

Intelligent Transportation Systems
700 Kipling Street, Suite 2500
Lakewood, Colorado 80215
Phone (303) 512-5834
FAX (303) 239-0848



CTMS/CTIS INTEGRATION **Contract Routing No. 04 HAA 00063**

CTMS Coding Standards

Date: 11-May-04

Version 0.1

This document describes the coding and development standards that apply to the ITS CTMS project.

Approved By

Robert Wycoff
CDOT ITS Office

Signature: _____

Date: _____

John Williams
CDOT ITS Office

Signature: _____

Date: _____

Prepared By:



CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

Revision History

Date	Version	Description	Author
20-Apr-04	0.1	Draft version	Jason Westra Trey Grubbs
07-May-04	0.2	Revisions	Jason Westra Trey Grubbs
11-May-04	0.3	Formatting changes, table of contents etc	Sachin Saindane
20-May-04	0.4	Added Client Standards for resource bundles	Trey Grubbs Sachin Saindane

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

Table of Contents

1.	Introduction	4
1.1	Purpose	4
1.2	Scope	4
1.3	Definitions, Acronyms, and Abbreviations	4
2.	Standards	5
2.1	Documenting Method Headers	5
2.2	Getters for Constants	5
2.3	Accessors for Collections	5
2.4	Standards for Parameters (Arguments) to Member Functions	5
2.5	Ordering Member Functions	5
2.6	Standards for Compilation Units	6
2.7	Reuse	6
2.8	Use Wildcards on Imports	6
3.	Additional CTMS Project Coding Standards	6
3.1	Common Standards	6
3.2	Accessing Remote Services	7
3.3	Use of Singleton Pattern, Caching	8
3.4	Error Handling	8
3.5	Logging	8
4.	Client Coding Standards	10
4.1	Database Access	10
4.2	UI text	10
4.3	Class Member Declarations	11
4.4	Panel Construction	11
4.5	UI Component Factory	11
4.6	Client Logging	11
5.	J2EE Standards	11
5.1	Database Access	12
5.2	Transaction Management	13
5.3	EJBs	13
5.4	Web Applications, JSP, Servlets, and Struts	14

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

1. Introduction

The following document describes the coding and development standards that apply to the ITS CTMS project. Unless otherwise noted, all standards in the AmbySoft, Inc. standards guide will apply. The guidelines listed here are alternatives and additions to the standards as detailed in the AmbySoft Java Coding Standards document (<http://www.ambysoft.com>).

Code examples are in Courier New 10 font. While examples of poor coding practices, that should not be followed are in Courier New 10 Red font.

The overrides to the appropriate sections in the AmbySoft Java Coding Standards document are as follows:

1.1 Purpose

The purpose of this document is to help developers identify and follow a standard coding practice. This will help minimize errors and improve performance.

1.2 Scope

The scope of this document is the CTMS/CTIS project.

1.3 Definitions, Acronyms, and Abbreviations

Please see the CTMS/CTIS Glossary.

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

2. Standards

The overrides to the appropriate sections in the AmbySoft Java Coding Standards document are as follows:

2.1 Documenting Method Headers

(Refer to AmbySoft guide Section 2.3.1)

1. Use Javadoc for protected and public methods since these may be called from outside the class itself either publicly or from a subclass. Adding Javadoc to private methods is optional and encouraged.
2. Document an override of an ancestor method in the method header
3. Document an interface implementation in the method header (e.g., list the interface where the method is located).
4. Minimum method Javadoc is:
 - a. @param – for each parameter
 - b. @exception – for each exception thrown
 - c. @return
5. Optional method header Javadoc comments:
 - a. @bug - for well known bugs in the method
 - b. @author – for original author, and each major contributor to the methods logic.
6. A standard comment template should be used in the IDE to default these values for the developers automatically when a new class or method is created.

2.2 Getters for Constants

(Refer to AmbySoft guide Section 3.4.2.2)

Getters for constants are not required for the project.

2.3 Accessors for Collections

(Refer to AmbySoft guide Section 3.4.2.3)

1. Adding to removing from lists – using addXXX when adding to the end of a collection.
2. Use removeXXX rather than deleteXXX when removing an object from a collection.

2.4 Standards for Parameters (Arguments) to Member Functions

(Refer to AmbySoft guide Section 5)

1. Ignore alternative 5.1.1
2. Ignore alternative 5.1.2
3. Alternative 5.1.3 will apply for accessors and mutators on dumb, data transfer classes such as value objects, JavaBeans, and state classes. Otherwise, naming the argument exactly what the instance variable is named causes a breakdown in encapsulation and a maintenance nightmare if the instance variable is ever renamed.

2.5 Ordering Member Functions

(Refer to AmbySoft guide Section 6.1.4.2)

1. There is no need to list methods in alphabetical order. Any modern IDE does this for you from the source tree browser.
2. Methods that are defined because of implementing an interface should be documented as such and placed at the beginning of other public methods for the class.
3. Group interface methods together when a class implements multiple interfaces and methods.

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

2.6 Standards for Compilation Units

(Refer to AmbySoft guide Section 6.4)

1. Never have multiple classes in a single .java source file unless they are inner classes.

2.7 Reuse

(Refer to AmbySoft guide Section 7.1)

1. The 100% Java standard for third party code libraries is not a necessity.

2.8 Use Wildcards on Imports

(Refer to AmbySoft guide Section 7.1)

1. Use wildcards for any number of imports over 5 classes. Some IDEs can recognize this automatically for you (e.g. Eclipse).

3. Additional CTMS Project Coding Standards

3.1 Common Standards

The following are additional coding standards that are followed on the CTMS project that are common across applications and tiers.

1. The following copyright should be placed at the top of all Java classes.

```

/*
 * Copyright 2004 CDOT-ITS, Inc. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * - Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * - Redistribution in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * Neither the name of CDOT-ITS, Inc. or the names of
 * contributors may be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * This software is provided "AS IS," without a warranty of any
 * kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND
 * WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY
 * EXCLUDED. CDOT-ITS AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR

```

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

* DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL CDOT-ITS
 * OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR
 * FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR
 * PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF
 * LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE,
 * EVEN IF CDOT-ITS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

*
 * You acknowledge that Software is not designed, licensed or intended
 * for use in the design, construction, operation or maintenance of
 * any nuclear facility.

*
 * <p>Title: CTMS</p>
 * @author YOUR_NAME
 * @version 1.0
 * Created on THE_DATE

*/

2. List instance variables (class attributes) at the top of the class.
3. Names of constants should include contextual information to prevent clashes. For example, MAXVAL could easily clash with someone else's constant of the same name. Add more words for context, e.g., PORT_NUMBER_MAXVAL would be a good name for a constant referring to the maximum value that can be assigned to a port on a machine.
4. Do not duplicate important information. Store it in a Constants class or in a properties file.
5. Do not hardcode data into classes. Use properties instead.
6. Do not code path information that is OS platform specific. Use a utility class that can easily map all path-related functionality for files.
7. A singleton (see Section 3) will be used to provide a single point of access to resources that are loaded in the application. To keep environment-specific paths out of the application, this singleton must only load the resources it references from the classpath using a mechanism similar to this:

```
InputStream is = this.getClass().getClassLoader().getResourceAsStream("config.properties");
```

```
Properties props = new Properties();
props.load(is);
```

3.2 Returning Collections and Maps

3.2.1 *Public methods returning collections or maps should return the interfaces, java.util.Collection or java.util.Map. When used internally (in a private method), it is valid to return concrete types such as java.util.ArrayList or java.util.HashMap.*

3.3 Accessing Remote Services

1. Remote services include anything looked up in JNDI such as:
 - a. EJB homes
 - b. JMS topics, queues, and connection factories
 - c. DataSources
2. All access to remote services (including EJB to EJB) will be through a ServiceLocator pattern. Base class methods that make accessing the ServiceLocator easier are fine and encouraged as long as the method delegates to the ServiceLocator.
3. The ServiceLocator is the only object in the system that maintains a connection to JNDI through the InitialContext.

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

4. Use of the empty constructor `InitialContext()` is recommended unless security credential information is required. Changes to JNDI lookup settings should be done in *jndi.properties* and this file should be included in each server deployment.
5. It is recommended that you use component-level environments (`java:comp/env/`) when accessing resources in JNDI from within a Web Container or EJB Container. There are different areas in JNDI where objects are located (e.g., global and component environment name spaces). Use of the global JNDI tree is not recommended, because it is not portable, it is not easy to change the JNDI names of components without modifying source code, and Container security is by-passed. Likewise, component environment namespaces generally provide faster JNDI lookups since there is not a remote name service request. However, since a Service Locator pattern is used in the application, the benefits of a local, optimized naming service is less of a factor.

3.4 Use of Singleton Pattern, Caching

1. Use the singleton pattern to cache information that is neither dynamic, nor too large. Use a LRU caching pattern such as the one from Apache Commons to cache information that is too large and must allow rolling off least-used objects.
2. In general, singletons are a misnomer as there will be 1 per class loader, NOT per JVM. To have a real, singleton pattern, investigate JMX (Java Management Extension) MBeans. For most uses, the general singleton pattern is sufficient.
3. Accessor method. The accessor method to get a handle to a singleton will always be `getInstance()`.
4. Initialize a singleton in its accessor method (e.g., `getInstance()`), not class attribute declaration.

3.5 Error Handling

1. See Section 1.5 – Logging. Never use `exception.printStackTrace()`. Instead, log errors when handling exceptions like this:

```
log.error("Get xyz failed", exception);
```
2. Use chained exceptions to abstract upper layers from lower level detailed errors. CTMS will initially use the chained exceptions implementation new in JDK1.4.
3. To ensure unnecessary error logging is not incurred, do not log errors at each level of a nested/delegated call stack. Instead, chain exceptions and log the final chained exception in an error handler at the level of the *originator* of the call stack. The exception *cause* can be used to log each level of the chain to show where the error occurred.

3.6 Logging

1. Apache Commons-Logging API will be the standard logging API. This set of APIs can be used to abstract any logging framework including JDK1.4, Log4J, and others. Thus, if the project moves to another logging framework, the Apache Commons-Logging API shelters the code from this change.
2. Log4J will be the standard logging framework. It is also the standard logging framework used in many J2EE Servers. To configure logging, an XML configuration file called, `log4j.xml`, is used. Do not use the `log4j.properties` file, which is harder to interpret and update than the XML version. .
3. Developers are free to add new logger (and category?) entries in the development `log4j.xml`. However, the test, stage, and production `log4j.xml` files should only be modified by the technical lead(s).
4. Common Instrumentation: INFO level logging should be used to log duration (in milliseconds) of data access calls and the calls from the Struts controllers to the delegate. This information is routinely useful during integration testing and production rollout. Any other INFO level messages must be approved by the architect or team lead.
5. The six logging levels used by Log are (in order):
 1. trace (the least serious)
 2. debug

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

3. info
4. warn
5. error
6. fatal (the most serious)
6. Performance is often a concern when logging in an application. Therefore, before logging, always check to see if the appropriate level is enabled. Then, create the message and log it. For example,

```

if (log.isDebugEnabled()) {
    ... concat a bunch of messages here ...
    log.debug(theResult);
}

```

7. Put the correct level of logging inside the correct log-level check. For example:

```

// Do not practice this coding
if (log.isDebugEnabled()) {
    log.error(someMessage);
}

```

8. Do not log each message separately by calling the logging API several times in a row when the messages could be concatenated together and the logging API called just once. For example,

```

//Do not practice this coding
log.debug("Error:");
log.debug(message);
log.debug("...quitting now...");

```

```

//Instead, do this
StringBuffer buffer = new StringBuffer("Error:");
Buffer.append(message);
Buffer.append("...quitting now...");

```

9. Use `StringBuffer.append()` to concatenate several messages into a single message that is logged, if the messages are related. This is a common Java standard that should be followed not only for logging, but in other areas where multiple Strings are concatenated. When concatenating Strings using the "+" operator, excess amounts of Strings are created causing the garbage collector to overwork. `StringBuffer` does not create excess String objects in its appends and is more performant.

```

//Do not practice this coding
String error = "Error:"+message+"...quitting now...";
log.debug(error);

```

```

// Instead, see StringBuffer example above

```

10. Never use `exception.printStackTrace()`. Instead, log errors when handling exceptions like this:

```

log.error("Get xyz failed", exception);

```

11. Each class should have the following defined:

- a. A logger or logger parent setup in the `log4j.xml` configuration file.
- b. Declare a `Log` as a class attribute and get it from the `LogFactory`.

For example, if the `log` is not used in a static method:

```

import org.apache.commons.logging.*;

```

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

```
public class LogExample {
    private Log log = LoggerFactory.getLogger(getClass());
}
```

If the log is used in a static method:

```
private static Log log = LoggerFactory.getLog(Foo.class);
```

4. Client Coding Standards

The following are additional coding standards that are followed on the CTMS project for developing Java client applications that interface with the J2EE Server.

4.1 Database Access

Client-based connections such as from an external batch job cannot use server-side connection calls and should instead get connections directly from a DriverManager.

4.2 UI text

- All text that is presented on any screen in the system should be captured a properties files corresponding to the type of message. All keys used in each file should clearly identify the location of its use. For instance the poll status label on the menu should be referred to by the key `ctms_map_nav_menu_poll_status`.
 - `ctms_errors.properties` – contains all error messages that either logged or presented to the user via a dialog.
 - `ctms_images.properties` – contains all the icons and image references that are used in the client. (e.g. the key value pair for the splash screen should be `ctms_splash_image=/resource/images/splash.jpg`)
 - `ctms_labels.properties` – contains all the labels that are used in the client.
 - `ctms_menus.properties` – contains all the menu items that are used in the client
 - `ctms_messages.properties` – contains all the messages or notifications that are presented to the user.
- All properties should be obtained using a singleton resource handler that is populated at the start of the client application.
- If the string/resource is used several times by a class or if the class is instantiated or initialized more than once, the string should be declared as static final.
- All labels that prompt user entry and are accompanied by an entry field should always end with the ‘:’ character.
- All labels that are used on an element where an interaction will result in another dialog being launched, should end with “...”
- All default values for resource lookups should be appended by an underscore (“_”). For Example, please use the following standard.

```
ApplicationProperties.getInstance().getProperty("ctms_label_ex_default_title", "Exception Dialog_");
```

This way, the UI will present “Exception Dialog_” as the title, thus clearly identifying the label as missing in the resource file.
- Use MessageFormat rather than concatenation or appends for complex messages. For instance, a message for user Trey logging out should be: `MessageFormat.format("User {0} has logged out", new Object[] { username });`

Thereby avoiding the usage of several value-key pairs in the resource bundle.

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

4.3 Class Member Declarations

1. All class attributes should be declared with an underscore after the variable name the variable names that directly relate to swing objects should communicate the type of object they reference (e.g. a JLabel for exit should be declared as lExit_) The implementer should at a minimum stay consist within the working file. The following table serves as a guideline for some of the components.

Swing Component	Suggested Prefix
JButton	b, jb
JPanel	p
JFrame	f
JLabel	l
JComboBox	cb
JTable	t
ArrayList	al

2. All statics defined within a class should be referred to using the class name.

4.4 Panel Construction

Panels should be constructed in a piecewise fashion in that each major component of the panel is constructed in its own method (e.g. a panel with a table and a button panel should have at least two methods: makeButtonPanel() and makeTablePanel())

4.5 UI Component Factory

The UI Component factory should be used for the creation of all Swing components.

4.6 Client Logging

The client logging should adhere to the common standards, however, the log.isDebugEnabled() check is optional when instrumenting code for debug purposes.

5. J2EE Standards

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

The following are additional coding standards that are followed on the CTMS project related to the coding of J2EE applications.

5.1 Database Access

1. Database access should only be done through the EJB layer, never from a Web component directly.
2. All server-based connections are through the application server's configured `javax.sql.DataSource(s)`. This is enforced in the connection utility class, which should always be used.
3. Never use JDBC transactions. There should be no `connection.setAutoCommit()` calls in the source code. Instead, use JTA transactions. See Section 5 – Transaction Management for more information.
4. Use a database utility class to manage connections including getting and closing them. The database utility class should also manage `ResultSet` and `Statement` closing as well. The designated utility class for the CTMS project is `cdot.util.JdbcHelper`. Use this for all non-CMP JDBC calls.
5. The method that is the originator of a request to open a database resource (`Connection`, `ResultSet`, or `Statement`) should be the only method that closes the resource. For instance, if Method A (“originator”) gets a JDBC `Connection` and passes it as an argument to Method B (“delegate”), then Method B should not close the `Connection`. Method A should close the connection when Method B returns either on error or success.
6. Always ensure resources are closed in a `finally {}` block. Note: A `finally {}` block can be used with just a `try {}` block. There is no need to also have a `catch {}` block.

*** Use `JdbcHelper` methods since this utility class handles this ordering automatically.

7. DB resources should be closed in this order: `ResultSet`, `PreparedStatement`, `Connection`. Closing them out of order is an error. Not closing the former resource prior to closing the later will leave resources hanging for a while. For example:

```
/**
 * Database Utility class' close() method.
 * @param rs      ResultSet to close or null
 * @param stmtnt  Statement to close or null
 * @param conn    Connection to close or null
 */
public void close(ResultSet rs, Statement stmtnt, Connection conn) {
    // close ResultSet first
    if (rs!=null) {
        try {
            rs.close();
        } catch(exception ex) {}
    }
    // close Statement second
    if (stmtnt!=null) {
        try {
            stmtnt.close();
        } catch(exception ex) {}
    }
    // close Connection last
    if (conn!=null && !conn.isClosed()) {
        try {
            conn.close();
        } catch(exception ex) {}
    }
}
```

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

}

*** Use JdbcHelper methods since this utility class handles this ordering automatically.

8. Use PreparedStatements, not Statements whenever possible. PreparedStatements provide caching and parameter data type checking and default values.

5.2 Transaction Management

1. Always use JTA, never JDBC transactions.
2. Transactions are only managed in the EJB layer, never directly from a Web class like a JSP, servlet, or Action.
3. Unless absolutely necessary (see architect or team lead), use Container Managed transactions. Some business logic may require specific transaction management such as performing logic on bulk loads where each row is a separate transaction. When possible, have the looping logic in a separate EJB and call a Container Managed transaction EJB method that is set to RequiresNew to achieve this end.
4. For reading, use NotSupported for transactions unless CMP entity beans are involved in the system – in which case, use Supports or Required.
5. For inserts, updates, or deletes, use Required or RequiresNew, depending on the circumstances. If the method's outcome for commit versus rollback must NOT affect anything else, it is generally a good case for RequiresNew.
6. When an EJB throws an Application exception that will require the rollback of data, `EJBContext.setRollbackOnly()` must be called before the exception is thrown.
7. Per the EJB specification, "System exceptions" such as RuntimeException and RemoteException (as well as any of their subclasses) should be caught and EJBException rethrown instead.
8. Do not call `EJBContext.setRollbackOnly()` when throwing a System exception in a container managed EJB. The container knows that System exceptions are its cue to rollback the transaction anyway.

5.3 EJBs

1. Hold onto references to Stateless Session Bean (SLSB) EJBObjects. Do not re-create them from their EJB Home.
2. Do not remove SLSB instances by calling `SLSB_ejbObject.remove()`. The call is a no-op since SLSB are either pooled or created and destroyed per request anyway.
3. Always call `SFSB_ejbObject.remove()` on Stateful Session Beans (SFSB) EJBObjects when the instance is no longer required. This is required to inform the EJB Container that it can release any resources related to the SFSB.
4. As per the EJB specification, class attributes that are static must also be final. If it cannot be final, it cannot be static.
5. If the method is not a remote interface method, make it protected (acceptable) or private (preferred).
6. Perform initialization in the `setSessionContext()` or `setEntityContext()` methods.
7. Do not hold onto a Connection as an attribute in an EJB. They must be closed and returned to the Connection pool between requests.
8. When EJBs use dynamic properties in the application, use properties from a properties file or database rather than defining settings as bean environment properties in the EJB deployment descriptor. Properties files are easier to modify and bundle per environment (e.g., dev, test, stage, prod) in the build process. Also, depending on how the properties files are packaged in the application, they can be modified and reloaded without requiring a full EJB hot deploy to get the new settings.
9. As per the EJB specification, do not code threading logic in EJBs.
10. It is OK to access methods that are synchronized in another class. However, EJB methods must never be synchronized, as per the EJB specification.
11. There should be no presentation code in the EJB layer. In other words, you should never have Swing classes or `java.servlet.http.HttpServlet` classes imported in EJB source code.

CTMS/CTIS	Version: 1.1
CTMS Coding Standards	Date: 11-May-04

12. Stateless Session Beans naming convention: end with SLSB
13. Stateful Session Beans naming convention: end with SFSB
14. Container-Managed Entity Beans naming convention: end with CMP
15. Bean-Managed Entity Beans naming convention: end with BMP
16. Message-Driven Beans naming convention: end with MDB
17. Never directly call and entity bean from the client. Always use a session bean façade.
18. Use local EJBs for everything except MDBs (which have no local equivalent) and “remoteable” EJBs in the Delegate Layer.

5.4 Web Applications, JSP, Servlets, and Struts

1. Struts is the standard JSP framework and guides many decisions around the web application.
2. System exceptions should not be buried, but should forward to a friendly error page. The configuration files such as web.xml and struts-config.xml should be configured with Error Handlers to catch and perform this logic.
3. Application exceptions should not be buried, but logged and the user forwarded to a page that allows them an option to continue working or quit the system.
4. Database access is prohibited from the Web layer.
5. There should be no transaction management code in the Web layer, Struts based or not.
6. Actions must inherit from `BaseAction`.
7. Forms must inherit from `BaseForm`