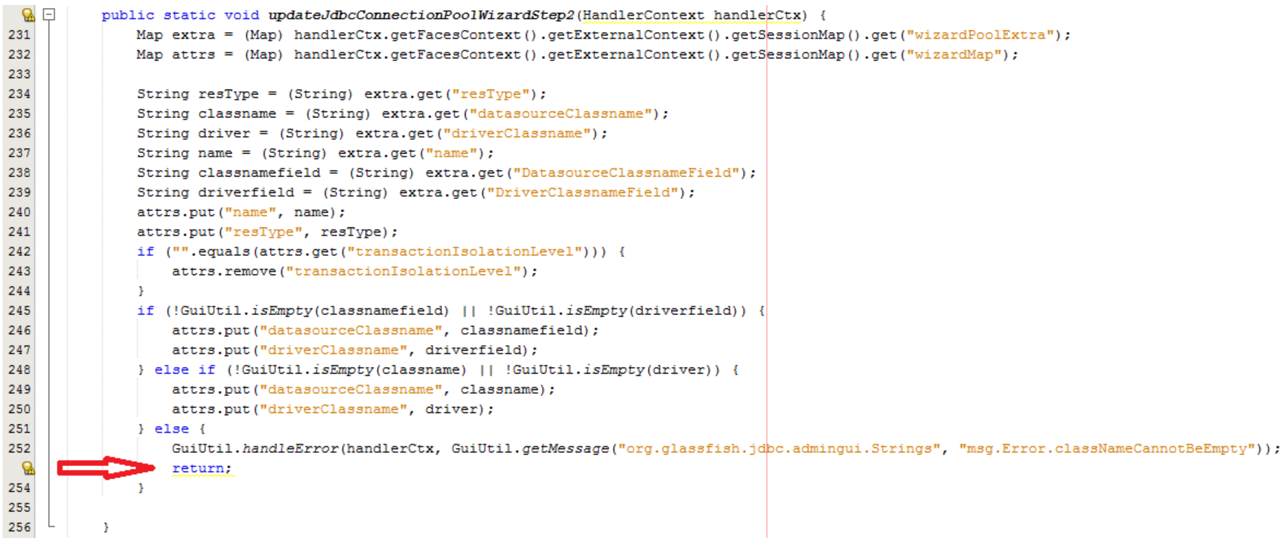


```
232         Map attrs = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardMap");
```

Method Calls

36. Check that method returned values are used properly

There’s a return statement not necessary at line 253 because after it the method ends and does not have to return values.

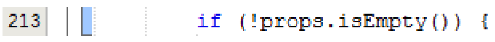


```
        public static void updateJdbcConnectionPoolWizardStep2(HandlerContext handlerCtx) {
231         Map extra = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardPoolExtra");
232         Map attrs = (Map) handlerCtx.getFacesContext().getExternalContext().getSessionMap().get("wizardMap");
233
234         String resType = (String) extra.get("resType");
235         String classname = (String) extra.get("datasourceClassname");
236         String driver = (String) extra.get("driverClassname");
237         String name = (String) extra.get("name");
238         String classnamefield = (String) extra.get("DatasourceClassnameField");
239         String driverfield = (String) extra.get("DriverClassnameField");
240         attrs.put("name", name);
241         attrs.put("resType", resType);
242         if ("".equals(attrs.get("transactionIsolationLevel"))) {
243             attrs.remove("transactionIsolationLevel");
244         }
245         if (!GuiUtil.isEmpty(classnamefield) || !GuiUtil.isEmpty(driverfield)) {
246             attrs.put("datasourceClassname", classnamefield);
247             attrs.put("driverClassname", driverfield);
248         } else if (!GuiUtil.isEmpty(classname) || !GuiUtil.isEmpty(driver)) {
249             attrs.put("datasourceClassname", classname);
250             attrs.put("driverClassname", driver);
251         } else {
252             GuiUtil.handleError(handlerCtx, GuiUtil.getMessage("org.glassfish.jdbc.admingui.Strings", "msg.Error.classNameCannotBeEmpty"));
                return;
254         }
255
256     }
```

Computation, Comparisons and Assignments

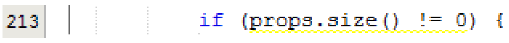
49. Check that the comparison and Boolean operators are correct

At the line 213, it would be better to use



```
213         if (!props.isEmpty()) {
```

instead of



```
213         if (props.size() != 0) {
```

4.2.4 if(poolsData != null)

<u>Lines 261-291</u>

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).

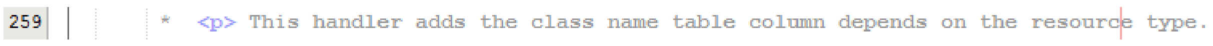
The preferred "Allman" style as bracing style is not used; however, it is okay because the "Kernighan and Ritchie" style is used consistently.

File Organization

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

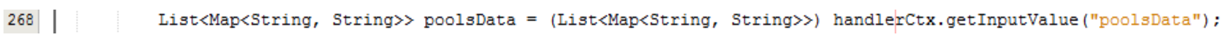
The following lines have a length greater than 80 characters and is practical (according to the principles explained at the chapter 4.2 Wrapping lines of the Oracle documentation http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html) to insert a line break:

- 259:



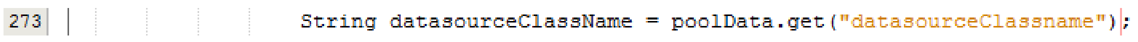
```
259      *  <p> This handler adds the class name table column depends on the resource type.
```

- 268:



```
268         List<Map<String, String>> poolsData = (List<Map<String, String>>) handlerCtx.getInputValue("poolsData");
```

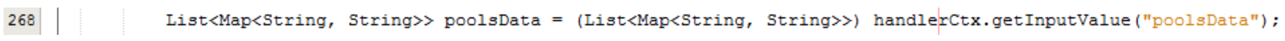
- 273:



```
273                     String datasourceClassName = poolData.get("datasourceClassname");
```

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

At line 268 the constructor is not used for variable **poolsData** of type List<Map<String, String>>



```
268         List<Map<String, String>> poolsData = (List<Map<String, String>>) handlerCtx.getInputValue("poolsData");
```

5 Any other problem you have highlighted

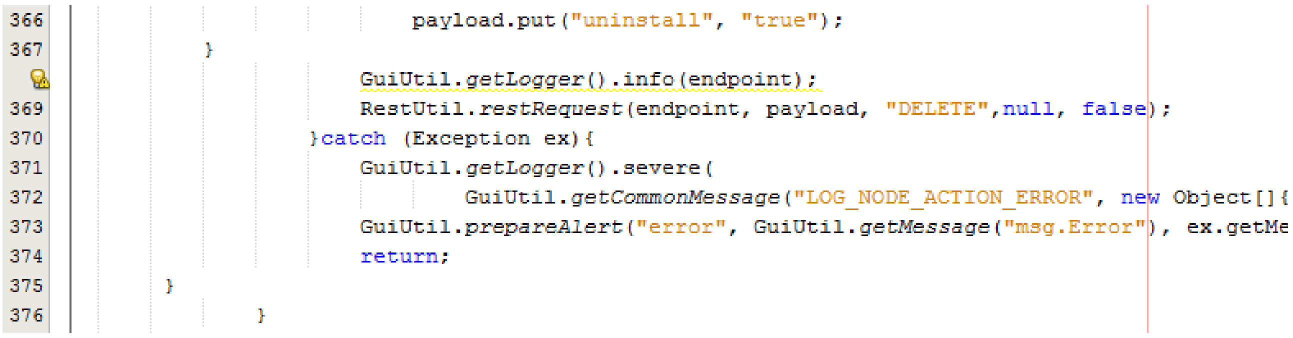
5.1 public class ClusterHandler

5.1.1 for(Map oneRow : rows)

Lines 319-377

Indentation

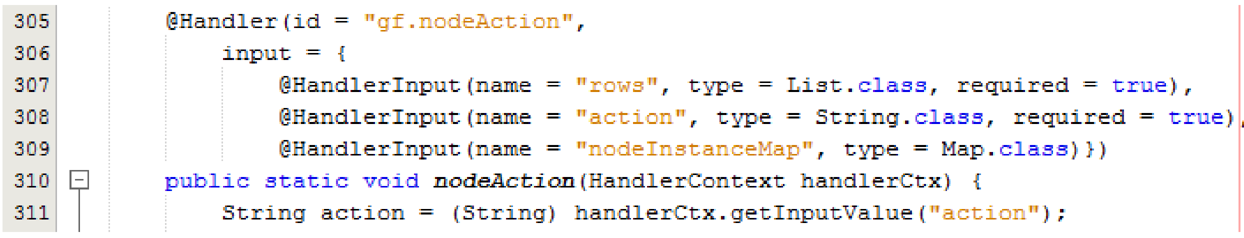
Lines 367 and 375 have a wrong indentation.



```
366                           payload.put("uninstall", "true");
367           }
                          GuiUtil.getLogger().info(endpoint);
369                       RestUtil.restRequest(endpoint, payload, "DELETE",null, false);
370                   }catch (Exception ex){
371                       GuiUtil.getLogger().severe(
372                               GuiUtil.getCommonMessage("LOG_NODE_ACTION_ERROR", new Object[]{
373                       GuiUtil.prepareAlert("error", GuiUtil.getMessage("msg.Error"), ex.getMe
374                       return;
375        }
376               }
```

Bugs

“nodeInstanceMap” is an optional parameter, so if “nodeAction” is called without the “nodeInstanceMap” parameter, the method could delete a node that has still an instance associated to it.



```
@Handler(id = "gf.nodeAction",
    input = {
        @HandlerInput(name = "rows", type = List.class, required = true),
        @HandlerInput(name = "action", type = String.class, required = true),
        @HandlerInput(name = "nodeInstanceMap", type = Map.class)})
public static void nodeAction(HandlerContext handlerCtx) {
    String action = (String) handlerCtx.getInputValue("action");
```

6 APPENDIX

6.1 Used tools

- Microsoft Word 2013 to write this document
- Code Analyzer 0.7.0 to check the percentage of comments in the source code

6.2 Hours of work

Each member of the group has worked on this document for 24 hours.