norther.org

Tammi Application Framework

Technical Specification

Version 6.1.1

Revision History

Date	***Version***	***Description***	***Author***
12.12.02	1.0.0	Initial version	imp
02.01.03	1.0.1	Packages and configuration added	imp
03.01.03	1.0.2	Implementation added	imp
08.01.03	1.0.3	Revised version	map
20.01.03	1.0.4	Minor changes from build 31	imp
03.07.03	1.1.0	Updated for release 1.1	imp
05.07.03	1.1.1	Clarification of concepts	imp
31.08.03	1.1.2	Swing changed to JFC	imp
10.10.03	1.1.3	Some typos corrected	imp
03.11.03	2.0.0	Release 1.1 renumbered to 2.0	imp
05.01.04	2.1.0	Updated for release 2.1	imp
05.01.05	2.2.0	Updated for release 2.2	imp
11.01.05	2.3.0	Updated for release 2.3	imp
07.06.05	3.0.0	Updated for release 3.0	imp
10.08.09	6.1.0	Updated for release 6.1	imp
01.12.09	6.1.1	Updated for release 6.1.4	imp

Table of Contents

1 Introduction

The Tammi Application Framework is a Java™ component based development framework and run-time container for applications supporting web browsers, mobile terminals and/or JFC based user interfaces. Tammi application components can implement independent business logic themselves or act as proxies to native libraries, remote programs and other kinds of manageable resources.

1.1 Purpose

This document contains a technical specification of Tammi. The on-line manual of Tammi supplements this document by providing a summary of features and illustrative examples on how to apply the framework. In addition, API documentation of Tammi packages and classes provide more detailed information.

1.2 Scope

The scope of this document is to describe design, implementation, configuration and other technical issues on which Tammi itself as well as its applications and derivatives are based.

1.3 Definitions, Acronyms and Abbreviations

Term	***Explanation***
BSF	Bean Scripting Framework
CGI	Common Gateway Interface
DOM	Document Object Model
DTD	Document Type Definition
EJB	Enterprise Java Bean
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JNDI	Java™ Naming and Directory Interface
JMX	Java™ Management Extensions
JRE	Java™ Runtime Environment
JSP	Java™ Server Pages
JVM	Java™ Virtual Machine
MBean	Managed Bean (a specific Java class)
MVC	Model – View – Controller design pattern
OJB	ObJect Relational Bridge
RI	Reference Implementation
RMI	Remote Method Invocation
SAX	Simple API for XML
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WUI	Web User Interface

Term	Explanation
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations

1.4 References

[Ant]	Name Link	*Apache Ant Build Tool* http://jakarta.apache.org/ant.index.html
[Apache]	Name Link	*The Apache Software Foundation* http://www.apache.org/index.html
[BSF]	Name Link	*Apache Jakarta Bean Scripting Framework* http://jakarta.apache.org/bsf/index.html
[Castor]	Name Link	*Castor XML Binding Framework* http://castor.exolab.org/index.html
[CCA]	Name Link	*Common Component Architecture Forum* http://www.cca-forum.org/index.html
[Coding]	Name Link	*Java Coding Standard for Tammi* http://tammi.sourceforge.net/docs/spec/coding.pdf
[Coms]	Name Link	*Apache Jakarta Commons* http://jakarta.apache.org/commons/index.html
[CPJ]	Name Link	*Concurrent Programming in Java* http://gee.cs.oswego.edu/index.html
[DJava]	Name Link	*DynamicJava Interpreter* http://www-sop.inria.fr/koala/djava/index.html
[Flex]	Name Link	*Flex.org – Rich Internet Applications* http://flex.org/
[FM]	Name Link	*<FreeMarker>* http://freemarker.org/index.html
[Groovy]	Name Link	*An agile dynamic language for the Java Platform* http://groovy.codehaus.org/
[HSQL]	Name Link	*HSQL Database Engine* http://hsqldb.sourceforge.net/index.html
[JAF]	Name Link	*JavaBeans™ Activation Framework* http://java.sun.com/products/javabeans/glasgow/jaf.html
[Jasper]	Name Link	*Jasper Reports* http://jasperreports.sourceforge.net/index.html
[Java]	Name Link	*Java™ 2 Platform, Standard Edition* http://java.sun.com/j2se/index.html
[JFree]	Name Link	*JFree Charts and Reports* http://www.jfree.org/
[Jini]	Name Link	*Jini™ Network Technology* http://www.sun.com/jini/index.html

[JMail]	Name Link	*JavaMail™ API* http://java.sun.com/products/javamail/index.html
[JMX]	Name Link	*Java™ Management Extensions (JMX™)* http://jcp.org/aboutJava/communityprocess/final/jsr003/
[JMXSpec]	Name Link	*JMX™ Instrumentation and Agent Specification* http://java.sun.com/aboutJava/communityprocess/final/jsr003
[JSP]	Name Link	*JavaServer Pages™* http://java.sun.com/products/jsp/index.html
[Log4J]	Name Link	*Apache Jakarta Log4J Logging Package* http://jakarta.apache.org/log4j/docs/index.html
[OJB]	Name Link	*Apache DB Object/Relational Bridge* http://db.apache.org/ojb/index.html
[Servlet]	Name Link	*Java™ Servlet Technology* http://java.sun.com/products/servlet/index.html
[Tomcat]	Name Link	*Apache Jakarta Tomcat Servlet Container* http://jakarta.apache.org/tomcat/index.html
[Turbine]	Name Link	*Apache Jakarta Turbine Servlet Framework* http://jakarta.apache.org/turbine/index.html
[Velocity]	Name Link	*Apache Jakarta Velocity Template Engine* http://jakarta.apache.org/velocity/index.html

1.5 Overview

The rest of the document provides a technical description of Tammi including installation instructions, applied design patterns, available packages, implementation guidelines and configuration notes.

2 Installation Instructions

Tammi provides a run-time container for executing Java components installed under Tammi as plug-in applications. The framework itself may be installed as a web application running as a filter chain or a servlet under some servlet container [Servlet], or it may be installed as a stand-alone system. The stand-alone system can be configured to provide HTTP and HTTPS services via socket connectors.

2.1 Framework

2.1.1 Installation Files

The stand-alone installation is packaged into one deployment file, named "tammi-<version>.<extension>", which can be extracted to a desired location in the directory hierarchy. The extension is either "zip" or "tar.gz" depending on the corresponding compression method.

The web application installation is packaged into one web archive file, named "tammi.war", which can be located in the web application directory of the servlet container to be applied.

2.1.2 Directory Structure

The directory structure and contents of the framework installation without documentation are presented below. Note that all subdirectories of components are not presented. Note also that most directories may have been embedded into jar archives.

Example

```
/bin                         -> commands, binaries and native libraries
/mmd                         -> static multimedia data
..../default
......../tammi
............/spray
................/css
................/image
................/script
/etc                         -> configuration directories
..../config                  -> property files
......../tammi
............/core
............/flex
............/gate
............/root
............/spray
............/sprig
............/manual
..../script                  -> run-time scripts
..../security                -> key stores and certificates
..../start                   -> startup scripts
......../tammi
............/core
............/root
............/leaf
............/spray
............/sprig
............/manual
..../stop                    -> shutdown scripts
..../test                    -> test scripts
/lib                         -> jar archives of Java libraries
/lic                         -> license files and copyright notices
/res                         -> localized resources
/tpl                         -> markup templates
..../default
```

The directories may contain subdirectories defining namespaces for component files included in them. The namespace of the framework itself is “tammi”.

After installation, the framework can be started with the default configuration by running the shell commands in the “startup.sh” file in the “bin” directory under Unix/Linux, or by running the batch commands in the “startup.bat” file under Windows. The Java JRE version 1.5 or newer must have been installed and available (version 1.4 can be applied if Java™ Management Extensions [JMX] is installed separately).

The “initd/tammid_<os>_x32.sh” shell script can be applied to install the framework as a Unix daemon. Correspondingly, the “tammis_w32.exe” executable installs the framework as a Windows service. Available options are displayed with the command line option /?.

The default ports are http://localhost:8080 and https://localhost:8443. The user name and password for management are “admin” and “admin” correspondingly.

2.2 Applications

2.2.1 Installation Files

By default, applications to be plugged into Tammi are packaged into a deployment file, named "tammi-<name>-<version>.<extension>". The extension is either "zip" or "tar.gz" depending on the corresponding compression method. The deployment file may be extracted either to the same installation root directory as the framework, or to its own directory next to the framework directory.

2.2.2 Directory Structure

The directory structure and contents of the Hello application installation are presented below. It applies the directory structure of the framework but uses its application name "hello" as its namespace within the directory hierarchy. Other applications may utilize more directories depending on their requirements. After installation, the framework must be restarted to load the plug-in application.

Example

```
/mmd                         -> static multimedia data
..../default
......../hello
............/image
/etc                         -> configuration directories
..../config                  -> property files
......../hello
..../start                   -> startup scripts
......../hello
/res                         -> localized resources
/tpl                         -> markup templates
```

3 Design Patterns

Tammi's architecture is compliant with Java™ Management Extensions [JMXSpec]. Applications are formed by independent components plugged into Tammi, and configured dynamically during startup and run-time with scripting engines, such as DynamicJava [Djava] or Groovy [Groovy]. Applications follow the Model – View – Controller (MVC) design pattern separating presentation from content. The presentation is implemented as templates in some markup language, like HTML, interacting with content producers through context tools within template contexts. A template engine, such as Velocity [Velocity], parses the templates composing the web user interface (WUI). A filter chain mechanism [Servlet] controls the process. Stand-alone applications may support JFC based user interfaces as well.

Tammi's component model supports integration of versatile Java technologies, open source packages and existing applications into a consistent development framework of professional quality. A specific objective is to support transition of legacy systems in the engineering domain to component-based solutions smoothly and efficiently by applying the proxy pattern to avoid complete rewrites of the code base.

3.1 Component Model

The Common Component Architecture [CCA] Forum is a group of researchers defining a component architecture for high performance computing. Tammi does not conform to CCA but is committed to the same principles and goals, listed below, in the engineering domain.

Like scientific applications, applications in the engineering domain are also often assembled from large blocks of code into monoliths. Software reuse is obtained by linking with software libraries obtained either from third parties, or created in house, from scratch. A major disadvantage of this approach is that software boundaries (function interfaces and global symbols) are frequently not well thought out. This can lead to internal code dependencies making the monolithic application difficult to modify and maintain.

Components are designed with standard, clearly defined interfaces, which tend to protect them from changes in the software environment outside their boundaries. Applications are composed at run-time from components selected from a component pool. Because components communicate only through well-defined interfaces, when an application needs to be modified, a single component can be modified (or exchanged for a similar component), without fear of disturbing the other components making up the application.

The framework provides the glue that binds components together. It is used to compose separate components from a component pool into a running application. It allows components to be linked together and to make calls on specific component interfaces. Additionally, the framework can provide information about the run-time environment.

3.2 JMX™ Architecture

Application components in Tammi's architecture share common services and communicate with each other through a centralized object registry server. The components may be distributed in the network when applicable, but they can as well reside on the local host.

The architecture is implemented on top of JMX™ [JMX]. The JMX architecture is divided into three levels [JMXSpec]:

- Instrumentation level
- Agent level
- Manager level

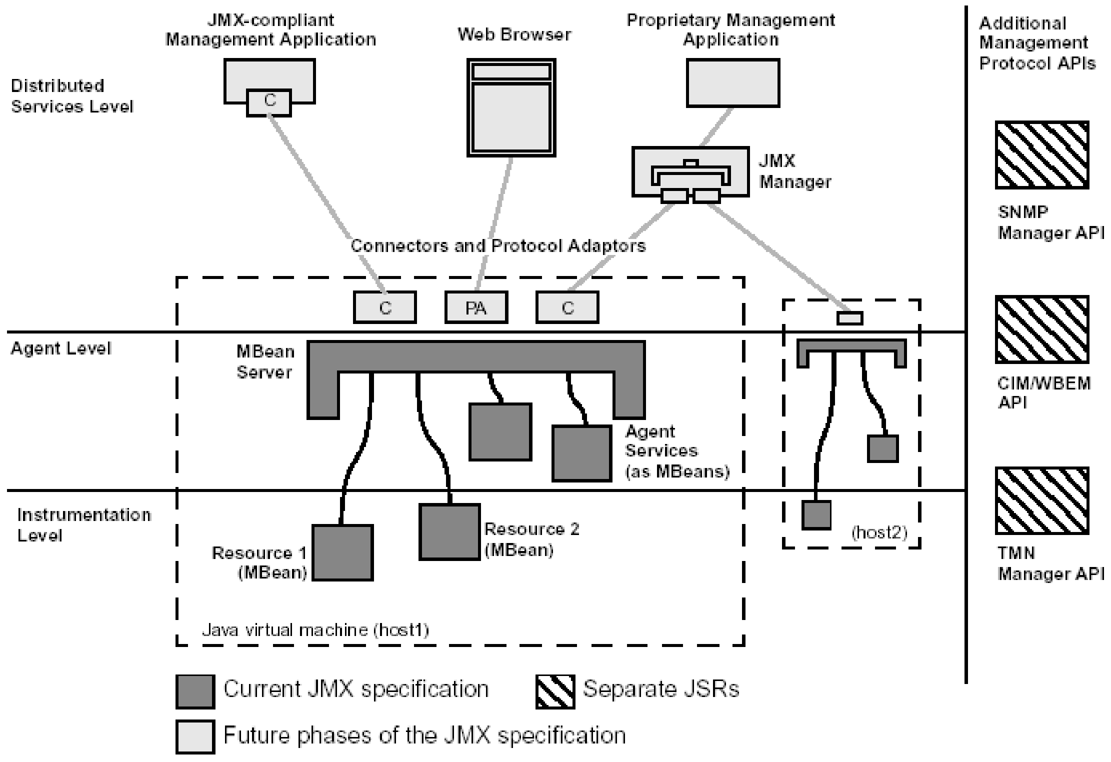
JMX-compliant Management Application
Web Browser
Proprietary Management Application
Additional Management Protocol APIs
Distributed Services Level
C
JMX Manager
SNMP Manager API
Connectors and Protocol Adaptors
C
PA
C
Agent Level
MBean Server
CIM/WBEM API
Agent Services (as MBeans)
Instrumentation Level
Resource 2 (MBean)
Resource 1 (MBean)
(host2)
TMN Manager API
Java virtual machine (host1)
Current JMX specification
Separate JSRs
Future phases of the JMX specification

Figure 3.1 Relationships between the components of the JXM architecture [JMXSpec]

3.2.1 Instrumentation Level

The instrumentation level provides a specification for implementing manageable resources. A manageable resource can be an application, an implementation of a service, a device, a user, etc. It is developed in Java, or offers a Java wrapper, and has been instrumented so that it can be managed by JMX-compliant applications.

The instrumentation of a given resource is provided by one or more Managed Beans, or MBeans, which are Java objects implementing a specific interface describing the attributes and operations they provide for accessing the resources behind them. The instrumentation of a resource allows it to be manageable through the agent level.

MBeans do not require knowledge of the agent with which they operate. Products can be made manageable without having to understand management systems. Existing products can provide proxy MBeans wrapping the original functionality making existing resources manageable with relatively light effort.

The instrumentation level specifies also a notification mechanism. This allows MBeans to generate and propagate notification events to components of the other levels.

The basic functionality of the Tammi framework is implemented as MBeans. Third-party libraries has been integrated to the framework through proxy MBeans. Correspondingly, new application components may be implemented directly as MBeans, or they can be integrated through a proxy layer. A generic adapter MBean may be applied to introduce any Java class to the framework without modifying the code of the original class.

3.2.2 Agent Level

The agent level provides a specification for implementing agents. Management agents directly control the resources and make them available to management applications. Agents are usually located on the same host as the resources they control, although it is not a requirement.

This level builds upon and makes use of the instrumentation level, in order to define a standardized agent to manage resources. The agent consists of an MBean server and a set of services for handling MBeans. In addition, an agent will need at least one communications adapter or connector.

An agent does not need to know which resources it will serve: any manageable resource can use any agent that offers the services it requires. Managers access an agent’s MBeans and use the provided services through a protocol adapter or connector. Agents do not require knowledge of the remote management applications that use them. Agents can be implemented without having to understand the semantics of the manageable resources, or the functions of the management applications.

Tammi can be configured to create a new JMX agent to serve MBeans of its applications, or to register itself to the MBean Server of an existing agent.

3.2.3 Manager Level

The manager level provides the interfaces for implementing managers. This level defines management interfaces and components that can operate on agents or hierarchies of agents. These components can:

- Provide an interface for management applications to interact transparently with an agent and its manageable resources through a connector
- Expose a management view of an agent and its MBeans by mapping their semantic meaning into the constructs of a data-rich protocol (for example HTML or SNMP)
- Distribute management information from high-level management platforms to numerous agents
- Consolidate management information coming from numerous agents into logical views that are relevant to the end user’s business operations
- Provide security

The manager level contains client browsers and integrated client platforms using the functionality, information and other resources provided by Tammi applications.

3.2.4 Future Concepts

The APIs of the JMX specification can implement flexible and dynamic solutions, which can leverage other emerging technologies. For example, JMX solutions can use lookup and discovery services and protocols such as Jini™ connection technology [Jini], Universal Plug'n'Play (Upnp), and the Service Location Protocol (SLP).

The JMX Remote API standardizes a solution for exporting JMX API instrumentation to remote applications. The remote API will provide a mechanism for remote access that is very similar to the local client API. This means that remote clients can call the familiar MBeanServer operations and can register for MBean notifications.

Jini can go a step further providing spontaneous discovery of resources and services on the network, which are then managed by through a JMX application.

3.3 Deployment

Tammi is running on an agent server connected optionally to a separate web server through appropriate protocol connectors. Clients can access Tammi either directly through an HTTP connector or via the separate web server. Stand-alone clients can connect to Tammi through a graphical user interface.

An optional database server can be accessed through database service brokers available for vendor specific database implementations.

Application components implement their manageable resources as MBeans registered to the MBean server of the JXM agent. Applications can reside in the same JVM as Tammi, as local applications on the same host, or as remote applications on a remote server. In the two latter cases, the application components are implemented as proxy MBeans for accessing remote application resources.

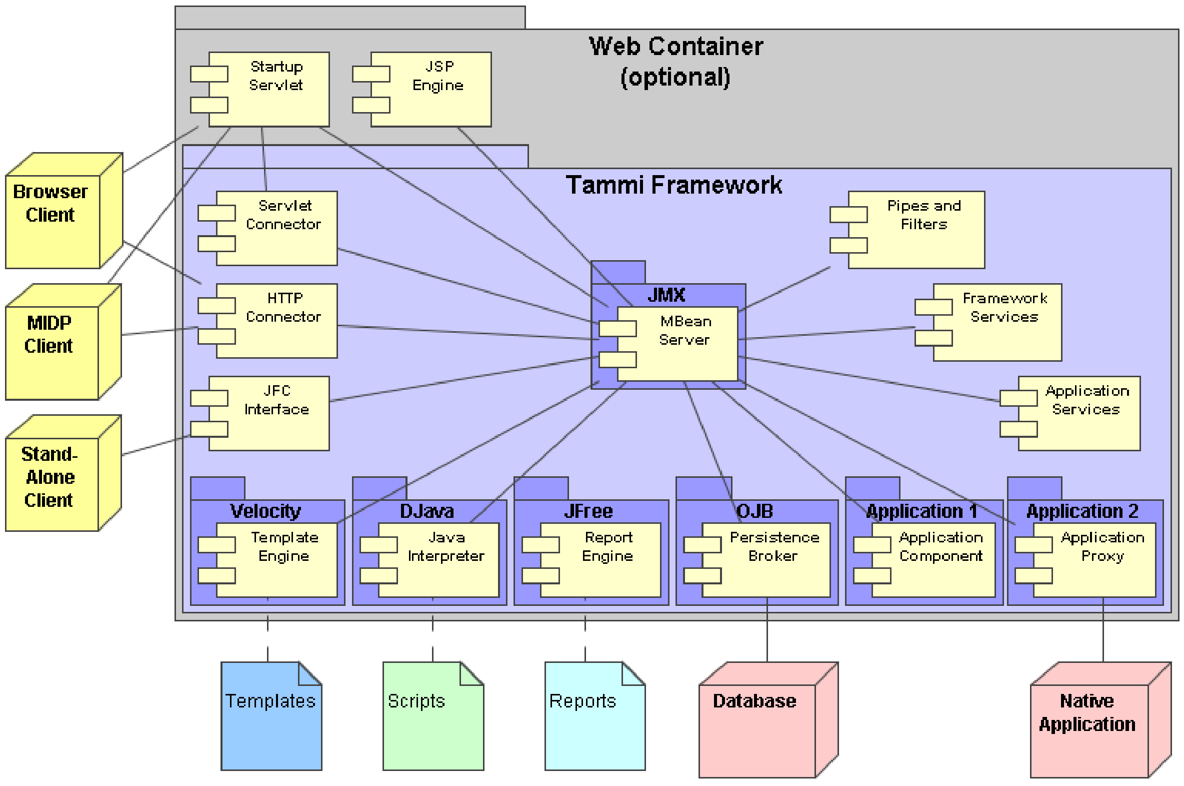

Figure 3.2 The deployment diagram

3.4 Dynamic Configuration

Tammi starts up as an empty run-time container that is configured for a specific application and purpose with configuration scripts. The configuration model has been adopted from that of the servlet technology [Servlet], by extending it to cover all aspects of internal components of Tammi, applications plugged into Tammi during startup, and run-time configuration of both new and loaded components.

The goal of dynamic configuration is to exploit efficiently common functionality provided by the framework without a need to develop, build and maintain several versions of the framework. This is especially useful in application domains where requirements, specifications, applications and components are changing frequently, but the overall architecture remains the same.

Extensive dynamic configuration sets specific requirements for configuration management. It must be capable of changing almost any component property both during startup and run-time. Correspondingly, component implementations must accept configuration changes during run-time. Tammi supports property files and BSF scripts for configuration. Applications follow the MVC design pattern to support independent configuration of user interface templates, too.

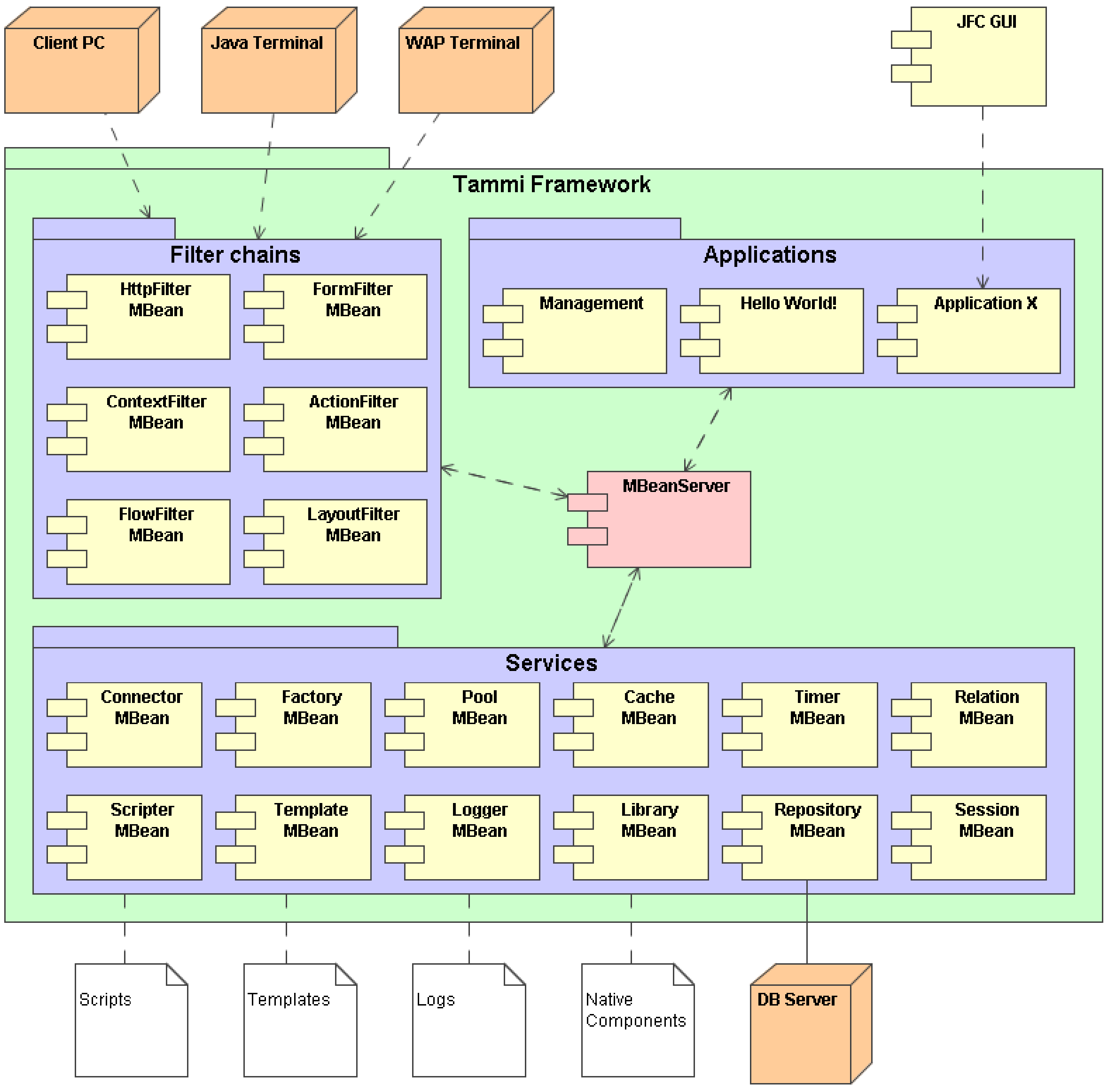

Figure 3.3 The configuration diagram

3.4.1 Property Files

A Java property file specifies a set of permanent properties represented as key – value pairs. Tammi supports loading of property files to Configuration objects, which perform desired type conversions and maintain the properties during run-time. Configurator can treat the properties loaded to Configuration as attributes of another MBean and set their values through the exposed MBean interface of the corresponding MBean.

Configurables extending the Configurable class inherit MBean specific property file support, but implementations must handle the effect of changes in property values to their behavior.

3.4.2 Bean Scripting Framework

The Bean Scripting Framework [BSF] is architecture for incorporating scripting into, and enabling scripting against, Java applications and applets. Using BSF, an application can use scripting, and become scriptable, against any BSF-supported language. When BSF supports additional languages, the application will automatically support the additional languages. Scripts in any BSF-supported language can be run directly on the command line as well.

The list of scripting languages supported by BSF is ever growing. Both Java-implemented languages (such as Netscape Rhino and Jacl) as well as non-Java ones such as Tcl and Perl will be supported. On Win32 platforms, active scripting languages including VBScript and JScript are supported. Scripting language support is provided by language specific BSFEngine interface implementations.

BSF is integrated in Tammi through Scripter. The recommended scripting language is DynamicJava [DJava].

3.4.3 DynamicJava

DynamicJava [DJava] is a Java source interpreter. It executes programs written in Java, like described in the Java Language Specification, in addition with scripting features.

DynamicJava is suitable for loading different application configurations to Tammi both during startup and run-time. Tammi processes automatically any DynamicJava scripts located in its startup and shutdown directories. However, configuration scripts should not be used for programming new functionality for application components.

3.4.4 Management Interface

The management interface of Tammi provides a user interface supporting operations with which to create or unregister any MBean. The interface lists registered MBeans, which can be modified by changing values of their public fields. The interface allows also invoking of public methods of MBeans. All primary data types and data types with registered CustomConverter are supported.

Saving of changes made through the management interface is currently not supported.

3.5 User Interface

The user interface of plug-in applications is based on the page-building model of Turbine [Turbine]. The user interface page is typically formed by a static portion described with a markup language in a user interface template, and a dynamic portion provided by content producer components of the application.

The selection of the template for a client request is based on a template parameter included in the request URL or the body of the request. If such a parameter is not specified or not found, the default template will be applied. Templates are parsed by TemplateEngines. The recommended template engine is Velocity [Velocity], but FreeMarker [FM] and JSP [JSP] are also supported.

The content producers place the dynamic portion of the user interface page into a Context object, the contents of which are made available to template engines before they parse the template. Actions, implementing also the Task interface, may be applied to load content into the context, as actions are activated before builders. Applications may extend the Action class to customize content generation.

Templates may form nested hierarchies supporting building of more complicated user interfaces. The layout model of Turbine is based on a layout template specifying the layout for a target template and any number of additional templates. The target template provides the actual request specific content, or screen, within the page. Additional templates provide navigation support or other generic content within the page.

The target template is typically formed by one or more form templates, further consisting of input and output control templates.

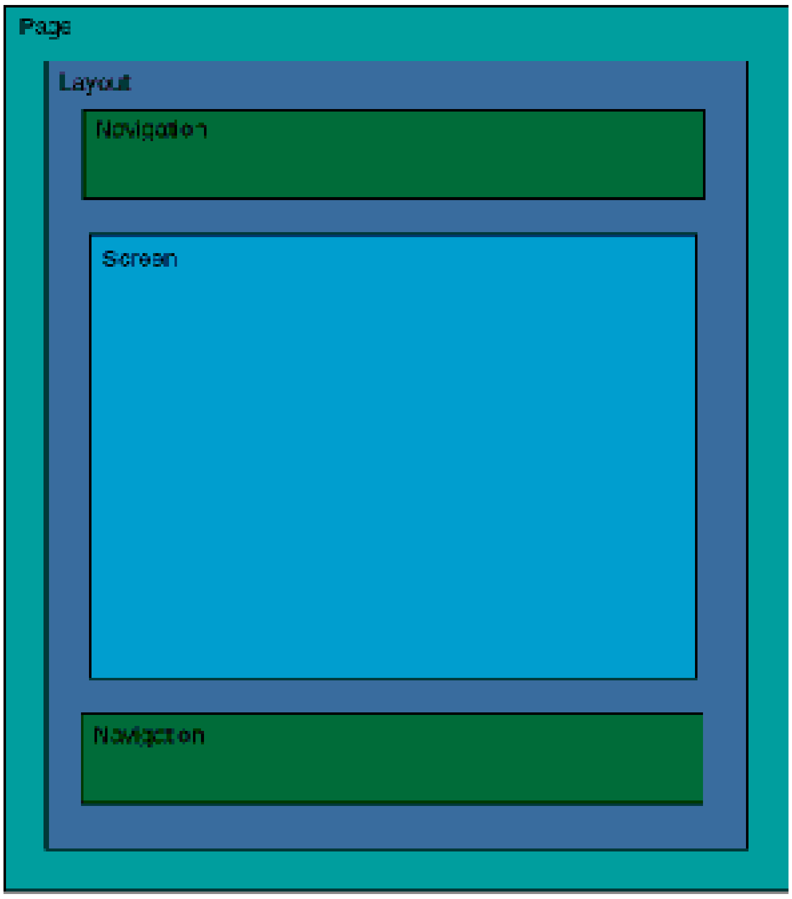

Figure 3.4 The page layout model [Turbine]

3.6 Filter Chain

Tammi applications follow the MVC design pattern, which aims at separating presentation from content within applications:

- Model – content producer components implemented as MBeans
- View – one of the template engines parsing a user interface template
- Controller – a filter chain selecting the template and content producers for each of the client requests

Pipe describes a chain of Filters that should be processed sequentially. Each Connector receiving client requests is connected to one pipe. The connector passes the flow of requests to the pipe, and it is required that a filter somewhere in the pipe (often the last one) must process the request and create the corresponding response, rather than trying to pass the request on to the next filter in the chain [Servlet].

The request flow proceeds via calls to the doFilter method of pipes and filters. The method has three parameters: a request, a response and a chain. The request and response are implementations of the corresponding interfaces in the javax.servlet package. The chain is an implementation of the HttpFilterChain interface extending the javax.servlet.FilterChain interface. It carries information about the request flow itself between pipes and filters. Filters can get a reference to the current pipe through it and override the original request and response objects by passing their own implementations to the chain. Both Pipe and Filter interfaces, implemented by pipes and filters correspondingly, extend the javax.servlet.Filter interface.

Other information between filters is passed via request and session attributes. Any object expecting to be such an attribute can implement a RequestBindingListener or SessionBindingListener interface to be notified about binding/unbinding events.

Any number of filters in the filter chain can implement the BranchingFilter interface directing the request flow to one of alternate pipes. Typically, the last filter in the pipe is a template filter selecting the correct template and invoking a template builder building the final user interface page. The preceding filters refine request data and load content into the context to be used by the template engine during parsing of the template.

Note that the doFilter method of the HttpFilterChain object is called recursively, although presented in the figure below as if there were two separate HttpFilterChain instances. Also, there can be any number of filters in the filter chain, but only one is presented in the figure.

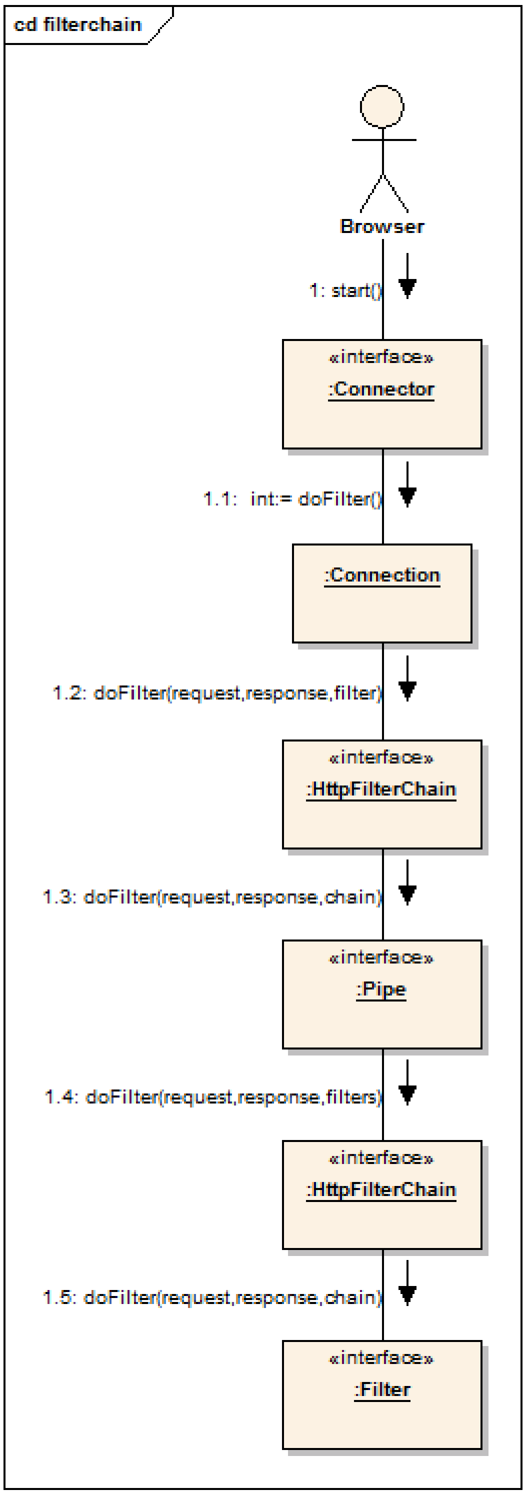

Figure 3.5 The filter chain mechanism

3.7 Component Lifecycle

To integrate independent components and dynamic configuration to a working system, a framework must be provided around them. The framework is a set of interfaces, helper classes, and services that support development of MBean based components, and define guidelines on how to write code that plugs into the framework.

The framework allows:

1 Partition of shared functionality into generic MBeans that are used through their public interfaces and not through their actual implementations

2 Reduced effort in MBean development since they become plug-ins for the framework and share functionality provided by other MBeans

3 Creation of a common server for all MBeans included in the framework

4 Centralized configuration management

The control over components included in Tammi happens through the lifecycle interfaces of their MBeans defining their common lifecycle.

3.7.1 Lifecycle Interfaces

The lifecycle interfaces define methods applied by the framework to control the lifecycle of MBeans. These interfaces are not meant to be used alone as MBean interfaces, but can be extended to be a part of one.

Manageable

Defines methods to be called when the implementing MBean is registered to and unregistered from the MBean server.

ReferableMBean

Defines methods supporting a direct reference to the implementing MBean to achieve competent performance.

AdaptableMBean

Defines methods for adapters being capable of managing objects that are not originally implemented as MBeans.

Configurable

Defines methods supporting configuration through property files.

AccessController

Defines methods restricting access to the implementing MBean.

Startable

Defines start and stop methods controlling the state of the implementing MBean.

Executable

Defines methods allowing the implementing MBean to run in its own thread.

CacheClient

Defines methods to monitor and control the internal cache of the implementing MBean.

Recyclable

Defines dispose and recycle methods to be called when the implementing object is pooled and reused.

Poolable

Extends the Recyclable interface by allowing the implementing object to control reuse by itself.

For example, an object implementing the Manageable, Startable and Executable interfaces might have a lifecycle like the one presented below.

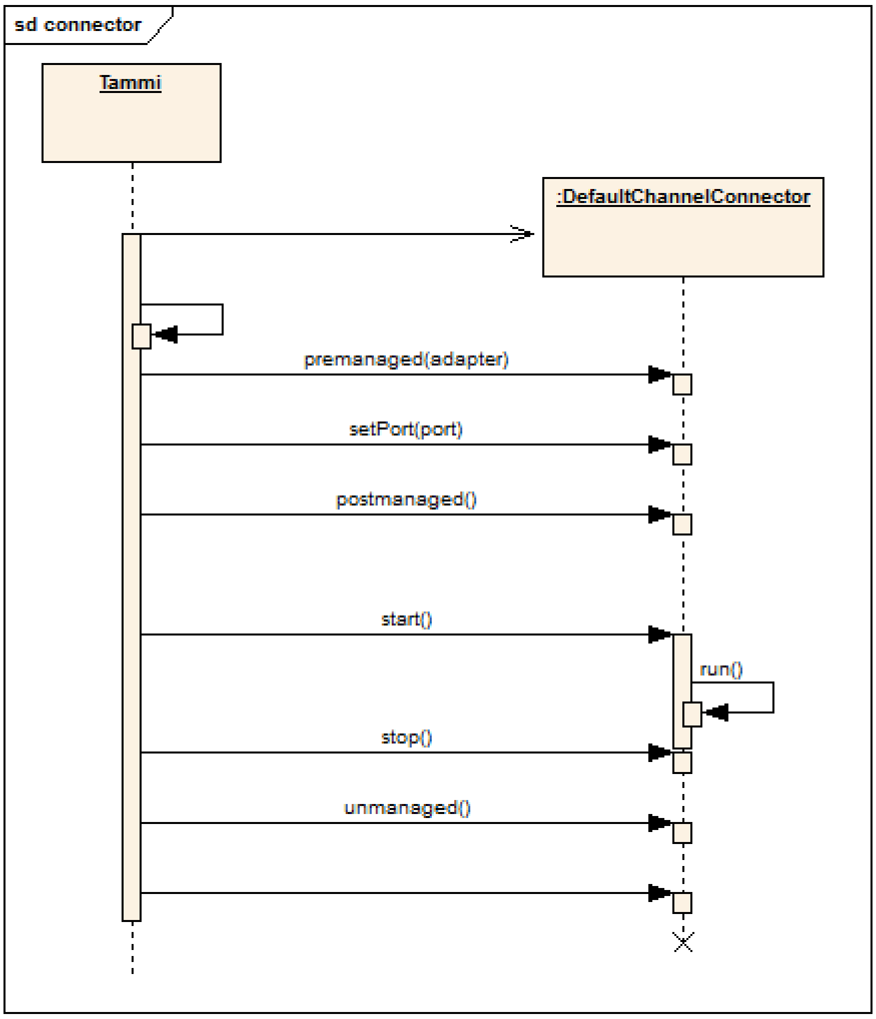

Figure 3.6 The lifecycle of a connector

Note that the methods may be called several times during the lifecycle.

3.8 Persistence Layer

Tammi's persistence layer is based on an approach that maps objects to persistence mechanisms in such a manner that simple changes to the relational schema do not affect the object model. This approach makes objects independent on the persistence layer allowing development of large-scale, mission critical applications. The disadvantage is somewhat slower performance when compared to more straightforward solutions.

3.8.1 Persistence Broker

The implementation of the persistence layer is derived from the persistence broker API of the Object/Relational Bridge [OJB]. A persistence broker implementing the Persister interface provides methods for retrieval, storage and deletion of objects. A persistence framework specific implementation of the broker can be retrieved from Factory. In addition to OJB, the persistence layer currently supports Castor XML Binding Framework [Castor].

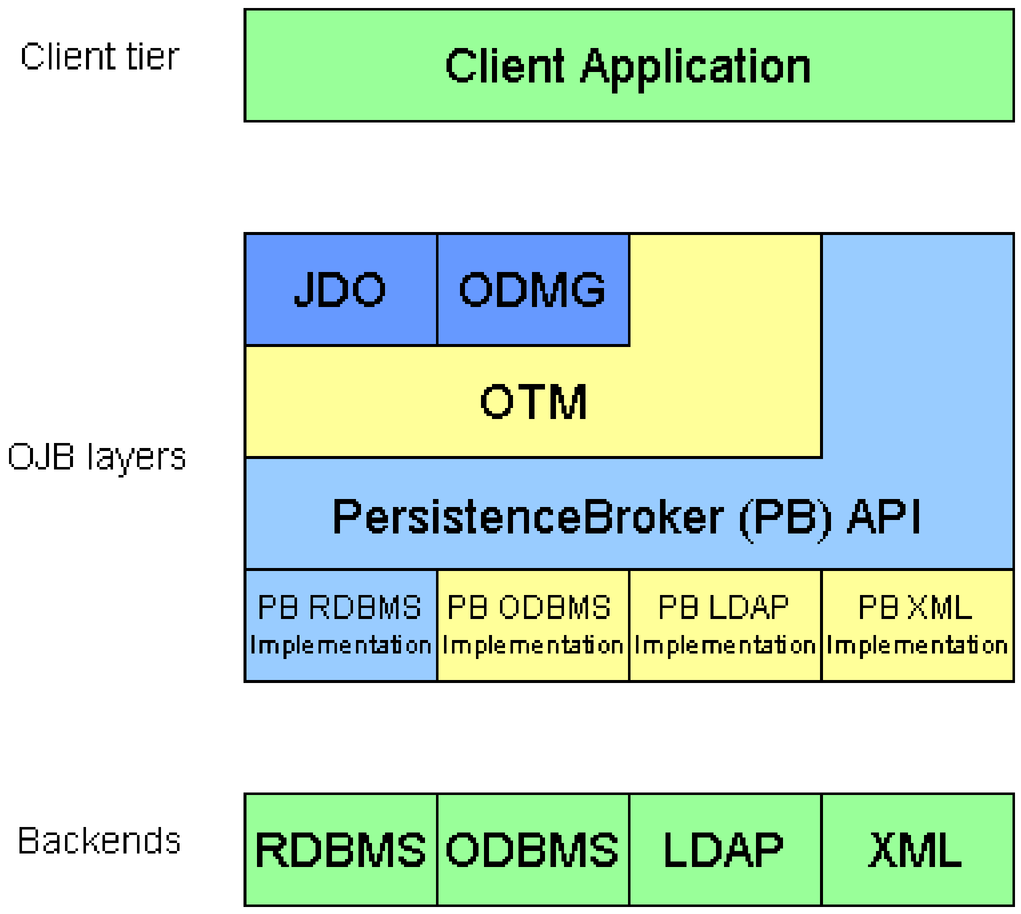

Figure 3.7 The PB API of OJB [OJB]

3.8.2 Dynamic Mapping

Configuration of the mapping between the object model and the database schema depends on the persistence framework to apply. OJB maintains a class descriptor repository that can be configured either directly or via XML files. The configuration may take place both at startup and during run-time. The mapping is defined between named application classes and relational database tables.

In addition to a static mapping between the classes and tables, Tammi supports Variables, which can dynamically adapt to hold any number of new attributes of specified types. The classes corresponding to a particular set of attributes may also be created and loaded during run-time. By applying run-time Variables makes it possible to develop extremely dynamic persistence solutions.

The most dynamic persistence approach supported by Tammi is based on the Common Warehouse Model defining a generic relational meta model of any SQL based relational database schema.

By designing and implementing the data model of the application through the relational meta model and Variables, the persistence of business objects can be fully managed during run-time allowing dynamic changes to the model without application rebuild, too.

4 Available Packages

Tammi is divided into main packages deployed as separate plug-in applications:

- `acorn` contains miscellaneous classes not MBeans themselves but to be utilized by any MBean included in Tammi
- `core` contains foundation MBeans providing base services and classes both for Tammi itself and plug-in applications
- `root` contains MBeans implementing the persistence layer of Tammi
- `leaf` contains JFC MBeans supporting dynamic building of localized graphical user interfaces for desktop applications
- `spray` contains MVC MBeans implementing the filter chain mechanism and page-building model of Tammi
- `sprig` contains MBeans providing report and chart tools both for JFC and HTML based user interfaces
- `gate` contains MBeans providing a meta-model of the other MBeans to dynamically generate new persistent capable applications
- `flex` contains MBeans for building rich internet applications with Flex [Flex]
- `kit` contains a set of build tools

The main packages are divided into functional sub-packages as specified in the Java Coding Standard for Tammi [Coding].

In addition, Tammi distribution may include the on-line manual as a separate plug-in application.

4.1 Acorn Packages

Acorn packages include utilities, tools and helper classes shared by MBeans. The HTTP package contains a static HTTP parser, the I/O package provides stream and file classes for specific purposes, the MIME package provides support for content types, the net package helps URL encoding and decoding, the security package provides tools related to security and the util package contains implementations of thread-safe collections.

The assembler package enables users to represent Java objects in byte code as helper classes with which to analyze, create and manipulate binary classes.

4.2 Core Packages

Core packages contain MBeans implementing base services and classes grouped according to functionality they provide. Typically, services support several instances but one of them can be the default one. The default instance is usually registered in the default JMX domain.

4.2.1 Base Package

The base package defines interfaces and implements classes to be applied to MBeans implementing lifecycle interfaces. The ReferableMBean interface defines methods with which to refer to its implementing MBeans directly to achieve competent performance provided by direct references to objects while maintaining the dynamic features of the distributed JMX architecture.

The Referable class is the default implementation of ReferableMBean, which can be applied as a helper class to extend when implementing application specific MBeans. When adapters are preferred over extensions, the Adaptee class provides base methods for accessing the framework services.

Startable maintains state, Registry and Container act as containers of other MBeans. Domain maintains the default domain and Loader supports dynamic class loading by applying the helper classes in the acorn.assembler package.

LogException can be thrown to create the corresponding error log entry automatically. Logger defines logging functionality, and the static methods of its abstract implementation provide basic tracing facilities.

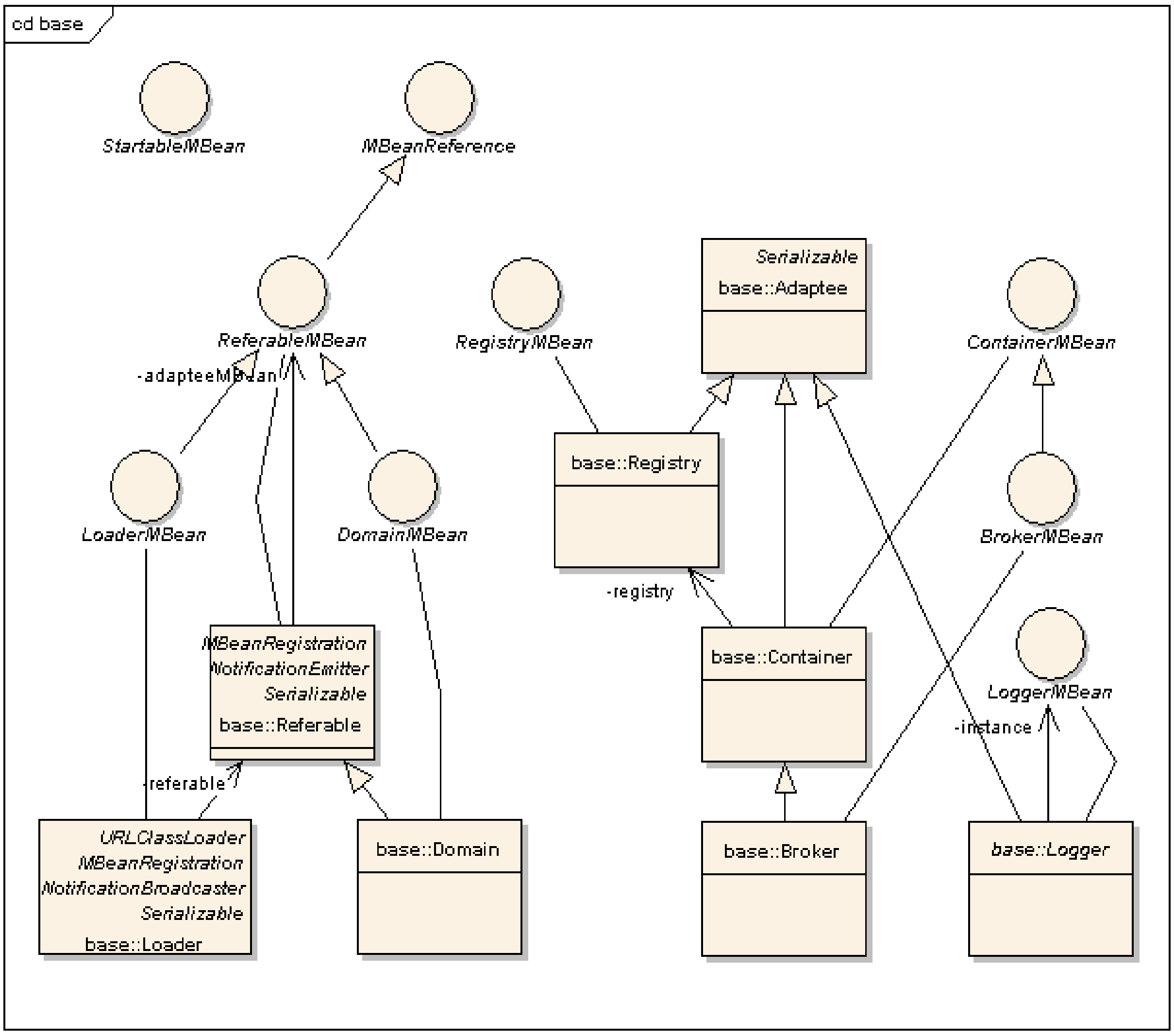

Figure 4.1 The base package

The base package contains also Broker and its default implementation for resolving registered MBeans.

4.2.2 Cache Package

The cache package contains MBeans providing caching capabilities for different purposes.

Cache supports expiration of old entries in its enclosed java.util.Map implementation. ResourceCache extends Cache by adding methods to cache resources.

SharedCache defines an interface for a set of caches within different domains and sharing the same expiration mechanism. Any client needing a cache can create a new namespace by providing a unique domain parameter in access methods of SharedCache. Objects without the domain parameter are cached in the default domain. The clients of SharedCache are responsible for any naming conflicts.

CacheClient is an interface to be implemented by any MBean maintaining a cache and allowing external control over it, e.g. by CacheMonitor.

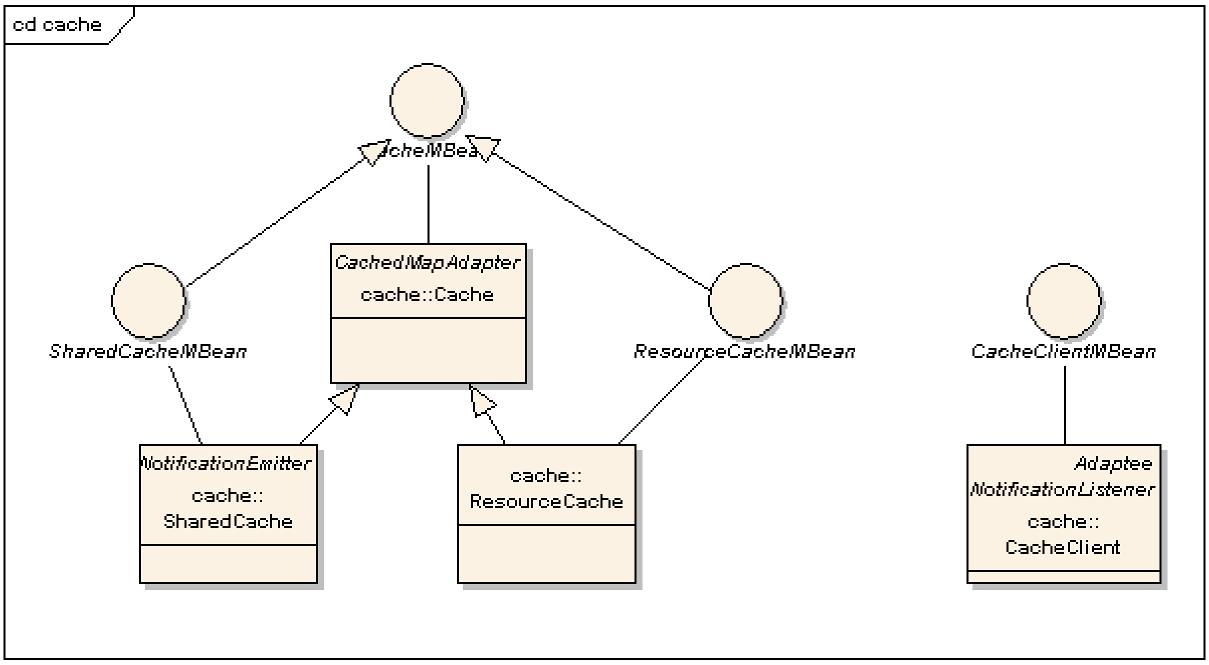

Figure 4.2 The cache package

4.2.3 Config Package

The config package provides configurable extensions to Factory, Converter or any customized MBean. In addition, it contains different versions of Configurator for external configuration of MBeans.

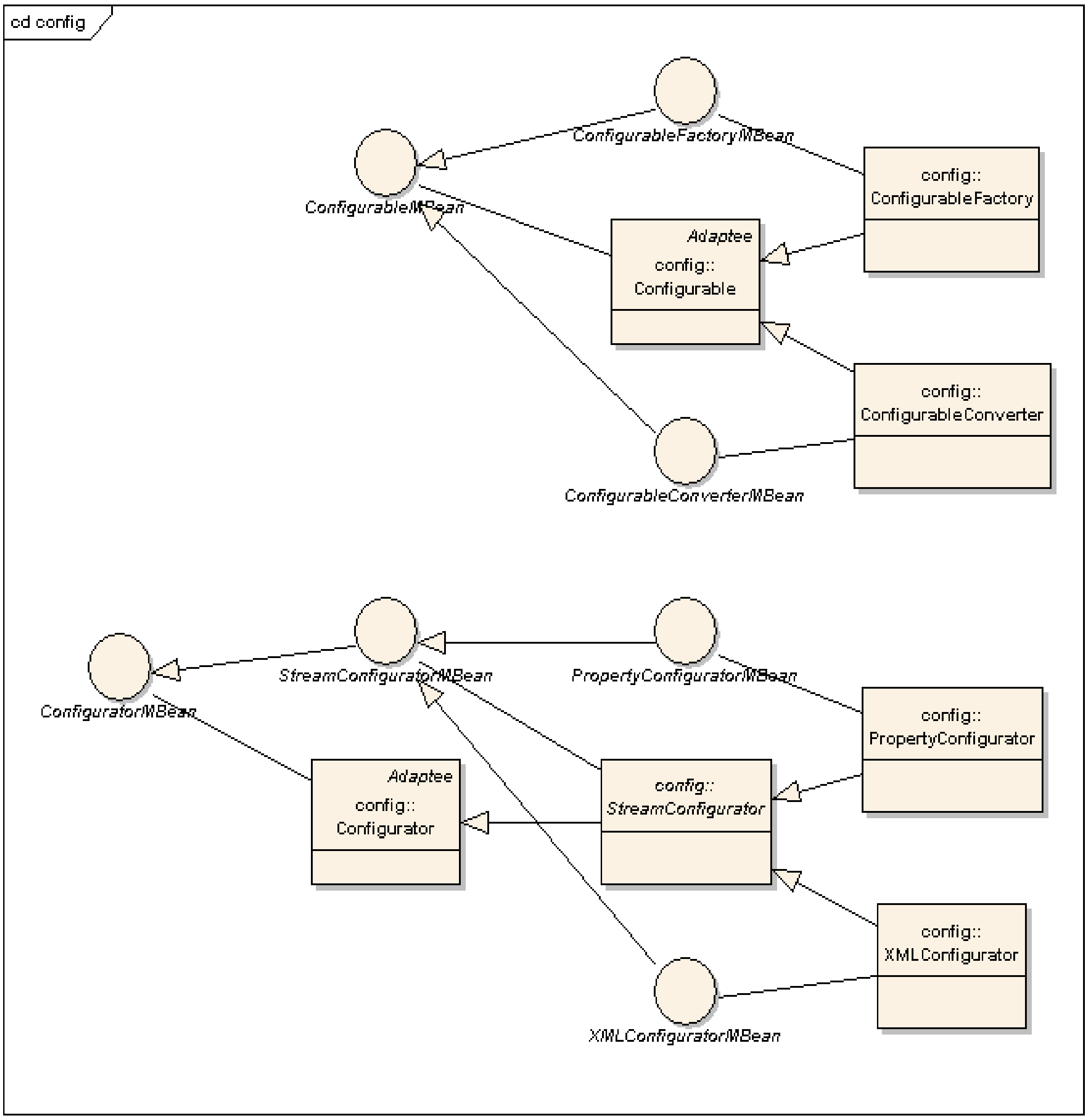

Figure 4.3 The config package

4.2.4 Converter Package

The converter package defines converters to perform translations from different class types to others. Converters between java.lang.String and primitive types and some common class types are included in the framework.

Implementations of ObjectConverter may provide additional converters.

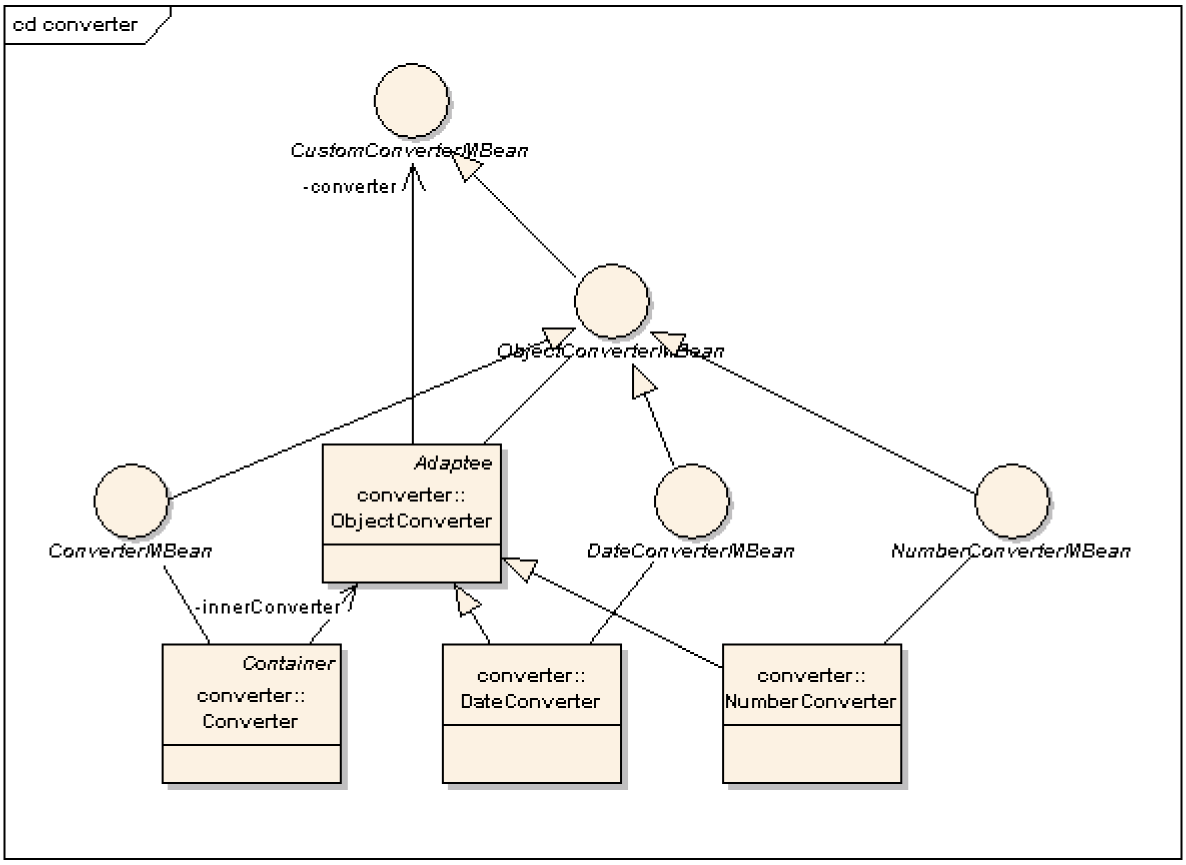

Figure 4.4 The converter package

4.2.5 External Package

The external package contains Library providing a JNI mechanism to dynamically access native libraries from within Tammi applications.

4.2.6 Factory Package

The factory package contains implementations of object factories and object pools.

Factory instantiates objects from the given class name using either the given class loader or an applicable one found from the class loader repository. The default class loader will be used if neither one is specified.

Factory provides the following benefits compared to Class.forName():

- Support for parameters in constructors
- Internal class loader repository in the MBean server
- Optional class specific factories to be used for customized instantiation
- Integration with Pool supporting recycling of objects created by the factory

Class specific factories must implement ObjectFactory and register themselves to Factory in order to replace the default factory when applicable; e.g. to instantiate XML parsers, SSL sockets, etc, which require specific instantiation not supported by the default factory.

Pool extends Factory by adding support for pooling objects instantiated from the given class name or java.lang.Class object reference. Pooling of objects stabilizes memory consumption and reduces garbage collection making response times in server applications more predictable.

When a new instance is requested from Pool, it first checks its pool if one is available. If the pool is empty, a new object will be instantiated from the given class. If the class is specified by its name, the request to create an instance will be forwarded to Factory.

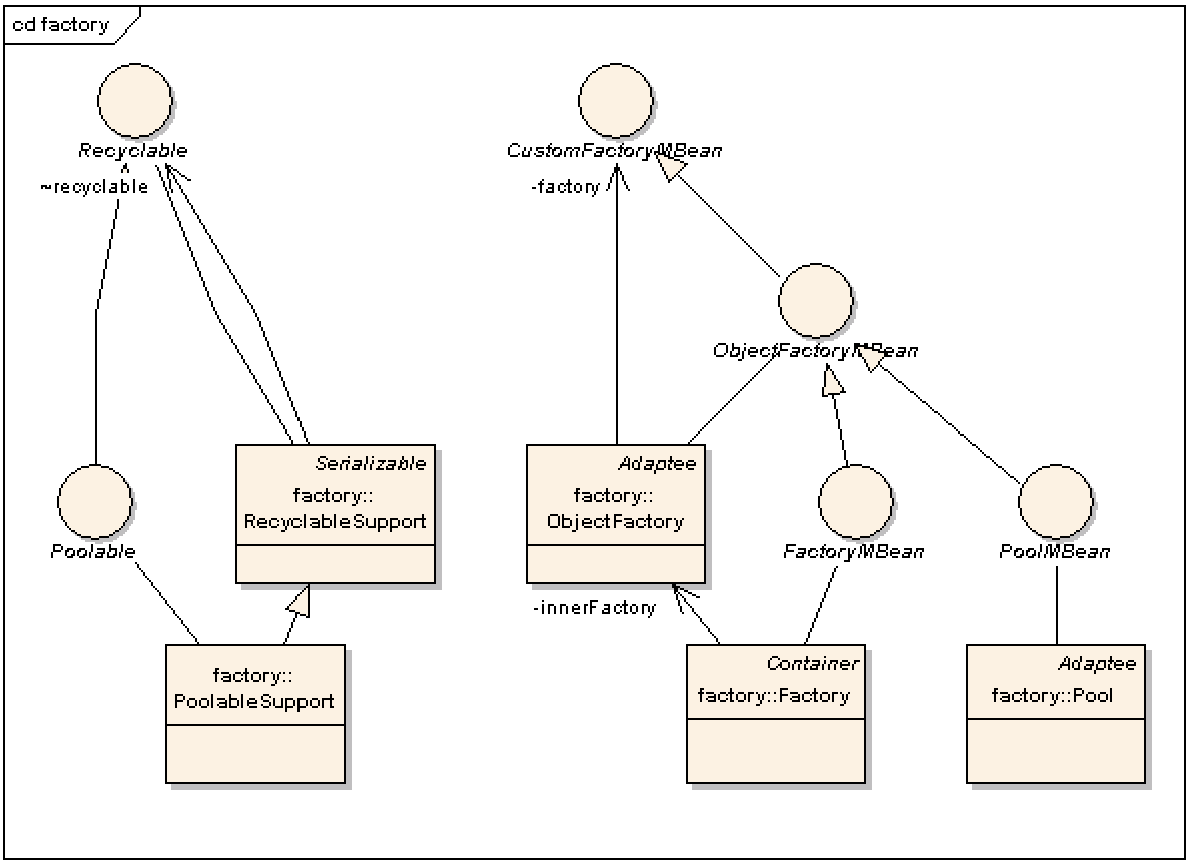

Figure 4.5 The factory package

For pooled objects implementing the Recyclable interface, a recycle method will be called when they are taken from the pool, and a dispose method will be called when they are returned to the pool. Implementations of the methods should initialize and release the pooled instances correspondingly. Objects that do not implement the interface can also be pooled if they do not need to perform any specific actions during pooling. The RecyclableSupport class can be extended to get a minimal implementation of the Recyclable interface.

4.2.7 IO Package

The io package contains PathFinder and its default implementation for resolving path names.

4.2.8 Locale Package

ContentTypeMap maintains mappings between MIME types and the corresponding file name extensions, and between locales and the corresponding character encodings. The mappings are typically defined in property files located in user's home directory, in the Java home directory, or in the jar archive of the current class.

The mappings between locales and the corresponding character encodings are specified using the Java property file syntax, where the locale specification is the key of the property and the charset is the value of the property. The locale specification consists of three items:

```
<lang>_<COUNTRY>_<VARIANT>
```

The variant can be whatever is appropriate for the application, like a markup language specification, a browser specification, etc. However, it should not equal any of the language and country specifications. ContentTypeMap looks for charsets using the following search order:

```
<lang>_<COUNTRY>_<VARIANT>=<charset>
<COUNTRY>_<VARIANT>=<charset>
<lang>_<VARIANT>=<charset>
<VARIANT>=<charset>
<lang>_<COUNTRY>=<charset>
<COUNTRY>=<charset>
<lang>=<charset>
```

ContentTypeMap contains defaults for several language mappings and more specific ones can be specified in an optional property file, e.g. wml=UTF-8. It caches results of the search, which should guarantee sufficient performance.

The mappings between MIME types and the corresponding file name extensions are specified using the same syntax as the `mime.types` file of the Apache Server, i.e.:

```
<mimetype> <ext1> <ext2>...
```

ContentTypeMap contains defaults for most common MIME types, like text/plain, text/html, text/x-hdml, text/vnd.wap.wml, image/gif and image/jpeg. More specific ones can be specified in an optional MIME types file.

ResourceFinder is a localized resource resolver. It searches for localized property files from resource directories and the current class path by applying the same search rules as ContentTypeMap. If no resources are found, it applies the search mechanism of java.util.ResourceBundle.

4.2.9 Logger Package

The logger package contains concrete implementations of Logger based on different logging libraries.

4.2.10 Mail Package

The mail package provides a simple e-mail service.

4.2.11 Model package

The AdaptableMBean interface extends javax.management.modelmbean.ModelMBean defining a generic adapter for any Java class to be managed without implementing the corresponding MBean interface. ReferableModelMBean provides a default implementation based on the JavaBeans specification and using introspection to expose public attributes and methods of the adapted class for management.

4.2.12 Monitor Package

The monitor package contains CacheMonitor monitoring caches maintained by other MBeans and Finalizer maintaining a shared reference queue.

4.2.13 Naming Package

The naming package contains factories for directory contexts.

4.2.14 Net Package

The net package contains factories and extensions for basic sockets and other net classes provided by the standard Java net package.

4.2.15 Realm Package

The realm package contains MBeans and helper classes for maintaining authenticated principals.

4.2.16 Scripter Package

The scripter package contains Scripter for script managers, its BSF based default implementation and scripting language engines. The scripts can be in files or passed through an input stream to the script manager.

4.2.17 Security package

The security package contains MBeans and helper classes for access control, secure communication and encryption.

AccessController can be implemented by MBeans controlling resources protected by permissions allowing access for specific principles only. The controller supports both allowed and denied permissions. The RegexPermission class extends java.security.Permission by identifying proteced resources with regular expressions instead of explicit names.

4.2.18 Startup Package

The startup package contains Startup responsible for starting up and shutting down the Tammi system. The startup sequence is described in more detail in Configuration Notes.

4.2.19 Thread Package

The thread package contains MBeans supporting execution of multi-threaded tasks.

All clients running tasks in separate threads should apply Executor. By configuring Executor instances for different purposes, the thread priorities can be managed without interference between tasks.

Scheduler executes time-based tasks providing similar functionality as common system-level utilities, such as `at` in Unix. Scheduler maintains a task queue that may be used to execute java.lang.Runnable commands in any of three modes:

- Absolute (run at a given time)
- Relative (run after a given delay)
- Periodic (cyclically run with a given delay)

Timeout is a specific scheduler for executing timeout commands for I/O operations and other tasks with limited time.

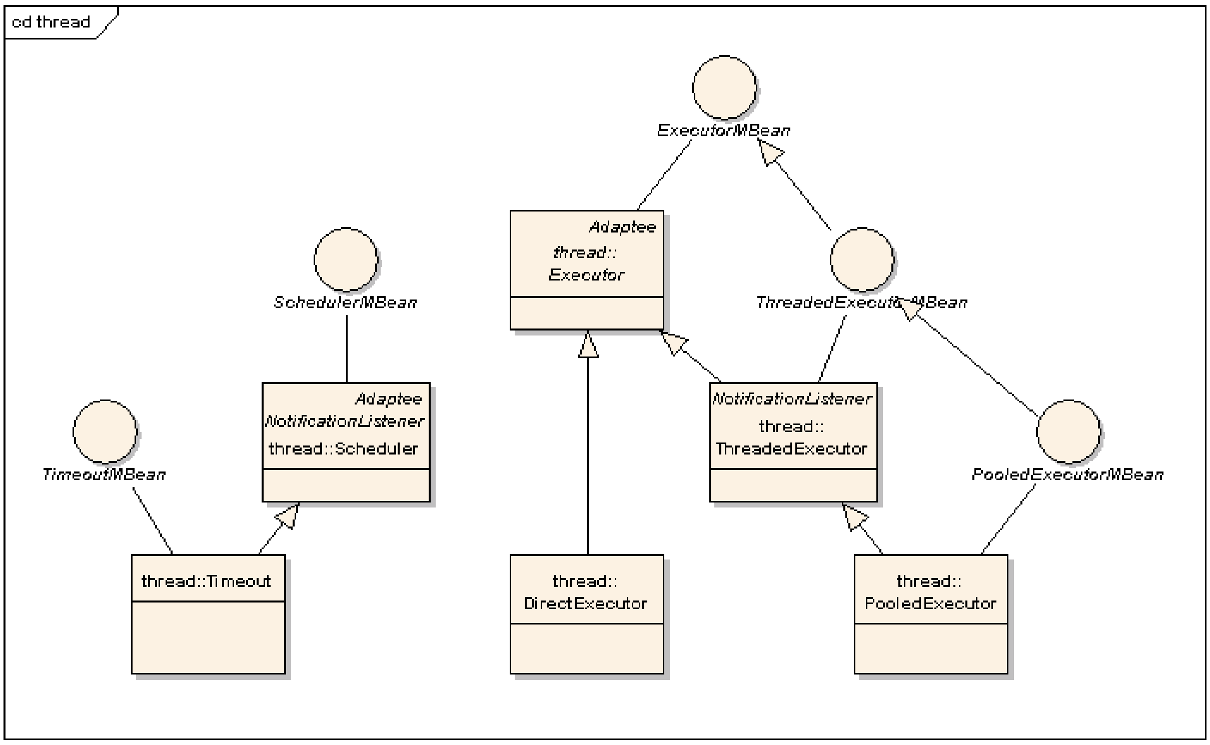

Figure 4.6 The thread package

4.2.20 Util Package

The util package contains various utilities and helper classes for implementing MBeans.

4.2.21 RT Package

The rt package contains extensions to javax.management.DynamicMBean, which bring run-time dynamics to managed data structures.

Variable extends ReferableMBean with DynamicMBean methods. VariableX provides a default implementation, which can maintain any number of dynamic attributes defined during run-time.

VariableRegistry can be applied to register predefined attributes for constructing new Variables during run-time.

4.2.22 XML Package

The XML package contains factories for XML parsers.

4.3 Root Packages

Root packages implement the persistence layer of Tammi. The current implementation supports two persistence frameworks: the Object/Relational Bridge [OJB] provides persistence for Java objects against relational databases, while the Castor XML Framework [Castor] maps Java objects to XML documents.

4.3.1 DB Package

The db package contains the Persister interface through which to access an implementation specific persistence framework. Factory may be applied to create instances of the implementing class.

RepositoryClient, MultiRepositoryClient and their default implementations provide common interfaces and support classes for MBeans to configure an appropriate persistence layer for their internal requirements.

4.3.2 Locale Package

The locale package contains a persistent resource bundle implementation to maintain localized resources in a database.

4.3.3 OJB Package

The ojb package contains the OJB [OJB] specific Persister implementation and the corresponding factory.

4.3.4 Realm Package

RepositoryRealm in the realm package defines an interface to realms maintaining user data in a repository. Its default implementation supports both id-based and simple persistent users and roles.

4.4 Leaf Packages

4.4.1 JFC Package

The jfc package contains a JFC object factory and a few extended JFC classes.

4.5 Spray Packages

Tammi's filter chain mechanism is based on the Filter interface extending the javax.servlet.Filter interface. The base filters implementing the interface are located in the filter package. Other packages in spray contain mainly specific filter implementations but also connectors for accessing the filters.

4.5.1 Authenticator Package

The authenticator package contains authenticator filters for different authentication schemes specified by the HTTP protocol and other specifications.

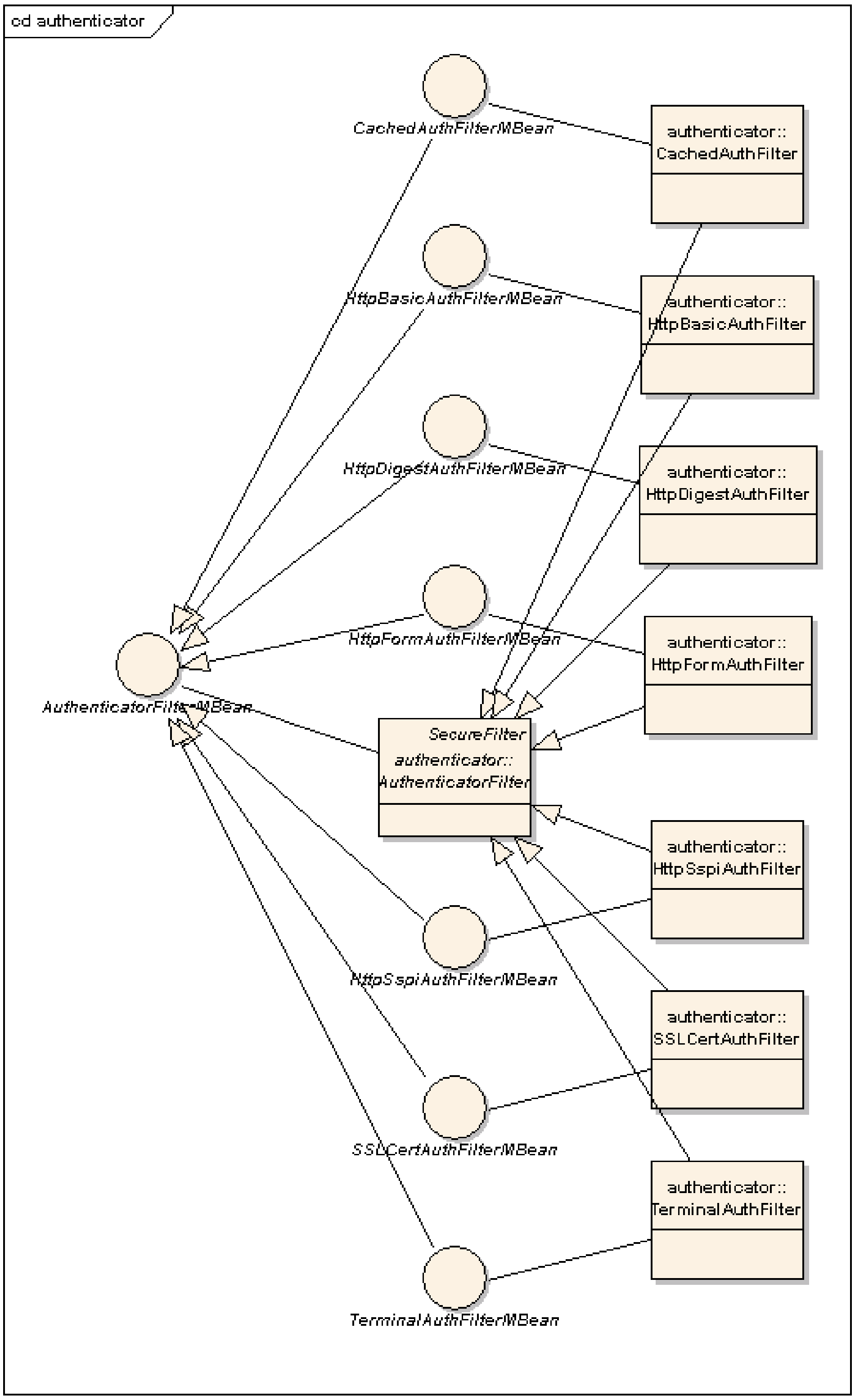
cd authenticator
CachedAuthFilterMBean
HttpBasicAuthFilterMBean
HttpDigestAuthFilterMBean
HttpFormAuthFilterMBean
AuthenticatorFilterMBean
HttpSspiAuthFilterMBean
SSLCertAuthFilterMBean
TerminalAuthFilterMBean
authenticator:: CachedAuthFilter
authenticator:: HttpBasicAuthFilter
authenticator:: HttpDigestAuthFilter
authenticator:: HttpFormAuthFilter
SecureFilter
authenticator:: AuthenticatorFilter
authenticator:: HttpSspiAuthFilter
authenticator:: SSLCertAuthFilter
authenticator:: TerminalAuthFilter

Figure 4.7 The authenticator package

4.5.2 Connector Package

The connector package contains connectors and connections for communication with client side applications.

Connector is a generic interface for connectors. It is not Pipe or Filter itself but one is typically attached to it. Connectors serve as entry points listening to input from different sources and passing received requests to pipes. Connectors apply a connection based request processing and load balancing mechanism. Further processing of requests is delegated to filters in attached pipes.

4.5.3 Engine Package

The engine package contains MBeans for template processing.

TemplateEngine is a generic interface implemented by specific template engines.

ContextFilter creates a hierarchical context for tools accessible by subsequent filters in the same pipe and especially by templates. Context tools can be any Java objects, but those implementing the ContextListener interface will be notified about binding/unbinding events and new requests.

4.5.4 Filter Package

The filter package contains MBeans and helper classes implementing the filter chain mechanism.

Pipe defines an interface for pipes and Filter for filters. KeyFilter applies a filter specific key to request parameters to filter requests. SecureFilter protects a filter with an access controller.

TerminationFilter marks the end of the pipe. ClosingFilter closes the response after the request has been processed by preceding filters. FilterException can be thrown when filtering of the request fails.

BranchFilter is an interface to filters directing the request flow to one of several alternative pipes and filters.

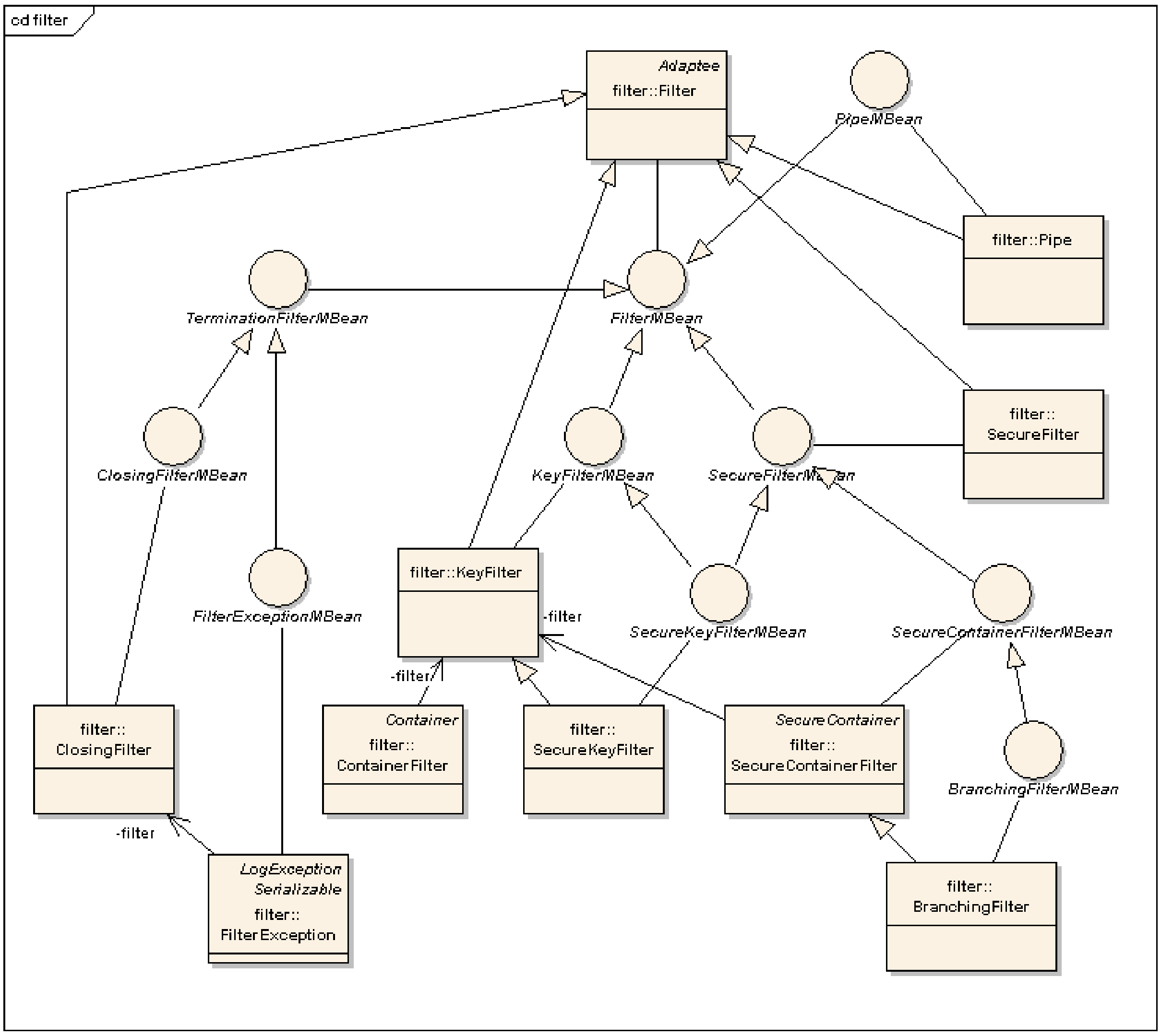

Figure 4.8 The filter package

4.5.5 Freemarker Package

The freemarker package contains an implementation of the FreeMarker template engine [FM] and its helper classes.

4.5.6 Loader Package

The loader package contains the Task interface and its abstract implementation.

TaskLoader loads, caches and executes specific Task implementations, i.e. TemplateActions and FlowStepActions, requested by clients.

4.5.7 Media Package

The media package contains MBeans for content management.

ContentFilter processes static files and CGIBinFilter executes CGI scripts.

ServiceFilter is a service-level branch filter choosing an appropriate service pipe or filter for incoming requests based on mappings maintained by its configuration.

4.5.8 Protocol Package

The protocol package contains filters for parsing protocol specific requests.

HttpFilter parses HTTP requests, HttpRelayFilter relays requests to remote servers or clusters, and IpMaskFilter allows or blocks requests from masked IP addresses.

4.5.9 Remote Package

The remote package contains RMI based implementation of Connector supporting remote procedural access to Tammi applications.

4.5.10 DB Package

The db package provides RepositoryFilter and its default implementation to access Tammi's persistence layer directly from templates. RespositoryTool must be configured to ContextFilter to exploit the filter.

4.5.11 Servlet Package

The servlet package contains MBeans for running Tammi as a servlet under some servlet container including an implementation of the JSP template.

4.5.12 Session Package

The session package provides support for session management. The state of a request is mostly maintained by the request itself, but sessions can be used to maintain state information for consecutive requests from the same user.

SessionManager produces and manages sessions according to the servlet specification [Servlet]. It is not intended to be accessed directly but through the request.

4.5.13 Template Package

The template package contains page-building MBeans.

PageFilter renders single page templates while LayoutFilter complies with the page-building model of Turbine [Turbine]. Template engines to be used by the filters are registered with template file name extensions determining the specific engine to apply for rendering the corresponding templates.

FormFilter renders HTML forms with the aid of FormTool.

FlowFilter renders predefined page sequences with the aid of FlowTool.

TaskFilter invokes request specific tasks.

4.5.14 Terminal Package

The terminal package contains MBeans for user terminal, user agent and locale specific request processing.

4.5.15 Velocity Package

The velocity package contains an implementation of the Velocity template engine [Velocity] and its helper classes.

4.6 Sprig Packages

Root packages implement reporting and charting tools. The current implementation supports two reporting frameworks: JasperReports and JFreeReport.

4.6.1 Report Package

The report package contains reporting tools.

ReportEngine and its abstract implementation provide the base for reporting.

4.6.2 Chart Package

The chart package contains charting tools.

ChartGenerator and its abstract implementation provide the base for charting.

4.6.3 JFree Package

The jfree package contains the JFree specific implementations of ReportEngine and ChartGenerator.

4.6.4 Jasper Package

The jasper package contains the Jasper specific implementation of ReportEngine.

4.7 Third Party Libraries

Tammi utilizes a set of third party libraries including the following in alphabetical order:

- Apache Ant Build Tool [Ant]
- Apache DB Object/Relational Bridge [OJB]
- Apache Jakarta Bean Scripting Framework [BSF]
- Apache Jakarta Commons [Coms]
- Apache Jakarta Log4J Logging Package [Log4J]
- Apache Jakarta Velocity Template Engine [Velocity]
- Concurrent Programming in Java [CPJ]
- DynamicJava Interpeter [DJava]
- <FreeMarker> [FM]
- HSQL Database Engine [HSQL]
- Jasper Reports Package [Jasper]
- Java™ 2 Platform, Standard Edition [Java]
- Java™ Servlet Technology [Servlet]
- JavaBeans™ Activation Framework [JAF]
- JavaMail™ API [JMail]
- JFree Charts Package [JFree]
- JFree Reports Package [JFree]

5 Implementation Guidelines

5.1 Coding Standard

Tammi's source code conforms to the Java Coding Standard for Tammi [Coding]. The recommendation for plug-in applications is to apply the same standard.

5.2 Implementing MBeans

MBeans are base objects of Tammi application components. An object is a good candidate to be implemented as an MBean when:

- An object exposes common functionality or data to be used by other objects not in the same layer, package, or component as the object itself
- An object needs to expose its state to be monitored or administrated
- An object can be configured independently at startup or during run-time

5.2.1 Standard MBeans

In order to be manageable in an MBean server, a standard MBean explicitly defines its management interface. The interface of an MBean defines methods it makes available both for reading and writing its attributes and for other objects to invoke.

Standard MBeans rely on a set of naming rules that should be observed when defining the interface of their implementation. The management interface of a standard MBean is composed of:

- The public constructors of the MBean
- The attributes of the MBean exposed through getter and setter methods
- Other methods of the MBean exposed for public use in the MBean interface
- The notification objects and types that the MBean may broadcast

The class of a standard MBean must implement an interface that is named after the class followed by a suffix `MBean`. This interface defines the methods that are exposed for public use. Methods of the MBean’s class, which are not listed in this interface, are not accessible through the MBean server.

Attributes are always accessed via accessor methods. For readable attributes, there is a getter method to read the attribute value. For writable attributes, there is a setter method to allow the attribute value to be updated:

```
public AttributeType getAttributeName();

public void setAttributeName(AttributeType value);
```

For Boolean type attributes, it is recommended to define the following getter method:

```
public boolean isAttributeName();
```

Standard MBeans implementing the javax.management. MBeanRegistration interface will be informed about registrations to the MBean server through callback methods.

Example

```
package org.norther.tammi.manual.sample;

/**
 * FibonacciCounter is an implementation of a simple Fibonacci counter.
 * Note that this code is only instructional.
 */
public class FibonacciCounter
{
    /**
     * The counter.
     */
    private int fibonacciCounter;

    /**
     * Default constructor.
     */
    public FibonacciCounter()
    {
    }

    /**
     * Gets the value of the counter.
     *
     * @return the counter.
     */
    public int getCounter()
    {
        return fibonacciCounter;
    }

    /**
     * Sets the value of the counter.
     *
     * @param counter the counter.
     */
    public void setCounter(int counter)
    {
        fibonacciCounter = counter;
    }
    /**
     * Counts the Fibonacci number specified by the preset counter.
     *
     * @return a Fibonacci number.
     */
    public int count()
    {
        int fibonacci;
        int counter = getCounter();
        if (counter > 0)
        {
            fibonacci = 1;
            if (counter > 2)
            {
                int temporary;
                int previous = fibonacci;
                for (int i = 3; i <= getCounter(); i++)
                {
                    temporary = previous;
                    previous = fibonacci;
                    fibonacci += temporary;
                }
            }
        }
        else
        {
            fibonacci = 0;
        }
        return fibonacci;
    }
}


package org.norther.tammi.manual.sample;

/**
 * CounterMBean is an MBean interface to a simple Fibonacci counter.
 * Note that this code is only instructional.
 */
public interface CounterMBean
{
    /**
     * Gets the value of the counter.
     *
     * @return the counter.
     */
    public int getCounter();


    /**
     * Sets the value of the counter.
     *
     * @param counter the counter.
     */
    public void setCounter(int counter);


    /**
     * Counts the Fibonacci number specified by the counter.
     *
     * @return a Fibonacci number.
     */
    public int count();
}
```

```
package org.norther.tammi.manual.sample;

/**
 * Counter is an MBean implementation of the simple Fibonacci counter.
 * Note that this code is only instructional.
 */
public class Counter extends FibonacciCounter implements CounterMBean
{
    /**
     * Default constructor.
     */
    public Counter()
    {
        super();
    }
}
```

5.2.2 Referable MBeans

Referable MBeans implement ReferableMBean in the core.base package, or one of its extensions. The interface defines a getMBean method returning a direct reference to the corresponding MBean. The reference can be obtained from the MBean server and is applied to achieve better performance when there is a need to call the methods of the referenced MBean frequently. The reference must not be kept for a long period of time, as it becomes invalid when the MBean is unregistered from the MBean server. Instances of the MBeanHandle class can be applied to keep direct references to MBeans up-to-date automatically, as they listen to registration events of their referents and update the references correspondingly. Therefore, referable MBeans must also implement the javax.management.NotificationBroadcaster interface to enable MBeanHandle support.

ReferableMBean can be implemented by extending the Referable class or by applying an implementation of AdaptableMBean. The latter approach allows any Java class to be ReferableMBean with minimum effort.

Configurable and its implementation in the core.config package provide support for property files. Registry, Container and their implementations in the core.base package act as containers of other MBeans. Startable in the core.base package defines start and stop methods for MBeans maintaining an internal state.

New MBeans meant to be referable must extend ReferableMBean, or one of its extensions. It is not enough to implement the ReferableMBean interface together with another MBean interface, as methods of only one MBean interface can be exposed for public use. Thus all public methods of an MBean implementation must be collected to one MBean interface.

The above notice doesn't apply to AdaptableMBean implementations as they expose both ReferableMBean methods and adapted methods dynamically.

Example

```
package org.norther.tammi.manual.sample;

import org.norther.tammi.core.base.ReferableMBean;

/**
 * ReferableCounterMBean extends CounterMBean to be referable.
 * Note that this code is only instructional.
 */
public interface ReferableCounterMBean extends ReferableMBean, CounterMBean
{
}


package org.norther.tammi.manual.sample;

import javax.management.*;

import org.norther.tammi.core.base.Referable;
import org.norther.tammi.core.base.ReferableMBean;

/**
 * ReferableCounter extends Counter to be referable.
 * Note that this code is only instructional.
 */
public class ReferableCounter extends Counter
    implements ReferableCounterMBean, MBeanRegistration,
        NotificationBroadcaster
{
    /**
     * The referable adapter.
     */
    private Referable referable = new Referable(this);

    /**
     * Default constructor.
     */
    public ReferableCounter()
    {
        super();
    }

    public ObjectName preRegister(MBeanServer server, ObjectName name)
        throws Exception
    {
        return  referable.preRegister(server, name);
    }

    public void postRegister(Boolean done)
    {
        referable.postRegister(done);
    }

    public void preDeregister() throws Exception
    {
        referable.preDeregister();
    }
    public void postDeregister()
    {
        referable.postDeregister();
    }

    public ReferableMBean getMBean()
    {
        return referable.getMBean();
    }

    public ObjectName getObjectName()
    {
        return referable.getObjectName();
    }

    public boolean isRegistered()
    {
        return referable.isRegistered();
    }

    public MBeanServer getMBeanServer()
    {
        return referable.getMBeanServer();
    }

    public void addNotificationListener(NotificationListener listener,
        NotificationFilter filter, Object handback)
    {
        referable.addNotificationListener(listener, filter, handback);
    }

    public void removeNotificationListener(NotificationListener listener)
        throws ListenerNotFoundException
    {
        referable.removeNotificationListener(listener);
    }

    public MBeanNotificationInfo[] getNotificationInfo()
    {
        return referable.getNotificationInfo();
    }
}
```

5.2.3 Adaptable MBeans

The model MBean specification of JXM defines the javax.management.modelmbean.ModelMBean interface providing a management template for managed resources not implemented as standard MBeans. The JMX agent must provide the corresponding implementation class named javax.management.modelmbean.RequiredModelMBean. This model MBean implementation is intended to provide ease of use and extensive default management behavior for the instrumentation.

The AdaptableMBean interface and its default implementation in the core.model package extend model MBeans by implementing the ReferableMBean interface and providing automatic introspection of managed resources based on the JavaBeans specification.

Example

```
MBeanServer server = getMBeanServer();

ObjectInstance instance = server.createMBean(
    "org.norther.tammi.core.model.AdapterX",
    (ObjectName) null,
    new Object[]{ new FibonacciCounter() },
    new String[]{ "java.lang.Object" });

ObjectName name = instance.getObjectName();
server.set(name, "Counter", new Integer(5));
Integer fibonacci = (Integer) server.invoke(name, "count");
```

5.2.4 Executable MBeans

A set of Executors, a thread factory MBean, Executable and its implementation in the core.thread package provide a model for implementing multi-threaded MBeans. MBeans implementing Executable and the run method defined by it can be executed under a specific Executor.

- A direct executor is provided for consistency and it executes Executables in the caller's thread
- A threaded executor executes Executables in new threads created for each executable
- A pooled executor executes Executables in threads obtained from a thread pool and is capable of queuing executables when no threads are available

5.2.5 Dynamic MBeans

Standard MBeans are suitable for straightforward structures, where the exposed functionality and data is well defined in advance and unlikely to change often. When the data structures are likely to evolve often over time, the implementation must provide more flexibility, such as being determined dynamically during run-time. Dynamic MBeans bring this adaptability and provide an alternative MBean implementation with more elaborate capabilities.

Dynamic MBeans implement the predefined javax.management.DynamicMBean interface, which exposes the attributes and operations at run-time. Instead of exposing them directly through method names, dynamic MBeans implement a method, which returns all attributes and method signatures.

Dynamic MBeans offer the same capabilities that are available through standard MBeans when accessed through the MBean server. However, their attributes and methods cannot be accessed directly like methods of ReferableMBeans.

Note that the Variable interface, its default implementation and the VariableMBeanAttributeInfo class in the core.rt package provide a prepared model to apply dynamic MBeans. The dynamic information of them can be defined neatly in the scripts.

Example

```
package org.norther.tammi.manual.sample;

import javax.management.*;

/**
 * DynamicCounter implements Counter dynamically.
 * Note that this code is only instructional.
 *
 * @author Ilkka Priha
 */
public class DynamicCounter
    extends Counter
    implements DynamicMBean
{
    /**
     * The MBeanInfo.
     */
    private MBeanInfo mBeanInfo;
```

```
   /**
    * Default constructor.
    */
   public DynamicCounter()
   {
       super();

       // Create the bean info (this should be generated
       // based on some dynamic information instead of
       // hardcoded one to exploit its potential).
       try
       {
           mBeanInfo = new MBeanInfo(
               DynamicCounter.class.getName(),
               "A dynamic Fibonacci counter.",
               new MBeanAttributeInfo[]
               {
                   new MBeanAttributeInfo(
                       "Counter","int",
                       "The counter for a Fibonacci number.",
                       true, true, false)

               },
               new MBeanConstructorInfo[]
               {
                   new MBeanConstructorInfo(
                       "Default constructor.",
                       DynamicCounter.class.
                           GetConstructors()[0])
               },
               new MBeanOperationInfo[]
               {
                   new MBeanOperationInfo(
                       "Counts a Fibonacci number.",
                       DynamicCounter.class.
                           getMethod("count", new Class[0]))
               },
               new MBeanNotificationInfo[0]);
       }
       catch (Exception x)
       {
       }
   }


   public MBeanInfo getMBeanInfo()
   {
       return mBeanInfo;
   }


   public Object getAttribute(String name)
       throws AttributeNotFoundException,
              ReflectionException,
              MBeanException
   {
       if (name.equals("Counter"))
       {
           return new Integer(getCounter());
       }

       throw new AttributeNotFoundException(name);
   }
```

```
public void setAttribute(Attribute attr)
    throws AttributeNotFoundException,
           InvalidAttributeValueException,
           ReflectionException,
           MBeanException
{
    String name = attr.getName();
    Object value = attr.getValue();
    if (name.equals("Counter"))
    {
        try
        {
            setCounter(((Integer) value).intValue());
        }
        catch (Exception x)
        {
            throw new ReflectionException(x);
        }
        return;
    }

    throw new AttributeNotFoundException(name);
}


public AttributeList getAttributes(String[] names)
{
    AttributeList list = new AttributeList(names.length);
    for (int i = 0; i < names.length; i++)
    {
        try
        {
            list.add(
                new Attribute(names[i],
                    getAttribute(names[i])));
        }
        catch (Exception x)
        {
        }
    }
    return list;
}


public AttributeList setAttributes(AttributeList attrs)
{
    Attribute attr;
    AttributeList list = new AttributeList(attrs.size());
    for (int i = 0; i < attrs.size(); i++)
    {
        try
        {
            attr = (Attribute) attrs.get(i);
            setAttribute(attr);
            list.add(
                new Attribute(attr.getName(),
                    getAttribute(attr.getName())));
        }
        catch (Exception x)
        {
        }
    }
    return list;
}
```

```
    public Object invoke(String method,
                         Object[] params,
                         String[] signature)
        throws ReflectionException
               MBeanException
    {
        if (method.equals("count"))
        {
            return new Integer(count());
        }

        throw new ReflectionException(
            new NoSuchMethodException(method));
    }
}
```

5.3 Identifying MBeans

An object name uniquely identifies an MBean within an MBean server. Clients use this object name to identify the MBean on which to perform operations. The class javax.management.ObjectName represents an object name, which consists of two parts:

- A domain name
- An unordered set of one or more key properties

5.3.1 Domain Name

The domain name is a case-sensitive string. It provides a structure for the naming space within a JMX agent or within a global management solution. The domain name part may be omitted in an object name, as the MBean server is able to provide a default domain. When an exact match is required, omitting the domain name will have the same result as using the default domain defined by the MBean server.

How the domain name is structured is application-dependent. The domain name string may contain any characters except those which are object name separators or wild-cards, namely the colon, comma, equals sign, asterisk or question mark (:,=*?). JMX always handles the domain name as a whole; therefore any semantic sub-definitions within the string are opaque to a JMX implementation.

Domain names used by Tammi are prefixed with “tammi” by default. The default domain is used for shared services. A few other domain names reserved by Tammi have been selected by functional criteria. Plug-in applications should use at least one application specific domain for their internal MBeans.

Domain names shouldn't be referenced directly from source code as both the applied naming scheme and individual domain names can vary depending on the configuration. Configuration scripts should create an applicable naming scheme and pass domain names to MBeans as attributes or in parameters of methods.

5.3.2 Key Property List

The key property list enables you to assign unique names to the MBeans of a given domain. A key property is a property-value pair, where the property does not need to correspond to an actual attribute of an MBean.

The key property list must contain at least one key property. It may contain any number of key properties, whose order is not significant.

Tammi uses a naming scheme where a key property named “type” is assigned to all MBeans. Its value is the name of the MBean's interface without the package name. This key helps to identify MBeans.

By default, Tammi registers MBeans to the default domain. The Referable class creates a unique object name for any MBean extending it that is registered without a predefined object name.

Direct references to key names from source code should be avoided. Keys can be used for classifying MBeans by different criteria, but the names of keys to use for classification should always be passed to MBeans as attributes or in parameters of methods.

If the key value presents an attribute of the MBean, the attribute should be read-only during the registration time of the MBean.

5.3.3 Pattern Matching

The object name can be either a qualified name or a query. The latter one contains wild-cards applied in pattern matching. The result of pattern matching can be a set of matching MBeans or the first one found.

The matching syntax for domain names is:

- * matches any character sequence, including an empty one
- ? matches any one single character

There is no wild-card matching performed neither on key property names nor on key property values. Only complete property-value pairs are used in pattern matching. While key properties are atomic, the list of key properties may be incomplete and used as a pattern:

- * matches any property-value pair, including an empty one

5.4 Accessing MBeans

MBeans can be accessed through the MBean server by identifying them by their object name. Within Tammi, the Broker class implements a large set of both static and non-static methods for resolving MBean references. The MBeanBroker class provides access to these methods from scripts and templates.

5.4.1 From Objects

Broker defines methods with which to resolve references to MBeans, and the Broker class provides static methods to get the default instance implementing the interface. The methods take a key as a parameter and try to resolve the key to a direct reference to the corresponding ReferableMBean, to a qualified object name of the MBean, or to a set of qualified object names, depending on the method. The key can be an MBean interface, an object name query with wild-cards, or an application specific alias. Object names should not be referenced directly from source code, but they should be hidden behind an alias or passed to MBeans as attributes or in parameters of methods.

For MBean interfaces, Broker tries to find the first MBean in the default domain implementing the specified interface. This method should be used when the client needs to access the default implementation of a service MBean. The configuration should maintain only one instance of each service MBean in the default domain.

For object name queries, Broker tries to find the first MBean matching the specified query, or alternatively all matching MBeans.

For aliases, Broker tries to resolve the MBean behind the alias. Application specific aliases can be configured in Broker at startup. Aliases are arbitrary strings mapped to qualified object names or queries.

5.4.2 From Scripts and Templates

The MBeanBroker class is a helper class providing access to Broker from scripts and templates. MBeanBroker supports basic data types and performs type castings automatically making it easier to use it from scripting environments. An instance of MBeanBroker is automatically added to all scripting and template contexts within Tammi. The reference to it is mapped to a `broker` key by default.

Scripts and templates have full access to all registered MBeans through the broker.

5.5 Handling URLs

A URL represents a Uniform Resource Locator, a pointer to a "resource" on the HTTP and other Internet protocols. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object. Each request has a URL specifying its target resource.

A URI represents a Uniform Resource Identifier, which is a superset of a URL not limited to locations but rather identifying resources with a name or a set of attributes. The difference between the two is not remarkable.

Example

```
http://server:8080/app/tammi/template/Index;jsessionid=A1SDZ2?id=foo
```

In general, a URL can be broken into several parts. The "http" scheme above specifies the protocol for locating the resource, the "server:8080" part specifies the host name and port of the resource, and the rest of the components specify the URL path. With the HTTP protocol, the URL path can contain specific components that are not actually part of the path locating the resource, but additional instructions related to request processing.

A semicolon (";") separates URL object parameters from the actual URL path and the components after a question mark ("?") specify a query string.

Also the components of the actual URL path have different meanings. The division is based on the servlet specification [Servlet]. The URL path can be divided into two parts:

- Context path specifies the external context of servlet applications
- Path info contains application specific information for pipes and filters

5.5.1 Relative vs. Absolute URLs

A relative URL needs not to specify all parts of a URL. If the protocol, host name, or port number is missing, the value is inherited from the fully specified URL of some previous request. The path part must always be specified.

Relative URLs are preferred to absolute ones as their protocol and host parts are resolved automatically. However, redirections and specific configurations require absolute URLs specifying all parts of the URL. The getLinkedURL and getRedirectedURL methods of the ProtocolExtension interface can be used for building URLs. They return absolute or relative URLs depending on the situation and configuration. The behavior of these methods can be configured in HttpFilter.

5.5.2 Context Path

If some external server, like a servlet container, relay or cluster, controls one or more Tammi instances, the context path contains context information for that external server.

The context path is returned by the getContextPath method of the javax.servlet.http.HttpServletRequest interface. It must be included in redirections and links.

5.5.3 Path Info

The path info contains any application specific information associated with the URL the client sent when it made a request. The path info follows the context path but precedes the object parameters and query string. HttpFilter can be configured to parse the path info as a set of key-value pairs to be added to request parameters.

The getPathInfo method of the javax.servlet.http.HttpServletRequest interface returns an unparsed path info. A set of parameter methods can be applied to access the parsed parameters.

5.5.4 Object Parameters

Object parameters are used internally by the HTTP services of Tammi. E.g. the session id for clients not supporting cookies is encoded in object parameters.

Object parameters are not visible to applications. However, they must be included in links depending on sessions by calling the encodeURL or encodeRedirectURL methods of the javax.servlet.http.HttpServletResponse interface before adding the links to the response.

5.5.5 Query String

A query string is optionally contained in the request after the path info and object parameters. It contains a set of key-value pairs that are parsed and added to request parameters.

The getQueryString method of the javax.servlet.http.HttpServletRequest interface returns an unparsed query string. A set of parameter methods can be applied to access the parsed parameters.

The addQueryString method of the ProtocolExtension interface adds parameters to links to be included in the response. The parameters are automatically added to locations returned by the encodeURL and encodeRedirectedURL methods of the javax.servlet.http.HttpServletResponse interface.

5.5.6 Redirection

Redirection of a request must include an absolute URL. The sendRedirect method of the javax.servlet.http.HttpServletResponse interface converts a location into an absolute URL and makes a redirection response. Note that if the redirected request depends on sessions, object parameters must be included in the location beforehand by calling the encodeRedirectURL method of the interface. RedirectionException in the content package can be thrown from a filter to perform redirection automatically.

The getRequestURL method of the javax.servlet.http.HttpServletRequest interface returns the current request URL, which can be given as a parameter to the encodeRedirectURL and sendRedirect methods.

5.5.7 Links

In some configurations controlled by an external server, links must be absolute. The context path must always be present in links. Links depending on sessions must also be checked to include the session id in their object parameters, if the client does not support cookies. The getLinkedURL and getRedirectedURL methods of the ProtocolExtension interface and the encodeURL and encodeRedirectURL methods of the javax.servlet.http.HttpServlet.Response interface can be used for building links. In addition, several aspects related to character encoding of URL paths and query strings must be taken into account. These are discussed in the following section.

LinkTool available in the template context contains several methods helping applications in building links within templates.

5.6 Character Encoding

Character encoding is a method (algorithm) for presenting characters in digital form by mapping sequences of code numbers of characters into sequences of octets. In the simplest case, each character is mapped to an integer in the range 0 - 255 according to a character code and these are used as such as octets. Naturally, this only works for character repertoires with at most 256 characters. For larger sets, more complicated encodings are needed. Encodings, which are also called charsets, have registered names, for example ISO -8859-! formerly known as Latin 1.

Processing requests based on various Internet protocols require mappings from computer specific bytes to characters of human languages in several places. In some cases, these mappings are well specified by the protocol while in others applying the correct one requires a predefined agreement between the client and the server. Such an agreement must be configured to the server.

5.6.1 URL Encoding

The RFC2396 defines a general syntax for URIs. It limits the allowed characters to “a”-“z”, “A”-“Z” and “0”-“9”. In addition, the following punctuation marks and symbols are not reserved for any specific purpose “-”, “_”, “.”, “!”, “~”, “*”, “'”, “(” and “)”. All other characters must be encoded as escaped octets consisting of the percent character “%” followed by the two hexadecimal digits representing the octet code. For example, “%20” is the escaped encoding for the US-ASCII space character. Encoding of parameters of a query string has an exception, which encodes spaces as plus signs “+”.

Incoming URLs are by default decoded based on this specification and outgoing links should follow the same specification. The static URLDecoder and URLEncoder classes contain a set of methods for decoding and encoding URL components.

If the decoded URL represents a string in some language not using the US-ASCII character set, the URL must be decoded once again to map the 8 bit octets of the above specification to language specific Unicode characters. The default encoding is the same as the request encoding. If the request doesn't specify an encoding, Tammi applies the encoding configured in HttpFilter. If none is specified, Tammi applies either UTF-8 or ISO-8859-1 depending on which one is accepted by the client.

The newer RFC2718 defines guidelines for URL schemes. It specifies that the default encoding from 8 bit octets to language specific characters should be UTF-8, which is reasonable as this encoding is capable of performing mappings to any language.

The implementations of the javax.servlet.http.HttpServletRequest interface perform URL decoding automatically based on the current configuration. Correspondingly, LinkTool performs URL encoding of links automatically.

5.6.2 Request Encoding

The RFC2068 defines the HTTP protocol version 1.1. A start-line, headers and body form a HTTP request. The start-line and headers contain typically characters only from the US-ASCII character set. The Content-Type header specifies the encoding of the body of the request. If the encoding is not specified, Tammi applies the encoding configured in HttpFilter. If none is specified, Tammi uses ISO-8859-1 by default.

The getCharacterEncoding method of the javax.servlet.ServletRequest interface returns the current request encoding.

Tammi provides support for parsing HTML Form based request bodies only; more complicated request bodies must be parsed by applications.

5.6.3 Response Encoding

The encoding of the response is decided by applications. Tammi provides ContentTypeMap for mapping language and country specific locales to character encodings. The locale for a specific pipe can be configured in LocaleFilter.

An explicit response encoding can be set with the setContentType method of the javax.servlet.ServletResponse interface.

The getCharacterEncoding method of the javax.servlet.ServletResponse interface returns the current response encoding.

5.6.4 Template Encoding

Templates are usually written using an encoding corresponding to the language of the contents of the template. Sometimes the editors used for writing templates do not support language specific encodings but some generic one, like UTF-8, must be applied. The default template encoding can be configured in TemplateEngine. Pipe specific encodings can be configured in PageFilter and LayoutFilter.

5.6.5 Resource Encoding

Property files can also contain localized resources. The same encoding rules apply to resources as to templates. The default resource encoding can be configured in ResourceFinder.

6 Configuration Notes

Configuration files of Tammi include property files, BSF scripts [BSF] and markup templates. They can be located in jar archives, local directories, or remote URLs. Configuration directories can be configured in PathFinder.

6.1 Property Files

The Configurable class implements methods to manage internal properties of an MBean. If the MBean wants to expose these methods to public use, its MBean interface can extend Configurable.

The properties of an MBean are maintained in a Configuration object. Configuration is an extended version of the java.util.Properties class supporting any character encoding in the property file, include statements to other property files within the property file, multi-value properties as java.util.Vectors, and several type conversion methods for accessing property values.

Configurable defines methods to specify a property file, property URL or individual properties. It resolves relative pathnames of property files by calling PathFinder. PathFinder maintains a list of directories containing property files mapped to its `config` key.

Configurator is capable of configuring other MBeans through their public attributes exposed for management.

6.2 BSF Scripts

BSF scripts can be executed through Scripter. During startup, scripts located in application jars and startup directories are executed automatically.

Startup looks for application jars from directories configured in PathFinder. PathFinder maintains a list of directories containing application jars mapped to its `lib` key.

In addition, Startup looks for scripts from startup directories specified in command line options and configured in PathFinder. PathFinder maintains a list of directories containing startup scripts mapped to its `start` key.

It is recommend that applications package their base scripts into jar archives and leave frequently changing scripts into startup directories. Script names should be prefixed with a three-digit number explicitly specifying the execution order.

6.3 Templates

The mechanism for finding templates depends on the specific template engine implementation. VelocityEngine and FreemarkerEngine use resource loaders for loading templates. The current implementation supports class path resource loaders, file resource loaders, jar resource loaders and property resource loaders. The resource loaders to apply in a specific configuration can be configured separately to each template engine instance.

Resource loaders resolve relative pathnames to template directories by calling PathFinder. PathFinder maintains a list of directories containing templates mapped to its `template` and `protected` keys. The latter one contains templates with restricted access. The list of directories containing jar archives is mapped to its `lib` key.

6.4 Startup Sample

Startup is responsible for starting up Tammi, initializing MBeans providing base services required by the framework, configuring the run-time environment and loading plug-in applications. The Startup class implements Startup and serves as the default startup object. It can be activated either by invoking its static main method when starting the JVM, or by simply instantiating it e.g. from a servlet, like StartupServlet does.

Startup creates an MBean server, creates and registers base MBeans and executes startup scripts located in application jars and startup directories. It supports the following command line options controlling the configuration:

- `-r(oot) <path>` specifies the root directory of relative file paths
- `-b(in) <path>` specifies the binary directory containing native libraries; more than one option or a comma separated list can be given to specify multiple directories
- `-d(ata) <path>` specifies the directory containing persistent data
- `-t(emp) <path>` specifies the directory containing temporary data
- `-c(config) <path>` specifies the configuration directory containing property files; more than one option or a comma separated list can be given to specify multiple directories
- `-s(tart) <path>` specifies the startup directory containing startup scripts; more than one option or a comma separated list can be given to specify multiple directories
- `-domain <domain>` specifies the default domain for MBeans
- `-wait` forces the main thread to wait until a shutdown method is called

During the first round, Startup collects all startup scripts from application jars into a sorted set. Then it collects scripts from startup directories and adds them to the same set. Scripts from the most recent sources override duplicates. After collection, the scripts in the set are executed in ascending order.

During the following rounds, Startup checks if any new startup directories have been added to PathFinder. Any new scripts found from the new directories will be executed and the check repeated until no new directories are found.

A typical startup creates, registers and starts an applicable set of services, creates factories for base objects and loads plug-in applications by executing application specific startup scripts.

Depending on the configuration, startup can activate a set of connectors listening to client requests from different input sources. An active connector passes received requests to a pipe attached to it. The connector pipe can contain filters directing the request flow to application specific pipes based on mappings between request parameters and pipe names. Depending on the application, the pipes can contain filters executing CGI scripts, returning static files, or rendering dynamic pages.

In the presented sample configuration, filters call recursively the doFilter method of the filter chain until one of them returns a response, or an exception is thrown. The doFilter method is represented as an unnamed call in the diagram. Branch filters direct requests to a set of branching pipes and filters. Branches are alternate routes; only one is selected for any single request. Filters can interact freely with other services.

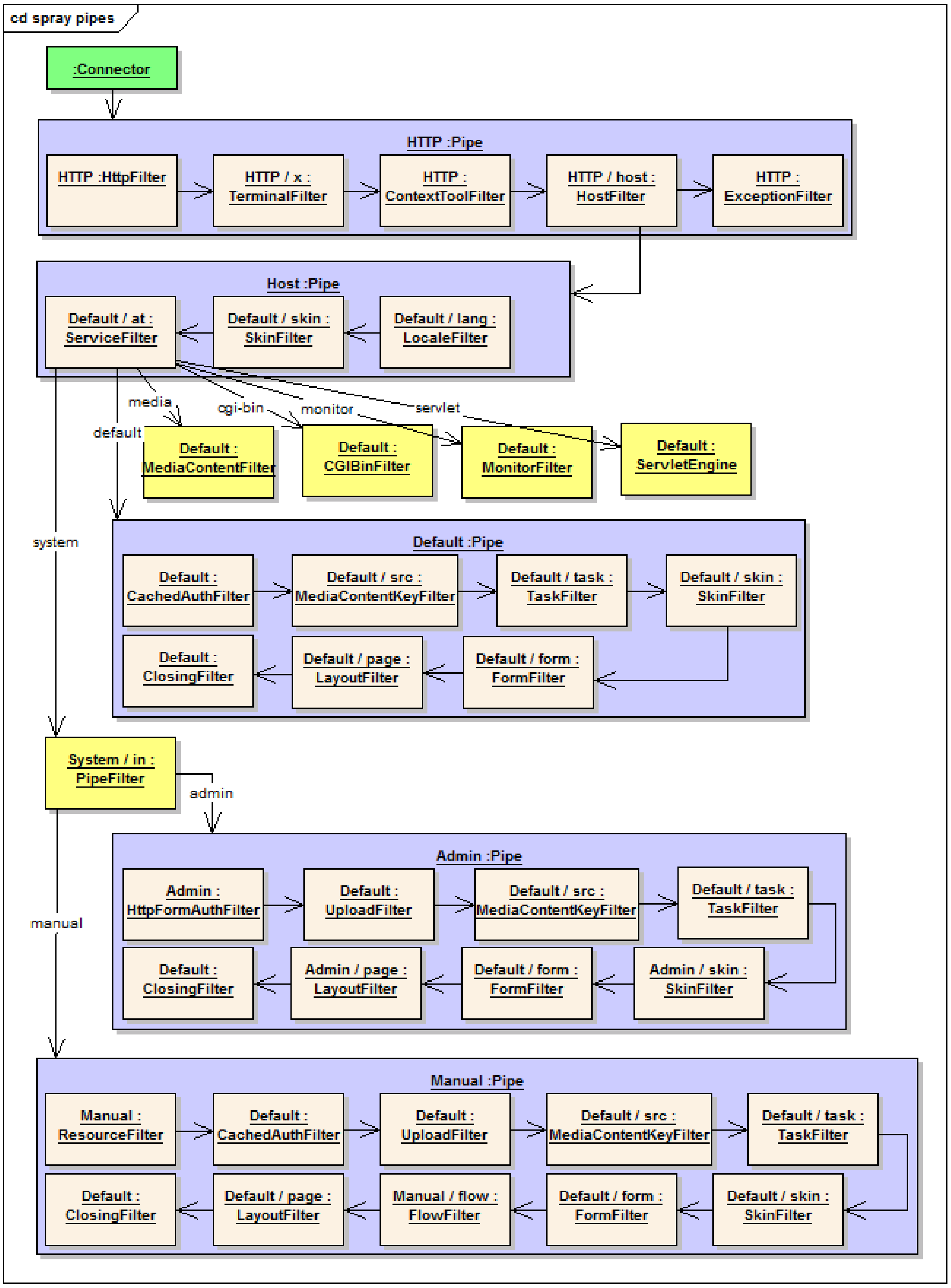
cd spray pipes
:Connector
HTTP :Pipe
HTTP :HttpFilter
HTTP / x : TerminalFilter
HTTP : ContextToolFilter
HTTP / host : HostFilter
HTTP : ExceptionFilter
Host :Pipe
Default / at : ServiceFilter
Default / skin : SkinFilter
Default / lang : LocaleFilter
media
cgi-bin
monitor
servlet
default
Default : MediaContentFilter
Default : CGIBinFilter
Default : MonitorFilter
Default : ServletEngine
system
Default :Pipe
Default : CachedAuthFilter
Default / src : MediaContentKeyFilter
Default / task : TaskFilter
Default / skin : SkinFilter
Default : ClosingFilter
Default / page : LayoutFilter
Default / form : FormFilter
System / in : PipeFilter
admin
Admin :Pipe
Admin : HttpFormAuthFilter
Default : UploadFilter
Default / src : MediaContentKeyFilter
Default / task : TaskFilter
Default : ClosingFilter
Admin / page : LayoutFilter
Default / form : FormFilter
Admin / skin : SkinFilter
manual
Manual :Pipe
Manual : ResourceFilter
Default : CachedAuthFilter
Default : UploadFilter
Default / src : MediaContentKeyFilter
Default / task : TaskFilter
Default : ClosingFilter
Default / page : LayoutFilter
Manual / flow : FlowFilter
Default / form : FormFilter
Default / skin : SkinFilter

Figure 6.1 A sample configuration

6.4.1 HttpFilter

DefaultHttpFilter is the default implementation of HttpFilter in the spray.protocol package. It parses requests based on the HTTP protocol. It can be configured to provide additional services, like parsing of session cookies.

6.4.2 TerminalFilter

DefaultTerminalFilter is the default implementation of TerminalFilter in the spray.terminal package. It defines the user terminal type based on user agent information of requests and selects an appropriate MIME type for the corresponding responses. Its default key for explicit terminal selection is 'x'.

6.4.3 ContextToolFilter

DefaultContextToolFilter is the default implementation of ContextToolFilter in the spray.engine package maintaining a hierarchical context of context tools. Context tools to be loaded automatically into the context by the filter are specified in its configuration. The standard context hierarchy consists of a request context, session context and global context. Additional contexts can be inserted into the hierarchy by other filters.

6.4.4 HostFilter

DefaultHostFilter is the default implementation of the HostFilter in the spray.protocol package. It selects the host pipe to apply based on the HTTP host request header. Its default key for explicit host selection is 'host'.

6.4.5 LocaleFilter

DefaultLocaleFilter is the default implementation of LocaleFilter in the spray.terminal package providing locale support for responses. Its default key for explicit locale selection is 'lang'.

6.4.6 ExceptionFilter

DefaultExceptionFilter is the default implementation of ExceptionFilter in the spray.filter package and it returns an evaluated error message as a response if the corresponding request was not accepted by any of the preceding filters.

6.4.7 ServiceFilter

DefaultServiceFilter is the default implementation of ServiceFilter in the spray.media package and it acts as the service-level branch filter. The service filter has a specific role to select the service specific pipe to which to direct requests. Its default key for explicit service selection is 'at'.

As plug-in applications may generate dynamically links to each other, they must have access to the mappings maintained by the service filter. The mappings are exposed to clients through methods defined by ServiceFilter.

6.4.8 MediaContentFilter

DefaultMediaContentFilter is the default implementation of MediaContentFilter in the spray.media package providing cached support to access both binary and markup files requested by users.

6.4.9 CGIBinFilter

DefaultCGIBinFilter is the default implementation of CGIBinFilter in the spray.media package executing CGI scripts.

6.4.10 HttpFormAuthFilter

DefaultHttpFormAuthFilter implements AuthenticatorFilter in the spray.authenticator package and it authenticates user principals by applying an HTTP login form.

6.4.11 MediaContentKeyFilter

DefaultMediaContentKeyFilter is an extension of DefaultContentFilter. It can be configured to search for content files from application specific directories. Its default key for explicit media file selection is 'src'.

6.4.12 TaskFilter

DefaultTaskFilter is the default implementation of TaskFilter in the spray.template package, and it invokes actions specified in request data. Actions are implementations of the Task interface and must implement an execute method. Actions can also be BSF scripts. TaskLoader loads specified actions and caches them for forthcoming invokes. Packages to search for actions can be configured in TaskLoader. Its default key for explicit task selection is 'task'.

6.4.13 SkinFilter

DefaultSkinFilter is the default implementation of SkinFilter in the spray.template package. It selects the skin to apply when rendering web content. Its default key for explicit skin selection is 'skin'.

6.4.14 FormFilter

DefaultFormFilter is the default implementation of FormFilter in the spray.template package and it provides support to render and validate HTML forms. It maintains mappings between class types and control templates to enable composition of complicated form templates from type specific input field templates. Its default key for explicit form theme selection is 'form'.

6.4.15 FlowFilter

DefaultFlowFilter is the default implementation of FlowFilter in the spray.template package. It executes flow specific step templates, variables and actions. Its default key for explicit flow selection is 'flow'.

6.4.16 LayoutFilter

LayoutFilter is the more complicated one of the two template rendering filters while PageFilter is the simpler one. DefaultPageFilter implements PageFilter in the spray.template package and evaluates a template specified in request data by passing its name to TemplateEngine. Their default key for explicit page selection is 'page'.

Templates can render sub-templates by calling a page tool if the tool is loaded into the context. Sub-templates may define common parts of user interface pages while the actual target template produces the request specific content.

DefaultLayoutFilter implements LayoutFilter in the template package and supports the page layout model. It does not return the target template directly, but evaluates a layout template enclosing the target template in the desired location through the page tool. LayoutFilter complies with the page-building model of Turbine [Turbine].

6.4.17 ClosingFilter

DefaultClosingFilter is the default implementation of ClosingFilter in the spray.filter package terminating the filter chain and closing the response if no other filter has done it.