Coding Guidelines vor Java Projects (e.g. practicals, seminars, diploma theses, etc.)

Rene Mayrhofer, November 2008

Syntax / code style

- Sun Java Coding style should be used (http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html) can be verified with the „checkstyle“ tool: http://checkstyle.sourceforge.net/
- Package names should conform to domain names
  - either project (e.g. org.openuat.<subpackages>) or university (at.ac.univie.cs.<subpackages>)
- File headers
  - Every source code file **must** have a copyright header
  - Contains name of author(s), date when the file was first created and potentially date when it was last changed
  - Contains a license under which this code can be used, .e.g. LGPL >= 3, GPL >= 2, BSD, Artistic License, etc. or simply „public domain“
  - Example:

```
/* Copyright Rene Mayrhofer
 * File created 2008-01-31
 * File last modified 2008-08-15
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
```

Documentation

- An **overview document** should describe the overall structure
  - does not have to be a formal UML class diagram, for simple projects a README or INTERNALS text file is sufficient
- **Every public interface** needs to be extensively documented
  - how to use it
  - how not to use it
  - assumptions
  - requirements before calling something
  - return codes
  - exceptions
  - different modes of use (especially when multiple methods can be used for the same action)
- Internal interfaces should also be documented appropriately (does not need to be as extensive as public interfaces, but at least describe what each method is supposed to do)
- Javadoc comments for **all public and protected member** variables (what are they for, which code writes and which code reads them)
- Code comments wherever necessary
  - „tricky“ or „hard“ code parts
  - member variables: value ranges, error/invalid values, null/unset values, etc.
  - assumptions at a specific code locations (e.g. „when we get here, then X is

guarenteed to be set“)
    - reasons why some error can not occur and is thus not explicitly checked for (e.g. „the above method calls would have raised an exception if X was not valid“)
    - which errors can be reasonably expected
    - etc.
    - but **not** comments on something that is obvious to anybody who can read English and understand Java syntax (e.g. „/* initialize x with 0 */ int x=0;“) - this only clutters the code and helps nobody!
    - Try to put yourself into another programmer's mind:
        - Would anybody else understand the structure?
        - Would anybody else understand this particular piece of code?
        - Would everybody understand how to use my code (and not caring what happens inside)?
        - Would anybody else be able to extend this code when assumptions change or new features are required?
        - And most importantly: Would anybody understand **why it has been done that way** when they use or extend my code?

Secure and defensive programming

- **Error checks, error checks, error checks**
  if (x == null || y < 0 || y >= 20) throw MyParameterException(“x or y is invalid”);
    - This way, the program still fails, but at least you know exactly where and why!
- Sanity checks, sanity checks, sanity checks
  if (z == null || z.foo < 0) {
  logger.warn(“Ouch, z is invalid. Recovering from the situation, but this really shouldn't happen.”);
  z = new MyWhateverObject(defaultValues);
  }
    - This way, you can spot problems early on.
    - **Don't ignore it when sanity checks fail** – either the sanity check is wrong and should be corrected, or the code is!
- We care first and foremost about working, secure, maintainable, and robust code. Only when it has all of these properties you may start optimizing for performance (on the code level – taking care of performance on the architecture/design level is something else). There are too many fragile and insecure programs out there, we don't want to create any more of them!
- **Use timeouts** instead of blocking indefinitely whenever waiting for some external event (e.g. reading from network, waiting for another thread to finish, etc.). When a timeout is hit, log a warning message when the program can gracefully recover from it (e.g. by telling the user about the timeout and letting them start the same action again) or an error when recovery is not possible.
  The **SafetyBeltTimer** class helps with this.
- Follow secure coding guidelines: http://java.sun.com/security/seccodeguide.html

Testing

- Use the **test first paradigm**!
    - define interfaces
    - write JUnit test cases before the implementation → they should all fail when first run

        - Correct unit test should succeed only when the behaviour is correct.
        - Need tests for both the “normal” **and for error cases** → test that the code correctly identifies and handles errors (e.g. by throwing the expected exception or returning the expected error code / null value)
        - Attention: assert statements in code that is never run (e.g. in callback code that is never called...) will not fail even when the code doesn't work!
    - finally write implementations until unit tests succeed
    - use unit testing coverage reports to find out which code is not yet tested
- **Don't write main routines for testing!** Write JUnit tests again and check for the expected outcome using Assert statements. Instead of manually checking the output of a main routine (which is also run manually), this makes sure that errors are not missed and, more importantly, those tests can be run again and again whenever code changes!
- Use tools for profiling
- Use tools for finding potential bugs (recommended to use all three!)
    - http://update.enerjy.com/eclipse
    - http://pmd.sourceforge.net/
    - but at least use findbugs if you only want to use one: http://findbugs.sourceforge.net/

and turn off those warnings you have verified are OK (but then comment why it is ok in this case!!)

```
@edu.umd.cs.findbugs.annotations.SuppressWarnings(value="NM_METHOD_NAMING_CONVENTION")
@SuppressWarnings("PMD.MethodNamingConventions")
```

Logging

- Instead of System.out.println, use a proper, run-time configurable logging package such as http://logging.apache.org/log4j/
- use debug/verbosity levels consistently, e.g.:
    - ERROR for something that administrators/users should take notice of during production use
    - WARN for something that the software can recover from (e.g. an error that has been automatically corrected by rollback) but that should not ideally happen,
    - INFO for something administrators/users may want to see when looking for problems (e.g. network transmission errors)
    - DEBUG for output that you would print using System.out.println during debugging (e.g. before and after a critical method call, values of parameters and variables, etc.)
    - TRACE for output that you would print using System.out.println during debugging but that is really verbose (e.g. from within loops)
- don't remove debug logging when debugging is done, but lower the logging level for this particular class (log4j.properties)
- For all logging lines in TRACE to INFO levels that print more than a static string (e.g. including variable values or even calling print methods) and thus consume processing power, but them into

```
if (logger.isInfoEnabled()) logger.info(“....” + x + “....” + y + “....”);
```

```
if (logger.isDebugEnabled()) logger.debug(“....” + x + “....” + y + “....”);
```

blocks. This way, the string objects only get created when they are going to be logged.
Note: **Really do this while writing the code!** More than once, performance problems were caused by logging that never made it to the log (e.g. DEBUG level) but that got evaluated in the code!

Source tree organization

- put source code under src/ (and then the packages structure)
- put documentation under doc/
- put compiled binaries under bin/ (e.g. generated by Eclipse or by a build script)
- put created JAR files, installation files, source or Javadoc ZIP files etc. under dist/
- use a central ant build script to facilitate the whole build process
- there should be at least the “compile”, “junit”, and “dist” targets for ant; recommended are also the “check” and “run” targets
- Calling “ant” without any other parameter within the main project tree should build everything correctly and run all JUnit tests. All other required steps (e.g. required build environment in addition to a standard JDK and ant installation) should be documented in a README file.

And yes, following these guidelines is going to be evaluated – unless there is a very good reason for ignoring any of the rules (and this reason is documented!), it will influence final grades.