

Check list for the KIEL review process

Steffen H. Prochnow*

File: review-checklist.tex
Revision: 1.9
Revisiondate: March 10, 2005
Revisiontime: 14:28:20
Compiledate: March 15, 2005
Compiletime: 14:37

1 Basic checks

Before you delve into the source code, your analysis needs to be corroborated running the following fundamental checks:

1.1 Checkstyle

Run the build script within the module directory using the command line:

```
> ./build.sh stylechecker
```

If this results in any warnings, then inspect these and comment them.

1.2 Jlint

Run the build script within the module directory using the command line:

```
> ./build.sh static-checker
```

If this results in any warnings, then inspect these and comment them.

1.3 Javadoc

Run the build script within the module directory using the command line:

```
> ./build.sh javadoc
```

If this results in any warnings, then inspect these and comment them.

*Thanks for substantial contributions to Florian Lüpke.

2 Assessing checks

2.1 General conspectus

- 2.1.1. Does the structure and the concept of the source code agree to the project manual of KIEL?
- 2.1.2. Do a random inspection of the javadoc output! Does the entries and conventions agree to the project manual?
- 2.1.3. Hence, are all identifiers and is all of the source code documentation written in English?
- 2.1.4. **Important:** Check interceptions that ensure (any limitations of) the software functionality. Intention: No error should appear if an unexpected value occurs.

Examples:

- catching the appearing of inter level transitions
- catching an unexpected parse token

2.2 Clarity

- 2.2.1. Is the code understandable?
- 2.2.2. If not: Are there any explaining comments?
- 2.2.3. Are there any “magic” numbers or strings? Use static fields instead.

2.3 Interfaces

- 2.3.1. Interfaces that are to be implemented by only one class should be removed.
- 2.3.2. Interfaces that are simply a collection of constants should be removed. Constants should be defined there where they are used mostly or mainly.

2.4 Classes

- 2.4.1. Unnecessary wrappers should be removed. A wrapper is unnecessary, if it does not implement more or in another way than the wrapped object.

Example:

```
public class DefaultHistoricalData implements Serializable {  
    private Vector statusHistory = new Vector();  
    public void addStatusHistoryEntry(StatusHistoryEntry statusHistoryEntry) {
```

```

    this.statusHistory.add(statusHistoryEntry);
}
public Enumeration getStatusHistoryEntries() {
    return statusHistory.elements();
}
public StatusHistoryEntry getStatusHistoryEntryAt(int pos) throws
    NoSuchElementException {
    return (StatusHistoryEntry)statusHistory.get(pos);
}
public int getStatusHistoryEntrySize() {
    return statusHistory.size();
}
public boolean hasStatusHistoryEntries() {
    return !statusHistory.isEmpty();
}
public void setStatusHistory(Vector newStatusHistory) {
    try {
        statusHistory = new Vector();

        Iterator iter = newStatusHistory.iterator();
        while (iter.hasNext())
            addStatusHistoryEntry((StatusHistoryEntry) iter.next());
    } catch (Exception e) {
        this.statusHistory = new Vector();
    }
}
}
}

```

2.4.2. Classes, which have or contain the name “test”, should be removed.

2.4.3. Abstract classes that only contain one subclass should be removed. The implemented code can be moved to the subclass.

2.4.4. Classes, whose static methods are to be focused only by one class, should be removed. Their code should be implemented in the calling class.

Example:

```

public class PlainMonthYearParser {

    final static public String FROM_MONTH = "MONTH";
    final static public String FROM_YEAR = "YEAR";

    final static public String TO_SINGLE_MONTH = "Monat";
    final static public String TO_SINGLE_YEAR = "Jahr";
    final static public String TO_MULTIPLE_MONTH = "Monate";
    final static public String TO_MULTIPLE_YEAR = "Jahre";

    public static String transform(String string) {
        if (endsWithMonth(string))
        {
            return replaceMonth(string);
        }
        if (endsWithYear(string))
        {
            return replaceYear(string);
        }
        return string;
    }
}

```

```

}
public static boolean endsWithMonth(String string) {
    return string.endsWith(FROM_MONTH);
}
public static String replaceMonth(String string) {
    String result = string;
    if (endsWithMonth(string))
    {
        result = string.substring(0, string.indexOf(FROM_MONTH));
        if (string.startsWith("1_"))
        {
            result += TO_SINGLE_MONTH;
        }
        else
        {
            result += TO_MULTIPLE_MONTH;
        }
    }
    return result;
}
public static boolean endsWithYear(String string) {
    return string.endsWith(FROM_YEAR);
}
public static String replaceYear(String string) {
    String result = string;
    if (endsWithYear(string))
    {
        result = string.substring(0, string.indexOf(FROM_YEAR));
        if (string.startsWith("1_"))
        {
            result += TO_SINGLE_YEAR;
        }
        else
        {
            result += TO_MULTIPLE_YEAR;
        }
    }
    return result;
}
}

```

2.4.5. Useless classes, which were possibly used for test cases, have to be removed.

Example:

```

public class BasicWindowMonitor extends WindowAdapter {

    /**
     * BasicWindowMonitor constructor comment.
     */
    public BasicWindowMonitor() {
        super();
    }

    /**
     * Insert the method's description here.
     * Creation date: (24.01.2002 07:49:26)
     */
    public void windowClosing(WindowEvent e) {
        Window w = e.getWindow();
        w.setVisible(false);
    }
}

```

```

        w.dispose();
        //System.exit(0);
    }
}

```

2.4.6. Classes that do nothing should be removed, unless their type is asked for.

Example:

```

public class JFrameWithWindowListener extends JFrame implements
    WindowListener {
    // constructor

    public JFrameWithWindowListener() {
        super();
        init();
    }

    public JFrameWithWindowListener(GraphicsConfiguration gc) {
        super(gc);
        init();
    }

    public JFrameWithWindowListener(String title) {
        super(title);
        init();
    }

    public JFrameWithWindowListener(String title, GraphicsConfiguration gc) {
        super(title, gc);
        init();
    }

    // init
    private void init() {
        this.addWindowListener(this);
    }

    // WindowListener implementation
    public void windowActivated(WindowEvent e) {}

    public void windowClosed(WindowEvent e) {}

    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }

    public void windowDeactivated(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowOpened(WindowEvent e) {}
}

```

2.4.7. Code that is contributed to the code only for testing components should only be dynamically referenced and offered by the class `ClientInfrastructureFactory` respectively `ServerInfrastructureFactory`.

Example:

(Pay attention to the fact that the constructor of `CustomerClientManager` is private.)

```
public class CustomerClientManager {  
  
    private static CustomerClientManager instance;  
  
    public CustomerClientManager getInstance() {  
        if (instance == null) {  
            if (ClientInfrastructureFactory.simulateCustomerClientManager())  
                instance = InfrastructureXY.createCustomerClientManager();  
            else  
                instance = new CustomerClientManager();  
        }  
        return instance;  
    }  
}
```

2.4.8. Classes for testing, which are contributed to the package `xy`, should be placed in the package `xy.test`.

2.4.9. Classes, which contain less than 10 lines of code, should be avoided (except interfaces, abstract classes and exceptions).

2.5 Initialization and Declarations

2.5.1. Are variables and class members of the correct type and appropriate mode?

2.5.2. Are variables declared in the proper scope?

2.5.3. No variable should be defined that is read only once.

Example:

```
public void reconnect() throws JMSEException {  
    connection = factory.createQueueConnection();  
    connection.start();  
    boolean transacted = false;  
    // CLIENT_ACKNOWLEDGE assures the persistent of received messages  
    session = (MQQueueSession) connection.createQueueSession(transacted, Session.  
        .CLIENT_ACKNOWLEDGE);  
    queue = session.createQueue(strQueueURI);  
    queueReceiver = session.createReceiver(queue);  
}
```

It's better to write:

```
public void reconnect() throws JMSEException {  
    connection = factory.createQueueConnection();  
    connection.start();  
    // CLIENT_ACKNOWLEDGE assures the persistent of received messages  
    // Parameter FALSE means NOT TRANSACTIONAL  
    session = (MQQueueSession) connection.createQueueSession(false, Session.  
        CLIENT_ACKNOWLEDGE);  
}
```

```
queue = session.createQueue(strQueueURI);
queueReceiver = session.createReceiver(queue);
}
```

2.6 Method Calls

2.6.1. Are parameters presented in an established or expected order?

Example:

bad:

```
rectangle(x, width, y, height)
```

better:

```
rectangle(x, y, width, height)
```

2.6.2. Is the correct method being called, or should it be a different method with a similar name (correct names, overloads)?

2.6.3. Are method return values used properly?

2.6.4. Elements that are not referenced should be removed. Be sensitive to dynamic referenced elements. Here all elements that are dynamic referenced or which are independent applications should be subsumed (all `set` methods of all `common` classes)

2.6.5. As a parameter for method the “smallest Interface” has to be asked for.

Example:

```
class XY extends ActionListener {
    ...
}

public void handleAction(XY listener) {
    listener.actionPerformed(xxxx);
}
```

should be replaced by

```
public void handleAction(ActionListener listener) {
    listener.actionPerformed(xxxx);
}
```

2.6.6. The following method calls should not be part of the code:

- `System.exit()`
- `System.out.println()`
- `Throwable`

- `Public static void main()`
- `System.setProperty()`, (except use in `LoginDialog`)
- `SwingUtilities.getRoot()`

2.6.7. `Synchronized` should not be used at `SingletonClass getInstance()` methods.

Example:

```
public class SingletonClass {
    private static SingletonClass instance;

    public SingletonClass getInstance() {
        synchronized(SingletonClass.class) {
            if (instance == null)
                instance = new SingletonClass();
            return instance();
        }
    }
}
```

It is better to use:

```
public class SingletonClass {
    private static SingletonClass instance = new SingletonClass();

    public SingletonClass getInstance() {
        return instance();
    }
}
```

Remark: If the constructor throws exceptions, the instantiation can be done at the static initializer.

2.6.8. Methods that contain more than 30 lines of code should be avoided.

2.6.9. Methods that are called only once should be avoided (cause: readability).

2.7 Arrays

2.7.1. Are there any indexing errors?

2.7.2. Can array indexes ever go out-of-bounds?

2.7.3. Use `Array` instead of `Vector` as return value. Advantage: You don't have to use casts. To convert a `Vector` an according `Array` use

```
XY[] array = (XY[]) aVectorOfXY.toArray(new XY[0])
```

Moreover you should substitute an `Enumeration` with an `Iterator`, because `Iterators` are more powerful (traversing to both directions, removing objects, etc.) and the standard since JDK 1.2.

2.8 Output Format

- 2.8.1. Are there any spelling or grammatical errors in displayed output?
- 2.8.2. Is the output formatted correctly in terms of line stepping and
- 2.8.3. spacing?

2.9 Exceptions

- 2.9.1. Only guarded exceptions should caught! (NullPointerException vs. Exception)!
- 2.9.2. Is the appropriate action taken for each catch block?
- 2.9.3. Exceptions should be replaced by their upper classes. They should be removed, if the upper class is caught.
- 2.9.4. Exceptions should not generated artificially.

Example:

```
String currentRoleId = authorization.getRoleId();

if (currentRoleId == null)
    throw new NullPointerException("CloseProcess.getAuthorization().getRoleId()
        returned null");

if (currentRoleId.startsWith("_")) currentRoleId = currentRoleId.substring
    (1);
```

2.10 Flow of control

- 2.10.1. In a switch statement, is any case not terminated by break or return?
- 2.10.2. Do all switch statements have a default branch?
- 2.10.3. Are all loops correctly formed, with the appropriate
- 2.10.4. initialization, increment and termination expressions?
- 2.10.5. Interrelated operations should executed in an **Action**.

2.11 Files and Streams

- 2.11.1. Are all Streams closed properly, even in the case of an error?
- 2.11.2. Are EOF conditions detected and handled correctly?
- 2.11.3. Temporary Files, especially for test cases, should be created by `File.createTempFile()`.