Describing and Improving Design Patterns Found in spring-amqp and spring-rabbit

Shruti Gupta – 16202254,University College Dublin

Author Note

As part of my industrial experience, I worked with spring-amqp and spring-rabbit libraries extensively to create the messaging layer for an application. I found the API's design to be quite flexible to extension and modification. After undertaking the Design Patterns module, I was of the opinion that the libraries had to be making use of some design patterns to achieve this level of adaptability, and if not perhaps a design pattern could be used to enhance its flexibility. To that end I set out to study the code of both the libraries, and this paper gives a summary of my findings.

Contents

1. Introduction

Spring is widely used in building industrial applications as it makes the development process easier and there are a wide number of Spring Projects to support various infrastructure needs of the application. spring-amqp and spring-rabbit are examples of Spring Projects that support development of the messaging layer for an application using AMQP and RabbitMQ respectively. The immediate concerns in developing such a huge project are maintainability and extensibility. At the same time, there will be a lot of repetitive boilerplate code similar to the ones in other messaging libraries for handling recurring situations like handling connections, handling message delivery and failures etc. Design patterns will be beneficial to use in both these situations, as they provide eloquent solutions to recurring .problems. In this paper, the codebase of both the libraries is studied to identify any design pattern implementation and contexts where a design pattern might be useful to implement and the findings from this study are listed here.

2. Literature Review

To understand the application of the patterns better, it will be helpful to have some context about the libraries.

2.1 What is AMQP?

AMQP i.e. the Advanced Message Queuing Protocol is an open standard for Message Oriented Middleware (MOM) communication [1]. It enables a wide range of applications to connect with each other irrespective of their internal design or choice of programming language.

2.2 Main AMQP components

An AMQP messaging system has three main components: Publisher(s), Consumer(s) and Broker/Server(s) [2]. A broker contains of exchanges which consists of bindings to queues. Publishers publish messages to exchanges. Consumers read messages from queues.

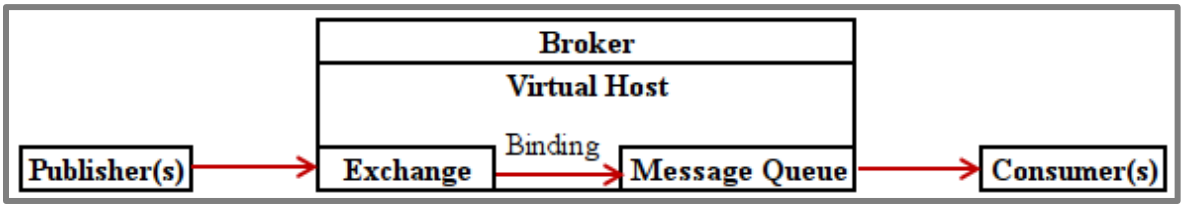


Figure 1: AMQP Architecture [2]

2.3 What is RabbitMQ?

RabbitMQ is an open source message broker that supports the 0-9-1 version of the AMQP protocol [1] by Pivotal. They provide clients for RabbitMQ in multiple programming languages.

2.4 What is spring-amqp?

It's a spring framework to support development of AMQP-based messaging layer in JAVA. On top of the existing AMQP components, it introduces abstractions of its own like connection

factory, messaging template, listener container to facilitate better management of the AMQP resources and enable message driven program execution. This is an abstract library and can't be used on its own.

2.5 What is spring-rabbit?

It's the RabbitMQ implementation of the spring-amqp abstract library. It uses the JAVA client provided by RabbitMQ to create the actual connection to the broker, but has its own management layer to manage the connection and channel objects. A ConnectionFactory is a convenience class to create a Connection object which is an abstraction of the actual socket connection. However, any operation on the broker is carried out against the Channel object.

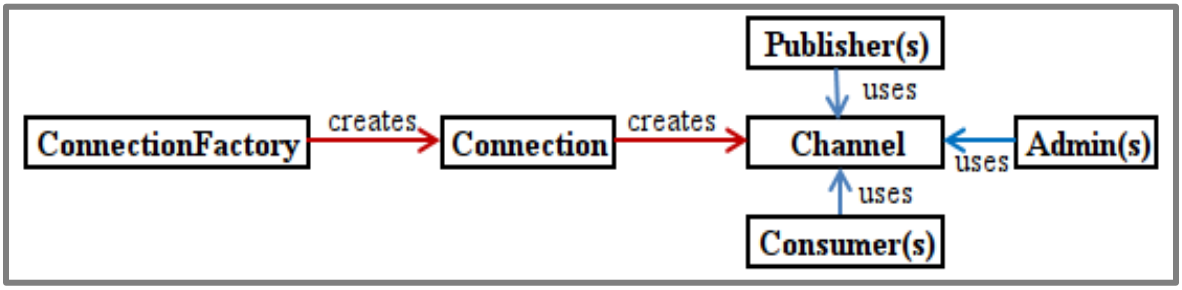


Figure 2: RabbitMQ JAVA Client abstractions

3. Design patterns

Although to explain working of these libraries is out of scope of this paper, a concise explanation is provided for the classes involved in the implementation of the pattern.

3.1 Template

The Template pattern encapsulates an algorithm wherein a template method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Considering the fact that these libraries facilitate message sending through a broker, as expected there is a significant amount of boilerplate code to create and manage connections as well as pre-sending and post-receiving operations on messages. At multiple places, the template pattern has been used to abstract the general procedure and delegate any specific implementation through inheritance.

One such example involves the creation of a listener container using a listener container factory. Currently in the code there are two types of listener containers and their corresponding listener container factories. However, the procedure by which a listener container factory creates a listener container can be abstracted into steps, which is done in the template method – createContainerInstance in the abstract class AbstractRabbitListenerContainerFactory. The step that varies according to the type of the listener container is represented by the abstract method createContainerInstance. Additionally, template design patterns allows for hook methods, which are methods that the abstract class provide a default implementation for, but if the subclass

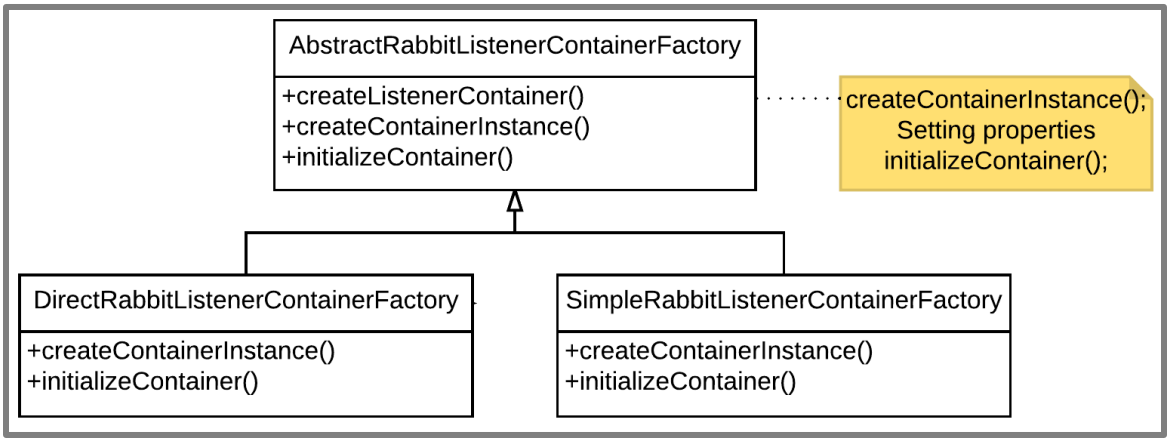


Figure 3: Template design pattern

wants, it can provide its own implementation of the hook method. In this example, the hook method is initializeContainer which is responsible for any operations that ought to be done immediately after the container is initialized. Its implementation in the abstract class has an empty body but is implemented by both the subclasses DirectRabbitListenerContainerFactory and SimpleRabbitListenerContainerFactory.

3.2 Strategy

Strategy pattern enables encapsulation of a family of algorithms and allows them to be interchangeable even at runtime. More often than not, when the coding principle – "Program to an interface and not a concrete implementation" is applied, and the interface has more than one concrete implementation, it can be said that strategy pattern has been applied. As a result of this, because Spring libraries give context specific solutions which involves orchestrating multiple classes of the same interface, Spring libraries are said to have multiple occurrences of the Strategy pattern. However, it's important to then consider the intent of the pattern, which is to encapsulate a family of algorithms to decide whether the pattern has been applied deliberately or not.

One such example is the class RabbitTemplate which is primarily concerned with sending and receiving messages. To achieve this, it uses a message conversion algorithm to extract the payload from the received message and convert it to POJO as well as the other way round. Currently the library has four different message conversion algorithms. To enable the client to select the message conversion algorithms and to switch between them, they share a common interface MessageConverter and all the methods in the class RabbitTemplate are programmed to this interface. The class also uses a message post processor algorithm to enable processing on the message after it has been processed by the MessageConverter object. There are three different types of message post processing algorithms as well, all of which share the interface MessagePostProcessor.

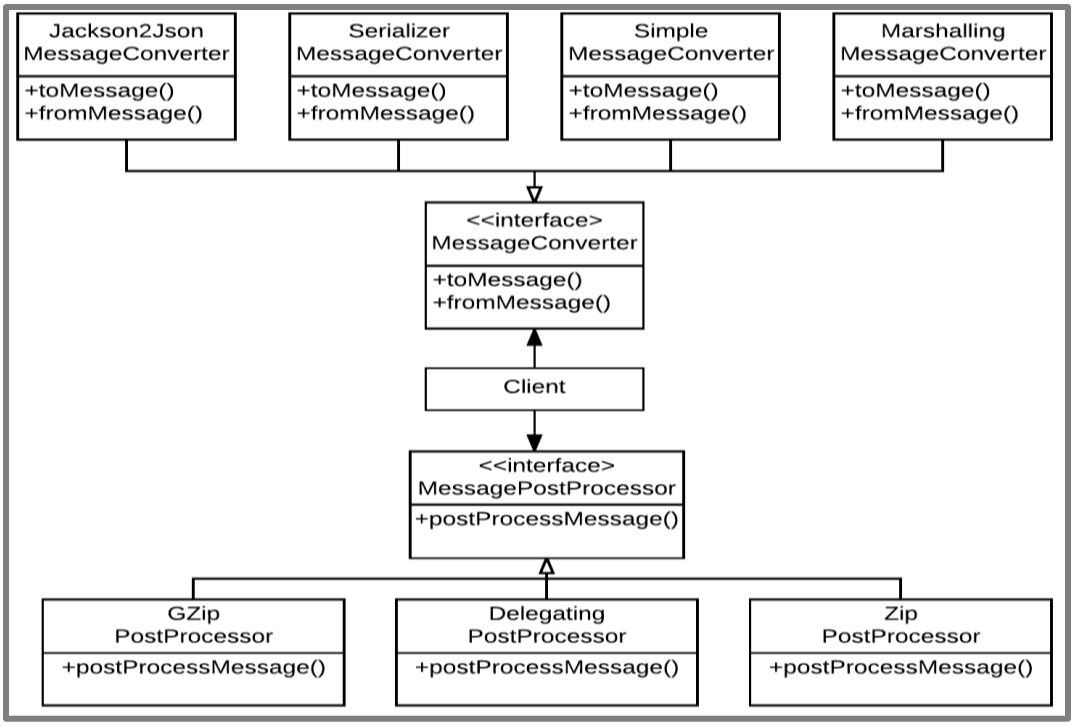


Figure 4: Strategy Design Pattern

An example of how context forces the pattern to change can be seen in the fact that RabbitTemplate doesn't hold a declaration of MessagePostProcessor interface but an array list of MessagePostProcessor objects. This enables the client to use not just one message post processing strategy, but multiple strategies or even all of them. This decision can be made at runtime by the client. Also for both these strategy interfaces, RabbitTemplate has a default strategy which is chosen at the time of instantiation, thereby not forcing the client to choose one.

3.3 Observer

The Observer pattern allows for multiple listeners to subscribe to updates from an object. When the state of that object changes, an update is published to all the listeners. All the listeners are independent and the set of actions taken after the reception of the update message is isolated.

In the spring library, the object that needs to be observed is an instance of Connection class. Any class wanting to subscribe to updates from the Connection class needs to implement the ConnectionListener interface. According to this interface, the updates are sent only in three situations – when the connection is established, closed or closed forcibly.

A few considerations need to be made based on the context while implementing the Observer pattern – where to store the list of observers, when to send notifications about the change – on every change or at the client's discretion?, what information should be sent along with the notification? In this case, the class AbstractConnectionFacotry - the client for Connection object contains methods to add or remove listeners (observers) as well as holds reference to the class CompositeConnectionListener which holds the list of connection listeners (observers). Also in

this case, the client decides when to send notifications to the observers. Every client application is free to implement the ConnectionListener interface and subscribe to the Connection object. However, it can't be predicted what information these observers might need about the Connection object, and hence instead of using the pull model, the push model of sending data is used wherein the reference to the connection object is passed to the observers.

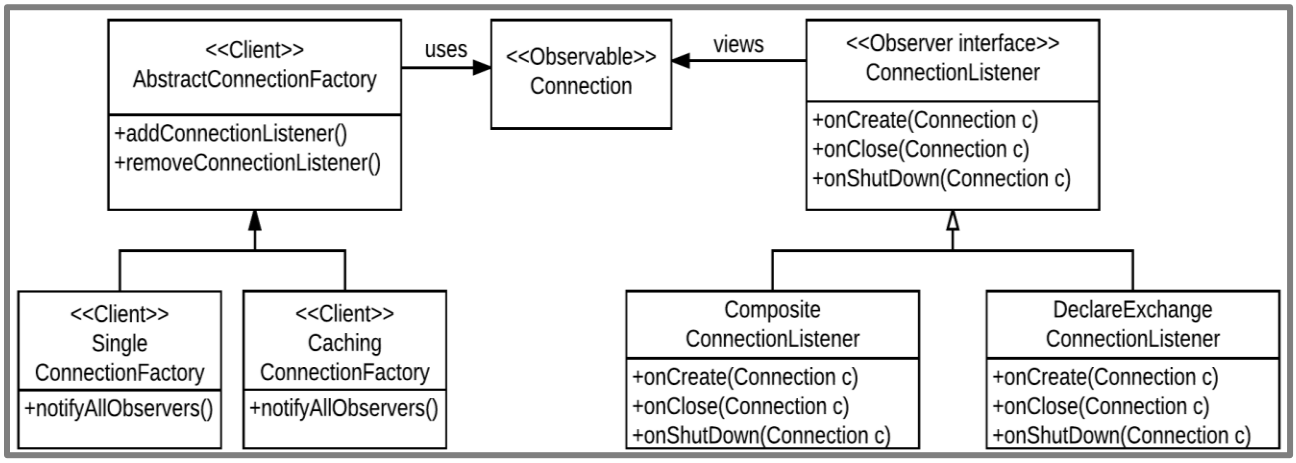


Figure 5: Observer design pattern

The AbstractConnectionFactory as the name implies is an abstract class and has concrete subclasses. Currently, the update notifications to the observers are sent from these concrete subclasses directly. However, to make the design more cohesive, the methods to send notifications can be defined in the abstract class and the concrete subclasses can then invoke them as and when required.

3.4 Factory

Factory pattern is a creational pattern that uses inheritance. It delegates class instantiation to subclasses. All of Spring libraries make heavy use of factory pattern for generating factory beans [3]. However, there are other occurrences too in the AMQP libraries.

Factory pattern is ideal in requirements involving parallel class hierarchies. This exists between the interfaces MessageListenerContainer and RabbitListenerContainerFactory. As the name implies, RabbitListenerContainerFactory is responsible for instantiation of a MessageListenerContainer. An abstract class AbstractRabbitListenerContainerFactory is defined that holds a reference to AbstractMessageListenerContainer. Deciding which listener container to instantiate is encapsulated in the factory method – createContainerInstance and is declared abstract.

The subclasses DirectRabbitListenerContainerFactory, SimpleRabbitListenerContainerFactory instantiate subclasses of AbstractMessageListenerContainer - DirectMessageListenerContainer and SimpleMessageListenerContainer respectively. The creator class does not provide any default implementation of the factory method.

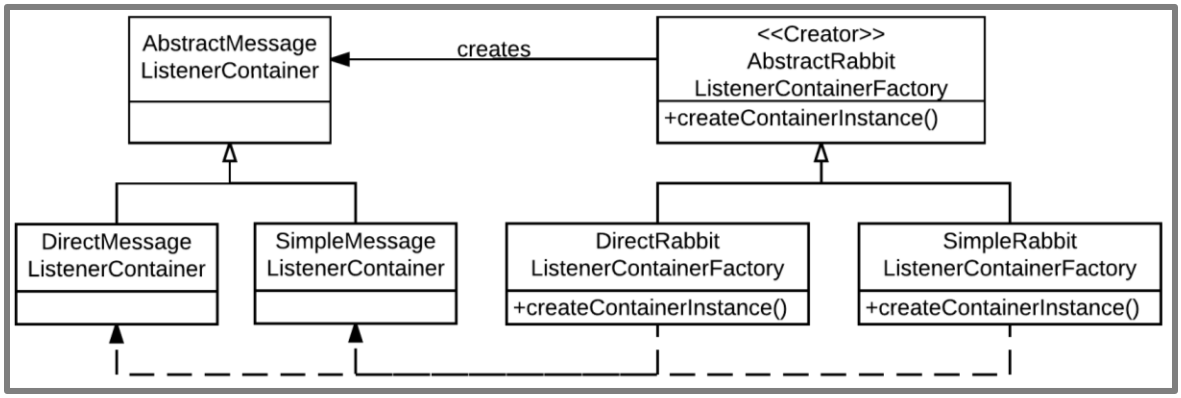


Figure 6: Factory design pattern

3.5 State

Implementations of this pattern were not found in the code. The possibility of using it with Connection or Channel classes was investigated as they are analogous to TCPConnection a popular use case for explaining the State design pattern. These objects have three states - 'Open', 'Closed' and 'Shutdown' states. However, there is no commonality between these states that can be abstracted. The 'Closed' or 'Shutdown' states merely represent that an operation can't be carried out when the Connection or Channel is in any of these states. However, one advantage to using the state pattern here is that connection/channel objects can be cached. When the client calls close on the connection/channel object, instead of actually closing the connection, it can be moved to the 'Closed' state allowing the reuse of the connection/channel object when the client requests to create another connection/channel.

3.6 Builder

Builder pattern allows for flexible construction of objects in steps without requiring multiple constructors with different number of arguments. There are many occurrences of the Builder pattern in the code.

An exchange and a queue are entities with a high number of properties most of which have default values and unless required by the client, don't need to be changed. The Builder pattern is ideal for both exchange and queue classes. An AbstractBuilder class is used to encapsulate the functionality that a client can pass in a map of property names and values to build the object (Exchange or Queue).

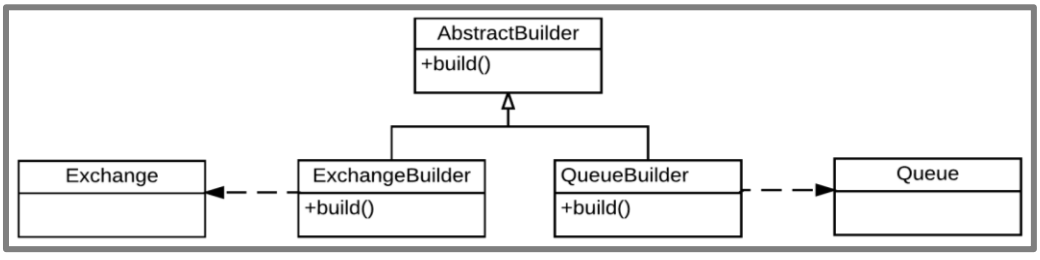


Figure 7: Builder design pattern

Its subclasses ExchangeBuilder and QueueBuilder declare the exchange specific and queue specific methods respectively as well as the build method which generates an Exchange or a Queue. This builder is used by the class RabbitAdmin, as it needs to declare these elements at startup.

Additionally, the Builder pattern will be advantageous, if used for the class AMQPEvent. Currently there are seven subclasses of AMQPEvent- three for describing start, close and shutdown events for the class AsyncConsumer and another similar three for ListenerContainer.

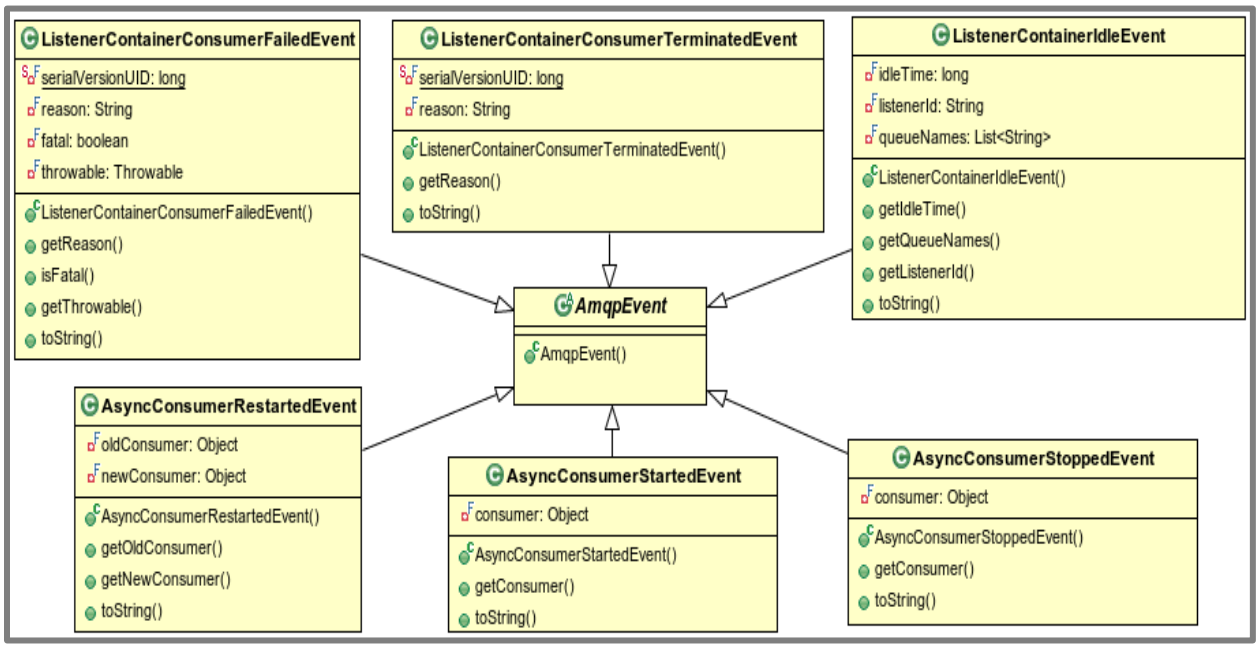


Figure 8: Current AMQPEvent subclasses structure

In future, there might be more types of AMQPEvents like admin operations. Instead of having multiple subclasses to describe each event, the class AMQPEvent can be modeled to have a string variable that gives the description of the source of the event, a throwable object if the event is caused by an exception and a Map<String,Object> which would store any extra objects required for a client to consume the event. As the variables are all optional, an AMQPEventBuilder class can be used here to create the AMQPEvent. This is similar to how a Response object is built for handling HTTP requests in web services. Fig. 8 represents the current class structure and Fig. 9 the proposed one.

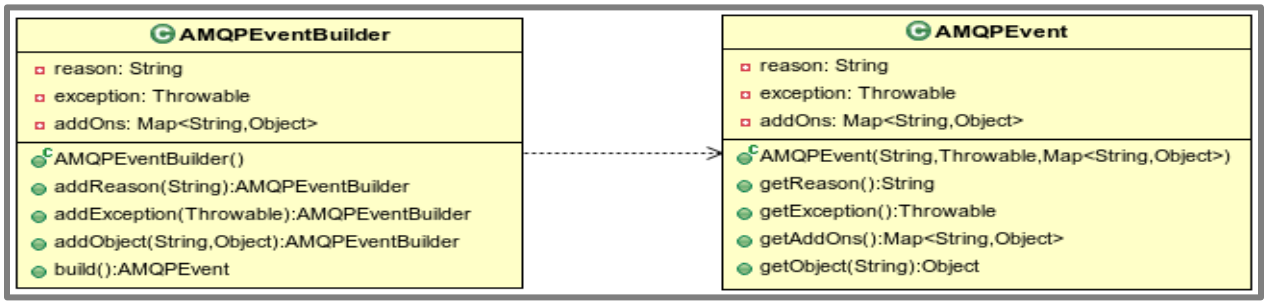


Figure 9: Proposed AMQPEvent subclasses structure

3.7 Adapter

The Adapter pattern allows two classes with incompatible interfaces to work together, as long as both the classes perform similar functions. Now any application using spring-amqp and spring-rabbit libraries to develop its messaging layer will use the ConnectionFactory interface provided by the spring library, but actual connections to a RabbitMQ broker can be created only using the ConnectionFactory class provided by the RabbitMQ JAVA client. However, the RabbitMQ JAVA library has only one implementation of ConnectionFactory without any super type (interface or abstract class). The Spring libraries on the other hand have provisioned multiple types of connection factories suitable for use in different contexts so that the client is prevented from writing a lot of infrastructure management code. To this end, an adapter class is needed between the Spring's interface ConnectionFactory and the RabbitMQ JAVA client's class ConnectionFactory. In this case, AbstractConnectionFactory acts as the adapter class and is composed of the Adaptee – ConnectionFactory class. All the methods in the ConnectionFactory class are present in AbstractConnectionFactory, and for most of them, the method call is simply delegated to the adaptee object, with exceptions for creating connections and destroying them, where additional processing is done prior to delegation.

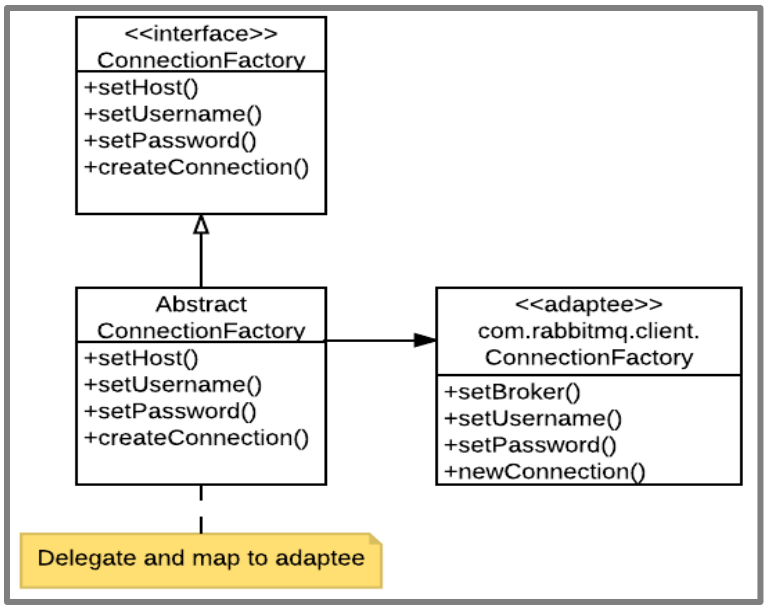


Figure 10: Adapter design pattern

3. 8 Decorator

The Decorator pattern allows adding behavior to classes at runtime using object composition. The use of this pattern is seen in code for adding behavior to the MessageConverter interface. Its basic subclasses are AbstractMessageConverter, ContentTypeDelgatingMessageConverter and two decorator classes MessagingMessageConverter, RemoteInvocationAwareMessageConverter. The decorators don't share a common interface as their functionalities are widely different. The decorator MessagingMessageConverter adds the functionality of mapping AMQP Headers after the process of message conversion, which it does using the instance of MessageConverter that was passed to it. If no instance was passed, it uses an instance of the default message converter – SimpleMessageConverter. This is a slight modification of the decorator pattern and illustrates

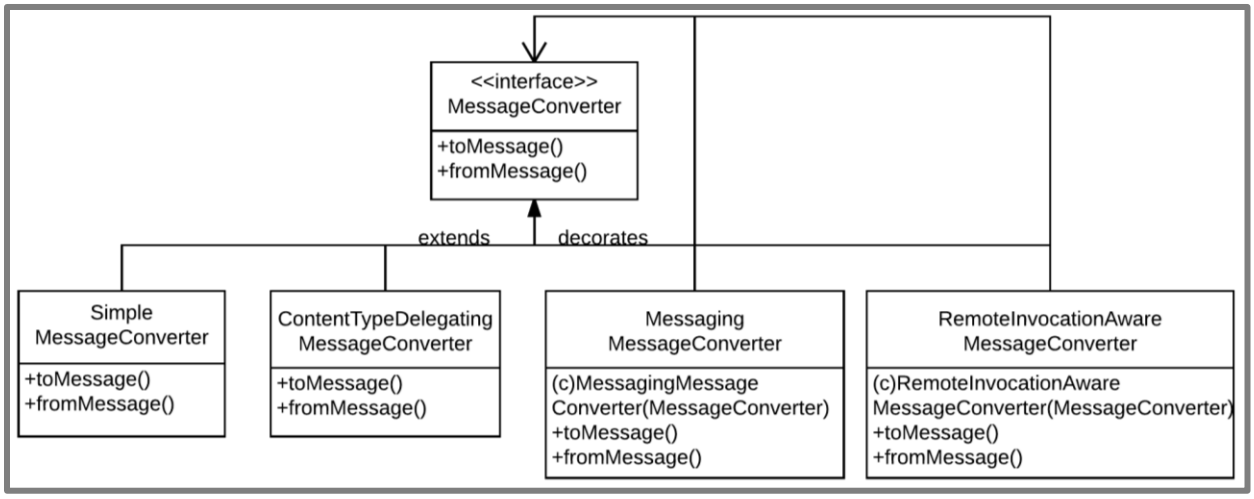
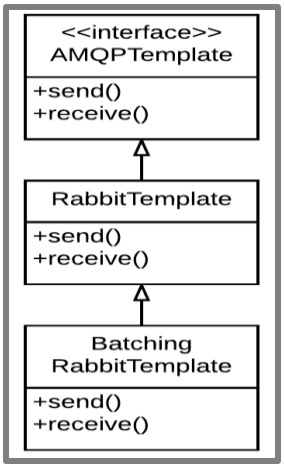


Figure 11: Decorator design pattern

how patterns are never applied in the exact same way everywhere, they adapt to the context. The other decorator wraps the outbound results from the message converter in a RMI context.



A context where applying decorator pattern could be beneficial is with AMQP Template, RabbitTemplate and BatchingRabbitTemplate. Currently BatchingRabbitTemplate is tightly bound to RabbitTemplate. However, if it were to be converted to a decorator object, batching could be applied to any AMQP template object, but as there is only concrete subclass of AMQP Template, it might seem like overkill. Although in future, if AMQPTemplate will have more concrete subclasses, the decorator pattern would be useful to apply here. Fig. 12 represents the current state and Fig.13 the proposed state.

Figure 12: Current BatchingTemplate structure

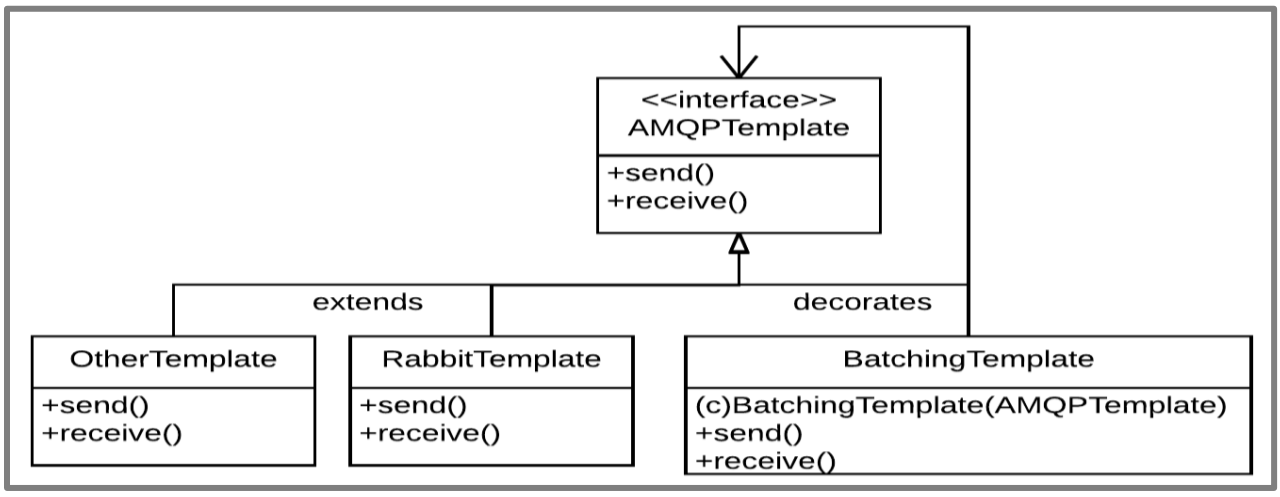


Figure 13: Proposed BatchingTemplate structure

3.9 Proxy

The interfaces ConnectionProxy or ChannelProxy could have implemented the proxy design pattern as they act as proxy for Connection and Channel interfaces. However, one of the intents of the Proxy design pattern is to restrict access to their real counterparts, and these interfaces allow direct access to the real Connection and Channel object. The other functionalities of these interfaces are decorating the real Connection and Channel objects, as seen in the section about Decorator pattern. Hence, it can be concluded that proxy pattern isn't being used here.

3.10 Composite

The composite pattern can be used to represent part-whole hierarchies and let clients deal with individual objects and composite objects in the same way.In the spring library, the class ContentTypeDelegatingMessageConverter implements the MessageConverter interface and is composed of multiple Message Converter objects. However, the composite pattern applied here is slightly modified to fit the context. Any operation against an instance of the ContentTypeDelegatingMessageConverter is executed against only one of its

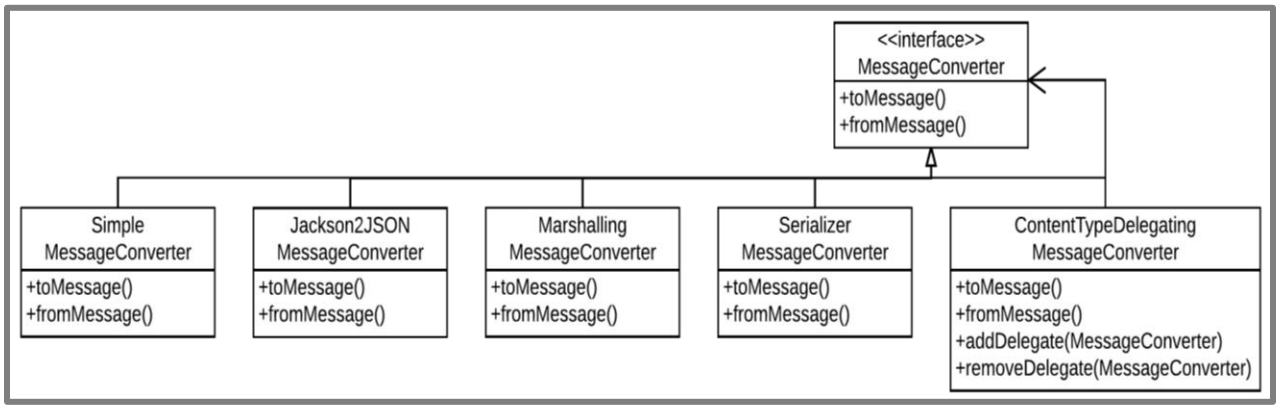


Figure 14: Composite Design Pattern

child-nodes as opposed to all of them, but this is an implementation detail; to the client any MessageConverter object and ContentTypeDelegatingMessageConverter appear the same and the latter is composed of multiple instances of MessageConverter. The child-node management methods and the list of child nodes are placed in the composite class. At the time of instantiation, if no MessageConverter instances are passed, by default it has one child node – instance of the class SimpleMessageConverter.

3.11 Façade

The Façade pattern defines a simplified interface called as the façade interface, that orchestrates multiple other interfaces which are together responsible for the working of a sub-system.The class RabbitTemplate is responsible for sending and receiving of messages. However, the implementation of the multiple overloaded send and receive methods contains complicated code that involves invoking operations on a number of interfaces that together make up the messaging

sub-system. The interfaces ConnectionFactory, Connection and Channel are used to obtain a channel to execute the operation. The interfaces Message, MessageProperties, MessageConverter and MessageListenerContainer are used in building a message and receiving a reply. However, the client need not bother with the underlying orchestration. It can directly invoke the send/receive methods of the RabbitTemplate class.

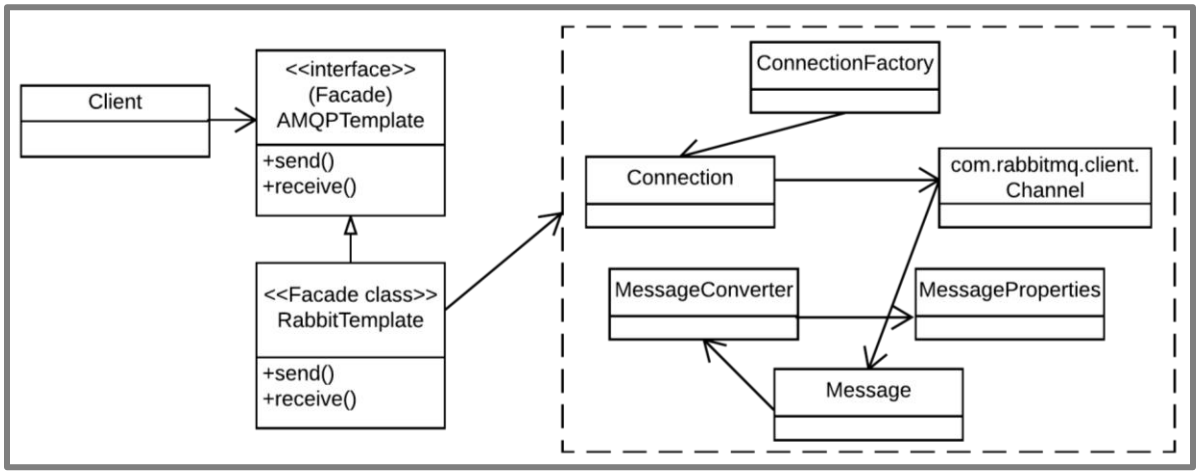


Figure 15: Facade design pattern

RabbitTemplate is a concrete class which implements the interface AMQPTemplate. So AMQPTemplate is the façade interface and RabbitTemplate is the Façade class.

4. Conclusion

For a giant project like Spring that is widely used, maintainability, modification and extension of the codebase is crucial. Design patterns show how to build systems with good object oriented design qualities which help achieve these goals. The fact that so many implementations of design patterns were found in the code base reaffirms this fact. This study exercise also helps understand how all the different design patterns work together and how certain problems require a combination of design patterns to be resolved. The findings also reiterate certain assertions made about design patterns by the Gang of Four [4] in their book namely i) large systems nearly use all of the design patterns. ii) Design pattern is just a template and is often modified according to the context.

5. References

[1] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues and S. Ullah, "Performance evaluation of RESTful web services and AMQP protocol," 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), Da Nang, 2013, pp. 810-815.

[2] H. Subramoni, G. Marsh, S. Narravula, Ping Lai and D. K. Panda, "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand," 2008 Workshop on High Performance Computational Finance, Austin, TX, 2008, pp. 1-8.

[3]J.Long, "What's a FactoryBean?", Spring.io, 2017. [Online]. Available: https://spring.io/blog/2011/08/09/what-s-a-factorybean
[4] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, “Design patterns: Elements of reusable object-oriented software,” Reading: Addison-Wesley, p. 4

APPENDIX

Listed here are additional examples of design patterns implementation in the spring codebase, which couldn’t be part of the main paper due to length constraints.

1. Adapter

The class SimpleConnection acts as an adapter class between Spring’s interface Connection and the interface Connection in the RabbitMQ JAVA client library. It is composed of the latter to which, it delegates its method calls to with limited or no prior processing. The motivation in this case too remains similar to Spring requiring its own abstractions of the Connection object which are quite different from those present in the RabbitMQ JAVA client library.

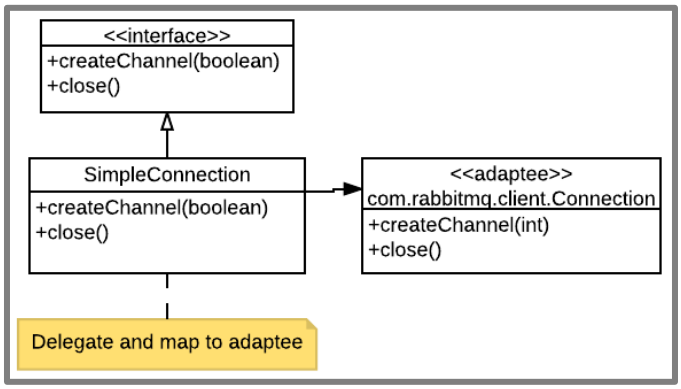


Figure 1: Adapter design pattern (2nd)

2. Factory pattern

A parallel class hierarchy exists between ConnectionFactory and ConnectionProxy. The factory method – createConnection in the AbstractConnectionFactory class creates an instance of the interface ConnectionProxy. SingleConnectionFactory and CachingConnectionFactory extend AbstractConnectionFactory to produce SharedConnectionProxy and ChannelCachingConnectionProxy respectively – both implementations of ConnectionProxy.

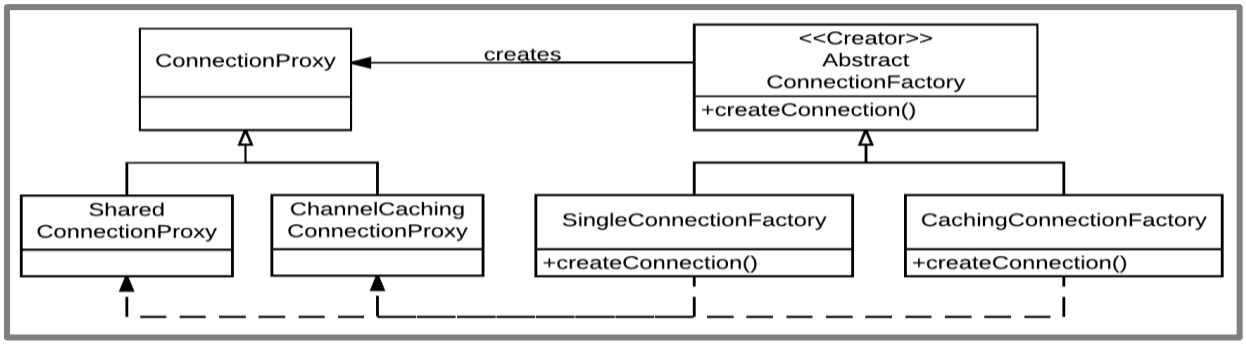


Figure 2: Factory design pattern (2nd)

3. Decorator

The Connection interface has a simple implementation – SimpleConnection and two decorator implementations SharedConnectionProxy and ChannelCachingConnectionProxy which take in a Connection object and add extra behaviors of sharing and sharing within a pool respectively. It might be argued that this is actually a case of proxy pattern, but proxy is nothing but decorator pattern in disguise and as there is no instantiation or access control happening here, it is the implementation of decorator pattern.

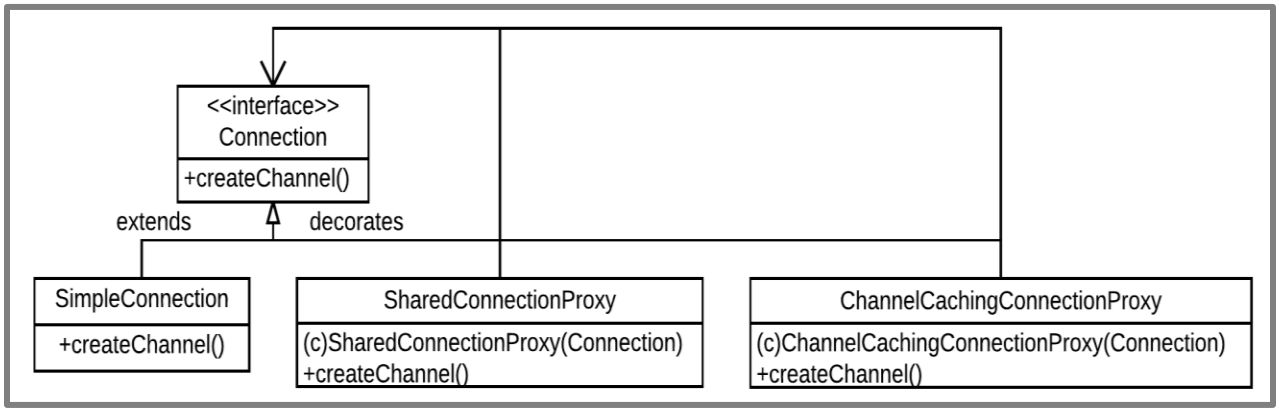


Figure 3: Decorator design pattern (2nd)

4. Composite

The class DelegatingDecompressorPostProcessor implements the MessagePostProcessor interface and is composed of multiple MessagePostProcessor instances.

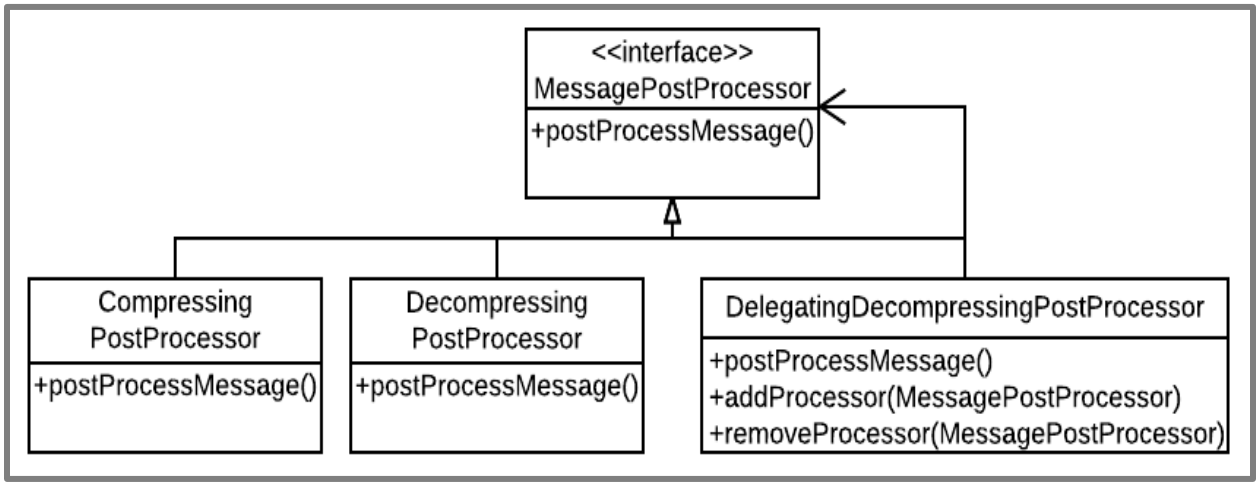


Figure 4: Composite design pattern (2nd)

The classes CompositeChannelListener and CompositeConnectionListener are examples of usage of Composite pattern, as they are composed of multiple child-nodes which are instances of 'Channel Listener' and 'Connection Listener' respectively and any method invoked against an

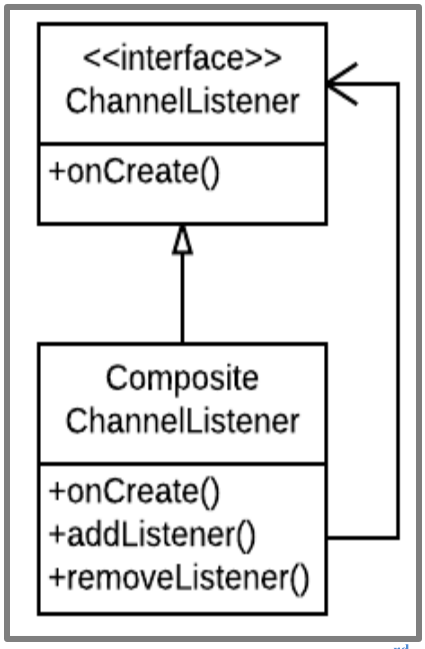


Figure 5: Composite design pattern (3rd)

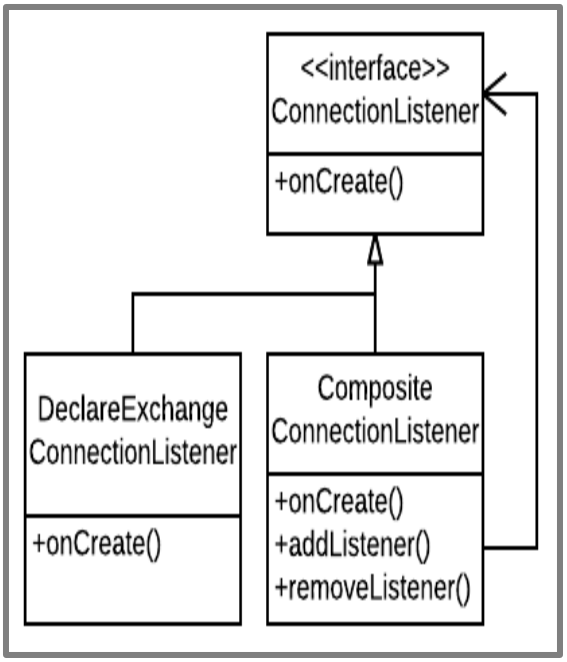


Figure 6: Composite design pattern (4th)

instance of 'CompositeChannelListener' or 'CompositeConnectionListener' is invoked on all the child-nodes of the respective class.