[Programming Naming and Coding Conventions](https://wiki.eisgroup.com/display/CRC/EIS+Programming+Naming+and+Coding+Conventions)

# Purpose

The purpose of the document is to provide code standard for developers who work on EIS Suite extensions.

# Scope

The document covers the following topics:

* Naming Conventions
* Coding Conventions

 It does not describe the general java code standard which is covered in the [Google Java Coding Conventions](https://google.github.io/styleguide/javaguide.html), but elaborates the development best practices specific for EIS projects.

# Coding Conventions

## **General Naming and Coding Conventions**

 Project name prefix is to be used in Class names only if Class overrides/extends some base functionality.

 For example:

* QuoteRateActionImpl - base service is to be extended
* AbcQuoteRateActionImpl - for ABC project

## **Comments in the code**

1. Comments should follow generic requirements (length and indentation)
2. Each class and interface must have JavaDoc comments with a short description of its purpose and responsibility
3. It is better to keep code “self-commented”
4. In comments describe the business logic behavior, not the code behavior
5. Don‘t touch existing comments during common development process (it can cause additional issues during merge).

## **Annotations**

1. All overridden methods must have @Override annotation
2. Contract of public API of each module should be expressed by appropriate JSR-305 annotations if it is possible.
3. When an override is annotated the following principle of subcontracting must be kept: Subclasses in an inheritance hierarchy are allowed to weaken preconditions (but not strengthen them) and strengthen postconditions and invariants (but not weaken them).

## **Exception Handling**

1. Do not use checked exceptions
2. Use**ProcessingException**for system failures that user cannot recover from
3. Use **ValidationException** to report data errors that user could recover from
4. Do not use exceptions as validation mechanism - there should be validation services build to validate user input and called from UI. Exceptions should only to be used if service detects invalid data on business transaction processing that should be validated in previous flow steps.

## **Internationalization**

**Do not use** Locale.getDefault() throughout the application. Instead **inject i18n bean** and use it's method

|  |
| --- |
| i18n.getLocale() |

If bean injection is not available in a specific context then use I18n static method

|  |
| --- |
| I18n.getDefaultLocale() |

## **Spring Beans**

* spring beans names should start from lowercase
* When defining bean names in spring xml configuration id tag is preferred instead of name. As it enforces xml parser validation if <idref local=""> tag is used.

**Spring Beans Naming**

|  |
| --- |
| <**beans**>  ...      <**bean** id="lookupDAO" class="com.exigen.ipb.base.lookups.dao.LookupJPADAO"></**bean**>      <**bean** id="numberRangeDAO" class="com.exigen.ipb.base.numbergenerator.dao.NumberRangeJPADAO"></**bean**>        <**bean** id="codeValueBinder" class="com.exigen.ipb.base.lookups.CodeValueBinder"></**bean**>      <**bean** id="codeValueProvider" class="com.exigen.ipb.base.lookups.CodeValueProvider">          <**property** name="lookupDAO"><**idref** local="lookupDAO"/></**property**>      </**bean**>  ...  </**beans**> |

## **Lookup maintenance**

1. All lookups should be loaded via CSV files, direct updates strictly prohibited
2. Each lookup should be represented as one file for each product but for all states (No CSV files per states)
3. All files must locate in folder domain-name\domainname-deploy\src\main\resources\csv
4. For versioning VCS should be used, no versioning via file names
5. CSV file name pattern: <lookupName>\_<product>.csv. For example: ABCDoNotRenewReason\_SS.csv
6. All change sets related to lookups should be moved to a separate file - lookup xml file (see next point). Each modification of CSV file should trigger creation of the a new liquibase changeset. Always use cumulative approach for lookup data change.  That means that the same csv file should be reused and contain full data.
7. Under each deploy project there should be one(single) lookup xml file:
   1. products/auto/abc-auto-liquibase-deploy/db - abc-lookups.xml for AUTO (CSA and SS) lookups
   2. products/home/abc-home-liquibase-deploy/db - abc-lookups-home.xml for HOME lookups
   3. products/pup/abc-pup-liquibase-deploy/db - abc-lookups-pup.xml for PUP lookups

Any lookup values delete must be product specific! For example, if you want to update all lookup values for SS product:

**Not allowed**

|  |
| --- |
| DELETE FROM LOOKUPVALUE WHERE LOOKUPLIST\_ID IN (SELECT ID FROM LOOKUPLIST WHERE LOOKUPNAME='ABCAgentType'); |

**Must be**

|  |
| --- |
| DELETE FROM LOOKUPVALUE WHERE LOOKUPLIST\_ID IN (SELECT ID FROM LOOKUPLIST WHERE LOOKUPNAME='ABCAgentType') AND (PRODUCTCD='ABC\_SS'); |

Sample scenarios how to work with lookups:

| **New Lookup** | 1. Create a new CSV file 2. Create a new changeset in lookups xml file |
| --- | --- |
| **Modify lookup which already exists in lookups xml file** | 1. Modify an existing CSV file with new values 2. Update an existing changeset (update author and id) 3. If existing changeset does not contain delete section then add it and delete all existing lookup data to be able to correctly load new values |
| **Modify lookup which currently does not**  **exist in lookups xml file** | 1. Gather all required values from the existing changesets and CSV files 2. Move data to a new CSV file 3. Create a new changeset in lookups xml file 4. Delete all existing changesets and CSV files related with this lookup    1. If it is impossible to gather all required values or it is takes too much time, export lookup data from QA DB of appropriate app version |

## **Materialized view maintenance**

1. All changesets should be placed into file: (Direct updates strictly prohibited)
   1. \src\products\common\xxx-common-liquibase-deploy\src\main\resources\db\xxx-views-common.xml.
2. Each view should be created in separate sql file.
3. All SQL files to be in the folder /sql/views/\*.
   1. Sample: /sql/views/create-bilsumins\_v.sql
4. SVN/Mercurial/Git should be used for versioning

 Sample scenarios how to work with materialized views:

| **New materialized view** | 1. Create a new SQL file which actually creates new materialized view 2. Create 2 new changesets in file abc-views-common.xml 3. Modify the value of attribute ‘common\_set\_id’ in the configuration section of the abc-views-common.xml file . 4. Modify the value of attribute ‘author\_id’ in the configuration section of the abc-views-common.xml file. 5. Modify the value of attribute ‘comment\_text’ in the configuration section of abc-views-common.xml file. |
| --- | --- |
| **Modify existing materialized view** | 1. Modify existing SQL file 2. Modify the value of attribute ‘common\_set\_id’ in the configuration section of the abc-views-common.xml file. 3. Modify the value of attribute ‘author\_id’ in the configuration section of the abc-views-common.xml file. 4. Modify the value of attribute ‘comment\_text’ in the configuration section of the abc-views-common.xml file. |

## **Casting PolicySummary to Subtypes**

 The following rules of policy type casting must be kept by developers:

1. In common code com.exigen.ipb.policy.domain.PolicySummary can be cast without type checking only to com.exigen.ipb.policy.domain.PolicyEntity.
2. For product specific logic, a <eis:serviceSubstitute /> tag  should be utilized : [Spring bean override during runtime (execution context aware proxies)](https://wiki.eisgroup.com/pages/viewpage.action?pageId=122390269)
3. In all other cases, a developer must check type of PolicySummary instance before casting it to a subtype.

## **Privilege Based Access**

Our application is privilege based - it means that we cannot use user group/role checking in code. Any US which requires validation based on user group/role must be rejected. BA should provide privilege name.

In general case, all permission based US-es should be implemented by the following steps:

1. Create a new permission, update code to restrict some functionality by this permission
2. Provide information about permission and configuration SQL script for role to QA
   1. Permission should not be associated with any role by default (via liquibase)
3. Update production scripts for role - permission mapping

## **Static Classes**

Active usage of static classes leads to the following issues:

1. Introducing high coupling between classes and as a result significantly increased cost of any change that affects the static class, because there is no way to localize ripple effect of the change.

To avoid all those issues the following rules need to be followed by developers and verified by reviewers:

1. Creating new static classes can only be introduced through Architecture review and **approval.**
   1. Singleton Spring beans should be used instead. To simplify using such classes a developer can use sub-project specific framework.
   2. Note: utility classes from external libs can be used.
2. Using project static utility classes like com.exigen.abc.docgen.transformation.utils.DocGenMappingUtils is forbidden. If you need to use such class, please convert this class to POJO (and inject it with by Autowired annotation) or create a wrapper and make the utility static class deprecated. Here is an example of the refactoring:
   1. Deprecated static class: com.exigen.abc.docgen.transformation.utils.DocGenMappingUtils
   2. Wrapper: com.exigen.abc.docgen.transformation.utils.ABCDocGenMappingService
   3. The wrapper can be accessed either via Spring injection (annotation @Autowired can be used in this case, because it won't be any conflicts with EIS beans)

Please see more in the [Unit and Integration Testing Guidelines](https://wiki.eisgroup.com/display/CRC/Unit+and+Integration+Testing+Guidelines) section.

## **Current Date for Jobs**

All jobs that require to use the business date logic should use a special method

|  |
| --- |
| BussinesCalendarService#getBussinesDay(SchedulingOption schedulingOption) |

to get a 'current date’. The ‘current date’ means a current business day for job processing. If job is run after a midnight it must work in the same way as if it was run before midnight previous day.

For example, if a job searching for policies that reached a R-30 day point, it should not start processing new set of policies (R-29) after a midnight. All business rules and requirements should use this special business day to prevent processing policies that are not supposed to be processed. For example, if we send a cancellation notice in 15 days it should not be 14 days because a job was run after midnight.

The interface

|  |
| --- |
| BussinesCalendarService#getBussinesDay(SchedulingOption schedulingOption) |

returns date for which Scheduled job should process tasks. Business end time is taken from JobsConfiguration lookup value.