Working document for the

Unofficial Extended Java Coding Guidelines

Whitepapers.dk

1. Table of Contents

1. Table of Contents 2
2. Document history 4
   2.1. Revision History 4
3. Purpose and intended audience 5
4. Naming standards 6
   4.1. Naming identifiers 6
   4.2. Naming of common methods 6
   4.3. When to use prefix methods by get..() or find…() 7
   4.4. Interface naming convention 7
   4.5. Naming interface implementations 7
   4.6. Naming EJB-components 7
   4.7. Value Objects (Transfer Objects / J2EE design pattern) 8
   4.8. Naming DAO classes (for EJBQL / direct SQL database access) .. 8
   4.9. DAO superstructure 8
5. Constants 10
6. Transaction control 11
   6.1. When to use 11
   6.2. Setting the transaction timeout and number of application server execution threads 11
7. Making use of code style templates 12
8. Choosing package names 13
9. Javadoc 14
   9.1. Active use of method deprecation 14
   9.2. Document short and adequitely 14
   9.3. Document once 15
   9.4. Document API levels 15
   9.5. Document homogeneous 16
10. Error handling 17
   10.1. Differentiate between business logic (data) exception and system exception (ressources, database, middleware, communication,..) 18
   10.2. Checked exceptions 18
   10.3. Unchecked exceptions 20
   10.4. Error codes and ResourceBundles 21
   10.5. Logging errors 21
   10.6. Nested exceptions 21
11. Internationalization 22
12. Logging 23

12.1. Logging strategy: defensive/offensive/levels 23
12.2. Commons and *Log4J* logging setup 23
12.3. Levels 24
12.4. Where to log 24
13. Versioning of source code 26
14. Validation 27
14.1. Data input validation at the front-end (Applet/Webstart client) 27
14.2. Server-side business logic validations (middle tier) 28
15. Test strategy 29
15.1. Naming convention 29
15.2. How to setup a test case 29
15.3. JUnit test of API 31
15.4. Server test 32
16. Antipatterns 33
16.1. Don’t use design patters without justified cause 33
16.2. Don’t introduce additional abstraction unless functionality added 33
16.3. Declare class instance variables as private 33
16.4. Don’t use classes *Vector* and *Enumerator* 33
16.5. Use static initializers Initialization of instance variables 33
16.6. Declare variables as close as possible to where they are used 33
16.7. Responsibilities for the programmer 34
16.8. Don’t use deprecated methods 34

2. Document history

2.1. Revision History

Date	Version	Author	Change description/reason
2006-11-23	1.0	www.seniorkonsulent.dk	Initial document
2007-07-09	1.1	www.seniorkonsulent.dk	Minor updates

3. Purpose and intended audience

Sun Java source formatting guidelines are often to vague and not suitable for a specific project. Consequently the need for more strict guidelines may arise. This document intends to describe source formatting standards for for practical Java/J2EE programming. Intension is to ensure, that developers in a project have guidelines to follow, which obeys common rules while coding.

This document is a working document to be adapted to the special requirement for a specific project. It is not intended to preach tamper-proof truth. It presents a structured attempt to provide a solid foundation for debate. Common-knowledge regarding Java syntax dictated by SUN Microsystems should not be found here. Such conventional rules should be known by the developers in the project by default.

The document may be used as basis for:

- specification basics for outsourced programming projects
- unifying team efforts
- shaking up a project starting to head the wrong way

Following subjects will be adressed:

- project-specific standards
- package name
- *javadoc* documentation
- error handling in general
- exception handling
- data validation
- logging
- code test strategy (JUnit)

Considerations out-of-scope of this document are architectural matters like the choice of design patterns on the EJB-tier, etc.

Developers, who need to be introduced to the basic syntax conventions for the Java language, should read Standard Conventions from SUN Microsystems: http://java.sun.com/docs/codeconv/

4. Naming standards

Currently present are the necessary standards for the various component types (interfaces, classes, variables).

4.1. Naming identifiers

General naming standards will not be addressed in this document. Special conventions applyying to internationalized projects:

- English literals are used.

- In general use long expressive names instead of short abbreviations. This does not apply to identifiers used to support the programming language power (like iterators, etc.) or the specific single-letter names i,j,k for indentifiers used in short loops, where the meaning is obvious.

- Whenever an identifier should contain business related values, a name should be selected from a standard terminology list, created for the purpose to assemble alle business relevant terminology.

- Allowed characters in identifiers are alfanumeric characters (a..z, A..Z, 0..9). Non-constant variable names should not use entirely capital letters.

- Identifiers for constants may use underscore (_) and can have their names entirely specified with capital letters. Variables should not use this symbol.

4.2. Naming of common methods

It is a known fact that a number of method names are used repeatedly. Most known are *.get* and *.set* methods for exchanging values between objects. But a number of other method prototype definitions are commenly used:

- Finder methods: *findBy…()* or *findAll(): Collection*
- Create methods: *add…(): void*
- Read methods: *get…()*
- Write methods: *set…(): void*
- Update methods: *update…(): void*
- Delete methods: *remove…(): void*
- Question methods: *is…():boolean*

Avoid plentitude of alternatives: “get” or “search” instead of “find” – “insert”, “push” or “put” instead of “add” - “delete” instead of “remove” – “find” could return an array of objects instead of a *Collection* etc. Naming for method *.findXXX* should use *XXX* to implicate what the *Collection* acturally contains. Whatever choice is made, it should be strategic and applied throughout the code base.

Part of the program documentation is appropriately chosen method names. Where good names are chosen, additional documentation are often not needed.

4.3. When to use prefix methods by get..() or find…()

In general methods should be prefixed get…() where:

- there is a corresponding set..() method
- the get- functions are faster and do less error-checking than the find- functions
- get methods returns a result within rather limited boundaries (fx. calculation result)

Prefixing methods by find..() applies more rarely and in typical situations:

- where an actual search algorithm applies
- the method applies logic to fx. validate arguments and may throw a number of "invalid-input" exceptions.

There are no strict guidelines to methods prefixing. But make sure that the naming conventions are applied in the same way, everywhere. For enty beans, findBy..() is often used to give a hint about, which arguments are essential for retrieving the result.

4.4. Interface naming convention

Interface classes have their names prefixed with 'I' in a similar way to C++ notation.

4.5. Naming interface implementations

Implementations of common interfaces should just have normal class names. It is often seen but not good programming recommended practice to use suffix *Impl* after the name of classes implementing an arbitrary interface.

4.6. Naming EJB-components

EJB-naming convention should follow rules:

- **Stateless session beans**:
  Must have suffix *SB*. In addition, the home-interface/implementation class must furthermore be suffixed by *Home/Bean*.

- **Statefull session beans**:
  Must have suffix *SSB* + *Home/Bean* for home-interface/implementation class.

- **Entity beans**:
  Entity beans with local/remote interfaces should be prefferred for optimal performance.

  Use suffixes local/localHome for local interface/local home-interface.

  Don't use special suffixes for remote interface except Home for the remote home-interface.

Primary key classes must be suffixed *PK*.

- **Message Driven beans**:
  Must be suffixed *MDBean*

Naming conventions are essential because ant buildfiles and possible *XDoclet* setup may depend on the use of these structured rules for compiling and building the project correctly.

4.7. Value Objects (Transfer Objects / J2EE design pattern)

This denotes the classes responsible for transferring data between different tiers – which contains no business logic, is called *Value Objects* (VO).

The value object itself should be read-only and immutable.

All value objects in the project must inherit from a single class *dk.companyname.app.common.ValueObject* and be suffixed *VO*.

A value object may contain additional value objects. If it contains multiple value objects of the same type, they are stored in a *List*. Here the class *dk.companyname.app.common.ValueObjectList* used. It may implement 4 methods returning *ArrayLists*: *presentList*, *addedList*, *removedList* and *originalList*. one that holds the present VO's, a second that holds all the VO's that was added to the first list, third that holds the VO's that was removed from the first list, and last a fourth list that holds the original list of VO's.

ValueObjectList may be used to provide some objects to the client, and the client can add and remove these items form the list. If you want to provide a list of object to the client, that the client can't change, i.e *add* or *remove* use an *Arraylist*. Only use *ValueObjectList* og *ArrayList* to store multiple objects in another object.

4.8. Naming DAO classes (for EJBQL / direct SQL database access)

The application will usually access the database through entity beans and only rarely through direct SQL.

Class name suffix *DAO* (acronym for Data Access Object) indicates class, which either access entity beans or perform direct SQL database access. Under no circumstances should the application access the database otherwise than through a DAO class.

4.9. DAO superstructure

Every DAO class should inherit from the class *dk.companyname.app.dao.AbstractDAO*, which contains all common standard functionality for execution SQL, closing result sets, etc.

AbstractDAO.java

General changes for handling the database through direct SQL will confine to this class alone.

5. Constants

For constants there are usual strategies:

- declaring them as final static variables in a specific set of classes. The variables are accessed either directly as public variables or through get..()-methods
- declaring in resource bundles and accessing runtime by reading property files on startup
- declaring in database along with stamdata and accessing runtime

If a database is already in use by the application, it does not make much sense to use resource bundles additionally. Then it is usually better to keep most things in the database. Still there may be constants, which is system-related more than business related. These constants may be declared in specific constant classes under all circumstances.

Hardcoding values is not an acceptable strategy and should not be allowed in the code as a general rule. Values should be set from appropriate initialization code.

Usually constants will be used across packages and tiers. Subsequently two general classes are used for constants

- *dk.companyname.common.app.APPConstants*: All constants used throughout the business logic code.

- *dk.companyname.common.app.DBConstants*: All constants used in direct database access (SQL). This includes all constants defining all tables & columns used for DAO-access.

6. Transaction control

6.1. When to use

Methods should only be declared part of a transaction if rollback of actions (updates in database) is actually required at some point in the process. Methods just retrieving and displaying data from the database should not be under transactional control.

Excessive use of transactions may cause a large number of ‘hanging’ threads in the application server, which may block for incoming requests.

6.2. Setting the transaction timeout and number of application server execution threads

These two setting should always be optimized together. A transaction timeout should be selected, which will:

- allow transactions to complete under most normal circumstances
- prevent the application server from ending up in a situation, where all executable threads are blocked, while waiting for excessive long transaction timeout

It is important to only apply transaction control for rather fast processes, which terminates in a limited amount time. Otherwise the only outcome is to increase the transaction timeout on the application server,

An application server thread count should be selected, which will:

- not introduce an excessive performance penalty due to the time overhead required for thread management (of threads which are mostly idle)
- always leave threads free for the server to process necessary requests

Optimizing the variables is very much a trial-and-error process. But the transactional flow should under all circumstances be designed to minimize transactional windows to support low transactional timeout values.

7. Making use of code style templates

For most IDE tools such as Eclipse, there are plugins available, which will format code according to a template. This will format the source code according to generic tab-size, bracket positioning, etc

Two conditions must be met for successful use of such:

- definition of which code template to use (or creation of such one)
- the programmer making actual usage of the template, while coding

A code style template should be applied *during* the project to enforce discipline on participants. Despite usage of a code style template, there are still many ways to diverge in coding style, which the template cannot correct. In the end, code style templates cannot change the actual code. A simple example is whether to use {} in conditional sentences, where not required:

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

Strictly the first set of {} are not required around a unary statement (such as setting variable=value).

8. Choosing package names

As a minimum, different package names should be applied for code which is intended to run on:

- client
- server
- exchanged by above two parts

In each case there should be a package hierachy separating the differently abstraction layers. A description should explain the program separation in packages, used throughout the project. Contents will be target for several changes/expansions during the development fase.

Fx.

- *dk.companyname.applicationname.client:* classes implementing the client

- *dk.companyname.applicationname.client.gui:* classes for client GUI

- *dk.companyname.applicationname.client.delegate:* client façade to midtier

- *dk.companyname.applicationname.webtier:* whatever runs in a servlet engine

- *dk.companyname.applicationname.midtier:* whatever requires a container

- *dk.companyname.applicationname.common:* classes used across tiers

9. Javadoc

Javadoc is the term for comment notation used to cocument java programs, which can be extracted by the standard tool javadoc.exe.

Javadoc serves purposes:

- programming logic documentation directly in source code
- api documentation by browsing extract by javadoc.exe

Javadoc documents closest to the code, but still we might expect some overlab between the javadoc and the textual class descriptions in the UML design case tool.

More information on javadoc is found at *http://java.sun.com/j2se/javadoc/*

9.1. Active use of method deprecation

Occasionally methods or entire classes becomes 'out-of-fashion' for fx. business logic reasons. Other methods or classes should be preferred. Then use the *Javadoc* tag

@deprecated

to signal deprecation of methods. And subsequently AVOID ANY FURTHER USE OF THOSE METHODS.

9.2. Document short and adequitely

Javadoc should be written for:

- Package (one file with the given name "package.html" for each package)
- Class
- Attribute (when the attribute semantics are complex)
- Method
- Contructor

Required - and sufficient – javadoc tags:

- @param
- @return
- @exception

- @see

Other tags are really not necessary, assuming *javadoc* should mainly be an aid for the programmer.

Don't comment on set-/get-methods, which only performs standardized action of assigning/returning values. Instead choose intuitive names for the methods

Antipattern for *Javadoc* to be avoided:

```
/**
 * Method setting the value for myAttrib
 * @param myAttrib String assigned to attribute myAttrib
 */
private void setMyAttrib1(String myAttrib) {
  this.myAttrib = myAttrib;
}
```

9.3. Document once

Don't cut'n paste comments. Find the most appropriate place for the comments and only place the comment there. Otherwise it will not be possible to update the many identical blocks of documentation various places. Additional detailed documentation in the interface implementation methods should not repete comments found in the interface, but instead supplement specificly implementation technical details, etc. This in turn implies that different implementation classes of the same interface should have perceptible differences in the *Javadoc*.

Documentation of interface methods should not be transferred to the matching implementation. Instead, a @see-tag in the implementation classes should reference the interface methods, thereby providing easy navigation to the documentation.

Where implementation documentation appear unnecessary, use the @see-tag. Note that several IDEs like Jbuilder may report warnings (for instance for the lack of parameter tags) regarding the javadoc. This should not be considered a critical problem and can be ignored.

9.4. Document API levels

- *Classes*
  Justify the class - explain responsabilities – specify technical functional role – avoid repeating UML design case tool information

- *Methods and constructors*:
  Document only *public* methods/constructors. Expectations to input and output for API. Techical functional role –preconditions for calling – which arguments may be null or not - explain set of values returned – using tags @param and @return. For returned Collection, Iterator etc. it is especially important to explain, what classes are actually wrapped inside.

Explicitly state where input-object may be changed by a method – but in most cases the method should prefereably create own copies of data – keeping changes to a minimum. Explain thoroughly the reasons for exceptions through the @exception tag.

9.5. Document homogeneous

Documentation style should be as homogeneous as possible, using the same business terms and to the same degree of detail. This can can be problematic using *Javadoc* from several programmers gathered in the same document. For *Javadoc* the guidelines should be observed:

- inspect *Javadoc* conflicts in a widely used IDE's such as Eclipse.
- where appropriate, use html-codes, italic <i></i>, etc. for accentuation.
- descriptions at all levels (class, method, …) are only included in the *Javadoc* extract until the first blank line after the first sentence.
- specify datatype (simple or name of class/interface) on *@param, @return* or *@exception* before the actual prose description. Add one (1) blank between the variable name and the type for *@param*

10. Error handling

Define a hierarchy of exceptions and document use on each system layers.

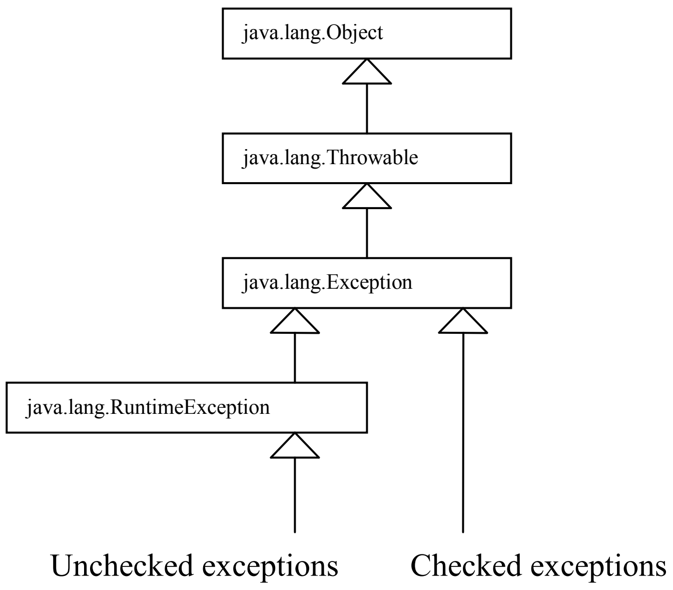
It is important with a clear strategy for handling and propagation of exceptions thrown from the various tiers to any boundary (typically the GUI) in the application. This must be settled from the very beginning of the programming. Good practice is to propagate errors using many business-logic specific exceptions rather than a few generic exceptions containing plenty, different hardcoded messages.

Don't apply empty 'catch' blocks: *try {} catch(Throwable t) {}*
Don't use *System.out.println* printout directly to console. Use generic logging.

Try to report the contents of nearby important variables, for which tings might have gone wrong. Most often, just knowledge that an error situation occurred is insufficient – it is also necessary to provide a possiblity to find the reason. In troubleshooting situations it's necessary to be able to follow the program flow. Exception handling should differentiate according to the following exception categories:

1. *Checked exceptions*, i.e. exceptions of type Exception (except RuntimeException) or user defined (business) exceptions (typically raised by validation errors, etc.).

2. *Unchecked exceptions*, i.e. exceptions of type RuntimeException (NullPointerException, IndexOutOfBoundsException, etc.) or Error (like VirtualMaschineError).



Antipattern of an often seen bad way of handling exceptions:

```
try {
  ... //something potentially throwing an exception
} Catch(Exception e) {    //not recommended practice
  System.out.println("Unexpected Exception " +
  e.getMessage());
}
```

10.1. Differentiate between business logic (data) exception and system exception (ressources, database, middleware, communication,..)

User defined exceptions must inherit from one of abstract classes:

- *dkcompanyname.applicationname.common.exceptions.BusinessException*
- *dkcompanyname.applicationname.common.exceptions.SystemException.*

The *BusinessException* may be extended, when it is an exception that is raised because of some business rules (fx. data validation/business rule violation) – and the *SystemException* may be extended when some kind of system error is raised (fx. *SQLException or IllegalAccessException*).

10.2. Checked exceptions

Checked exceptions may handled differently:

- *propogated unchanged through layers:* thrown exceptions are propagated untouched through the method call tree. Most appropriate where an exception cannot be handled properly where it occurs. Fx. in a constructor method – or generally in in the EJB-tier, where an exception need to be propagated to the GUI, which in turn handles the received exception, by issuing an error message to the user.

- *caught and wrapped by a narrow family of exceptions:* thrown exceptions are caught and situation-describing text information is wrapped within another re-thrown type of exception . Often the re-thrown exceptions are user defined business exceptions (fx. XxxProcessflowException). The process of catching, wrapping and re-throwing exceptions may happen at multiple levels. Difference from before is that categorisation of exception is required in order to re-throw the righ kind of business exception, which the upper receiving layers will dispatch upon (e.g. GUI).

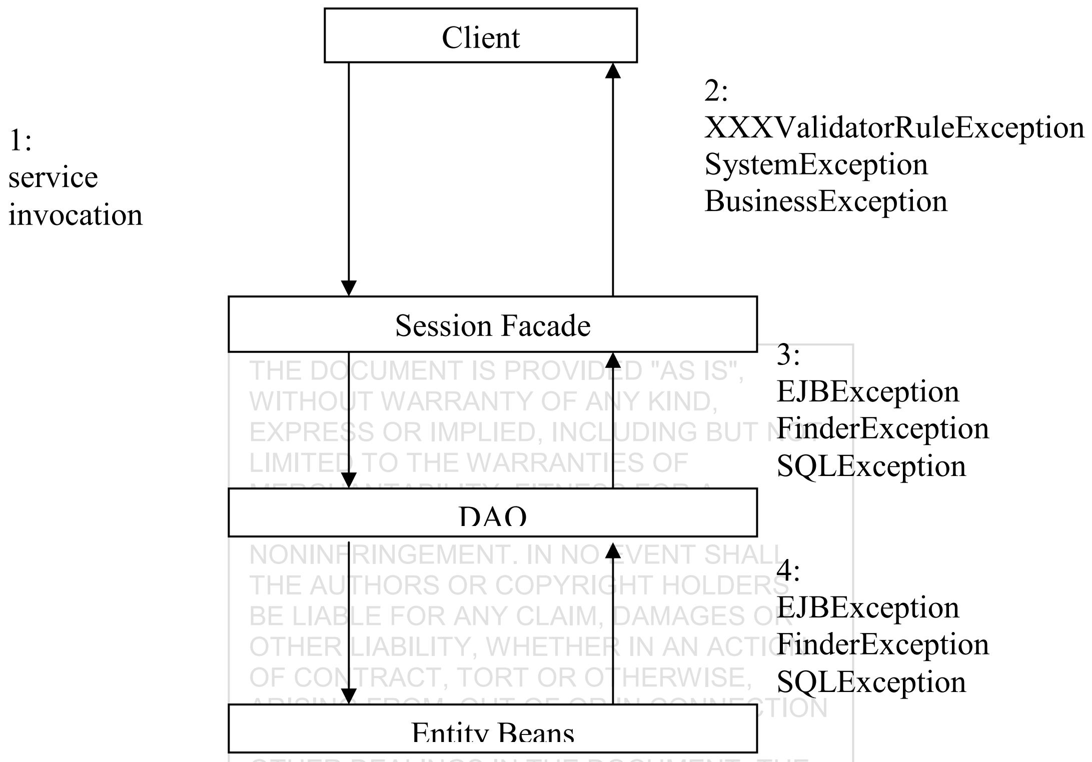

Client
1: service invocation
2: XXXValidatorRuleException SystemException BusinessException
Session Facade
3: EJBException FinderException SQLException
DAO
4: EJBException FinderException SQLException
Entity Beans

Figure above illustrates:

1. client invokes a service method (presumably on a webservice/stateless session bean) on the middle-tier. On success, the method provides expected return value/object.

2. On failure, the session bean method throws an exception, which need to be propagated to the client for appropriate exception handling. The listed exception types are:

dk. companyname.applicationname.common.exceptions.XXXValidatorRuleException

- an example of a user defined exception inheriting from *BusinessException* generated during business rule validation. Usually this exception will be thrown unchanged to the client.

dk. companyname.common.exceptions.SystemException

- thrown to the client in case of unexpected technical errors occur during the processing of the request. Such non-businesss-logic errors are often refferred to as system exceptions (fx. SQLException, IOException), etc. Since the client is unable to anticipate the often numerous resons for these errors, the expteions are handled equally and just reported to the end user as a "Technical error… (name of exception)… Report problem to system administrator and try again later..". Such exception and extended classes account for “unforseeable” situations with technical errors. Fx may an argument (null) in a call to a session bean method not comply with acceptable set-of-values (non-null). In that case the datacase constraints will cause a SQL exception which may subsequently be caught and re-wrapped into an appropriate SystemException.

Example of a method in the client, which performs a block of code potentially causing various exceptions, which the client has to react to in different ways:

```
public void actionMethodXXX() {
 try {
   ... //action causing exception of type ExceptionA,SystemException + misc. others
   ...
   ... //other action, causing exception of type ExceptionB, SystemException + others
 } catch(ExceptionA ea) {  //Generate a warning message and…
    ...
 } catch(SystemExceptionB eb) { //Generate error message “Contact system admin”..
    ...
 } finally {
    ...
 }
}
```

or alternatively:

```
public void actionMethodXXX() throws ExceptionA, SystemExceptionB {
   ... //action causing exception of type ExceptionA,SystemException + misc. others
   ...
   ... //other action, causing exception of type ExceptionB, SystemException + others
}
```

In the first case, exceptions are handled – in the latter, the calling method is forced to handle the exceptions by declaring them in the throws clause.

Fx. a checked exception would be:

java.lang.Object

 └ java.lang.Throwable

  └ java.lang.Exception

   └ java.io.IOException ← checked!!!

10.3.Unchecked exceptions

This type of exceptions should be specificly handled (normally not required by the Java compiler). Exceptions of this type are normally caused by programmatic logic errors – and are not accceptable in a production environment. It makes no sense to try handling such unforseen and abnormal program flow (*NullPointerException*, *ArrayIndexOutOfBoundsException*, etc.). However in the session beans that the client may call, all runtimeexception will be causght and and a new SystemException will be thrown instead since the client obviously will be unable to handle fx. a nullpointerException raised on the server.

Fx. an unchecked exception would be:

java.lang.Object

└ java.lang.Throwable

  └ java.lang.Exception

    └ java.lang.RuntimeException ← anything extending this is unchecked

      └ java.lang.NullPointerException

10.4.Error codes and ResourceBundles

Don't hardcode messages. Get them from ressourcebundles. Or at a very minimum keep them in a centralized class, so they may later be more easily moved to a resource bundle.

10.5.Logging errors

When an unhandled exception reach the executing JVM thread, a stacktrace will be produced by default . A stacktrace may also be produced at any time using the *.printStackTrace()* method on the java.lang.Throwable interface. This may be done while catching system exceptions – before throwing an application specific *SystemException*. See Logging for further datails.

10.6.Nested exceptions

Situations occur, where a thrown exception has to be converted to an exception on a higher conceptual level, before being re-thrown to the invoking method. Described in section for Checked exceptions . For all kinds of business exceptions the initially thrown exception itself should not be embedded into the newly instantiated business exception (actually the initial exception could be hidden in an attribute in the exception, which is being re-thrown). However the text message from original exception are often used for the new exception. Only deviation from this strategy is *SystemException*, which is being instantiated with the original exception (see Where to log).

11. Internationalization

Internationalization is an area, where a decision should be made before starting the project. Otherwise it can be rather resource demanding to introduce internationalization later.

12. Logging

Logging may be applied for a number of reasons: runtime error, debug or for history tracing needs.

- Error logging is used to record information about unexpected occurrences.

- Debug logging is used to record program or data flow information. Usually applied in development and troubleshooting situations

- History logging is normally an integrated part of the application to gather information about access to restricted resources or a traceable background for the system state (transaction milestones).

In the first two situations, application logging may be done using *CommonsLogging* or *LOG4J* (open source from the Jakarta-project). LOG4J may route logging trace to output a target defined for the logger (console, file, database). Debug-tracing directly to the console through *System.out* should be avoided. Deciding for a logging strategy is essential for optimal benefit from logging. Error debug and (especially) historic debug tracing should normally not have configurability to be switched off. Debug logging should have that ability.

12.1.Logging strategy: defensive/offensive/levels

Defensive strategy (short and adequate output) is suitable for error logging. Only results closely related to the error situation is recorded, because error situations may occur in a huge number of variants. Accounting detailed for all will require too many resources.

Offensive techniques (lots of debug output) are suitable debug logging, where it is essential to validate functionality for a limited program segment and, where each little piece of information may end up being significant.

History logging is implemented according to use case requirements using program API.

Logging should be applied on each application level. This means independent debug setup for each application. The server has its own serverlogs to record occurrences.

12.2.Commons and *Log4J* logging setup

The jar-file *commons-logging.jar* is used for commons logging.. For *Log4J* the jar-file *log4j-1.2.8.jar* (or whatever version) is used. Either file may be packed into the root of the application *.ear* file, and should be mentioned in the ejb-manifest.mf file that will be used while making the jar file that contains the beans, this file is in vss under config/plain.

A property file commons-logging.properties is needed to for comons logging and log4j.properties for *Log4J* logging. When a property file is packed into a jar file containing a bean, it will be available from its class path. A property file placed in the domain directory of the weblogic server, i.e weblogic/user_projects/ProofOfConceptDomain may have effect for all applications throughout the server.

Classes performing logging information may look like:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Foo { // class attribute defines logger for entire class:
...
private static final Log log = LogFactory.getLog(<ClassName>.class);
...
//actual logging by invocation of a method, which decides the level of logging:
log.debug("Text-to-be-logged");
```

Log4J may equally well be initialized programmaticly. This is described in documentation.

General information regarding CommonsLogging is found at
http://jakarta.apache.org/commons/logging.html
General information regarding *Log4J* is found at
http://jakarta.apache.org/log4j/docs/index.html

12.3.Levels

CommonsLogging offers 5 levels of logging:

- *info*: Normally there is no differentiation in the use of the levels *info* or *debug*. So just apply “debug” as below.
- *debug*: Used for normal debug.
- *warning*: Record indications of abnormal program flow, which should not prevent the application from terminating correctly.
- *error*: Situations causing instantiation of *SystemException* should be prefixed by logging at this level, see Where to log). The programmer will normally not actively use this logging level, which is normally only activated after error checks.
- *fatal*: Is NOT used

Standard debug syntax rules dictates that debug is configured so logging occurrences usually contains a timestamp, system name (application name and code location) and descriptive text explaining the situation.

12.4.Where to log

Error log-statement should be placed at lowest level in the program, where an exception is being caught in a catch-statement (and most detailed information available about the error state). Logging is performed whenever a system runtime exception is caught and transformed to (re-thrown as) a *SystemException*. The 3rd argument to the *SystemException* constructor is the original thrown exception, which is used in the automatic error logging in order to secure, that the logging can signal exactly where the error appeared.

13. Versioning of source code

Version in may be put into the source code, so it comes out in the compiled binary.

All Java source files should contain following lines right after the class definition:

```
...
public static final String AUTHOR = "Author: ";
public static final String REVISION = "Revision: ";
...
```

This is useful, where confirmation of version of source is needed. Inspection could be done using a Java Decompiler (fx. JAD) or directly by inspection of hex codes for compiled code.

Manipulation of the REVISION string contents may be manipulated when retrieved from CVS. Using appropriate notation, the string may be populated during the CVS transfer process to reflect the current versioning level.

14. Validation

Validations may be performed on a number of different levels. Validation falls in different categories:

- input validation (client)
- business logic rule validation (server)
- data integrity validation (database)

Input validations are often done on-the-fly during the typing in the client dialogs. Access to database for value confirmation is mostly NOT required.

Business logic rule validation are performed AFTER invoking an action with fx. an Ok/Save/Cancel/etc. button (default, usual business rules validation) and DOES usually require access to database for calculation of complex values or compare.

Data integrity validation are usually done by introducing a number of constraints in the database (non-null, uniqueness, referential constraints,..). But even though the database enforces integrity validations, appropriate validation checks should be done in the application code to avoid system SQL insert exceptions.

As a rule of thumb, the programmer should aim at invoking middle-tier as few times as possible.

14.1. Data input validation at the front-end (Applet/Webstart client)

Generally:

- client should perform as much syntax validation as possible

- input validation is done on "document-level", i.e. data is only being validated, when the user signals end of input submission by applying Ok/Save/Cancel/etc.

Only if stated in use case should immediate validation be applied, for instance by performing some cross-checking right after the user tabs change focus between input fields.

Validation may be simplified or become unnecessary with use of appropriate GUI objects, which puts restrictions on input (radio-button choice selector, drop-down lists/combo-boxes with pre-populated values.

Values required from the server while validation should be obtained through a unit operation if possible to minimize communication with the server. The data should be available when opening the user dialog.

14.2. Server-side business logic validations (middle tier)

Do not duplicate validations performed on the front-end.

Also the business logic validation should be centralized. They should be performed at a programmatic level, where it is possible to retrieve values from the database. Checks should not be scattered throughout all layers or pieces of source code, so it becomes unclear, what validations are actually performed.

15. Test strategy

When changes are introduced into a system, it should be possible to tell if the system continues to work without unexpected side effects. Unit tests provides facility to accomplish this objective. There are basic types of unit tests:

- server-independent (stub) tests (API test, does not require a running server)
- server tests (unit tests, which requires server running and access to database)

Both types of tests may be implemented with the same testing framework (fx. *JUnit*).

JUnit is a framework, where tests are incorporates into the code at the same time as functional logic is implemented. In *JUnit*, each *testcase* is represented by a class with a number of methods where the declarative name starts with *"test…"*, A *testcase* is performed by executing all methods with such name signature. When one of the method in the business class encounters an error, this may be reported by *JUnit* during the test run. *JUnit* allows functional tests of al parts of the application at all times.

Running unit test should be done each time *before* source code is committed to the versioning system. Server-independent unit tests may be run in extension of the compile process.

Next time someone asks you for help debugging, help them write a test…

15.1.Naming convention

Test class names should have as prefix the same name as the class they test. But test classes should also have "Test" as suffix. Fx. for a class named *FooClass* the appropriate naming for a test class should be *FooClassTest*.

Test class package name should
- more or less mirror the package for class to test
- contain the section ´*.test.*´ somewhere in the package name
- reflect, if the unit test is a test of the client or server functionality
- separate server-independent tests and server tests.

15.2.How to setup a test case

The full standard documentation for *JUnit* is available at
http://www.junit.org/index.htm
The fastest way to be introduced to *JUnit* is by use/writing examples. Assume that a test case has to be written for the class

dk.companyname.app.midtier.FooClass:
package dk.companyname.app.midtier;

public class FooClass {

```
 ...
 public int calculateSomething(int arg1, String arg2) {
      ...
   return result;
 }
 ...
}
```

To test the method *calculateSomething*, a test class *FooClassTest* is written. It must inherit from *junit.framework.TestCase*:

```
package dk.companyname.app.midter;
import junit.framework.*;

public class FooClassTest extends TestCase {
 protected void setUp() {
   //I assertions/assignments may be applied here if any state
   //is reqired before subsequent test methods are run
 }

 protected void tearDown() {
   //Applied after test case termination. Disables resources (objects), from
   //the 'setup' method. Fx. Database connection
 }
 ...
 //All methods with naming prefix "test" are automatically run by the JUnit
 //framework. Methods should have NO arguments or return value (void)
 public void testFooClass1() {
   FooClass fc = new FooClass();
   Double result = fc.calculateSomething(5,"DATA");
   assertEquals(25, result);
 }
 public void testFooClass2() {
   FooClass fc = new FooClass();
   Double result = fc.calculateSomething(16," MORE_DATA");
   assertEquals(4, result);
 }
 ...
}
```

All test classes in a package should be referenced from one test class inheriting from To every test class in a package a gathering test class named *AllTests* is written:

```
package dk.companyname.app.midtier;

import junit.framework.*;
public class TestAll {

public static Test suite()
 TestSuite suite = new TestSuite();
 suite.addTestSuite(dk.companyname.app.midtier.test.FooClassTest.class);
 suite.addTestSuite(dk. companyname.app.midtier.test.AnotherClassTest.class);
 ...
 return suite;
}

public static void main(String[] args) {
  junit.textui.TestRunner.run(suite());
}
```

}

To run test from command-line

java TestAll

To run test case using textual interface:

java junit.textui.TestRunner FooClassTest

To run test case using JUnit graphical interface (showing a swing window with green progress bar if all test passedand red progress bar if any failed):

java junit.swingui.TestRunner FooClassTest

The test suite may be run accordingly:

java junit.swingui.TestRunner TestAll

The sequence of events starts by running the main method in the *TestAll* class. Notice that the static main method is implicitly implemented in the superclass *TestCase*. *JUnit* will automatically run method *.setUp* in *FooClassTest* and subsequently run all void argument methods in this class, which have method names prefixed by *test*. Methods are run in no guarantied order. Meanwhile the results generated by all *assert*-methods are being displayed. Finally after having executed alt test methods in *FooClassTest*, the method teardown is being run.

15.3. JUnit test of API

APIs on each logical level should be unit tested. This includes:

- Business delegate API in GUI
- Session bean API
- Entity bean API
- DAO API
- Validator classes
- Security system API
- Central utility classes (performing business logic calculations)

According to each Java package in your application, define a corresponding *TestSuite* class that contains all the tests for validating the code in the package.

Define similar *TestSuite* classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application.

Ensure your build process includes compilation of all tests. This should ensure your tests are always up-to-date with the latest code and keeps the tests fresh.

Test cases should not cover trivial *set/get*-methods or initialization of classes with a no-function void constructor. There are no finite guidelines for writing adequate unit testing. But as a general rule is that everything, which:

- looks like an API
- is reused multiple places

is a possible candidate for unit testing. The more a class is reused, the more detailed the test should be.

(Server) tests may be controlled by a small jsp web-application, where a button may be used to trigger all the test cases. See Server test for further description.

15.4.Server test

Server tests are unit tests, which require a running server container.

Server testing may depend on a specific setup of data in the database. The circumstances for setting up test data should be well documented beyond just a SQL script.

16. Antipatterns

Design patterns describe correct strategy for solving a problem. Antipatterns describes on the other hand an often used (but incorrect) strategy for solving a problem.

16.1.Don’t use design patters without justified cause

Design patterns are intended to solve some common, general problems. Unless these problems are present, using design patters will just bring anything positive. Most design patterns are used without making a specific point out of their use. When the need for a design pattern comes natural, there is not reason to explain, why and where it is used.

16.2.Don’t introduce additional abstraction unless functionality added

Avoid encapsulate classes, which foreign package names into a self-written class. Only do so if additional functionality is introduced somehow. Introducing a new abstraction layer also introduces increases the risk that others will misuse of fail to use the introduced API. Better to keep old API and make sure all programmers use the same standards.

16.3.Declare class instance variables as private

When a variable is private, declare it private accordingly! Most instance variables are not supposed to be accessed by other classes and should subsequently be declared private. Accessing through set/get methods are standard programming practice..

16.4. Don’t use classes *Vector* and *Enumerator*

Classes *Vector* and *Enumerator* should never be used in the project. Always substitute with classes *ArrayList* and *Iterator*.

16.5. Use static initializers Initialization of instance variables

Initialization of instance variable can be done directly when they are declared in the class header. They could equally well be initialized in class constructor, servlet *init()* method, enterprise bean *ejbCreate()* method, etc. but then it is more difficult to get a quick overview of which values the variable may actually contain.

Initialization should only be performed in the constructor, if the initial value of the variable depends on the context in which the class instantiation is being peformed. Only initialize once.

16.6. Declare variables as close as possible to where they are used

Don’t declare one variable high above and use it in numerous contexts. Declare where before use and try to restrict use to that place.

16.7. Responsibilities for the programmer

- When writing EJBQL/SQL, ensure that correct database constraints are in place. If expecting one result and there is no uniqueness constraint, multiple results could be returned and cause exceptions. Also ensure correct indexes are available to allow sufficient performance. Indexes may be verified with the database facility SHOW PLAN/EXPLAIN or through 'Toad' similar.
- Help the garbage collector by setting unused data structures to null. The garbage collector should be able anyway to do necessary clean-up but helping is common programming practice.
- Only do imports of classes actually used. There will be IDE facilities to note where this is not the case.
- Work *with* the IDE and not *against* it. Take the time necessary to get familiar with a new IDE.

16.8. Don't use deprecated methods

The use of *deprecated* methods should not be allowed. Subsequently it is not advisable to disable check for deprecated methods in the *Compiler options* settings for the IDE in use.