MQSeries Everyplace - Administration tool Design

Version 1.0

17 August 2000

Phill van Leersum
MQSeries Development
IBM United Kingdom Laboratories
Hursley Park
Hursley
Hampshire, SO21 2JN
UK

phillvl@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, August 2000

This edition applies to Version 1.0 of *MQSeries Everyplace - Administration tool* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates.

Information contained in this report has not been submitted to any formal IBM test and is distributed “AS-IS”. The use of this information and the implementation of any of the techniques is the responsibility of the reader. Much depends on the ability of the reader to evaluate these data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and results obtained in other environments may vary significantly.

Trademarks and service marks

The following terms, used in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- MQSeries
- Everyplace

The following terms are trademarks of other companies:

• Java	Sun Corporation

Summary of Amendments

Date	Changes
17 August 2000	Initial release

Preface

This document describes the architecture of a platform-independent, low footprint, extensible admin solution for MQSeries Everyplace.
It makes use of UIA recommendations to avail itself of UIA compliant exchangeable renderers.
The architecture uses a layered approach to maximise reuse of components, and facilitate customisation, by IBM or its customers.
The design presented here is dependant upon the design of the UIA-compliant metamodel/rendering interface documented separately.

Bibliography

List any supporting publications here otherwise delete this page. Use following format:

- *User Interface Architecture*
- *MetaModel XML format*

1 Introduction

This document describes the architecture of a platform-independent, low footprint, extensible admin solution for Mseries Everyplace.
It makes use of UIA recommendations to avail itself of UIA compliant exchangeable renderers.
The architecture uses a layered approach to maximise reuse of components, and facilitate customisation, by IBM or its customers.
The design presented here is dependant upon the design of the UIA-compliant metamodel/rendering interface documented separately

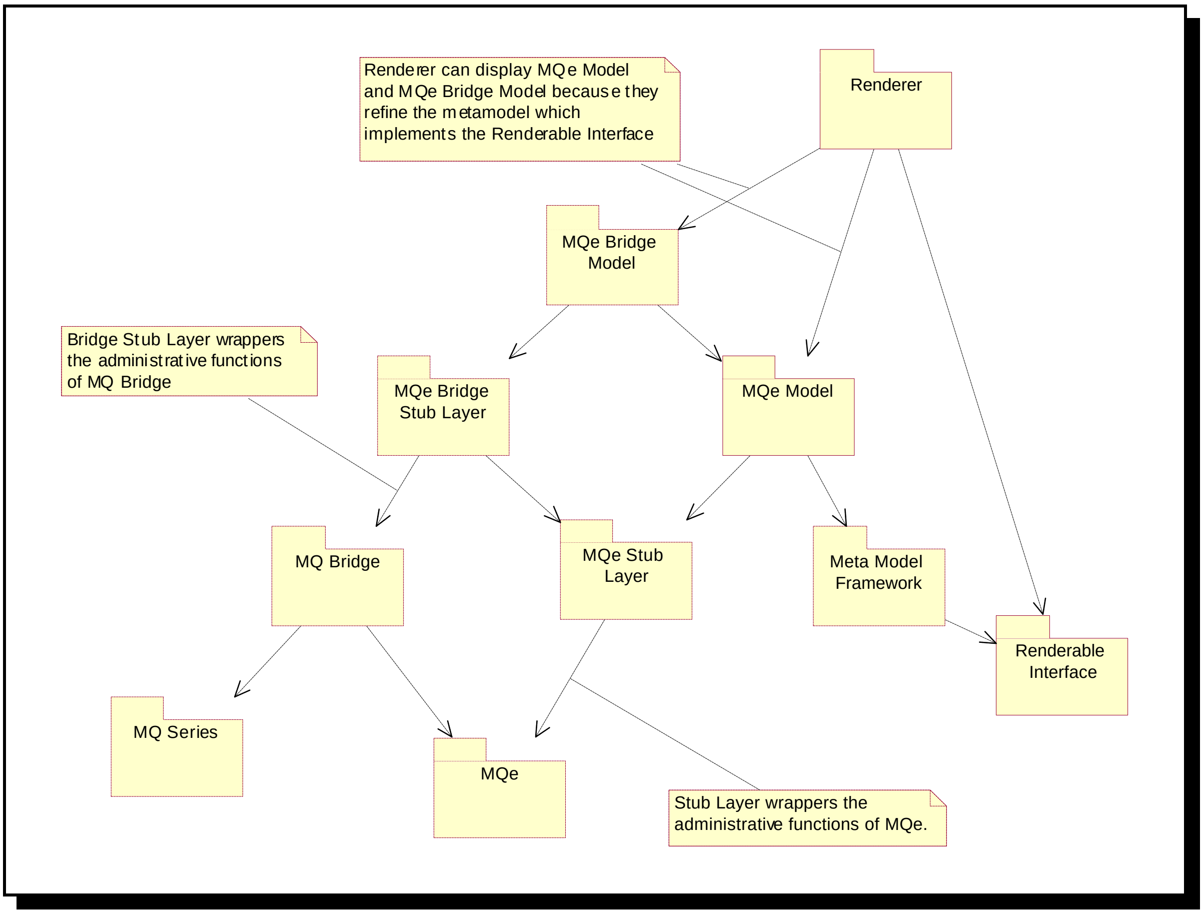
2 Architecture

Overall concept: Provides a layered implementation of RendereInterface for Mqe. Complies with UIA recommendations by using the rendering interface, and allowing the renderer the control of the GUI (this technology described elsewhere). The advantage of this approach is that a Renderer, once written, can be reused for many tools. This gives a common look-and-feel to al such tools, and ensures compliance with the UIA design guidelines.

The designed has been prototyped in fully-portable Java 1.1 compliant code. This means that it will work on any platform that provides a Java 1.1 compliant JVM. Swing is not required.
Although the architecture is layered, and designed for reuse, it does not imply a large footprint. The entire prototype, including the simple renderer but excluding MQe.jar, is a little less than 140k bytes.

The factory pattern for object creation is used throughout this architecture, to allow customisation of its behaviour through subclassing. Essentially, the factory pattern allows an instance of one class (A 'factory class') to act as a factory for instances of another class. This is achieved by the factory object displaying an instance method that creates an instance of the desired object (sometimes caled a 'factory method'). This factory method can then be overriden by the implementor of a subclass of the factory class. The overide can then create instances of its desired class (which must be a subclass of the original created class). The pattern allows large amounts of reuse of code to be integrated seamlessly with specialisations of the same code.
The code has been organised into ditinc packages, so that the modules can be reused most effectively. The clear division between base MQe administrative code and MQ Bridge administrative code allows delivery of the explorer in two forms:

- MQeExplorer: capable of administering all the base MQe objects. Minimum footprint. No dependancy on MQSeries
- MQeBridgeExplorer: Adds the ability to administer the MQe Bridge objects to the capabiliies of the MQe Explorer. Has a dependancy upon MQ Series for correct function.


Renderer can display MQe Model and MQe Bridge Model because they refine the metamodel which implements the Renderable Interface
Renderer
MQe Bridge Model
Bridge Stub Layer wrappers the administrative functions of MQ Bridge
MQe Bridge Stub Layer
MQe Model
MQ Bridge
MQe Stub Layer
Meta Model Framework
Renderable Interface
MQ Series
MQe
Stub Layer wrappers the administrative functions of MQe.

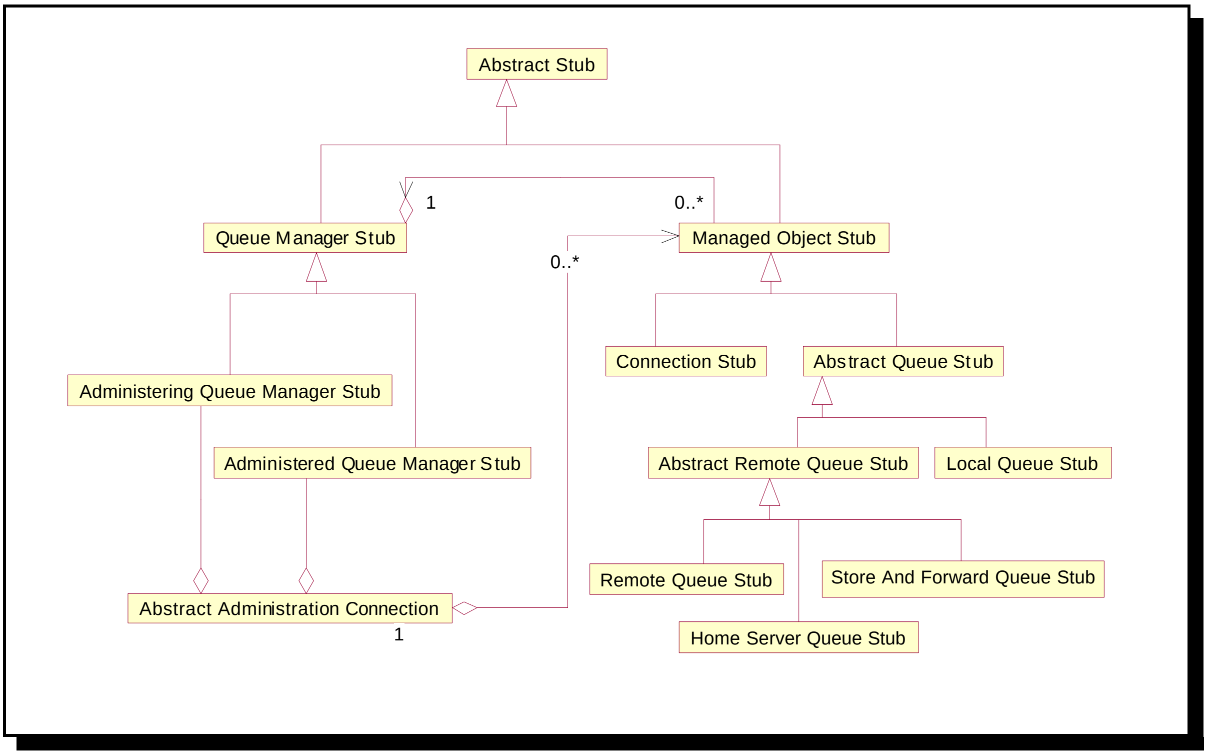
- MQSeries:
- MQe: Base MQe code.
- MQ Bridge: code for bridging between MQe and MQ Series.
- Stub Layer: This layer wrappers Mqe with classes that simplify the most common operations on MQe objects. This can easily be extended (by IBM, or by customers) to include new types of MQe objects.
- BridgeStub Layer: This layer wrappers MqeBridge with classes that simplify the most common operations on MQeBridge objects. This can easily be extended (by IBM, or by customers) to include new types of MQeBridge objects. This package extends the Stub Layer and is dependent upon it.
- Renderable Interace: provides a standard interface through which GUIs may render an exposed model, without detailed knowledge of the underlying model. Decouples the GUI from the underlying model. Allows any GUI to render any compliant model.
- MetaModel Framework: Provides a framework that facilitates the implementation of the Renderable interface. Allows much/most of the GUI representation to be encoded in XML. Provides xml rendering capabilities for the Model objects to allow them persistence/storage.
- Model Layer: exposes a subset of the functionality of the Stub Layer to the Renderer by implementing the RendererInterface (indirectly, by extending the metamodel).
- BridgeModel Layer: exposes a subset of the functionality of the BridgeStub Layer to the Renderer by implementing the RendererInterface (indirectly, by extending the Model layer and the metamodel). The Bridge model is an extension of, and dependent upon, the Model Layer.
- Renderer Implementation: A simple renderer has been implemented. This unsophisticated because it is built entirely using the highly portable AWT 1.1 code.   More sophisticated GUIs are expected to be provided by (amongst others) IBM Ease Of Use organisation.

3 Stub Layer

This layer wrappers Mqe with classes that simplify the most common operations on MQe objects. MQe is administered by the sending of administration messages. The particular message type, its contents, and its addressing are all critical for correct results.
Since the hierarchy of the stub layer reflects the logical hierarchy of Mqe objects, then addition to the latter can easily be reflected in the former.
This layer conforms to the highest standards of OO design. It is intended for release to customers as a tool for simplifying MQe administration, and for extension to accomodate customers subclasses of MQe resources.
The objects currently administered are

- MQeQueueManager
- MQeConnection
- MQeQueue
- MQeRemoteQueue
- MQeHomeServerQueue
- MQeStoreAndForwardQueue

The common operations implemented on all Mqe objects are Create, Delete, Update, and InquireAll. In addition, all queues and connections may have aliases added/deleted. Store And Forward queues can have queue manager entries added and deleted.
Operations are performed by polymorphic method dispatch on the hierarchy of stub objects. For synchronous solutions, the response message from MQe is returned to the method caller. For asynchronous solutions, the invoker can register as a listener and have the message returned to a cal-back method.


Abstract Stub
1
0..*
Queue Manager Stub
Managed Object Stub
0..*
Connection Stub
Abstract Queue Stub
Administering Queue Manager Stub
Administered Queue Manager Stub
Abstract Remote Queue Stub
Local Queue Stub
Remote Queue Stub
Store And Forward Queue Stub
Abstract Administration Connection
1
Home Server Queue Stub

This hierarchy strictly enforces the pattern that 'only none-leaf classes shall be instantiated'. While some may argue that this leads to uneccessary classes, it has been demonstrated that the code written in this style is more naturally extensible, and more resistant to defects.
Thus, for example, while LocalQueue does not represent a large refinement of the behaviour of AbstractQueue, and could be considered superfluous, it is subtly different. hs sbtle difference, when ignored, can send unwanted ripples of complexity and confusion through the design as the code is extended, mantaned, and modified.

3.1MQe Stub

represents all common stub behaviour.
implements create, delete, update, inquireAll.
Obtains correct admin message by polymorphic adminMessage() method dispatch.

3.2 Administering Queue Manager

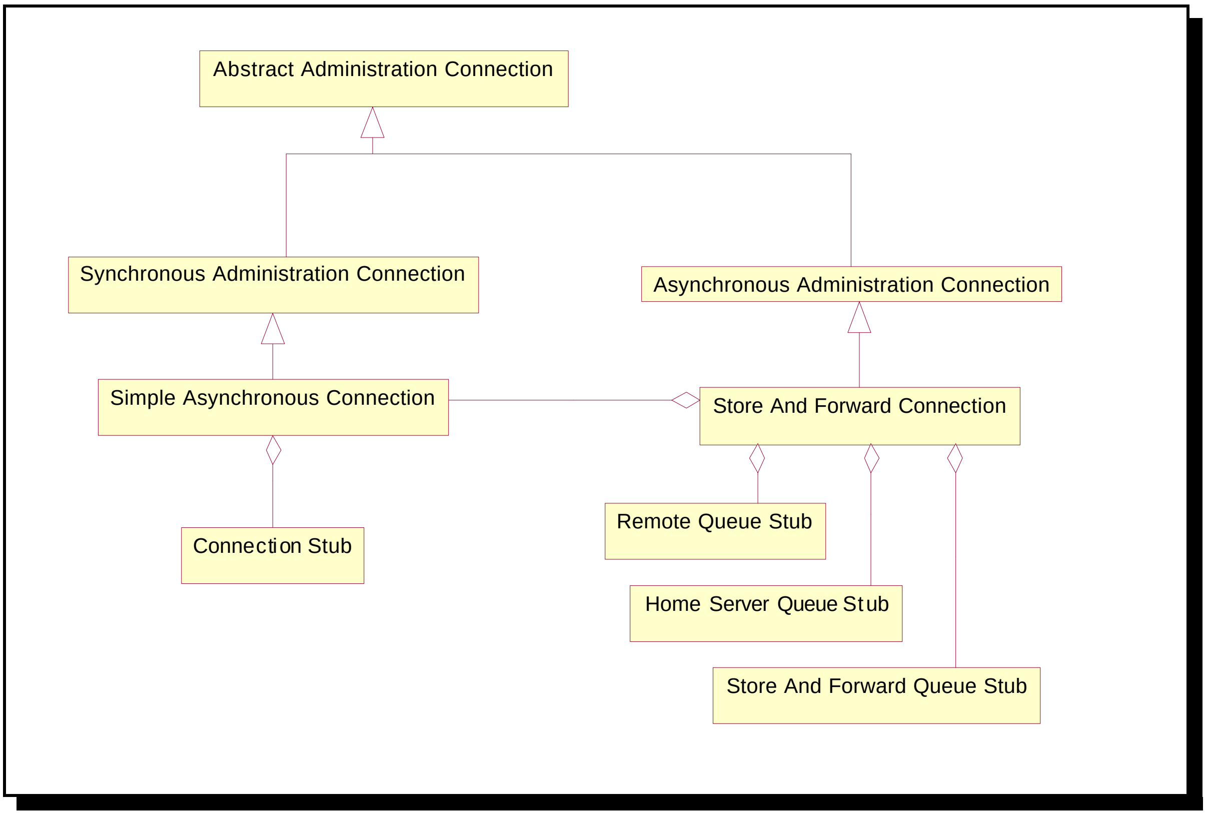
Each instance of the stub layer requires a queue manager as an entry point to the MQe system. Currently this is a queue manager created specificaly for this purpose. In later implementations it could easily be a queue manager that already exists.
The Administering Queue Manager is the MQe resource that actually places messages into, and receives messages from, the MQe system.
The Administering Queue Manager acts as a factory for Administered Queue Managers.

3.3 Administered Queue Manager Stub

Each MQeQueueManager being administered is represented by an AdministeredQueueManager stub. This stub allows operations on the queue manager.
Administrative Queue Manager stubs act as factories for stubs representing all the types of managed objects.
The Administrative Queue Manager stub allows the child managed objects to route their messages via the administrative connection.

3.4Administrative Connection

Each Administered Queue Manager stub is connected to the Administering Queue Manager by and Administrative Connection.
The connection controls the resources that allow the message to flow from the Administering Queue Manager to the MQeQueueManager that is represented by the Administered Queue Manager
The connection controls the dispatch and receipt of messages to and from the managed MQeQueueManager, via the Adminisering Queue Manager.


Abstract Administration Connection
Synchronous Administration Connection
Asynchronous Administration Connection
Simple Asynchronous Connection
Store And Forward Connection
Connection Stub
Remote Queue Stub
Home Server Queue Stub
Store And Forward Queue Stub

The simplest connection is a synchronous connection. In this only an MQeConnection from the Administering Queue Manager to the managed MQeQueueManager is required. Messages are addrssed to the managed queue manager, and MQe uses the connection to ensure that they arrive.

Reply messages are requested to the administered queue managers admin reply queue, and the connection polls this queue until either a reply is received, or a time-out occurs.
A more complex Admin Connection could involve the creation of a remote queue, a store and forward queue, and a home server queue, with asynchrounous return of messages.

3.5 Managed Object Stub

A managed object stub represents any MQe resourced managed by a queue manager.
Maintains a reference to the parent queue manager.
Implements the add/delete alias functionality. The correct constants for the messages are derived polymorhically from the subclasses.

3.6 Conection Stub

Implements connection specific functionality
Implements the polymorphic adminMessage() method.

3.7 Abstract Queue Stub

3.8 Local Queue Stub

Implements the polymorphic adminMessage() method.

3.9 Abstract Remote Queue Stub

3.10 Remote Queue Stub

Implements the polymorphic adminMessage() method.

3.11 Store and Forward Queue Stub

Implements the polymorphic adminMessage() method.

3.12Home Server Queue Stub

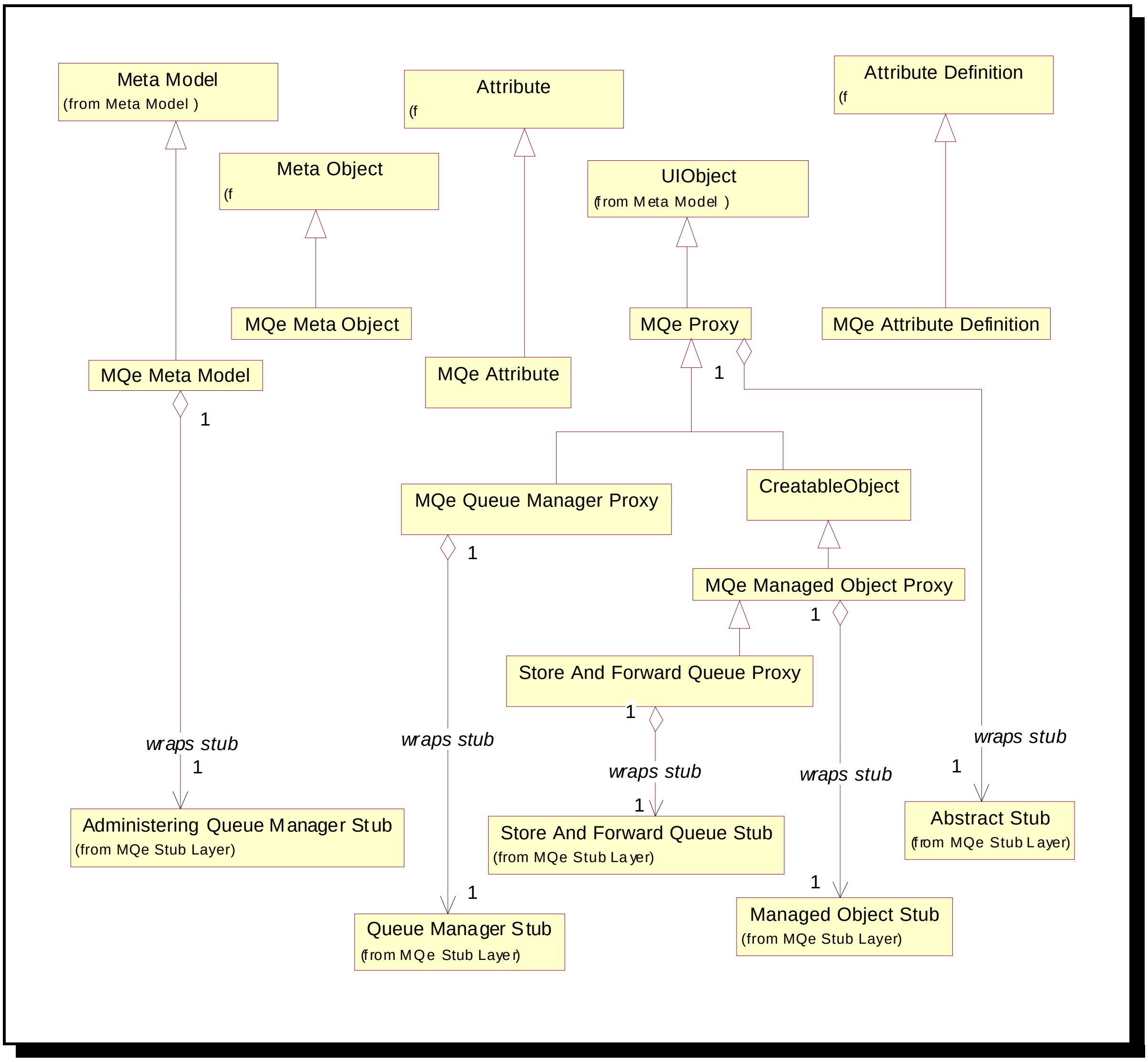
Implements the polymorphic adminMessage() method.

4 Model Layer

exposes a subset of the functionality of the Stub Layer to the Renderer by implementing the Renderer Interface. Acheives this by subclassing the metamodel appropriately. Thedesign of the metamodel and of the rendering interface are described separately.

- Flattens the hierarchy defined in the stub layer
- provides GUI display logic
- Wraps the stubs from the stub layer
- place to put attribute dependence.
- designed to add UI visibility to stub

4.1Design



4.1.1 MQe Meta Model

Subclasses Meta Model in order to overide the factory methods for creating Meta Objects. In this factory method it creates and returns an instance of MQe Meta Object.
Contains an instance of Administering Queue Manager Stub.

4.1.2 MQe Meta Object

Subclasses Meta Object to overide the factory method for creating UIObjects. In this factory method it creates and returns an instance of one of the subclasses of MQe Proxy as appropriate.

4.1.3 MQe Attribute Definition

Subclasses Attribute Definition to overide the factory method for Attributes. Creates and returns an instance of MQe Attribute.

4.1.4 MQe Attribute

Implements functionality specific to MQe model attributes. For example, reading attrbute values from, and writing attribute values to, MQe Fields objects (base MQe).

4.1.5 MQe Proxy

Superclass for all MQe proxies.
Introduces the notion of wrapping a stub.
Implements generic methods for create/delete/updateinquireAll.
Implements generic method for reading all attributes from an MQeFields object.

4.1.6 Queue Manager Proxy

Subclass of MQe Proxy.
The stub this class contains is an instance of Administered Queue Manager Proxy.
Exposes creation methods for all managed objects.
Exposes connect/disconnect methods .

4.1.7 Managed Object Proxy

Subclass of MQe Proxy.
The stub contaned in this class is an instance of a subclass of Managed Object Stub.
Exposes the add/remove alias functionality.

4.1.8 Store And Forward Queue Proxy

Subclass of Managed Object Proxy.
The stub contaned in this class is an instance of a Store And Forward Queue Stub.
Exposes functionality for add/remove Queue Manager entry.

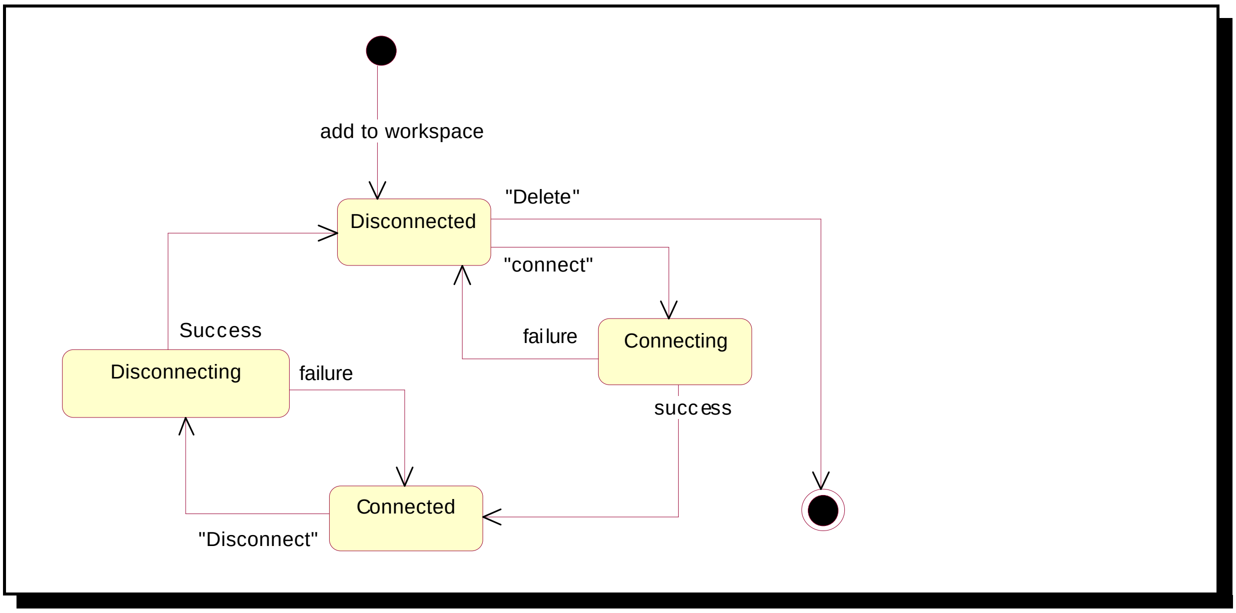
4.2 QueueManager State Model

The QueueManager states allow for connection/disconnection. Resources can be edited 'offline'.
The states are:

- Disconnected: No connection from administering machine to administered machine, (Blue).
- Connecting: Attempting to establish connection from administering machine to administered machine, (Cyan).
- Connected: A connection from administering machine to administered machine, exists (Blue).
- Disconnecting: Attempting to remove connection from administering machine to administered machine, (Cyan).

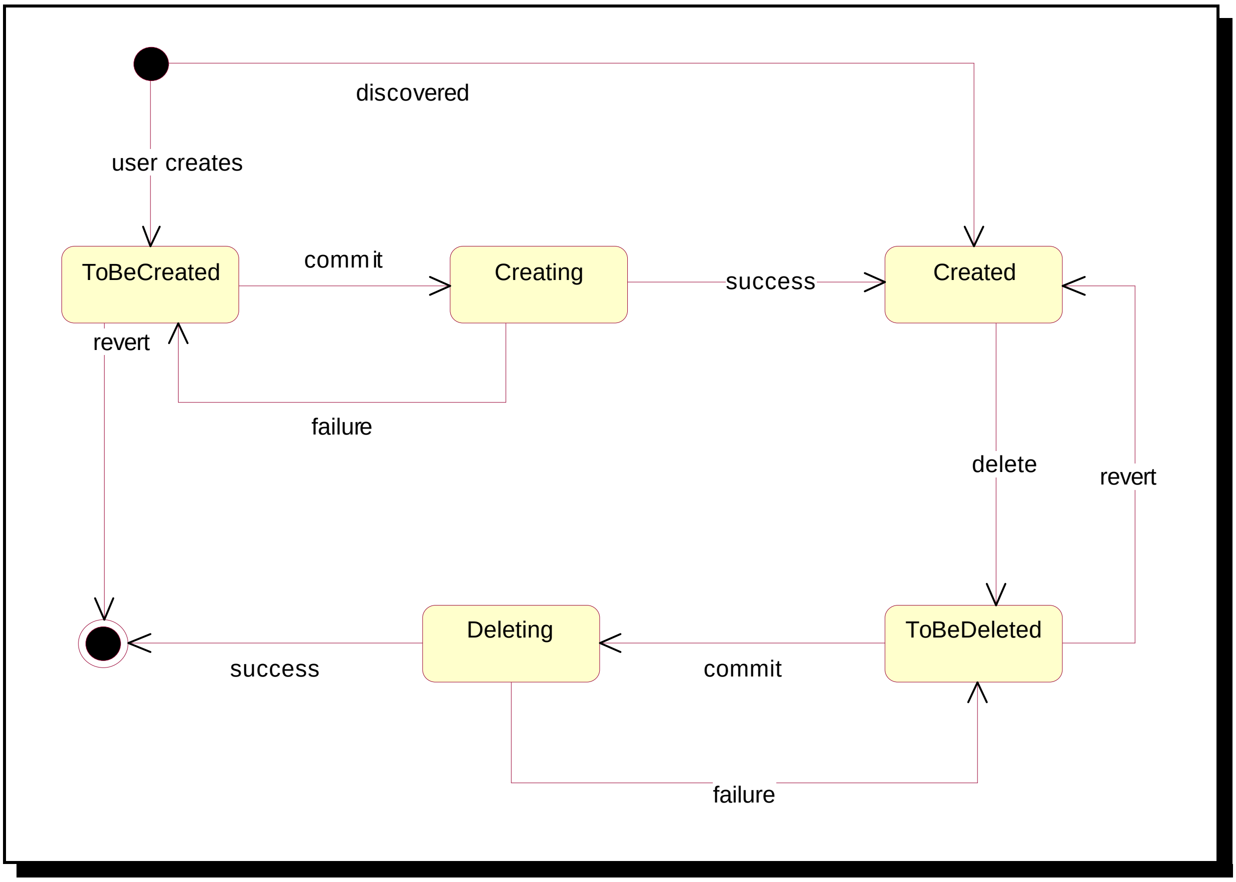
The events driving state chages are:

- add to workspace: The user adds a queue manager proxy to the workspace.
- "Connect": The user requests that the queue manager is connected;
- "Disconnect": The user requests that the queue manager is disconnected;
- success: an attempt to connect or disconnect succeeded.
- failure : an attempt to connect or disconnect succeeded.
- "Delete": The user requests that the queue manager is removed from the workspace.



4.3 Managed Object State Model

The creation and deletion allow for offline actions.
Actions are cached as states.
States can be commited or reverted.



States:

- ToBeCreated: initial state. indicates that the entity is waiting to be created on the real MQe system (Blue).

- Creating: indicates that the entity is being created on the real MQe system. Implies that a creation message has been sent (Cyan).
- Created: The entity is known to exist on the real MQe system (Black).
- ToBeDeleted: the entity requires deleting from the real MQe system (Red).
- Deleting: the entity is being deleted from the real MQe System (Magenta).

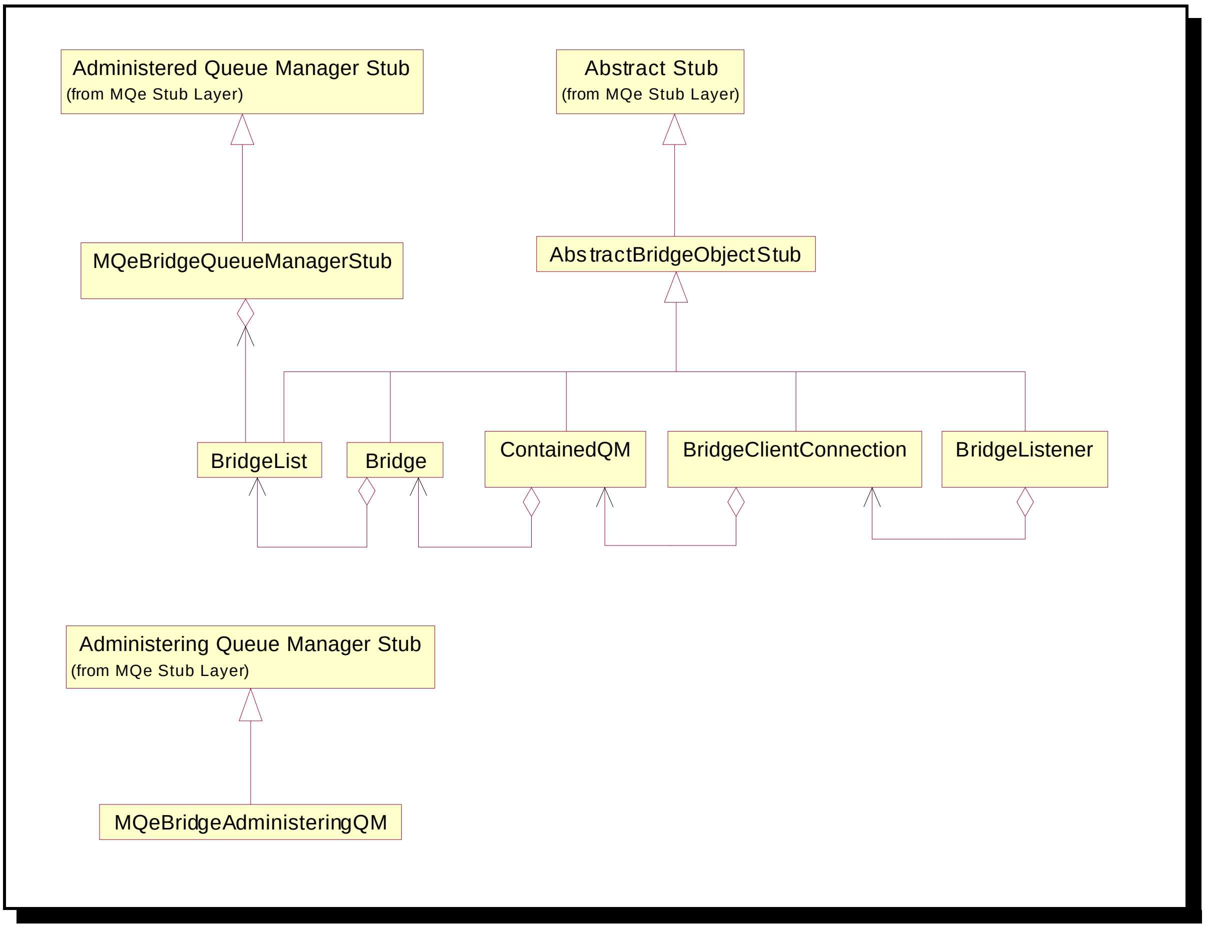
Events:

- user creates: user creates a proxy representing a resource
- discovered: a proxy is created to represent an MQe resource discovered by an inquiry.
- commit: user requests that pending changes be reflected in the real MQe resource. If the user is working 'online' then the 'commit' is attached invoked when any change is made. If the user is working offline, then 'commit' is invoked explicitly by the user. Commit implies the sending of an admin message to the underlying MQe resource.
- revert: the user undoes the pending changes. this is only meaningful when the user is working 'offline', and there are changes to be made.
- delete: the user specifies that the resource is to be deleted.
- success: an admin reply message is received indicating that the pending operation (creation or deletion) has been successful.
- failure: an admin reply message is received indicating that the pending operation (creation or deletion) has been unsuccessful.

.

5 Bridge Stub Layer

The bridge stub layer extends the stub layer and wrappers the administrative functions of the MQ bridge components.



5.1 MQeBridgeAdministeringQM

Subclasses AdministeringQueueManagerStub, soley to overide the factory method for creating administered queue managers.

5.2 MQeBridgeQueueManagerStub

Subclasses Administered Queue Manager Stub.
Provides factory method for creating BridgeLists.
Provides functionality to query existence of bridge list.

5.3Abstract Bridge Object Stub

Provides ability to start/stop/start all.

5.4 BridgeList

Provides a local representation of the Bridges object on the remote Bridge Queue Manager.
Specialises CreatableObject.
Provides factory method for creating Bridges

5.5 Bridge

Provides a local representation of the Bridge object on the remote Bridge Queue Manager.
Provides factory method for creating ContainedQMs

5.6 ContainedQM

Provides a local representation of a BridgeQueueManagerProxy object on the remote Bridge Queue Manager.
Provides factory method for creating BridgeClientConnections

5.7 BridgeClientConnection

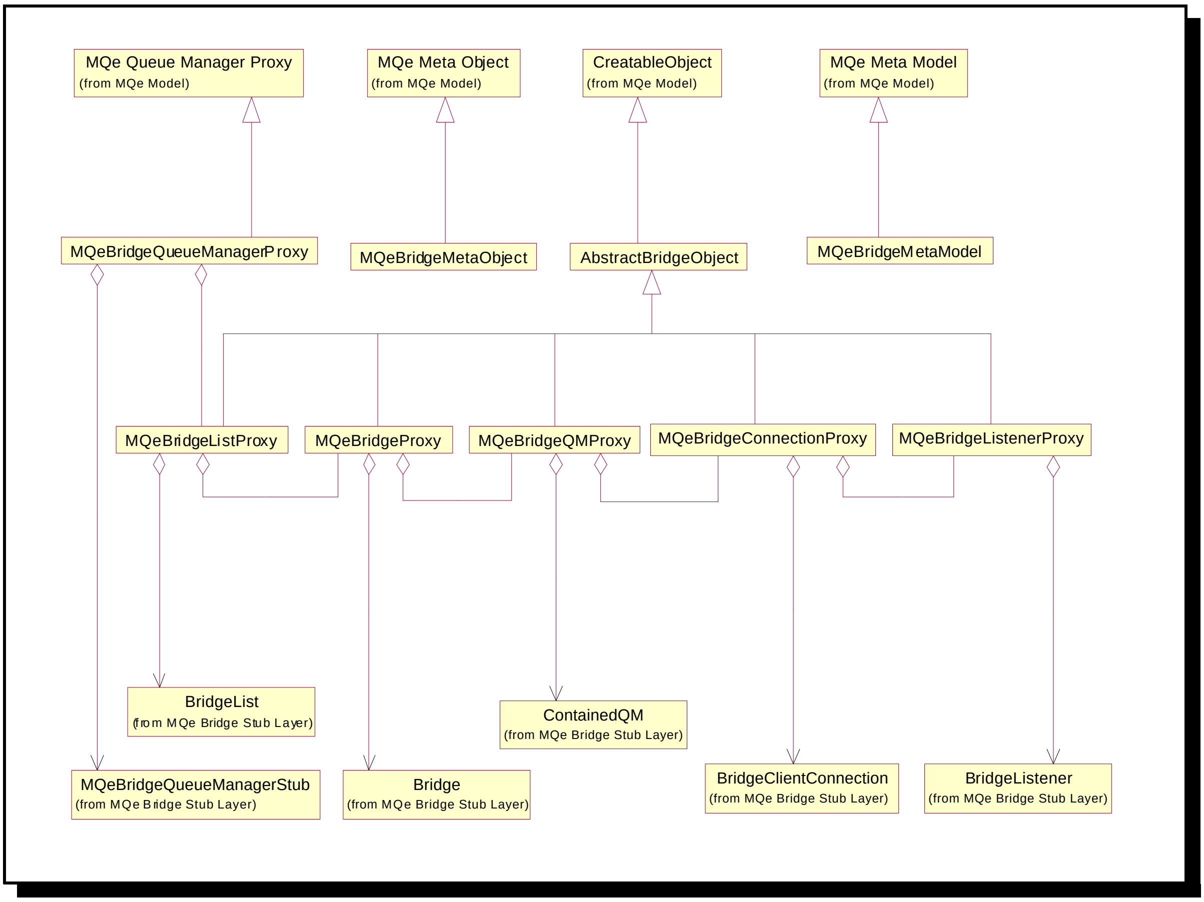
Provides a local representation of the BridgeConnection object on the remote Bridge Queue Manager.
Provides factory method for creating BridgeListeners.

5.8BridgeListener

Provides a local representation of a BridgeConnectionListener object on the remote Bridge Queue Manager.

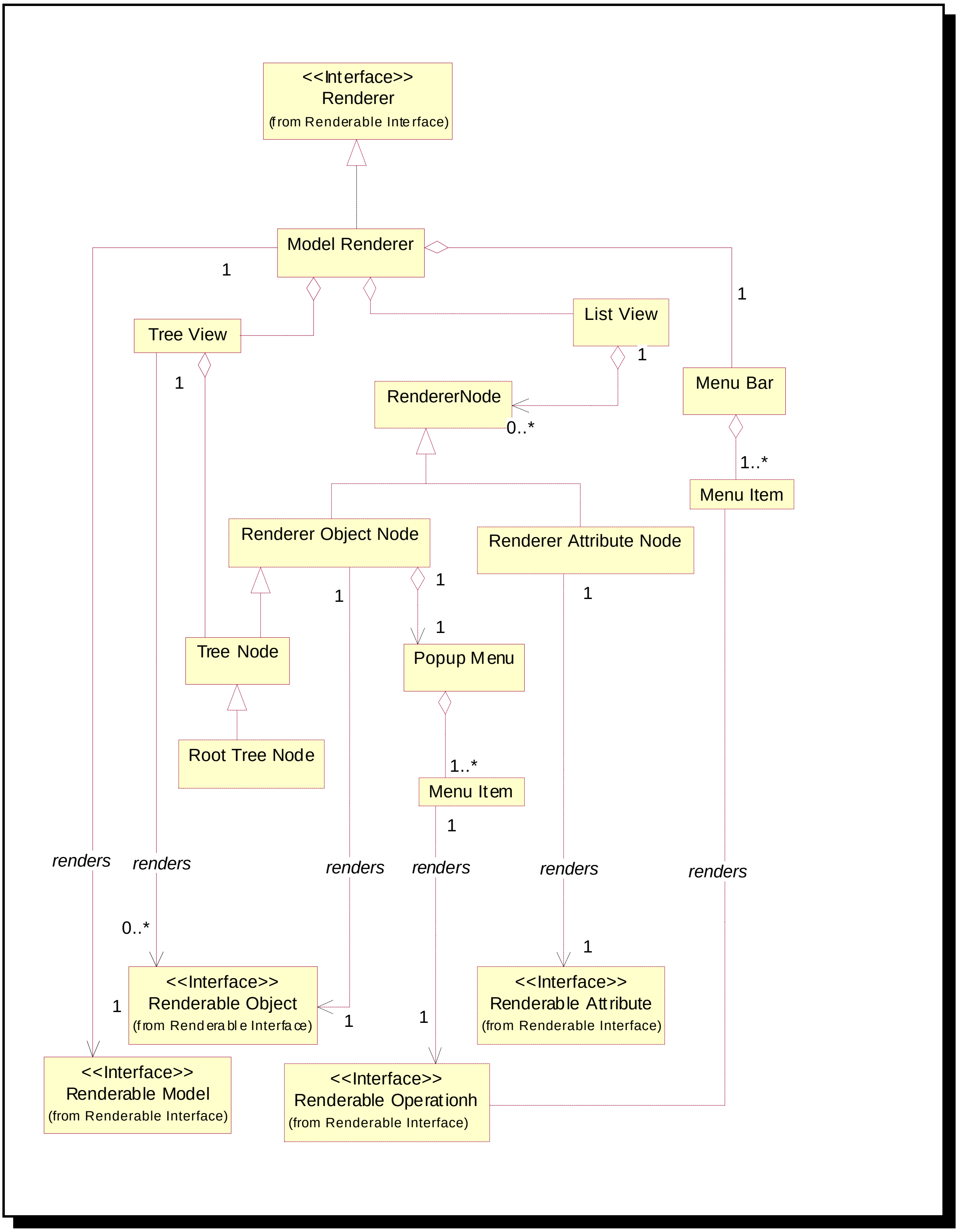
6 Bridge Model Layer

The bridge model extends the MQe model, and wrappers the bridge stub layer.



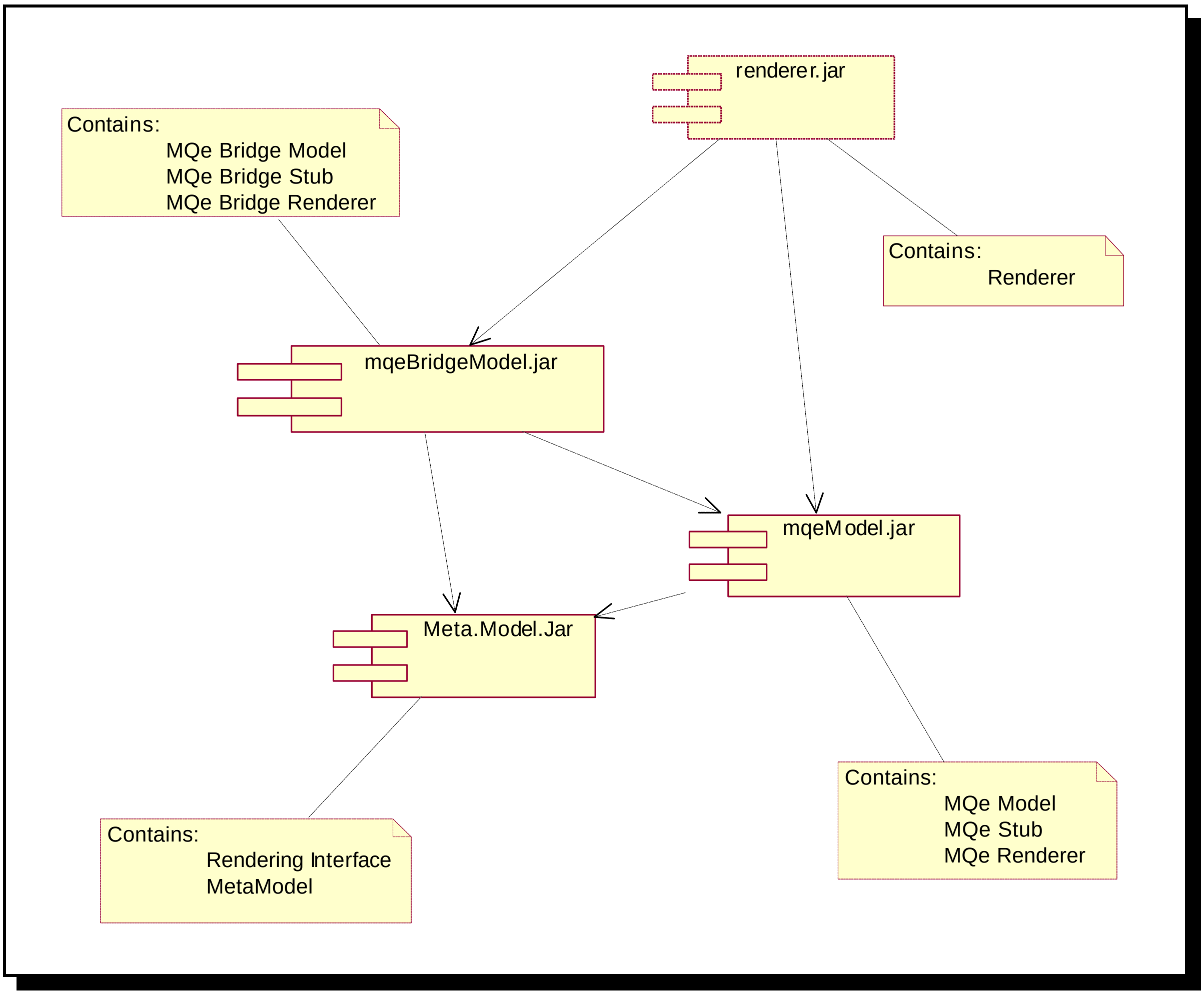
7 Renderer

A simple renderer has been implemented. This unsophisticated because it is built entirely using the highly portable AWT 1.1 code. More sophisticated GUIs are expected to be provided by (amongst others) IBM Ease Of Use organisation.



8 Packaging for Delivery

The Packages are be combined into four jar files for delivery:



For the simple MQe Explorer version the mqeBridgeModel.jar is not required.
Third parties can replace the renderer.jar with their own renderer
MQe Renderer and MQe Bridge Renderer are two packages that contain a single class each. This class has only static methods, and is used soley to create the correct metamodel from the metamodel xml file, and invoke the renderer. In future versions these packages should not be part of the mqeModel and mqeBridgeModel jars.