THE UNIVERSITY OF WESTERN
ONTARIO DEPARTMENT OF
COMPUTER SCIENCE

CS2212B Introduction to Software Engineering

Project Description--- Winter 2019

Introduction

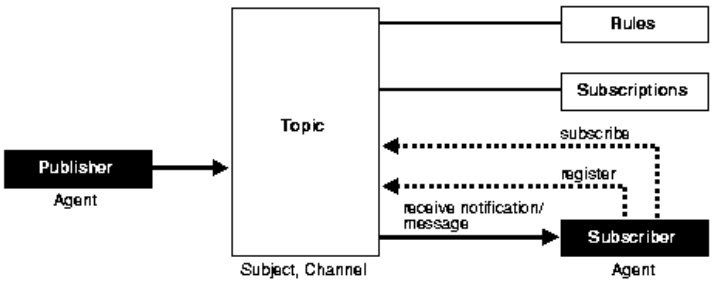
This project is about the specification, design and implementation of a simple publish subscribe system. The system may be simple in the operations it offers, but it provides ample opportunities to exercise your design skills, and practice the material presented in class. More specifically, you are expected to implement and use a number of design patterns including the Singleton, the State, the Strategy, the Proxy, and the Observer design patterns.

Project Description

What is a publish-subscribe system

A publish-subscribe system (pub/sub system) is a system composed of *publishers* and *subscribers*. The publishers generate *events* and post them on a shared medium (i.e. the *channel*), where many subscribers have access to. The publishers do not know, and do not need to know, who the subscribers in the system are, and who consumes the published events. Publishers just generate events that are sent to a channel. For our project, the channel, on which a publisher publishes its messages is determined by the *strategy* the publisher is associated with. We consider that events have a type. Similarly, subscribers subscribe to specific channels, in order to get access to the events posted in those channels. A channel knows its subscribers and *notifies* them when a new event is *posted* in the channel. Upon reception of an event, subscribers may perform an action, depending on their *state.*
A good description of the pub/sub architectural style can be found at:
https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern, and
https://docs.oracle.com/cd/B10501_01/appdev.920/a96590/adg15pub.htm

The schema of a pub/sub system is as follows:



From: https://docs.oracle.com/cd/B10501_01/appdev.920/a96590/adg15pub.htm

System Actors and Entities

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.
Subscriber: 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.
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 channels in the system.
Event: An entity that denotes a piece of information posted on a channel. An event has an event ID, a reference to its publisher, and a payload which is an *EventMessage* object. The *EventMessage* has a header and a body.

Use Cases

UC1: Publishing an event

关于
This use case pertains to a publisher posting (i.e. publishing) an event on one or more channels. The publisher is associated with publishing strategies which encapsulate the logic that determines the channel or channels to which an event will be posted. Note that events can be of different types. They have a header and a body (i.e. payload), and are generated by an event factory. The publication of an event is performed via the publishing strategy, associated with a publisher, and can occur in two variations a) a publisher specifying which event to publish and consequently relying on the strategy to determine to which channel to post it to and b) a publisher relying on the strategy to both generate the appropriate event and decide to which channel to post it to. For this project, we have a configuration file named *strategies.str* that denotes which publisher is associated with which publishing strategy. The file is organized in tuples of the form <publisher-ID, strategy-ID>. Each line of this configuration file is one such tuple. The operation of publishing an event on a specific channel is handled by a Channel Event Dispatcher module which finally posts the event using the channel's interface. Note that *publisher-ID* and *strategy-ID* have implemented in the sample code as of type *int* (see Strategies.str file).

The precondition of this use case is that the publisher and the event must exist. If the channel the event is about to be published does not exist then it is created. The publishing occurs via a strategy each publisher is associated with. The strategy refers to channels by name.

The postcondition of this use case is that there should be an event in the queue of the specified channel, and the channel exists (either it is created as part of this event publication, or existed before).

UC2: Channel notifying subscribers of event been published

This use case deals with the situation where an event has just been posted to a channel and the subscribers of the channel need to be notified. 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 only if it is not blocked on this channel. In the implementation sample you have, all subscribers are notified.

The precondition of this use case is that the event must be posted on a channel, and the channel has subscribers.

The postcondition is that all subscribers on this channel which are not blocked, are notified.

UC3: Subscriber handling an event

This use case relates to a subscriber handling an event, after being notified by the channel that the event has been posted. The subscriber handles the event based on its current state, which means that, the same event may be handled differently by the same subscriber depending on its current state. For this project we have a configuration file named states.sts that denotes which subscriber is associated with which state. The file is organized in tuples of the form <subscriber-ID, state-ID>. Each line of this configuration file is one such tuple. Note that *subscriber-ID* and *state-ID* have implemented in the sample code as of type *int* (see States.sts file)

The precondition of this use case is that the subscriber must exist, have a state, and be a subscriber in the channel the event is posted.

The postcondition of the use case is that the subscriber handles the event according to the logic dictated by its state.

UC4: Controlling access to a channel

Controlling access to a channel is handled by a Channel Access Control Module, which provides services to block or unblock a subscriber, as well as to check whether a subscriber is blocked on a specific channel or not. We will assume that the decision to block or unblock a subscriber on a channel can be the responsibility of an Administration Server, which has references to the Channel Access Control Module, as well as the Channel Pool Manager Module for obtaining the list of all available channels and their subscribers. In the case of blocking a subscriber, if the subscriber does not exist the blocking returns null, otherwise returns true and adds the specified subscriber in the blocked list for this channel. In the case of un-blocking, if the subscriber does not exist the un-blocking returns null, otherwise returns true and the specified subscriber is removed from the blocked list of this channel.

The precondition of this use case is that the channel exists.

The postconditions are that the subscriber must be added in list of blocked subscribers of the specified channel (blocking case), or removed from the list blocked subscribers of the specified channel (un-blocking case).

UC5: Subscriber subscribes to a channel

A subscriber can subscribe or unsubscribe to a channel. A subscriber utilizes the services of the Subscription Manager Module, which handles the subscription or un-subscription request. In turn, the Subscription Manager utilizes the subscribe and unsubscribe services of the channel to perform the requested operation (subscribe, unsubscribe).

The precondition of the use case is that the channel exists.

The postcondition is that the subscriber is added in the list of subscribers for the given channel.

What to Implement

For this project you will implement the following:

I1: Implement the Event Factory and three or more Concrete Event Types

The implementation will be based on an EventFactory class that follows the Factory Method design pattern. You can see an example of implementation and use of the Factory Method pattern in the creation of states in the StateFactory class.

I2: Implement three or more Concrete States for subscribers

Implement three or more concrete states. The names of the states have to come from an enumeration (see Statename.java). Your new concrete states will have to implement the IState interface. You may want to alter the code given to you and use an AstractState class, instead of an IState interface, from which the concrete state classes will inherit from. Use your new states in your running demo to show that they handle events appropriately.

I3: Implement three or more Concrete Strategies for publishers

Implement three or more concrete states. The names of the states have to come from an enumeration (see StrategyName.java). Your new concrete states will have to implement the IStratergy interface. Use your new states in your running demo. You may want to alter the code given to you and use an AstractStrategy class instead of an IStrategy interface, from which the concrete strategy classes will inherit from.

I4: Implement the Singletons for:

Channel Access Control
Channel Creator
Channel Discovery
Channel Event Dispatcher
Subscription Manager

You already have an example of the Singleton design pattern available in the provided code for the ChannelPoolManager class. Your code should implement the *Singleton* design pattern.

I5: Notify subscriber only if the subscriber is not blocked in a Channel

Here you will prohibit a channel to notify a subscriber, if the subscriber is blocked on this channel. You can maintain a list of blocked subscribers in your program using a map, or another data structure. Such a map will associate (blocked) subscribers to channels. In this respect, the existence of an entry in this map or data structure denotes that the subscriber is blocked in this specific channel. You can add or delete entries in this map or data structure from your main program that drives the scenarios you will demonstrate the use of your program.

Project Deliverables

First Deliverable: Software Requirements Specification Document (SRS)

The Software Requirements Specification Document (SRS) will follow the template posted in the OWL site. The SRS is due on Friday February 8, 2019.

Your first deliverable is an SRS document that follows the template of the sample SRS posted in the course Web site. The SRS document will include the following as per the template:

1. Diagrams and specifications for the Use Cases discussed in Section 3.
- Use Case Descriptions for scenaria UC1-UC5
- Use Case diagrams for scenaria UC1-UC5
- Specification Level Collaboration Diagrams for scenaria UC1-UC5
- Sequencing Diagrams for scenaria UC1-UC5

2. Domain model class diagram
3. Project activities work plan (GANTT diagram) (see https://en.wikipedia.org/wiki/Gantt_chart and "*How to Create a Basic Gantt Chart in Microsoft Project 2016*" https://www.youtube.com/watch?v=J9uctgUaEic)
4.Non-functional requirements (as per the SRS template posted)

Please follow the SRS template to complete this deliverable.

Second Deliverable: Software Design Document (SDD)

The Software Design Document (SDD) will follow the template posted in the OWL site. The SDD will contain the component diagram and a description of the interfaces each component in the diagram. It will also contain a deployment diagram. The SDD is due on Monday March 8, 2019. Please use the SDD Template which will be provided. Make sure the components in the Architecture specification section include all APIs and Interfaces fully described. More details will be posted.

Third Deliverable Implementation

The implementation is due on Monday April 8, 2018.

Output

The output of your program in the console should print every important action that occurs in the system. These are:

- Creation of a channel: "Channel x created"
- Creation of a publisher: "Publisher x created"
- Association of a publisher to a strategy: "Publisher x has strategy y"
- Creation of a subscriber: "Subscriber x created"
- Association of a subscriber with a state: "Subscriber x is on state y"
- Subscription of a subscriber to a channel: "Subscriber x subscribes to channel y"
- Publication of an event by a publisher: "Publisher x publishes event y"
- Channel Dispatcher posts the event: "Channel x has event y from publisher x"
- Subscriber is notified and handles event: "Subscriber x receives event y and handles it at state z"
- Subscriber is blocked on a channel" "Subscriber x is blocked on channel y"

- Subscriber is un-blocked on a channel" "Subscriber x is un-blocked on channel y"

Testing

Your code should run properly the Functionality implemented in I1-I5. The key points are

- Posting of an event should be compliant with the publisher`s strategy
- Only non-blocked subscribers of a channel are notified
- Handling of an event should be compliant by the subscriber`s state

Non-Functional Requirements

NFR1: The system has to be able to handle at least 100 channels

NFR2: The system has to be able to handle at least 200 publishers and 200 subscribers at various configurations

Driving your Program

In order to execute your program, you will need to modify the Orchestration class and its main method, so that you can read a configuration file you will write in order to specify a given scenario. More specifically, after you load your Channels.chi, States.sts, and Strategies.str files where you create the channels, the publishers and the subscribers, you need to read a file (you will write) denoting a path of actions so that a specific scenario can be enacted (e.g. a sequence of publications, blockings, unblockings etc.). The file will have entries in each line of the form:

For publishing an event:

PUB <publisher-ID> <event-type> <event-message-header> <event-message-paylod>

PUB <publisher-ID>

For subscribing to a channel:

SUB <subscriber-ID> <channel-name>

For blocking a subscriber from a channel:

BLOCK <subscriber-ID> <channel-name>

UNBLOCK <subscriber-ID> <channel-name>

In this respect your file will look like (you do not need the comments in your file)

SUB 0 cars	// subscriber with ID *0* is subscribing to channel *cars*
PUB 0 TypeA h1 p1	// publisher with ID *0* publishes event with type *TypeA* and with
	// header *h1* and payload *p1*
BLOCK 0 cars	// Subscriber with ID *0* is blocked from channel *cars*

Notes

You are free to alter the code to add new design patterns if you wish, or keep the code as is as a base and implement on top of it.

Class Diagram of Sample Implementation

The class diagram of a sample implementation is provided below.