Politecnico di Milano

Software Engineering 2 Project
a.y. 2015-16

MyTaxiService
Inspection Document
Version 1.0

Casati Fabrizio, 853195
Castelli Valerio, 853992

Referent professor: Di Nitto Elisabetta

January 3, 2016

Contents

Chapter 1

Code description

1.1 Assigned classes and methods

The task of our group was to analyze the `ActiveJmsResourceAdapter` class of the Java GlassFish Application Server[1] project. This class is located in the `com.sun.enterprise.connectors.jms.system` package.

In particular, we were asked to perform a global code inspection of the class and then focus specifically on its `createManagedConnectionFactory` and `createManagedConnectionFactories` methods.

1.2 Functional role

This section provides an overview of the functional role of the `ActiveJmsResourceAdapter` class, and specifically of its `createManagedConnectionFactory` and `createManagedConnectionFactories` methods, in the general context of the whole GlassFish Project.

In order to make the analysis more clear and understandable, we provide a brief list of acronyms and definitions which are used throughout the code:

- **EIS: Enterprise Information System**. It's a generic term that refers to any kind of information system (or component thereof) used in an enterprise context, including ERP (Enterprise Resource Planning) systems, message brokers and others.
- **RA: Resource Adapter**. It's a component that implements the adapter design pattern to let two systems communicate with each other when there is a discrepancy between the offered and required interfaces. In the specific case of GlassFish, resource adapters are used to interface with external EISs.

[1] `https://glassfish.java.net/`

- **MQ: Message Queue**. Despite the name, this component usually doesn't simply implement a location where messages are stored until delivery, but takes care of the whole stack of messaging functionalities to enable communications between interconnected components.
- **MB: Message Broker**. It's a component that enriches the functionalities of a message queue by also providing appropriate mechanisms to translate messages between different formats. This is essential to adapt them to the different kinds of senders and receivers that exchange messages across the network.
- **imq properties**: set of configuration properties of the IBM WebSphere MQ software suite. GlassFish implements mechanisms to handle them as part of the resource adapter that provides the interface with IBM MQ compliant message brokers. They are supported mainly for compatibility reasons, since they've been superseded by later revisions of the MQ API.
- **AS7 properties**: set of properties of Application Servers, implemented both by JBoss and GlassFish. They include the already mentioned imq properties.
- **XA standard**: specification written by The Open Group for distributed transaction processing (DTP). It describes the interface between the global transaction manager and the local resource manager. The goal of XA is to allow multiple resources (such as databases, application servers, message queues, transactional caches, etc.) to be accessed within the same transaction, thereby preserving the ACID properties across applications. XA uses a two-phase commit to ensure that all resources either commit or rollback any particular transaction consistently (all do the same). In the case we are considering, the role of global transaction manager is implemented by a GlassFish instance, while the local resource managers are the different message queues (or message brokers) to which the application server is connected. [2]
- **RM: Resource Manager**. It is the component responsible for managing a set of resources. In the case we are considering, each message queue (or message broker) is a different resource manager.
- **MCF: Managed Connection Factory**. It is the common interface implemented by all the adapter classes which provide connections toward an external service. As specified by the official Oracle Java EE Documentation[3], a connection factory provides the necessary methods to create a connection to a provider and encapsulates all the required

[2] `https://en.wikipedia.org/wiki/X/Open_XA`

[3] `https://docs.oracle.com/javaee/5/tutorial/doc/bnceh.html`

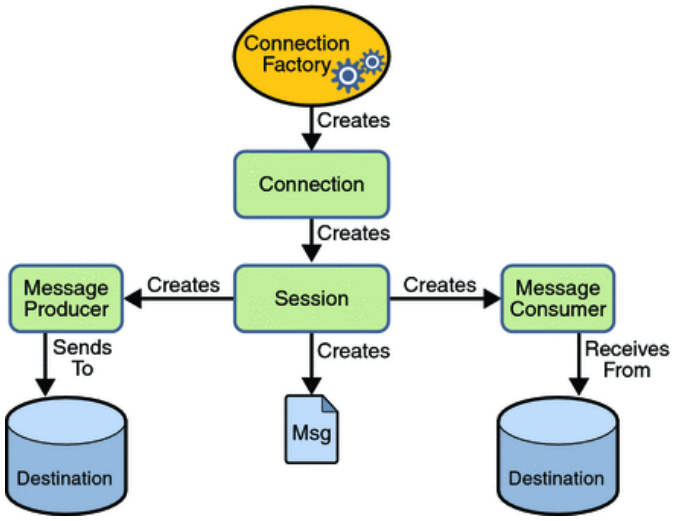
connection configuration parameters. In the case we are considering, the MCF is used to create and handle connections to message brokers implementing the JMS API.

Given these premises, it's then easy to understand that the primary purpose of `ActiveJmsResourceAdapter` is to provide a communication interface between GlassFish and any kind of external message broker supporting the JMS API.

As defined by Wikipedia[4],

> the **Java Message Service (JMS) API** is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a messaging standard that allows application components based on the Java Enterprise Edition (Java EE) to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous.

The full JMS specification is available at Oracle's Developer Website[5]; we include the key architectural diagram here:



The `ActiveJmsResourceAdapter` component is essential for GlassFish to provide its hosted applications the messaging functionalities of the JMS API without having to implement a full message broker directly; instead, the system administrator is able to freely choose one (or more) of the existing commercial message brokers and configure GlassFish to route all JMS calls to it. In practice, it gives hosted applications a virtual, monolithic

[4] `https://en.wikipedia.org/wiki/Java_Message_Service`
[5] `http://www.oracle.com/technetwork/java/jms/index.html`

JMS message broker that they can interact with, fully abstracting all the implementation details of the specific adopted software solution.

In particular, it has the ability to connect with multiple message brokers at the same time to evenly distribute the amount of messages to be routed for performance and scalability reasons. Since each message broker has its own specific set of APIs, the actual implementation of the communication adapter is not directly provided by `ActiveJmsResourceAdapter`, which only acts as a general interface, but is offered by a set of different classes (one for each message broker) implementing the `ManagedConnectionFactory` interface which are dynamically loaded at runtime when needed.

Each distinct message broker is identified in GlassFish with a connector, which in turn can be associated with multiple connections at the same time. All the implementation details required to correctly interact with the message broker, including the name of the adapter class to be loaded, the values of the configuration properties and parameters and the maximum number of supported connections are specified by a `ConnectorConnectionPool` object. Despite the name, this object doesn't implement itself a connection pool, as it doesn't keep track of the connections that have been opened toward a certain message broker, instead being specifically concerned with the configuration details of the interface. The information provided by the `ConnectorConnectionPool` object is then used to dynamically load and instantiate the required `ManagedConnectionFactory` class.

The `createManagedConnectionFactory` method is precisely involved with the setup of the `ManagedConnectionFactory` object responsible for the communication with a specified JMS compliant message broker. The first instruction is a call to the implementation of this method defined by the `ActiveResourceAdapter` superclass, which performs three tasks:

- It dynamically loads the class implementing the desired `ManagedConnectionFactory` implementation and instantiates it;
- For every property enumerated in the given `ConnectorConnectionPool`, it uses Java reflection to find out which is the corresponding setter method implemented by the `ManagedConnectionFactory` class and invokes it. This is essentially done because different `ManagedConnectionFactory` implementations support different sets of properties, which thus cannot be exposed at the interface level and therefore need to be discovered using Java reflection on the actual allocated class.
- If everything has been successful, it returns a reference to the created `ManagedConnectionFactory` instance; otherwise, it returns null.

However, this code is only able to configure properties for which there exists a setter named using the `<attribute, setAttribute(value)>` pat-

tern. Apparently, older implementations of the MQ API only provided properties in the form of `<property, value>` tuples which had to be passed to a `setProperty(String property, String value)` method; this properties starting with the "imq" prefix are thus not set by the initial call of the super `createManagedConnectionFactory` method.

For this reason, if a valid `ManagedConnectionFactory` instance is returned, the method proceeds to check all the properties stated in the `ConnectorConnectionPool` object looking for those who have "imq" as a prefix and, if any of them is found, it uses Java reflection to obtain the `setProperty(String property, String value)` method of the `ManagedConnectionFactory` instance and invokes it with the corresponding `<property, value>` tuple. If no such method is found, an exception is raised to notify the user that the configuration file is probably wrong (as it's trying to set a property that is not supported by the specific `ManagedConnectionFactory` instance).

For compatibility reasons, the transition from imq properties using the `setProperty(String property, String value)` method to properties set with the `<attribute, setAttribute(value)>` pattern has led to the necessity of supporting both the old and the new naming schemes. This means that most properties still have an old, deprecated imq version and a newer non-imq version which should be used in lieu of the old one. If the two are present with different values in the same file, the newer non-imq one should be considered as the legal one.

However, since the imq loop is executed after all "newer" properties have already been set, if a lazy developer has actually included both the imq version and the newer non-imq version of the same property in the same file, the value of the imq version will overwrite the non-imq one as its corresponding setter is called later.

Since this is in contrast with the specification, GlassFish includes a fix for this behavior which consists of simply re-invoking all the setter methods for the non-imq properties, with the single exception of the `addressList` property which is removed in case the address is equal to localhost or is empty (as it is considered to be "less important" than a non-empty and non-localhost imq `addressList`). This check is only performed if the interfacing module is a JMS module, which in fact is a superfluous check as `ActiveJmsResourceAdapter` only deals with JMS modules.

As we were mentioning earlier, `ActiveJmsResourceAdapter` is able to associate multiple message brokers to the same virtual connector in order to achieve better scalability and performances. The interactions between all the resource managers (in this case, message brokers) are governed by the XA standard, which uses two-phase commit to ensure that transactions related to the processing of messages are correctly handled by all nodes.

The `createManagedConnectionFactories` method is invoked when it's necessary to gather all the instances of `ManagedConnectionFactory` associ-

ated with a pool of message broker addresses.

As the author of the code states in the comment at the beginning of the method, this is particularly useful when it is necessary to send a message to all the message brokers in a broker cluster. To achieve this, it is necessary to simultaneously get a reference to all the message brokers in the cluster in order to eventually instantiate a connection with each of them.

The process is actually quite straightforward. At first, the method determines how many message brokers are part of the cluster by counting the number of remote addresses in the pool. If only one message broker is present, a single `ManagedConnectionFactory` is created; otherwise, a list of all addresses is obtained by analyzing both the imqAddressList and the AddressList properties specified by the `ConnectorConnectionPool` configuration. It should be noted that, if both properties are defined, the actual set of addresses that will be considered will depend on the order in which the two properties are stated. Furthermore, if no address is specified in the `ConnectorConnectionPool` or if the only address found is localhost, the address configuration is overridden by the set of addresses defined by the `addressList` attribute of `ActiveJmsResourceAdapter`. This behavior doesn't seem to be consistent and is not clearly documented, as we will discuss later.

For each address that has been obtained, a `ManagedConnectionFactory` is created. The process is similar to what we've already seen for the single `ManagedConnectionFactory`: the super method is invoked for the initial allocation and configuration phase, then all imq properties are set. The conflict resolution policy for imq and non-imq properties adopted in this case is different from what we've seen in the single factory case: this time, only the `addressList` property is resolved giving more importance to its non-imq version, while for all other properties the imq version overrides the non-imq one if both are present. It should be noted that the reasons for this discrepancy in the conflict resolution policy between `createManagedConnectionFactory` and `createManagedConnectionFactories` method is not documented, so there is no way to understand if this was made on purpose or if it's a bug in the code.

Chapter 2

Code issues

This chapter highlights all the bad practices that we have recognized during the inspection of `ActiveJmsResourceAdapter`, and specifically in methods `createManagedConnectionFactory` and `createManagedConnectionFactories`. The issues we have identified can be either traced back to specific violations of the checklist or to other bad programming practices.

2.1 Notation

- Single lines of code are denoted as L.123.
- Intervals of lines of code are denoted as L.123-456.
- Specific points in the code inspection checklist are denoted as **C1**, **C2**, ...

2.2 Checklist issues

2.2.1 `ActiveJmsResourceAdapter` class

1. **C1** Constant `ADDRESSLIST` at L.239 and variable `addressList` at L.255 rely only on Java's case sensitivity to be distinguished one from the other.
2. **C1** Variable `addressList` at L.255 and variable `urlList` at L.253 refer to similar concepts and should probably be more clearly described.
3. **C1** Variable `nm` at L.306 and variable `sm` at L.250 don't really suggest that they represent instances of `GlassfishNamingManager` and `StringManager`, respectively.
4. **C1** The name of variable `brkrPort` at L.257 is unnecessarily short (variable `brokerInstanceName` at L.297 uses the full word "broker").

5. **C1** The name of method `handles(ConnectorDescriptor cd, String moduleName)` at L.1161 doesn't clearly describe what this method is supposed to do. In practice, this method returns a boolean which is true if the `moduleName` is the name of a JMS resource adapter, but this is never stated in the name of the method itself.

6. **C5** The name of method `listToProperties` at L.1101 is not a verb. The same is valid for method `postConstruct` at L.342 and for method `postRAConfiguration` at L.860.

7. **C7** There is a huge number of constants which do not follow the naming convention at L.166-202, L.207-211, L.213, L.233-237, L.242, L.246, L.262-264.

8. **C8** and **C9** are not followed consistently throughout the code. Most of the class uses four spaces indentation, but tabs are also occasionally used as at L.357, L.493, L.537-556, L.992-997, L.1008-1010-, L.1026-1030, L.1102-1112, L.1351-1357, L.1973-1975.

9. **C10** Method `public void postConstruct()` at L.340 and method `private boolean isDAS()` at L.1739 don't follow the bracing style used throughout the rest of the class.

10. **C11** There are a number of single-line `if` statements that do not follow the convention. It would be very time consuming to enumerate them all, but at L.1245, L.1254, L.1303, L.1307 there are a few examples of this behavior.

11. **C14** There are a number of occurrences in which the line length exceeds the 120 character limit: at L.153 (130), L.423 (134), L.604 (169), L.756(123), L.794 (130), L.1355 (137), L.1358 (136), L.1841 (131), L.1922 (125), L.1958 (141), L.2005 (132), L.2018 (141), L.2137 (130), L.2259 (125), L.2282 (149), L.2287 (132), L.2293 (131), L.2317 (126), L.2330 (145), L.2331 (129), L.2337 (135), L.2339 (125), L.2346 (129), L.2358 (129), L.2361 (125), L.2366 (149), L.2393 (135), L.2407 (133), L.2409 (135), L.2440 (123), L.2471 (128), L.2587 (135), L.2602 (131).

12. **C18** Some methods don't have an explanation of what they're supposed to do. Also, large pieces of code are left uncommented or only have extremely brief explanations of their purpose. This makes the process of verifying the behavior of the code very hard, as it's not clear what the expected result should be and what's the rationale behind it.

13. **C19** None of the occurrences of commented out code contains an explanation as to why it has been commented out.

14. **C23** Javadoc is not complete. In particular, the following public methods are not documented:

    - `public int getAddressListCount()` at L.2512.
    - `public static String getBrokerInstanceName(JmsService js)` at L.1137.
    - `public boolean getDoBind()` at L.1534.
    - `public Set<String> getGrizzlyListeners()` at L.441
    - `public void handleRequest(SelectableChannel selectableChannel)` at L.2553.
    - `public boolean handles(ConnectorDescriptor cd, String moduleName)` at L.1161.
    - `public boolean initializeService()` at L.2538.
    - `public void postConstruct()` at L.342.
    - `public void setMasterBroker(String newMasterBroker)` at L.2578.
    - `public void setup()` at L.562.
    - `public void validateActivationSpec(ActivationSpec spec)` at L.1162.

15. **C25** There are a number of issues in the order of declarations:

    - Static variables are not listed in order of visibility. Package level and protected static variables are not used, however private and public static variables are mixed together. See L.156-246, L.262-271, L.276-291 for reference.
    - Instance variables are interleaved with static variables instead of being declared all at once after them. See L.250-259, L.273, L.295-332 for reference.

16. **C27** There are a number of issues:

    - This class is probably too long, counting 2612 lines of code (including comments). It should probably be refactored to be more manageable.
    - There are a number of methods which seem to be too long and should probably be split. They are:
        - `private void setAvailabilityProperties()` at L.582-762.
        - `private void loadDBProperties(JmsAvailability jmsAvailability, ClusterMode clusterMode)` at L.764-828.
        - `private void setLifecycleProperties()` at L.919-1099.

  - `private void setJmsServiceProperties(JmsService service)` at L.1601-1673.
  - `public ManagedConnectionFactory [] createManagedConnectionFactories (com.sun.enterprise.connectors.ConnectorConnectionPool cpr, ClassLoader loader)` at L.1860-1941.
  - `public ManagedConnectionFactory createManagedConnectionFactory(com.sun.enterprise.connectors.ConnectorConnectionPool cpr, ClassLoader loader)` at L.1970-2043.
  - `public void updateMDBRuntimeInfo(EjbMessageBeanDescriptor descriptor_, BeanPoolDescriptor poolDescriptor)` at L.2061-2220.
- Method `public Set<String> getGrizzlyListeners()` at L.441 is breaking encapsulation.
- Methods `createManagedConnectionFactory` and `createManagedConnectionFactories` partly contain duplicated code.

17. **C44** Constants at L.268-271 should probably be refactored as a single enum with four different values.

2.2.2 `createManagedConnectionFactory` method

1. **C1** Parameter `cpr` and variables `mcf` (L.1971), `s` (L.1976) and `array` (L.2011) are not particularly meaningful.

2. **C2** Variable `s` at L.1976 is named with a single character although it's not properly a "throwaway" variable.

3. **C10** There are a number of `if` and `for` statements that don't follow the bracing style followed by the rest of the class. This happens at L.2007-2008 and L.2012-2013, L.2014-2015.

4. **C18** This method is effectively undocumented. There are a number of instructions, especially related to the resolution of replicated properties, which are totally left uncommented. This makes the process of verifying the behavior of the code very hard, as it's not clear what the expected result should be and what's the rationale behind it.

5. **C28** Typecasts at L.1979 and L.2016 could potentially produce an unhandled `ClassCastException` if another thread has messed up the content of the set returned by `cpr.getConnectorDescriptorInfo().getMCFConfigProperties()`.

6. **C30** The invocation `configProperties.toArray()` at L.2011 doesn't check if `configProperties` is not `null`.

7. **C30** The assignment `Iterator it = s.iterator()` at L.1977 should check that `Set s` is not `null` to avoid raising an unhandled `NullPointerException`.

8. **C30** The assignment `String propName = prop.getName()` at L.1980 should check that `prop` is not `null` to avoid raising an unhandled `NullPointerException`. Since the collection is not accessed in a thread-safe way, the returned item could be `null`, resulting in a dangerous situation. For the same reason, the `propName` variable itself should be checked.

9. **C30** The `if` statements at L.2017-2018 should verify that `property` is not `null` before accessing it.

10. **C33** Attribute `moduleName` is declared in L.2006 which is not at the beginning of a block.

11. **C33** Attribute `setMethodAction` is declared in L.2025 which is not at the beginning of a block.

12. **C36** The invocation of a method via java at L.1989 reflection returns an `Object` object which is not stored nor used. This may be coherent with the expected behavior of the `setProperty(String, String)` method which is found via reflection, but it's worth checking.

13. **C36** Method `configProperties.remove(property)` at L.2020 could return `false` if the specified `property` is not present in the `configProperties` set. In practice this shouldn't happen if threads are managed correctly, as `property` is an item of the array obtained by invoking `configProperties.toArray()`, however it would probably be a good idea to put an extra check.

14. **C36** Method `setMethodAction.run()` at L.2027 could potentially return an `Object` object which is not store nor used. In practice, `setMethodAction.run()` always return `null`, so this is not particularly significant; however, it highlights a bad design decision, as `setMethodAction.run()` should probably be a void method instead.

15. **C37** and **C38** The usage of an iterator at L.1976-1979 is potentially not thread safe. A different thread could attempt to make modifications to the set returned by `cpr.getConnectorDescriptorInfo().getMCFConfigProperties()`, producing an inconsistent state.

16. **C42** Error messages could be more informative. Given the way they're written, they seem to be intended mostly to be read and interpreted by the same developer who has written the code, which in general is not the case. In particular:

- The error message at L.1992-1994 says that the setter method could not be found, but it could be even clearer by also specifying that the error has been raised while trying to perform java reflection on a `ManagedConnectionFactory` object. We think this could be an important information to give to developers for troubleshooting purposes, as we would expect to be able to invoke the `setProperty(String, String)` method if a non-`null` imq property is specified for the given `ManagedConnectionFactor`.
- The error message at L.1997-1999 is marked as "severe", but doesn't actually report what kind of exception has been raised.
- The log message at L.2019 is not sufficiently informative. It suggests that something could be wrong if the `addressList` property is `null`, empty or set to localhost, but it doesn't actually say why that should be the case in the message.
- The error messages at L.2031-2036 could probably be unified.
- All error messages should probably be put in constants somewhere for better maintainability.

17. **C44** `!"".equals(prop.getValue())` at L.1984 should be replaced by `!prop.getValue.isEmpty()`, which is more understandable and readable.

18. **C44** `"".equals(property.getValue())` at L.2018 should be refactored as `property.getValue().isEmpty())` and `"localhost".equals(property.getValue())` should be refactored as `property.getValue().equals("localhost")`. Also, `"localhost"` should be a constant.

19. **C44** The whole block of code at L.2010-2024 is a workaround to the fact that `cpr.getConnectorDescriptorInfo().getMCFConfigProperties()` returns a generic `Set` object instead of a more appropriate `Set<ConnectorConfigProperty>` using generics. Because of this, the developer has to manually convert the `Set` into an `Array` of `Object` objects and then manually test that they are actually instances of `ConnectorConfigProperty`. This is a terrible design decision and should be refactored.

20. **C50** The whole `try-catch` block at L.2009-2038 is badly designed.

    - The invocation `configProperties.toArray()` at L.2011 doesn't check if `configProperties` is not `null`. This could result in an unnecessary `NullPointerException`.
    - The typecast at L.2016 could result in a `ClassCastException` being raised if someone has messed around in inappropriate ways

with the `configProperties` collection. This modification should not be allowed by design, instead of being caught this way.

21. **C52** The `catch(Exception ex)` clause at L.1996 is very generic. This could be appropriate for the context as it may be difficult to predict all possible exceptions of the `setProperty(String, String)` method, supposing that this method can be developed in many different ways depending on the specific `ManagedConnectionFactory` implementation (in fact, it's not explicitly defined by the interface); however, it is still useful to highlight this, as maybe it could be further refined depending on the exact nature of the object we are expecting to deal with.

22. **C52** The `catch(Exception ex)` clause at L.2028 is similarly bound to be generic in order to handle the possible exceptions raised by `setMethodAction.run()` at L.2027. However, it could be refined to properly catch exceptions mentioned in point 11 of this list.

23. **C53** The `catch()` clauses at L.1991, L.1996 and L.2028 are not properly handling the error, limiting themselves to logging it. This is probably necessary given the fact that the caught exceptions are not under the developer's control as they mostly depend on invocation of methods obtained by reflection; however possible improvements should be considered for those exceptions that are raised by the code that the developer has written (as mentioned in point 11 of this list), even by just improving the content of the error messages.

24. **C56** The `while` loop at L.1978-2003 could behave strangely if set over which it's iterating is modified by a concurrent thread. It should be isolated and executed in a synchronized way.

2.2.3 `createManagedConnectionFactories` method

1. **C1** Parameter `cpr` and variables `mcfs` (L.1866), `mcf` (L.1893) and `s` (L.1874) are not particularly meaningful.

2. **C1** The name of variable `addressProp3` at L.1922 is not totally satisfying: it represents for sure an address property, but why it has the `3` suffix is not clear.

3. **C2** Variable `s` at L.1874 is named with a single character although it's not properly a "throwaway" variable.

4. **C18** This method is effectively undocumented. There are a number of instructions, especially related to the resolution of replicated properties, which are totally left uncommented. This makes the process of

verifying the behavior of the code very hard, as it's not clear what the expected result should be and what's the rationale behind it.

5. **C28** Typecasts at L.1877 and L.1896 could potentially produce an unhandled `ClassCastException` if another thread has messed up the content of the set returned by `cpr.getConnectorDescriptorInfo().getMCFConfigProperties()`.

6. **C30** The assignment at L.1875 should check that `Set s` is not `null`.

7. **C30** The assignment at L.1878 and the `if` statement at L.1879 should check that `prop` is not `null`.

8. **C30** The assignment at L.1894 should check that `Set s` is not `null`.

9. **C30** The assignments at L.1897-1898 and the `if` statement at L.1899 should check that `prop` is not `null`.

10. **C33** Attribute `mcfs` is declared at L.1866 which is not at the beginning of a block.

11. **C33** Attribute `tokenizer` is declared at L.1883 which is not at the beginning of a block.

12. **C33** Attribute `addressProp3` is declared at L.1922 which is not at the beginning of a block.

13. **C33** Attribute `addressProp` is declared at L.1929 which is not at the beginning of a block.

14. **C33** Attribute `setMethodAction` is declared at L.1931 which is not at the beginning of a block.

15. **C36** The invocation of a method via java at L.1905 and L.1907 reflection returns an `Object` object which is not stored nor used. This may be coherent with the expected behavior of the `setProperty(String, String)` method which is found via reflection, but it's worth checking.

16. **C36** Method `setMethodAction.run()` at L.1933 could potentially return an `Object` object which is not store nor used. In practice, `setMethodAction.run()` always return `null`, so this is not particularly significant; however, it highlights a bad design decision, as `setMethodAction.run()` should probably be a void method instead.

17. **C37** and **C38** The usage of an iterator at L.1875-1877 and L.1894-1896 is potentially not thread safe. A different thread could attempt to

make modifications to the set returned by `cpr.getConnectorDescriptorInfo().getMCFConfigProperties()`, producing an inconsistent state.

18. **C42** Error messages could be more informative. Given the way they're written, they seem to be intended mostly to be read and interpreted by the same developer who has written the code, which in general is not the case. In particular:

    - The error message at L.1910-1912 says that the setter method could not be found, but it could be even clearer by also specifying that the error has been raised while trying to perform java reflection on a ManagedConnectionFactory object. We think this could be an important information to give to developers for troubleshooting purposes, as we would expect to be able to invoke the setProperty(String, String) method if a non-`null` imq property is specified for the given ManagedConnectionFactory.
    - The error message at L.1915-1917 is marked as "severe", but doesn't actually report what kind of exception has been raised.
    - Logged message at L.1864 contains the word "AJMSRA". It would probably be more comprehensible if it contained the full "ActiveJMSResourceAdapter" name of which it is the acronym.
    - All error messages should probably be put in constants somewhere for better maintainability.

19. **C44** `!"".equals(prop.getValue())` at L.1899 should be replaced by `!prop.getValue.isEmpty()`, which is more understandable and readable.

20. **C52** The `catch(Exception ex)` clause at L.1914 is very generic. This could be appropriate for the context as it may be difficult to predict all possible exceptions of the `setProperty(String, String)` method, supposing that this method can be developed in many different ways depending on the specific `ManagedConnectionFactory` implementation (in fact, it's not explicitly defined by the interface); however, it is still useful to highlight this, as maybe it could be further refined depending on the exact nature of the object we are expecting to deal with.

21. **C52** The `catch(Exception ex)` clause at L.1934 is similarly bound to be generic in order to handle the possible exceptions raised by `setMethodAction.run()` at L.1933.

22. **C53** The `catch(Exception ex)` clause at L.1934 is empty. This is a very poor design choice, as one would expect that at least some logging would be performed to document that an exception has been raised.

23. **C53** The `catch()` clauses at L.1909 and L.1914 are not properly handling the error, limiting themselves to logging it. This is probably necessary given the fact that the caught exceptions are not under the developer's control as they mostly depend on invocation of methods obtained by reflection; however the content of the error messages could probably be improved, for example by including the full exception type in the log at L.1916.

24. **C56** The `while` loop at L.1895-1921 could behave strangely if set over which it's iterating is modified by a concurrent thread. It should be isolated and executed in a synchronized way.

2.3 Other issues

- String literal "imq" at L.1984 should be refactored as a constant.
- String literals "imqAddressList" at L.1879 and L.1904, "AddressList" at L.1879, "localhost" at L.1884 and "imq" at L.1899 should be refactored as constants.
- There are a number of occurrences in which private instance variables are directly exposed by getter methods without cloning them. While this is perfectly acceptable for private methods, it's not appropriate for public methods as it is a violation of the encapsulation principle of object oriented programming that can potentially lead to very serious issues. For example, the `public Set<String> getGrizzlyListeners()` method at L.441 is precisely following this kind of approach. Despite having already reported this issue in the checklist, we are reporting it again here as it doesn't seem to be limited to this class, but affects large portions of the GlassFish codebase; another example of this can be found in the `public Set getMCFConfigProperties()` of the `ConnectorDescriptorInfo` class, which also returns a `Set` instead of a `Set<ConnectorConfigProperty>`. These issues force the author of the code to perform all kinds of checks to make sure the content of the variable is not modified in unsafe ways and pose a threat to the stability and security of the software.
- There is a lot of code performing similar functionalities but in a different fashion. This is not duplicated code, strictly speaking: the code itself is not identical, but it's implementing very similar functionalities. This can be seen, for instance, by comparing the `createManagedConnectionFactory` and `createManagedConnectionFactories` method, which implement different policies for dealing with the resolution of duplicate-properties. This is confusing at best and dangerous at worst, as it can lead to strange and inconsistent behaviors: in this case, for

instance, the system resolves conflicts differently if it's dealing with a single message broker or with multiple message brokers, but one would expect the former to be just a specialization of the latter.

- Key parts of the code are not thread safe. This comes partly as a result of the author not using specific synchronization techniques and thread safe classes, and partly because of the violation of encapsulation we have already mentioned. Depending on the actual usage of this class inside GlassFish, this could possibly lead to strange issues which could prove very difficult to track down and solve.

Appendix A

Checklist

Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.
6. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: `_windowHeight`, `timeSeriesData`.
7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

8. Three or four spaces are used for indentation and done so consistently.
9. No tabs are used to indent.

Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

11. All `if`, `while`, `do-while`, `try-catch`, and `for` statements that have only one statement to execute are surrounded by curly braces. Example: avoid this:

```
if ( condition )
    doThis();
```

instead do this:

```
if ( condition )
{
    doThis();
}
```

File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

13. Where practical, line length does not exceed 80 characters.

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.

16. Higher-level breaks are used.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

Java Source Files

20. Each Java source file contains a single public class or interface.

21. The public class is the first class or interface in the file.

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

Class and Interface Declarations

25. The class or interface declarations shall be in the following order:

    (a) class/interface documentation comment;
    (b) class or interface statement;
    (c) class/interface implementation comment, if necessary;
    (d) class (static) variables;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);

iv. last private instance variables.

(f) constructors;

(g) methods.

26. Methods are grouped by functionality rather than by scope or accessibility.

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

29. Check that variables are declared in the proper scope.

30. Check that constructors are called when a new object is desired.

31. Check that all object references are initialized before use.

32. Variables are initialized where they are declared, unless dependent upon a computation.

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. Check that parameters are presented in the correct order.

35. Check that the correct method is being called, or should it be a different method with a similar name.

36. Check that method returned values are used properly.

Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

39. Check that constructors are called when a new array item is desired.

Object Comparison

40. Check that all objects (including Strings) are compared with `equals` and not with `==`.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).
45. Check order of computation/evaluation, operator precedence and parenthesizing.
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero.
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
49. Check that the comparison and Boolean operators are correct.
50. Check throw-catch expressions, and check that the error condition is actually legitimate.
51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.
53. Check that the appropriate action are taken for each catch block.

Flow of Control

54. In a `switch` statement, check that all cases are addressed by break or return.
55. Check that all switch statements have a default branch.
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.
58. Check that all files are closed properly, even in the case of an error.
59. Check that EOF conditions are detected and handled correctly.
60. Check that all file exceptions are caught and dealt with accordingly.

Appendix B

Hours of work

To redact this document, we spent 20 hours per person.