stSoftware development guidelines

stSoftware
sustainable technology

Table of Contents

1 CHECK LIST

You are finished a task only after ALL of the check list below have been completed and not before

1.1 Requirements

Do I understand the task and what is trying to be achieved/resolved ?

(Each task is an issue that needs to be resolved. Any instructions/steps on how to resolve this are a guide only to get you down the correct path. You must understand issue so that you can make the necessary modifications.)

1.2 Problem Solving

- Am I using the ide debugger. (I can't stress this enough. If you do not know how to use it, then ask.)
- Am I using System.out.println to debug (These are forbidden. LOGGER.debugg it should be used and the logging properties mode should be set to debug so that they can be seen in log files.)
- NullPointerExceptions are the easiest exception to fix. Simply look at the code and determine the variable that is being used. i.e person.name() - person would be null to cause the exception.
- Have I tried to reproduce the problem the simplest way possible. (Excluding all the noise will make it easier to find)
- Have I created a self test to reproduce (Do this where possible. This will force you to find the simplest way to reproduce the error and will also make if easier to fix.)
- The next step after reproducing the error is to find the point at which the failure occurs. To do this find the first step that works and keep moving through the test until you get to the line of code that is the issue.

1.3 Development

- How can my code be broken? Can there be null pointers? Have I made assumptions about the state of code, data and variables that may not necessarily be true.
- Have I run checkstyle and pmd checks and make adjustements as needed? (Keep running until both pass.)
- Have I used DBField constants ?
- Have I added appropriate comments in your code ?
- Would another developer be able to follow what I have done ?
- Have I created DBMessage objects for error messages used in validationList.createError method ?
- Is IDE showing possible error hints. (i.e. red/yellow squiggly underlined varibales, highlighted text)

1.4 DBClass/DBFields

- Have you added a suitable description. A description based on the filed name is not a good description. for e.g paymentAmount filed should not have description of 'Payment Account'.
- Should the DBClass be a static list, abstract?
- Have I made a new key on a class and could it be a string instead of a numeric?
- Have I set numeric key fields as autoincrement?
- For linked fields, should the setting be 'delete or clear'
- Are field names starting with the class name, i.e for the Contact class, to add a name field, the field should be called `name`, not `contactName`

1.5 Screen Modifications

- Is the alignment and spacing of fields considered with rest of screen fields.
- Has a colon (:) been added after label in data entry screen and no colon for portal and report column headers
- Is the spelling correct?
- Are field labels/section headers starting with the class name, i.e for the Contact class, to add a name field, the field should be called name, not Contact Name
- Alignment. Particularly in portals and reports, numeric fields should be aligned right and the label should be set to placement = 'east'

1.6 Testing

- Has it been tested well ?
- Did it solved the issue that the task was trying to solve?
- Can my supervisor break my code/find issues in my changes ?
- Have I either created an automated self test our a manual test so that my supervisor can validate that my changes are correct ?

2 Java Coding STYLE

2.1 Overview

The purpose of this section is to **briefly** cover general coding standards for Java programs developed by the ST Software Development Team.

2.2 Comments

Before general code layout is discussed, comment requirements must be understood. This section covers comment requirements, which are put into perspective in the subsequent section covering general layout.

Comments are an extremely important aspect of code maintainability. There is often a notion that commenting code takes “too much time” but nothing can be further from the truth. Most of a code’s lifecycle is spent on updates and maintenance and is often performed by someone other than the original author. Furthermore, all developers spend some amount of time reviewing the code, replaying the logic in his/her head to make sure it is doing the proper thing. That time can be used to either update or add the comments. In addition to making the code comprehensible for other coders, comments serve as a sanity check for the original author. Therefore, comments are a must! The subsequent sections describe the comment requirements for all Java files.

Note: All tab characters must be replaced with spaces. This ensures OS compatibility. **Each tab character must be replaced by at least 4 spaces. Any reference to tab in terms of indentation in the following document implies 4 spaces.**

2.2.1 File Header

This consists of simply the class name and the copyright notice in a standard Java comment block. This is the first block of comments that appear within the Java file. This comment block should be completely left-justified with no leading tabs. An example follows:

```
/*
 *  Copyright (c) 2000-2017 ASP Converters pty ltd
 *
 *  www.aspconverters.com.au
 *
 *  All Rights Reserved.
 *
 *  This software is the proprietary information of
 *  ASP Converters Pty Ltd.
 *  Use is subject to license terms.
 */
```

The above should be the very first text in a Java file and can contain a standard copyright notice.

2.2.2 The Class Comment Block

This is the main comment block that is picked up by JavaDoc. It is very important that it exist and should immediately precede the class declaration. It should contain a detailed description of the class purpose and, if applicable, usage examples. In addition, it should explicitly reference classes that are related in any way by using the @see javadoc tag. It should contain author and version tags and the thread mode. It should also be left-justified with no leading tabs. An example follows:

```
/**
 * The <code>Foo</code>, performs all foo-like functions. Usage of Foo
 * is often just to illustrate some technical example,
 * having no real meaning beyond the example.
 * Geeks, in particular, really love Foo.
 *
 * Usage of <code>Foo</code> often involves using <code>Bar</code>
 *
 * <b>Note :</b> <code>Foo</code> is in no way connected to the rock band
 * Foo Fighters.
 *
 * <i>THREAD MODE: SINGLE THREADED command </i> Foo is only used by one thread
 *
 * @author      Joe Public
 * @version     $Revision: 1.10 $
 * @since       24 Mar 2005
 *
 * @see com.foobar.Bar
 */
```

2.2.3 Method Comments

All public and protected methods must be commented (even private ones should be). The comment block should clearly identify the purpose of the method, the arguments to the method (if any), the return value (if appropriate) as well as any exceptions that may be thrown. Standard javadoc tags facilitate the commenting process. A getter and setter example follows:

```
/**
 * Sets the id for this instance of <code>Foo</code>.
 *
 * @param int _id the new value for the ID of
 * this <code>Foo</code> object.
 *
 * @exception IllegalArgumentException if the supplied value is invalid.
 */
public void setFooFunction( final int id )

/**
 * Gets the id for this instance of <code>Foo</code>.
 *
 * @return the ID for this <code>Foo</code> object or <code>null</code>
 * if ID has not yet been set.
 */
public int getFooFunction()
```

2.2.4 Public and Protected Variable Comments

Since public variables also show up in the JavaDoc, they must be commented. Proper commenting consists of a comment block immediately preceding the variable. The comment block should be 4 spaces from the left.

```
/**
 * This list contains asset identifiers associated with the security<br>
 * One security can have multiple identifiers
 */
public List assetIdentifiers;

/** This table contains the association between users and entitlements*/
private HashMap<String, String> userMap;
```

2.2.5 In-line Comments

In line comments consist of single-line, “double-slash” comments or block comments. All code should contain some amount of in-line comments to describe what is happening every few line of code. It is also particularly important to comment anything that is not very obvious from the code or that breaks with a “standard” pattern.

All in-line comments should start at the current indentation level. Consider the following example:

```
// Get the status and class of the user for verification purposes
String userStatus = user.getString(User.DBFIELD_STATUS);
String userType = user.getString(User.DBFIELD_TYPE);

// First see if the user is inactive and do appropriate processing
if (userStatus.equals(User.STATUS_INACTIVE))
{
    this.handleInactive(user);
}
// Now deal with active users
else
{
    // For preferred active users, go to the free upgrade processing
    if (userType.equals( User.TYPE_PREFERRED))
    {
        this.processFreeUpgrade(user, service);
    }
    // Process non-preferred users.
    else
    {
        // First calculate the upgrade price
        int price = calculatePrice(service);
        // Now process the upgrade with the price
        processUpgrade(user, service, price)
    }
    ...
    ...
    ...
```

Same line comments are also allowed in some cases. These are comments that exist on the same line as the code. They should be used sparingly and only if both the code line and/or the comment are very short. For example:-

```
private static final int DEFAULT_TIMEOUT = 5000;  // milliseconds
```

2.2.6 In-line Block Comments

Block comments may also be used if either a comment description goes beyond a single line or they may be used to delineate code groupings that are not explicit. Consider the following example.

```
/**
 * Begin methods that satisfy the FooBar interface.
 */

    ...
    ...
    ...

/**
 * End FooBar interface methods
 */
```

2.3 General Code Layout

Where do variables go, where to put public and private methods, inner classes, etc.

2.3.1 Package Specification and Import Statements

The package specification and all import statements should follow the file header comment (see last section) and may (optionally) be indented by 4 spaces. A blank line should separate the package spec and the import statements. A cumulative example follows:

```
/*
 * Copyright (c) 2005 XYZ inc
 *
 * www.XYZ.com
 *
 * All Rights Reserved.
 *
 * This software is the proprietary information of
 * XYZ.
 * Use is subject to license terms.
 */
 package com.xyz.utilities.net;

 import java.util.*;

/**
 * The <code>Foo</code>, performs all foo-like functions. Usage of Foo
 * is often just to illustrate some technical example,
 * having no real meaning beyond the example.
 * Geeks, in particular, really love Foo.
 *
 * Usage of <code>Foo</code> often involves using <code>Bar</code>
 *
 * <b>Note :</b> <code>Foo</code> is in no way connected to the rock band
 * Foo Fighters.
 *
 * <i>THREAD MODE: SINGLE THREADED command </i> Foo is only called by one thread
 *
 * @author     Joe Public
 * @version    $Revision: 1.10 $
 * @since      24 Mar 2005
 *
 * @see com.foobar.Bar
 */
public class Test
{
}
```

2.3.2 Class Indentation

The class declaration must begin from the left margin.

```
public class Test
{
}
```

2.3.3 Static and Instance Variable Declaration

Static and instance variables should follow the class declaration line and should be grouped by access modification first, then by functional grouping. All static and instance variables must have a comment since they are crucial to understanding what happens “under the covers” of the method signatures. Also, public variables must be commented to include additional information such as why they are public and/or the consequences of their respective usage (also since they show up in the javadoc).

All static and instance variables should be indented 4 spaces from the class declaration.

```
public static final String ASSET_BACKED_FACTOR = "AssetBackedFactor";
```

2.3.4 Method Indentation

The requirement here is that all method calls must be indented 4 spaces from the class declaration. When the class declaration (for inner classes) is indented 4 spaces, methods must be indented at 8 spaces

```
class A
{
    private void doSomething()
    {
    }

    class InnerA
    {
        private void doingSomethingInside()
        {
        }
    }
}
```

2.3.5 Method Grouping

Methods should be logically grouped together whenever possible. Candidate logical groupings are: group by getter method, group by setter method, group by interface method, etc. All method groupings should be delineated by a beginning and end comment block as specified in section 2.2.6.

2.3.6 Inner Classes

All inner classes should be grouped in their own discrete section delineated by comment blocks (e.g. “Begin inner classes” and “End inner classes”). Inner class declarations should begin in the same column of the outer class methods and variables (i.e. 4 or 8 spaces from left). Inner class methods and variables should be further indented relative to the inner class declaration and follow the same rules as for outer classes (i.e. variables at the beginning of each class, methods and variables 4 spaces from the inner class declaration).

2.3.7 Indentation

General Indentation Rules

If statements, while loops, try and catch blocks, etc.

All statement blocks (those enclosed by curly brackets) must be indented by a 4 spaces. Nested statement blocks imply further indentation corresponding to the level of nesting. For example:

```
if( num < 5 )
{
    boolean okay = doSomething();
    if( okay )
    {
        doSomethingElse();
    }
}
else
{
    nowDoSomethingCompletelyDifferent();
}
```

As is also illustrated in the previous example, curly brackets should always be used for conditional blocks, loops and try/catch blocks, even if what is between the brackets is a single statement.

2.3.8 The Dreaded Curly Bracket Issue

Leading curly bracket on a new line!

3 Java Coding Standards

3.1 Exception handling

- Rule MUST NOT CATCH AND IGNORE Exceptions.

- On the very rare occasion that you need to IGNORE an Exception. Then a large detailed comment on why this is the right thing to do, thus making it harder to ignore an exception than to do the right thing.

3.1.1 Static Exceptions

- If the debug level is enabled then ou must throw a new exception so that the correct stack trace is shown.

```
 DBObject obj = result.next();
if(obj == null)
{
	/**
	 * if the logger is in debug mode then throw new exceptions so that the
	 * correct stack trace is thrown
	 */
	if( LOGGER.isDebugEnabled())
	{
		throw new NotFoundException("Not Found");
	}
	else
	{
		throw NOT_FOUND_EXCEPTION;
	}
}
else if( result.hasRow(1))
{
	/**
	 * if the logger is in debug mode then throw new exceptions so that the
	 * correct stack trace is thrown
	 */
	if( LOGGER.isDebugEnabled())
	{
		throw new TooManyRowsException("Found " + result.getEstRowCount() + " rows");
	}
	else
	{
		throw TOO_MANY_ROWS_EXCEPTION;
	}
}
```

- Remove the stack trace from the static exceptions so that misleading information is not shown.

```
/** for performance reasons */
private static final NotFoundException NOT_FOUND_EXCEPTION = new NotFoundException("Not Found");
private static final TooManyRowsException TOO_MANY_ROWS_EXCEPTION TOO_MANY_ROWS_EXCEPTION = new
TooManyRowsException("Too many rows returned");

static
{
	/** prevent miss leading stack traces from being shown */
	NOT_FOUND_EXCEPTION.setStackTrace(new StackTraceElement[0]);
	TOO_MANY_ROWS_EXCEPTION.setStackTrace(new StackTraceElement[0]);
}
```

3.2 Variable Naming

Variable naming standards are also an important component of code readability. The naming standards below are comprised of certain known Java naming standards, some enforced usage of language constructs as well as standards specific to this project.

3.2.1 Static Variables

Static Final Variables (Constants)

Static final variables should all be capitalized with words separated by an underscore.

```
public static final int STATUS_INACTIVE = 20;
```

References to **static final** variables should be qualified by the class name but do not need to be qualified if they are private constants (since the CAPS denote them as constants and they will never be used out of the class scope). If however, the constants are not private, they should be qualified as a matter of good coding practice (even if they are referenced in the declaring class). These rules differ for non-final statics (see below).

Instance Variables

Naming of instance variables follows a the Java standard of using “camel case” which calls for lower case usage of the first word and upper case usage of subsequent words. For example the instance variable declaration for a first name would be as follows:

```
private String userName = null;
```

While declaration of instance variables is no different from other variables, the important standard to use is one that is build into the Java language. All instance variables referenced from within code should use the “this” qualification as follows:

```
this.userName = userName;
```

However, it is mandatory to be able to uniquely identify the instance variables from the function variables or parameters in order to make the code easily readable.

Parameters

As with most other Java variables, the camel case specification should be used. However, since there is no qualification mechanism for parameters All parameters should be declared **final**.

```
public void setName( final String userName)
{
    this.userName = userName;
}
```

Local Variables

Local method variables also follow the camel case standard but do not need to be qualified.

3.3 Miscellaneous

3.3.1 The ternary operator

This operator should be avoided as far as possible.

However, an occasional use of this operator is permitted for very simple logic. For example:

```
boolean flag = ( yesNo.equalsIgnoreCase( "Y" ) ? true : false );
```

3.3.2 Log4j

We should avoid using System.out and System.err all log messages should use the log4j API. In general the logger used is the class name of the program that is writing the message. The code generator will pick up the keyword #LOGGER and change the logger to the current class, this helps keeping everything in-sync.

```
private static final Logger LOGGER = CLogger.getLogger("com.xyz.App");//#LOGGER
```

3.3.3 Use of Annotations

Type Annotations allows us to write annotations in more places than before. The compiler can then verify these annotations, for example identifying uses of null values, accidental value modifications, and cases where data crosses a trust boundary without proper validation. By moving some annotatable information from the Javadoc (understood only by people) and into

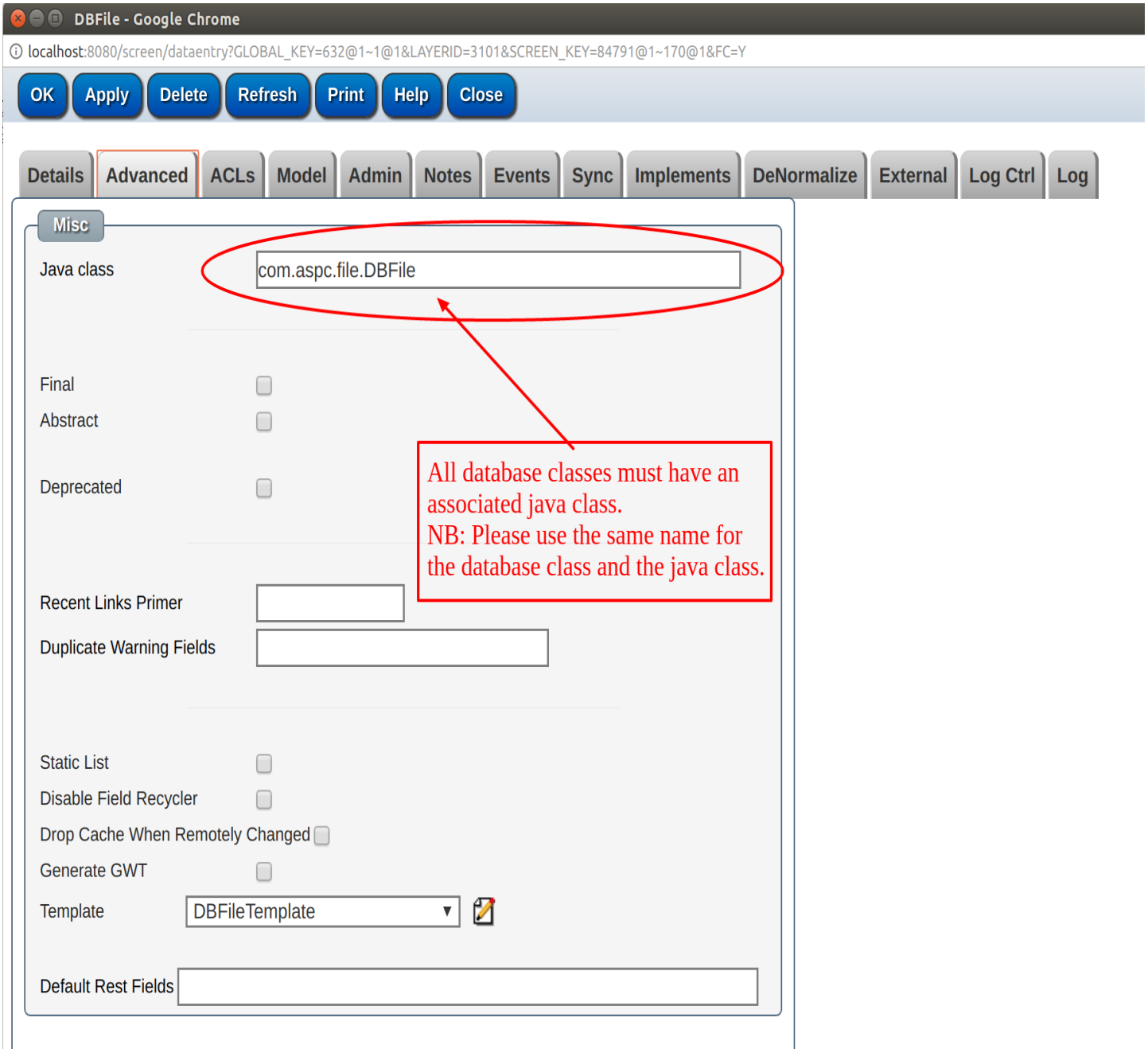
the code (understood by both people and analyzers), it is easier to understand intent and verify the absence of certain errors.

- @NonNull– The compiler can determine cases where a code path might receive a null value, without ever having to debug a NullPointerException.
- @ReadOnly– The compiler will flag any attempt to change the object. This is similar to Collections.unmodifiableList, but more general and verified at compile time.
- @Regex– Provides compile-time verification that a String intended to be used as a regular expression is a properly formatted regular expression.

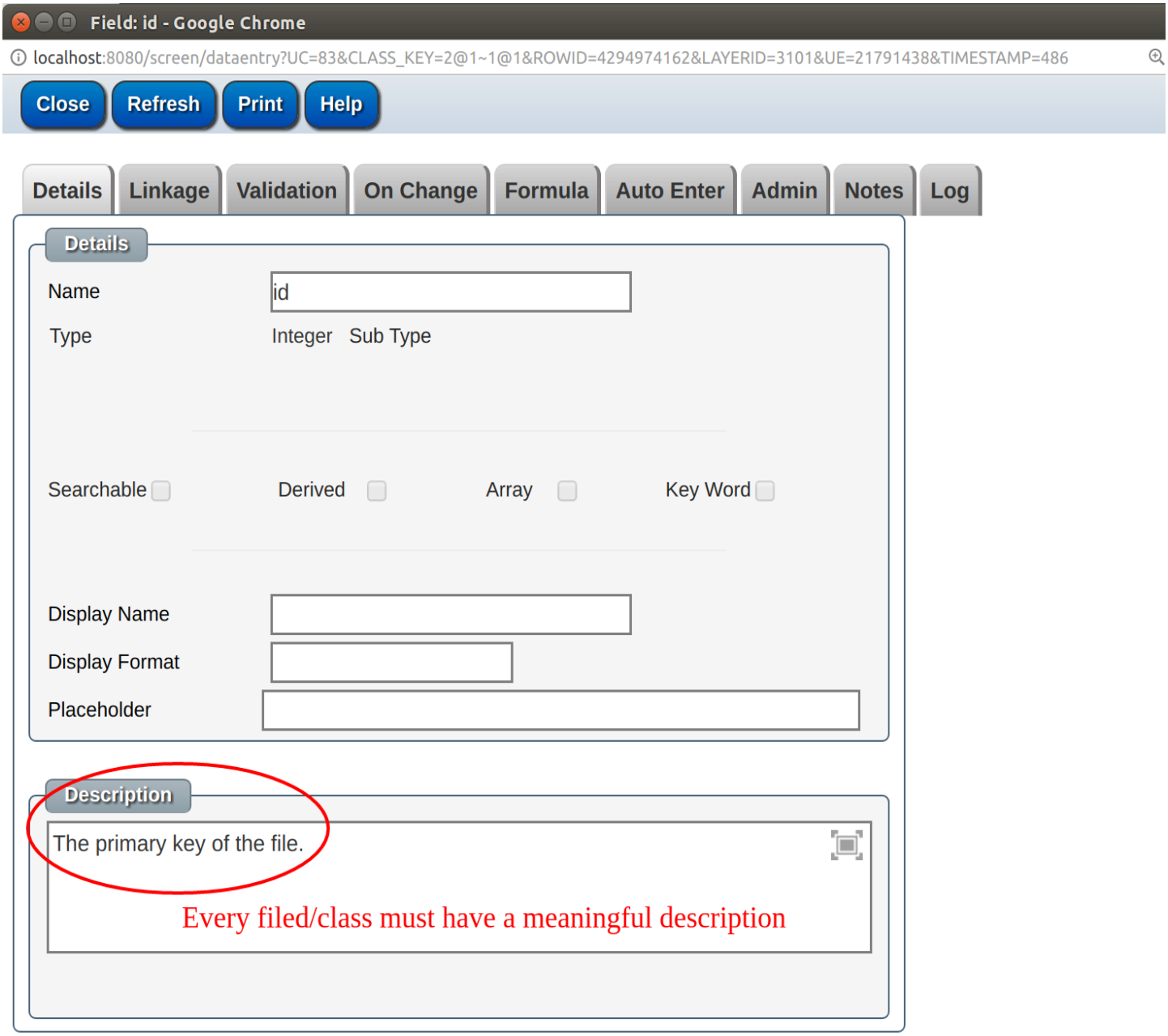
```
Private @Nonnull @CheckReturnValue String doSomething( @Nonnull int id, @Nullable String string)
{
}
```

3.4 Defining DBObjects

3.4.1 A database class must have an associated Java class.



3.4.2 Fields/Classes must have a meaningful description.



4 Self Test Guidelines

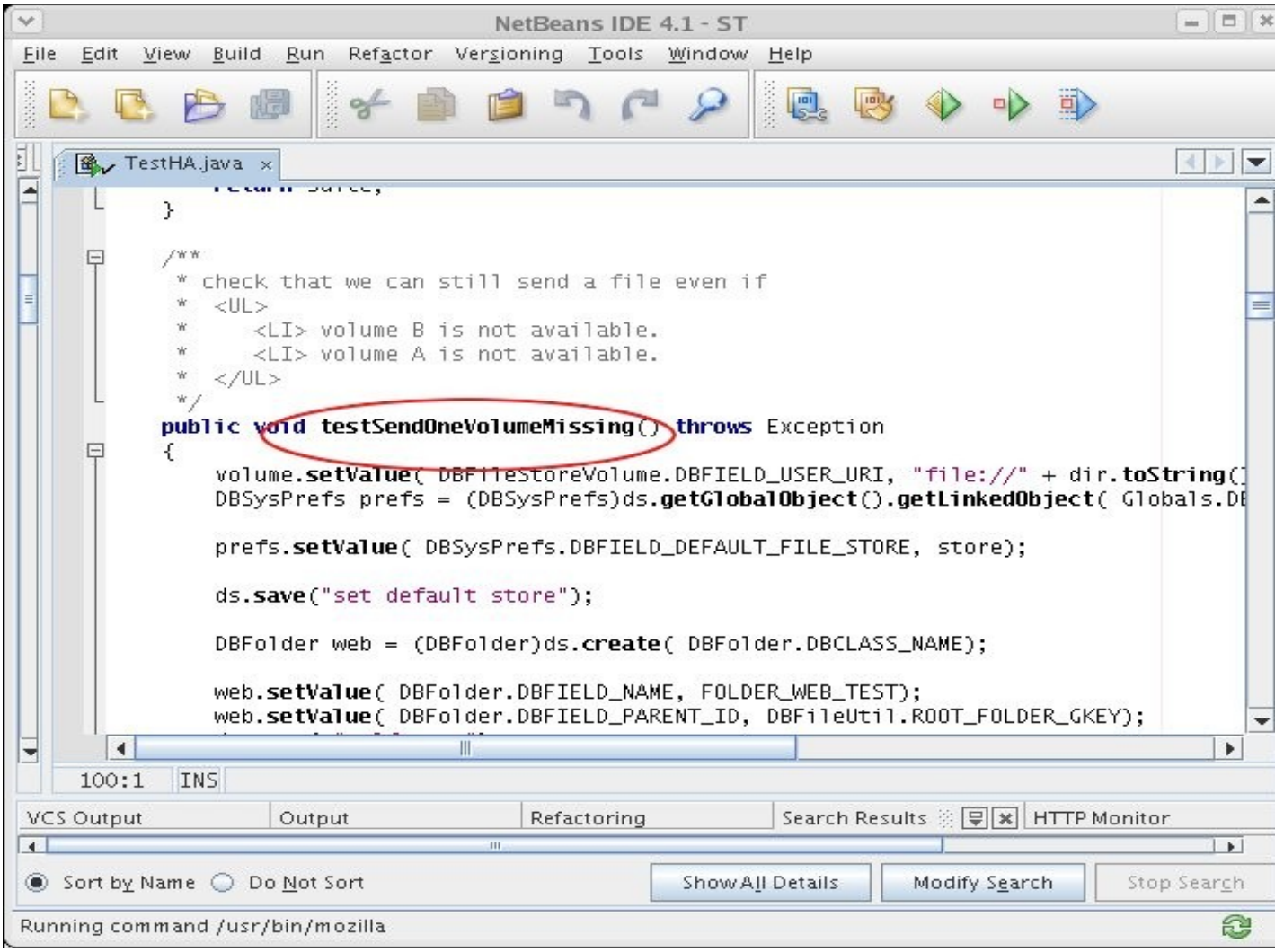
4.1 Overview

The self test framework within ST Software is developed using JUnit's framework. Following points must be followed in order to use the framework

- Every class must have a corresponding Test class (not required if the class has no Java code)
- Every overridden DBObject method must have a corresponding Test method
- No blank Test methods are allowed
- The test class must be created in the selftest package under it's base class package.
- For example, when base class package is `com.aspc.file.DBFile.java`, the self test class package will be `com.aspc.file.selftests.TestDBFile.java`

4.2 Test Methods

- For the Automated test, test method must start with 'test' word. Example: testXXX()



4.3 Sample Test Class

```
/*
 * Copyright (c) 2002 - 2017 ASP Converters pty ltd.
 *
 * www.aspconverters.com.au
 *
 * All Rights Reserved.
 *
 * This software is the proprietary information of
 * ASP Converters Pty Ltd.
 * Use is subject to license terms.
 */
package com.aspc.abc.selftest;

import com.aspc.SelfTestVirtualDBTestUnit;
import junit.framework.TestSuite;

/**
 * This is a skeleton of an example selftest class
 *
 * The following datasource is already defined in a parent class and
 * can be used to access DBObjects
 *
 *     protected MutableDataSource ds;
 *
 * Required changes are marked with "TODO"
 *
 * <br>
 * <i>THREAD MODE: SINGLE-THREADED self test unit</i>
 *
 * @author      TODO:Your Name
 * @since       August 2002
 */
public final class TestExample extends VirtualDBTestUnit
{
    /**
     * default constructor
     */
    public TestExample(String testName)
    {
```

```
        super(testName);
    }

    /**
     * required method
     */
    public static Test suite()
    {
        TestSuite suite = new TestSuite(TestExample.class);
        return suite;
    }

    /**
     * standard main
     */
    public static void main(String[] args)
    {
        parseArgs(args);
        junit.textui.TestRunner.run(suite());
    }

    /**
     * this method should populate any data sources with the data required
     * to perform all the tests
     *
     * TODO: implement code to populate this test
     */
    protected void setUp() throws Exception
    {
        super.setUp();
        // your setup code
        // the variable "ds" is defined in a parent class and can be used
        // to access DBObjects
    }

    public void testXXX()throws Exception
    {
        ;// TODO: Your code here
    }
}
```

5 Database driven messages

5.1 What is DBMessage?

DBMessage provides a mean to create and store dynamic messages to be used in the system. The message can be combination of static and dynamic content. Dynamic part of the message, resolved during runtime, can comprise of field values stored within tables.

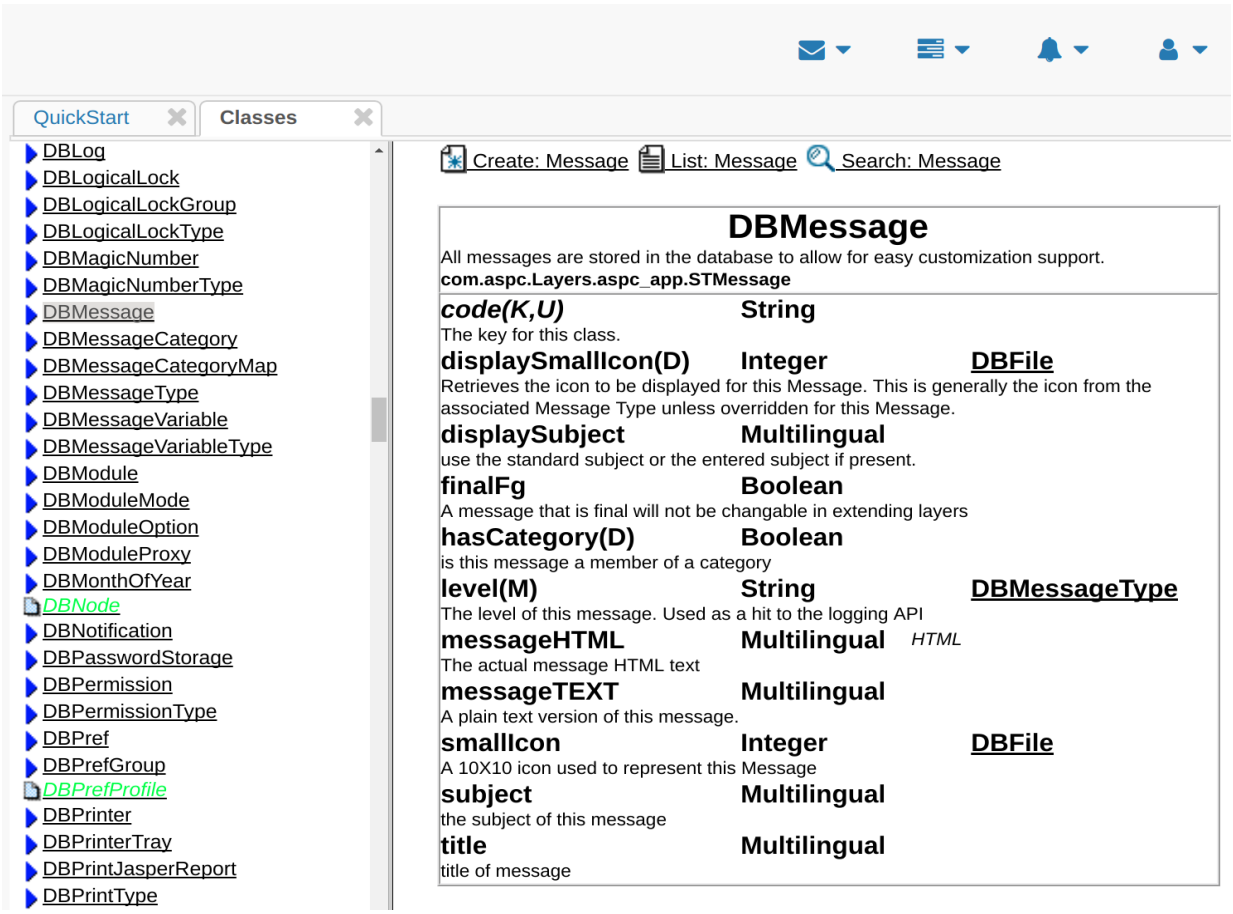
DBMessage allows the various types of the messages that appear within an application to be changed at any given point of time during the project lifecycle without having to re-build the code, since the actual message is stored in the database and is not hard-coded in the application.

Furthermore DBMessage also allow for Multilingual support by using Unicode which uses a universal character set. In addition system administrators can easily change the text displayed for a DBMessage within Database settings.

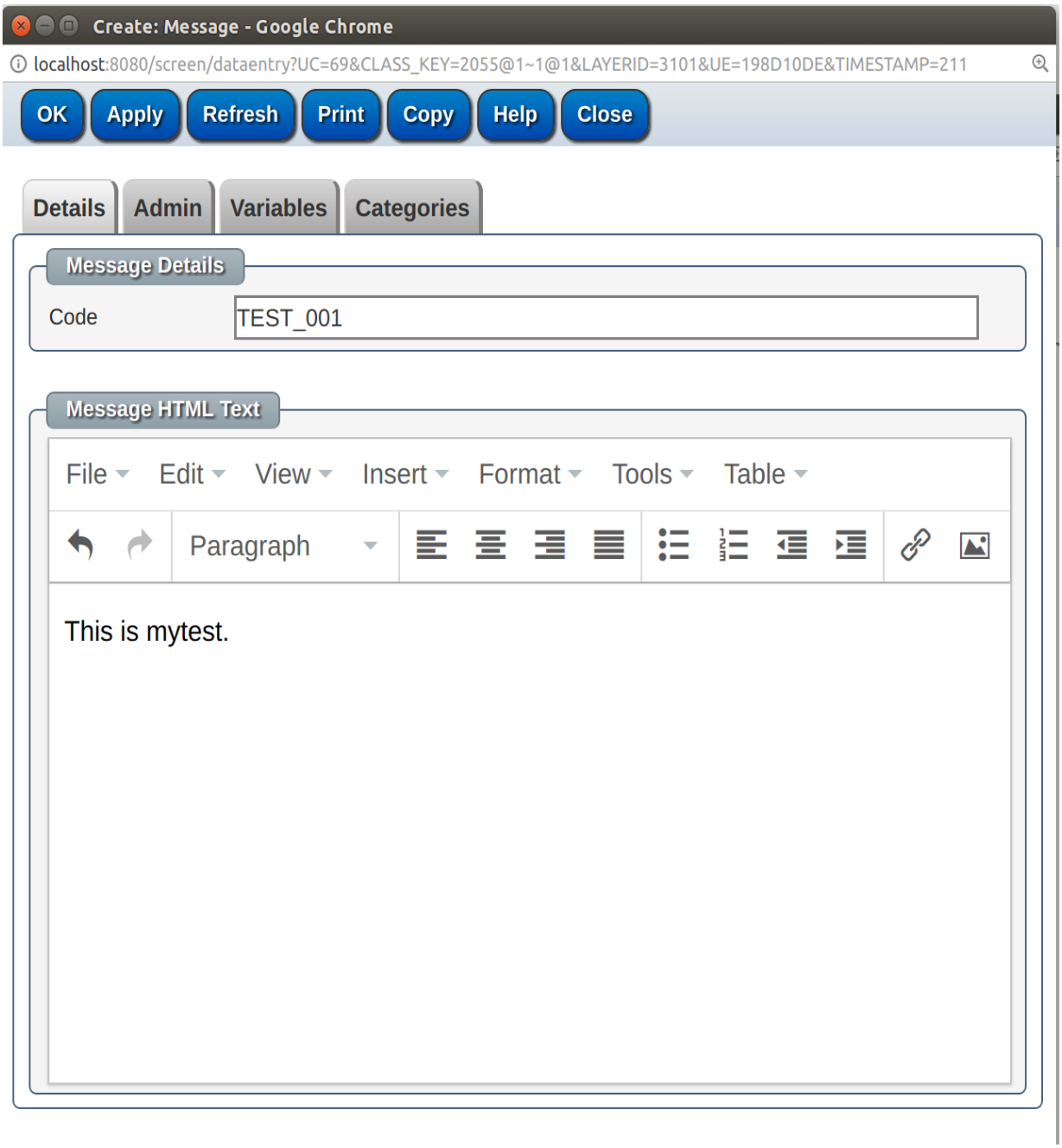
5.2 How to create DBMessage?

Follow the steps below to create a DBMessage

1. Select Customize->Classes while logged in into the correct layer. Choose DBMessage from the selection of classes

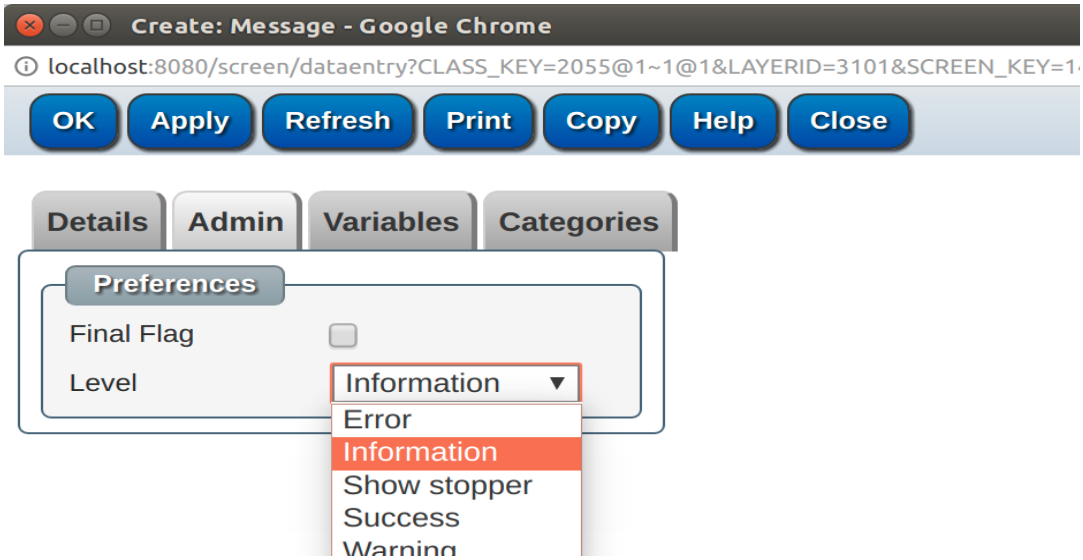


2. Click on Create: Message link. This will display the following window



3. Type in a string identifier within the 'Code' text box. This code will uniquely identify the DBMessage. You can set various styles to the message.

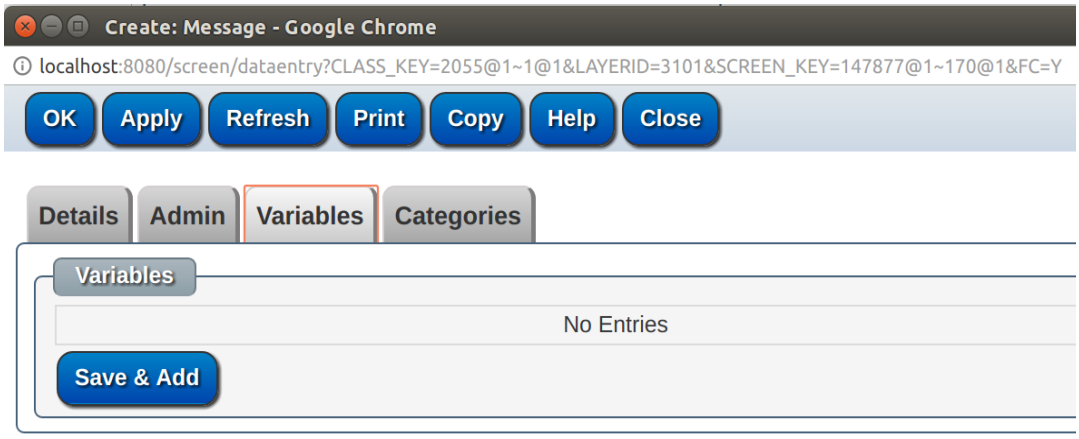
4. Using the admin tab on this window, different message levels (Error, Information, Warning etc.) can be set. This will be used in the future to display appropriate icon on the message box containing the DBMessage depending on the message level.



5.3 Variables

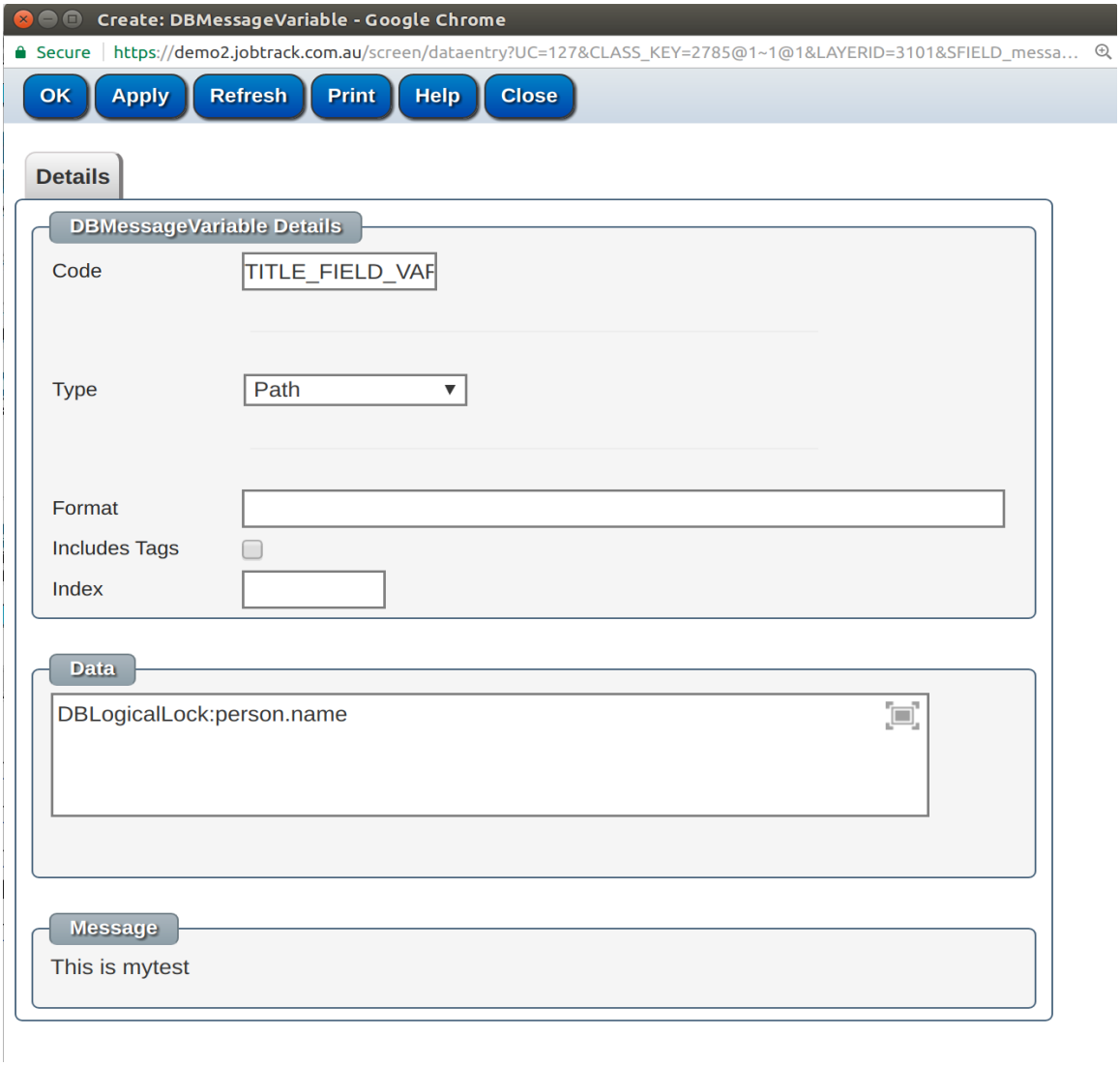
Variables are used to add dynamic content to the DBMessage. This is done by adding ${var} to the message text, where {$var} represents the code of the specific variable. Each Variable added is related to an argument that is passed within the source code.

To create a new variable click on the Add button on variables tab as shown below:



Selecting the Add button will invoke the variables data entry screen.

5.4 The Variable data entry screen



- The *Code* field is the unique key used to identify the variable that matches ${Var}.
- The *Type* field is the type of the variable which may be a simple string or a specific class path.
- The *Format* field identifies how the variable is to be displayed. For example, you may set the a date variable to be displayed as 01 JAN, or 01 JAN 2001, 01/01/2001, etc.
- The *index* field is the position in the list of arguments passed to the source code.

5.4.1 Types

Path types:

For a variable of type ‘Path’. The logical path of the variable is entered into ‘Data’ section of the screen e.g. DBLogicalLock:person.name.

The path is then applied to the object passed in as an argument at the index position to retrieve a value.

The path variable can also be used to retrieve from Globals. e.g. Globals:currentLogin.loginId

String type:

This is used to when the argument passed is a string, and the DBMessage will display the value of the string set in the 'Data' section.

5.5 How to invoke DBMessage?

Example 1

```
/**
 * Example of how to fetch the HTML text version of the message.
 * This text component can then be added to a HTML page directly.
 *
 * The passed datasource holds the user"s preference for the
 * language and timezone to use when generating the message text.
 */
HTMLComponent text;

text = DBMessage.fetchComponent(
    context.getDS(),
    DBMessage.LIST_DB_LOGICAL_LOCK_BREAK,
    context.getBase(),
    lock
);
```

Example 2

```
/**
 * Example of how to fetch the plain text version of the message.
 * The resulting plain text can then written to a log file or
 * displayed on the console.
 *
 * The passed datasource holds the user"s preference for the
 * language and timezone to use when generating the message text.
 */
String plainText;

plainText = DBMessage.fetchText(
    context.getDS(),
    DBMessage.LIST_DB_LOGICAL_LOCK_BREAK,
    context.getBase(),
    lock
);
```

Example 3

```
/**
 * additional validations.
 *
 * @return A validation error.
 * @param list The full validation list errors & warnings.
 * @param field The field to validate.
 * @throws Exception A serious problem.
 */
protected ValidationError extValidateField(
    DBField field,
    ValidationList list
) throws Exception
{
    ValidationError ve;

    ve = super.extValidateField( field, list);

    if( ve != null) return ve;

    double dps = getDouble( DBFIELD_DPS);

    /**
     * You can"t have a negative DPS
     */
    if( dps < 0 )
    {
        ve = list.createError(
            DBMessage.LIST_NEG_DPS,
            this,
            field,
            new Double( dps)
        );

        return ve;
    }
}
```

6 Generating Java Code

6.1 Overview

Every table created within correct layer is associated with a Java class. An application is used to generate such a class, the output of which is a fully documented java class containing all the fields defined within the table.

6.2 Field and Class name constants.

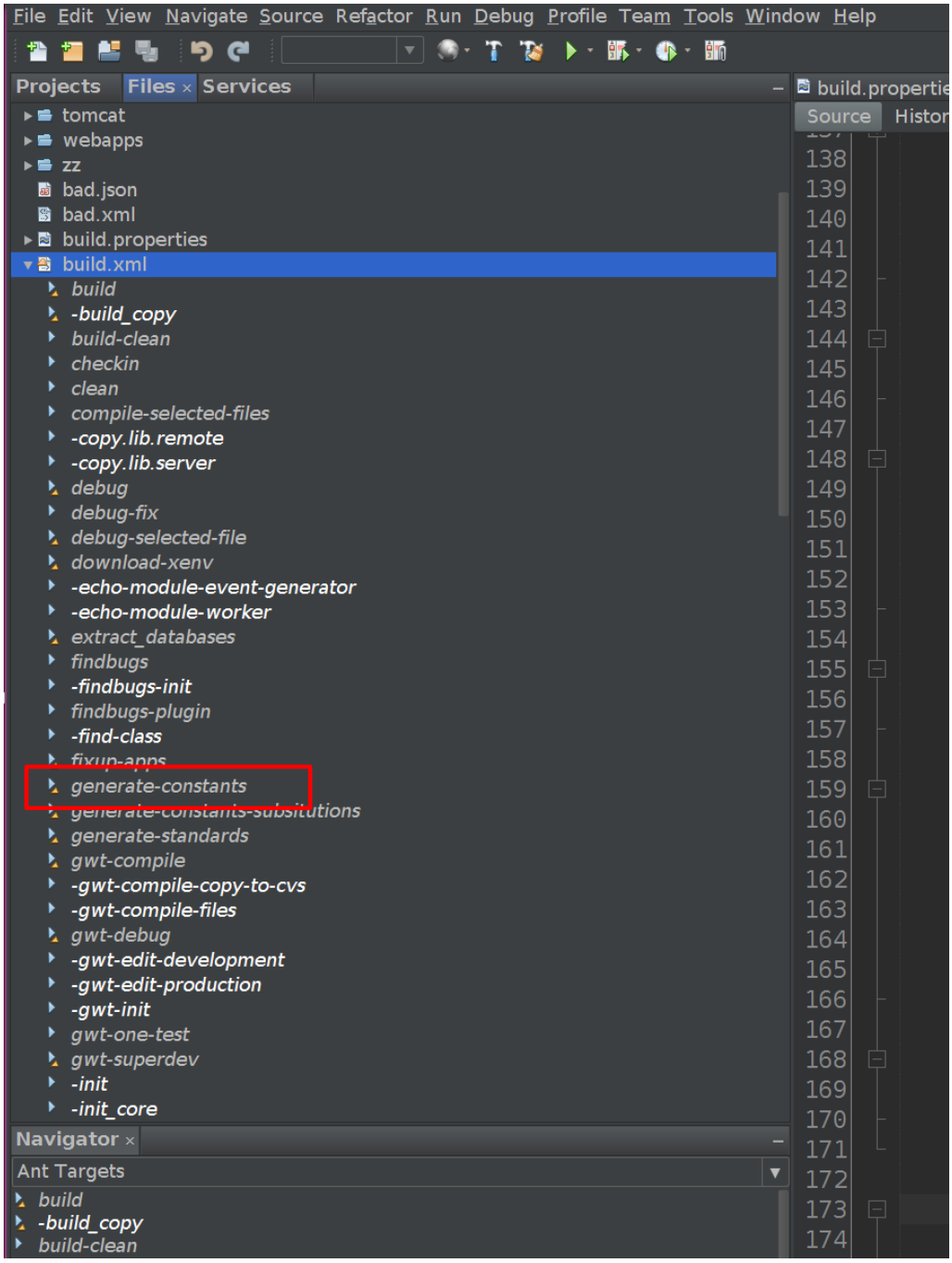
```
//#ASPC_GEN_START <editor-fold defaultstate="collapsed" desc="Auto generated constants">
/** A job which will be processed in the background. */
public static final String   DBCLASS_NAME="DBJob";

/** A job which will be processed in the background. */
public static final com.aspc.DBObj.GlobalId DBCLASS_GID = GlobalId.cacheGlobalId("2555@1");

/**
 *  The primary key for this job.                                          <BR>
 *                                                                         <BR>
 *   (Integer, KEY)
 */
public static final String DBFIELD_ID="id";

/**
 *  The file that holds the raw commands of the users request.             <BR>
 *                                                                         <BR>
 *   --&gt; {@link com.aspc.DBObj.File.DBFile DBFile} (Integer, Searchable)
 *                                                                         <BR>
 *  <B>Notes</B><BR>
 *                                                                         <BR>
 *  Validation
 *  ----------
 *  1) We should only have one of "user command file", "auto command file" and commandRawTx.
 */
public static final String DBFIELD_COMMAND_USER_FILE="commandUserFile";
```

6.3 Run code generator.



7 Code Review

7.1 Completion of a task

Before completing a bug/task, the developer should go through IDE hints and the Auto Comment tool on any code that you have touched or created.

8 MULTI-THREADS

- Class must have Multi-Thread in header if it designed to handle multi threads.
- Multi-threaded classes MUST be reviewed by another programmer.
- Multi-Thread program MUST be reviewed.

8.1 SECONDARY CACHE DESIGN

```
/*
 * Copyright (c) 2000-2004 ASP Converters pty ltd
 *
 * www.aspconverters.com.au.
 *
 * All Rights Reserved.
 *
 * This software is the proprietary information of
 * ASP Converters Pty Ltd.
 * Use is subject to license terms.
 */
import com.aspc.DBObj.*;
import com.aspc.DBObj.Listeners.*;

/**
 * Use a secondary cache to fetch a "Thing". We are in a multi-machine / multi-processor /
 * multi-user / multi-threaded environment. Records can and do change at any time from one
 * line to the next. A few points to look out for:-
 *
 *    1) All work must be done with local variables so that we don"t get Null
 *       Pointer Exceptions when the cache is cleared while we are in this method.
 *
 *    2) Threads can take their copies of object variables which are only flushed/sync"d when
 *       synchronized is called on the object.
 *
 *    3) Database queries etc. can take a while to run (specially if the query returns multiple rows)
 *       a record in the result set maybe changed and a message sent/clear cache called before the
 *       result is returned. This case must be handled ( it happens a lot with bulk records)
 *
 *    4) Synchronizing the method synchronizes the whole Object. So for complex objects like Company
 *       or DBClass which may have many secondary caches we would be blocking a fetch of something
 *       that is in memory due to a fetch of something else that is not.
 *
 *    5) Having complex logic within the synchronized block which calls other objects with
 *       synchronized blocks it is easy to cause Java deadlocks. A deadlock within Java will NEVER
 *       return unlike a normal database deadlock.
 */
public class Bits extends DBObject implements DependanceListener, ReloadEventListener
{
    /**
     * Std. DBObject constructor.
     *
     * @param def The class of this object
     * @param dataSource The datasource for this object
     * @throws Exception A serious problem occurred
     */
    public Bits(DBClass def, DataSource dataSource) throws Exception
    {
        super( def, dataSource);
    }
```

```
/**
 * Sample secondary cache.
 *
 * Step 1.
 *   Enter a synchronized block so that we see a clean version of cache of "Thing" and the cache
 *   of thing is prevented while we are within this block.
 *
 * Step 2.
 *   If the cache handle is null then create a new handle and set the local copy. If any
 *   clear cache events are now called it"ll clear the handle and the next call will reload the
 *   cache but this call will continue with the local handle.
 *
 * Step 3.
 *   If what the handle points to is null then do the Slow search and set the local handle.
 *   There is some question on whether we should sync the setting of the local handle, I
 *   believe not as another thread not getting the new value (which is flushed fairly frequently)
 *   would just result in another search. If the object"s version of the handle hasn't been
 *   cleared i.e. same as the local version it is now set.
 *
 * Step 4.
 *   Return the value of the local handle which will never be null.
 */
public Thing getCacheThing() throws Exception
{
    Thing holder[] = null;

    // OPTIONAL A: Safer if sync block is here
    synchronized( this)// Step 1.
    {
        holder = cacheThing;

        if( holder == null)
        {
            // OPTIONAL B: Faster if the sync block is here. ( must have A or B)
            holder = new Thing[1];

            cacheThing=holder; // Step 2
        }
    }

    if( holder[0] == null) // Step 3.
    {
        DBQuery q = new DBQuery( Thing.DBCLASS_NAME, getDS());

        q.addClause( /* A complex/slow search criteria */);

        DBObject obj = q.findOne();
        // Enter a new synchronized block to prevent reordering of instructions
        synchronized( this)
        {
            holder[0] = obj;
        }
    }

    return holder[0]; // Step 4.
}

/**
 * A dependent of Bits has been added. This may effect the secondary cache so we should clear it.
 *
 * @param addedKey The dependent added
 * @param sourceFieldKey The field that points to this object
 */
public void eventDependantAdded( GlobalKey addedKey, GlobalKey sourceFieldKey)
{
     clearCache( addedKey);
}
```

```
	/**
	 * A that we are watching has been changed
	 *
	 * @param obj The DBObject that was reload.
	 */
	public void eventReload( DBObject obj )
	{
		clearCache( obj.getGlobalKey());
	}

	/**
	 * A dependent of Bits has been removed. This may effect the secondary cache.
	 *
	 * @param removedKey The DBObject was removed.
	 * @param sourceFieldKey The linked field
	 */
	public void eventDependantRemoved( GlobalKey removedKey, GlobalKey sourceFieldKey)
	{
		clearCache( removedKey);
	}

	/**
	 * We should only clear the cache if the record changed could have possibly effected the cache.
	 * This method we be called MANY times. So it is cheaper just to clear the cache if in any doubt.
	 * Eg. If you are holding the primary security for this Company and the class of the changed
	 * object is "security" don"t go selecting it here to work out if you should clear it or not.
	 *
	 * This is automatically called by eventDataLoaded() in DBObject which does a programmer check
	 * that you have call the super.clearCache( changedKey);
	 */
	protected void clearCache( GlobalKey changedKey)
	{
	    super.clearCache( changedKey);

	    if( /* only clear if the changed object effects the secondary cache */)
	    {
			synchronized( this)// minimize the time we spend in synchronized blocks
			{
				cacheThing = null;
			}
	    }
	}

	private Thing[] cacheThing;
}
```

9 SyncBlock A Replacement of the synchronized keyword

To prevent deadlocks we have implemented a new class called SyncBlock this is intended as a drop in replacement of the keyword.

When you call the method take() on a SyncBlock object you'll block up until the specified maximum number of seconds and then an error will be thrown if you are unable to obtain the lock on this object. You must ALWAYS call **release()** on any sync SyncBlock that you have obtained the lock on.

The SyncBlock differs from the keyword **synchronized** in that it is interruptible and that it will timeout if it blocks for too long.

The SyncBlock enhances a normal java.lang.concurrent.Lock in that it will interrupt the blocking thread if it holds the lock for too long and if the thread holding the lock is not alive a new lock object will be created and a fatal email will be generated with the details. If a lock fails to be obtained (which would have been a deadlock) a fatal email will be generated with the details of the two threads and the blocking thread will be interrupted and the calling thread will have an error thrown.

Please see below an example of how to use, the associated test cases and the code listing itself.

/home/parminder/src/ST/com/aspc/DBObj/VirtualDB.java

```
/**
* notify all the class listeners
* @param type the type of change MODIFY,DELETE or CREATE
* @param gk the global key to notify of.
*/
private void notifyDBClassListeners( final String type, final GlobalKey gk)
{
	String key = gk.getClassId().toString();
	/**
	 * DEADLOCK found when this was a synchronized block.
	 * now we are using a SyncBlock lock object which will timeout after 2 minutes if not successful.
	 *
	 * Once you have taken the lock the next statement must be the start of the try block so we neve
	leave this
	 * section without releasing the lock.
	 */
	List list = null;

	dbClassListenersLock.take();
	try
	{
		list = dbClassListeners.get(key);
	}
	finally
	{
```

```
                /**
                 * Always release the lock if obtained.
                 */
                dbClassListenersLock.release();
        }

        if(  list != null)
        {
                int pos =0;
                while( true)
                {
                        DBClassListener listener;
                        dbClassListenersLock.take();
                        try
                        {
                                if( pos >= list.size()) break;
                                listener = (DBClassListener)list.get( pos);
                                }
                                finally
                                {
                                        /**
                                         * Always release the lock if obtained.
                                         */
                                        dbClassListenersLock.release();
                                }
                                pos++;

                                try
                                {
                                        switch (type)
                                        {
                                                case DBData.NOTIFY_MODIFIED:
                                                        listener.eventObjectModified( gk, this);
                                                        break;
                                                case DBData.NOTIFY_DELETED:
                                                        listener.eventObjectDeleted( gk, this);
                                                        break;
                                                case DBData.NOTIFY_CREATED:
                                                      listener.eventObjectCreated( gk, this);
                                                      break;
                                                default:
                                                      LOGGER.error( "Wrong type:" + type);
                                                      break;
                                        }
                                }
                                catch( Throwable t)
                                {
                                      SUPPRESSED_ISSUE.set(t);
                                      LOGGER.warn( "ignored exception in listener", t); // Sr, 12/05/2005 Bug
                                                                                       #5224
                                }
                        }
                }
        }

}
```

/home/parminder/src/ST/com/aspc/remote/util/misc/SyncBlock.java

```
/**
 * release the lock
 */
public void release()
{
	syncLock.unlock();
}

/**
 * try to get the lock
 * @return true if we got the lock
 */
public boolean tryLock( )
{
	return syncLock.tryLock();
}

/**
 * Try for a few seconds to get the lock
 * @param seconds the number of seconds
 * @return true if we got the lock
 * @throws InterruptedException interrupted.
 */
public boolean tryLock( final int seconds) throws InterruptedException
{
	return syncLock.tryLock(seconds, TimeUnit.SECONDS);
}

/**
 * take the lock and throw an error if you can't get it.
 */
public void take()
{
	try
	{
		SyncLock tempLock = syncLock;
		if (syncLock.tryLock(blockSeconds, TimeUnit.SECONDS) == false)
		{
			ERROR_COUNT.incrementAndGet();
			Thread ownerThread = syncLock.getOwner();

			if(ThreadUtil.isAliveOrStarting( ownerThread) == false)
			{
				synchronized( this)
				{
					if( tempLock == syncLock)
					{
						syncLock = new SyncLock();
					}
				}

				CLogger.fatal(LOGGER, this + " never released by " + ownerThread);
				take();
				return;
			}

			StringBuilder sb = new StringBuilder( toString());
			sb.append("\n");
```

```
                    Thread currentThread = Thread.currentThread();

                    sb.append("Failed to get lock for thread: ").append(currentThread).append("\n");

                    for (StackTraceElement ste : currentThread.getStackTrace())
                    {
                        sb.append("\t").append(ste).append("\n");
                    }

                    if( ownerThread != null)
                    {
                        sb.append("\nLock held by thread: ").append(ownerThread).append("\n");

                        for (StackTraceElement ste : ownerThread.getStackTrace())
                        {
                            sb.append("\t").append(ste).append("\n");
                        }
                        sb.append("Interrupting holding thread");
                        ownerThread.interrupt();
                    }
                    else
                    {
                        sb.append("NO OWNER THREAD found");
                    }

                    CLogger.fatal(LOGGER, sb.toString());

                    throw new DataBaseError("could not get the lock on: " + name);
                }
            }
            catch (InterruptedException ex)
            {
                ERROR_COUNT.incrementAndGet();
                Thread.interrupted();
                CLogger.warn(LOGGER, "could not take lock on " + name, ex);
                Thread.currentThread().interrupt();
                throw new DataBaseError("could not get the lock on: " + name, ex);
            }
        }

        class SyncLock extends ReentrantLock
        {
            private static final long serialVersionUID = 42L;

            public SyncLock( )
            {
                super( true);
            }
            /**
             * get the owner thread
             * @return the owner thread
             */
            @Override
            public Thread getOwner()//NOPMD
            {
                return super.getOwner();
            }
        }
```

/home/parminder/src/ST/com/aspc/SelfTest/remote/misc/selftest/TestSyncBlock.java

```
/**
 * check we recover from a lock that is never released.
 * @throws InterruptedException
 */
public void testNeverReleased() throws InterruptedException
{
	final SyncBlock block = new SyncBlock( "never release", 2);

	Runnable r = block::take;
	Thread t = new Thread( r);
	t.start();

	t.join( 120000);

	block.take();
}

/**
 * check that we actually do block
 * @throws InterruptedException
 */
@SuppressWarnings("empty-statement")
public void testBlock() throws InterruptedException
{
	final SyncBlock block = new SyncBlock( "long time", 10);

	Runnable r = () -> {
		block.take();
		try
		{
			Thread.sleep(120000);
		}
		catch (InterruptedException ex)
		{
			LOGGER.warn("interrupted");
		}
		finally
		{
			block.release();
		}
	};
	Thread t = new Thread( r);
	t.start();

	t.join( 1000);

	try
	{
		block.take();
		fail( "should not succeed");
	}
	catch( Throwable tw)
	{
		;// this is good
	}
	t.interrupt();

	t.join( 5000);
```

```
        block.take();
    }


    /**
     * check that deadlocks are handled
     * @throws Exception a test failure
     */
    @SuppressWarnings("SleepWhileInLoop")
    public void testDeadlockHandled() throws Exception
    {
        a=new A();
        b=new B();
        Thread at = new Thread( a);

        at.start();
        Thread bt = new Thread( b);

        bt.start();

        long start = System.currentTimeMillis();
        while( start + 120000 > System.currentTimeMillis())
        {
            if( a.calling && b.calling ) break;
            Thread.sleep(100);
        }

        synchronized( marker)
        {
            marker.notifyAll();
        }
        LOGGER.info("waiting for detection");
        at.join(240000);
        bt.join(240000);

        if( a.theException == null && b.theException == null)
        {
            fail( "The threads were not interrupted");
        }

        assertFalse( "should have finished",  ThreadUtil.isAliveOrStarting(at));
        assertFalse( "should have finished",  ThreadUtil.isAliveOrStarting(bt));
    }

    class A implements Runnable
    {
        private final SyncBlock block = new SyncBlock("A block", 2);
        boolean calling;
        Throwable theException;

        @Override
        public void run()
        {
            try
            {
                callB();
            }
            catch( Throwable e)
            {
                theException = e;
                LOGGER.warn( "got cancelled", e);
```

```
            }
        }

        public void hello()
        {
            block.take();
            try
            {
                LOGGER.info("hello A");
            }
            finally
            {
                block.release();
            }
        }

        private void callB() throws InterruptedException
        {
            block.take();
            try
            {
                calling=true;
                synchronized( marker)
                {
                    marker.wait(120000);
                }
                LOGGER.info("call B");
                b.hello();
            }
            finally
            {
                block.release();
            }
        }
    }

    class B implements Runnable
    {
        boolean calling;
        Throwable theException;
        private final SyncBlock block = new SyncBlock("A block", 2);

        @Override
        public void run()
        {
            try
            {
                callA();
            }
            catch( Throwable e)
            {
                theException = e;
                LOGGER.warn( "got cancelled", e);
            }
        }

        public void hello()
        {
            block.take();
            try
            {
```

```
                LOGGER.info("hello B");
            }
            finally
            {
                block.release();
            }
        }

        private void callA() throws InterruptedException
        {
            block.take();
            try
            {
                calling=true;
                synchronized( marker)
                {
                    marker.wait(120000);
                }
                LOGGER.info("call A");
                a.hello();
            }
            finally
            {
                block.release();
            }
        }
    }

    private A a;
    private B b;
```