Concordia University

SOEN 6471 – Advanced software Architecture

*Professor Peter Rigby*

Facebook Chat Instant Messenger

Identification of Substantial Patterns

(FBCIM)

*Bala Arun Reddy Vatti 6228569 balaarunreddyv@gmail.com*

*Deepu Kumar 5972515 Kumar.deep@live.in*

*Pankaj Kapania 6249922 pankajkapania@yahoo.com*

*Paramjeet Singh 1370189 er.paramjeetsingh@yahoo.com*

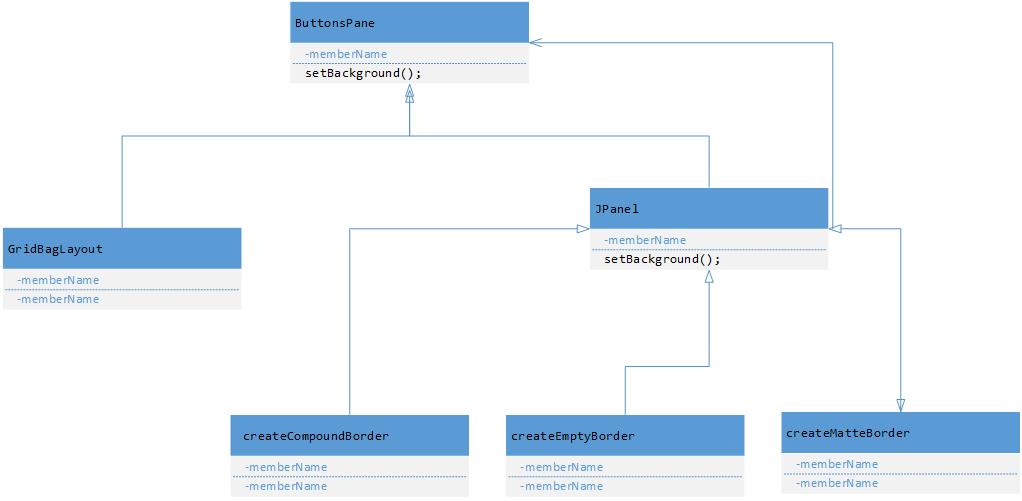
*Gilles Desrochers 1255185 gilles.desrochers@hotmail.com*

**Bala Arun Reddy Vatti - Decorator Design** **Pattern**

*“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Client-specified embellishment of a core object by recursively wrapping it. Wrapping a gift, putting it in a box, and wrapping the box.”*

**Reference:** [*http://sourcemaking.com/design\_patterns/decorator*](http://sourcemaking.com/design_patterns/decorator)

Diagram showing the interacting classes for the Decorator Pattern

****

Decorator Pattern implementation in the FBCIM application clearly goes with the way the design pattern is explained in the book referenced above. This pattern uses in part the **JPanel** java swings framework classes as its base.

**The decorator pattern are:**

**ButtonsPane – (Swing package)** Interface for objects that can have responsibilities added to them dynamically.

**GridBagLayout -** Defines an object to which additional responsibilities can be added.

**JPanel -** Maintains a reference to a Component object and defines an interface that conforms to the Component's interface.

**CreateCompoundBorder, CreateEmptyBorder CreateMatteBorder -** Concrete Decorators extend the functionality of the component by adding state or adding behavior.

Source code illustrating FBCIM’s implementation of the Decorator design pattern

/\*\*

\* Contains control buttons.

\*

\* **@author** Aleksey Prochukhan

\* **@version** 1.0

\*/

**private** **class** ButtonsPane **extends** JPanel {

/\*\*

\* Constructs buttons pane object.

\*/

ButtonsPane() {

**super**(**new** GridBagLayout());

setBackground(FBChatColors.DIALOG\_BUTTONS\_BG);

setBorder(

BorderFactory.createCompoundBorder(

BorderFactory.createMatteBorder(0, 1, 0, 0, FBChatColors.DIALOG\_BG),

BorderFactory.createCompoundBorder(

BorderFactory.createMatteBorder(1, 0, 0, 0,

FBChatColors.DIALOG\_BUTTONS\_TOP\_BORDER),

BorderFactory.createEmptyBorder(6, 0, 6, 0)

)

)

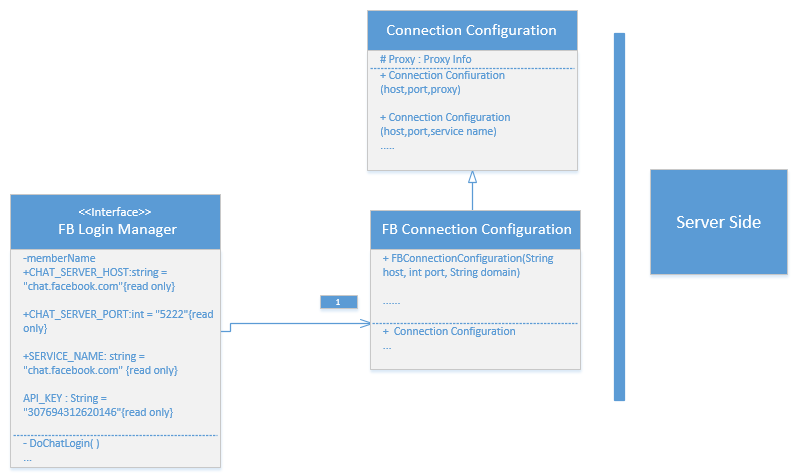
);

**Deepu Kumar - Remote Proxy Pattern**

The Remote proxy acts as a local representative to a remote object. What is a remote object? It could be an object that lives in the heap of a different java Virtual Machine or more often, an object that is running in a different address space. What is a local representative? It is a local object that you can call methods on, who in turn uses a remote object to achieve its goals. Your Client object performs as though it is making remote method calls. The proxy pretend to be the remote object, but it is just a stand in for the real thing.

**Reference**: O'REILLY Publications Head first, Source making.

**Implementation:** One use of the Remote Proxy pattern is identified. Here FB Login manager is acting as a client class and FB connection configuration is acting as a Client helper class. The Client side and server side are running on separate systems and are connected through a remote proxy. Client side includes client class and client helper while the server side has server helper server object. Server side exposes a few functions which are presented on the client side by the proxy. The UML diagram below illustrates the relation between client and server side, and is followed by the code implementation.



CODE:

**public** **class** FBConnectionConfiguration **extends** ConnectionConfiguration {

**public** FBConnectionConfiguration(String host, **int** port, String domain){

**super**(host, port, domain);

SASLAuthentication.registerSASLMechanism("X-FACEBOOK-PLATFORM",MySASLXFbkPlatformMechanism.**class**);

SASLAuthentication.supportSASLMechanism("X-FACEBOOK-PLATFORM", 0);

setSASLAuthenticationEnabled(**true**);

setRosterLoadedAtLogin (**true**);

}

}

**public** **class** FBLoginManager

**implements** FBLoginFrameListener {

**private** **static** **final** Logger LOG = Logger.getLogger(FBLoginManager.**class**.getName());

**enum** FBLoginState { LOGIN\_QUEUED, FB\_LOGIN\_STARTED, FB\_LOGIN\_SUCCESSFUL, CHAT\_LOGIN\_STARTED,

CHAT\_LOGIN\_SUCCESSFUL, LOGIN\_FAILED, LOGIN\_CANCELED,

}

/\*\* Target host. \*/

**public** **static** **final** String CHAT\_SERVER\_HOST = "chat.facebook.com";

/\*\* Target port. \*/

**public** **static** **final** **int** CHAT\_SERVER\_PORT = 5222;

/\*\* Service name. \*/

**public** **static** **final** String SERVICE\_NAME = "chat.facebook.com";

**public** **static** **final** String API\_KEY = "307694312620146";

/\*\* The app. context to work with. \*/

**private** FBChatContext context;

/\*\* Login manager listener. \*/

**private** FBLoginManagerListener listener;

/\*\* Keeps current login state. \*/

**private** FBLoginState state;

/\*\* Actual login frame with built in browser. \*/

**private** FBLoginFrame fbLoginFrame;

/\*\* <code>true</code> if silent login should be performed. \*/

**private** **boolean** silentLogin = **false**;

**public** FBLoginManager(FBChatContext context, FBLoginManagerListener l) {

// Save properties.

**this**.context = context;

**this**.listener = l;

// Construct facebook login frame.

**this**.fbLoginFrame = **new** FBLoginFrame(context, **this**);

// Set default login state.

setState(FBLoginState.LOGIN\_QUEUED);

}

**public** FBLoginFrame getLoginFrame() {

**return** fbLoginFrame;

}

**public** **void** setSilentLogin(**boolean** newVal) {

**this**.silentLogin = newVal;

}

**public** **void** login() {

**if** (isLoginInPogress()) {

**return**;

}

String oAuthToken = context.getSettings().getOAuthToken();

**if** (silentLogin && (oAuthToken != **null**) && !oAuthToken.equals("")) {

setState(FBLoginState.CHAT\_LOGIN\_STARTED);

} **else** {

setState(FBLoginState.FB\_LOGIN\_STARTED);

}

}

**private** **boolean** isLoginInPogress() {

**switch** (state) {

**case** FB\_LOGIN\_STARTED:

**case** FB\_LOGIN\_SUCCESSFUL:

**case** CHAT\_LOGIN\_STARTED:

**case** CHAT\_LOGIN\_SUCCESSFUL:

**return** **true**;

**default**:

**return** **false**;

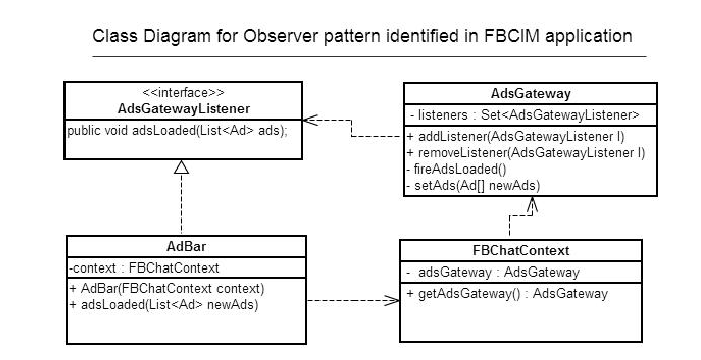
}

}

**Pankaj Kapania - Observer Design** **Pattern**

The “***Observer pattern*** *assumes that the object containing the data is separate from the objects that display the data, and that these display objects observe changes in data. When we implement observer pattern we refer to the data as the* ***subject*** *and each of the displays as* ***observers****. Each of these observers* ***registers*** *its interest in the data by calling a public method in the subject. Then each observer has a known* ***interface*** *that the* ***subject*** *calls when the* ***data changes*.”**

**Reference:** “*The Design Patterns Java Companion*” by **James W.Cooper** Addison Wesley Design Pattern Series.



The Observer Pattern implementation in the FBCIM application clearly goes with the way the design pattern is explained in the book mentioned above. Here the object that has the data is **ADsGateway** and the object that displays the data is **AdBar**. Both **AdsGateway** and **AdBar** are different objects with no knowledge of each other (as shown in the above class diagram) hence maintaining separation of concerns.

1. **AdsGatewayListener (interface):** This is an ***interface*** that exposes methods which must be implemented by the Observer (**AdBar)**. The Subject (**AdsGateway**) has knowledge of only **AdsGatewayListener** through which it calls the methods on **AdBar,** whenever ads are loaded from the server.
2. **AdsGateway (Subject):** This class plays the role of ***Subject*** in this implementation. It gets ads from the server and informs **observers** (**AdBar**) that ads have been loaded. **AdsGateway** does so by calling the **adsLoaded()** method exposed by the **AdsGatewayListener** interface and implemented by **AdBar**.
3. **FBChatContext (Middleman):**This object is the ***middle man*** that decouples **observers** from **subject**. It is not necessary to have such middle man while implementing observer pattern. In this implementation it is performing the role of middleman by exposing the **getAdsGateway()** method which **AdBar** calls when it needs to **register** or **unregister** with the **AdsGateway**.
4. **AdBar (Observer):** This object is an ***observer*** in the implementation of this pattern. It registers with the AdsGateway through the **context** object by calling the **addListener()** method**.**  It implements the **AdsGatewayListener** interface so that it may receive notifications (ultimately from AdsGateway) of loading of ads.

Source code illustrating FBCIM’s implementation of the Observer design pattern

**Interface:**

**public** **interface** **AdsGatewayListener** {

**public** **void** adsLoaded(List<Ad> ads);

}

**Subject:**

**public** **class** **AdsGateway** {

/\*\* The list of assigned listeners. \*/

**private** **final** Set<AdsGatewayListener> listeners;

**public void addListener(AdsGatewayListener l) {**

**synchronized** (listeners) {

listeners.add(l);

}

}

**public void removeListener(AdsGatewayListener l) {**

**synchronized** (listeners) {

listeners.remove(l);

}

**private void fireAdsLoaded() {**

**synchronized (listeners) {**

**for(AdsGatewayListener l : listeners) {**

**try** {

l.adsLoaded(ads);

} **catch** (Throwable t) {

}

**Observer:**

**public** **class** **AdBar** **extends** JPanel

**implements AdsGatewayListener {**

// Listen for ad updates.

context.getAdsGateway().addListener(this);

/\*\*

\* Invoked when the list of ads is updated.

\*/

**public** **synchronized** **void** adsLoaded(List<Ad> newAds) {

**Middle man**

**public** **class** **FBChatContext** {

**public** AdsGateway getAdsGateway() {

**return** adsGateway;

}

**Paramjeet Singh - Observer** **Pattern**

* **Observer Pattern:** This Pattern defines a one-to-many relationship between set of objects. When the state of one object changes, all of its dependents are notified. The subject and Observers define the one-to-many relationship. The Observers are dependent on the subject such that when the subject state changes, the observers get notified. Depending on the style of notification the observer may be updated with new values. There are many observers and they rely on the subject to tell them when its state changes. So there is a relationship between the ONE subject to MANY Observers. Using the observer pattern, a subject can register an unlimited number of observers. If a new listener wants to register with the subject, no code change in the subject is necessary. Using the listener pattern decouples the subject from its observers. Only the observers have direct knowledge about the subject.

**Reference**: Head First Design Patterns, O'REILLY Publications.

* **Project Implementation:** In the FBCIM application the Observer pattern was designed to implement many different kind of functionalities and the one I chosen is of the Sign out functionality in the application. The high level working is that there is a class named FBChatContext.java in which there is are two functions named addlistener(...) and removelistener(...). AddListener and RemoveListener functions are used to add and remove the Observers from the list. The parameter passed by the functions is the interface reference FBChatContextListener, and through polymorphism we can pass the observer. All the Listeners are added to the List of type FBChatContextListener. Here FBCIMApp.java implements the interface FBChatContextListener and adds itself to the listener list. The figure below shows the UML diagram detailing the relations and is followed by the Code implementing the same.



**CODE:**

**public class FBChatContext{**

**...**

/\*\* The list of registered listeners. \*/

**private** Set<FBChatContextListener> listeners;

/\*\* Constructor \*/

**public** FBChatContext(TrayNotifier trayNotifier, FBIcons fbIcons, FBCIMSettings settings) {

...

}

**public** **void** addListener(FBChatContextListener l) {

**synchronized** (listeners) {

**if** (!listeners.contains(l)) {

listeners.add(l); } }

}

**public** **void** removeListener(FBChatContextListener l) {

**synchronized** (listeners) {

listeners.remove(l);

}

}

**private** **void** fireUserSignedOut() {

**synchronized** (listeners) {

**for** (FBChatContextListener l : listeners) {

**try** {

l.userSignedOut();

} **catch** (Throwable t) {

*LOG*.log(Level.*SEVERE*, "Failed to notify listener that user signed out from chat!", t);} } } }

...}

**public** **interface** FBChatContextListener{

**public** **void** userAvatarLoaded();

**public** **void** userSignedOut();

**public** **void** presenceUpdated(Presence presence);

}

**public** **class** FBCIMApp **implements** FBChatContextListener

{ /\*\* The application context to work with. \*/

**private** FBChatContext context;

..

/\*\*\* Invoked when user signed out of facebook chat \*/

**public** **void** userSignedOut() {

initContext();

start(**null**);

}

**public** **void** initContext()

{

**this**.context = **new** FBChatContext(trayNotifier, fbIcons, settings);

**this**.context.addListener(**this**);

}**...**

**}**

**Gilles Desrochers - Observer Design** **Pattern**

The Observer pattern “defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” [<http://sourcemaking.com/design_patterns/observer>].

Diagram showing the interacting classes for the Observer pattern



The UpdateDownloader represents the model in the pattern. It maintains the list of listeners. In the course of execution of its tasks, UpdateDownloader‘s state will change, and it will notify listeners of these events. It does so through its “fireDownload” family of methods. Each of these methods will iterate through the list of listeners and call each listener’s appropriate method. For example, fireDownloadProgress will iterate through listeners and call each listener’s downloadProgress method.

The UpdateDownloader has no knowledge of the classes that implement UpdateDownloadListerner, hence the separation between the listeners and UpdateDownloader is maintained. UpdateDownloader is coupled only to the listener interface and is independent of classes that implement this interface. This design allows for the dynamic addition of numerous listeners to UpdateDownloader.

The FBCIM implementation of the Observer pattern however, varies slightly from the structure and examples presented at the sourcemaking website. The sourcemaking example has the Observer (or listener) derived classes add themselves to the Observer list (the Observer list is managed by the Subject class). This is done by calling the add functionality within the Observer’s constructor as follows:

**class** BinObserver **extends** Observer {

**public** BinObserver( Subject s ) {

s.attach( this ); } // Observers register themselves

**public** **void** update() {…}

}

Within FBCIM, UpdateManager is closely coupled to the UpdateDownloadDialog class and adds both itself and the UpdateDownloadDialog object to UpdateDownloader’s listeners.

The source code that follows illustrates the concepts discussed in this section.

Source code illustrating FBCIM’s implementation of the Observer design pattern

**public** **interface** UpdateDownloadListener {

**public** **void** downloadStarted();

**public** **void** downloadProgress(**long** downloaded, **long** total, **long** bytesPerSecond, **long** secondsLeft);

**public** **void** downloadCompleted(File f);

**public** **void** downloadFailed(Throwable t);

}

**public** **class** UpdateManager **implements** UpdateDownloadListener {

**private** UpdateDownloader updateDownload = **null**;

**private** **void** downloadUpdate(String url) {

…

updateDownload = **new** UpdateDownloader(url, updateFile);

UpdateDownloadDialog updateProgressDialog = **new** UpdateDownloadDialog(loginFrame);

// Attach listeners.

updateDownload.addListener(updateProgressDialog);

updateDownload.addListener(**this**);

…

}

**public** **void** downloadStarted() {}

**public** **void** downloadProgress(**long** downloaded, **long** total, **long** bytesPerSecond, **long** secondsLeft) {}

**public** **void** downloadFailed(Throwable t) {}

**public** **void** downloadCompleted(File f) {…}

…

}

**public** **class** UpdateDownloadDialog **extends** CustomDialog **implements** UpdateDownloadListener {

**public** UpdateDownloadDialog(Frame owner) {}

**public** **void** downloadStarted() {…}

**public** **void** downloadProgress(**long** downloaded, **long** total, **long** bytesPerSecond, **long** secondsLeft) {…}

**public** **void** downloadCompleted(File f) {…}

**public** **void** downloadFailed(Throwable t) {…}

}

**public** **class** UpdateDownloader **extends** Thread {

/\*\* List of registered listeners. \*/

**private** Set<UpdateDownloadListener> listeners;

UpdateDownloader(String url, File f) {

…

**this**.listeners = Collections.*synchronizedSet*(**new** HashSet<UpdateDownloadListener>());

}

**public** **void** addListener(UpdateDownloadListener l) {

**synchronized** (listeners) {

listeners.add(l);

}

}

**public** **void** fireDownloadStarted() {

**synchronized** (listeners) {

**for** (UpdateDownloadListener l : listeners) {

**try** {

l.downloadStarted();

} **catch** (Throwable t) {

*LOG*.log(Level.*SEVERE*, "Failed to notify listener that download has been started!", t);

}

}

}

}

**public** **void** fireDownloadProgress() {…}

**public** **void** fireDownloadCompleted() {…}

**public** **void** fireDownloadFailed(Throwable error) {…}

**public** **void** run() {…}