Concordia University

SOEN 6471 – Advanced software Architecture

*Professor Peter Rigby*

Final Report on

Facebook Chat Instant Messenger

(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*

Table of Contents:

[Project Description 3](#_Toc353724356)

[A general overview of the project and why it interests us: 3](#_Toc353724357)

[Maturity of the project (e.g., alpha, stable), developers are listed as working on the project 3](#_Toc353724358)

[Project Domain 3](#_Toc353724359)

[Project Size and Scope 4](#_Toc353724360)

[Project Size: Physical SLOC 4](#_Toc353724361)

[Is the project of reasonable size? 4](#_Toc353724362)

[Execution of the program/application 4](#_Toc353724363)

[Group Members 5](#_Toc353724364)

[Personas, Stakeholders, and Actors of FBCIM 6](#_Toc353724365)

[Persona 6](#_Toc353724366)

[Stakeholders 6](#_Toc353724367)

[Actors 7](#_Toc353724368)

[Fully dressed Uses Cases 8](#_Toc353724369)

[UML Diagram 11](#_Toc353724370)

[Class Diagram of Actual System 12](#_Toc353724371)

[Comparison of Domain Model (DM) to Class Diagram (CD) 13](#_Toc353724372)

[Refactoring 17](#_Toc353724373)

[Source code and Suggested Refactoring 21](#_Toc353724374)

[Explanation on Suggested Refactoring 22](#_Toc353724375)

[Identify a Substantial Patterns 23](#_Toc353724376)

[Refactoring Implementation 33](#_Toc353724377)

[Appendices 35](#_Toc353724378)

# Project Description

## A general overview of the project and why it interests us:

Facebook chat IM (FBCIM) is a free and open source desktop instant messenger (IM) application that allows chatting with all Facebook contacts in real time with its unique, swift and easy to use tabbed message window interface. You can see and respond to chats right from your desktop. Whether you're browsing other websites or using another app, you don't have to click away to stay connected. It supports the open-source XMPP protocol, connects primarily between Facebook users and synchronizes with the website. It’s written in Java and released under the GNU LGPL.

The tabbed interface supports chatting with various friends at once. You can minimize the application and be notified with instant alerts even if your web browser is not open in Facebook. The application will require you to disclose your location when sending messages. FBCIM currently supports all Windows operating systems (including Windows 98, SE, ME, server, XP, Vista, 7 and Metro).

FBCIM is simple and convenient to use and offers many great features such as emoticons for Facebook, text formatting, status update notification, sound alerts, chat history and many more. With FBCIM you'll never miss a friend and you'll have more fun while chatting with them. This innovative instant messaging application, which connects to Facebook Chat, provides the convenience and ease of an instant messenger. This novel application caught our attention and has a potential for continued growth and use, that is why we are interested and have selected FBCIM for our project.

## Maturity of the project (e.g., alpha, stable), developers are listed as working on the project

Currently FBCIM is a stable release (the latest release is 2.1.4651 according to its website (<http://www.filehorse.com/download-facebook-messenger/> ) and is in widespread use over globe. Throughout the entire period of development, the project has been consistent in terms of the number of commits. The last commit was made in Jan 2013. This project contains approximately 7071 SLOC.

## Project Domain

FBCIM has emerged from the realm of Instant Messaging which is a form of synchronous communication over the internet. Instant Messaging offers quick transmissions of text messages from sender to receiver by offering certain advantages over e-mail based communication. Internet messaging applications are not limited to messaging but it further allows functions such as file transfer, maintaining a contact list, and simultaneous conversations. Various instant messaging protocols, both open source and proprietary, are available. These include SIP, APEX, and XMPP.

FBCIM is built using open source Extensible Messaging and Presence Protocol (XMPP). FBCIM allows you keep a list of contacts you interact with. You can interact with anyone on your friend list or contact list as long as that person is online. You can also retrieve history of conversations.

# Project Size and Scope

## Project Size: Physical SLOC

In FBCIM we have a total of **8** Packages, a total **69** classes and the Physical SLOC is **7071**. This project is built using **Smack,** which is an open-source Java library for **XMPP** instant messaging. It also uses other frameworks such as **Xstream** (a simple library to serialize objects to XML and back again), **Google gson** and **restfb**.

## Is the project of reasonable size?

We believe that our project is of reasonable size given the number of packages (8) and number of classes (69) it has. The application builds on the aforementioned frameworks. The following graph depicts the SLOC distribution among packages. It is evident from the graph that the UI package has the most percentage of SLOC. Components like Chat, Update, and Core are relatively of the same size. All in all, a manageable project for a team of 5 as far as exploring architecture and performing refactoring is concerned.

## Execution of the program/application

We have compiled and assessed this project and believe it to be of a reasonable size for a term project.

# Group Members

**Bala Arun Reddy**

I have been developing various chatting/messenger application for many year. I developed a chatting system which can be used via local LAN/W-LAN suing Java Applets; Working on a desktop application which has billion of user would improve my skill to enhance to the next level. I will observe, learn, experiment, re-factor the architectural design of the application which help me a lot in the real world for developing more efficient, enhanced, reliable and user friendly applications, I believe that's the main intent of this course as-well.

**Deepu Kumar**

I am indulging with software development field since two years only and I mostly work with object-oriented programming languages, however I have worked on IPhone application based on my academic projects so I am aware of xcode as well. Therefore I am familiar with designing an architecture and implementation of few design patterns with xcode. I firmly believe that these skills will help me in understanding FBCIM’s current architecture and I’ll be able to contribute by helping understand the code architecture and behavior. Apart from this I would really go in depth of re-factoring in context with this project.

**Paramjeet Singh**

I have worked as software professional at various positions for around 5 years. I worked in PLM Domain and has been working on development of PLM solutions and also providing consulting to various clients in Europe and North America. During my career path I have never been involved in the team working on the architecture of the software; most probably I was fresh in my career. As now my career has grown much long ahead, it would be great to have hands on by working on domain layer, refactoring, design patterns and other vital software architecture concepts. This project on which we would be going to work is based on Communication domain thus giving me great opportunity to create new skills.

**Pankaj Kapania**

I have been in software development for more than 5 years now. During this tenure, I worked in various web development projects using J2EE technologies and other related frameworks like struts and hibernate etc. I have particular interest in working on the domain layer. During all these years I did not get much to work on desktop based application. So I think this is my opportunity. In this project I would be striving for identifying the architecture specifically at the domain layer consisting of the core business logic. Also, I would strive for applying the re-factoring that we will be learning in this course.

**Gilles Desrochers**

The majority of the software development projects I have been involved in have been in C/C++ application. My expertise has been more on desktop application development which should prove useful to that portion of the project. My experience in C++ is adaptable to Object Oriented languages such as Java. I look forward to applying techniques to improve a software application’s quality and I anticipate expanding my knowledge of the communication domain.

# Personas, Stakeholders, and Actors of FBCIM

## Persona

Samantha is a young entrepreneur who just started a small software firm in Montreal, Canada. Her software firm builds mobile applications and intermediate level web applications. Samantha mostly designs applications and outsources development related work to overseas developers. She has a team of 5 overseas developers. Samantha and her development team communicate over the phone which is a costly medium to communicate especially if a call is initiated from developers to Samantha regarding development related work. Samantha and her team of developers are looking for an Internet messaging application that allows communication in an inexpensive and secure manner. An important aspect of their communication is that they want to record communications for later references. Another important concern is sometimes Samantha or developers would need an immediate answer rather than waiting for an answer over e-mail. More often than not Samantha is out of the office meeting new clients and sometimes in remote areas where her mobile service provider does support services. Yet she still needs to contact her employees in a remote and secure manner. Samantha and her developers are Facebook users. So a Facebook messaging application would be a perfect solution for them as they are looking for a free IM application. Also they would not have to go through the Facebook registration process as they are already current users. Samantha regularly connects with people while at the computer, usually through Facebook. She’ll get in touch with friends to make plans for the evening, or sometimes just to touch base with her partner to decide on supper. She’ll sometimes let everyone know what she’s up to by post images or text to her Facebook page.

## Stakeholders

3 stakeholder groups have been identified for FBICM.

* **Users (Clients):** Users are stakeholders who actually make use of the application. The Chat application is built to assist the clients communicate in a one-to-one or one-to-many framework.
* **Support Team:** This Stakeholder group’s implication is to provide assistance to the Users with the operation of the application. As part of support responsibilities, the Support Team logs deficiencies and pertinent information reported by users.
* **Developers (FBCIM organization):** These Stakeholders form the team that developers, owns and manages application. These stakeholders will make modifications to the application when appropriate and as such have direct relationship with the application. Since these stakeholders also are the Owners, any impact on the application will have an effect on them.

## Actors

3 different actors have been identified for FBCIM.

* **User:** a human actor who interacts with the FBCIM System. After receiving authentication through (or using) a Facebook account, the user can open a chat window with a specific friend present in the friend list. The user (sender) can then start sending messages to the friend (other user /receiver). Communication can be initiated with any other friend (user / receiver) who is online and the chat can continue for as long as the communication link is open.
* **Restfb:** This is non-human secondary actor that the FBCIM System connects to. This secondary actor allows FBCIM System to send information to the Facebook system. It is a low level HTTP based API that supports the FBCIM System to query existing data from the Facebook system. It also allows new data to be sent to the Facebook system from FBCIM System.
* **Facebook System**: This is another non-human secondary actor, which is the primary source of information. It holds user data. It restricts unauthorized access to user’s data by providing safety measures. Facebook System supports FBCIM System to operate and fulfill its function.

# Fully dressed Uses Cases

Fully Dressed Use Case: UC-001

|  |  |
| --- | --- |
| **ID** | UC-001 |
| **Use case** | Send a chat message to other user |
| **Description** | Use case describes the scenario where in user sends a chat message to other Facebook user |
| **Level** | User Goal Level |
| **Primary Actor** | User |
| **Supporting Actors** | Restfb, Facebook System |

**Pre-Conditions**

* The User should have registered with Facebook System.
* The User should have FBCIM installed on their local machine with internet access.

**Post Conditions**

**Success end condition**

* The message has been transferred to another Facebook user.

**Failure end condition:**

* In case of failure to connect to the network/server, a message will be displayed to the user.
* In case of any software error, the error is logged and transferred to the support team. If the support team is unable to rectify the error, the issue is passed on to the development team.

**Main Success Scenario**

1. The user launches the FBCIM client application.
2. The system prompts the FBCIM user to connect with the Facebook System.
3. The FBCIM user provides required credentials for login.
4. The system logs the user into his Facebook account. **[Alt course A]**
5. The system retrieves the user’s friend list and displays it.
6. The FBCIM user selects a friend to chat with.
7. The system displays the Chat message window.
8. The FBCIM User types a message to be sent.
9. The system forwards the message to the correct recipient. **[Alt course B]**
10. The recipient receives the message successfully.
11. Use case ends.

**Alternate course A:**

The customer is not logged-in into the system.

A.1 The system displays an error message related to the login.

A.2 The user follows the instructions provided.

A.3 The user is unable to login.

A.4 The system informs the user to contact the support team.

A.5 The use case ends.

**Alternate course B:**

The system is unable to deliver the message to the intended recipient.

B.1 The system is unable to deliver the message to the intended recipient.

B.2 The system informs the user about the message delivery failure.

B.3 The use case ends.

Fully Dressed Use Case: UC002

|  |  |
| --- | --- |
| **ID** | UC-002 |
| **Use case** | Loading FBCIM client on start up |
| **Description** | This use case guides the user through setting of automatic Loading of FBCIM client on start up. |
| **Level** | User Goal Level |
| **Primary Actor** | User |
| **Supporting Actors** | Restfb, Facebook System |

**Pre-Conditions**

* The client should have hardware adequate to run the FBCIM.
* The client should have network access for communication with Facebook for login.
* The hardware should support the OS required to run FBCIM.

**Post Conditions**

**Success end condition**

* System - FBCIM Client is loaded on start up.

**Failure end conditions**

* In case of failure to connect to the network/server, a message will be displayed to the user.
* In case of any software error during installation or launching of the software, the error is logged and transferred to the support team. If the support team is unable to rectify the error, the issue is passed on to the development team.

**Main Success Scenario**

1. The FBCIM User launches the FBCIM client application.
2. The system prompts the FBCIM user to connect with Facebook.
3. The FBCIM user provides the required credentials for login.
4. The system logs the user into his Facebook account. **[Alt course A]**
5. The FBCIM User selects the option to Load client on start up.
6. The System accepts the required change. **[Alt course B]**
7. The system saves the changes to the load configuration file.
8. The FBCIM user restarts the system.
9. The Operating System loads the newly updated configuration file.
10. The Operating System detects the new changes.
11. The FBCIM Client is loaded on start up.
12. Use case ends here.

**Alternate course A:**

The customer is not logged-in into the system.

A.1 The System provides the error message related to login.

A.2 The User follows the instructions provided.

A.3 The User is unable to login.

A.4 The system informs the user to contact the support team.

A.5 The use case ends.

**Alternate course B:**

B.1 The system detects an issue with the changes.

B.2 The System notifies the user about the problem and root cause.

B.3 The use case ends.

# UML Diagram

The figure below presents a Domain model for a simple chat application which makes use of a user’s Facebook account.



A Facebook user has a Facebook account. Within the account, the user maintains a personal profile and personalizes the account by configuring various parameters of the account.

A Facebook account includes a Wall on which the user may post various types of data including files, text, and images. The visibility and access of information are controlled by the user. The amount of data, or capacity of the account, is constrained by the Facebook account. Within the account, the Facebook user manages a list of Friends. Friends must also be Facebook users.

A communication device can be a computer, phone or other device with which the user may access his/her Facebook account. The Chat application uses the User’s login credential to access Account information. This information enables the application to (1) manage conversations between the user and Facebook friends and (2) post items to the account Wall.

# Class Diagram of Actual System

Facebook chat IM (FBCIM) is a free and open source desktop instant messenger (IM) application that interfaces with Facebook and provides a means for chatting with Facebook contacts. The application is composed of 69 classes. Figure 1 identifies the essential elements of the Facebook Chat Instant Messenger (FBCIM) application.



Figure 1 FBCIM Abbreviated Class Diagram

The application entry point is found in FBCIMApp. This object launches the application and creates the FBChatContext object. The core of the application lies in FBChatContext.

Only one instance of FBChatContext is instantiated. FBChatContext uses FBConnection and FBLoginManager to establish a connection to Facebook and retrieve its contacts list. FBChatContext then uses the services of FBChatManager for the creation of chats.

A chat, or FBChat, created by the FBChatManager is always paired up with an FBContact. A chat has a reference to its related contact. Whenever FBChatContext recieves a chat object from the FBChatManager, it uses an FBContactList to manage its chat and contact pairs.

## Comparison of Domain Model (DM) to Class Diagram (CD)

Figure 2 Domain Model and Class Diagram

Similarities and Differences

There are several common elements between the two models. The CD:FBContact is equivalent to the DM:Friend element. Both models make use of Login services and Connection/ Communication and associated properties. The primary topic of the application, the Chat, is also represented in both models. The CD:FBContactList finds its equivalent in the DM:FacebookAccount’s association to DM:Friends.

The differences manifest themselves in various ways. The Facebook domain elements such as the DM:Wall are not found in the CD. These elements are accessed by the FBChatContext through the connection to Facebook and this function is not extracted to a specialized component.

The CD makes use of Pure Fabrications, not found in the DM, such as the FBChatManager for the construction of chats.

Another subtle difference is in the use of contacts. The DM classified all Friends and the Facebook account owner as Facebook Users. The CD however makes a distinction between the two. The owner’s information is maintained within the CD:FBChatContext while friends are identified as CD:FBContact. The owner is not an FBContact.

In terms of Architecture, the differences between the Domain Model and the Class Diagram are likely the result of design decisions made as the project evolved. The Wall and the Postings found in the Domain Model are elements that may fit well conceptually but are difficult to implement in that framework. As such File Transfer and Facebook Status Update are implemented independently. A significant difference between the Domain Model and the Class Diagram is the Class Diagram’s FBChatContext. The FBChatContext is a dominant component of the architecture and overly so. Its disparity from the Domain Model mark it as a prime element for review in analysis and rework of the architecture.

Tools

Enterprise Architect (Sparx Systems) was used for the generation of the overall class diagram (please refer to Class Diagram in Appendix), which proved invaluable for identifying relationships between objects. The tools recognized most relationships but failed to identify links to classes used by generic type declarations. Visio (Microsoft) was used for the less comprehensive diagrams presented in this section. The online tool Gliffy ([www.gliffy.com](http://www.gliffy.com) ) was used for the partial class diagrams (Figure 3 and Figure 4).

Sample Classes

The method and attributes declarations for FBChatContext and FBChatManager are presented below, including a brief discussion of relationships between the two classes. The relationship between the two classes is also illustrated in Figure 1 and Figure 2.

**public class** FBChatContext {

…

**private** FBContactList contactList;

**private** FBChatManager chatManager;

…

//Sets facebook chat connection.

**public** **void** setXmppConnection(XMPPConnection newVal)

**throws** Throwable {

…

**this**.chatManager = **new** FBChatManager(**this**);

contactList.addContactListListener(**this**.chatManager);

…

}

**public** FBChat chat(FBContact targetContact) {

FBChat chat = chatManager.getChat(targetContact);

**return** chat;

}

**public** FBContact getFBContact(String contactId) {

**return** contactList.getFBContact(contactId);

}

**private** **class** SaveChatHistoryTask **extends** TimerTask {

…

@Override

**public** **void** run() {

chatManager.saveChatHistory();

}

}

**public** **class** FBChatManager {

**implements** FBContactListListener {

**private** **final** Map<FBContact, FBChat> chatMap;

**private** FBChatContext context;

**private** FBChatHistoryArchive archive;

**public** FBChatManager(FBChatContext context) {

**this**.context = context;

**this**.archive = **new** FBChatHistoryArchive(context.getArchiveDir());

**this**.chatMap = Collections.*synchronizedMap*(**new** HashMap<FBContact, FBChat>());

}

**public** FBChat getChat(FBContact contact) {

FBChat chat;

**synchronized** (chatMap) {

chat = chatMap.get(contact);

**if** (chat == **null**) {

chat = **new** FBChat(context, contact);

**if** (context.getSettings().getSaveHistory()) {

chat.setHistory(archive.getHistory(contact));

}

chatMap.put(contact, chat);

}

}

**return** chat;

}

**public** **void** saveChatHistory() {

…

**synchronized** (chatMap) {

**for** (FBChat chat : chatMap.values()) {

archive.saveHistory(chat.getContact(), chat.getHistory());

}

}

}

**public** **void** updateGamesBarVisibility() {

**boolean** gamesBarVisible = context.getSettings().getDisplayGamesBar();

**synchronized** (chatMap) {

**for** (FBChat chat : chatMap.values()) {

chat.getChatFrame().setGamesBarVisible(gamesBarVisible);

}

}

}

As evidence by the code sample, the FBChatContext creates the FBChatManager, and each object has a reference to the other.

The FBChatContext manages a list of FBContacts. With its FBContacts, the FBChatContext uses the FBChatManager to get Chats. The FBChatContext also uses the FBChatManager’s service to save chat history.

The FBChatManager needs the FBChatContext for creating / retrieving Chats (getChat() method). This is an interesting twist since it’s the FBChatContext that makes the call to the FBChatManager:getChat() method, and could pass itself as input rather than relying on the FBChatManager private reference to the FBChatContext. The context is also used by FBChatManager for updating the user interface, a functionality that likely should be extracted to a user interface related class.**Code Smell**

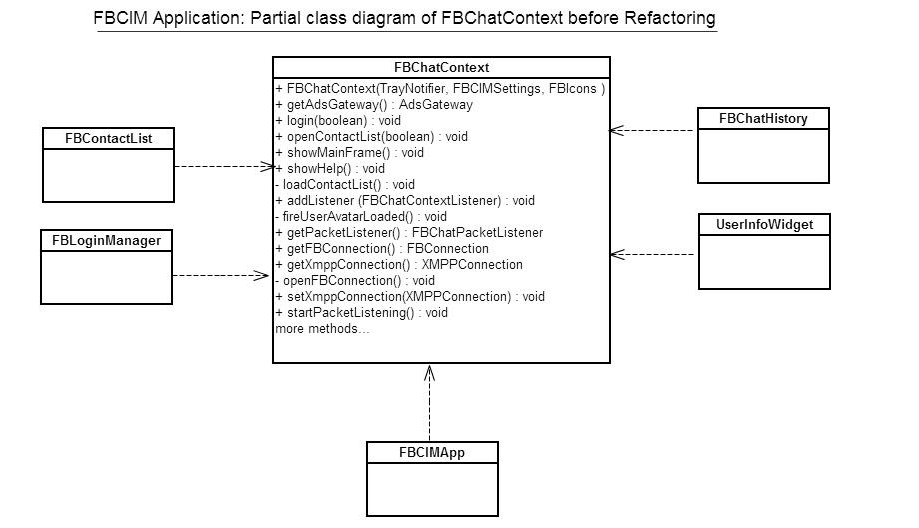


Figure 3: Partial class diagram of FBChatContext before refactoring.

After taking a printout of whole project and collating into a 34” by 33” page, it became clearer that class FBChatContext is doing more than it should. It has been assigned with multiple responsibilities. Responsibilities such as creation of connections, initializing the application context, and tracking and modifying the user status are a few of the many key responsibilities. Multiple responsibilities make FBChatContext less cohesive. In order to fulfill the aforementioned responsibilities, FBChatContext has been coupled with a high number of classes.

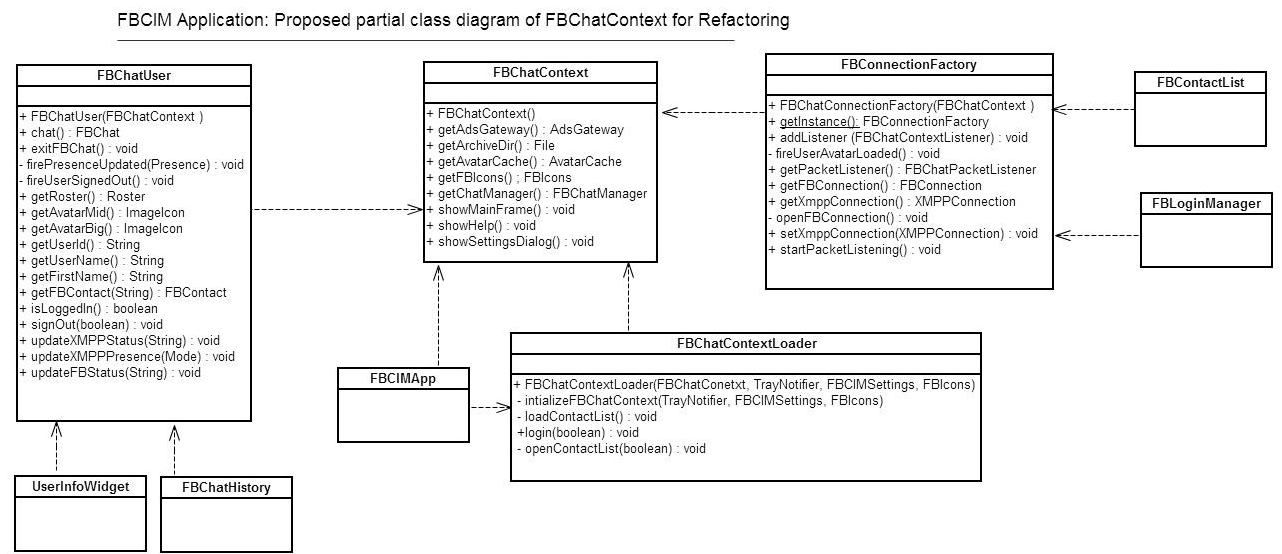
The following are q few noticeable points that make FBChatContext a first choice to re-factor:

1. FBChatContext has a Physical SLOC around 600 and a Logical SLOC around 320.
2. FBChatContext has been coupled to 20 classes (from a total of 69 classes) which accounts for 29% of the software.
3. FBChatContext contains 41 methods (including parameterized constructor) and a few of them are excessively long (e.g. signOut spans 60 lines).
4. FBChatContext has methods to connect with the FaceBook system, to initialize the application context, to track and modify information and status pertaining to currently logged-in user and to provide central access of related information to other objects of the FBCIM application. All these responsibilities make FBChatContext less cohesive and highly coupled class.

**Code Smells** from FBChatContext are as follows.

1. **Large Class:** Aforementioned discussion on more than one responsibility fulfilled by FBChatContext indicates Large Class code smell.
2. **Feature Envy:** Methods named showTrayPopup(FBContact contact)and getAdsGateway()smell of Feature envy as it may makes more sense to have these methods are in class FBContactList.
3. **Long Method:** Methods including (but not limited to) setXmppConnection(XMPPConnection newVal)and signOut(boolean notify)are examples of long methods in the FBChatContext class. It is also important to note that the parameterized constructor of FBChatContext, [FBChatContext(TrayNotifier trayNotifier, FBIcons fbIcons, FBCIMSettings settings)] is a Long Method offender. It creates an inner class which in turn implements a listener interface, implements 3 methods, and use 3 different private classes. In all the FBChatContext constructor spans 40 lines of code for unrelated responsibilities.
4. **Data Clumps:** Fields (instance variables) including (but not limited to) roster, vCard, avatarMid, avatarBig are an examples of data clumps. In an effort to bring higher cohesion to the system, functionality could be refactored around data clumps, regrouping related methods to new classes (making use of the Move Method refactoring technique).

# Refactoring



**Figure 4: Proposed Partial class diagram of FBChatContext for refactoring.**

Our prime focus is to increase cohesion and lower coupling of FBChatContext. We can achieve this by applying refactoring techniques such as **Extract Method**, **Move Method**, **Extract Class** and **Rename Method** to the aforementioned code smells. In applying those techniques, we can move the responsibilities of FBChatContext to 3 new classes, reducing dependencies between FBChatContext and other software classes and thereby lowering coupling. The introduction of 3 new classes will also increase cohesion. We will now describe the new class diagram of the FBCIM architecture (partial) in the context of FBChatContext.

1. **FBChatContextLoader:** FBChatContext class is the core class as it initializes the application context. FBCIMApp calls the parameterized constructor of the FBChatContext class to create its instance. FBChatContext(TrayNotifier trayNotifier, FBIcons fbIcons, FBCIMSettings settings) does more than it should, as already explained in the previous section. We can move that responsibility of initializing FBChatContext to FBChatContextLoader. A step by step explanation of the refactoring techniques required to construct FBChatContextLoader are now presented.
2. Moved the constructor code into new method named initializeFBChatContext()(**Extract Method**).
3. Created a new Class FBChatContextLoader (**Extract Class)**.
4. Copy methods initializeFBChatContact(), loadContactlist(), openContactList(boolean showMainFrame) into the new class FBChatContextLoader (**Move Method)**. It is important to note that 3 private classes also need to be moved from FBChatContext to FBChatContextLoader as they were solely used in the application initialization. Those classes were originally part of the FBChatContext java file.
5. Modify the references in FBCIMApp to initialize FBChatContext via FBChatContextLoader. The following is the code that implements that modification.

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

**After Refactoring:** ctxIntializer = new FBChatContextLoader(context, trayNotifier, fbIcons, settings);

1. Test the code after building the new jar through build-fbcim.xml file.
2. Remove the following methods from original class FBChatContext initializeFBChatContact(), loadContactlist(), openContactList(boolean showMainFrame).
3. Move method named login(boolean silentLogin)from FBChatContext to FBChatContextLoader to remove **Feature Envy** code smell (**Move Method)**.
4. Test the code once again to test changes mentioned in step g).

Note that class FBChatContextLoader is a candidate for refactoring techniques like Extract method and Move method but this will not be addressed at this time as our primary focus is the FBChatContext class. Also, since FBChatContext class has no dependency on FBChatContextLoader class, refactoring FBChatContextLoader will not benefit FBChatContext.

1. **FBConnectionFactory:** FBChatContext class was also responsible for creating connection objects required for interaction with the Facebook System. Refactoring techniques and the Factory Pattern will allow moving this responsibility to a new class we name FBConnectionFactory. FBLoginManager and FBContactList are coupled with FBChatContext to retrieve XMPPConnection and FBConnection respectively. Generation of FBChatConnectionFactory allows us to move these dependencies from FBChatContext to FBConnectionFactory. This reduces coupling and increases cohesion for FBChatContext. The Refactoring techniques applied to move the mentioned responsibilities to the newly created FBConnectionFactory are explained:
2. Created a new class named FBConnectionFactory(**Extract Class**).
3. Copy named methods including (but not limited to) setXmppConnection(XMPPConnection newVal), FBConnection getFBConnection(),public XMPPConnection getXmppConnection(). into the new class (**Move Method)**.
4. In FBLoginManager and FBContactList, replace references to FBChatContext with FBConnectionFactory.
5. Test the code after building the new jar with build-fbcim.xml file.
6. Remove methods copied from FBChatContext.
7. Test the code for the changes made in step e).
8. **FBChatUser:** The FBChatContext class was also responsible for keeping track of information and status pertaining to logged-in user. Refactoring techniques allow moving this responsibility to a new class, FBChatUser. FBChatHistroy, UserInfoWidget and TrayPopUpMenu are examples of a few classes that have been coupled with FBChatContext to retrieve user information through methods like getUserId(), getUserName() and getFirstName(). Generation of FBChatUser allows us to move these methods (thus dependencies) from FBChatContext to FBChatUser, hence reducing coupling and increasing cohesion for FBChatContext. Refactoring techniques applied are explained below in **step by step** instructions for creating FBChatUser.
9. Creat a new class named FBChatUser(**Extract Class**).
10. Copy methods named above, including but not limited to, getUserId(), getUserName() and getFirstName() into the new class named FBChatUser (**Move Method)**.
11. Change references of FBChatContext to FBConnectionFactory inside FBChatHistroy, UserInfoWidget and TrayPopUpMenu.
12. Test the code after building the new jar with build-fbcim.xml file.
13. Remove code of methods copied from FBChatContext.
14. **FBChatContext:** FBChatContext class now has a relatively low number of methods and more importantly, the remaining fit better with the intent of the class. The remaining methods include public AvatarCache getAvatarCache(), public FBIcons getFBIcons(), public File getArchiveDir(), public FBCIMSettings getSettings(). By their very nature, these methods should be in a class that is central to the application and offer their services available from a single point of contact.

# Source code and Suggested Refactoring

**public** **class** FBCIMApp

**implements** FBChatContextListener {

// ……………………… code removed due to limited space……………………………………

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

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

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

}

}

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

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

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

**this**.trayNotifier = trayNotifier;

**this**.fbIcons = fbIcons;

**this**.settings = settings;

**this**.trayNotifier.setContext(**this**);

// ……………………… code removed due to limited space……………………………………

**this**.fbChatMainFrame = **new** FBChatMainFrame(**this**);

**this**.fbLoginManager = **new** FBLoginManager(**this**, **new** FBLoginManagerListener() {

**public** **void** loginSuccessful(**boolean** showMainFrame) {

openContactList(showMainFrame);

loadContactList();

}

// ……………………… code removed due to limited space……………………………………

**public** **void** loginCanceled() {

**new** MessageDialog(**null**, "Error", "Login to Facebook chat canceled!", "OK").showMessageDialog();

signOut(**true**);

}

}

**public** **void** login(**boolean** silentLogin) {

fbLoginManager.setSilentLogin(silentLogin);

fbLoginManager.login();

}

**public** **void** openContactList(**boolean** showMainFrame) {

// ……………………… code removed due to limited space……………………………………

}

**private** **void** loadContactList() {

// ……………………… code removed due to limited space……………………………………

**this**.serviceTimer.schedule(**new** SaveChatHistoryTask(),*HISTORY\_SAVE\_INTERVAL*,*STORY\_SAVE\_INTERVAL*);

**this**.serviceTimer.schedule(**new** UpdateAdsTask(), *UPDATE\_ADS\_INTERVAL*, *UPDATE\_ADS\_INTERVAL*);

}

}

}

**private** **class** UpdateFBStatusTask **extends** TimerTask {

**public** **void** run() {

contactList.updateContactsFBStatus(skipOffline);

**if** (!skipOffline) {

skipOffline = **true**;

}

}

}

Note: Apart from FBCIMApp and FBChatContext there are 3 other private classes that are being instantiated and referred solely in constructor method. Only one private class has been shown due to limited space.

# Explanation on Suggested Refactoring

The FBChatContext class is the core class as it initializes the application context. FBCIMApp calls the parameterized constructor of FBChatContext class to create its instance. FBChatContext(TrayNotifier trayNotifier, FBIcons fbIcons, FBCIMSettings settings) does more than it should as already explained in the previous section. We can move the responsibility of initializing **FBChatContext** to **FBChatContextLoader**. Continuing the refactoring effort and moving additional responsibilities out of **FBChatContext** to **FBConnectionFactory** and **FBChatUser** further improves the application’s architecture.

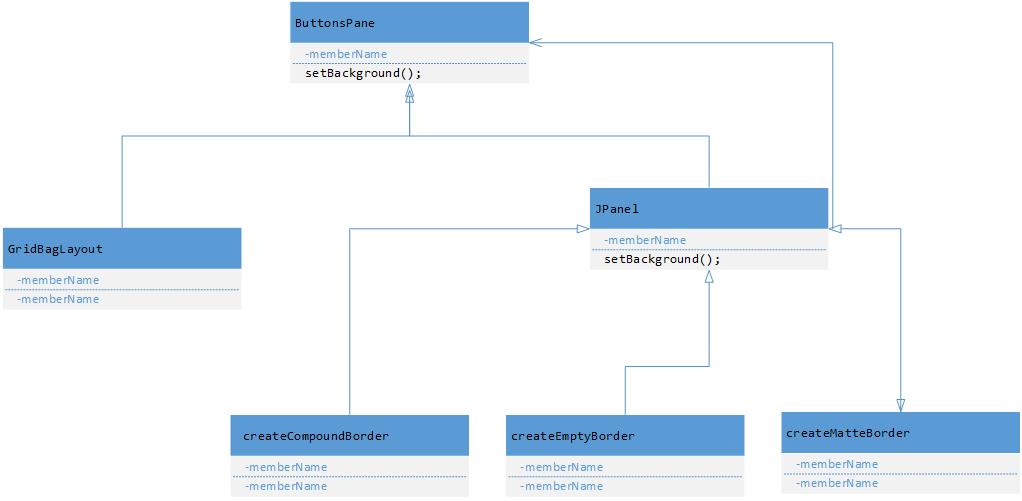
# Identification of Substantial Patterns

**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)

)

)

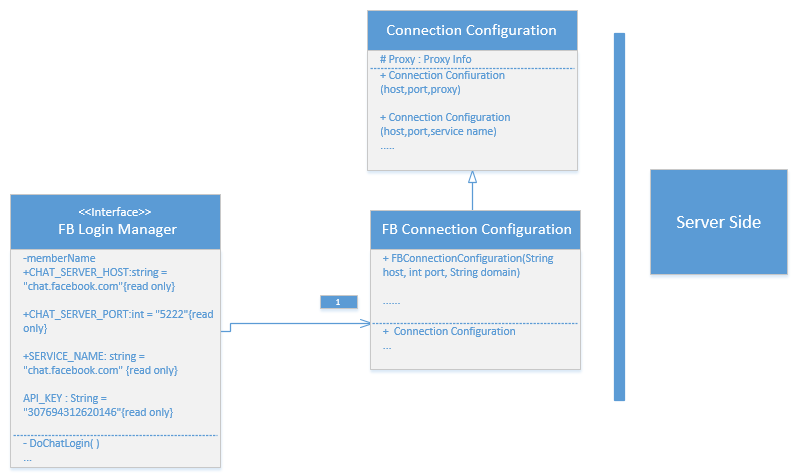
);

**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**;

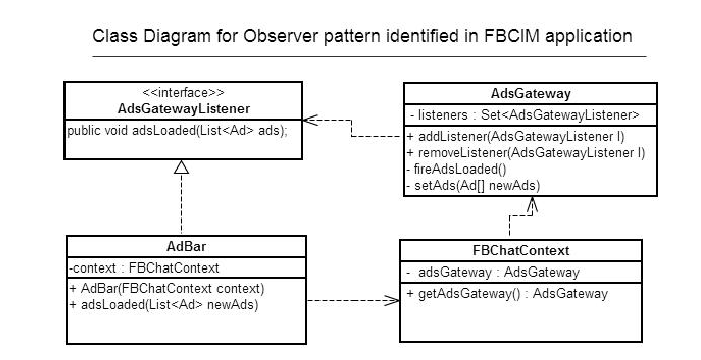
}

}

**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() {…}

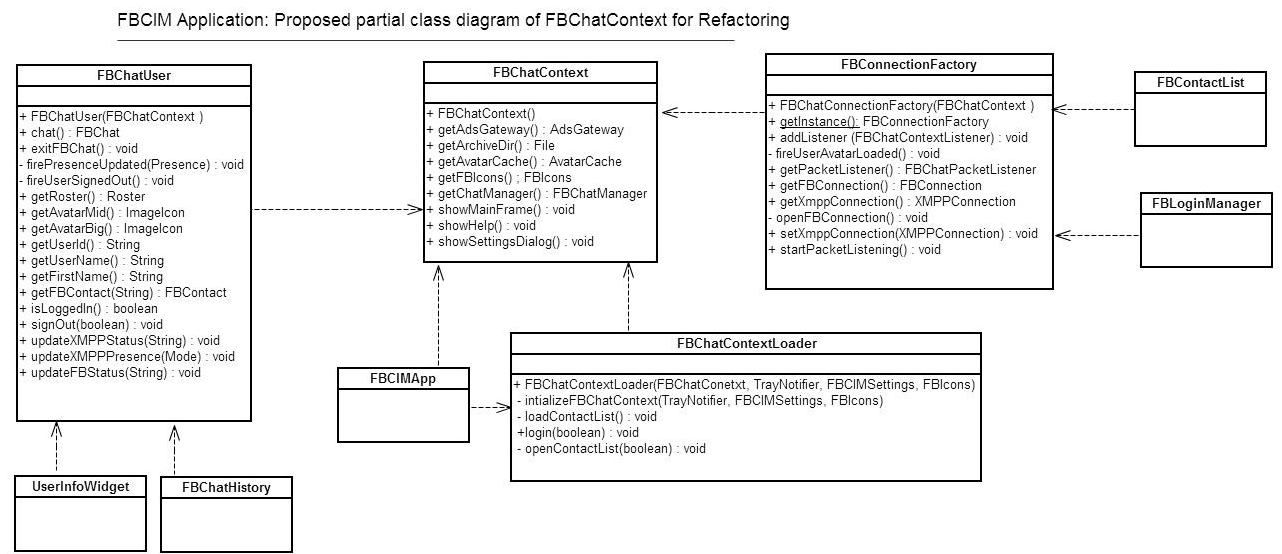
**public** **void** fireDownloadFailed(Throwable error) {…}

**public** **void** run() {…}

}

# Refactoring Implementation

The previous deliverable outlined a refactoring of the class FBChatContext which lead to the creation of three new classes. The proposed partial class diagram is presented below.



The suggested refactoring required to achieve the proposed design is as follows:

1. **FBChatContextLoader**
2. Moved the constructor code into new method named initializeFBChatContact()(**Extract Method**).
3. Created a new Class FBChatContextLoader (**Extract Class)**.
4. Copy methods initializeFBChatContact(), loadContactlist(), openContactList(boolean showMainFrame) into the new class FBChatContextLoader (**Move Method)**. It is important to note that 3 private classes also need to be moved from FBChatContext to FBChatContextLoader as they were solely used in the application initialization. Those classes were originally part of the FBChatContext java file.
5. Modify the references in FBCIMApp to initialize FBChatContext via FBChatContextLoader
6. Test the code after building the new jar through build-fbcim.xml file.
7. Remove the following methods from original class FBChatContext initializeFBChatContact(), loadContactlist(), openContactList(boolean showMainFrame).
8. Move method named login(boolean silentLogin)from FBChatContext to FBChatContextLoader to remove **Feature Envy** code smell (**Move Method)**.
9. Test the code once again to test changes mentioned in step g).
10. **FBConnectionFactory**
11. Created a new class named FBConnectionFactory(**Extract Class**).
12. Copy named methods including (but not limited to) setXmppConnection(XMPPConnection newVal), FBConnection getFBConnection(),public XMPPConnection getXmppConnection(). into the new class (**Move Method)**.
13. In FBLoginManager and FBContactList, replace references to FBChatContext with FBConnectionFactory.
14. Test the code after building the new jar with build-fbcim.xml file.
15. Remove methods copied from FBChatContext.
16. Test the code for the changes made in step e).
17. **FBChatUser**
18. Creat a new class named FBChatUser(**Extract Class**).
19. Copy methods named above, including but not limited to, getUserId(), getUserName() and getFirstName() into the new class named FBChatUser (**Move Method)**.
20. Change references of FBChatContext to FBConnectionFactory inside FBChatHistroy, UserInfoWidget and TrayPopUpMenu.
21. Test the code after building the new jar with build-fbcim.xml file.
22. Remove code of methods copied from FBChatContext.
23. **FBChatContext:** The remaining methods public AvatarCache getAvatarCache(), public FBIcons getFBIcons(), public File getArchiveDir(), public FBCIMSettings getSettings() should be in a class that is central to the application and offer their services available from a single point of contact.

Attached to this document is a series of diff files, detailing the work done in the initial coding exercise for the implementation of this design.

Patchsets 1-x.patch detail the coding carried out in the development of the FBChatContextLoader. The initial code created the class **FBChatContextLoader** as a singleton class. This seemed logical since there is only one context used by the application. However, due to the widespread use of the context within the application, this step had not been thought through, was premature and eventually removed. The remainder of the refactoring was executed as planned.

Patchsets 2-x.patch initiated the second refactoring identified in the redesign. The **FBConnectionFactory** class was created and completed but not fully integrated into the application.

This completes the refactoring implemented thus far in FBCIM. The patchsets mentioned above can be found in the zip file (Patchset.zip) embedded within this document in the Appendix.

# Appendices

1. SLOC Count of FBCIM.



1. Java files for FBCIMApp and FBChatContext.



1. Class Diagram generated using Enterprise Architect



1. Patchsets

