Aranea Developer's Guide

Jevgeni Kabanov

1. Introduction

This document is intended for developers contributing to Aranea code base and contains the main standards imposed on the integrated code. It comprises the official guideline to Aranea code development and review. It also serves as an introduction to the Aranea code base and its development.

2. Root Layout

Aranea root directory contains the following files and directories:

Filename	*Description*
`doc/`	Contains a subdirectory with an Ant build file for each document like Reference Manual or White Paper (`doc/reference/`, `doc/white-paper`).
`doc/support/`	Contains support files for DocBook XSL.
`etc/`	Contains non-Java sources (Like Javascript, XML, CSS files) as well as configuration and other support files.
`examples/`	Contains a subdirectory for each example application with Ant build file, web resources and web application descriptors.
`examples/lib/`	Contains third-party libraries that example applications are dependent on.
`examples/common/`	Contains common example files and sources.
`lib/`	Contains third-party libraries that Aranea is dependent on.
`src/`	Contains Aranea Java sources and accompanying files (e.g. Javadoc HTML files)
`tests/`	Contains tests and mock files for Unit & Integration testing.
`build.xml`	Master build file.
`Changelog.txt`	Should mark all the changes between releases.
`License.txt`	Apache License.
`Notice.txt`	Should contain notices for all third-party libraries used in the distribution.
`Readme.txt`	Should contain basic information about distribution and release including *known issues*.

3. Building and Testing

Aranea is built by the master build file `/build.xml`. The following targets are used:

Target	*Description*
`build-all` (default)	Builds Aranea and all of the examples.

Target	*Description*
`build`	Builds Aranea .JAR files
`clean`	Cleans built Aranea files.
`clean-all`	Cleans built Aranea and example files.
`doc`	Builds Aranea documentation (DocBook, JavaDoc, ...).
`dist`	Builds whole Aranea distribution.

Each example also contains it's own `build.xml` file, that can be used to build, run and test the example. Examples use Jetty web container to run themselves. The corresponding target is `run-app` and typically the application will be available at `http://localhost:2000/`. Some applications also need the database to be started using target `run-database`.

4. Modules

Aranea source and distribution is split into modules for simplifying dependency management. On the source level it is done via Java packages starting from the `/src/org/araneaframework/<modulename>` directories. On the distribution level this is done via `aranea-<modulename>.jar` files.

The module separation have following functions:

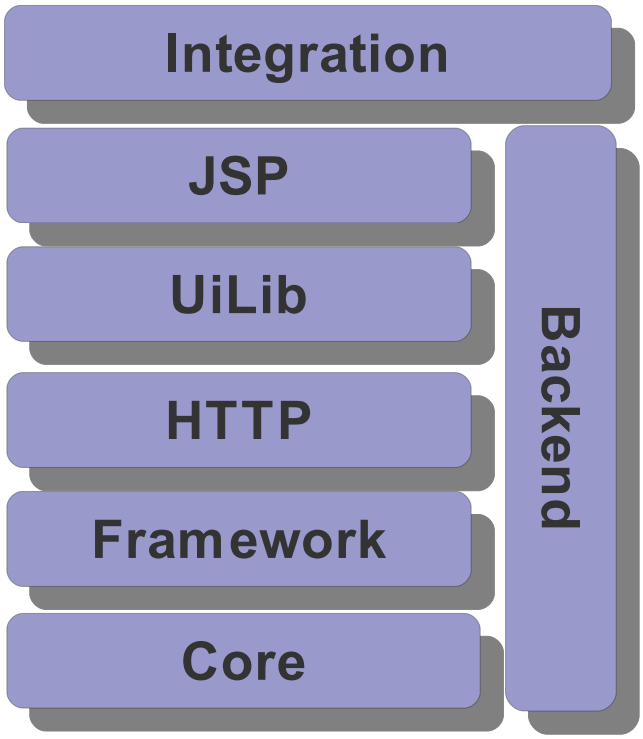
- To separate the large interdependent chunks of code into separate distribution units.
- To provide a possibility to include only a subset of Aranea and dependent libraries.
- To create a layered dependency graph among modules.

Therefore a new module should be created only when there is a sufficiently large chunk of code that clearly fits into a layered dependency model and is clearly separated from others.

Currently Aranea has the following modules:

Module	*Description*	*Dependencies*
`core`	Contains Aranea core interfaces and their base/standard implementation. Should be as minimal as possible and shouldn't depend on anything unless absolutely necessary. Includes both the `/src/org/araneaframework/*` and `/src/org/araneaframework/core/**`.	-, ???
`framework`	Contains interfaces and implementations that make up the executable Aranea framework (filters, routers, etc) that do not depend on web. Includes the `/src/org/araneaframework/framework/**`.	aranea-core, ???
`http`	Contains interfaces and implementations that make up the executable Aranea framework (filters, routers, etc) that depend on web (specially servlet) interfaces. Includes the `/src/org/araneaframework/http/**`.	aranea-core, aranea-framework, ???
`uilib`	Contains a widget repository that supports different web	aranea-core, aranea-

Module	*Description*	*Dependencies*
	GUI elements, including forms and lists. Includes the `/src/org/araneaframework/uilib/**`.	framework, aranea-http, ???
`jsp`	Contains JSP tags and support files. Includes the `/src/org/araneaframework/jsp/**`.	aranea-core, aranea-framework, aranea-http, aranea-http, ???
`integration`	Contains a subpackage for each integrated third-party framework or library. Includes the `/src/org/araneaframework/integration/**`.	aranea-*, ???
`backend`	Contains files unrelated directly to UI, but providing some necessary logics. Includes the `/src/org/araneaframework/backend/**`.	-, ???



5. Naming and Layout

5.1. Java packages

The root of the module should contain classes that make up the main contract of the module. This includes core interfaces and exceptions, but never implementations.

The first level of Java packages under the module should be arranged using a vertically split hierarchy. This means that instead of referring to the functionality provided by the classes (like io, http, ...) we should refer to their roles (util, support, widget, context, ...). Exceptional cases are when the package hierarchy is too unbalanced and is not expected to grow.

The further arrangement can combine both horizontal and vertical splitting and is up to the discretion of the programmer.

NB! Package names are *always* singular.

5.2. Java Interface Names

Interfaces names should not have any prefixes should refer to the functionality provided by the interface. Interface may have a suffix that refers to the pattern or idiom implemented by the interface (e.g. Context, Listener, ...).

5.3. Java Class Names

Class names consist of an optional prefix, the descriptive name and zero to many suffixes.

Prefix is generally added only in two cases:

1. “Base” prefix is added to abstract classes that serve as base implementation of a particular interface (e.g. BaseWidget implements Widget).
2. “Standard” is added to non-abstract (possibly final) classes that server as a standard implementation of a particular interface (e.g. StandardEnvironment implements Environment). **NB!** “Standard” should generally not be used as the prefix to classes the programmer either instantiates or subclasses in Java code.

Note that if their can be several different implementations of the same interface “Standard” should not be added and they should be distinguished by their descriptive name.

Suffixes can be used and combined pretty arbitrary, but the following rules should be followed:

1. Generally main interface (referring to the class role) that the class implements or the abstract class that the class extends should be reflected in its name as the last suffix. This concerns Widgets, Components, Services, Messages, Constraints and so on.
2. Pattern(s) that the class implements should also be reflected in its name. In most cases this will not be the last suffix (as in *RouterServise, *FilterWidget), but if this leads to redundancy it may be last as well (RelocatableServiceDecorator).
3. Some roles don't have a corresponding interface:
    - Non instantiated utility classes containing static methods should have a suffix “Util”.
    - Instantiated utility classes containing virtual methods should have a suffix “Helper”.

NB! A general rule of thumb is that suffixes should not be redundant and in such a case it is allowed to omit them (e.g. StandardServiceFilterChainFilterService is redundant and can be contracted to StandardFilterChainService). Generally it's preferred to leave the last suffix intact as it uniquely identifies the class role, so it's better to contract the descriptive name.

Further are brought some example of correct class names:

- StandardEnvironment
- BaseApplicationComponent
- RoutedMessage
- StandardServiceAdapterWidget
- StandardMountPointFilterService
- SystemFormHtmlTag

- FormTag
- StandardFileImportService

5.4. Java Method Names

Some conventions concerning common method names prefixes:

Name prefix	*Description*
`get`	Getter methods should be idempotent and should not generally modify any state. They should return null or an empty collection instead of throwing exceptions.
`require`	Require is similar to a getter, but throws an exception if the entity cannot be found.
`set`	Set may update state, but should generally be idempotent. Set may return "this" to facilitate chained calls.
`add`	Add may update state and should not be idempotent. It may also return "this".

6. Documentation

6.1. Javadoc and Comments

Every class should have the following text in the header:

```
/**
 * Copyright 2006 Webmedia Group Ltd.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
**/
```

Every class and method should be documented. Obvious parameters and return values don't have to be. Generally avoid redundancy in Javadoc.

The following tags should be present:

Tag	*Description*
`@author`	For every author of the class in the form Name Sirname (username@domain.com)

Tag	*Description*
`@since`	For all versions of framework starting with 1.1 should mark classes added in those versions.
`@see`	Use liberally! E.g. contexts may link to implementing classes and vice versa.
`@deprecated`	Try to never deprecate API. It's not going to be removed in any case :(.

6.2. Reference Manual

Reference manual should be the primary source of information, not JavaDoc. Ergo things can be copied from RefMan to Javadoc. Every meaningful feature (e.g. every context) must have a meaningful entry in the RefMan.

6.3. Code Tags

“XXX” and “TODO” are acceptable comment tags that can be used to mark correspondingly temporary hacks and places needing later development. NB! Tags are not a substitution for issue tracking, anything important should be marked there.

7. Code Style

7.1. Exceptions

Aranea supports an idiom that GUI framework should not require checked exceptions. Therefore all abstract methods overridable by the programmer should allow throwing Exception's (throws Exception). This includes both classes with abstract methods and interfaces that are used for callbacks (e.g. methods are never called by the programmer).

All Aranea public interface methods may throw checked Exceptions only in really exceptional cases.

7.2. Assertions

All methods should use Assert class to check the preconditions. Postconditions and invariants are optional for the particular cases. Specially all methods with object parameters should provide a precondition or logics for handling null values.

7.3. Tests

Tests should cover 100% of core, framework, http, uilib and backend modules. The rest are optional.

8. Design

Some rules of the thumb for designing Aranea code:

- Interfaces should always be preferred over classes as method parameters.
- Aggregation (containment) is preferred over inheritance unless very clumsy.
- Utility methods are preferred over new classes.
- Every context should be responsible for one task only (e.g. flow management). All other tasks

should be achieved by composing different contexts instead of