PowerMatcher Core and Extension

Architecture and Design

Authors:

Aldo Eisma (IBM)

Release: **0.9**

Revision: 2

Date: 5-3-2013

TABLE OF CONTENTS

TABLE OF FIGURES

VERSION HISTORY

Revision	Description	Author
0.7-1	Initial version for PowerMatcher release 0.7.	A.H. Eisma (IBM)
0.9-1	Updated for PowerMatcher release 0.9	A.H. Eisma (IBM)
0.9-2	Minor corrections to document scope description.	A.H. Eisma (IBM)

1. Introduction

1.1 The PowerMatcher concept

The PowerMatcher concept is based on the micro-economic principle of demand and supply.
Supply and demand is one of the most fundamental concepts of economics and it is the backbone of a market economy. Demand refers to how much (quantity) of a product or service is desired by buyers. The quantity demanded is the amount of a product people are willing to buy at a certain price; the relationship between price and quantity demanded is known as the demand relationship. Supply represents how much the market can offer. The quantity supplied refers to the amount of a certain good producers are willing to supply when receiving a certain price. The correlation between price and how much of a good or service is supplied to the market is known as the supply relationship. Price expresses the willingness to pay for supply and demand.
The relationship between demand and supply underlie the forces behind the allocation of resources. In market economy theories, demand and supply theory will allocate resources in the most efficient way possible.
When supply and demand are equal (i.e. when the supply function and demand function intersect) the economy is said to be at equilibrium.
The same mechanism is used in PowerMatcher. PowerMatcher core application provides the marketing mechanism for the determination of the equilibrium, while the agents work as actors representing demand and/or supply.

1.2 PowerMatcher Core versus Extension

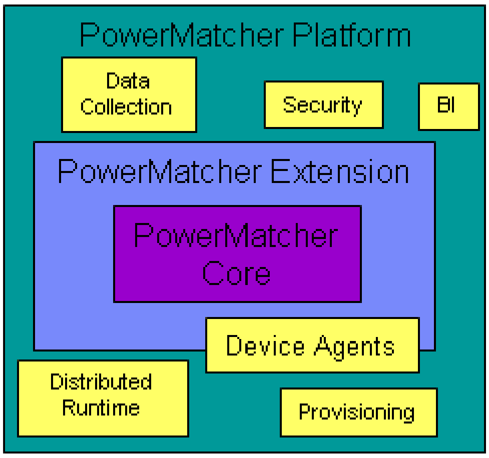


Figure 1 - PowerMatcher Core and Extension in a PowerMatcher Platform

- **PowerMatcher Core (reference implementation)**
  1. A Java component model and framework for “plug & play” PowerMatcher components.
  2. Strict separation between PowerMatcher API and protocol layer.
  3. Common PowerMatcher components like Auctioneer, Concentrator, etc.
  4. Protocol adapter for MQ Telemetry messaging bus.

- **PowerMatcher Extension** adds the following to the core as sample extensions
  1. Centralized platform for managing multiple, independent, PowerMatcher clusters.
     - *Dynamic provisioning*: software & configuration updates for PowerMatcher clusters.
     - *Data collection*: logging PowerMatcher and telemetry data in a database.
  2. Run-time platform for PowerMatcher clusters.
     - *Centralized*: Auctioneer and other agents running in the server environment.
     - *Remote*: Gateway for running concentrators and other agents on distributed sites using IBM’s MQTT MicroBroker for distributed messaging.
  3. Agent library containing sample agent and concentrator implementations.

- The **PowerMatcher Platform**, which is out of scope for the reference implementation, delivering the full-fledged production capability for running PowerMatcher systems.
  1. Data collection for capturing monitoring and measurement data.

2. Business Intelligence.
3. Security and other non-functional requirements.
4. Software component and configuration provisioning.
5. Project and device specific PowerMatcher components.

1.3 Scope of this document

This document covers the architecture and design of PowerMatcher Core and Extension. The design of a full-fledged production grade PowerMatcher Platform is out of scope for the reference implementation.

2. Architecture and Design of PowerMatcher Core and Extension

2.1 PowerMatcher protocol

2.1.1 Protocol stack

A PowerMatcher system consists of event-based software agents that communicate via asynchronous messages. To support representation independence and distribution in heterogeneous environments, the communication between agents is based on a layered model, as shown in Figure 2.

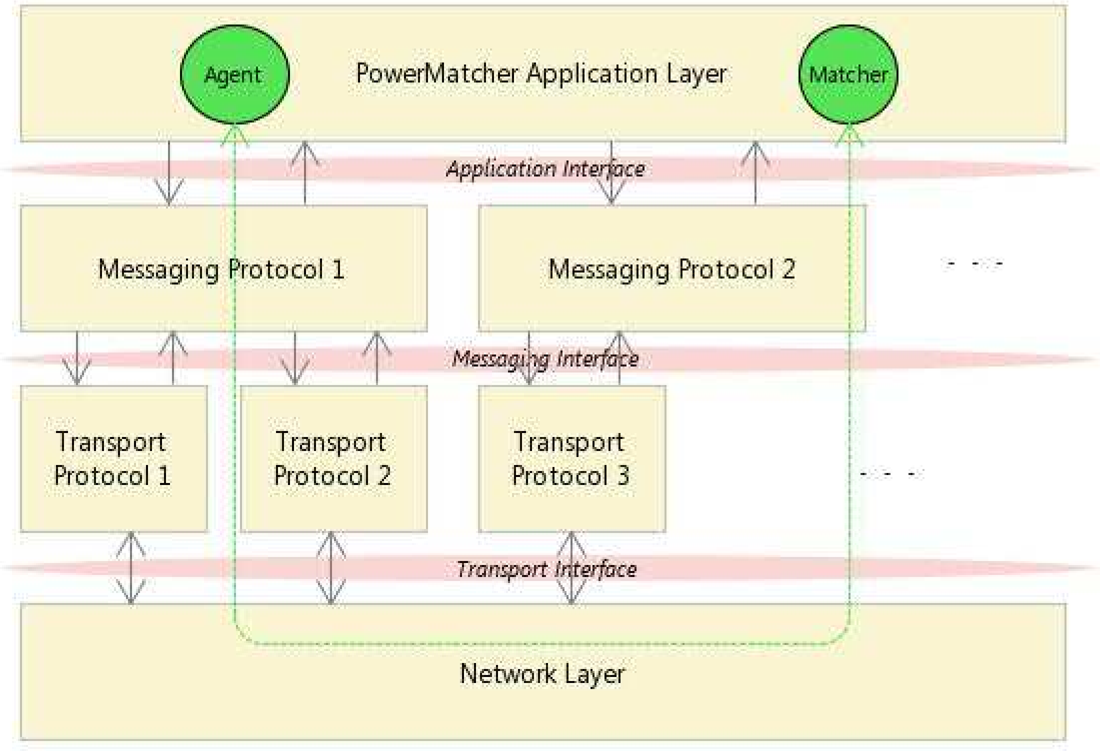


Figure 2 - Protocol stack

This figure shows the interaction between a software agent with the PowerMatcher Agent role and a software agent with the PowerMatcher Matcher role through the protocol stack.

- The ***Application Layer*** implements an ***Application Interface*** for sending and receiving PowerMatcher events, Telemetry events and logging events in a protocol independent manner. The *Application Interface* is implementation dependent.
- The ***Messaging Layer*** defines a protocol and messages in a transport independent manner in the ***Messaging Interface***. PowerMatcher provides a broadband and a narrowband protocol for PowerMatcher that can be used side by side.
- The ***Transport Layer*** defines the connection-oriented or connectionless transport of messages between software agents. PowerMatcher provides two publish-subscribe messaging transports.

When an agent and its parent matcher are deployed in the same runtime environment, it is possible to connect them directly, as shown in the next figure. Direct connections are more efficient and are highly scalable. Messaging connections for multiple protocols as well as direct connections are supported in parallel for the same matcher.

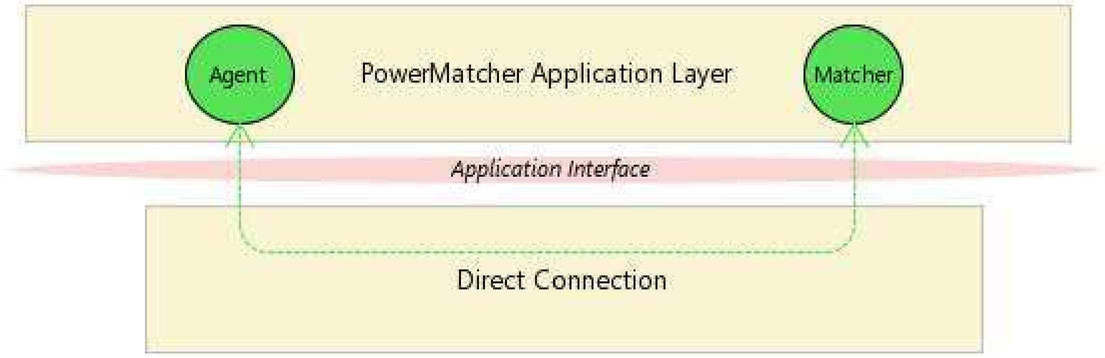


Figure 3 - Protocol stack with direct connection

2.1.2 Application level protocol

The *Application Layer* implements an *Application Interface* for sending and receiving PowerMatcher and other events in a protocol independent manner. The *Application Interface* is implementation dependent. An example is PowerMatcher, where the Application Interface is implemented as a Java API.

The application level protocol for PowerMatcher is specified in **Error! Reference source not found.**, the telemetry and logging protocols are specified here.

2.1.3 PowerMatcher broadband messaging protocol

The broadband messaging protocol in PowerMatcher is a binary internal protocol. It will be replaced with the standard broadband messaging protocol once the standard has been defined. The broadband protocol is labelled "INTERNAL_v1".

The PowerMatcher broadband messaging protocol is specified in [1].

2.1.4 PowerMatcher narrowband messaging protocol

The narrowband messaging protocol in PowerMatcher is a highly optimized binary protocol that is based on the NXP PowerMatcher for the HAN protocol rev. 6. It will be replaced with the standard narrowband messaging protocol once the standard has been defined. The narrowband protocol is labelled "HAN_rev6".

The PowerMatcher narrowband messaging protocol is also specified in [1].

2.1.5 PowerMatcher logging messaging protocol

The protocol adapters for the PowerMatcher broadband and narrowband protocol are also responsible for logging PowerMatcher events. Logging of the events is separated from the events themselves so that events can be logged selectively and with additional information.

Logging is implementation specific and thus not part of the PowerMatcher protocols specified in [1].
The PowerMatcher logging messaging protocol implemented in PowerMatcher is covered in section 2.7.3.

2.1.6 Telemetry and control messaging protocol

The PowerMatcher implementation provides a separate protocol for reporting sensor data and remote control of devices. This feature is implementation specific and not part of the PowerMatcher specification.
The telemetry and control messaging protocol implemented in PowerMatcher is covered in section 2.8.

2.1.7 Transport protocol

2.1.7.1 The messaging bus

In PowerMatcher, PowerMatcher, logging, telemetry and other messages are exchanged either via direct connections within a single PowerMatcher runtime, or via a message bus that can be used by multiple applications. The message bus is there to facilitate n-m publish/subscribe messaging in a structured way. A publish/subscribe message bus supports loosely coupled components in the following way:

- A producer can publish messages on the bus on a named topic.
- When a message is published, it is delivered to all consumers that subscribe to that topic. There can be zero or more subscribers for a topic.
- A subscriber can subscribe to multiple topics through wildcards.
- Messages are not queued; a subscriber will not receive matching messages published before the subscription was made.

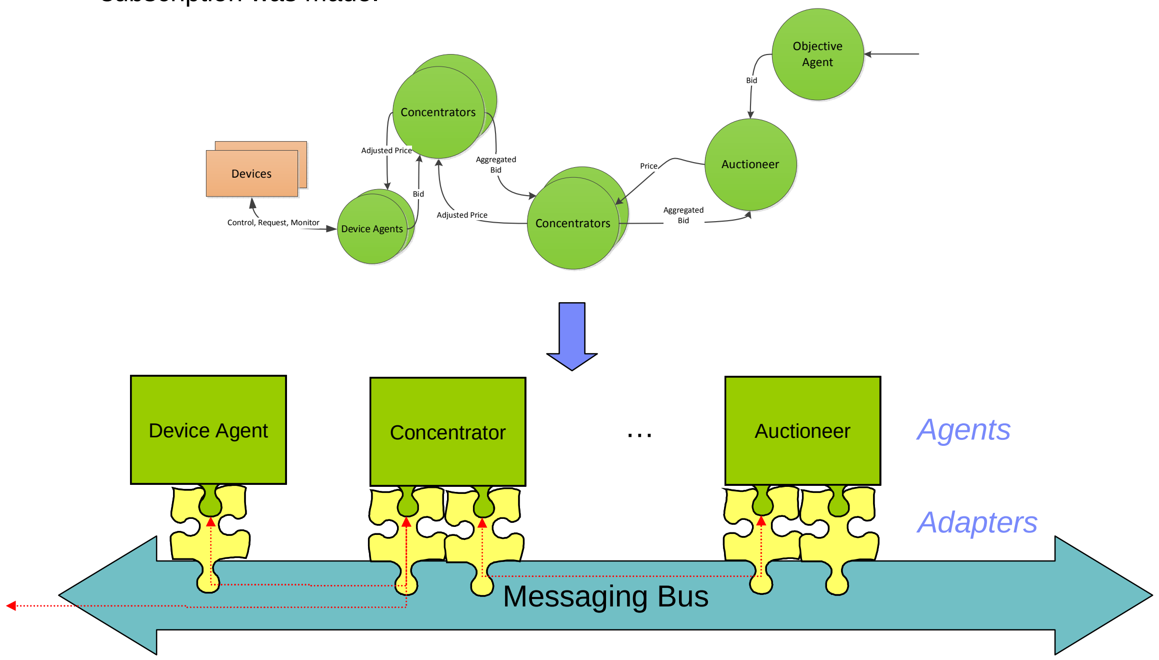


Figure 4 -- Agents, Adapters and the Messaging Bus

2.1.7.2 MQ Telemetry Transport

The messaging transport supported in this release is the MQ Telemetry publish/subscribe protocol[1], enabling distributed components residing on different locations with low bandwidth and processing power. The MQTT messages are transported over the TCP/IP network protocol, either in the plain for internal connections, or over SSL/TLS over the Internet. Agents connected to a MQ Telemetry broker act as MQTT-clients.

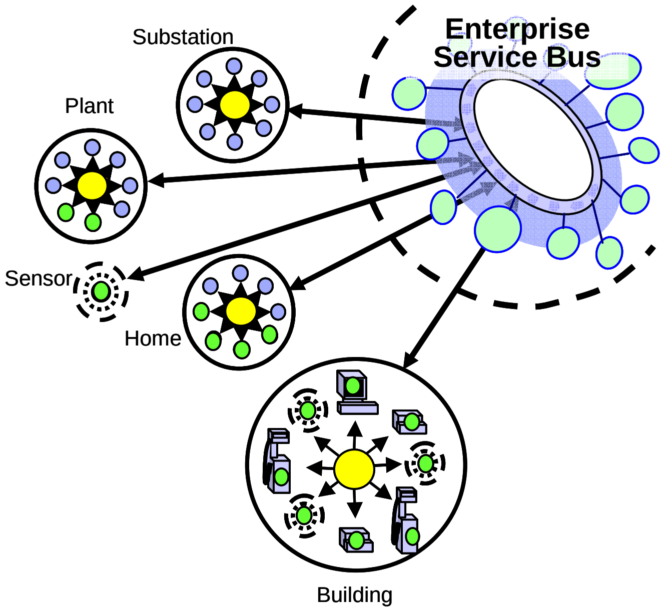


Figure 5 - MQ Telemetry Transport

[1] See http://mqtt.org for more information on the open standard MQ Telemetry protocol.

- MQTT is a machine-to-machine and "Internet of Things“ connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport by Arcom and IBM.
  - The MQTT protocol is public with a royalty-free license. See http://mqtt.org
  - Various open source client and broker implementations exist.
  - IBM ships commercial implementations, ranging from single device to highly scalable enterprise broker.
- Separate adapters for telemetry data collection, project specific interfaces, etc.

Currently popular languages (Java, C/C++, C#/.NET) and platforms are supported. Examples of commercial MQTT-client software implementations are:

- The redistributable Java and C clients supplied with IBM WebSphere MQ[3].
- The MQTTv5 client and Micro Broker supplied with IBM Lotus Expeditor, which provides C, Java and .NET API’s (see [2] for documentation and code samples). MicroBroker supports bridging between MQTT brokers.
- The open-source Eclipse Paho C and Java MQTTv3 client connects to, but does not include, a broker. Can connect to Mosquitto, WebSphere MQ, ...

PowerMatcher supports both the MQTTv3 protocol in Core, and the MQTTv5 protocol in Extension.

2.2 Agents, Adapters and the Messaging Bus

2.3 Subsystems and dependencies in PowerMatcher Core and Extension

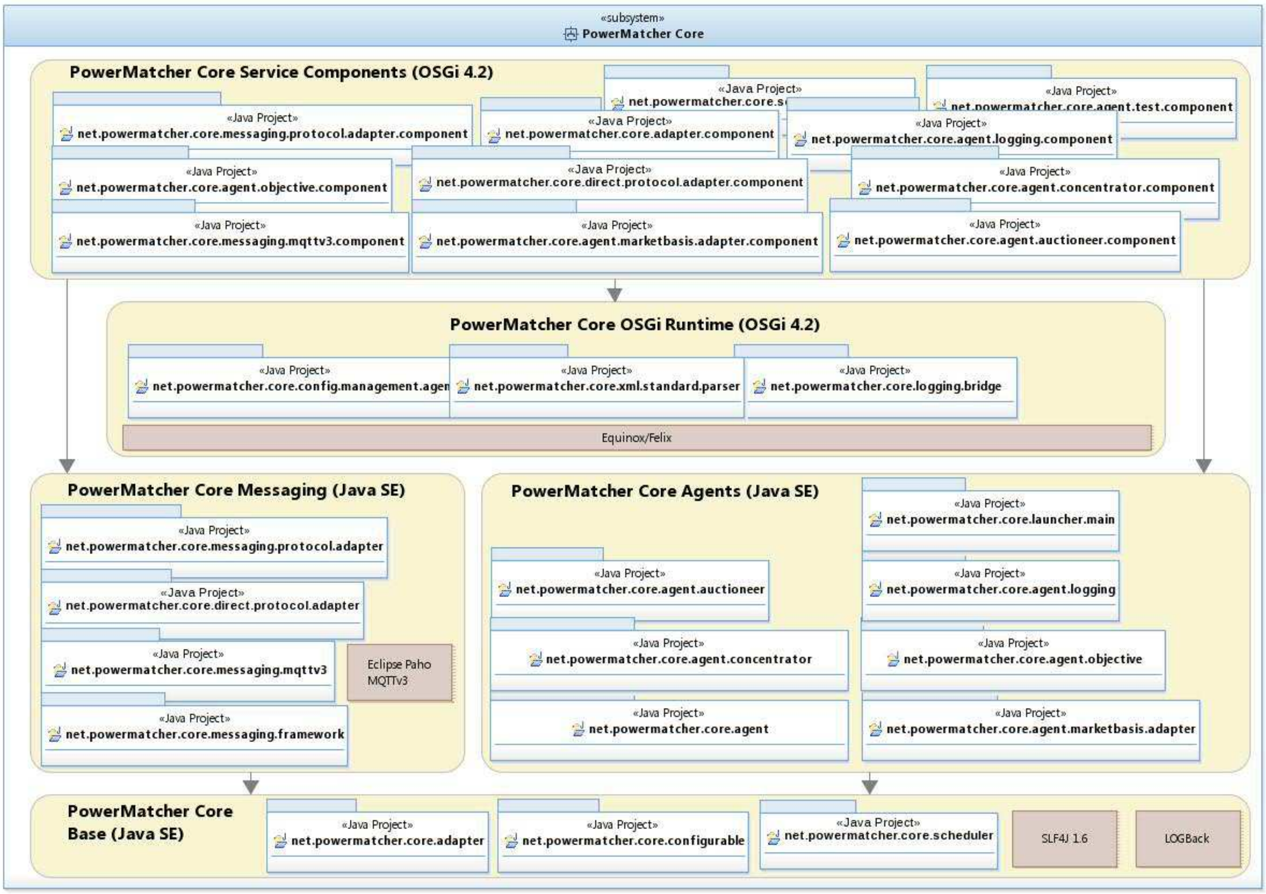

«subsystem»
PowerMatcher Core
PowerMatcher Core Service Components (OSGi 4.2)
«Java Project» net.powermatcher.core.s
«Java Project» net.powermatcher.core.agent.test.component
«Java Project» net.powermatcher.core.messaging.protocol.adapter.component
«Java Project» net.powermatcher.core.adapter.component
«Java Project» net.powermatcher.core.agent.logging.component
«Java Project» net.powermatcher.core.agent.objective.component
«Java Project» net.powermatcher.core.direct.protocol.adapter.component
«Java Project» net.powermatcher.core.agent.concentrator.component
«Java Project» net.powermatcher.core.messaging.mqttv3.component
«Java Project» net.powermatcher.core.agent.marketbasis.adapter.component
«Java Project» net.powermatcher.core.agent.auctioneer.component
PowerMatcher Core OSGi Runtime (OSGi 4.2)
«Java Project» net.powermatcher.core.config.management.agen
«Java Project» net.powermatcher.core.xml.standard.parser
«Java Project» net.powermatcher.core.logging.bridge
Equinox/Felix
PowerMatcher Core Messaging (Java SE)
«Java Project» net.powermatcher.core.messaging.protocol.adapter
«Java Project» net.powermatcher.core.direct.protocol.adapter
«Java Project» net.powermatcher.core.messaging.mqttv3
Eclipse Paho MQTTv3
«Java Project» net.powermatcher.core.messaging.framework
PowerMatcher Core Agents (Java SE)
«Java Project» net.powermatcher.core.launcher.main
«Java Project» net.powermatcher.core.agent.auctioneer
«Java Project» net.powermatcher.core.agent.logging
«Java Project» net.powermatcher.core.agent.concentrator
«Java Project» net.powermatcher.core.agent.objective
«Java Project» net.powermatcher.core.agent
«Java Project» net.powermatcher.core.agent.marketbasis.adapter
PowerMatcher Core Base (Java SE)
«Java Project» net.powermatcher.core.adapter
«Java Project» net.powermatcher.core.configurable
«Java Project» net.powermatcher.core.scheduler
SLF4J 1.6
LOGBack

Figure 6 - Subsystems and dependencies in PowerMatcher Core

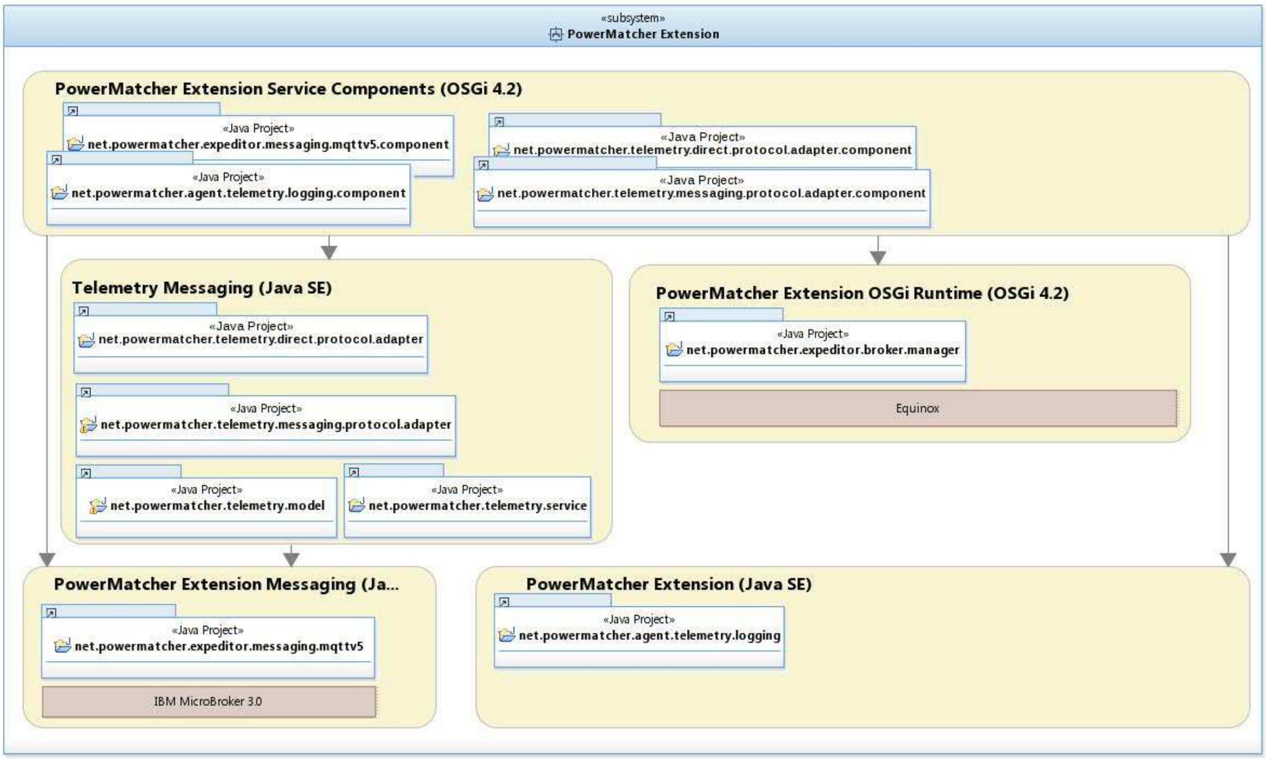

«subsystem»
PowerMatcher Extension
PowerMatcher Extension Service Components (OSGi 4.2)
«Java Project»
net.powermatcher.expeditor.messaging.mqttv5.component
«Java Project»
net.powermatcher.agent.telemetry.logging.component
«Java Project»
net.powermatcher.telemetry.direct.protocol.adapter.component
«Java Project»
net.powermatcher.telemetry.messaging.protocol.adapter.component
Telemetry Messaging (Java SE)
«Java Project»
net.powermatcher.telemetry.direct.protocol.adapter
«Java Project»
net.powermatcher.telemetry.messaging.protocol.adapter
«Java Project»
net.powermatcher.telemetry.model
«Java Project»
net.powermatcher.telemetry.service
PowerMatcher Extension OSGi Runtime (OSGi 4.2)
«Java Project»
net.powermatcher.expeditor.broker.manager
Equinox
PowerMatcher Extension Messaging (Ja...
«Java Project»
net.powermatcher.expeditor.messaging.mqttv5
IBM MicroBroker 3.0
PowerMatcher Extension (Java SE)
«Java Project»
net.powermatcher.agent.telemetry.logging

Figure 7 - Subsystems and dependencies in PowerMatcher Extension

2.4 The Base subsystem

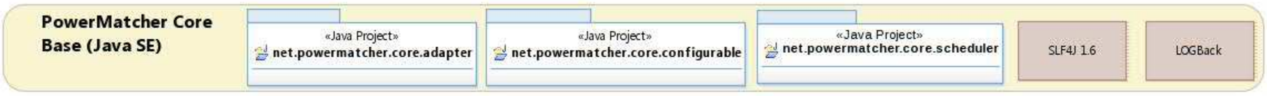

PowerMatcher Core Base (Java SE)
«Java Project»
net.powermatcher.core.adapter
«Java Project»
net.powermatcher.core.configurable
«Java Project»
net.powermatcher.core.scheduler
SLF4J 1.6
LOGBack

Figure 8 - Base subsystem

2.4.1 ConfigurationService, ConfigurableObject and BaseObject

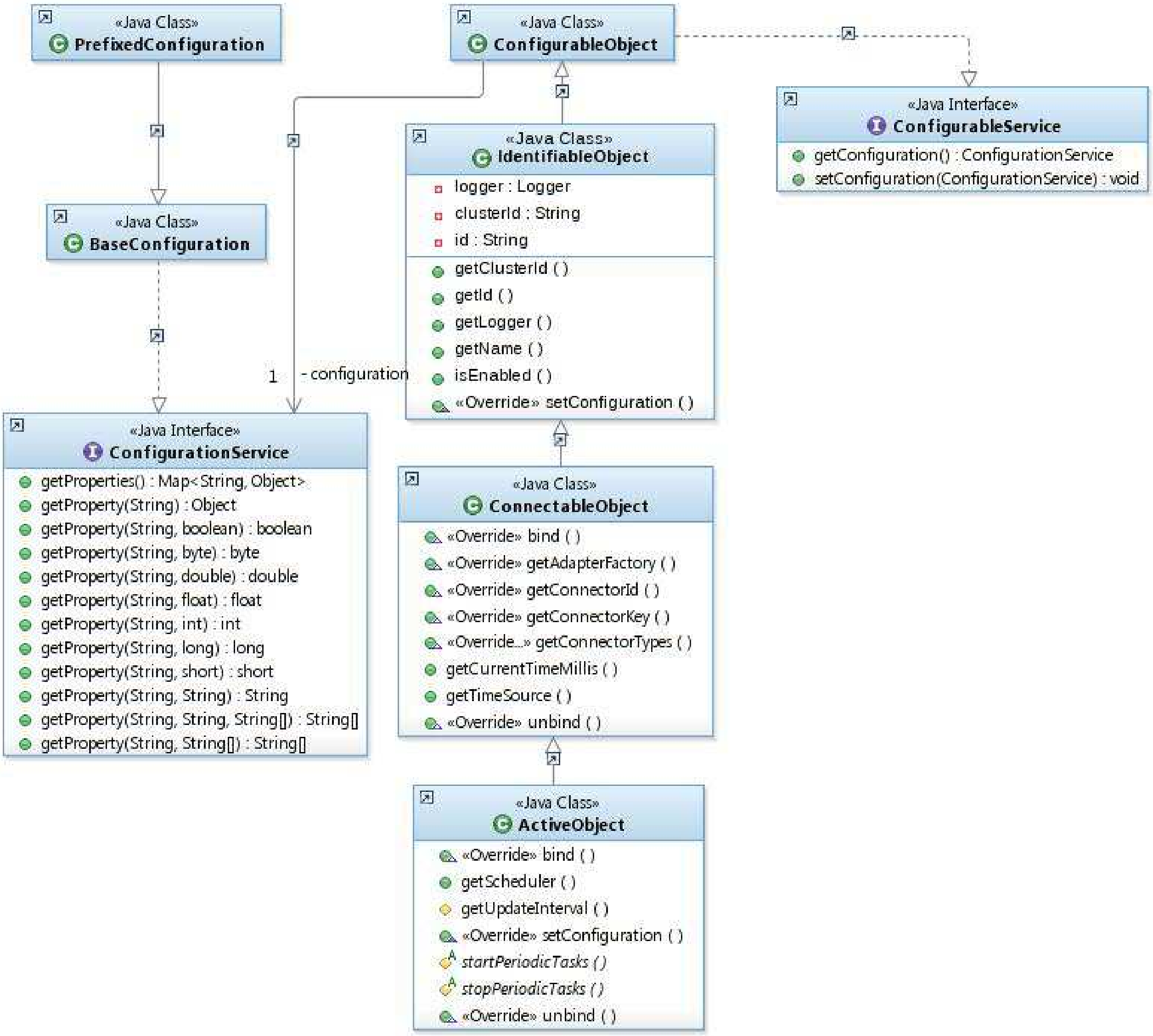

«Java Class»
PrefixedConfiguration
«Java Class»
ConfigurableObject
«Java Interface»
ConfigurableService
getConfiguration() : ConfigurationService
setConfiguration(ConfigurationService) : void
«Java Class»
IdentifiableObject
logger : Logger
clusterId : String
id : String
getClusterId ( )
getId ( )
getLogger ( )
getName ( )
isEnabled ( )
«Override» setConfiguration ( )
«Java Class»
BaseConfiguration
1
- configuration
«Java Interface»
ConfigurationService
getProperties() : Map<String, Object>
getProperty(String) : Object
getProperty(String, boolean) : boolean
getProperty(String, byte) : byte
getProperty(String, double) : double
getProperty(String, float) : float
getProperty(String, int) : int
getProperty(String, long) : long
getProperty(String, short) : short
getProperty(String, String) : String
getProperty(String, String, String[]) : String[]
getProperty(String, String[]) : String[]
«Java Class»
ConnectableObject
«Override» bind ( )
«Override» getAdapterFactory ( )
«Override» getConnectorId ( )
«Override» getConnectorKey ( )
«Override...» getConnectorTypes ( )
getCurrentTimeMillis ( )
getTimeSource ( )
«Override» unbind ( )
«Java Class»
ActiveObject
«Override» bind ( )
getScheduler ( )
getUpdateInterval ( )
«Override» setConfiguration ( )
startPeriodicTasks ( )
stopPeriodicTasks ( )
«Override» unbind ( )

Figure 9 - ConfigurationService, ConfigurableObject and BaseObject

2.4.2 Scheduler and Time Service

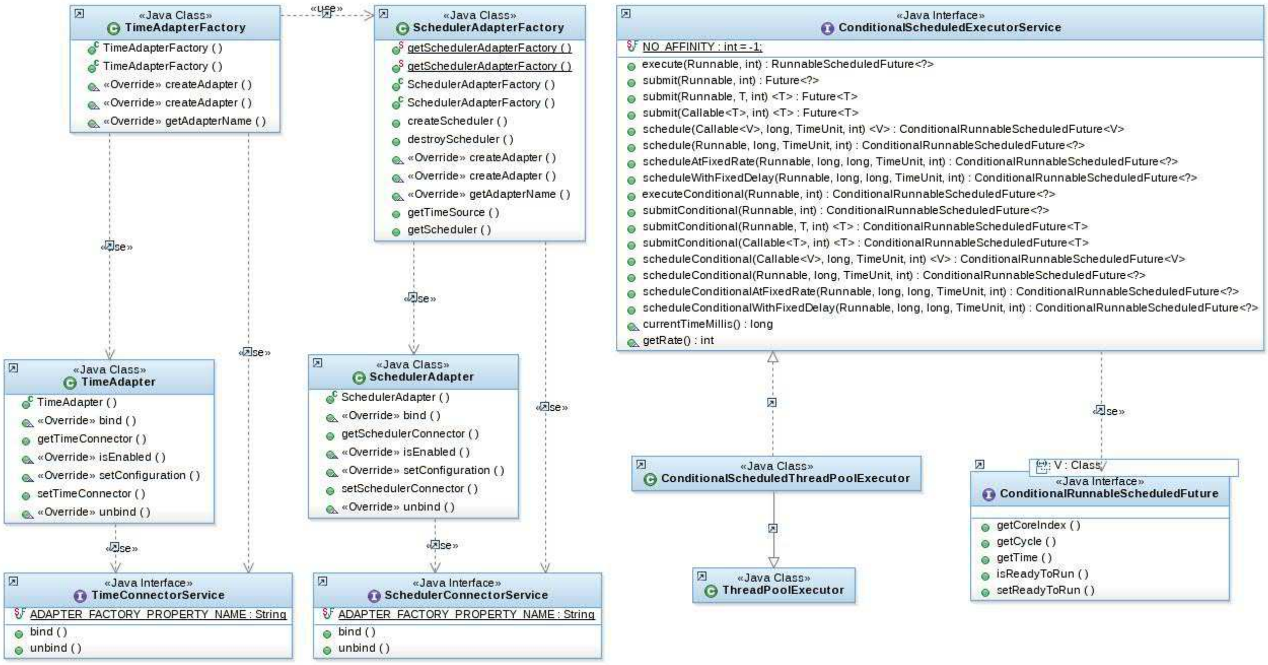

«Java Class»
TimeAdapterFactory
TimeAdapterFactory ()
TimeAdapterFactory ()
«Override» createAdapter ()
«Override» createAdapter ()
«Override» getAdapterName ()
«use»
«Java Class»
SchedulerAdapterFactory
getSchedulerAdapterFactory ()
getSchedulerAdapterFactory ()
SchedulerAdapterFactory ()
SchedulerAdapterFactory ()
createScheduler ()
destroyScheduler ()
«Override» createAdapter ()
«Override» createAdapter ()
«Override» getAdapterName ()
getTimeSource ()
getScheduler ()
«Java Interface»
ConditionalScheduledExecutorService
NO_AFFINITY : int = -1;
execute(Runnable, int) : RunnableScheduledFuture<?>
submit(Runnable, int) : Future<?>
submit(Runnable, T, int) <T> : Future<T>
submit(Callable<T>, int) <T> : Future<T>
schedule(Callable<V>, long, TimeUnit, int) <V> : ConditionalRunnableScheduledFuture<V>
schedule(Runnable, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
scheduleAtFixedRate(Runnable, long, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
scheduleWithFixedDelay(Runnable, long, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
executeConditional(Runnable, int) : ConditionalRunnableScheduledFuture<?>
submitConditional(Runnable, int) : ConditionalRunnableScheduledFuture<?>
submitConditional(Runnable, T, int) <T> : ConditionalRunnableScheduledFuture<T>
submitConditional(Callable<T>, int) <T> : ConditionalRunnableScheduledFuture<T>
scheduleConditional(Callable<V>, long, TimeUnit, int) <V> : ConditionalRunnableScheduledFuture<V>
scheduleConditional(Runnable, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
scheduleConditionalAtFixedRate(Runnable, long, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
scheduleConditionalWithFixedDelay(Runnable, long, long, TimeUnit, int) : ConditionalRunnableScheduledFuture<?>
currentTimeMillis() : long
getRate() : int
«use»
«use»
«use»
«use»
«Java Class»
TimeAdapter
TimeAdapter ()
«Override» bind ()
getTimeConnector ()
«Override» isEnabled ()
«Override» setConfiguration ()
setTimeConnector ()
«Override» unbind ()
«Java Class»
SchedulerAdapter
SchedulerAdapter ()
«Override» bind ()
getSchedulerConnector ()
«Override» isEnabled ()
«Override» setConfiguration ()
setSchedulerConnector ()
«Override» unbind ()
«use»
«Java Class»
ConditionalScheduledThreadPoolExecutor
V : Class
«Java Interface»
ConditionalRunnableScheduledFuture
getCoreIndex ()
getCycle ()
getTime ()
isReadyToRun ()
setReadyToRun ()
«use»
«use»
«Java Class»
ThreadPoolExecutor
«Java Interface»
TimeConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String
bind ()
unbind ()
«Java Interface»
SchedulerConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String
bind ()
unbind ()

Figure 10 – Scheduler and Time Service

2.4.3 Adapter and Connector

An Adapter provides one of possibly many implementations of services used by other components, including adapters.

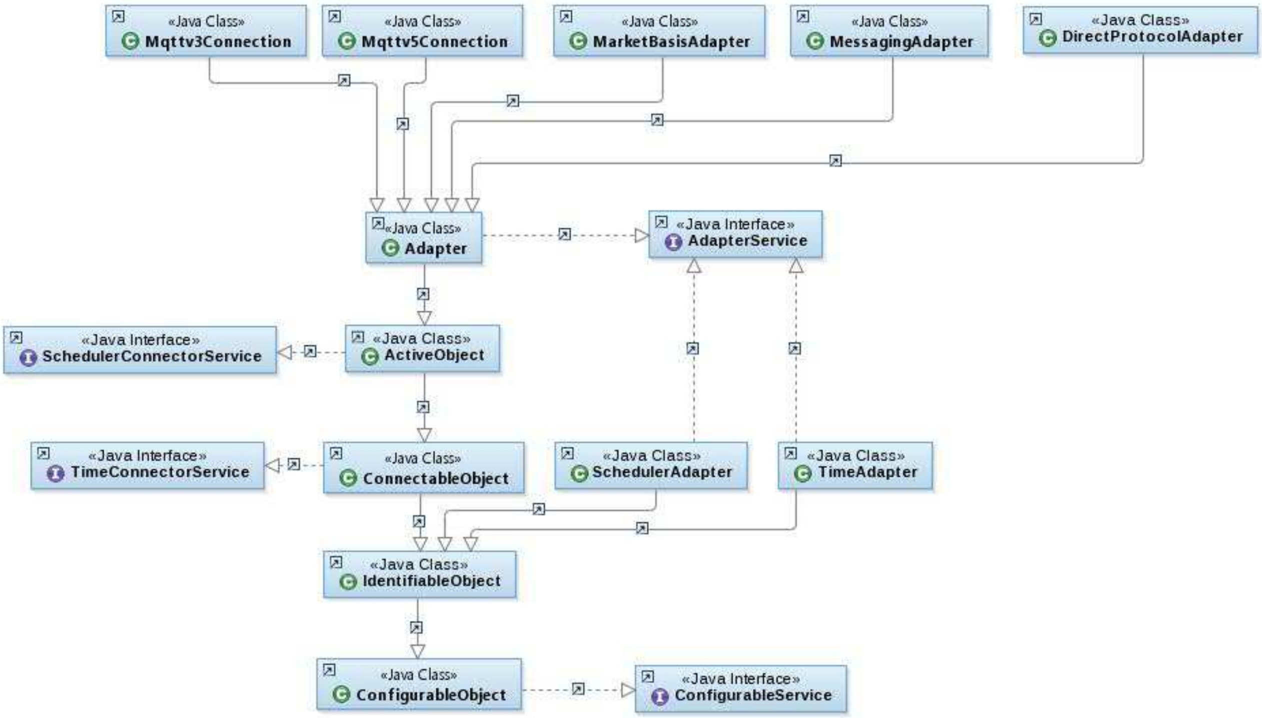


Figure 11 - Adapter hierarchy part 1

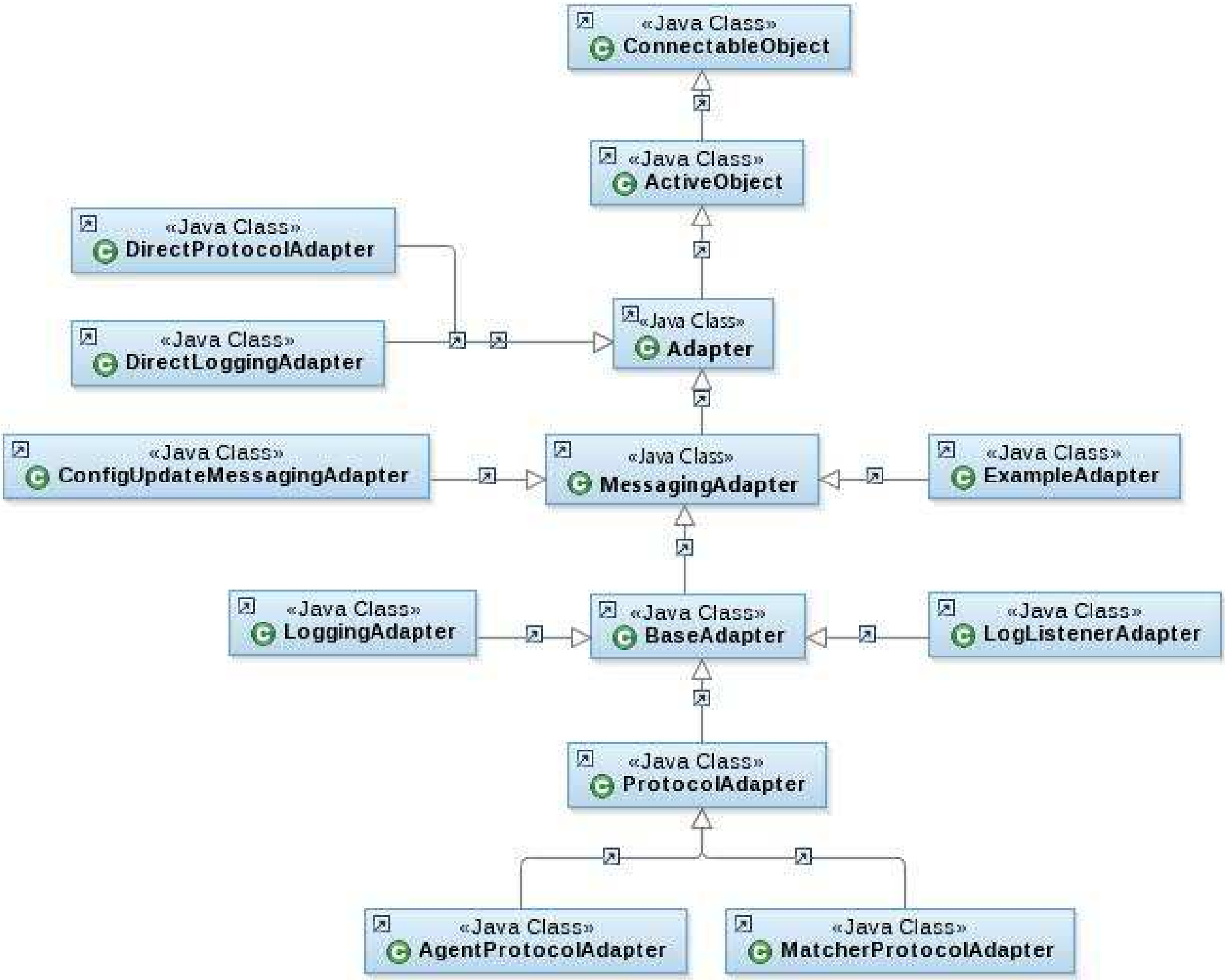

«Java Class» ConnectableObject
«Java Class» ActiveObject
«Java Class» DirectProtocolAdapter
«Java Class» DirectLoggingAdapter
«Java Class» Adapter
«Java Class» ConfigUpdateMessagingAdapter
«Java Class» MessagingAdapter
«Java Class» ExampleAdapter
«Java Class» LoggingAdapter
«Java Class» BaseAdapter
«Java Class» LogListenerAdapter
«Java Class» ProtocolAdapter
«Java Class» AgentProtocolAdapter
«Java Class» MatcherProtocolAdapter

Figure 12 - Adapter hierarchy part 2

An Adapter binds itself to the Connector of a component.

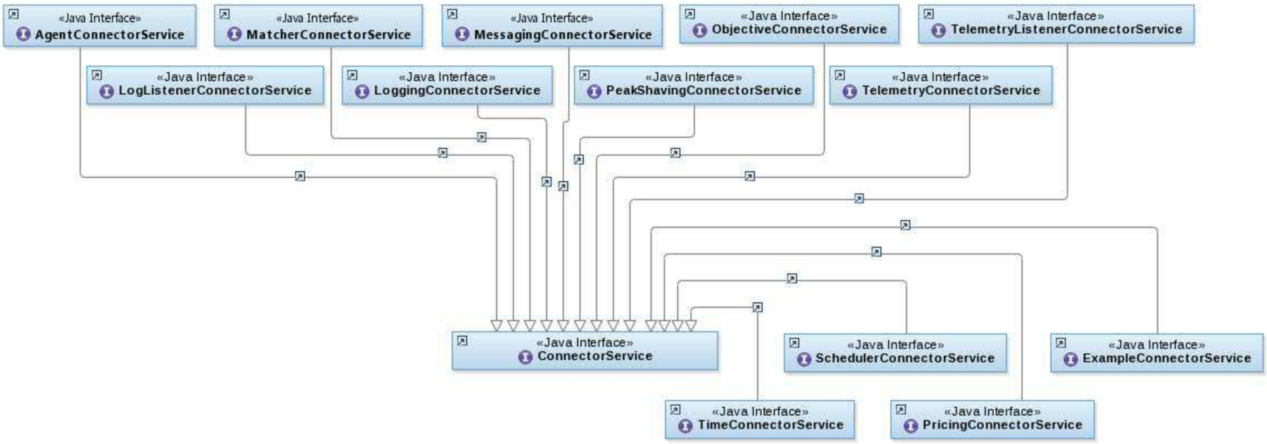
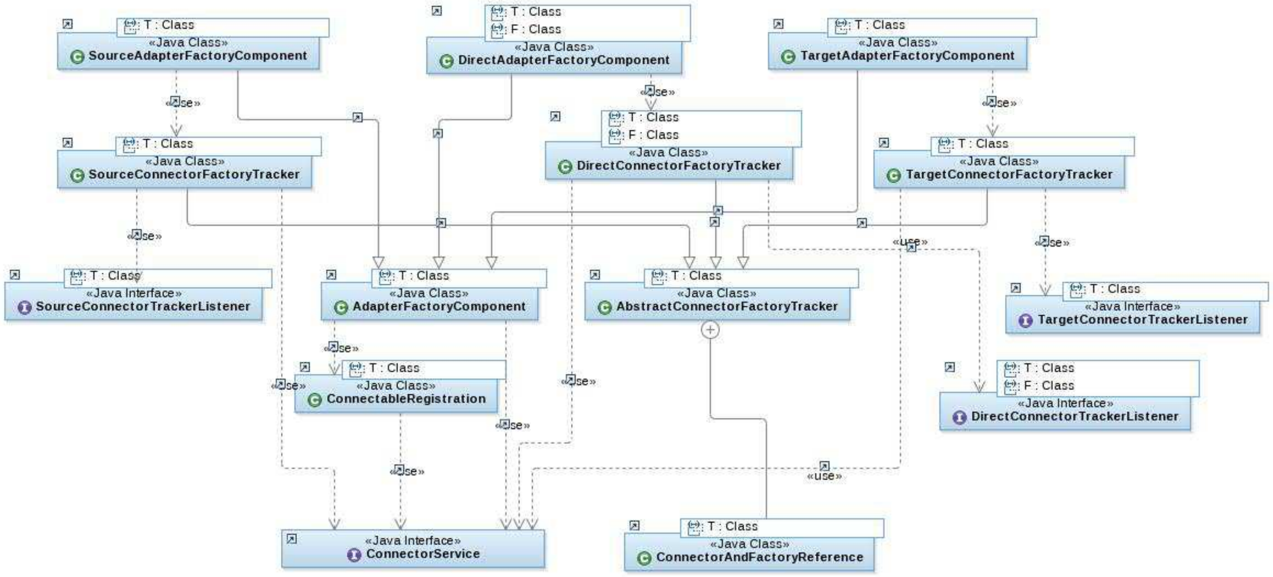

«Java Interface» AgentConnectorService
«Java Interface» MatcherConnectorService
«Java Interface» MessagingConnectorService
«Java Interface» ObjectiveConnectorService
«Java Interface» TelemetryListenerConnectorService
«Java Interface» LogListenerConnectorService
«Java Interface» LoggingConnectorService
«Java Interface» PeakShavingConnectorService
«Java Interface» TelemetryConnectorService
«Java Interface» ConnectorService
«Java Interface» SchedulerConnectorService
«Java Interface» ExampleConnectorService
«Java Interface» TimeConnectorService
«Java Interface» PricingConnectorService

Figure 13 - Connectors

More about this later when discussing specific adapters.

2.4.4 Connector, Connector Tracker and Adapter Factories

A Connector Tracker tracks the lifecycle of component connectors, in the role of a source connection, a target connection or a direct connection that directly binds a service consumer to a service provider. An Adapter Factory creates and configures adapters of a specific type. An Adapter Factory Component uses a Connector Tracker to create, bind, unbind and destroy adapters automatically as needed for other components.



2.5 The PowerMatcher Core subsystem

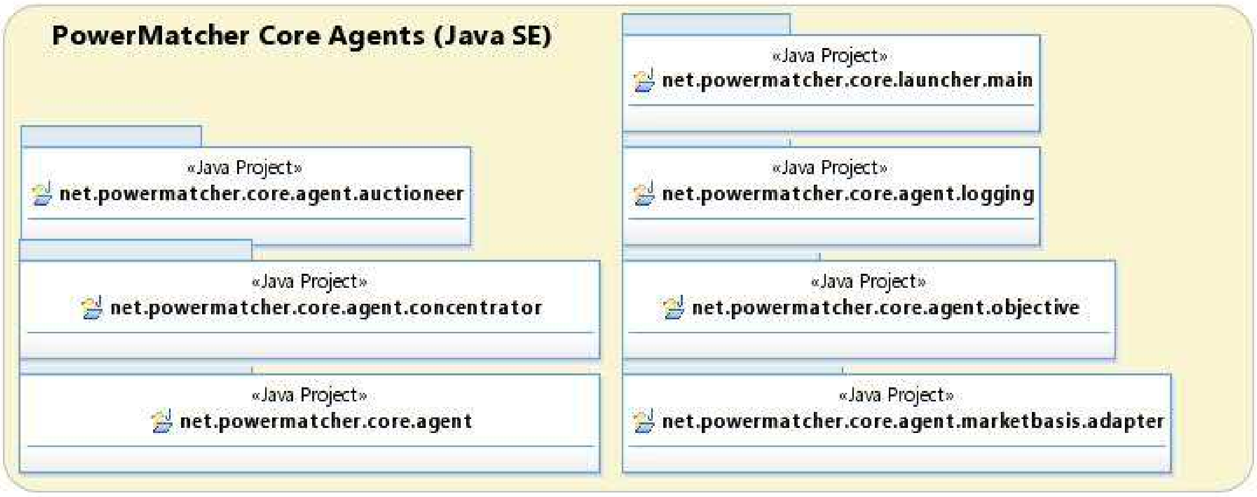

PowerMatcher Core Agents (Java SE)
«Java Project»
net.powermatcher.core.launcher.main
«Java Project»
net.powermatcher.core.agent.auctioneer
«Java Project»
net.powermatcher.core.agent.logging
«Java Project»
net.powermatcher.core.agent.concentrator
«Java Project»
net.powermatcher.core.agent.objective
«Java Project»
net.powermatcher.core.agent
«Java Project»
net.powermatcher.core.agent.marketbasis.adapter

Figure 14 - PowerMatcher Core subsystem

2.5.1 Agent and Matcher role interfaces and framework classes

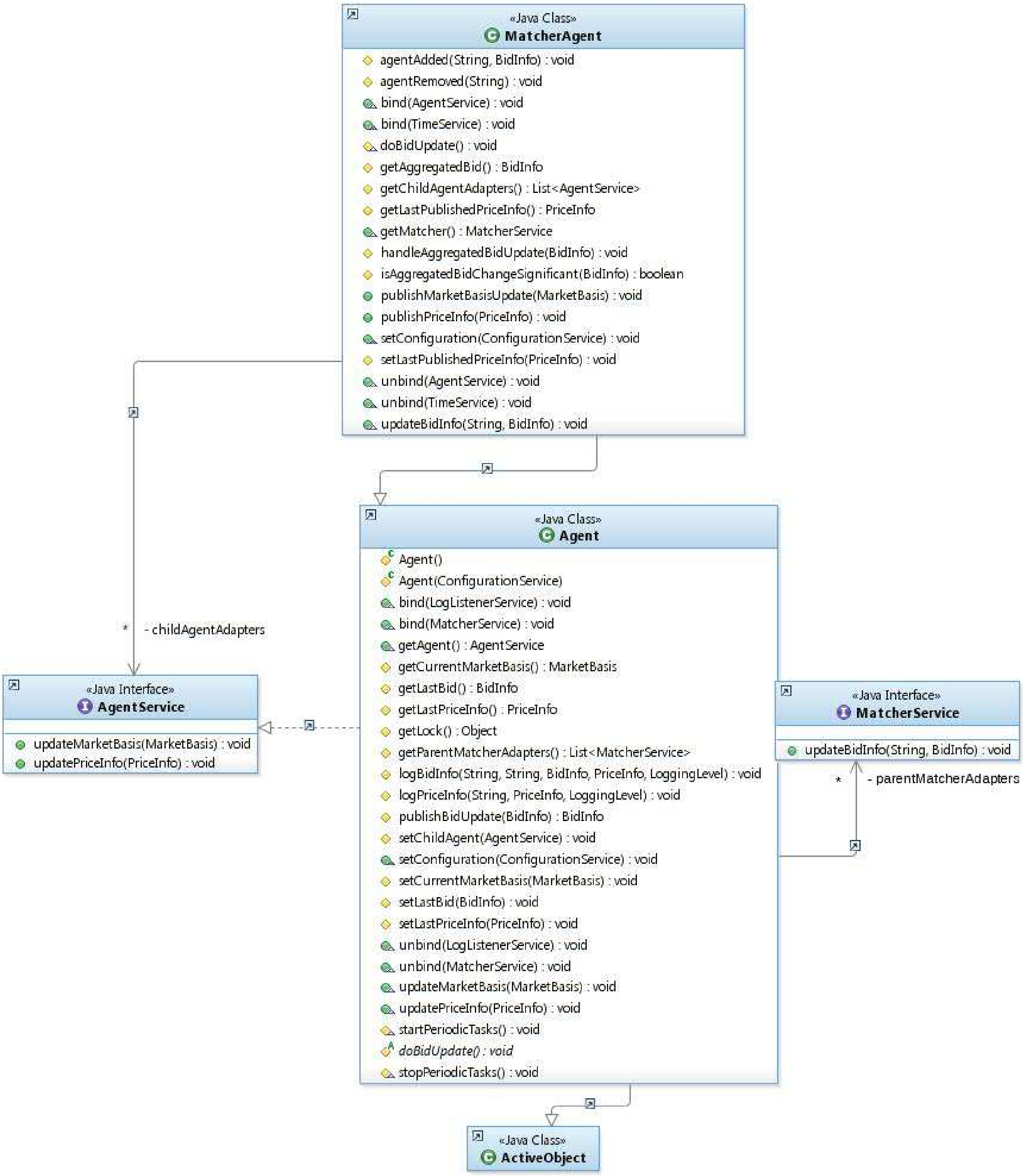

«Java Class»
MatcherAgent
agentAdded(String, BidInfo) : void
agentRemoved(String) : void
bind(AgentService) : void
bind(TimeService) : void
doBidUpdate() : void
getAggregatedBid() : BidInfo
getChildAgentAdapters() : List<AgentService>
getLastPublishedPriceInfo() : PriceInfo
getMatcher() : MatcherService
handleAggregatedBidUpdate(BidInfo) : void
isAggregatedBidChangeSignificant(BidInfo) : boolean
publishMarketBasisUpdate(MarketBasis) : void
publishPriceInfo(PriceInfo) : void
setConfiguration(ConfigurationService) : void
setLastPublishedPriceInfo(PriceInfo) : void
unbind(AgentService) : void
unbind(TimeService) : void
updateBidInfo(String, BidInfo) : void
«Java Class»
Agent
Agent()
Agent(ConfigurationService)
bind(LogListenerService) : void
bind(MatcherService) : void
getAgent() : AgentService
getCurrentMarketBasis() : MarketBasis
getLastBid() : BidInfo
getLastPriceInfo() : PriceInfo
getLock() : Object
getParentMatcherAdapters() : List<MatcherService>
logBidInfo(String, String, BidInfo, PriceInfo, LoggingLevel) : void
logPriceInfo(String, PriceInfo, LoggingLevel) : void
publishBidUpdate(BidInfo) : BidInfo
setChildAgent(AgentService) : void
setConfiguration(ConfigurationService) : void
setCurrentMarketBasis(MarketBasis) : void
setLastBid(BidInfo) : void
setLastPriceInfo(PriceInfo) : void
unbind(LogListenerService) : void
unbind(MatcherService) : void
updateMarketBasis(MarketBasis) : void
updatePriceInfo(PriceInfo) : void
startPeriodicTasks() : void
doBidUpdate() : void
stopPeriodicTasks() : void
*
- childAgentAdapters
«Java Interface»
AgentService
updateMarketBasis(MarketBasis) : void
updatePriceInfo(PriceInfo) : void
«Java Interface»
MatcherService
updateBidInfo(String, BidInfo) : void
*
- parentMatcherAdapters
«Java Class»
ActiveObject

Figure 15 - Agent and Matcher role interfaces and framework classes

Agents and Matchers are coupled via the Agent and Matcher interfaces. An agent implements the Agent interface, a matcher the Matcher interface.

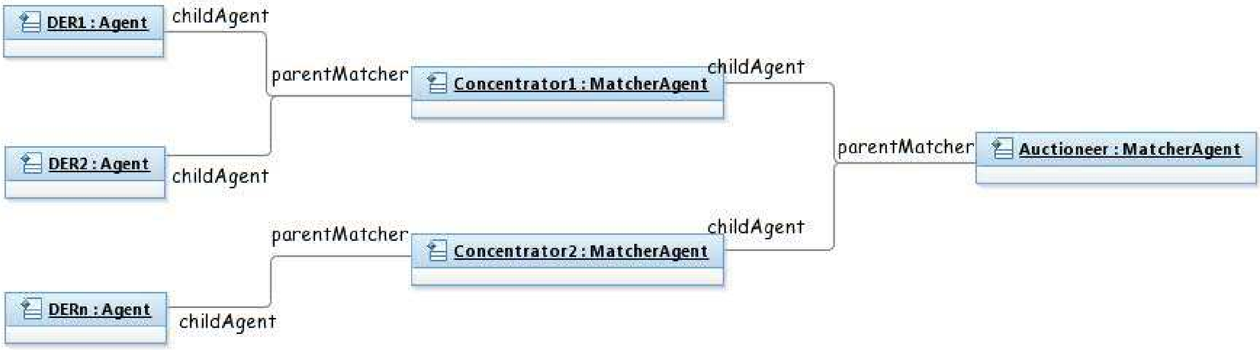


Figure 16 - Agent and Matcher logical role coupling

2.5.2 MarketBasis, PriceInfo, BidInfo, LogInfo and MarketBasisMapping

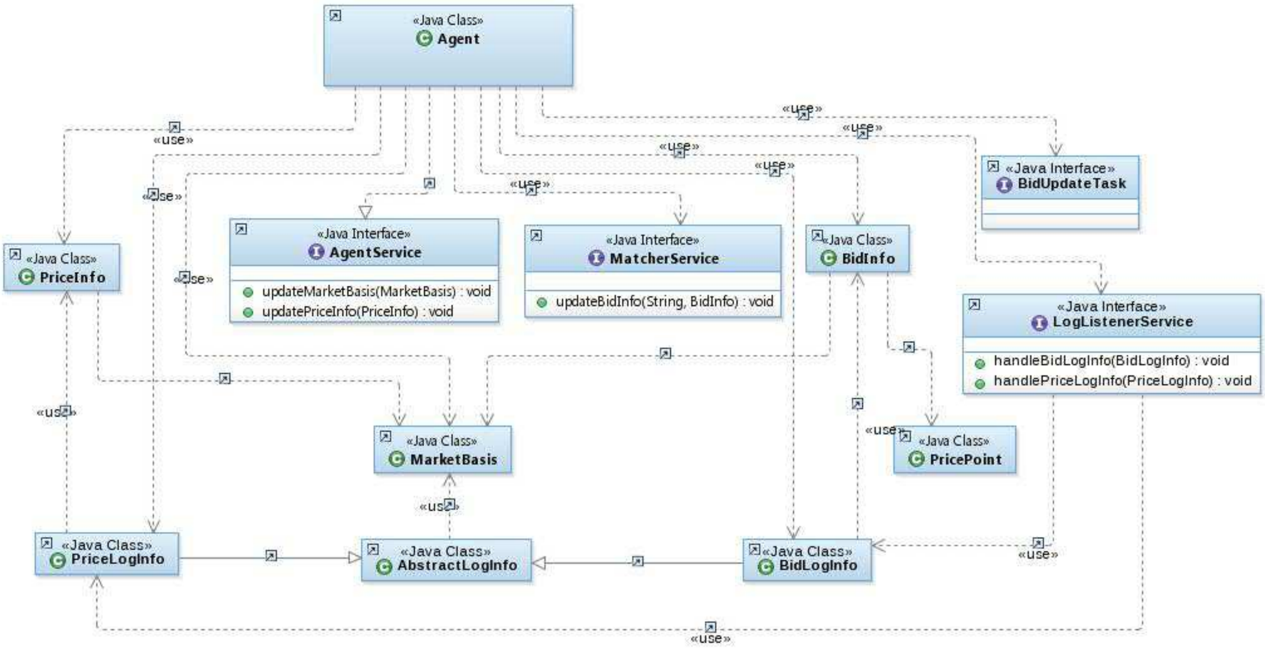

«Java Class» Agent
«use»
«Java Interface» BidUpdateTask
«Java Class» PriceInfo
«Java Interface» AgentService
updateMarketBasis(MarketBasis) : void
updatePriceInfo(PriceInfo) : void
«Java Interface» MatcherService
updateBidInfo(String, BidInfo) : void
«Java Class» BidInfo
«Java Interface» LogListenerService
handleBidLogInfo(BidLogInfo) : void
handlePriceLogInfo(PriceLogInfo) : void
«Java Class» MarketBasis
«Java Class» PricePoint
«Java Class» PriceLogInfo
«Java Class» AbstractLogInfo
«Java Class» BidLogInfo

Figure 17 - MarketBasis, PriceInfo, BidInfo and LogInfo overview

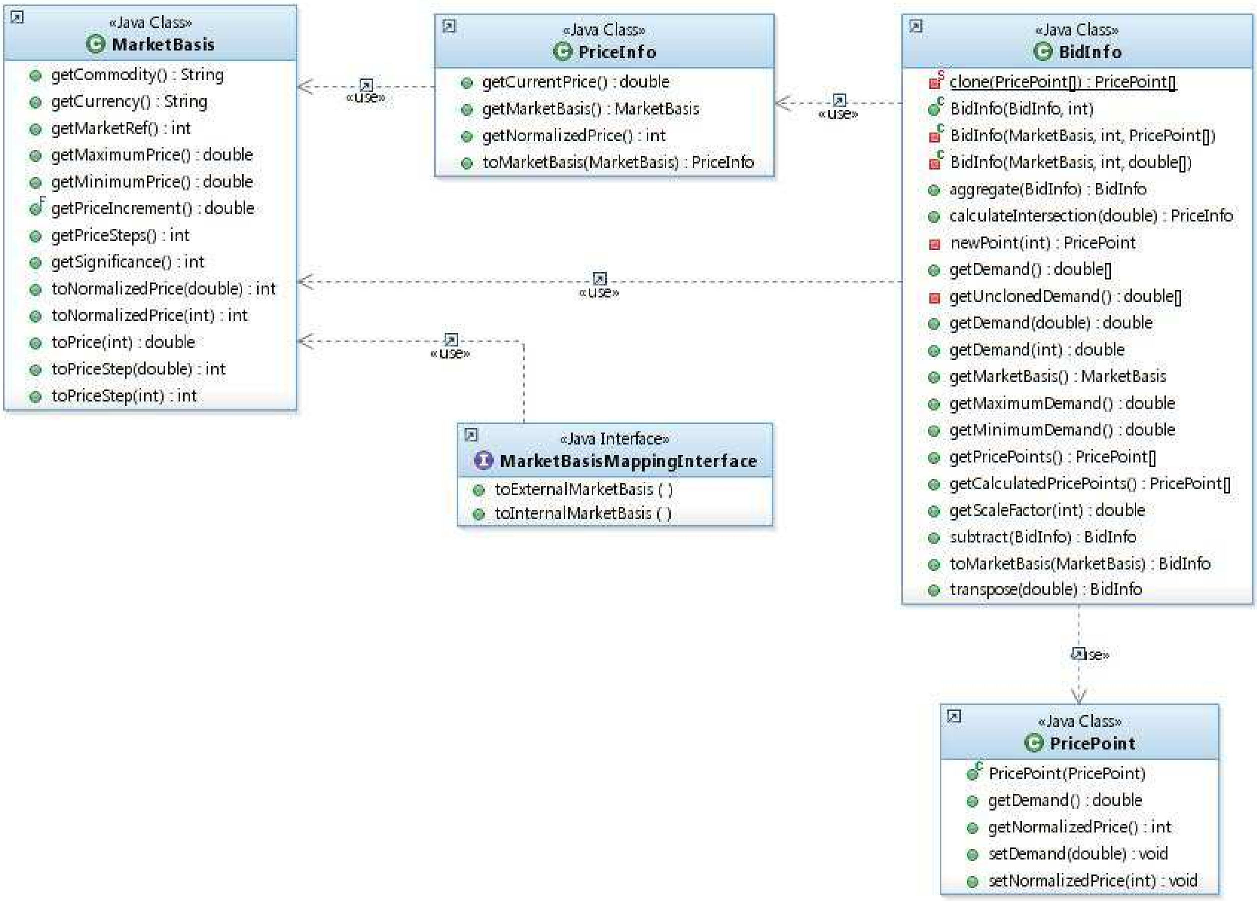

«Java Class»
MarketBasis
getCommodity() : String
getCurrency() : String
getMarketRef() : int
getMaximumPrice() : double
getMinimumPrice() : double
getPriceIncrement() : double
getPriceSteps() : int
getSignificance() : int
toNormalizedPrice(double) : int
toNormalizedPrice(int) : int
toPrice(int) : double
toPriceStep(double) : int
toPriceStep(int) : int
«use»
«Java Class»
PriceInfo
getCurrentPrice() : double
getMarketBasis() : MarketBasis
getNormalizedPrice() : int
toMarketBasis(MarketBasis) : PriceInfo
«use»
«Java Class»
BidInfo
clone(PricePoint[]) : PricePoint[]
BidInfo(BidInfo, int)
BidInfo(MarketBasis, int, PricePoint[])
BidInfo(MarketBasis, int, double[])
aggregate(BidInfo) : BidInfo
calculateIntersection(double) : PriceInfo
newPoint(int) : PricePoint
getDemand() : double[]
getUnclonedDemand() : double[]
getDemand(double) : double
getDemand(int) : double
getMarketBasis() : MarketBasis
getMaximumDemand() : double
getMinimumDemand() : double
getPricePoints() : PricePoint[]
getCalculatedPricePoints() : PricePoint[]
getScaleFactor(int) : double
subtract(BidInfo) : BidInfo
toMarketBasis(MarketBasis) : BidInfo
transpose(double) : BidInfo
«use»
«use»
«Java Interface»
MarketBasisMappingInterface
toExternalMarketBasis ( )
toInternalMarketBasis ( )
«use»
«Java Class»
PricePoint
PricePoint(PricePoint)
getDemand() : double
getNormalizedPrice() : int
setDemand(double) : void
setNormalizedPrice(int) : void

Figure 18 - MarketBasis, PriceInfo, BidInfo and MarketBasisMapping

2.5.2.1 Market basis,

In PowerMatcher, the market basis is defined by the following attributes:

Attribute	Type	Description
market basis reference	unsigned integer	Sequence number for the market basis, starting at 0. The sequence number may wrap between 0 and an arbitrary maximum value 2 ^ n -1, with a minimum of 2 ^ 8 -1 = 255.
commodity	string	Commodity for the market basis, for example "electricity".
currency	string	3 character ISO-4217 standard representation[2] of the currency for the market basis, for example "EUR".
minimum price	floating point	Minimum price for the market price.
maximum price	floating point	Maximum price for the market price.
price steps	unsigned integer > 0	The number of price steps between minimum and maximum price.
significance	unsigned integer	The number of significant digits in the market price[3], 0 = undefined.

Table 1 - Market basis attributes

The minimum price is counted as the first price step, the maximum price as the last. So, for example:

- Minimum price = € -0.20
- Maximum price = € 0.50
- Price step = € 0.10, number of steps is 8

[2] http://www.xe.com/iso4217.php

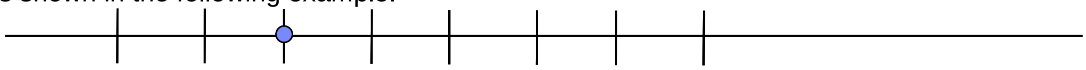
[3] Significance is currently not used in the PowerMatcher implementation, as the price steps attribute already constrains the precision. It could be used for printing numbers.

2.5.2.2 Price, normalized price unit and price step

The cluster price is normally expressed as a floating point value, but can also be expressed using a step index relative to the market basis. The price can be expressed as index in two alternative ways.

1. As price step, starting at index 0 for the minimum price.
2. As normalized price unit, with an index of 0 for the price closest to 0.0

This is shown in the following example.



Price	€ - 0.20	€ - 0.10	€ 0.00	€ 0.10	€ 0.20	€ 0.30	€ 0.40	€ 0.50
Step	0	1	2	3	4	5	6	7
Npu	-2	-1	0	1	2	3	4	5

Figure 19 - Price, price step and NPU example

2.5.2.3 Bid

A bid specifies the positive or negative demand over a price range, in a decreasing bid curve.
There are two ways to express bids in PowerMatcher.

1. The demand as vector with floating point value for each price step between minimum and maximum price as defined by the market basis.
2. As a line curve define by price points connected with straight line segments.

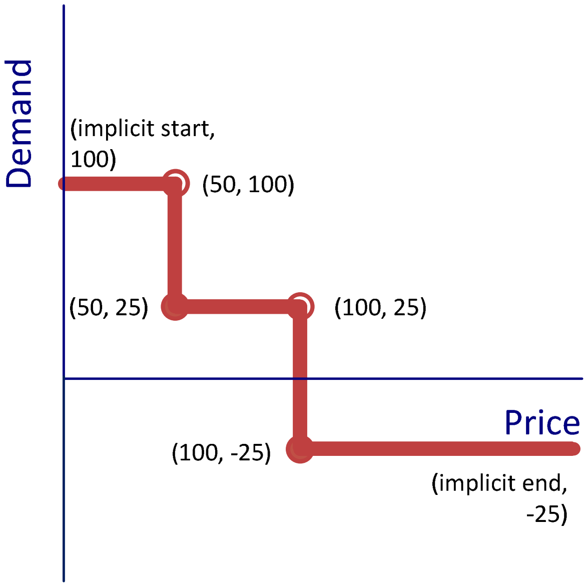


Figure 20 - Bid line curve example

The above example a bid curve is show that defines a price step with 2 points:

price[0]	= 50
demand[0]	= 100
price[1]	= 50
demand[1]	= 0

- Note that price 50 is used twice to define a price step using line curve semantics.
- The demand at price >= 50 is zero.
- The demand at price < 50 is +100.

Note that a flat bid curve can be represented by a single demand point where the price is any value in the range of the market basis.

The vector and line curve representation can be converted between one another.
In the PowerMatcher implementation the conversion is current lossless.

- From curve to vector, the demand at each price step is calculated as the intersection.
- From vector to curve, the points are calculated for the smallest number of matching line segments.

In a future implementation the conversion from vector to curve and vector to vector (in the case of aggregation) may be constrained by significance or a maximum for the number of price points.

2.5.3 AgentConnector, MatcherConnector and LoggingConnector

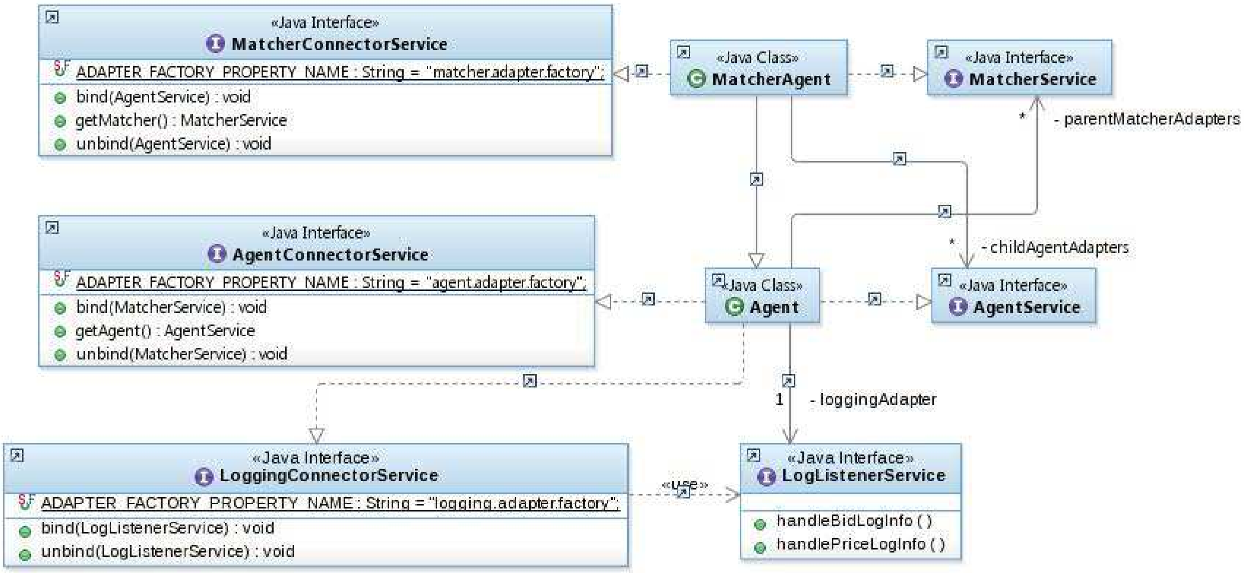

«Java Interface»
MatcherConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "matcher.adapter.factory";
bind(AgentService) : void
getMatcher() : MatcherService
unbind(AgentService) : void
«Java Class»
MatcherAgent
«Java Interface»
MatcherService
*
- parentMatcherAdapters
«Java Interface»
AgentConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "agent.adapter.factory";
bind(MatcherService) : void
getAgent() : AgentService
unbind(MatcherService) : void
*
- childAgentAdapters
«Java Class»
Agent
«Java Interface»
AgentService
1
- loggingAdapter
«Java Interface»
LoggingConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "logging.adapter.factory";
bind(LogListenerService) : void
unbind(LogListenerService) : void
«use»
«Java Interface»
LogListenerService
handleBidLogInfo ( )
handlePriceLogInfo ( )

Figure 21 – AgentConnector, MatcherConnector and LoggingConnector

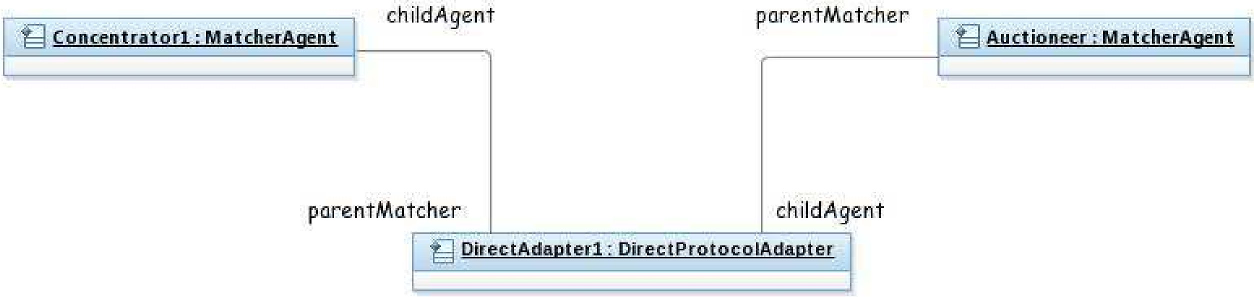

Concentrator1 : MatcherAgent
childAgent
parentMatcher
Auctioneer : MatcherAgent
parentMatcher
childAgent
DirectAdapter1 : DirectProtocolAdapter

Figure 22 - Agent and Matcher direct connection

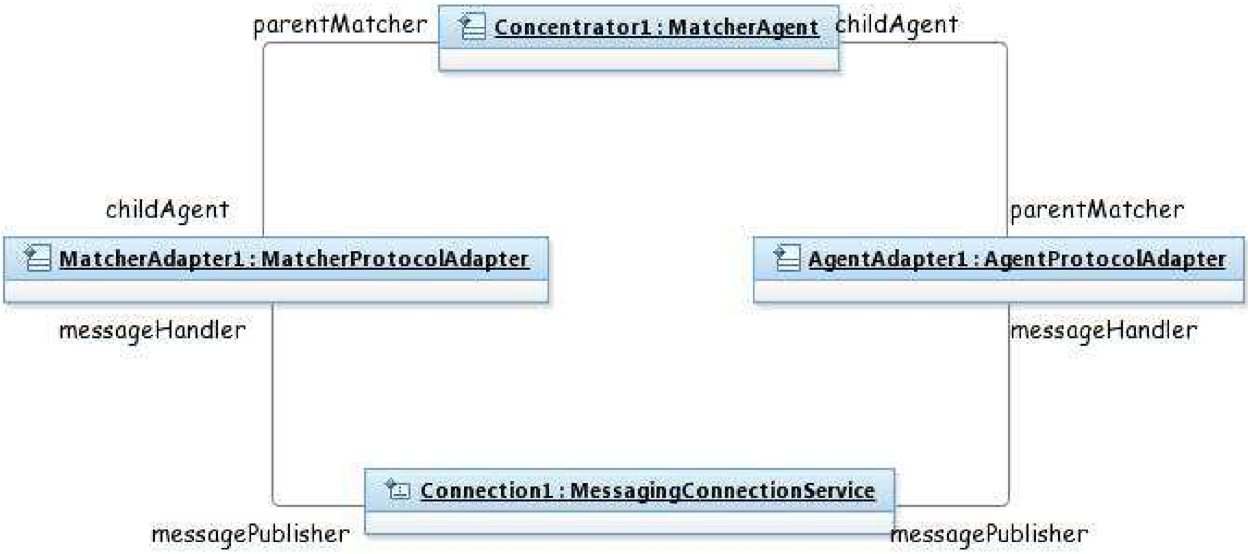

parentMatcher
Concentrator1 : MatcherAgent
childAgent
childAgent
parentMatcher
MatcherAdapter1 : MatcherProtocolAdapter
AgentAdapter1 : AgentProtocolAdapter
messageHandler
messageHandler
Connection1 : MessagingConnectionService
messagePublisher
messagePublisher

Figure 23 - Example object diagram for protocol and connection adapters

Note: one matcher adapter can proxy for multiple agents, and one agent can use multiple adapters.

2.5.4 Agent hierarchy in PowerMatcher Core and Extension

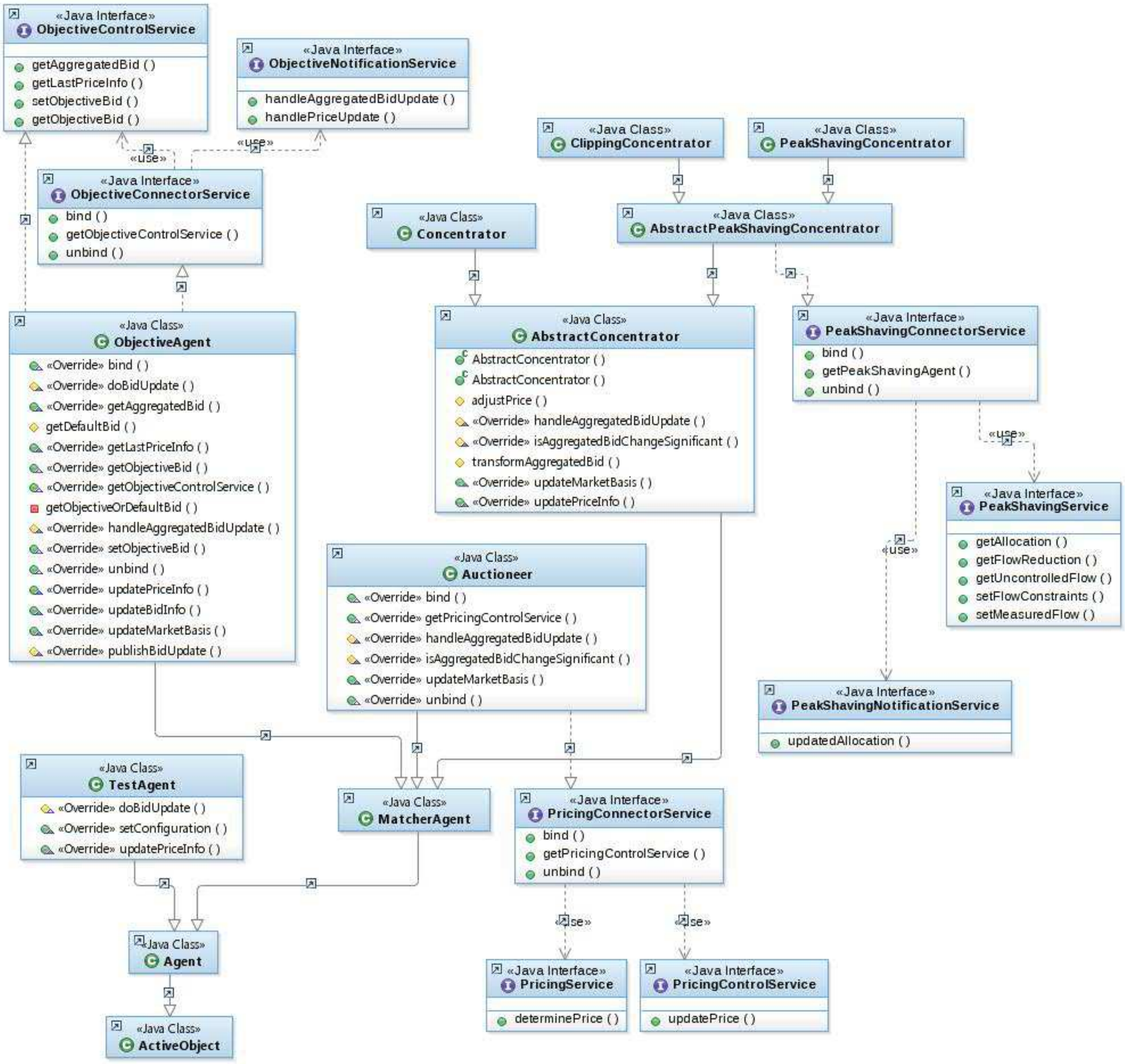


Figure 24 - Agent hierarchy in PowerMatcher Core and Extension

2.5.5 The Auctioneer

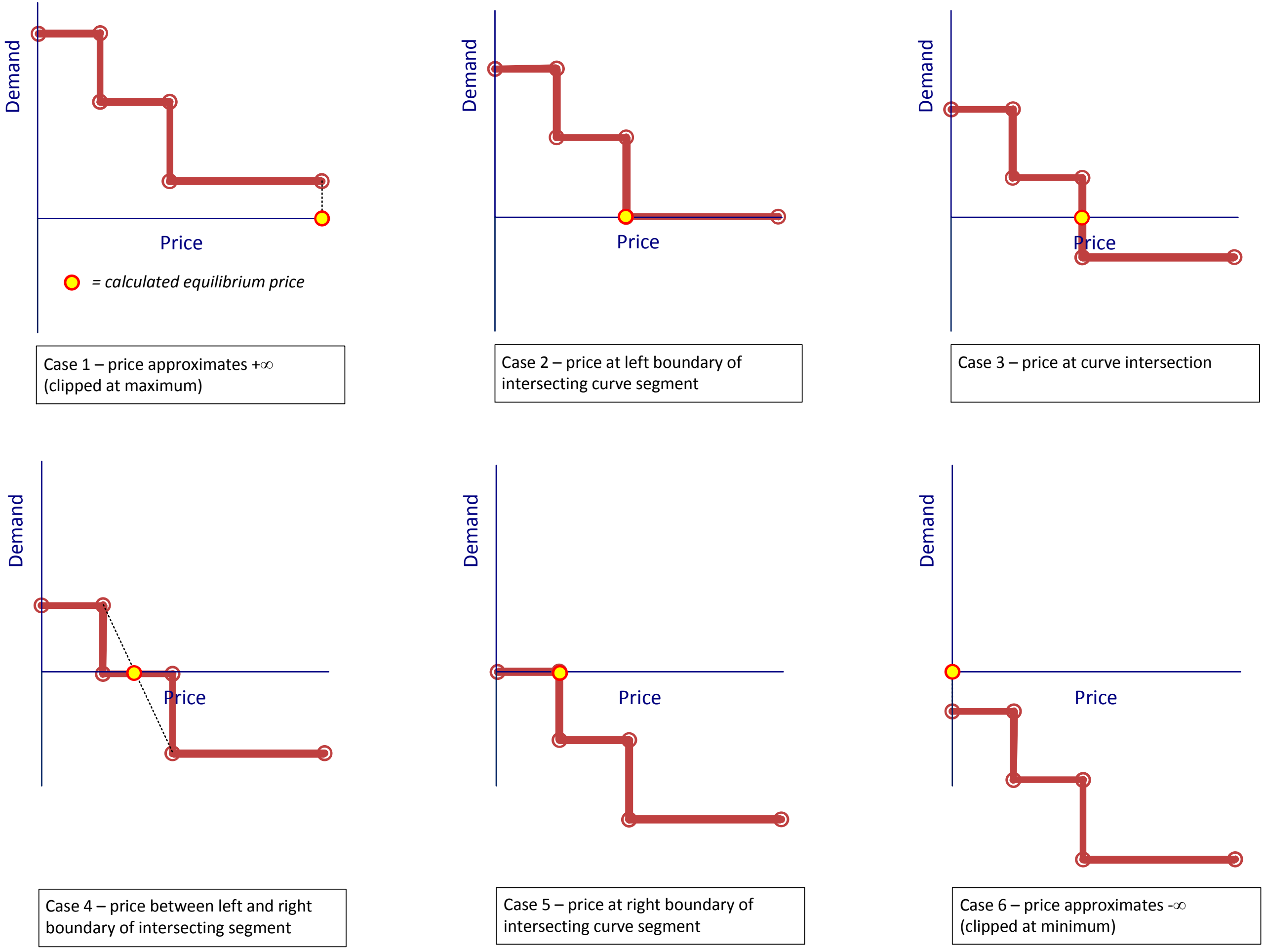
This PowerMatcher component performs the auctioning in the PowerMatcher cluster, and is the top level component in the dependency hierarchy of the PowerMatcher cluster. The auctioneer receives downstream bids from the cluster and publishes the cluster equilibrium price calculated from the most recently received set of bids. The price that is communicated contains a price and a market basis which enables the conversion to a normalized price or to any other market basis for other financial calculation purposes.

```
public class Auctioneer extends MatcherAgent {
   public Auctioneer(final ConfigurationService configuration) {
      super(configuration);
   }

   protected void handleAggregatedBidUpdate(final BidInfo newBid) {
      publishPriceInfo(newBid.calculateIntersection(0));
      publishBidUpdate(newBid);
   }

   protected boolean isAggregatedBidChangeSignificant(BidInfo newAggregatedBid) {
      return super.isAggregatedBidChangeSignificant(newAggregatedBid);
   }

}
```

Figure 25 - Auctioneer code fragment

1. Bids are aggregated as they are received (implemented in MatcherAgent)
2. Method handleAggregatedBidUpdate is called when:
   - The aggregation result is significantly different.
   - When the maximum time between updates has expired.
   - Immediately, when the maximum time between updates is 0.
3. Auctioneer publishes price update.
4. Auctioneer also publishes aggregated bid for objective/logging.

Equilibrium price calculation must return correct result for 6 different cases.



Case 1 – price approximates +∞ (clipped at maximum)

Case 2 – price at left boundary of intersecting curve segment

Case 3 – price at curve intersection

Case 4 – price between left and right boundary of intersecting segment

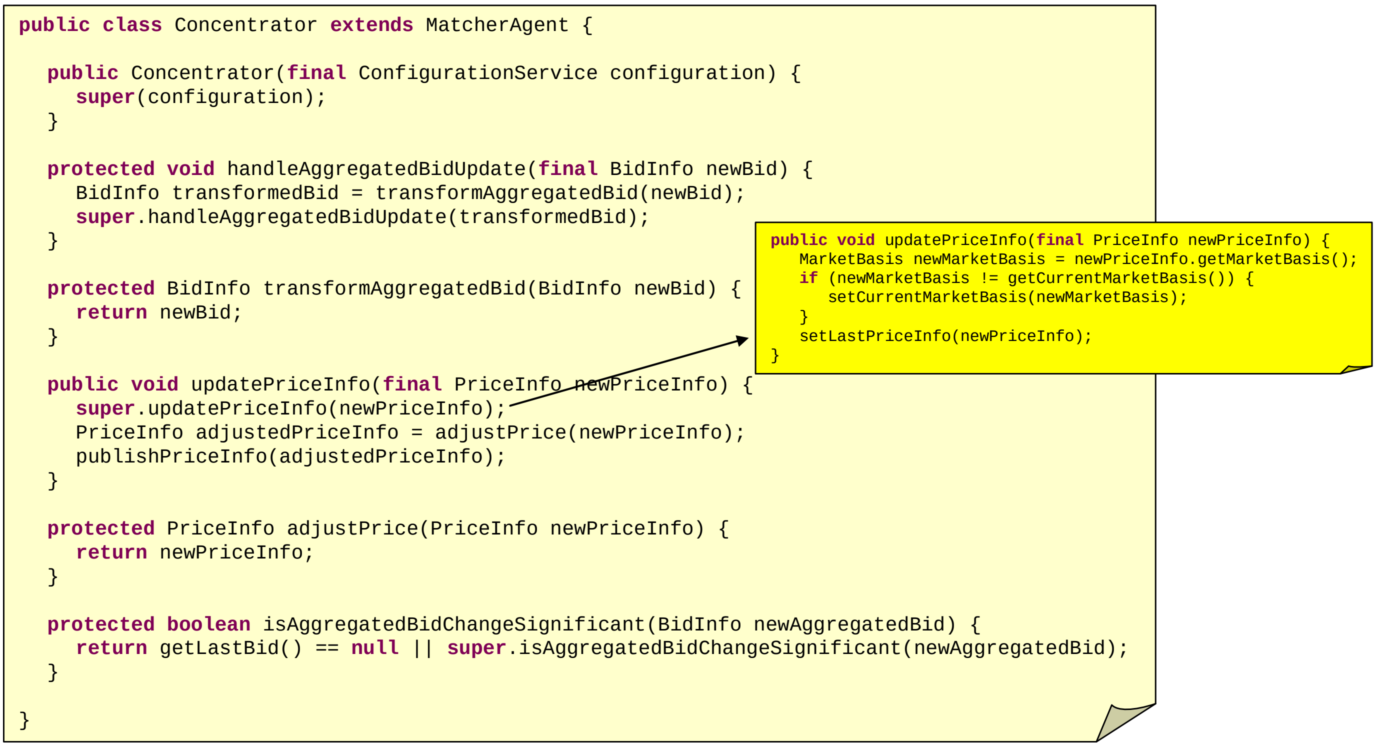
Case 5 – price at right boundary of intersecting curve segment

Case 6 – price approximates -∞ (clipped at minimum)

Figure 26 - Equilibrium price calculation cases

2.5.6 The Base Concentrator

A Concentrator is a PowerMatcher component that aggregates bids from its downstream cluster to a single bid that is forwarded upstream to another concentrator or to the auctioneer.



```
public class Concentrator extends MatcherAgent {

   public Concentrator(final ConfigurationService configuration) {
      super(configuration);
   }

   protected void handleAggregatedBidUpdate(final BidInfo newBid) {
      BidInfo transformedBid = transformAggregatedBid(newBid);
      super.handleAggregatedBidUpdate(transformedBid);
   }

   protected BidInfo transformAggregatedBid(BidInfo newBid) {
      return newBid;
   }

   public void updatePriceInfo(final PriceInfo newPriceInfo) {
      super.updatePriceInfo(newPriceInfo);
      PriceInfo adjustedPriceInfo = adjustPrice(newPriceInfo);
      publishPriceInfo(adjustedPriceInfo);
   }

   protected PriceInfo adjustPrice(PriceInfo newPriceInfo) {
      return newPriceInfo;
   }

   protected boolean isAggregatedBidChangeSignificant(BidInfo newAggregatedBid) {
      return getLastBid() == null || super.isAggregatedBidChangeSignificant(newAggregatedBid);
   }

}
```

```
public void updatePriceInfo(final PriceInfo newPriceInfo) {
   MarketBasis newMarketBasis = newPriceInfo.getMarketBasis();
   if (newMarketBasis != getCurrentMarketBasis()) {
      setCurrentMarketBasis(newMarketBasis);
   }
   setLastPriceInfo(newPriceInfo);
}
```

Figure 27 - Concentrator code fragment

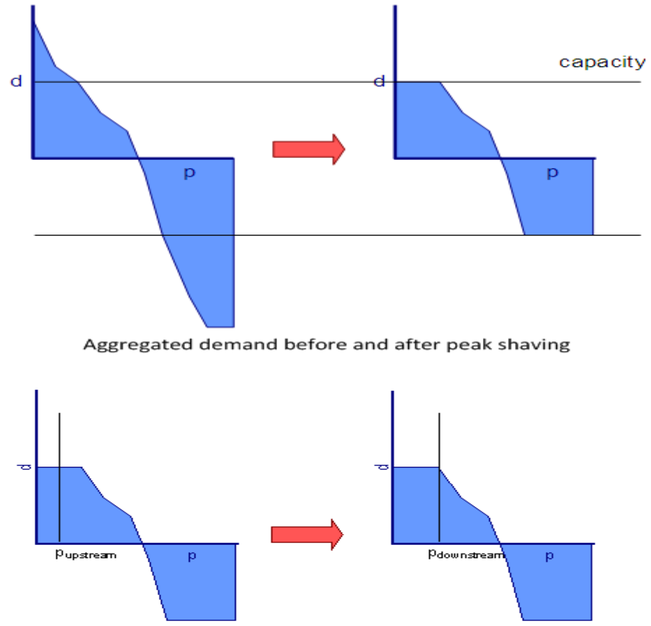
- In addition to Auctioneer, Concentrator receives price updates.
- Framework methods transform/adjust support specialized concentrator implementations, for example for peak shaving.

2.5.6.1 Peak Shaving Concentrators

In a PowerMatcher cluster the demand can be limited or reduced through market price adjustment. This is called peak shaving. Peak shaving is used to reduce the total demand for example at the level of the substation or building. Two different approaches to peak shaving are supported, covered in the following two subsections.

2.5.6.2 The Clipping Peak Shaving Concentrator

In the clipping approach a (variable) upper and/or lower capacity limit is set at the concentrator. Assume that the concentrator is receiving downstream bids that result in the aggregated bid curve as shown in the top-left curve of Figure 28.



Aggregated demand before and after peak shaving

Market price adjustment before and after peak shaving

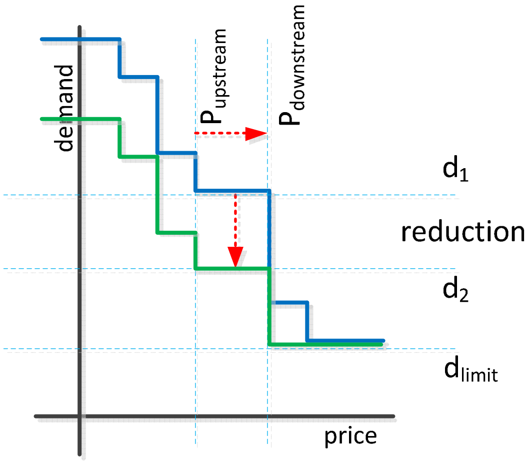
Figure 28 - Peak shaving by clipping

The concentrator will manipulate the downstream market price to fulfill the top-right bid curve. The adjusted aggregated bid curve is the clipped curve that the concentrator will submit upstream.
An example of the downstream price adjustment is shown in the bottom curves. If the upstream market price in the cluster is below the price of the cut point, the concentrator will increase the downstream market price to the cut point.

The capacity limits and the net effect of the resulting reduction are set by and reported back to an adapter. *Note: the connector interfaces and a sample adapter are to be implemented.*

2.5.6.3 The Reducing Peak Shaving Concentrator

In situations where only part of the total load on the substation (for example) is visible at the level of the PowerMatcher Concentrator, the approach most suitable the reduction transformation shown in Figure 29.



$P_{upstream}$	Market price received from auctioneer
$P_{downstream}$	Adjusted market price to achieve reduction from d1 to d2.
d_1	Demand of downstream cluster at market price from auctioneer.
d_2	Demand of downstream cluster after reduction.
d_{limit}	Lower limit for demand, for downstream resources that keep running at any price.

Figure 29 - Peak shaving through reduction

1. The substation total load is measured in near real time and reported to the concentrator via an adapter, together with the desired peak level to the Concentrator.
2. The Concentrator calculates if, and by how much, the load from the downstream cluster should be (further) reduced to keep the total load below the peak level.
3. The Concentrator reports the net effect of the requested reduction on the downstream load.

This is achieved as follows.
Assume that the Concentrator is receiving downstream bids that result in the aggregated bid curve as shown in the upper (blue) curve of Figure 29. The concentrator will adjust the downstream market price to transform the upper (blue) curve to the lower adjusted (green) curve. The transformed (green) bid curve is the curve that the concentrator will submit upstream to the auctioneer.

The desired power level d_2 is the reduced level from d_1, which is the power level at $P_{upstream}$ for the untransformed aggregated bid curve. The desired reduction is calculated from the total load and peak level coming from the substation, taking the reduction that is currently in effect into account.

The adjusted $P_{downstream}$ price is calculated by determining the price at the intersection of the desired power level d_2 and the untransformed aggregated bid curve. Every time $P_{upstream}$ changes, $P_{downstream}$ must be recalculated, even if the aggregated bid curve does not change.

Note that there is a limit to which a given peak shaving objective can actually be met.
If, for example, an energy resource must run (for example, once turned the device must be running for a minimum time), it will submit a bid for the demand of the heat pump at any price (the bid curve is a flat line). If in this situation a reduction is requested to reduce the demand below d_{limit}, the peak shaving concentrator will adjust the downstream market price to the maximum price, but this will not force the devices to turn off.

Note: the reducing peak shaving concentrator, the connector interfaces and a sample adapter are to be implemented.

2.5.7 The Base Objective Agent

An Objective agent is a PowerMatcher agent that influences the market price to achieve a certain cluster objective through submitting bids to the auctioneer.

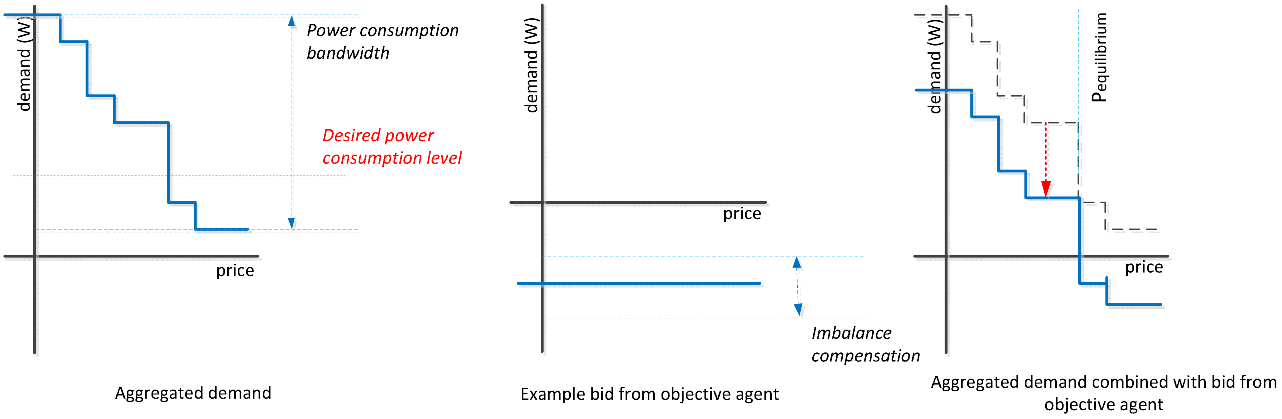

demand (W)
Power consumption bandwidth
Desired power consumption level
price
Aggregated demand
demand (W)
price
Imbalance compensation
Example bid from objective agent
demand (W)
Pequilibrium
price
Aggregated demand combined with bid from objective agent

Figure 30 - Aggregation of the objective bid

The downstream bids are aggregated by a Concentrator, as shown in the first example bid curve of Figure 30. This gives the bandwidth of the total power consumption of the heat pumps over the price range. The Objective Agent submits a flat bid curve to the Auctioneer to obtain the desired power consumption level, as for example in the second curve. The level of the bid curve the objective agent submits varies with the desired imbalance compensation. The Auctioneer aggregates the first and second bid curves and calculates the equilibrium price, as shown in the last bid curve.

2.5.8 The Logging Agent

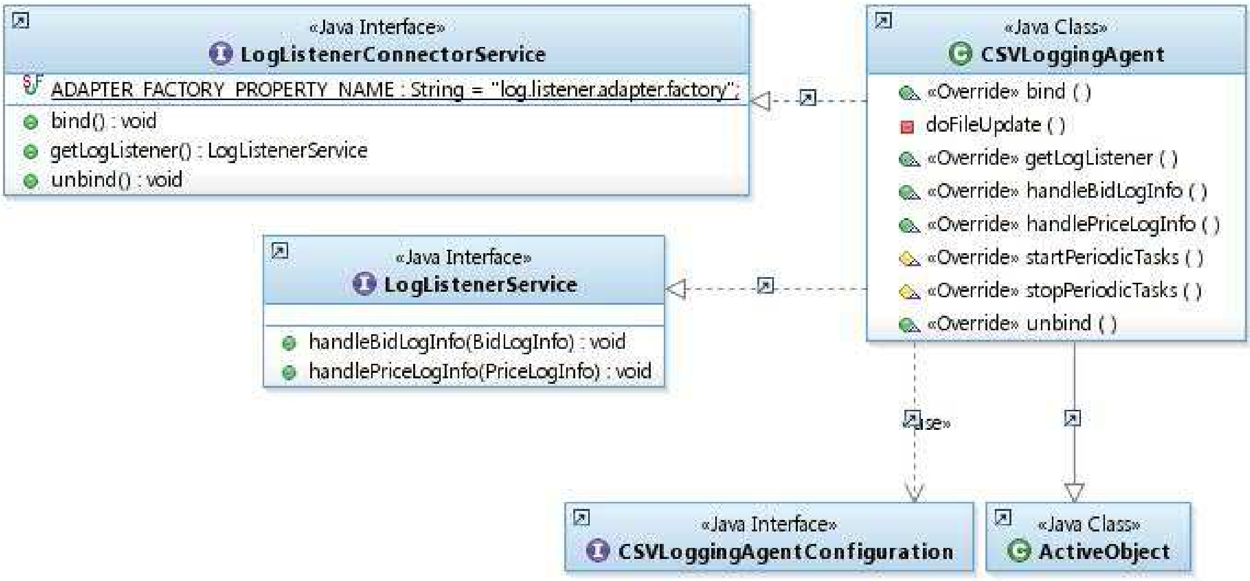

«Java Interface»
LogListenerConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "log.listener.adapter.factory";
bind() : void
getLogListener() : LogListenerService
unbind() : void
«Java Class»
CSVLoggingAgent
«Override» bind ( )
doFileUpdate ( )
«Override» getLogListener ( )
«Override» handleBidLogInfo ( )
«Override» handlePriceLogInfo ( )
«Override» startPeriodicTasks ( )
«Override» stopPeriodicTasks ( )
«Override» unbind ( )
«Java Interface»
LogListenerService
handleBidLogInfo(BidLogInfo) : void
handlePriceLogInfo(PriceLogInfo) : void
«use»
«Java Interface»
CSVLoggingAgentConfiguration
«Java Class»
ActiveObject

Figure 31 - The Logging Agent

The Logging Agent is an example agent that receives PowerMatcher logging events and Telemetry events and logs these events to files in the local file system in comma-separated value format.

2.6 The Messaging subsystem

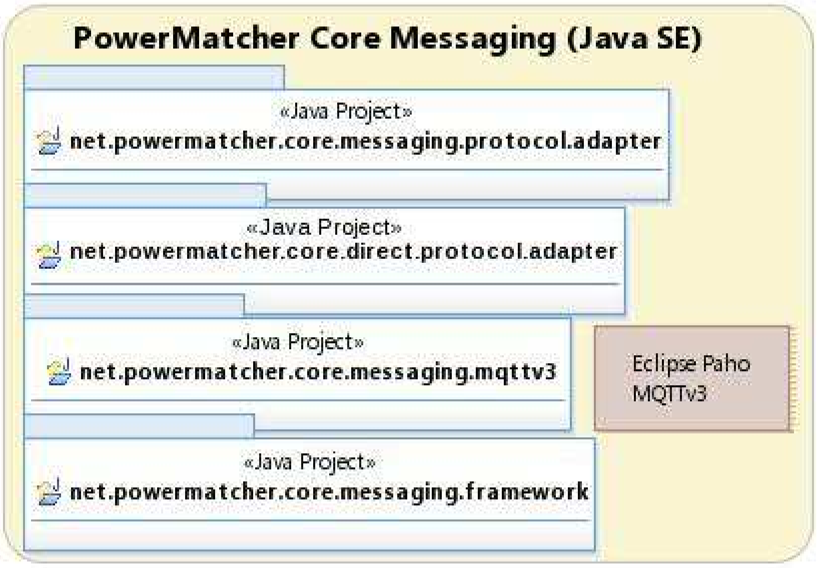

PowerMatcher Core Messaging (Java SE)
«Java Project»
net.powermatcher.core.messaging.protocol.adapter
«Java Project»
net.powermatcher.core.direct.protocol.adapter
«Java Project»
net.powermatcher.core.messaging.mqttv3
Eclipse Paho MQTTv3
«Java Project»
net.powermatcher.core.messaging.framework

Figure 32 - Messaging subsystem

2.6.1 MessagingAdapter, MessagingConnection and MessagingConnector

A MessagingAdapter is the framework class for protocol, telemetry and custom messaging interfaces. It requires a messaging connection (another adapter).

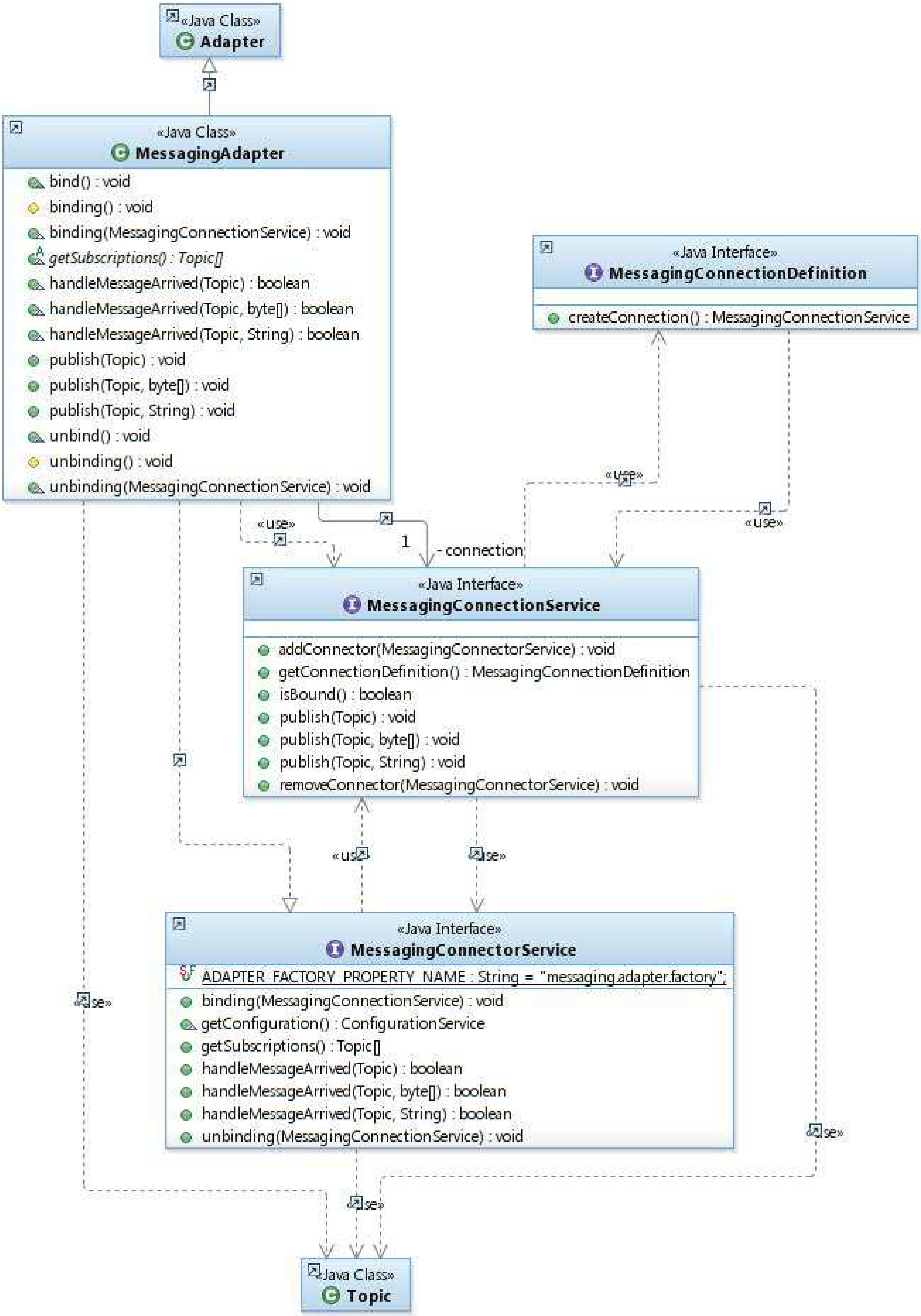

«Java Class»
Adapter
«Java Class»
MessagingAdapter
bind() : void
binding() : void
binding(MessagingConnectionService) : void
getSubscriptions() : Topic[]
handleMessageArrived(Topic) : boolean
handleMessageArrived(Topic, byte[]) : boolean
handleMessageArrived(Topic, String) : boolean
publish(Topic) : void
publish(Topic, byte[]) : void
publish(Topic, String) : void
unbind() : void
unbinding() : void
unbinding(MessagingConnectionService) : void
«Java Interface»
MessagingConnectionDefinition
createConnection() : MessagingConnectionService
«use»
«use»
«use»
1
- connection
«Java Interface»
MessagingConnectionService
addConnector(MessagingConnectorService) : void
getConnectionDefinition() : MessagingConnectionDefinition
isBound() : boolean
publish(Topic) : void
publish(Topic, byte[]) : void
publish(Topic, String) : void
removeConnector(MessagingConnectorService) : void
«use»
«use»
«Java Interface»
MessagingConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "messaging.adapter.factory";
binding(MessagingConnectionService) : void
getConfiguration() : ConfigurationService
getSubscriptions() : Topic[]
handleMessageArrived(Topic) : boolean
handleMessageArrived(Topic, byte[]) : boolean
handleMessageArrived(Topic, String) : boolean
unbinding(MessagingConnectionService) : void
«use»
«use»
«use»
«Java Class»
Topic

Figure 33 - MessagingAdapter, MessagingConnection and MessagingConnector

2.6.2 MQ Telemetry Messaging Connections

Two messaging connection components have been implemented. One for the open source Eclipse Paho MQTTv3 client, one for the IBM MicroBroker MQTTv5 client.

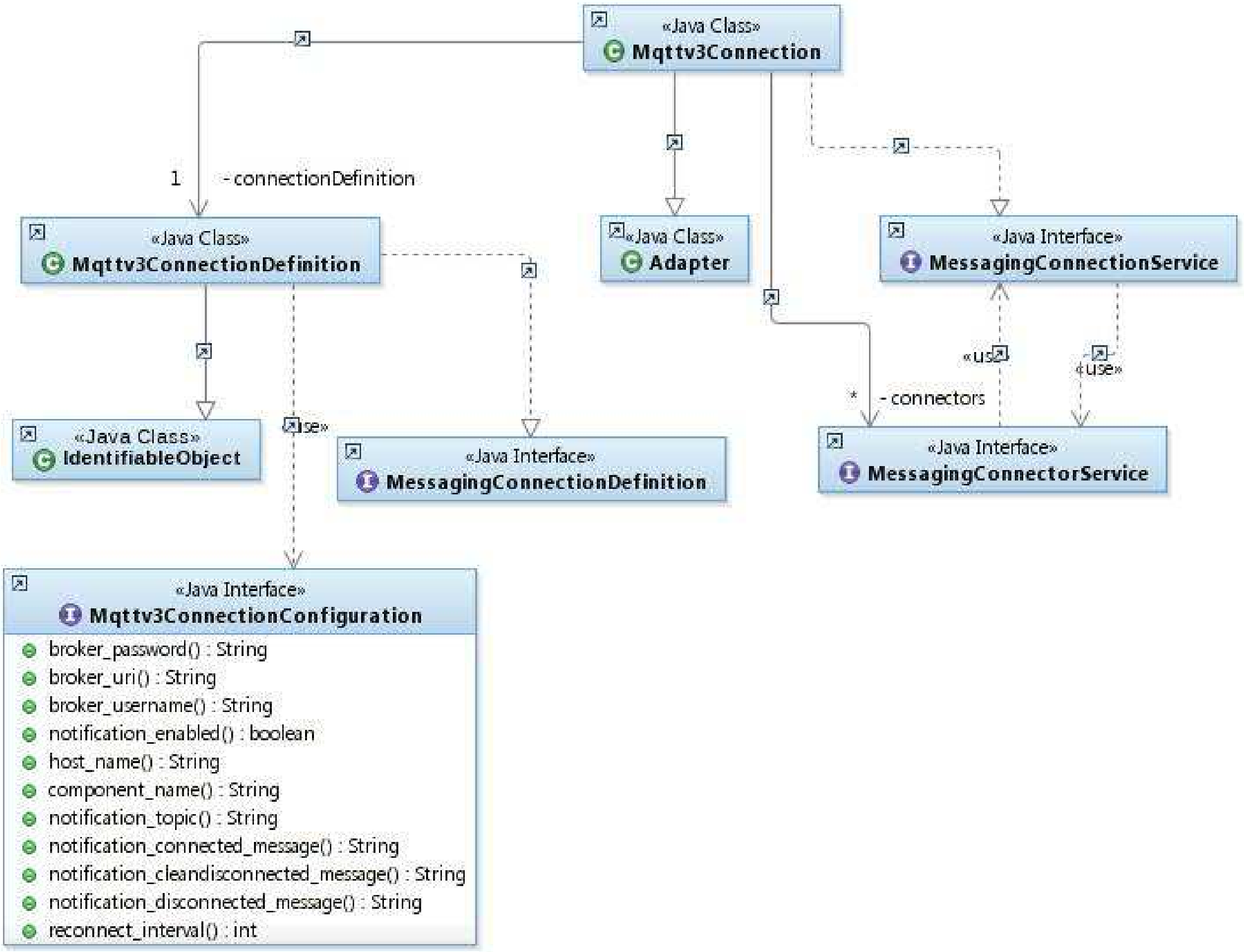

«Java Class»
Mqttv3Connection
1
- connectionDefinition
«Java Class»
Mqttv3ConnectionDefinition
«Java Class»
Adapter
«Java Interface»
MessagingConnectionService
«use»
«use»
*
- connectors
«Java Class»
IdentifiableObject
«use»
«Java Interface»
MessagingConnectionDefinition
«Java Interface»
MessagingConnectorService
«Java Interface»
Mqttv3ConnectionConfiguration
broker_password() : String
broker_uri() : String
broker_username() : String
notification_enabled() : boolean
host_name() : String
component_name() : String
notification_topic() : String
notification_connected_message() : String
notification_cleandisconnected_message() : String
notification_disconnected_message() : String
reconnect_interval() : int

Figure 34 - Mqttv3Connection

- *The Eclipse Paho MQTT client connects to, but does not include, a broker. Can connect to Mosquito, MicroBroker, WebSphere MQ, ...*
- *MicroBroker is a lightweight Java MQTTv5 broker and bridge shipped in IBM Lotus Expeditor Client 6.x.*

2.7 The PowerMatcher Messaging subsystem

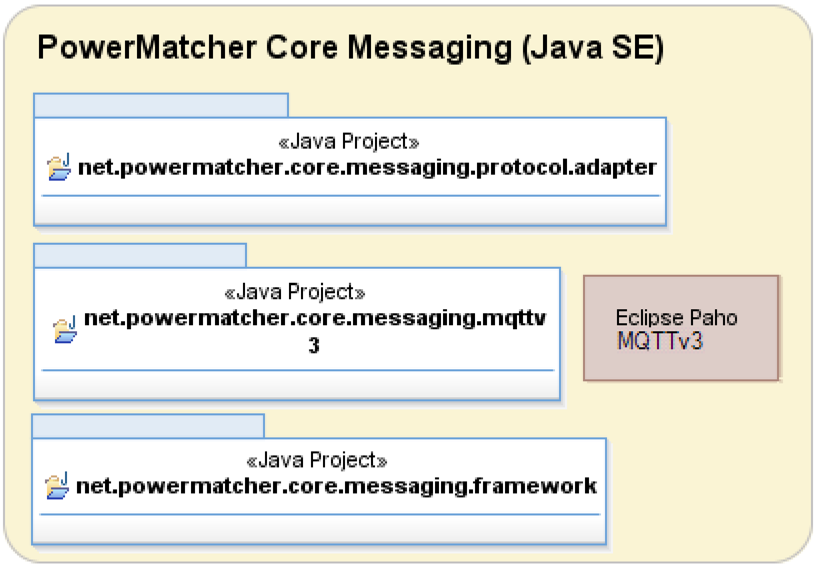

PowerMatcher Core Messaging (Java SE)
«Java Project»
net.powermatcher.core.messaging.protocol.adapter
«Java Project»
net.powermatcher.core.messaging.mqttv3
Eclipse Paho MQTTv3
«Java Project»
net.powermatcher.core.messaging.framework

Figure 35 - PowerMatcher Messaging subsystem

The PowerMatcher messaging protocol adapters proxy the Agent and Matcher roles for PowerMatcher events as well as for logging of PowerMatcher events separately. Agent and Matcher roles are implemented in separate adapters that operate on a messaging connection, as well as in adapters that support direct connections.

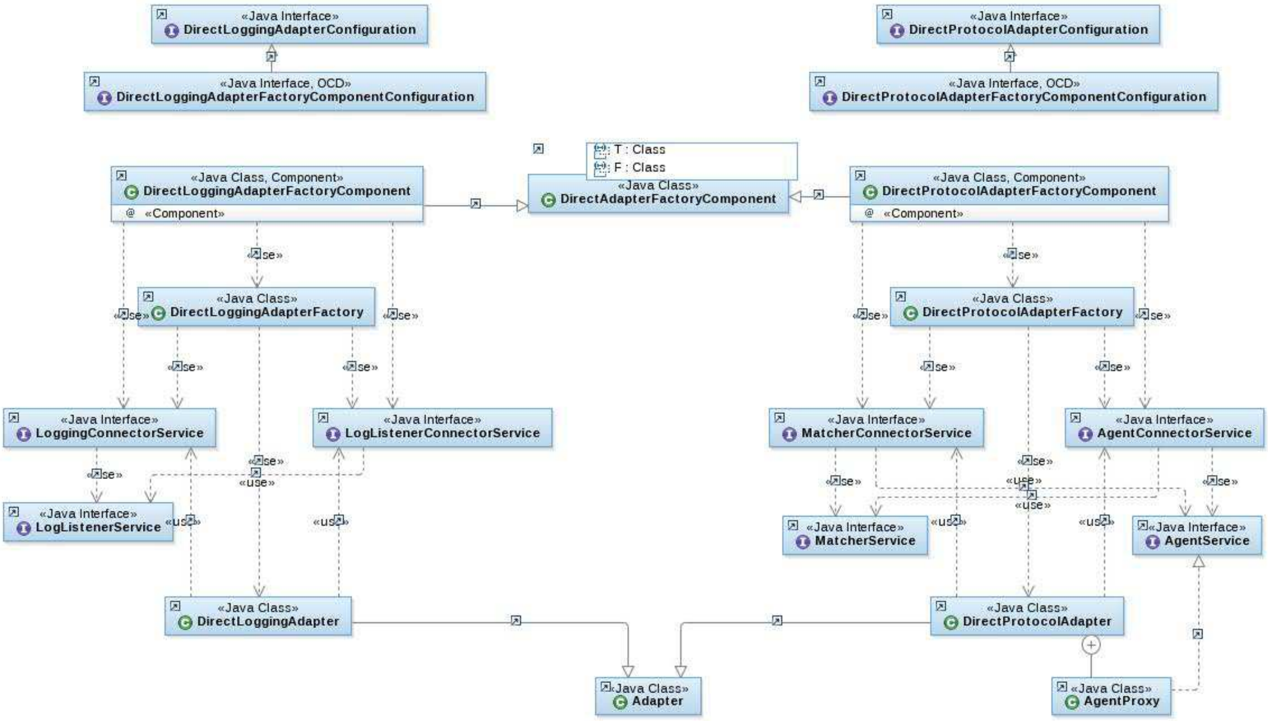

«Java Interface» DirectLoggingAdapterConfiguration
«Java Interface, OCD» DirectLoggingAdapterFactoryComponentConfiguration
«Java Interface» DirectProtocolAdapterConfiguration
«Java Interface, OCD» DirectProtocolAdapterFactoryComponentConfiguration
T : Class
F : Class
«Java Class» DirectAdapterFactoryComponent
«Java Class, Component» DirectLoggingAdapterFactoryComponent
«Component»
«Java Class, Component» DirectProtocolAdapterFactoryComponent
«Component»
«Java Class» DirectLoggingAdapterFactory
«Java Class» DirectProtocolAdapterFactory
«use»
«Java Interface» LoggingConnectorService
«Java Interface» LogListenerConnectorService
«Java Interface» MatcherConnectorService
«Java Interface» AgentConnectorService
«Java Interface» LogListenerService
«Java Interface» MatcherService
«Java Interface» AgentService
«Java Class» DirectLoggingAdapter
«Java Class» DirectProtocolAdapter
«Java Class» Adapter
«Java Class» AgentProxy

Figure 36 - PowerMatcher direct protocol adapters for PowerMatcher events and logging

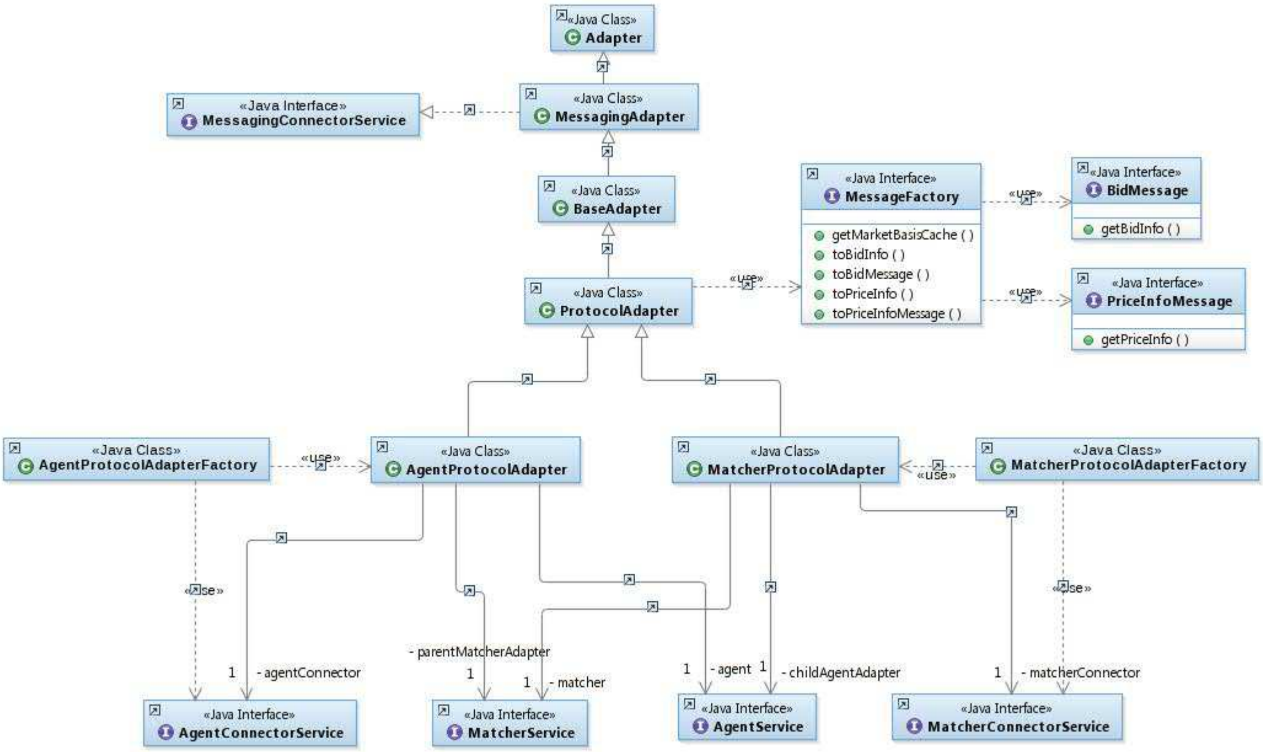

«Java Class» Adapter
«Java Interface» MessagingConnectorService
«Java Class» MessagingAdapter
«Java Class» BaseAdapter
«Java Interface» MessageFactory
getMarketBasisCache ( )
toBidInfo ( )
toBidMessage ( )
toPriceInfo ( )
toPriceInfoMessage ( )
«Java Interface» BidMessage
getBidInfo ( )
«Java Interface» PriceInfoMessage
getPriceInfo ( )
«Java Class» ProtocolAdapter
«use»
«Java Class» AgentProtocolAdapterFactory
«Java Class» AgentProtocolAdapter
«Java Class» MatcherProtocolAdapter
«Java Class» MatcherProtocolAdapterFactory
1 - agentConnector
- parentMatcherAdapter 1
1 - matcher
1 - agent
1 - childAgentAdapter
1 - matcherConnector
«Java Interface» AgentConnectorService
«Java Interface» MatcherService
«Java Interface» AgentService
«Java Interface» MatcherConnectorService

Figure 37 - PowerMatcher messaging protocol adapters for PowerMatcher events

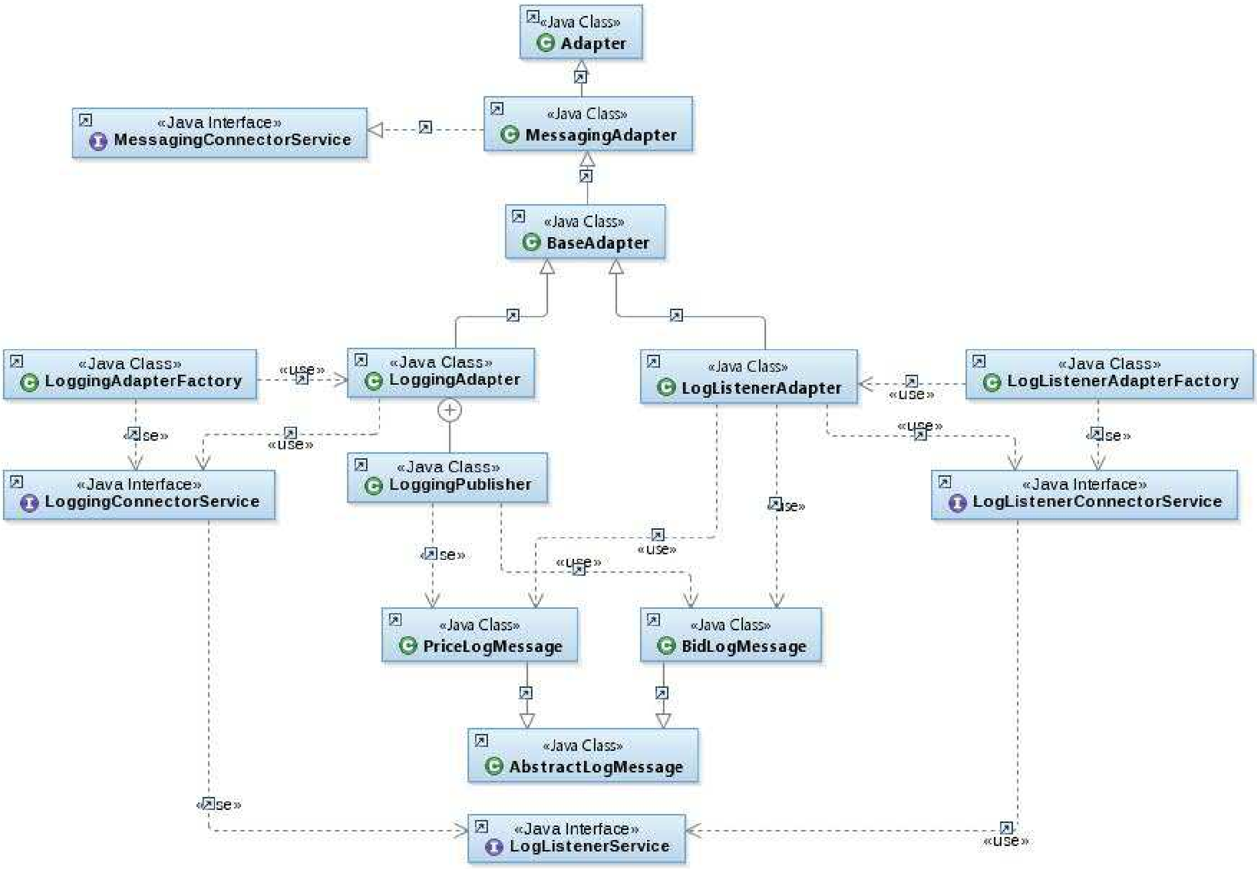

«Java Class» Adapter
«Java Interface» MessagingConnectorService
«Java Class» MessagingAdapter
«Java Class» BaseAdapter
«Java Class» LoggingAdapterFactory
«Java Class» LoggingAdapter
«Java Class» LogListenerAdapter
«Java Class» LogListenerAdapterFactory
«use»
«Java Interface» LoggingConnectorService
«Java Class» LoggingPublisher
«Java Interface» LogListenerConnectorService
«Java Class» PriceLogMessage
«Java Class» BidLogMessage
«Java Class» AbstractLogMessage
«Java Interface» LogListenerService

Figure 38 - PowerMatcher messaging protocol adapters for PowerMatcher logging

2.7.1 The NXP PowerMatcher for the HAN rev. 6 protocol adapter

The PowerMatcher messaging protocol implemented in this release is based on the message structured proposed by NXP in protocol specification rev. 6 [1].

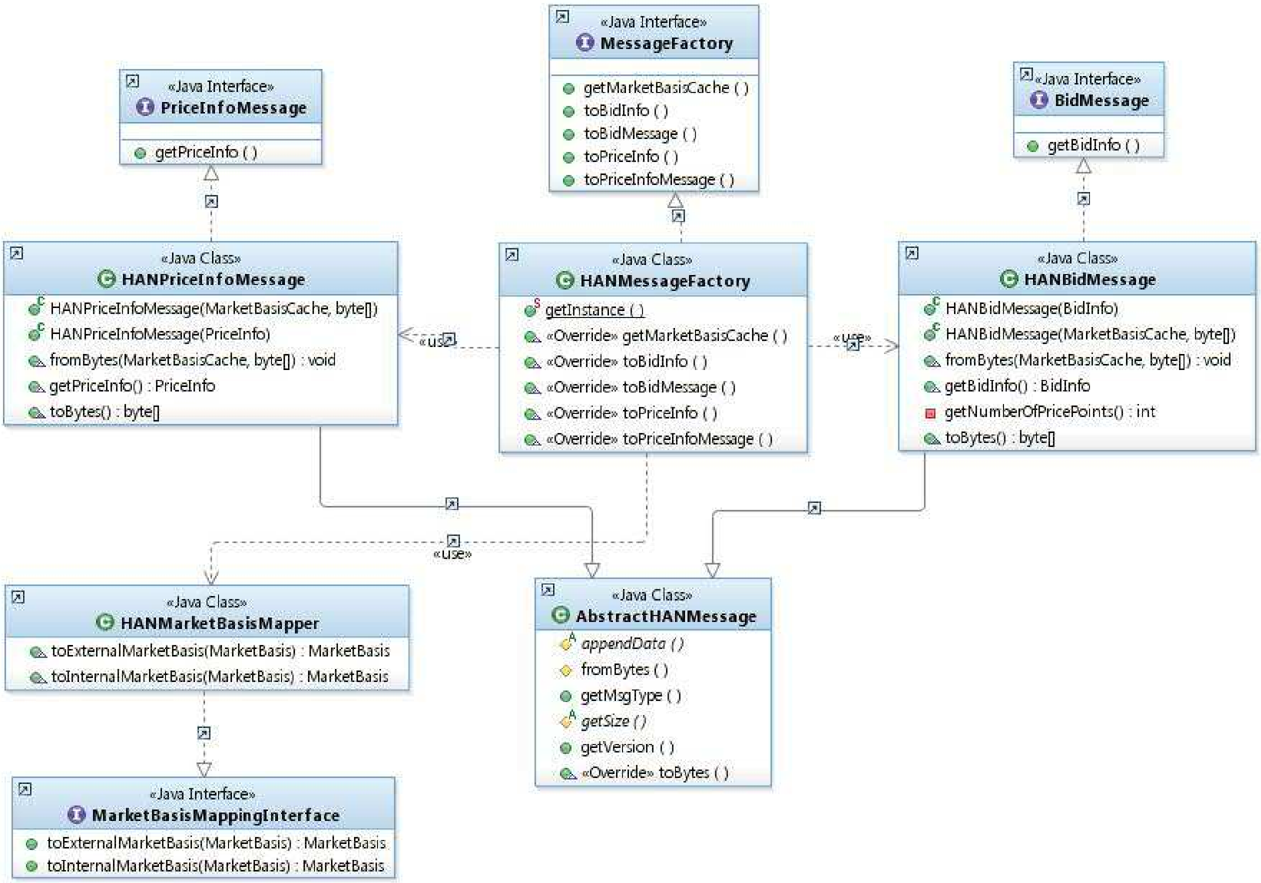

«Java Interface»
PriceInfoMessage
getPriceInfo ( )
«Java Interface»
MessageFactory
getMarketBasisCache ( )
toBidInfo ( )
toBidMessage ( )
toPriceInfo ( )
toPriceInfoMessage ( )
«Java Interface»
BidMessage
getBidInfo ( )
«Java Class»
HANPriceInfoMessage
HANPriceInfoMessage(MarketBasisCache, byte[])
HANPriceInfoMessage(PriceInfo)
fromBytes(MarketBasisCache, byte[]) : void
getPriceInfo() : PriceInfo
toBytes() : byte[]
«use»
«Java Class»
HANMessageFactory
getInstance ( )
«Override» getMarketBasisCache ( )
«Override» toBidInfo ( )
«Override» toBidMessage ( )
«Override» toPriceInfo ( )
«Override» toPriceInfoMessage ( )
«use»
«Java Class»
HANBidMessage
HANBidMessage(BidInfo)
HANBidMessage(MarketBasisCache, byte[])
fromBytes(MarketBasisCache, byte[]) : void
getBidInfo() : BidInfo
getNumberOfPricePoints() : int
toBytes() : byte[]
«use»
«Java Class»
HANMarketBasisMapper
toExternalMarketBasis(MarketBasis) : MarketBasis
toInternalMarketBasis(MarketBasis) : MarketBasis
«Java Class»
AbstractHANMessage
appendData ( )
fromBytes ( )
getMsgType ( )
getSize ( )
getVersion ( )
«Override» toBytes ( )
«Java Interface»
MarketBasisMappingInterface
toExternalMarketBasis(MarketBasis) : MarketBasis
toInternalMarketBasis(MarketBasis) : MarketBasis

Figure 39 - NXP PowerMatcher for the HAN rev. 6 protocol adapter

The protocol defines two messages.

- **Update Bid message:**
  The Update Bid message is a message from the agent to the matcher containing a HAN bid, expressing the need for the electricity represented in a normalized bid price and the power demand quantity. The maximum price of a normalized bid is 127. The minimum is -127.
- **Price Update message:**
  The new market price is communicated via the Price Update message that is sent by the matcher to the connected agents. This message contains the new market price (in normalized price units). It contains also information to convert it to a ‘real world currency’, if needed.

2.7.2 The Internal PowerMatcher protocol adapter

The second PowerMatcher messaging protocol implemented in this release is an internal implementation that is a direct, but efficient, mapping of the market basis, price and bid info classes.

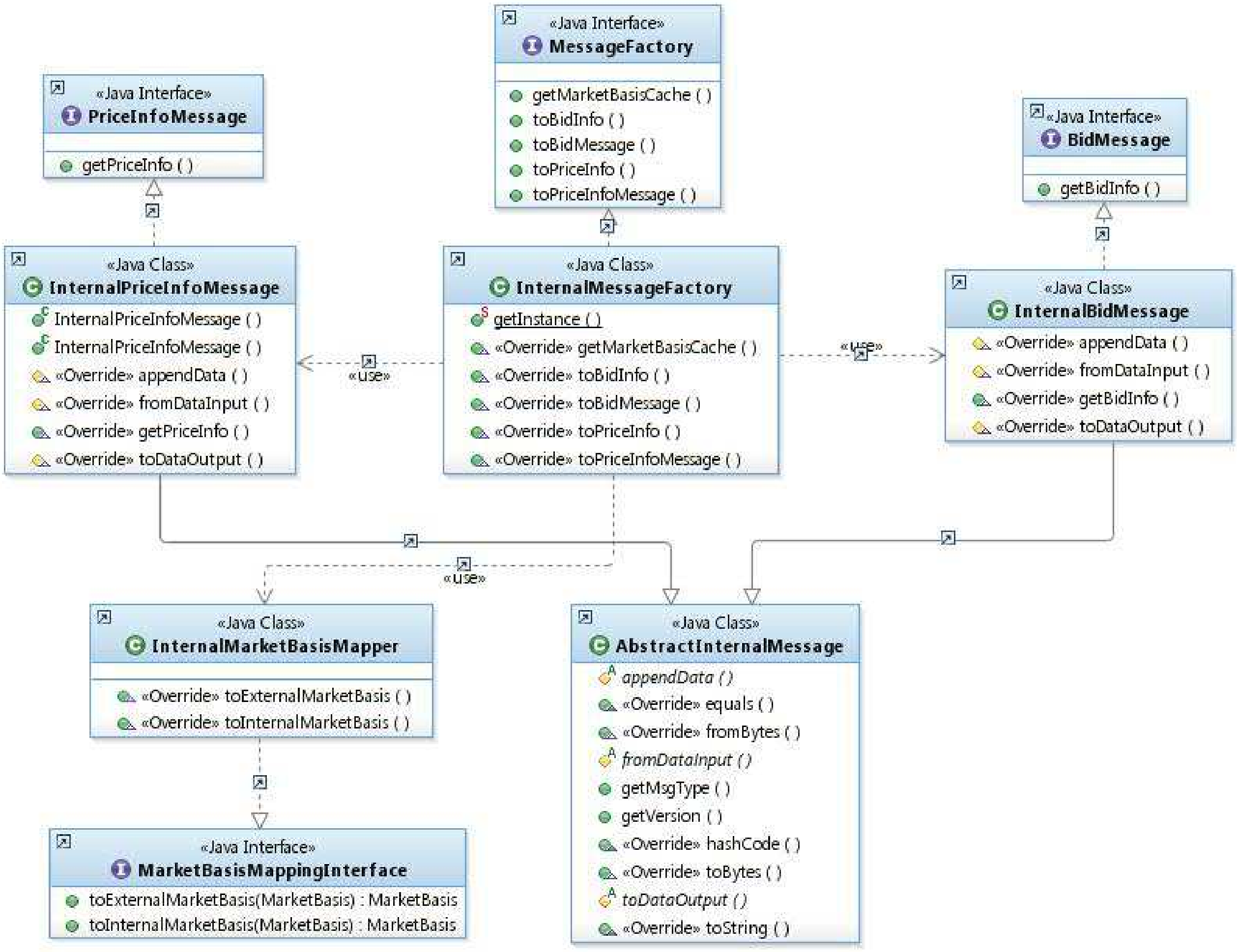


Figure 40 – The Internal PowerMatcher protocol adapter

2.7.3 PowerMatcher logging protocol

It is configurable per agent/matcher what events are logged:

- The Matcher adapter logs incoming agent bids and the published price (default).
- The Matcher adapter logs aggregated bid (optional)
- The Agent adapter logs published bid, received price and market basis updates (optional)

2.7.3.1 Topic namespace for the PowerMatcher logging protocol

The publish/subscribe topic namespace for the PowerMatcher logging has the sane root as for the PowerMatcher events:

```
PowerMatcher/{clusterId}
```

where `{clusterId}` is the simple identifier for the cluster.
The topics for the Price Info and Bid log messages are:

```
PowerMatcher/{clusterId}/{adapterId}/UpdatePriceInfo/Log
PowerMatcher/{clusterId}/{adapterId}/UpdateBid/Log
```

Where:

- `{adapterId}` is the identifier of the agent or matcher adapter that is publishing the log messages.
- The topic suffixes are *UpdatePriceInfo*, *UpdateBid*, and *Log* by default, but are configurable at protocol level.

A logging agent subscribes to receive PowerMatcher log messages using the following wildcard pattern:

```
PowerMatcher/{clusterId}/+/+/Log
```

2.8 The Telemetry Messaging subsystem

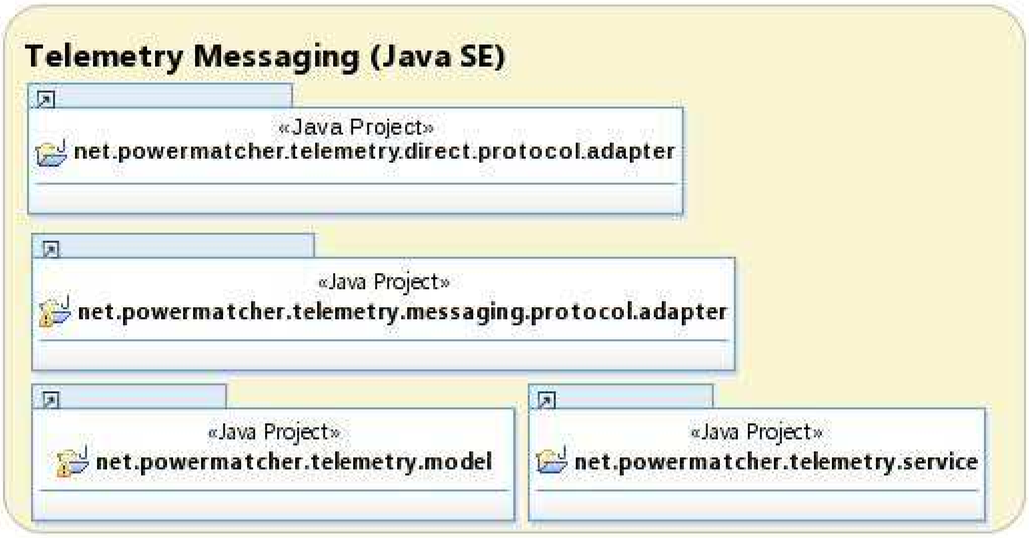

Telemetry Messaging (Java SE)
«Java Project»
net.powermatcher.telemetry.direct.protocol.adapter
«Java Project»
net.powermatcher.telemetry.messaging.protocol.adapter
«Java Project»
net.powermatcher.telemetry.model
«Java Project»
net.powermatcher.telemetry.service

Figure 41 - Telemetry Messaging subsystem

2.8.1 TelemetryData API

Telemetry captures monitoring data in a generic format. In this release stream encoding from Java API is in XML, as specified in the next section.

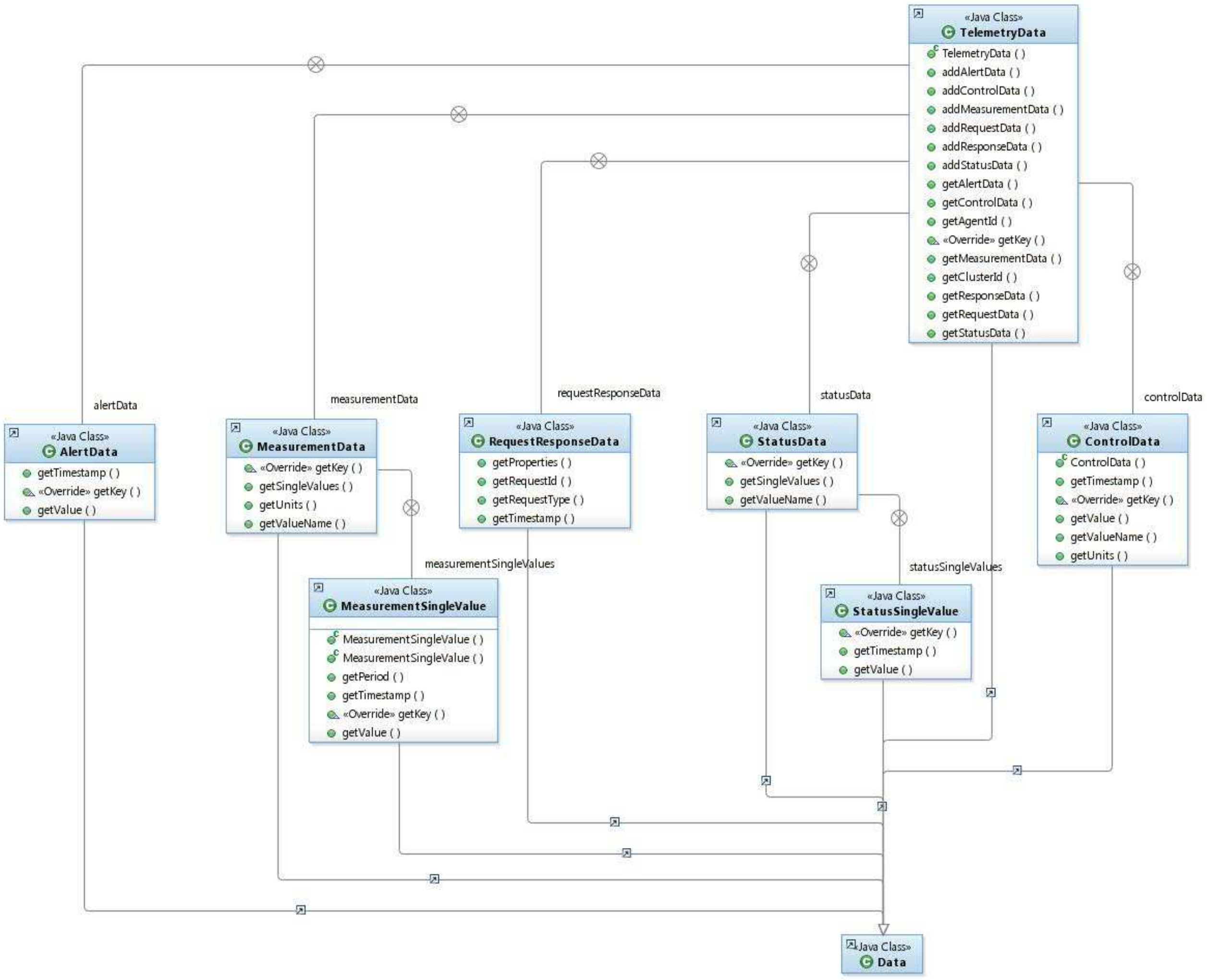


Figure 42 - TelemetryData class hierarchy

2.8.2 Telemetry and Control XML protocol

Measurement, status and other data collected during processing can be reported through the Telemetry interface. This interface specifies an XML message that can contain a variable number of data elements with a name, unit and one or more values.

XML encoding/decoding for the TelemetryData API is provided in separate jar.

The message format is defined by the `TelemetrySchema.xsd` and can be found in 2.8.2.1. The schema defines elements like ‘measurements’, ‘status’, ‘alert’, ‘control’, ‘request’ and ‘response’ which makes it suitable for all types of messaging.

The following table lists the most important element definitions for telemetry messages:

Element	Type	Description
clusterId	string	The cluster id of the agent associated with the telemetry event.
agentId	string	The id of the agent associated with the telemetry event.
Measurement	complex	Used for measurement data. Specifies the name (e.g. ‘temperature’) and unit (e.g. ‘C’ for ‘Celsius’). Contains one or more measurement values with an optional measurement period in seconds.
Status	complex	Used for status data. Specifies the status name and a child element containing the status value and timestamp.

There are no pre-defined values or formats for the string type fields and they are not enforced by the xml schema definition. However, the formats used and values should be known to the receiver (PowerMatcher logging agent) in order to be processed and stored in a database.

2.8.2.1 Topic namespace for the Telemetry messaging protocol

- Publish topics with topic levels, with a separate namespace per cluster:
  - Telemetry/*clusterId*/*agentId*

- Subscribe pattern with topic level wildcard #:
  - Telemetry/*clusterId*/#

2.8.2.2 Telemetry message example

The following message illustrates an example of a telemetry message containing 3 measurements with 4 measurement values.

```
<?xml version="1.0" encoding="UTF-8"?>

<telemetry clusterId="WestOrange" agentId="Serial Number Electricity Meter"
              xmlns="http://www.powermatcher.net/TelemetrySchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.powermatcher.net/TelemetrySchema TelemetrySchema.xsd
">
       <measurement valueName="currentUsage" units="W">
              <singleValue value="1.0" timestamp="2001-12-31T12:00:10.000+01:00"/>
       </measurement>
       <measurement valueName="meterReading" units="Wh">
              <singleValue value="2000.0" timestamp="2001-12-31T12:00:00.000+01:00"/>
              <singleValue value="2010.0" timestamp="2001-12-31T12:00:10.000+01:00"/>
       </measurement>
       <measurement units="Wh" valueName="periodicUsage">
              <singleValue value="10.0" timestamp="2001-12-31T12:00:00.000+01:00" period="60"/>
       </measurement>
</cpss>
```

This example and other examples are included in the code and serve as input for unit testing.

2.8.2.3 XSD schema for telemetry and control messages

The XML schema “TelemetrySchema” defines a message format that can be used for telemetry, control, alert, request and response messages. It contains a collection of report element definitions types suitable for most plausible data types.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- Copyright (c) 2012 Alliander.            -->
<!-- All rights reserved.                     -->
<!--                                          -->
<!-- Contributors:                            -->
<!--     IBM - initial API and implementation -->
<schema targetNamespace="http://www.powermatcher.net/TelemetrySchema" elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.powermatcher.net/TelemetrySchema">
    <element name="telemetry" type="tns:TelemetryType"></element>

    <complexType name="TelemetryType">
          <choice>
                    <element name="measurement" type="tns:MeasurementType" maxOccurs="unbounded" minOccurs="1"></element>
                    <element name="status" type="tns:StatusType" maxOccurs="unbounded" minOccurs="1"></element>
                    <element name="alert" type="tns:AlertType" maxOccurs="unbounded" minOccurs="1"></element>
                    <element name="control" type="tns:ControlType" maxOccurs="unbounded" minOccurs="1"></element>
                    <element name="request" type="tns:RequestType" maxOccurs="unbounded" minOccurs="1"></element>
                    <element name="response" type="tns:ResponseType" maxOccurs="unbounded" minOccurs="1"></element>
          </choice>
          <attribute name="clusterId" type="string" use="required"></attribute>
          <attribute name="agentId" type="string" use="required"></attribute>
    </complexType>
    <complexType name="MeasurementType">
        <choice minOccurs="0" maxOccurs="unbounded">
                    <element name="singleValue" type="tns:SingleDataValueType">
                    </element>
          </choice>
          <attribute name="valueName" type="string" use="required"></attribute>
          <attribute name="units" type="string" use="required"></attribute>
    </complexType>
    <complexType name="StatusType">
        <choice minOccurs="0" maxOccurs="unbounded">
                    <element name="singleValue" type="tns:SingleStatusValueType"></element>
          </choice>
          <attribute name="valueName" type="string" use="required"></attribute>
    </complexType>
    <complexType name="AlertType">
          <attribute name="value" type="string" use="required"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
    </complexType>
    <complexType name="ControlType">
          <attribute name="valueName" type="string" use="required"></attribute>
          <attribute name="value" type="string" use="required"></attribute>
          <attribute name="units" type="string" use="required"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
    </complexType>
    <complexType name="RequestType">
        <choice minOccurs="0" maxOccurs="unbounded">
                    <element name="property" type="tns:PropertyType"></element>
        </choice>
          <attribute name="requestType" type="string" use="required"></attribute>
          <attribute name="requestId" type="string" use="optional"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
    </complexType>
    <complexType name="ResponseType">
        <choice minOccurs="0" maxOccurs="unbounded">
                    <element name="property" type="tns:PropertyType"></element>
        </choice>
          <attribute name="requestType" type="string" use="required"></attribute>
          <attribute name="requestId" type="string" use="required"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
    </complexType>

    <complexType name="SingleDataValueType">
          <attribute name="value" type="float" use="required"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
          <attribute name="period" type="int" use="optional"></attribute>
    </complexType>
    <complexType name="SingleStatusValueType">
          <attribute name="value" type="string" use="required"></attribute>
          <attribute name="timestamp" type="dateTime" use="required"></attribute>
    </complexType>
    <complexType name="PropertyType">
                    <attribute name="name" type="string" use="required"/>
                    <attribute name="value" type="string" use="required"/>
                    <attribute name="logging" type="boolean" default="true" use="optional"/>
    </complexType>
</schema>
```

2.8.3 TelemetryService and TelemetryDataPublisher

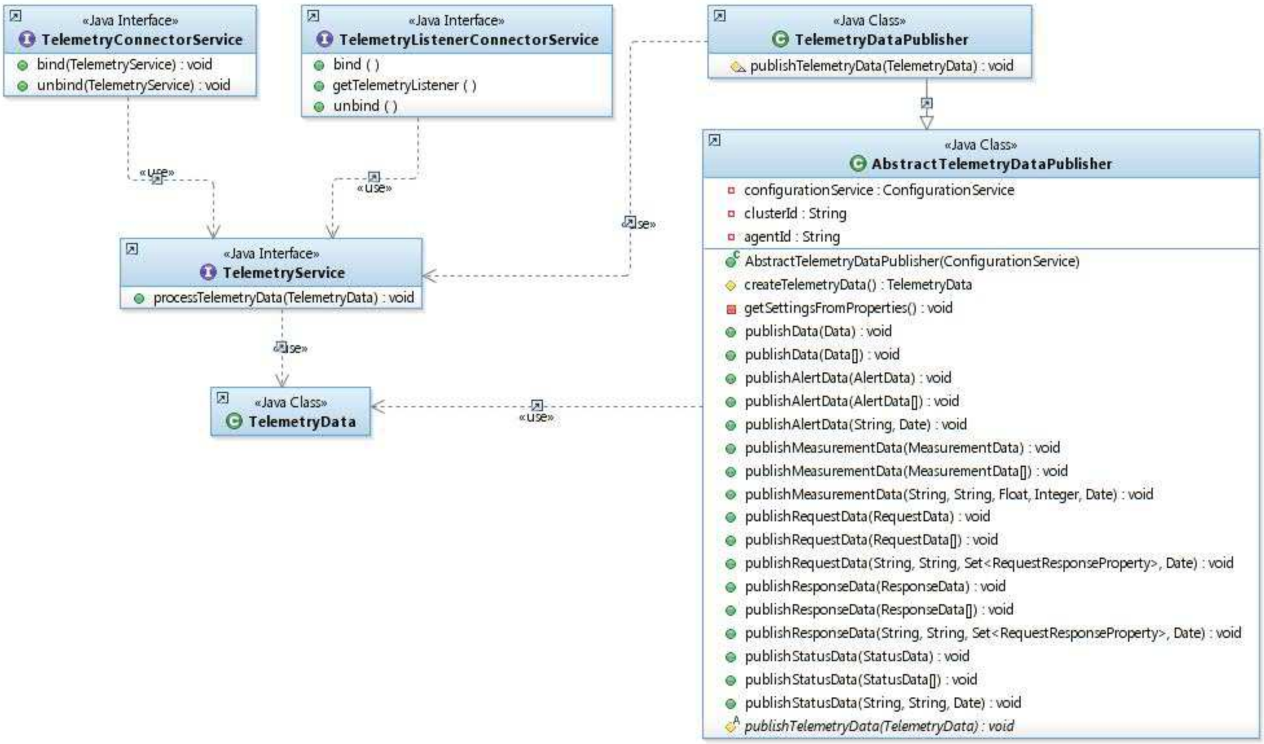

«Java Interface»
TelemetryConnectorService
bind(TelemetryService) : void
unbind(TelemetryService) : void
«Java Interface»
TelemetryListenerConnectorService
bind ( )
getTelemetryListener ( )
unbind ( )
«Java Class»
TelemetryDataPublisher
publishTelemetryData(TelemetryData) : void
«use»
«use»
«use»
«Java Interface»
TelemetryService
processTelemetryData(TelemetryData) : void
«use»
«Java Class»
TelemetryData
«use»
«Java Class»
AbstractTelemetryDataPublisher
configurationService : ConfigurationService
clusterId : String
agentId : String
AbstractTelemetryDataPublisher(ConfigurationService)
createTelemetryData() : TelemetryData
getSettingsFromProperties() : void
publishData(Data) : void
publishData(Data[]) : void
publishAlertData(AlertData) : void
publishAlertData(AlertData[]) : void
publishAlertData(String, Date) : void
publishMeasurementData(MeasurementData) : void
publishMeasurementData(MeasurementData[]) : void
publishMeasurementData(String, String, Float, Integer, Date) : void
publishRequestData(RequestData) : void
publishRequestData(RequestData[]) : void
publishRequestData(String, String, Set<RequestResponseProperty>, Date) : void
publishResponseData(ResponseData) : void
publishResponseData(ResponseData[]) : void
publishResponseData(String, String, Set<RequestResponseProperty>, Date) : void
publishStatusData(StatusData) : void
publishStatusData(StatusData[]) : void
publishStatusData(String, String, Date) : void
publishTelemetryData(TelemetryData) : void

Figure 43 –TelemetryService and TelemetryDataPublisher

2.8.4 TelemetryAdapter and TelemetryConnector

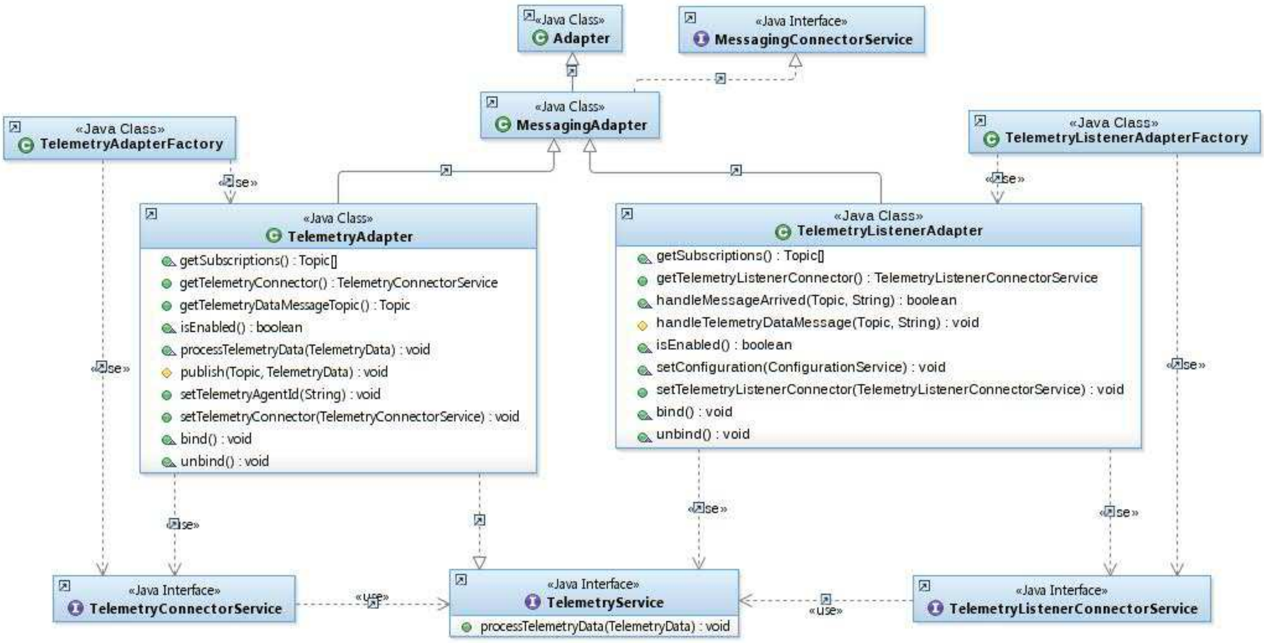

«Java Class»
Adapter
«Java Interface»
MessagingConnectorService
«Java Class»
MessagingAdapter
«Java Class»
TelemetryAdapterFactory
«Java Class»
TelemetryListenerAdapterFactory
«use»
«use»
«Java Class»
TelemetryAdapter
getSubscriptions() : Topic[]
getTelemetryConnector() : TelemetryConnectorService
getTelemetryDataMessageTopic() : Topic
isEnabled() : boolean
processTelemetryData(TelemetryData) : void
publish(Topic, TelemetryData) : void
setTelemetryAgentId(String) : void
setTelemetryConnector(TelemetryConnectorService) : void
bind() : void
unbind() : void
«Java Class»
TelemetryListenerAdapter
getSubscriptions() : Topic[]
getTelemetryListenerConnector() : TelemetryListenerConnectorService
handleMessageArrived(Topic, String) : boolean
handleTelemetryDataMessage(Topic, String) : void
isEnabled() : boolean
setConfiguration(ConfigurationService) : void
setTelemetryListenerConnector(TelemetryListenerConnectorService) : void
bind() : void
unbind() : void
«use»
«use»
«use»
«use»
«use»
«Java Interface»
TelemetryConnectorService
«use»
«Java Interface»
TelemetryService
processTelemetryData(TelemetryData) : void
«use»
«Java Interface»
TelemetryListenerConnectorService

Figure 44 – Messaging TelemetryAdapter and TelemetryConnector

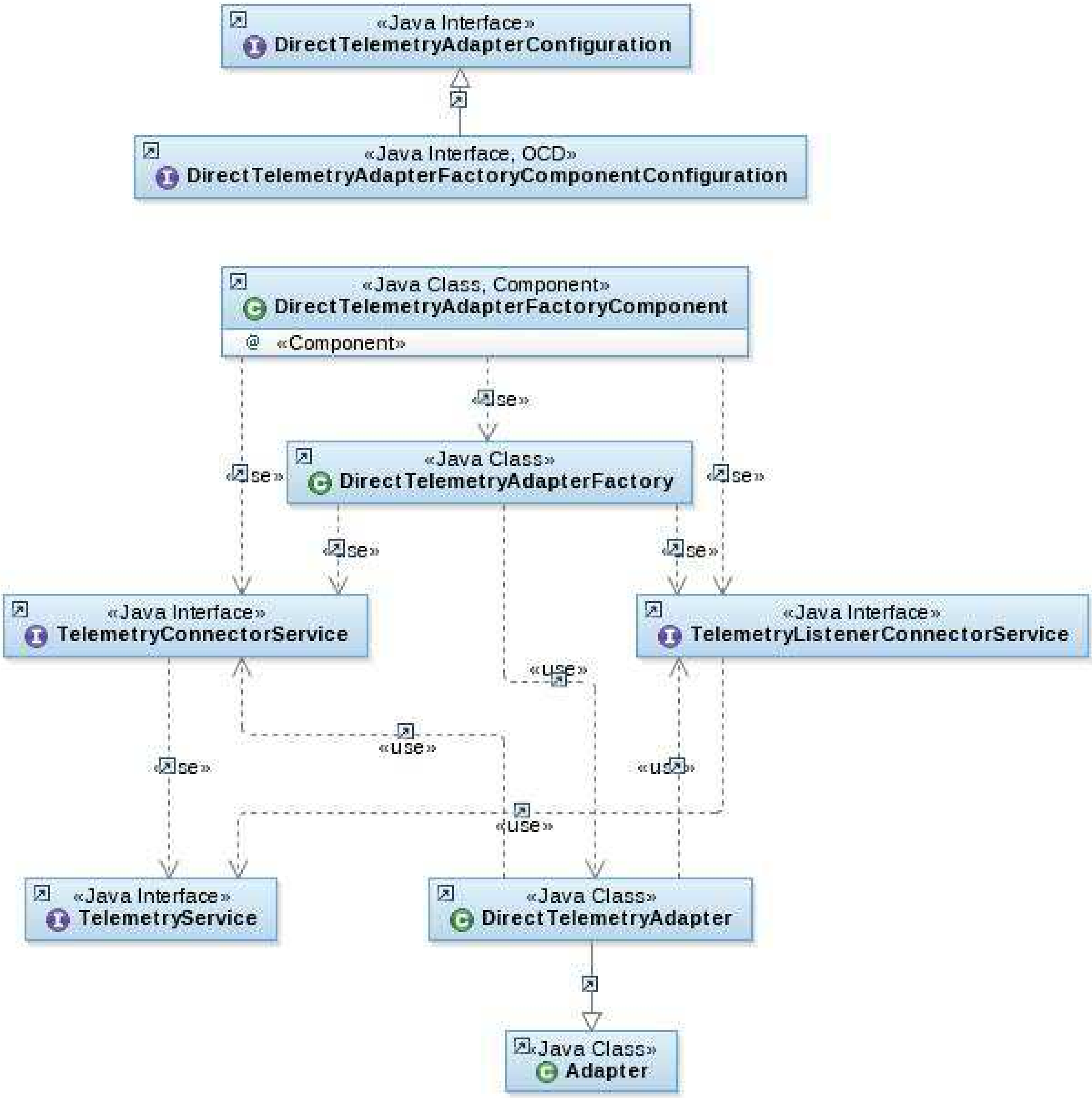

«Java Interface»
DirectTelemetryAdapterConfiguration
«Java Interface, OCD»
DirectTelemetryAdapterFactoryComponentConfiguration
«Java Class, Component»
DirectTelemetryAdapterFactoryComponent
«Component»
«use»
«Java Class»
DirectTelemetryAdapterFactory
«Java Interface»
TelemetryConnectorService
«Java Interface»
TelemetryListenerConnectorService
«Java Interface»
TelemetryService
«Java Class»
DirectTelemetryAdapter
«Java Class»
Adapter

Figure 45 – Direct TelemetryAdapter

2.9 *Utility agents in PowerMatcher Core and Extension*

2.9.1 The Framework TestAgent

The TestAgent can be configured to publish a series of single step bids with linear change in price and power.

2.9.2 The PowerMatcher Logging Agent

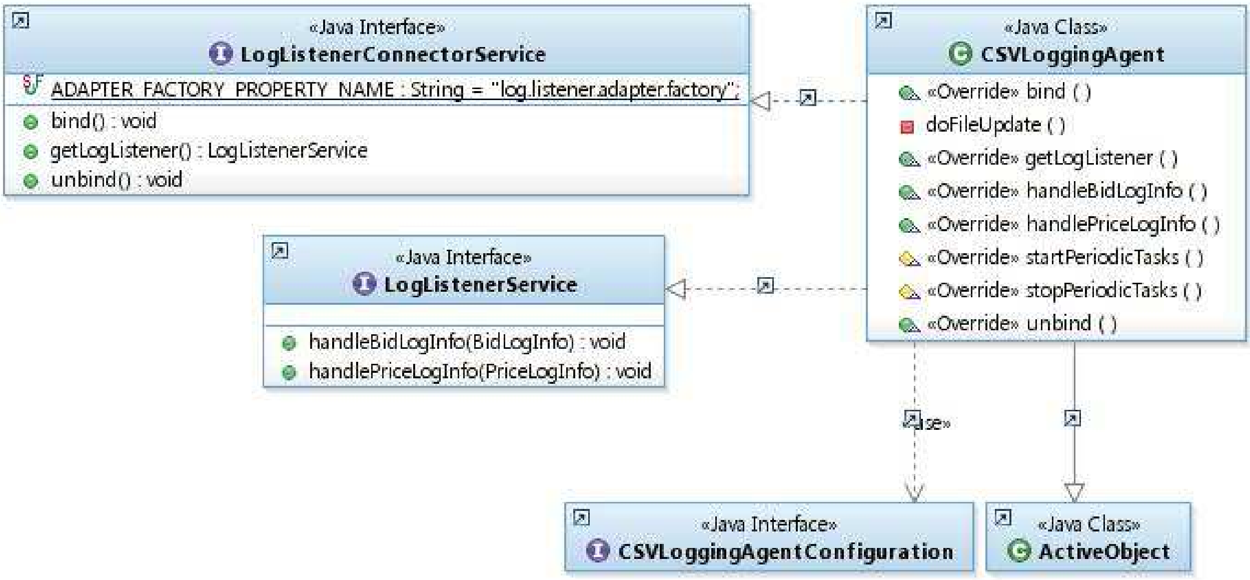

«Java Interface»
LogListenerConnectorService
ADAPTER_FACTORY_PROPERTY_NAME : String = "log.listener.adapter.factory";
bind() : void
getLogListener() : LogListenerService
unbind() : void
«Java Class»
CSVLoggingAgent
«Override» bind ( )
doFileUpdate ( )
«Override» getLogListener ( )
«Override» handleBidLogInfo ( )
«Override» handlePriceLogInfo ( )
«Override» startPeriodicTasks ( )
«Override» stopPeriodicTasks ( )
«Override» unbind ( )
«Java Interface»
LogListenerService
handleBidLogInfo(BidLogInfo) : void
handlePriceLogInfo(PriceLogInfo) : void
«use»
«Java Interface»
CSVLoggingAgentConfiguration
«Java Class»
ActiveObject

Figure 46 – PowerMatcher Logging Agent

The CSVLoggingAgent logs PowerMatcher bid and price log events with related market price in csv files at a fixed interval.

Bids:

- Timestamp
- Cluster ID
- Agent ID
- Bid number
- Bid curve
- Market basis ID

Price updates:

- Timestamp
- Cluster ID
- Agent ID
- Price update number
- Market price
- Market basis ID

Market basis updates:

- Timestamp
- Cluster ID
- Market basis ID
- Commodity (Electricity)
- Minimum price
- Maximum price
- Price steps

2.9.3 The Telemetry Logging Agent

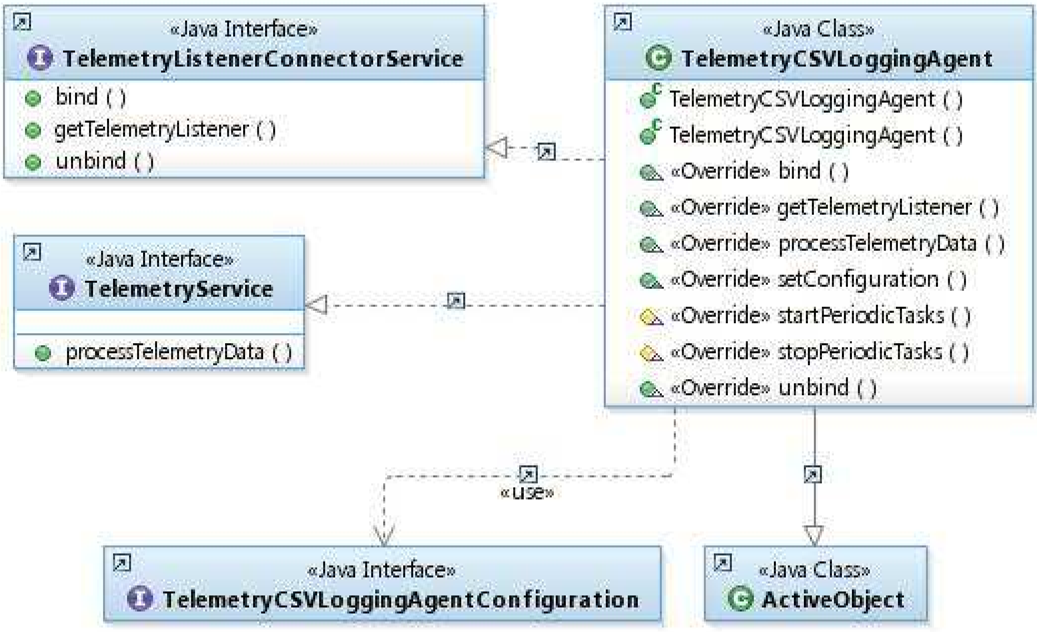

«Java Interface»
TelemetryListenerConnectorService
bind ( )
getTelemetryListener ( )
unbind ( )
«Java Interface»
TelemetryService
processTelemetryData ( )
«Java Class»
TelemetryCSVLoggingAgent
TelemetryCSVLoggingAgent ( )
TelemetryCSVLoggingAgent ( )
«Override» bind ( )
«Override» getTelemetryListener ( )
«Override» processTelemetryData ( )
«Override» setConfiguration ( )
«Override» startPeriodicTasks ( )
«Override» stopPeriodicTasks ( )
«Override» unbind ( )
«use»
«Java Interface»
TelemetryCSVLoggingAgentConfiguration
«Java Class»
ActiveObject

Figure 47 – Telemetry Logging Agent

The TelemetryCSVLoggingAgent logs telemetry measurement and status events in csv files at a fixed interval:

Measurement data:

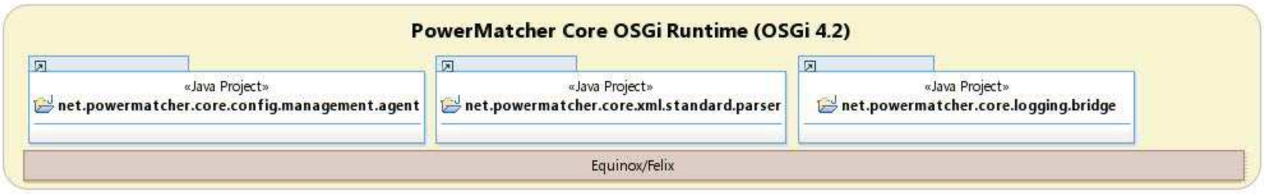
- Timestamp
- Cluster ID
- Agent ID
- Measurement type
- Measurement unit
- Measurement value
- Measurement period (optional)

Status data:

- Timestamp
- Cluster ID
- Agent ID
- Status type
- Status value

2.10 The OSGi Runtime subsystem

Although PowerMatcher applications can be run as plain Java applications, the OSGi platform provides dynamic configuration and software updates.


PowerMatcher Core OSGi Runtime (OSGi 4.2)
«Java Project»
net.powermatcher.core.config.management.agent
«Java Project»
net.powermatcher.core.xml.standard.parser
«Java Project»
net.powermatcher.core.logging.bridge
Equinox/Felix

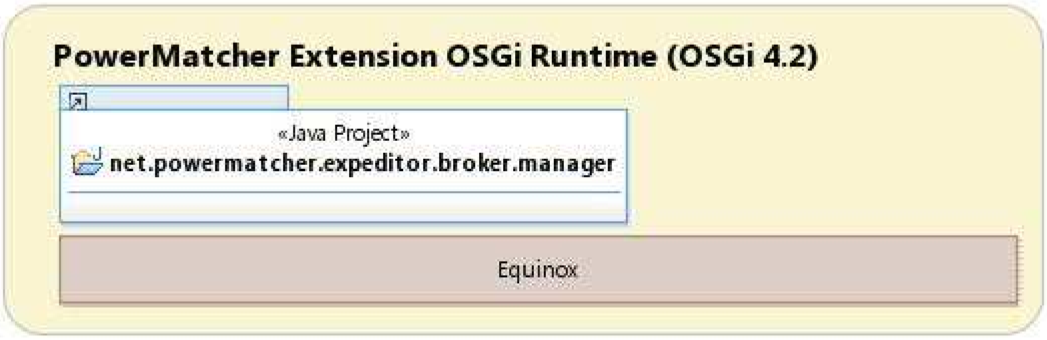

PowerMatcher Extension OSGi Runtime (OSGi 4.2)
«Java Project»
net.powermatcher.expeditor.broker.manager
Equinox

Figure 48 – PowerMatcher Core and Extension OSGi Runtime subsystem

2.10.1 Configuration Management Agent

Reads XML configuration specification for OSGi runtime from URL.
Processes dynamic configuration updates (add, update, delete) in OSGi ConfigAdmin.

2.10.2 Expeditor Broker Manager

Creates and configures local MQTT MicroBroker instance.
Configuration of MicroBroker Bridge for bridging topics between local and remote MQTT broker.

- Requires MicroBroker license (included with IBM Lotus Expeditor Device client)
- Requires Equinox, Felix is not supported by MicroBroker.

2.10.3 OSGi Runtime Deployment

There are 3 possible deployment options, as depicted in the following diagram: standalone with an embedded MicroBroker, standalone with an external broker, or distributed with a central enterprise broker.

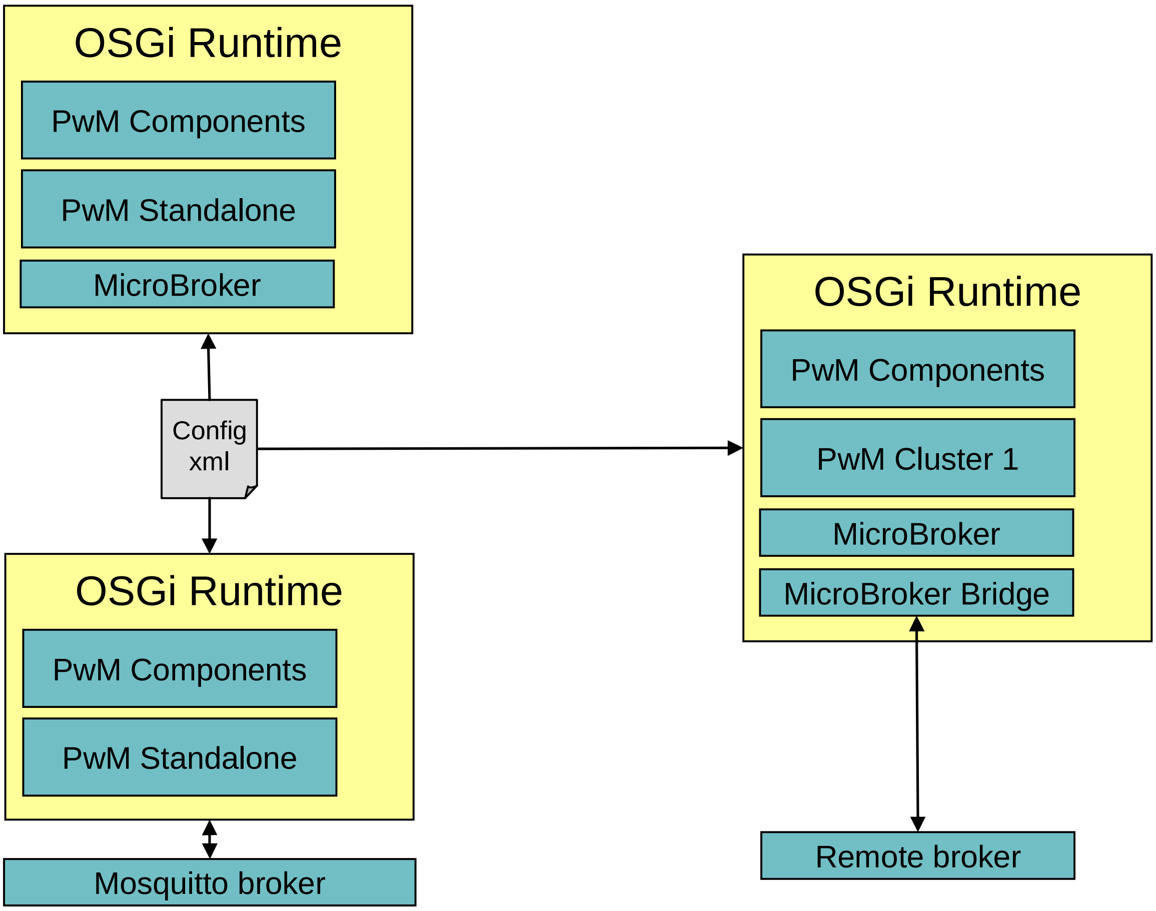


Figure 49 - OSGi Runtime deployment options

2.11 The Service Components subsystem

Components are implemented according to the OSGi Service Component Model (SCM).

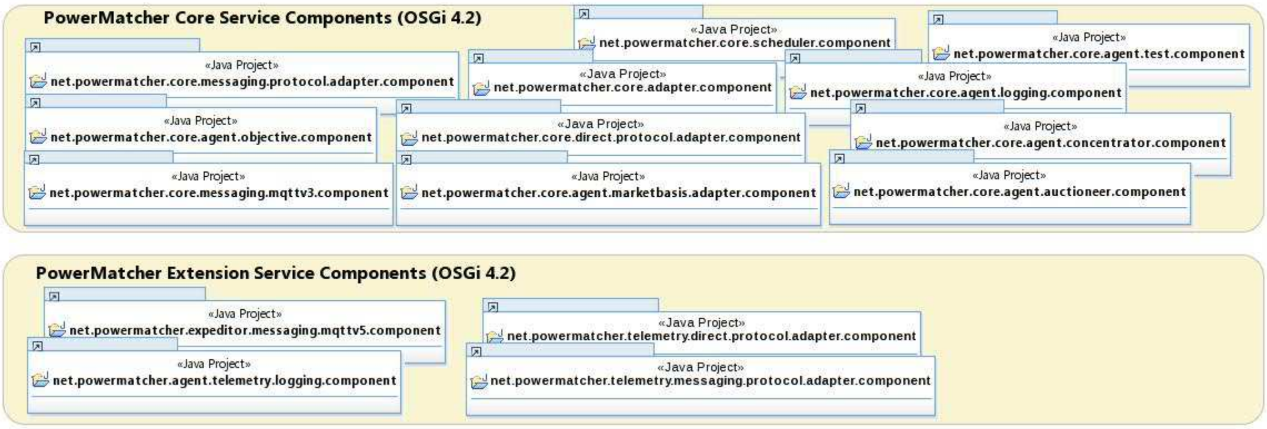

PowerMatcher Core Service Components (OSGi 4.2)
«Java Project» net.powermatcher.core.scheduler.component
«Java Project» net.powermatcher.core.agent.test.component
«Java Project» net.powermatcher.core.messaging.protocol.adapter.component
«Java Project» net.powermatcher.core.adapter.component
«Java Project» net.powermatcher.core.agent.logging.component
«Java Project» net.powermatcher.core.agent.objective.component
«Java Project» net.powermatcher.core.direct.protocol.adapter.component
«Java Project» net.powermatcher.core.agent.concentrator.component
«Java Project» net.powermatcher.core.messaging.mqttv3.component
«Java Project» net.powermatcher.core.agent.marketbasis.adapter.component
«Java Project» net.powermatcher.core.agent.auctioneer.component
PowerMatcher Extension Service Components (OSGi 4.2)
«Java Project» net.powermatcher.expeditor.messaging.mqttv5.component
«Java Project» net.powermatcher.telemetry.direct.protocol.adapter.component
«Java Project» net.powermatcher.agent.telemetry.logging.component
«Java Project» net.powermatcher.telemetry.messaging.protocol.adapter.component

Figure 50 – PowerMatcher Core and Extension Service Components subsystem

- Component and configuration metadata is generated from the commonly used aQute SCM annotations
- Strict separation from Java SE components to isolate and minimize OSGi dependencies.
- Components are instantiated from persistent OSGi configuration managed via the OSGi ConfigurationAdmin service.
  - Singleton components.
  - Factory components.
- Components dynamically register services in the OSGi service registry, like for example connector services.
- Adapter components track connector services (and other services) in the OSGi service registry, and bind to/bind from connectors as they become available/disappear.

2.11.1 Messaging Connection Components

Components to provide a messaging connection, using a tracker for MessagingConnectorService interfaces registered by other components that require a messaging connection.

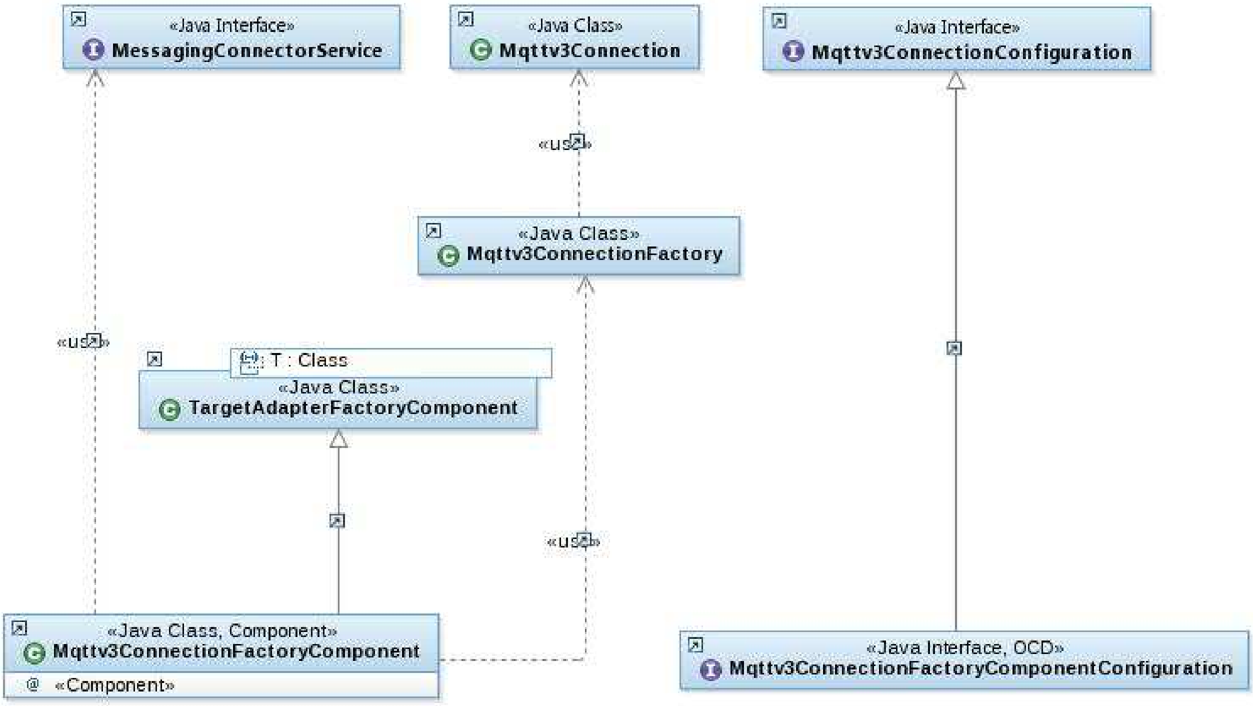

«Java Interface» MessagingConnectorService
«Java Class» Mqttv3Connection
«Java Interface» Mqttv3ConnectionConfiguration
«use»
«Java Class» Mqttv3ConnectionFactory
«use»
T : Class
«Java Class» TargetAdapterFactoryComponent
«use»
«Java Class, Component» Mqttv3ConnectionFactoryComponent
@ «Component»
«Java Interface, OCD» Mqttv3ConnectionFactoryComponentConfiguration

Figure 51 - Mqttv3ConnectionComponent and ConnectorTracker

2.11.2 PowerMatcher Protocol Adapter Components

Component to provide a PowerMatcher protocol adapter, using a tracker for Agent- and MatcherConnectorService interfaces registered by PowerMatcher components that require either an Agent adapter or and Agent and a Matcher adapter.

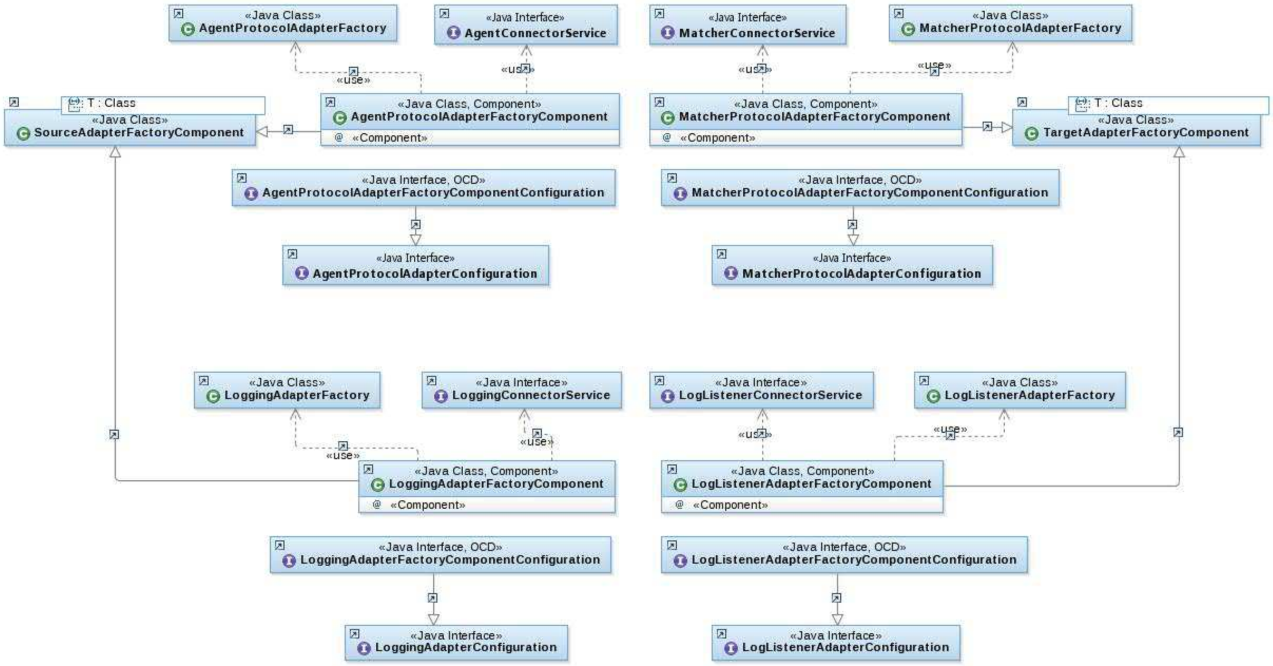

«Java Class» AgentProtocolAdapterFactory
«Java Interface» AgentConnectorService
«Java Interface» MatcherConnectorService
«Java Class» MatcherProtocolAdapterFactory
«use»
T : Class
«Java Class» SourceAdapterFactoryComponent
«Java Class, Component» AgentProtocolAdapterFactoryComponent
«Component»
«Java Class, Component» MatcherProtocolAdapterFactoryComponent
«Component»
T : Class
«Java Class» TargetAdapterFactoryComponent
«Java Interface, OCD» AgentProtocolAdapterFactoryComponentConfiguration
«Java Interface, OCD» MatcherProtocolAdapterFactoryComponentConfiguration
«Java Interface» AgentProtocolAdapterConfiguration
«Java Interface» MatcherProtocolAdapterConfiguration
«Java Class» LoggingAdapterFactory
«Java Interface» LoggingConnectorService
«Java Interface» LogListenerConnectorService
«Java Class» LogListenerAdapterFactory
«use»
«Java Class, Component» LoggingAdapterFactoryComponent
«Component»
«Java Class, Component» LogListenerAdapterFactoryComponent
«Component»
«Java Interface, OCD» LoggingAdapterFactoryComponentConfiguration
«Java Interface, OCD» LogListenerAdapterFactoryComponentConfiguration
«Java Interface» LoggingAdapterConfiguration
«Java Interface» LogListenerAdapterConfiguration

Figure 52 - Agent- and MatcherProtocolAdapterComponent

2.11.3 Telemetry Adapter Component

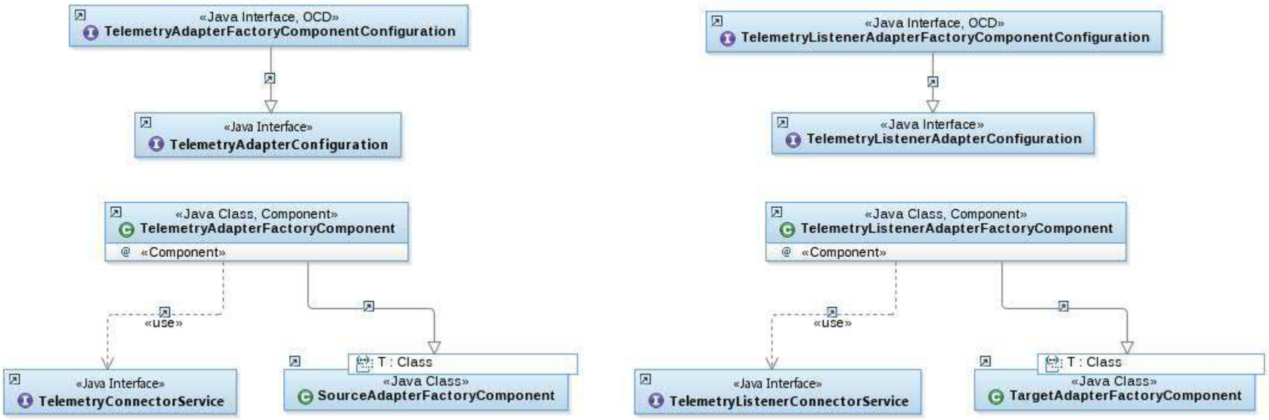

«Java Interface, OCD» TelemetryAdapterFactoryComponentConfiguration
«Java Interface, OCD» TelemetryListenerAdapterFactoryComponentConfiguration
«Java Interface» TelemetryAdapterConfiguration
«Java Interface» TelemetryListenerAdapterConfiguration
«Java Class, Component» TelemetryAdapterFactoryComponent
«Component»
«Java Class, Component» TelemetryListenerAdapterFactoryComponent
«Component»
«use»
«Java Interface» TelemetryConnectorService
T : Class
«Java Class» SourceAdapterFactoryComponent
«Java Interface» TelemetryListenerConnectorService
T : Class
«Java Class» TargetAdapterFactoryComponent

Figure 53 – TelemetryAdapterComponent

Components to provide an adapter for publishing and (optionally) receiving telemetry events, using a tracker for TelemetryConnectorService interfaces registered by other components that require a telemetry connection.

3. REFERENCES

[1]	PowerMatcher Communication Procol Specification, release 0.7.	05-02-2013
[2]	Lotus Expeditor Wiki : Lotus Expeditor 6.2.3 Documentation, section: “Developing an application using the MQTT API”, http://www-10.lotus.com/ldd/lewiki.nsf/dx/Developing_an_application_using_the_MQTT_API_XPD623	N/A
[3]	IBM WebSphere MQ Information Center, section: “Developing applications for WebSphere MQ Telemetry”, http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60000_.htm	N/A