Publisher-Subscriber System

Design Document

Computer Science 2212B

Version:	1.0
Print Date:	
Release Date:	TBA
Release State:	Initial
Approval State:	Draft
Approved by:	-
Prepared by:	Seif, Pouya, Din, Sarah, Tyler
Reviewed by:	Seif, Pouya, Din, Sarah Tyler
Path Name:	~/Group1_srs
File Name:	Group1_srs
Document No:	1

Document Change Control

Version	Date	Authors	Summary of Changes
1.0	03/12	Seif, Pouya, Din, Sarah, Tyler	Initial Write
2.0	03/21	Seif, Pouya, Din, Sarah, Tyler	Added Method Details, Diagrams
3.0	04/04	Seif, Pouya, Din, Sarah, Tyler	Added Design Decisions, Overview, and all other major components
4.0	04/07	Seif, Pouya, Din, Sarah, Tyler	Final revisions

Document Sign-Off

Name (Position)	Signature	Date
Seif	Seif	04/07
Pouya	Pouya	04/07
Din	Din	04/07
Sarah	Sarah	04/07
Tyler	Tyler R	04/07

Contents

1 Introduction

1.1 Overview

The following project is a prototype Publisher-Subscriber pattern system. Publish-subscribe is a message pattern that allows message creators, called publishers, to categorize published messages into classes without knowledge of the message receivers, subscribers. The document is divided into four sections that outline some of the decisions throughout the design phase. The resources and their references are cited accordingly.

The second and third sections of the document cover the major design decisions that were made followed by a discussion of the architecture, which includes component diagrams. The individual factories, for publisher, subscriber, state, strategy and event are considered for modification and implementation. The remaining components are only changed if the respective factories, once modified, warrant a change.

The fourth section contains the class diagrams depicting the entire system, showing how each class interacts with one another. The class diagrams are divided into their respective packages, with overall interactions shown using boxes. The class diagrams are of UML notation and an interface section is also included to describe the methods and attributes used within each package and respective class.

1.2 Resources - References

Pub-Sub System https://aws.amazon.com/pub-sub-messaging/
Pub-Sub System https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
Singleton Design Pattern http://www.ibm.com/developerworks/library/co-single/
Singleton Design Pattern https://www.javaworld.com/article/2073352/core-java-simply-singleton.html

2 Major Design Decisions

The first major design decision made was to implement the singleton design pattern in the ChannelAccessControl, ChannelCreator, ChannelDiscovery, ChannelEventDispatcher, and SubscriptionManager classes, which is a creational design pattern. Singletons are needed when objects are accessed by disparate objects throughout a software system, and therefore need a global point of access. Singletons ensure that only one instance of a class is created while providing a global point of access, and further allowing multiple instances in the future without

affecting a singleton class's clients. There are several potential consequences of the singleton design pattern. On the positive side, singletons simplify the structure of the program, allow the creation of a specific number of objects as opposed to just one, and allow for extension of the singleton object using specification. Negative consequences include the potential for the object to behave like a global variable if implemented incorrectly, higher operational cost, and additional caution when using parallel processes.

The second major design decision made was the use of a factory design pattern, another creational pattern. This design pattern allows us to define an interface in a class that can be used to construct objects. The kind of objects created are ultimately dictated by the type of classes that are applied to the factory interface. A related design decision made was to implement separate concrete classes for all event types, publisher types, subscriber types, strategies, and states. This provided superior modularity and organization of the code. All of the additional types were added to the enumerations of Event Factory, State Factory, Strategy Factory, and Subscriber Factory.

For the behavioral design decisions, the state pattern and strategy pattern were chosen. The state pattern allows for an object of a class to change its behavior according to its state. This behavior applies to all objects that are in the same state, which looks like the object has changed classes to outside clients. The strategy design pattern creates a family of algorithms which are called through a common interface, which allows client programs and algorithms that can be called to change independently. This allows for greater flexibility and reusability of code, and the algorithms that can be called from the client code can be changed dynamically at run time. Potential drawbacks include the cost to associate algorithms with objects, and the need for a common interface for all algorithms being used which is inflexible.

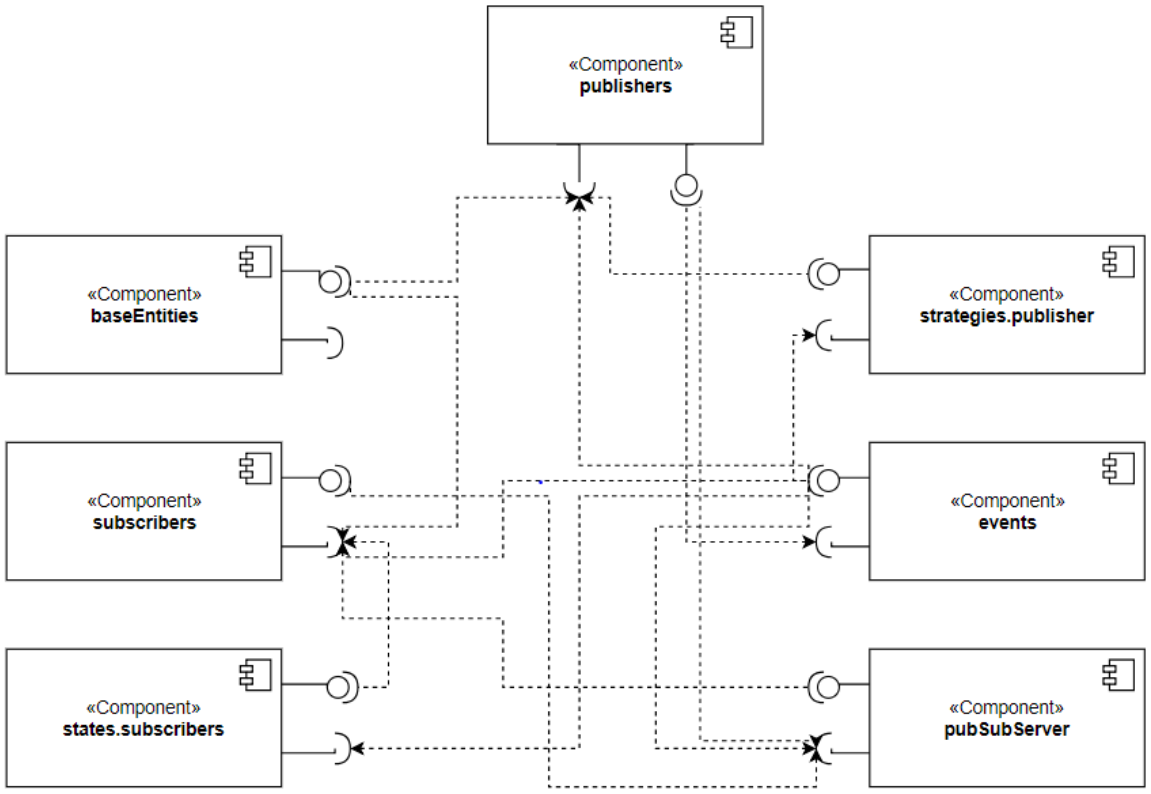
Additional, more specific design decisions made were to add Publisher ID Maker and Subscriber ID Maker classes to more effectively keep track of which publishers and subscribers are publishing and receiving events. These were implemented using ints instead of longs because the system only needed to handle 200 publishers or subscribers. The specific channels associated with each strategy were also chosen, and were grouped based on topic to remain logical. The decision was made to list channels that were not provided to us, to more effectively test if channels that didn't exist were being created.

3 Architecture

This project uses a Publisher-Subscriber system architecture, which is an Interacting Processes architecture. Interacting Processes architecture means that procedures are not invoked directly, and instead a component announces or broadcasts one or more events. There are multiple publishers that are each associated with strategies. The strategies are able to determine what event should be published and what channel the events will be published to. When a publisher

publishes an event, they can either specify the event that should be published or allow the strategy to dictate this. The event is pushed onto the queue of the specified channel, and then the channel is solely responsible for communicating with subscribers. Subscribers are able to subscribe to specific channels based on topic, and when an event is pushed onto the queue of a channel, the channel sends a notification to all of its unblocked subscribers. Channels are able to block and unblock subscribers. Subscribers have states which dictate how they handle events that they are notified of. This architecture is highly modular and reusable with a high degree of functional cohesion, which are key design objectives.

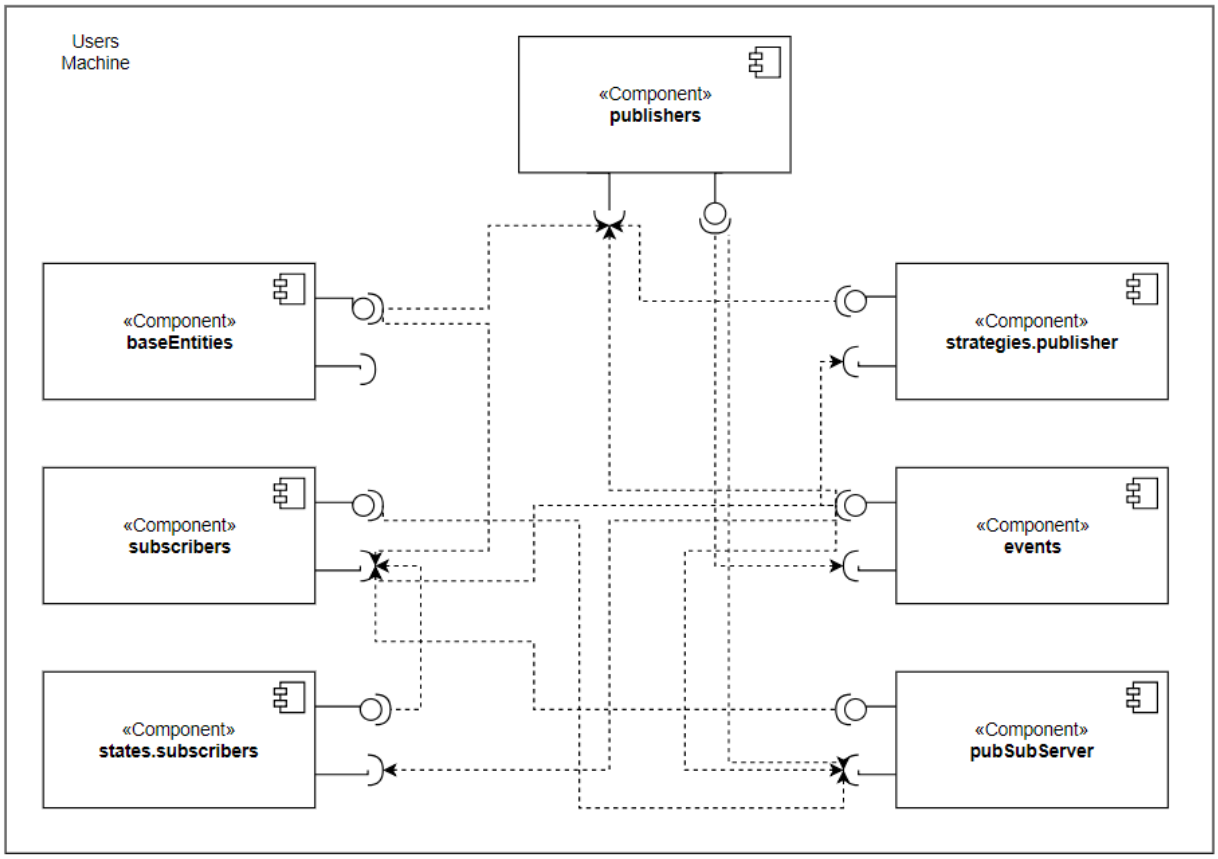
3.1 Component Diagram



Provided Interface	Incoming Interface Methods
baseEntities	
publishers	baseEntities.IEntity
	events.AbstractEvent
	strategies.publisher.IStrategy

	strategies.publisher.StrategyFactory
	strategies.publisher.StrategyName
subscribers	baseEntities.IEntity
	events.AbstractEvent
	states.subscriber.IState
	states.subscriber.StateFactory
	states.subscriber.StateName
	pubSubServer.SubscriptionManager
events	publishers.AbstractPublisher
pubSubServer	events.AbstractEvent
	subscribers.AbstractSubscriber
	publishers.AbstractPublisher
states.subscribers	events.AbstractEvent
strategies.publisher	events.AbstractEvent

3.2 Deployment Diagram

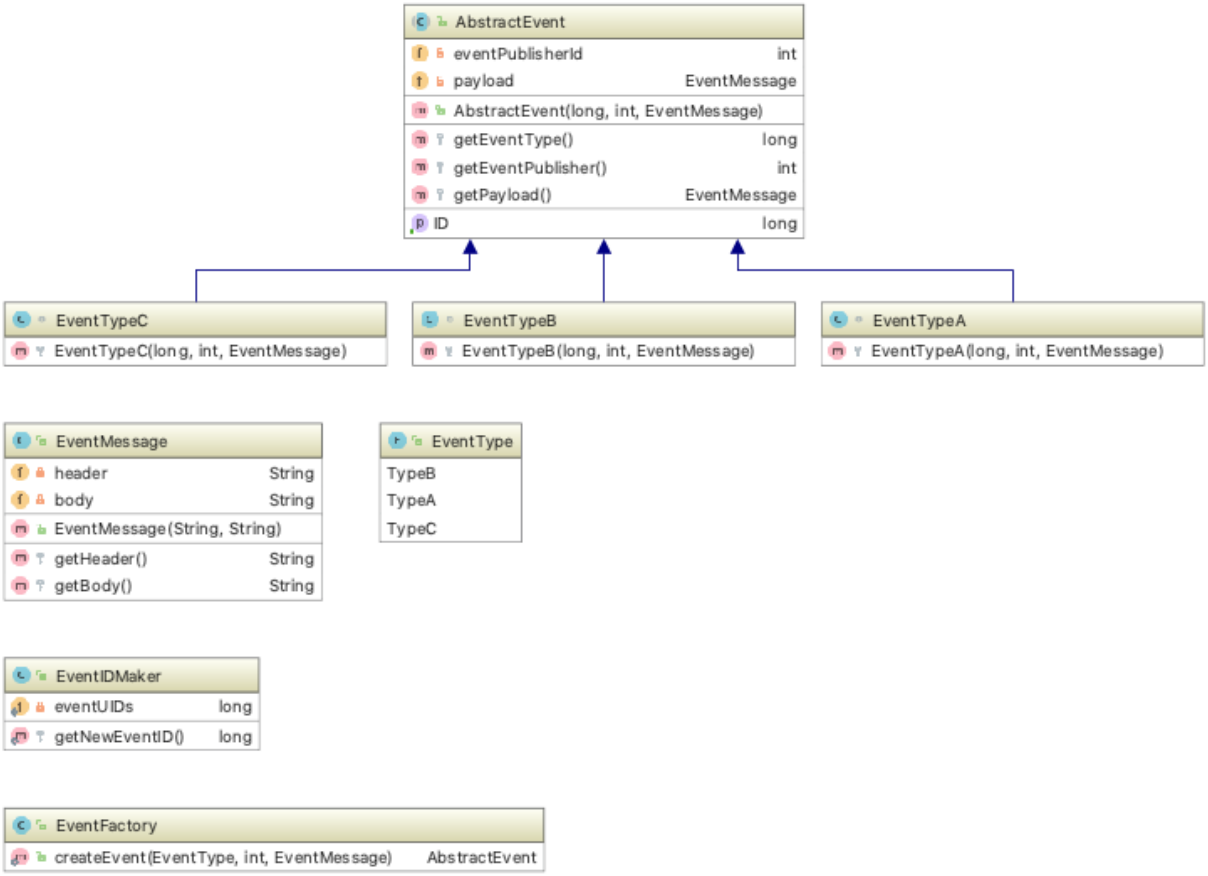


4 Detailed Class Diagrams

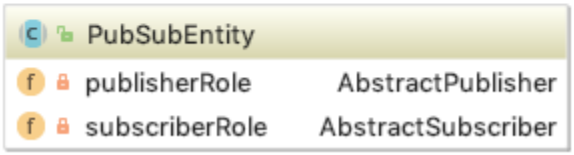
4.1 UML Class Diagrams

BaseEntities

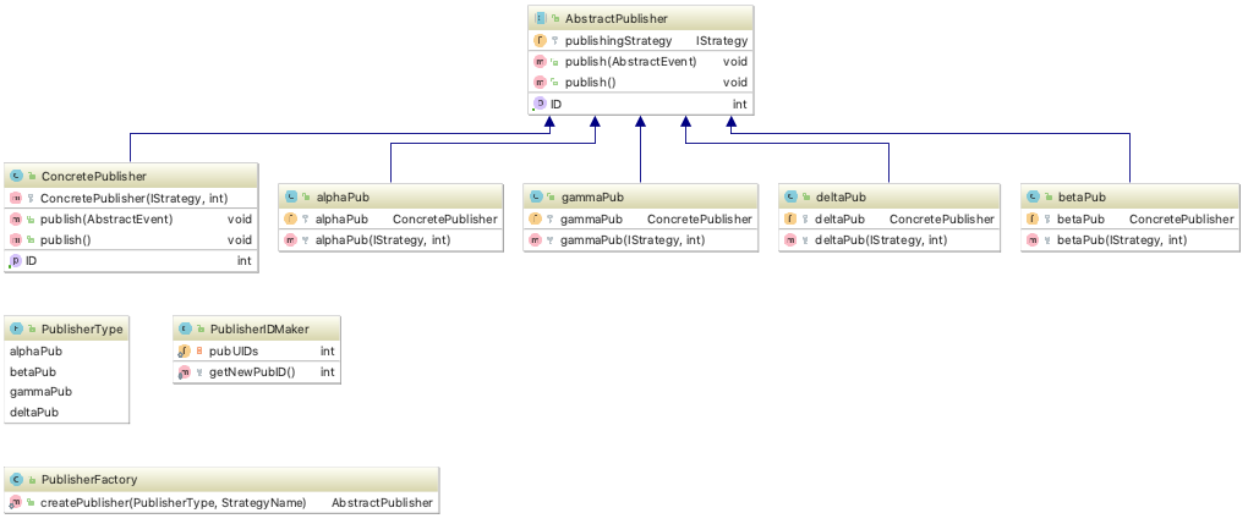
Events



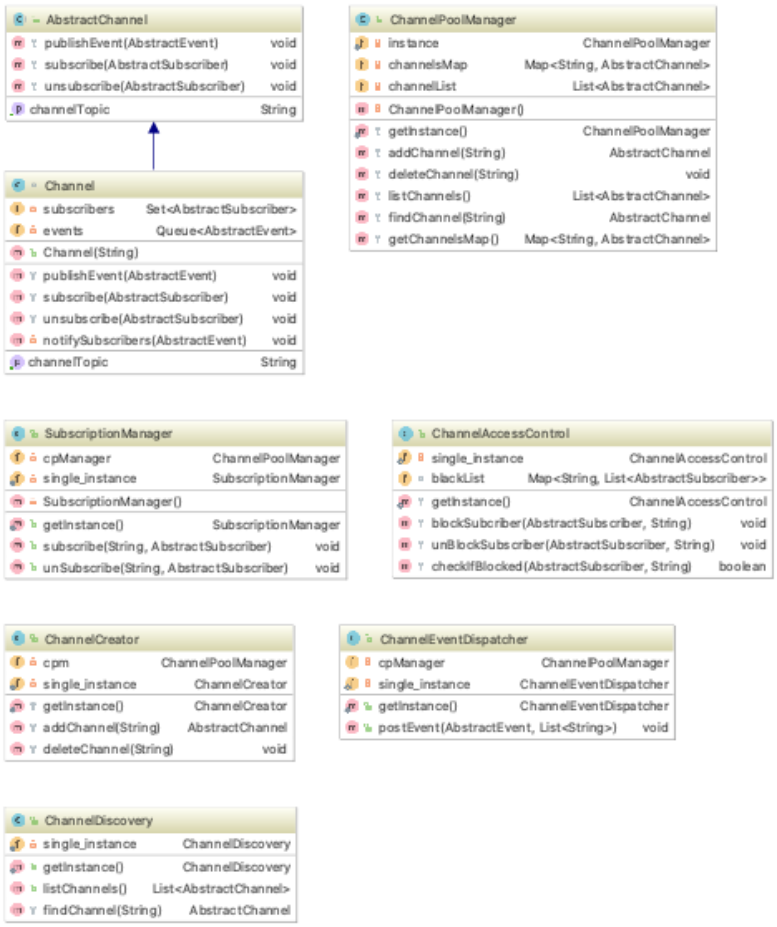
Orchestration



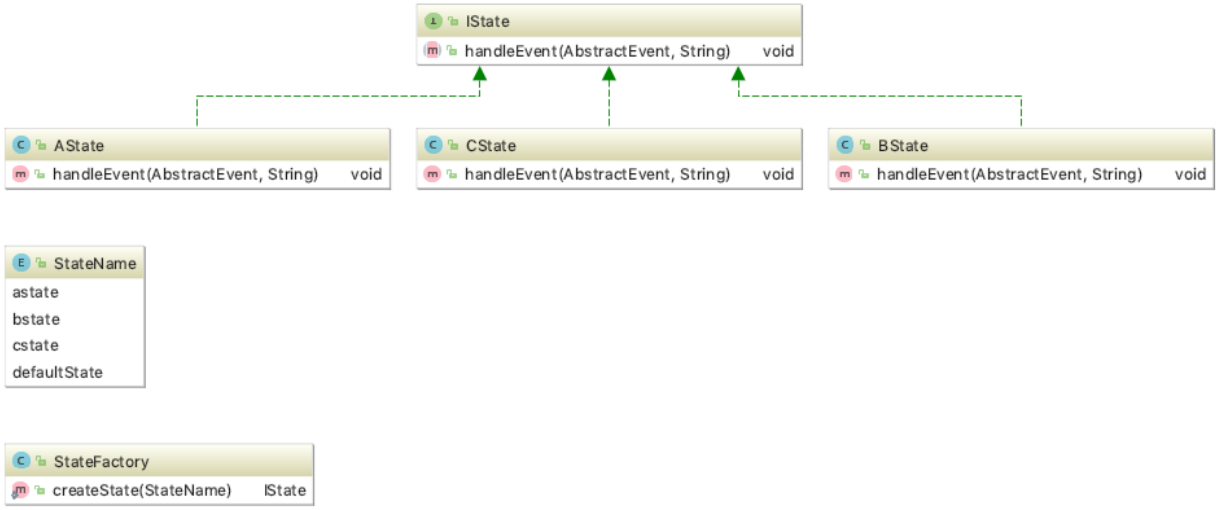
Publishers



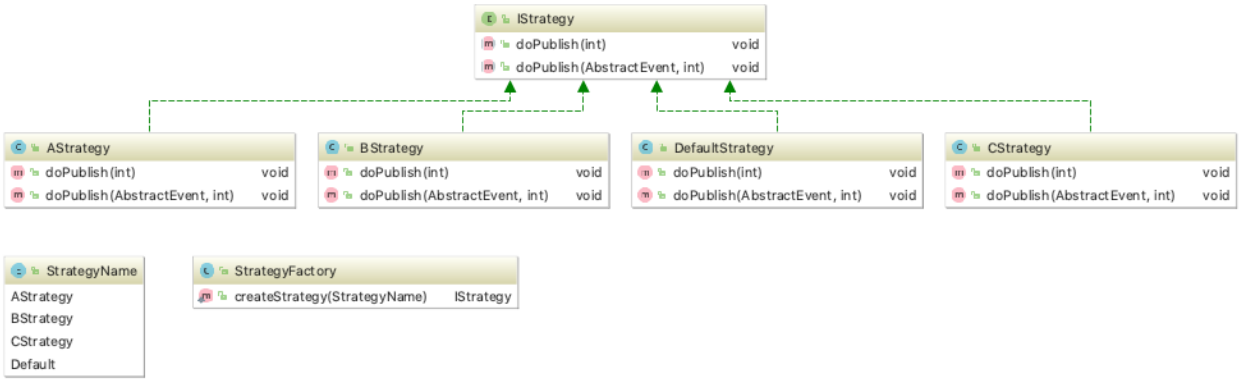
pubSubServer



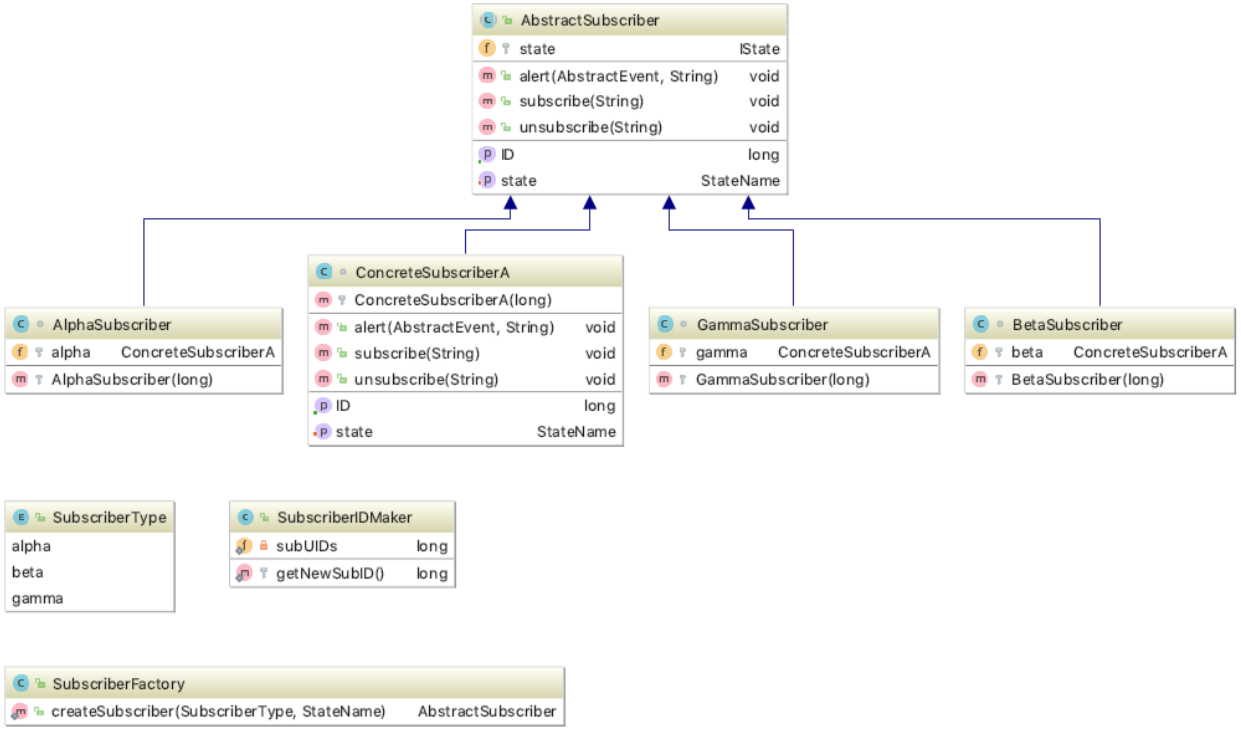
states.subscriber



strategies.publisher



Subscribers



4.2 Method Details

AbstractEvent

Data members:

private long eventId = -1;
private int eventPublisherId;
private EventMessage payload = null;

Methods:

public AbstractEvent(long eventID, int eventPublisher, EventMessage payload)
protected long getEventType()

protected int getEventPublisher()
protected EventMessage getPayload()
public long getID()

EventFactory

Data members:

Methods:

public static AbstractEvent createEvent(EventType eventType, int eventPublisherId, eventMessage payload)

EventIDMaker

Data members:

private static long eventUIDs = 0L;

Methods:

protected static long getNewEventID()

EventMessage

Data members:

private String header = null;
private String body = null;

Methods:

public EventMessage(String header, String body)
protected String getHeader()
protected String getBody()

EventType

Data members:

Methods:

public enum EventType

EventTypeA

Data members:

Methods:

protected EventTypeA(long eventID, int eventPublisherId, EventMessage payload)

EventTypeB

Data members:

Methods:

protected EventTypeB(long eventID, int eventPublisherId, EventMessage payload)

EventTypeC

Data members:

Methods:

protected EventTypeC(long eventID, int eventPublisherId, EventMessage payload)

Orchestration

Data members:

Methods:

public static void main(String[] args)
private List<AbstractPublisher> createPublishers() throws IOException
private List<AbstractSubscriber> createSubscribers() throws IOException

PubSubEntity

Data members:

private AbstractPublisher publisherRole = null;
private AbstractSubscriber subscriberRole = null;

Methods:

AbstractPublisher

Data members:

protected IStrategy publishingStrategy = null;
protected int id = 0;

Methods:

public void publish(AbstractEvent event) {};
public void publish() {};

ConcretePublisher

Data members:

Methods:

protected ConcretePublisher(IStrategy concreteStrategy)
public void publish(AbstractEvent event)
public void publish()
public int getID()

PublisherFactory

Data members:

Methods:

public static AbstractPublisher createPublisher(PublisherType publisherType, StrategyName strategyName)

PublisherType

Data members:

Methods:

public enum PublisherType

PublisherIDMaker

Data members:

private static int pubUIDs = 0;

Methods:

protected static int getNewPubID()

alphaPub

Data members:

protected ConcretePublisher alphaPub;

Methods:

protected alphaPub(IStrategy strat, int id)

betaPub

Data members:

protected ConcretePublisher betaPub;

Methods:

protected betaPub(IStrategy strat, int id)

gammaPub

Data members:

protected ConcretePublisher gammaPub;

Methods:

protected gammaPub(IStrategy strat, int id)

deltaPub

Data members:

protected ConcretePublisher deltaPub;

Methods:

protected deltaPub(IStrategy strat, int id)

AbstractChannel

Data members:

Methods:

protected void publishEvent(AbstractEvent event)
protected void subscribe(AbstractSubscriber subscriber)
protected void unsubscribe(AbstractSubscriber subscriber)
public String getChannelTopic()

Channel

Data members:

private Set<AbstractSubscriber> subscribers = new HashSet<AbstractSubscriber>();
private Queue<AbstractEvent> events = new ArrayDeque<AbstractEvent>();
private String channelTopic;

Methods:

protected void publishEvent(AbstractEvent event)
protected void subscribe(AbstractSubscriber subscriber)
protected void unsubscribe(AbstractSubscriber subscriber)
private void notifySubscribers(AbstractEvent event)
public String getChannelTopic()
public Channel(String channelTopic)

ChannelAccessControl

Data members:

Map<String, List<AbstractSubscriber>> blackList = new HashMap<>();
private static ChannelAccessControl single_instance = null;

Methods:

public static ChannelAccessControl getInstance()
public void blockSubcriber(AbstractSubscriber subscriber, String channelName)
public void unBlockSubscriber(AbstractSubscriber subscriber, String channelName)
public boolean checkIfBlocked(AbstractSubscriber subscriber, String channelName)

ChannelCreator

Data members:

private ChannelPoolManager cpm = ChannelPoolManager.getInstance();
private static ChannelCreator single_instance = null;

Methods:

public static ChannelCreator getInstance()
protected AbstractChannel addChannel(String channelName)
protected void deleteChannel(String channelName)

ChannelDiscovery

Data members:

private static ChannelDiscovery single_instance = null;

Methods:

public static ChannelDiscovery getInstance()
public List<AbstractChannel> listChannels()
protected AbstractChannel findChannel(String channelName)

ChannelEventDispatcher

Data members:

private ChannelPoolManager cpManager = ChannelPoolManager.getInstance();

private static ChannelEventDispatcher single_instance = null;

Methods:

public static ChannelEventDispatcher getInstance()
public void postEvent(AbstractEvent event, List<String> listOfChannels)

ChannelPoolManager

Data members:

private static ChannelPoolManager instance = null;
private Map<String, AbstractChannel> channelsMap = new HashMap<String, AbstractChannel>();
private List<AbstractChannel> channelList = new ArrayList<AbstractChannel>();

Methods:

private ChannelPoolManager()
public static ChannelPoolManager getInstance()
protected AbstractChannel addChannel(String channelName)
protected void deleteChannel(String channelName)
protected List<AbstractChannel> listChannels()
protected AbstractChannel findChannel(String channelName)
protected Map<String, AbstractChannel> getChannelsMap()

SubscriptionManager

Data members:

private ChannelPoolManager cpManager;
private static SubscriptionManager single_instance = null;

Methods:

public static SubscriptionManager getInstance()
public void subscribe(String channelName, AbstractSubscriber subscriber)
public void unSubscribe(String channelName, AbstractSubscriber subscriber)
private SubscriptionManager()

IState

Data members:

Methods:

public void handleEvent(AbstractEvent event, String channelName)

IStrategy

Data members:

Methods:

public void doPublish(int publisherId)
public void doPublish(AbstractEvent event, int publisherId)

StrategyName

Data members:

Methods:

public enum StrategyName

AbstractSubscriber

Data members:

protected IState state;
protected long id = 0;

Methods:

public void setState(StateName stateName)
public void alert(AbstractEvent event, String channelName)
public void subscribe(String channelName)
public void unsubscribe(String channelName)

ConcreteSubscriberA

Data members:

Methods:

protected ConcreteSubscriberA(long id)
public void setState(StateName stateName)

public void alert(AbstractEvent event, String channelName)
public void subscribe(String channelName)
public void unsubscribe(String channelName)
public long getID()

SubscriberFactory

Data members:

Methods:

public static AbstractSubscriber createSubscriber(SubscriberType subscriberType, StateName stateName)

SubscriberType

Data members:

Methods:

public enum SubscriberType

AlphaSubscriber

Data members:

protected ConcreteSubscriberA alpha;

Methods:

protected AlphaSubscriber(long id)

BetaSubscriber

Data members:

```
protected ConcreteSubscriberA beta;
```

Methods:

```
protected BetaSubscriber(long id)
```

GammaSubscriber

Data members:

```
protected ConcreteSubscriber A gamma;
```

Methods:

```
protected GammaSubscriber(long id)
```

SubscriberIDMaker

Data members:

```
private static long subUIDs = 0L;
```

Methods:

```
protected static long getNewSubID()
```

StateFactory

Data members:

Methods:

public static IState createState(StateName stateName)

StateName

Data members:

Methods:

public enum StateName

AState

Data members:

Methods:

public void handleEvent(AbstractEvent event, String channelName)

BState

Data members:

Methods:

public void handleEvent(AbstractEvent event, String channelName)

CState

Data members:

Methods:

public void handleEvent(AbstractEvent event, String channelName)

StrategyFactory

Data members:

Methods:

public static IStrategy createStrategy(StrategyName strategyName)

AStrategy

Data members:

Methods:

public void doPublish(int publisherId)
public void doPublish(AbstractEvent event, int publisherId)

BStrategy

Data members:

Methods:

public void doPublish(int publisherId)
public void doPublish(AbstractEvent event, int publisherId)

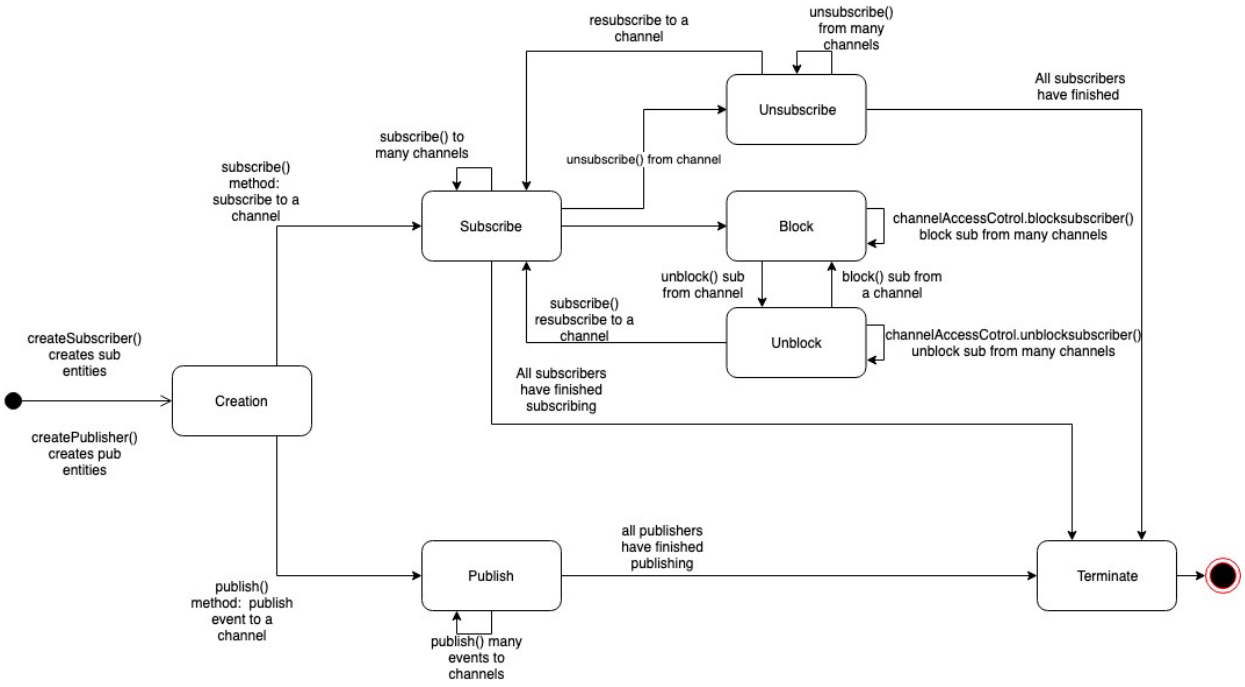
CStrategy

Data members:

Methods:

public void doPublish(int publisherId)
public void doPublish(AbstractEvent event, int publisherId)

5 State Class Diagrams



6 Domain Dictionary

6.1 Terms and Abbreviations

Term	Definition
UML	UML is an industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It was standardized by the Object Management Group.
Pub-Sub	Pub-Sub is an abbreviation for Publisher-Subscriber system, which is defined above as a message pattern that allows message creators, publishers, to categorize published messages into classes without knowledge of the message receivers, subscribers.
Singleton Design Pattern	The Singleton Design Pattern ensures a class has only one instance, and provides a global point of access to it.

Meeting Minutes - SDD & Code

Date: March 7
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Reviewed SDD requirements in depth based on project description and in-class notes, determined how to divide up the sections amongst group members, created timelines for completing and reviewing the SDD

Date: March 12
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Created outline of all needed changes and additions to the code, found references to assist in completion of the SDD

Date: March 21
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Reviewed work completed so far on the SDD, started to work on code based on outline created

Date: March 26
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Completed a significant portion of the code, compiled a list of questions to ask during office hours, worked on SDD

Date: April 4
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Finished remainder of code, made revisions to SDD to remain consistent with code

Date: April 7
Group Members: Seif, Pouya, Din, Sarah, Tyler
Topics Discussed: Final check through of code and SDD, discussion of presentation