Publish-Subscribe System
Requirements Model

Version:	1.0
Print Date:	February 25, 2019
Release Date:	
Release State:	Initial/Core/Final
Approval State:	Draft/Approved
Approved by:	
Prepared by:	Fahreen Bushra
Reviewed by:	
Path Name:	
File Name:	T3-ReqmtsModel.doc
Document No:	

Document Change Control

Version	Date	Authors	Summary of Changes

Contents

1 Introduction

1.1 Purpose

This document details the requirements of the system *Publish-Subscribe System (pub/sub system)*.

1.2 Overview

The project provides an elaborate outline aimed towards managing the publish-subscribe system. The channel will serve as an abstraction communication medium between the publisher and subscriber. Publishers may post events onto the channel, and the channel will notify subscribers. The channel will regulate the link between subscribers and publishers, by maintaining subscriber access. The aim of this project is to implement a prototype server system that enables a publish-subscribe system between three different actors (“Publisher”,“Subscriber”, & “Channel”).

1.3 References:

Pub/Sub architecture style:
https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
https://docs.oracle.com/cd/B10501_01/appdev.920/a96590/adg15pub.htm

System Diagrams and Descriptions:

http://agilemodeling.com/artifacts/classDiagram.htm

https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/

http://www.gatherspace.com/software-requirement-specifications/

2 Business Scenario Model

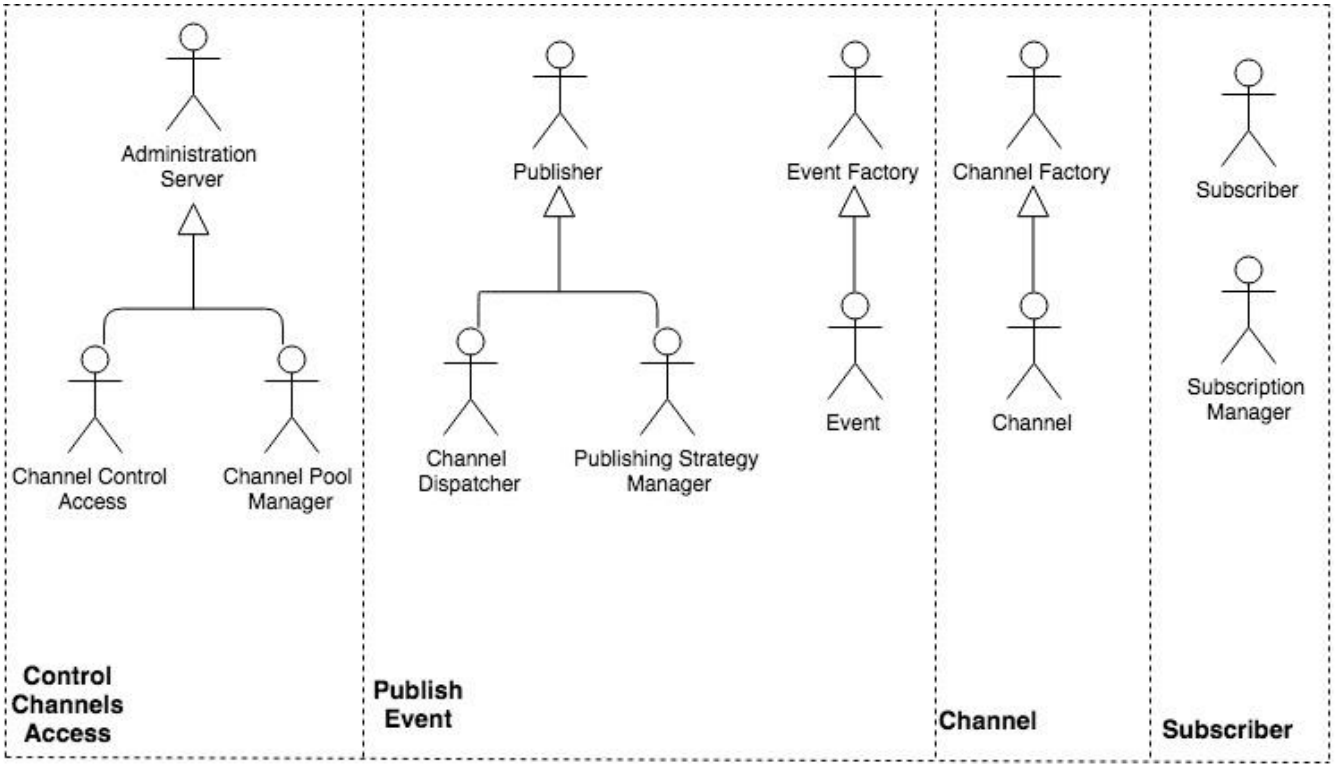
2.1 Actors

2.1.1 Overview

The actors in the Publish-Subscribe system include publishers,subscribers and the Publish-Subscribe server. Publishers post events to one or more channels based on a strategy or to a default strategy if no specific strategy exists. Subscribers handle events published on a channel depending on their state. The Pub/Sub Server acts as a communication medium between the publisher and subscriber and notifies the subscriber if a publisher has posted an event.

2.1.2 Actor Diagram

The figure below represents the actors in the system. Based on their interactions with the system, the actors are characterized into four categories: Control Channels Access, Publisher of Events, the Channel and the Subscribers.



2.1.3 Actor Definitions

2.1.3.1 Administration Server

Description	An entity that exists to selects subscribers to block or unblock them from a specified channel.
Aliases	Administrator
Inherits	None
Actor Type	Active - Person
Contact Person	
Contact Details	

2.1.3.2 Channel Control Access

Description	An entity that manages and modifies a subscribers access to a channel
Aliases	None.
Inherits	Administration server
Actor Type	Passive - External System
Contact Person	
Contact Details	

2.1.3.3 Channel Pool Manager

Description	An entity that checks the existence of a subscriber by obtaining a list of available channels and their subscribers
Aliases	None
Inherits	None

Actor Type	Passive - Person
Contact Person	
Contact Details	

2.1.3.4 Publisher

Description	An entity that publishes/posts events to one or more channels based on strategy or to a default channel if no specific strategy is selected
Aliases	None.
Inherits	None
Actor Type	Active - Person
Contact Person	
Contact Details	

2.1.3.5 Channel Dispatcher

Description	An entity that posts specific events to specific channels using the channel interface
Aliases	Channel Event Dispatcher
Inherits	Publisher
Actor Type	Passive - Person
Contact Person	
Contact Details	

2.1.3.6 Publishing Strategy Manager

Description	Determines the mechanisms in how a specific event will be published on the selected channel(s)
Aliases	None

Inherits	Publisher
Actor Type	Passive - Person
Contact Person	
Contact Details	

2.1.3.7 Event Factory

Description	An entity that creates an event by creating a new instance of an event
Aliases	None
Inherits	None
Actor Type	Passive - External System
Contact Person	
Contact Details	

2.1.3.8 Event

Description	An entity that denotes a piece of information posted on a chanel. An event consists of an event ID, a reference to its publisher and a payload - an *EventMessage* which has a header and a body
Aliases	None
Inherits	None
Actor Type	Passive - External System
Contact Person	
Contact Details	

2.1.3.9 Channel Factory

Description	An entity that creates a new channel by creating a new instance of a channel
Aliases	None.
Inherits	None.
Actor Type	Passive - External System
Contact Person	
Contact Details	

2.1.3.10 Channel

Description	An entity that denotes an abstraction of a communication medium. It maintains a list of subscribers and a queue of events. They notify subscribers once an event has been posted on a channel. There may be more than one channel on a system.
Aliases	None
Inherits	Channel Factory
Actor Type	Active - External System
Contact Person	
Contact Details	

2.1.3.11 Subscriber

Description	An entity who subscribes to one or more channels. They handle events published on a channel depending on their “state”
Aliases	None
Inherits	None
Actor Type	Active - Person
Contact Person	

Contact Details	

2.1.3.12 Subscription Manager

Description	An entity that handles the request of a subscriber to subscribe or unsubscribe to a channel
Aliases	None
Inherits	None
Actor Type	Passive - Person
Contact Person	
Contact Details	

2.2 Use Case Descriptions

This section documents the complete business scenarios within the scope of this project.

2.2.1 UC1: Publishing an Event

Description: In this use case, the Publisher posts an event on one or more channels. The mechanism in which the event is published is decided by the Publishing Strategy, associated with the Publisher. There are two types of strategies.

a) the publisher specifies which event to publish and the publishing strategy determines which channel to post it to

b) the publishing strategy generates both the appropriate event and decide to which channel to post it to.

After the event and list of channels have been determined, the Event Factory creates the events and Channel Factory creates any channels that may not exist. The Channel Event Dispatcher finally posts the event on the channels.

Actors:

1. Publisher: An entity that publishes/posts events to one or more channels. A publisher publishes an event to a channel based on a strategy.
2. Publishing Strategy: Determines the mechanism in which an event is published on channel(s). Each publisher is associated with a publishing strategy.
3. Channel Event Dispatcher: Posts the specified event to the specified channel(s), using the channel interface.
4. Event Factory: Creates a new instance of an event
5. Channel Factory: Creates a new instance of a channel

Preconditions:
Before this scenario can be performed:

1. The publisher and the event must exist
2. If the channel the event is about to be published does not exist, then it is created.

Post Conditions:
After this scenario is performed:

1. The specified channel exists (either it is created as part of this event publication, or existed before).
2. There is an event in the queue of the specified channel

Scenario Text:

1. Get Event
    a. Publisher specifies Event
    b. Event is created by Event Factory
2. Get Channel(s)
    a. Publishing Strategy Manager specifies one or more channels, and collects existing channels
    b. Channel Factory creates any channel specified by Publishing Strategy that does not exist
3. Post Event
    a. Channel Dispatcher posts event in all specified channels

Alternative Courses:
1.a Publishing Strategy specifies Event

Extends:
None.

User Interfaces:
Channel Interface is used by Channel Dispatcher to post events

Constraints:
Publisher is not associated with a Strategy, in that case the event is published using a predefined default strategy

Questions:
None.

Notes:
None.

Source Documents:
http://agilemodeling.com/essays/umlDiagrams.htm
http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

2.2.2 UC2 Channel Notifying Subscribers of Event Been Published

This scenario deals with the situation where an event has just been posted to a channel and the subscribers of the channel need to be notified.

Description:
Every channel has a corresponding topic, associates with a list of subscribers, and has a queue keeping a list of events posted to this channel. Upon addition of an event to the channel queue, the channel makes its subscribers aware of the new event posted, a subscriber is notified of the newly posted event if it is not blocked on this channel.

Actors:
1. Channel: An entity that denotes an abstraction of a communication medium. It maintains a list of subscribers and a queue of events. Once an event is added to the queue, the channel notifies its subscribers. There may be more than one channel in the system.
2. Subscribers: An entity that handles events published on a channel. A subscriber is subscribing to one or more channels. A subscriber has a "state" which determines how it handles an event.

Preconditions:
Before this scenario can be performed:
1. Event must be posted on a channel
2. The channel has subscribers
3. The channel checks that the subscribers have not been blocked

Post Conditions:
1.All subscribers on this channel which are not blocked are notified

Scenario Text:
1. Use *Publishing an Event*
2. Notify subscribers of an event that has been published
    1. Get list of subscribers
    2. Notify subscribers who are not blocked

Alternative Courses:
If a subscriber is blocked, maintain a list of blocked subscribers using a map or another data structure.

Extends:
None.

User Interfaces:
None.

Constraints:
Subscriber is blocked on the channel, in that case, cannot proceed to notify subscriber. Add them to a list of blocked subscribers.

Questions:
None.

Notes:
None.

Source Documents:
http://agilemodeling.com/essays/umlDiagrams.htm

http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

2.2.3 UC3: Subscriber Handling Events

In this project scenario, one subscriber will be notified by the system (channel) about an event that has been posted. The channel takes into account the subscriber's state to determine how to the subscriber deals with the event.

Description:

The scenario deals with how a subscriber handles an event once an event is posted by the publisher. The subscriber, who must be interconnected with the said channel through subscription prior to receiving the notification, will have a state associated with them which consists of a specific ID. By using this unique ID, the subscriber will handle the event according to the logic that is mandated by the state.

Actors:

The follows actors are associated with this scenario:

1. Subscriber: a member (entity) of a channel who has access to the events of such channel and can handle them accordingly. The subscriber is identified by their unique ID, as well as a state ID.

2. Publisher: An entity that publishes/posts events to one or more channels. A publisher publishes an event to a channel based on a strategy, or to a default channel(s) if the no specific strategy is selected.

Preconditions:

Prior to the scenario being performed:

1. The subscriber must exist as an entity. If they do not exist, then the channel cannot reach said entity.
2. The subscriber must be accountable for a state ID. This state ID determines how an event can be handled.
3. The subscriber must be subscribed to the channel which the event is posted in. If they are not, the channel will not notify them of the posted event.

Post Conditions:

After the preconditions are met, the following will occur:

1. The subscriber handles the event according to the logic dictated by its state.

Scenario Text:

1. Use *Publishing an Event*
   1.1. Perform publication via publishing strategy
   1.2. Generate appropriate event
   1.3. Queue event to correct channel
2. Notify Subscribers of Event
   2.1. Check to see if subscriber is blocked on channel
   2.2. Visit specific topic of channel and notify all subscribers of event
3. Determine State of Subscriber
   3.1. Call configuration file *states.sts* to learn subscriber's state
4. Handle Event
   4.1. Subscriber handles event according to their state

Alternative Courses:

None.

Extends:

None.

User Interfaces:

The file *states.sts,* which consists of a compilation of tuples of the form <subscriber-ID, state-ID>, is implemented of type *int* which determines the subscriber's unique personal ID as well as state ID. This allows the channel to recognize how a subscriber will handle a particular event.

Constraints:

None.

Questions:

None.

Notes:

None.

Source Documents:

http://agilemodeling.com/essays/umlDiagrams.htm

http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

2.2.4 UC4 Controlling access to a channel

In this scenario, the channel's Access Control Module will control a subscriber's access to a channel.

Description:

This scenario deals with controlling the subscriber's access to the channel. The Channel Access Control Module will provide the service to block or unblock a subscriber, and will also provide the service to check whether a subscriber is blocked from a specific channel or not. The decision to block or unblock a subscriber will come from the administrator.

Actors:

The follows actors are associated with this scenario:

1. Channel: A shared medium that subscribers have access to. This server is responsible for deciding whether to block or unblock a subscriber. It has references to the Channel Access Control Module and the Channel Pool Manager Module.

Preconditions:

Prior to the scenario being performed:

1. The channel must exist.

Post Conditions:

After the preconditions are met, the following will occur:

1. Subscriber must be added in the list of blocked subscribers of the specified channel (blocking case) OR,
2. Subscriber must be removed from the list of blocked subscribers of the specified channel (un-blocking case).

Scenario Text:

1. Select to block a subscriber.
   1.1. Obtain list of all available channels and their subscribers via Channel Pool Manager Module.
   1.2. Check if subscriber is blocked on the channel via Channel Access Control Module.
      1.2.1. Return null if subscriber is blocked.
      1.2.2. Block subscriber using Channel Access Control Module if subscriber exists and is not currently blocked.
2. Alternative scenario - unblock a subscriber (referenced below)

Alternative Courses:

1. Select to unblock a subscriber.
   1.1. Obtain list of all available channels and their subscribers via Channel Pool Manager Module.

1.2. Check if subscriber is blocked on the channel via Channel Access Control Module.
1.2.1. Unblock subscriber using Channel Access Control Module if subscriber exists and is currently blocked.

Extends:

None.

User Interfaces:

None.

Constraints:

None.

Questions:

None.

Notes:

None.

Source Documents:

http://agilemodeling.com/essays/umlDiagrams.htm

http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

2.2.5 UC5 Subscriber Subscribes to a Channel

In this project scenario, a subscriber can subscribe to unsubscribe to a channel.

Description:

The scenario deals primarily with the subscriber. The latter can choose to subscribe or unsubscribe to a channel. Upon choosing whether to subscribe or unsubscribe, a request is sent to the Subscription Manager Module which handles the request accordingly.

Actors:

The follows actors are associated with this scenario:

1. Subscriber: an entity that handles events published on a channel. The subscriber has a "state" for each channel which determines how they handle an event.
2. Subscription Manager: an entity that handles subscription requests from subscribers to subscribe to a channel
3. Channel: A shared medium that subscribers have access to. This server is responsible for deciding whether to block or unblock a subscriber. It has references to the Channel Access Control Module and the Channel Pool Manager Module.

Preconditions:

Prior to the scenario being performed:

1. The only precondition of the use case is that the channel exists. If the channel does not exist, then the Subscription Manager Module will not be able to fulfill the request.

Post Conditions:

After the preconditions are met, the following will occur:

1. The subscriber will be added in the list of subscribers for the given channel.

OR

2. The subscriber will be removed from the list of subscribers for the given channel.

Scenario Text:

1. Call Subscription Manager Module
    1.1.1 Take references for the channel
    1.1.2. Determine if channel exists or not
2. Add subscriber to list of subscribers in channel
3. Alternate scenario to subscribe (referenced below)

Alternative Courses:

1. Call Subscription Manager Module
    1.1.1. Take references for the channel
    1.1.2. Determine if channel exists or not
2. Remove subscriber from list of subscribers in channel

Extends:

None.

User Interfaces:

The subscriber (“ISubscriber”) object will have a reference ID, as well as the abstract channel name. Both are referenced in the functions within the Subscription Manager Module that is called upon.

Constraints:

None.

Questions:

None.

Notes:

None.

Source Documents:

http://agilemodeling.com/essays/umlDiagrams.htm

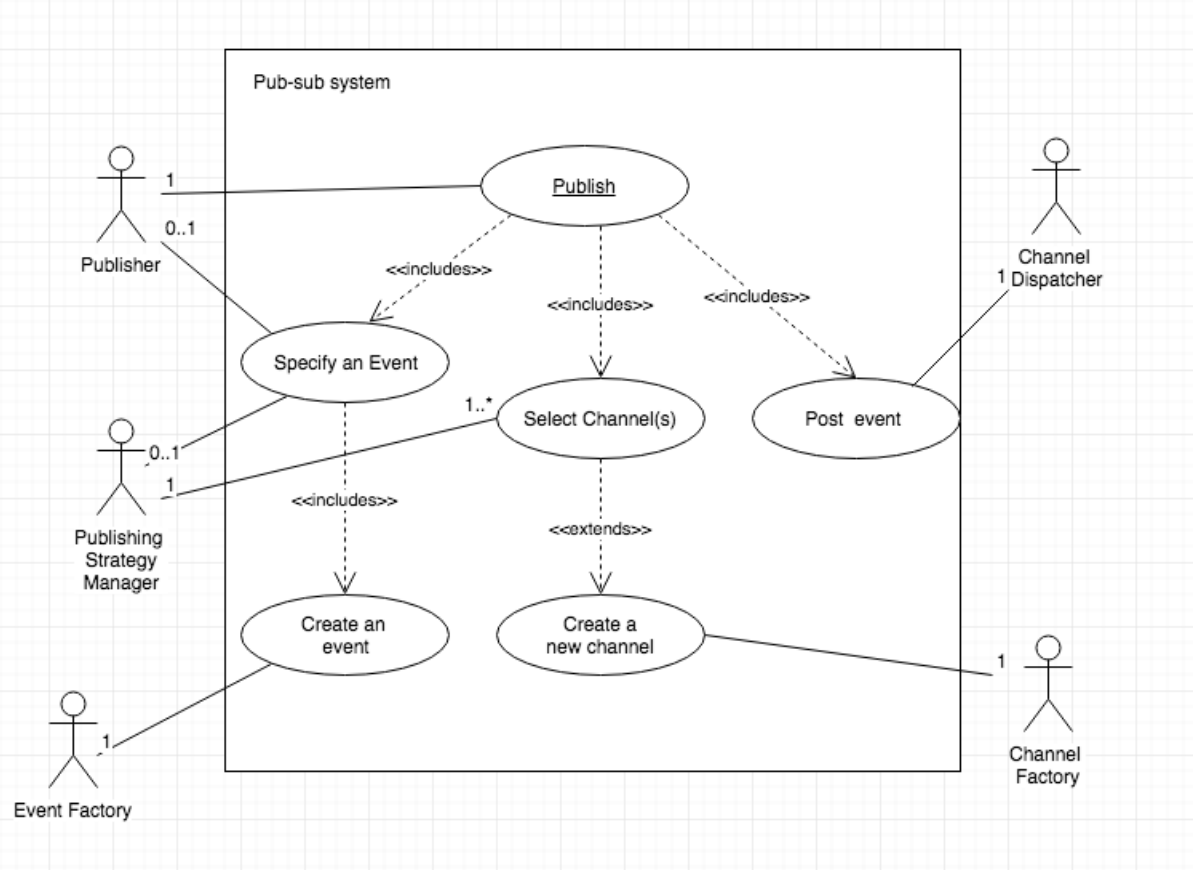
http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

2.3 Use Case Diagrams

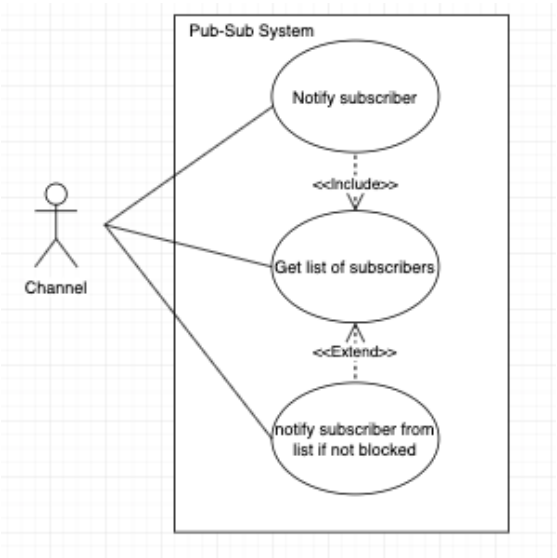
This section presents the business scenarios of the subject area in a graphical form.

Assign an appropriate title to the graphical scenario model and insert the diagram here.

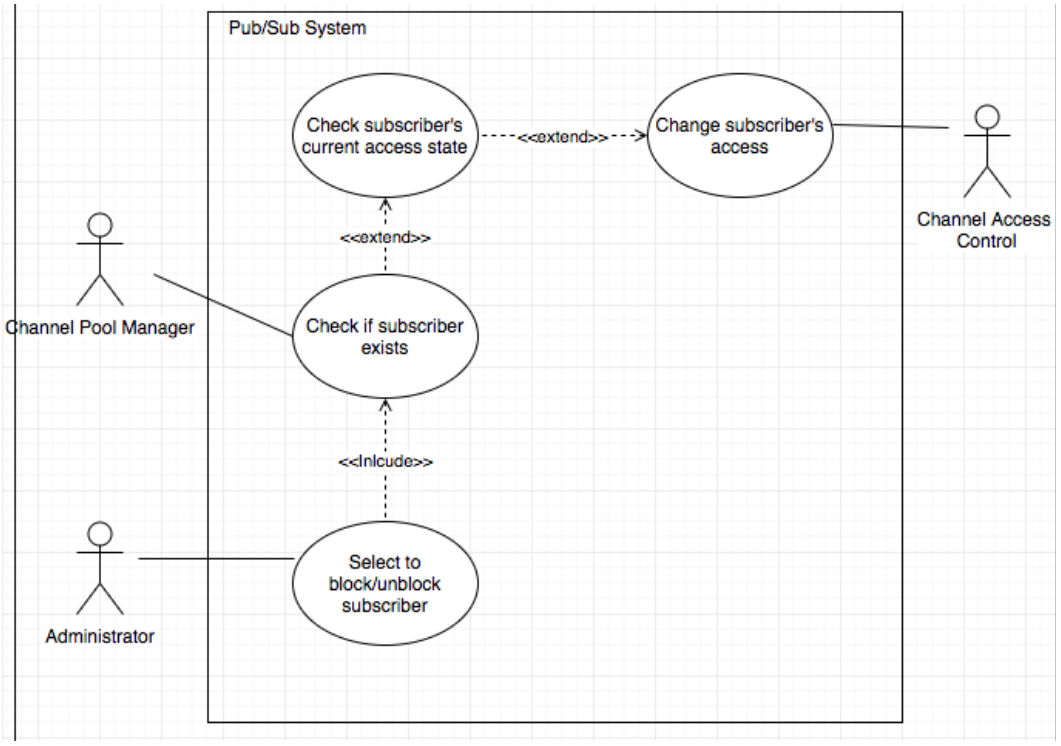
2.3.1 UC1: Publishing an event



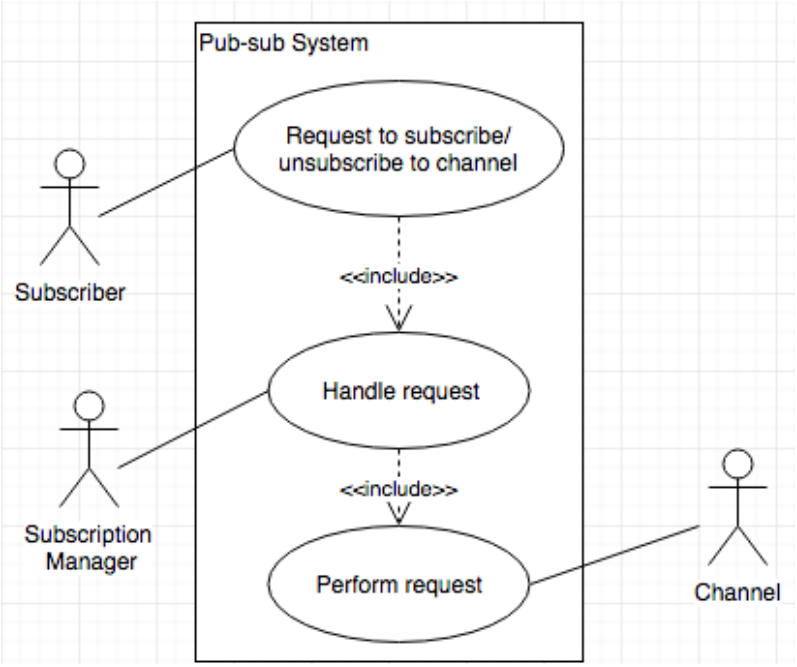
2.3.2 UC2: Channel notifying the subscribers an event has been published



2.3.3 UC4: Controlling access to a channel



2.3.4 UC5: Subscriber subscribes to a channel



3 Domain Model

3.1 Domain Model Class Diagram

The domain model class diagram for a Publish Subscribe System appears below:

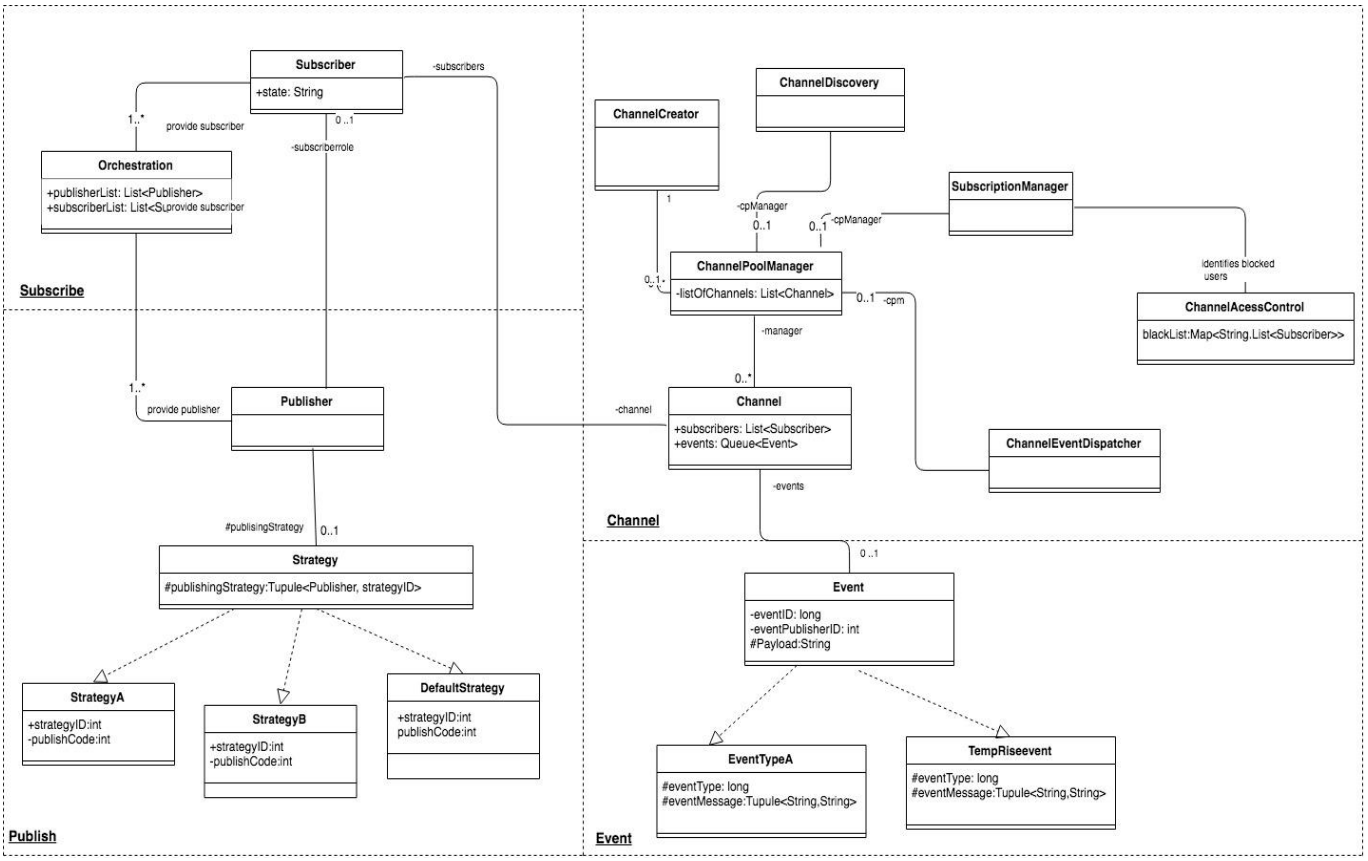


Figure 3.1.1 - Domain Model Class Diagram

3.2 Domain Model Class Definitions

Detailed below is an account of the major objects contained within the Publish Subscribe domain model shown above.

3.2.1.1 Publisher

Description	The publisher generates events and shares them to a shared medium - the channel. The publisher is associated with a strategy of the tupled form <publisher-ID, strategy-ID>. They use this information to associate a set of rules which allows them to post an event to a desired channel for subscribers to learn about.
Attributes	None.
Responsibilities	This object is responsible for publishing events to one or more channels based on a strategy.
Business Rules	The publisher must exist and have a proper ID/Strategy before generating an event and publishing to a channel.

3.2.1.2 Strategy

Description	This is the information associated with each publisher which denotes their ID and strategy. The strategy is read from a string file which allows the publisher to utilize this information in order to generate an event and post it to a channel. Without a strategy, the publisher cannot do this.
Attributes	publishingStrategy: A protected attribute, of type Tuple <Publisher, strategyID>. Each pair assigns a publisher with an strategy for publication. strategyID is denoted from a string file which is read from *strategies.str*. The information is implemented of type *int.*
Responsibilities	Give the publisher a strategy and ID to properly allow them to generate and post events to a channel.
Business Rules	The ID in the tuple must be of form int.

3.2.1.3 Subscriber

Description	This object represents an entity that handles events published on a channel. The subscriber is able to handle an event based on their state which is denoted to them by a file titled *states.sts* with information in the form of a tuple such as: <subscriber-ID, State-ID>.
Attributes	state - A public attribute of type String. A subscriber will behave differently based on its state. The state is read from a tuple in *states.sts.*
Responsibilities	This object may choose to subscribe to a channel(s) and/or event type. They are responsible for handling an event based on their given state.
Business Rules	A subscriber has a ‘state’ which determines how it handles an event.

3.2.1.4 Orchestration

Description	This class reads from the files *states.sts* and *strategies.str* in order to initialize the list of subscribers and list of subscribers for the pub-sub system.
Attributes	publisherList- A public attribute of type List<Publisher> subscriberList- A public attribute of type List<Subscriber> This object utilizes a buffered reader to read from the states and strategies file in order to fill two arrays.
Responsibilities	Initialize the assay of publishers and subscribers. Read from strategies/states file. Initialize list of channels.
Business Rules	None.

3.2.1.5 Channel

Description	This object represents an abstraction of a communication medium between the publisher and the subscriber.
Attributes	subscriber - A public attribute of type List<Subscriber> events- A public attribute of type Queue<Event>
Responsibilities	This object is responsible for maintaining a list of subscribers and a queue of events. -check if subscribers are not blocked -notify the subscribers that are not blocked
Business Rules	Once an event is added to the queue, the channel must notify its subscribers.

3.2.1.6 Channel Pool Manager

Description	This class provides a list of all the available channels and the subscribers attributed to them.
Attributes	listOfChannels - a private attribute of type List<Channel>.
Responsibilities	To returns a list of all available channels and their subscribers.
Business Rules	Once a subscriber is added to a channel, update the list of channels.

3.2.1.7 Channel Event Dispatcher

Description	This object is a helper method which provides an interface for the class *AbstractPublisher*. It takes the parameters of an “event” and the listOfChannels from the Channel Pool Manager in order to properly execute an event as posted.
Attributes	None.
Responsibilities	Post an event.
Business Rules	None.

3.2.1.8 Event

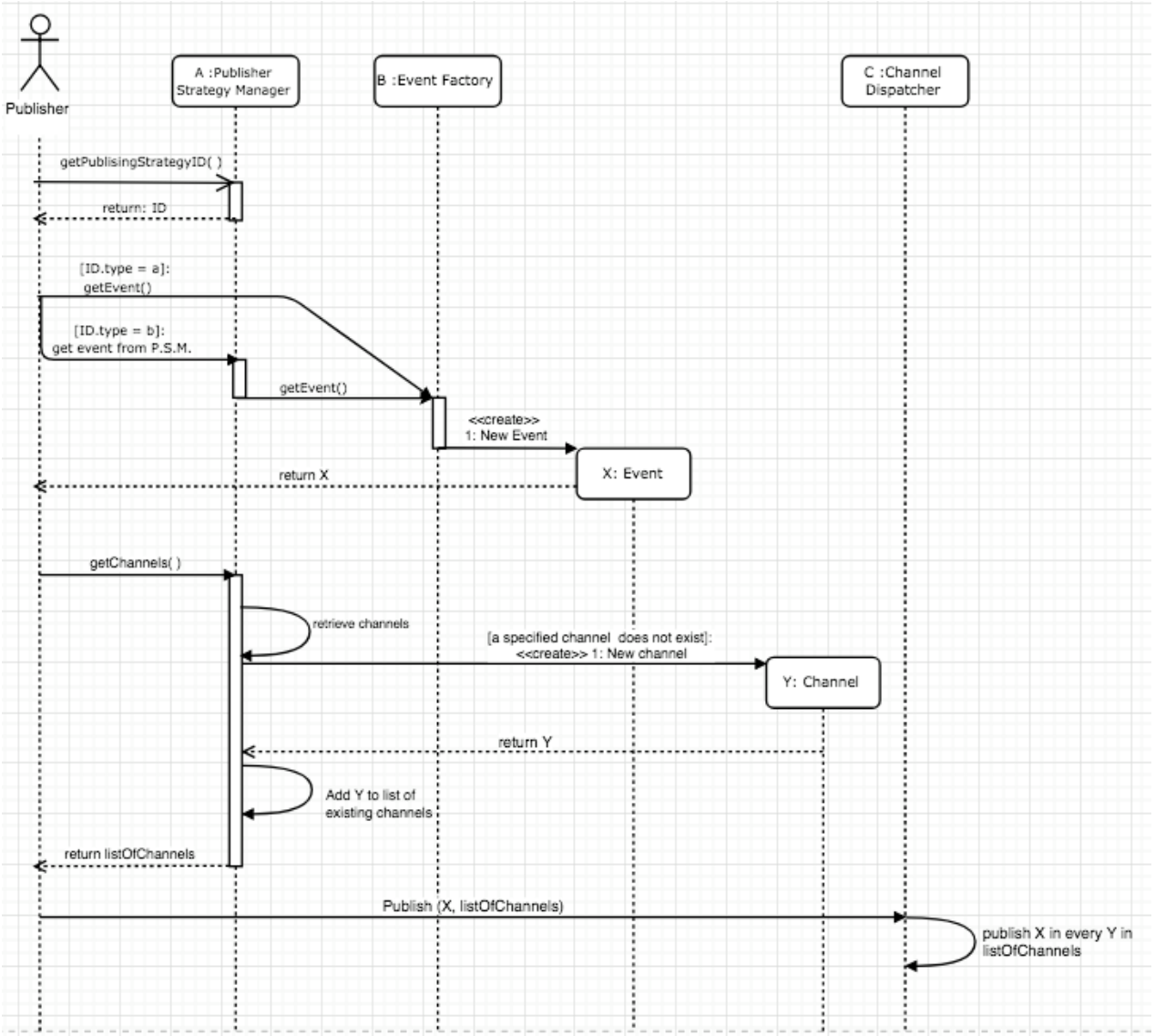
Description	An event is posted by the publisher to a desired channel. It has an ID, a message and an ID of the publisher who generated the event. The event is visible to subscribers of a channel who can handle the event according to their specified state. The event is generated by the publisher dependent on their specified strategy.
Attributes	eventID - private attribute of type long eventPublisherID- private attribute of type int which designates the object issuing the event

	payload- a private attribute of type String, which contains a header and body for an event.
Responsibilities	Display event and be handled by a subscriber of the channel posted to.
Business Rules	None.

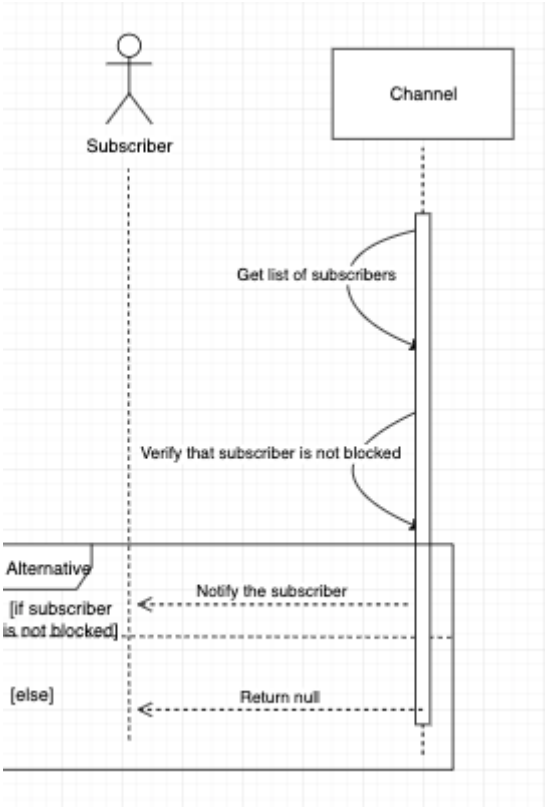
4 Interaction Diagrams

4.1 Sequencing Diagrams

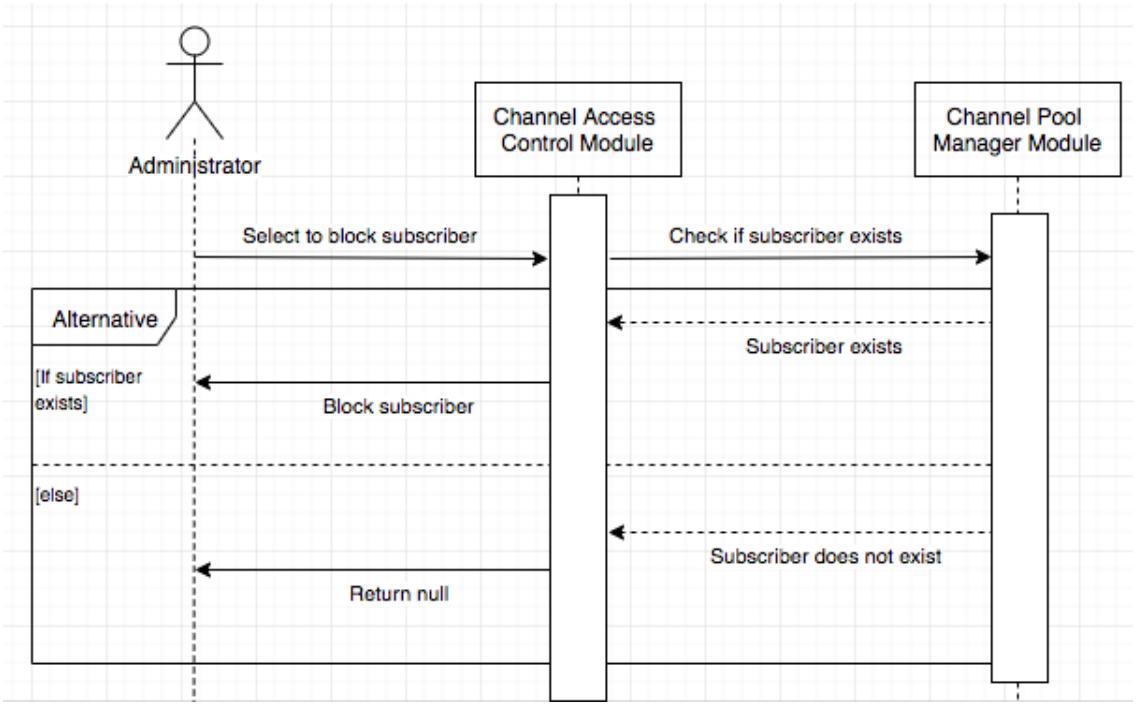
4.1.1 UC1: Publishing an event



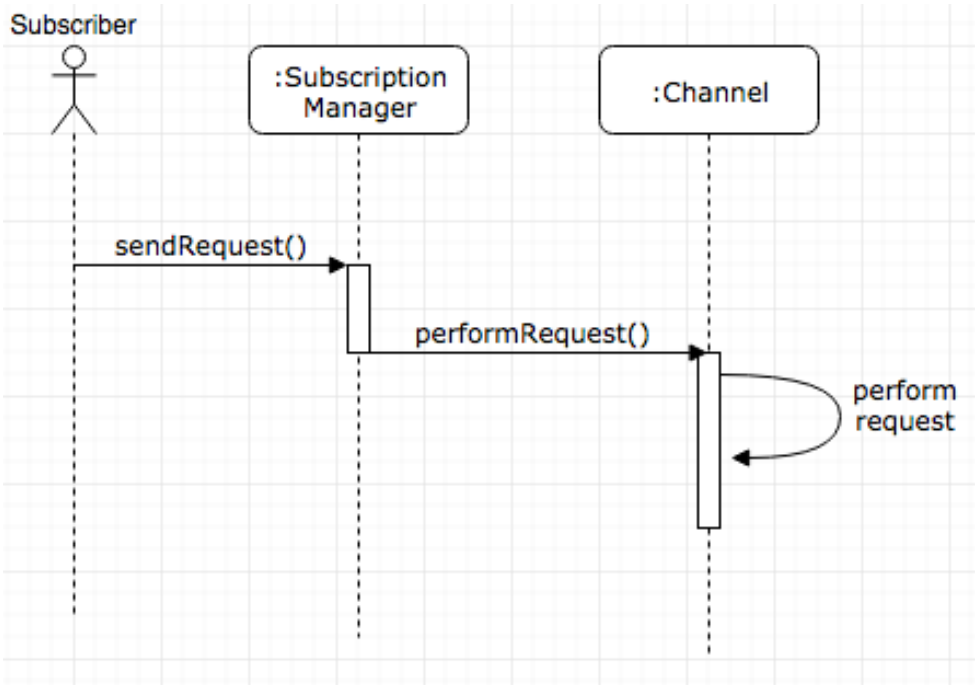
4.1.2 UC2: Channel notifying the subscribers an event has been published



4.1.3 UC3: Controlling access to a channel

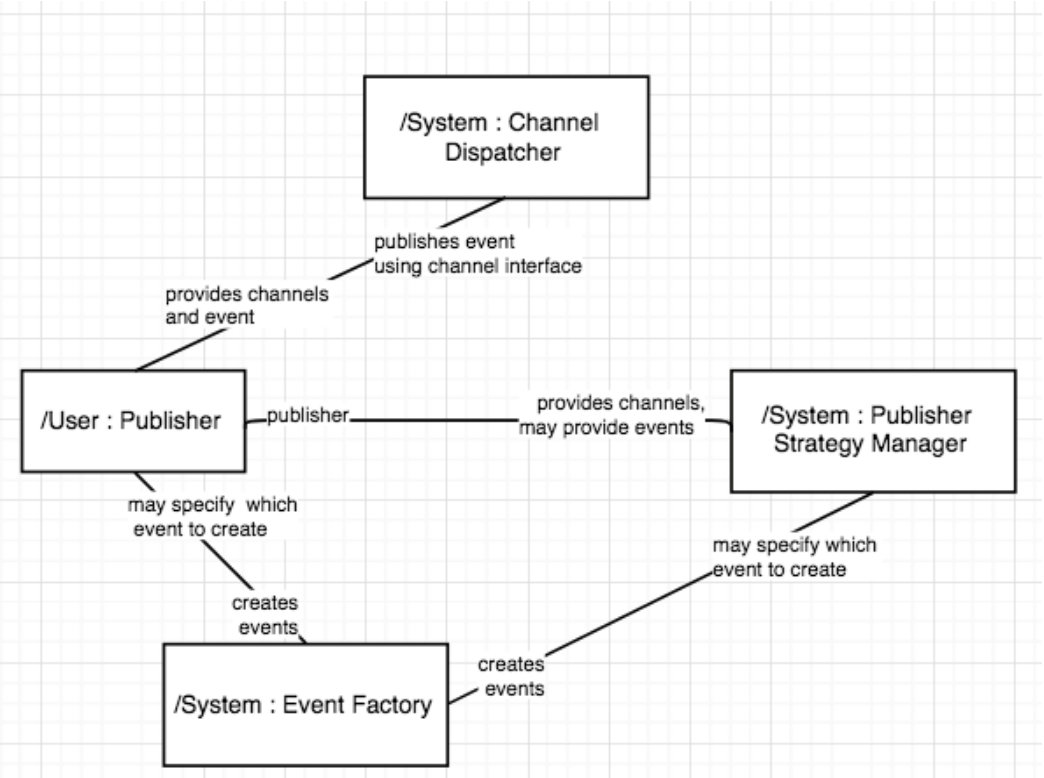


4.1.4 UC5: Subscriber subscribes to a channel

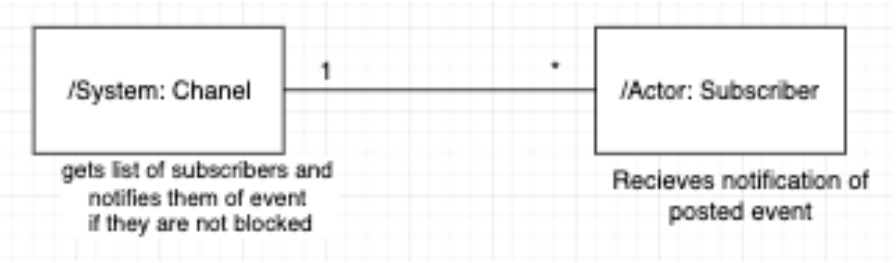


4.2 Collaboration Diagrams

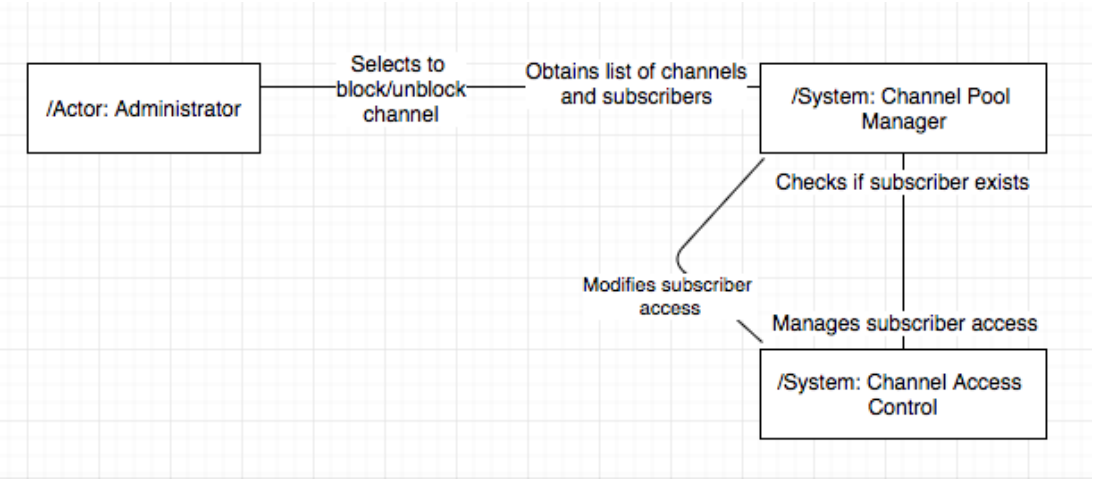
4.2.1 UC1: Publishing an event



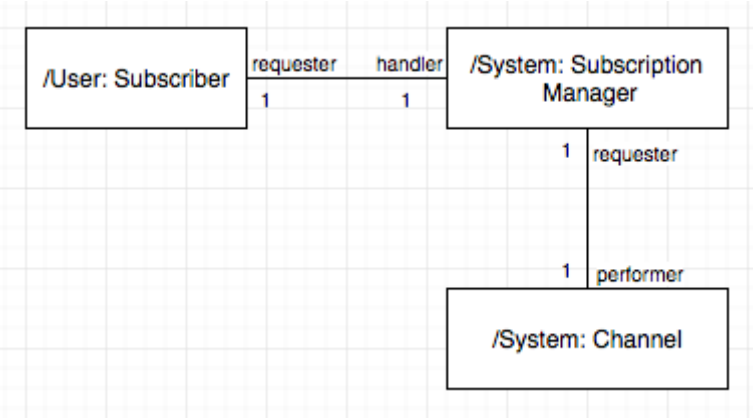
4.2.2 UC2: Channel notifying the subscribers an event has been published



4.2.3 UC4: Controlling access to a channel



4.2.4 UC5: Subscriber subscribes to a channel



5 Non-Functional Requirements Specification

5.1 Overview

The non-functional requirements of the system comprises of specifications that are necessary for the smooth operation of the system as a whole. This includes enabling technologies, capacity planning, network, workstations and operational parameters

5.2 Enabling Technologies

5.2.1 Target Hardware & Hardware Interfaces

Although the software will be primarily developed on Windows, it is cross-platform and can run on other operating systems, such as Unix-like systems. Because the events will be published on a shared medium (channel), the program will be run on an external server coded in Java via Eclipse IDE. Java version 8.0 must be installed on the machine before executing this program.

5.2.2 Target Development Environment

The software will be developed in a Windows environment and using Java. Eclipse Oxygen will be the Integrated Development Environment (IDE) used for programming. An external server database will be used to hold the data of all the channels and list of subscribers.

5.2.3 System Interfaces

The user will be provided with a GUI interface in order to subscribe and unsubscribe from channels, as well as to handle various events.

5.3 Capacity Planning

5.3.1 Permanent Storage

The recommended minimum storage for this software project is a user storage space of approximately 3GB. Greater storage will lower risks of the software crashing, and increase the speed of the software.

5.4 Network

The Pub-Sub system should have connectivity to the Internet with bandwidth sufficient enough to handle information spread across a multitude of domains and dynamic populations of publishers and subscribers. Furthermore, the network should possess efficient packet protocols to prevent delays

5.5 Workstations

The minimum system requirements for the computers used for the development, deployment and execution of the system are listed below.

- Java 8.0 or greater
- Eclipse 4.10 or greater
- Computer contained 126 gb of disk space
- Computer Memory was 8 gb 1600 MHz DDR3
- System processor was intel Core i7; 2.2 GHz
- System was developed on macOS
- The computer required connectivity to a LAN with bandwidth high enough to handle at least 100 channels and at least 200 publishers and 200 subscribers in a given instance
- A display setting with a resolution 1440 x 90 and a Intel HD 6000 graphics card

5.6 Operational Parameters

5.6.1 Useability

The publish-subscribe system has to be able to handle at least 100 channels. The system has to be able to handle at least 200 publishers and 200 subscribers at various configurations. Accordingly, a GUI will be implemented to meet these needs and provide a user-friendly dashboard. The interface must be simple for publishers and subscribers to navigate through, otherwise communication will be complex, defeating the entire purpose of the system.

5.6.2 Reliability

Recoverability & Backup

With regards to reliability, the publish-subscribe system requires a method of ensuring that data is not lost. To accomplish this, the system will be linked to the cloud and relay data live to the cloud. In the case of any failure or data loss, the cloud will be contacted to restore data.

Restart

To ensure an efficient stability of the system, the restart time can be measured and tested across several workstations. Higher stability would be achieved with less variability.

5.6.3 Maintainability

This system should be straightforward, making it easy to analyze, change and test. It will include classes that are implemented using abstract and standard functions. The abstraction will simplify the ability to modify the software, as abstracted classes are not required to be modified. In order to modify the software to meet novel user needs, only the implementation details of the abstracted classes need to be changed. This restricted freedom secures the software while providing room for modification.

5.6.4 Portability

The adaptability, installability and replaceability of the Pub-Sub System is fairly straightforward. For the adaptability and replaceability, review section 5.6.3 Maintainability. As for the installation, access to a web browser is required to download Java software. This comes the Java Runtime Environment (JRE) which consists of the Java Virtual Machine (JVM), Java platform core classes and supporting libraries. The system would be binary complementary with workstations of the same architecture as it will be programmed to be platform independent using Java. Since java is open source, the JVM should be deployable on any operating system as long as the standard dependencies and compilers are installed in the machine.

6 Domain Dictionary

6.1 Terms and Abbreviations

Term	Definition
Graphical User Interface (GUI)	Form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation
Tuple	Ordered set of values separated by a comma
Software Requirements Specification (SRS)	Document that fully describes the expected behaviour of a software system
Java Runtime Environment (JRE)	A set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.
Java Virtual Machine (JVM)	A virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.