EZ Intranet Messenger

Paolo Ascurra	5808758
Cherlyn Quan	9769536
Dong Li	4174356
Lawrence Tejada	6333176
Cheng Cheng	9718648

Summary of Project

EZ Intranet Messenger is a messenger that works on local area network (LAN), so no centralized server is required to setup the application. The messenger provides three main functions. The first one is real time chat. Multiple users (2, 3 or more) can communicate with others at the same time. Secondly, it supports instant message service. It allows user to send a message to other online users (just online users). Lastly, the messenger offers file transmission. Users could transfer a file, under 7 GB, from their computer to other users' computer. In addition, this application is very stable.

Class Diagram of Actual System

Class Diagram Description

In our actual UML class diagram, there is 26 classes in the core and 14 classes in the user interface. We will only focus on the code in the core. The following described classes define the main structures of the application. The other classes are support classes for the application.

Ezim and EzimContact
The Ezim is the starting point of the application. It contains the main method where the application begins. This class bonds with most of the other classes. The EzimContact class mainly contains method used to create an instant of a contact and their status.
Main Relationships: An Ezim can consists of many-to-many EzimContact.

EzimContactList
The EzimContactList keeps the list of contact for collaborating with the EzimContact class. The names in the list can be updated by adding , removing, updating or modifying so that the EzimContact class can use the list for sending messages.
Main Relationships: An EzimContactList contains 0 or many EzimContact.

EzimConf

The EzimConf class provide the preference of location, color, size, state icon, port and so on in order to let the user change them as necessary. The changed performance will be kept and displayed on the panel.
Main Relationships: an EzimConf contains one-to-one Ezim.

EzimMsgSender and EzimDtxSemantics

The EzimMsgSender class has methods to create and send an email to a contact in the contact list. The EzimDtxSemantics contain methods to send and receive emails and files to contacts.
Main Relationships: An EzimMsgSender sends to 0 or 1 EzimContact. An Ezim is composed of one EzimMsgSender. An EzimMsgSender uses an EzimDtxSemantics to send emails.

EzimFileSender, EzimFileResponder, EzimFileComfirmer, EzimFtxList, EzimFrxList

The EzimFileSender contains methods to send files according to the contact by figuring out the address and ports of recipients. The EzimFileResponder indicates the file transmission is accepted by the recipient or not. The EzimFileConfirmer provides the methods to confirm that the file transformation is done or not. The EzimFtxList keeps the list of files which have been sent by the user; meanwhile, the EzimFrxList keeps the list of files which come from the sender when the transmission is done.The EzimFileSender works for sending the files which can be accepted by the EzimFileResponder and can be kept in the EzimFtxList. When the file transmission is done, the EzimFileConfirmer will confirm it and the files will be kept in the EzimFrxList.

Main Relationships: An EzimFileSender sents 0 or more files to EzimContact though the Ezim class. One EzimFileResponder uses one EzimFtxList and one EzimFileConfirmer uses one EzimFrxList.

EzimAckSender, EzimAckTaker, EzimAckSemantics

The EzimAckSender is used to send an acknowledgment sender message for sending a message to broadcast. The EzimAckTaker is used to get an acknowledgement taker for a message that's been sent to broadcast. The EzimAckSemantics contains method to post a message, to update other contact's information, to update the contact list that are connected to the network and to receive all broadcast messages.
Main Relationships: An Ezim contains many EzimAckTaker. An Ezim uses an EzimAckSemantics to post and receive messages and update information. An EzimAckSemantics will be using EzimAckSender. An Ezim contains many EzimAckSender.

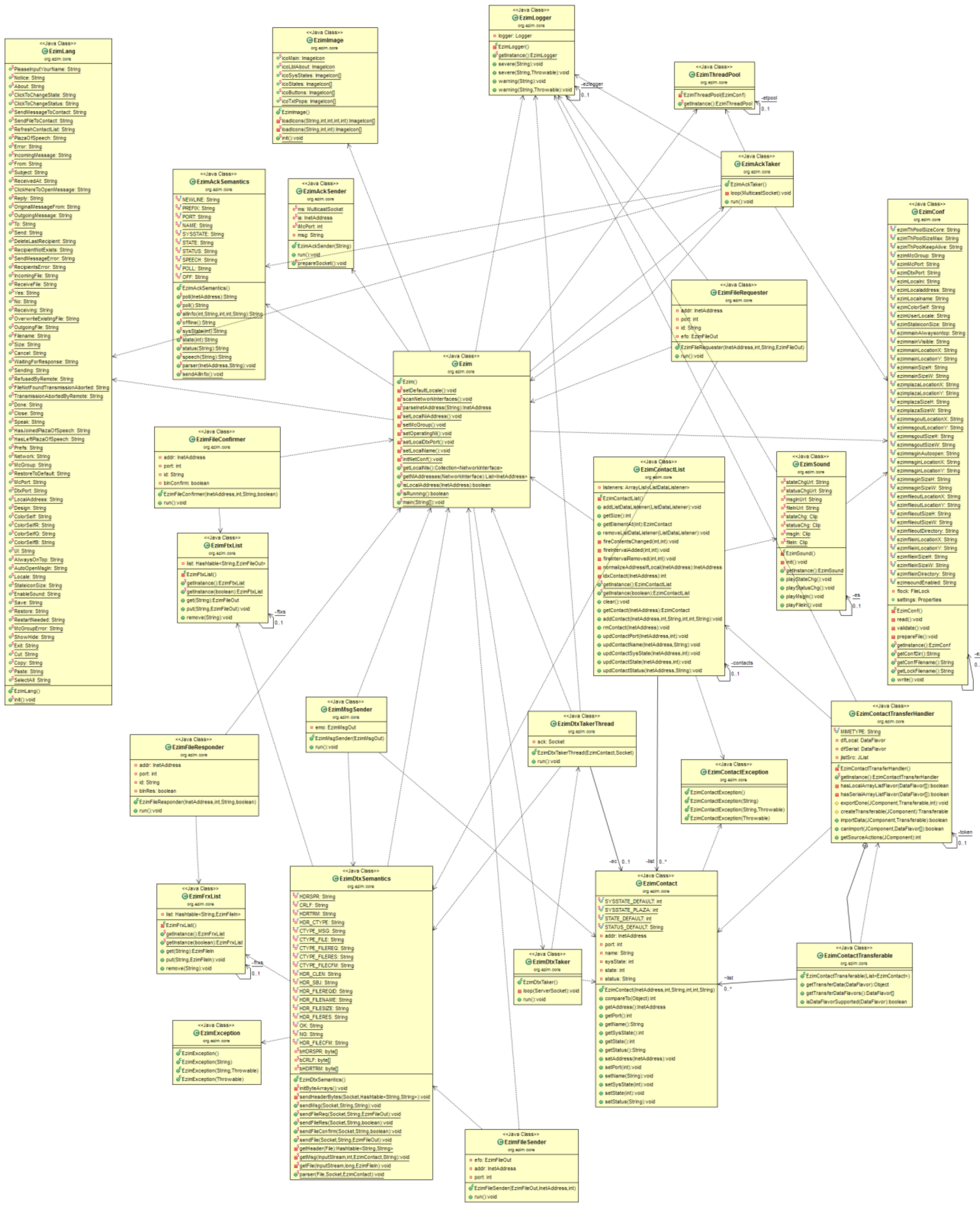

Figure.1 Actual Class Diagram

Conceptual Classes vs. Actual Classes

The conceptual classes that we've listed in the Milestone 2 are slightly different than the actual classes. The main classes are there; however, they are not related the same way as we thought it would have been. All of the classes are connected to the "Ezim" class, which is basically the center point of the program, the main method resides in this class.

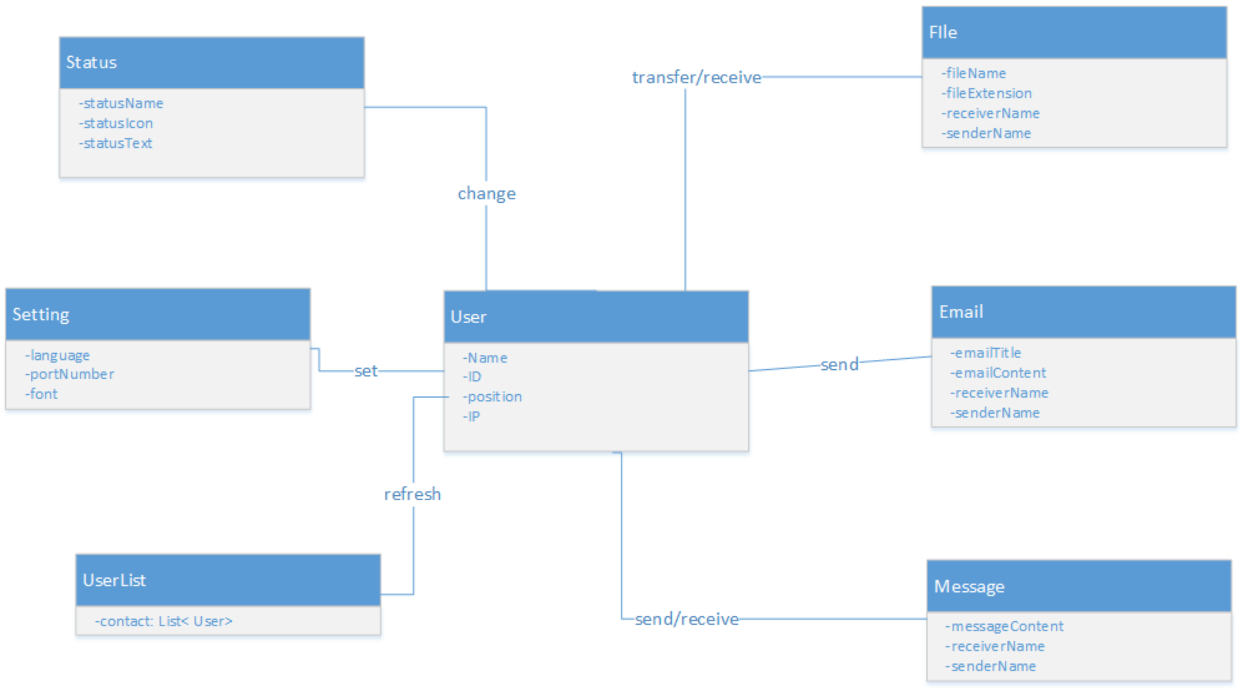

Fig 2. Conceptual Diagram

Equivalent classes

Conceptual Class	Actual Class
User	EzimContact, Ezim
UserList	EzimContactList
Setting	EzimConf
Email	EzimMsgSender, EzimDtxSemantics
Status	EzimContact
File	EzimFileSender, EzimFileResponder, EzimFileComfirmer, EzimFtxList, EzimFrxList
Message	EzimAckSender, EzimAckTaker, EzimAckSemantics

The status class defined in our conceptual class diagram has been combined into the EzimContact class in the actual class diagram.

By architecturing the system, we can see how each class is related to other classes. With this, it makes it easier to refactor the system.

The tool that we have used to create the UML diagram is called ObjectAid UML to do the reverse engineering. With this tool installed in Eclipse, we were able to drag and drop the classes to create our actual UML diagram. After having some classes dropped, we were able to add the associations between the classes. This tool helped us extract the classes and its associations and gave us a clear view of the system.

Class Relationships

```
// This is imported from the class EzimAckSemantics-----------------
import org.ezim.core.EzimAckSender;
public class EzimAckSemantics {
      // send out all local ACK info
      public static void sendAllInfo() {
            EzimMain emTmp = EzimMain.getInstance();
            EzimThreadPool etpTmp = EzimThreadPool.getInstance();
            EzimAckSender easTmp = new EzimAckSender(
                  EzimAckSemantics.allInfo (Ezim.localDtxPort,
                  Ezim.localName, emTmp.localSysState,
                  emTmp.localState, emTmp.localStatus)
            );
            etpTmp.execute(easTmp);
      }
}

//This is imported from the class EzimAckSender----------------------
public class EzimAckSender implements Runnable{
      private static MulticastSocket ms;
      private static InetAddress ia;
      private static int iMcPort;
      private String msg;
      public EzimAckSender(String strIn) {
            this.msg = strIn;
      }
      public void run() {
            DatagramPacket dp = null;
            byte[] arrBytes = null;
            try {
                if (null == EzimAckSender.ms)
                     throw new Exception("Ack socket not prepared
yet.");
                arrBytes = this.msg.getBytes(Ezim.dtxMsgEnc);
                if (arrBytes.length > Ezim.inBuf)
                     throw new Exception("Ack message too long.");
                dp = new DatagramPacket(arrBytes, arrBytes.length,
                     EzimAckSender.ia,EzimAckSender.iMcPort);
                synchronized(EzimAckSender.ms){
                     EzimAckSender.ms.send(dp);
                }
           }
           catch(Exception e){...}
     }
}
```

Code Smell and Possible Refactoring

The code smells for this application are list as following:

Duplicated Code

In the class EzimDtxSemantics and Ezim, there are same code that are used in different methods. In the Ezim class, there are repeating “if” statements with the same conditions as well as a for loop within. A possible fix to this problem is to extract the "if" statements and put it into its own method, giving it a proper name like "Pick a local address". Afterwards, fix the code so everything is under 1 "if" statement, instead of duplicates of the same condition. This would give the code an overall cleaner look.

Large class

The class EzimDtxSemantics has 889 lines of code, this makes the class difficult to maintain and read. One refactoring technique that we could use the Extract subclass technique. By doing this refactoring we could have a parent class EzimDtxSemantics and 2 child classes, the first child class would treat the reception of files, and the second class would treat the reception of messages.

Long Method

The method setLocalNiAddress in Ezim class, which has around 180 lines of code, is too large. It is hard to read and understand the method. This method is doing two things: setting the network interface as well as the network address. To solve this problem, extract methods could be used. Then setLocalNiAddress will be separated to two methods called setLocalNi and setLocalAddress.

Lazy class

The EZimPlainDocument Class is not related to other classes. In addition, this class is more related to the UI components of the project than to the core files. The refactoring techniqueto be used in this cases is the Inline Class, by doing this we could merge this small class that does very little to a bigger class. After the merge, we can delete the lazy class.

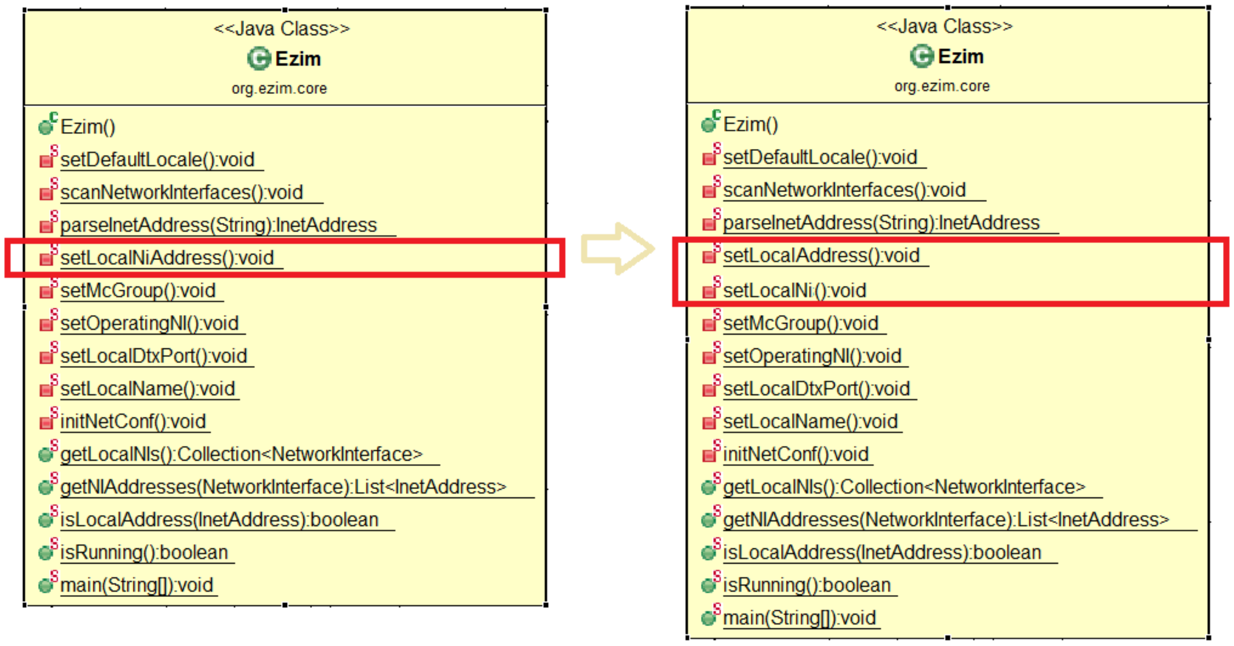

Figure.3 Extract method Refactoring

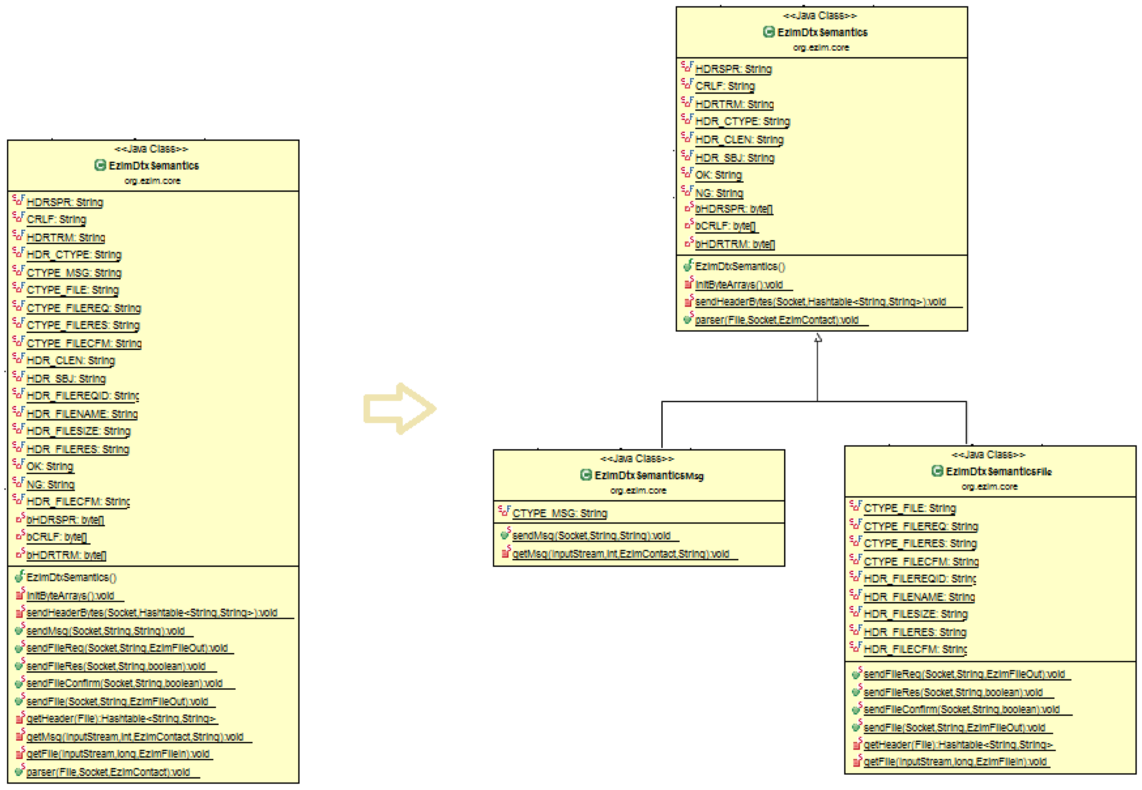

Figure.4 Inheritance Refactoring

The following is the duplicated code from class Ezim:

```
private static void setLocalNiAddress(){
        ……
        EzimConf ecTmp = EzimConf.getInstance();
        String strLclAdr = ecTmp.settings.getProperty(EzimConf.ezimLocaladdress);
        // confine our selectable network interfaces
        Collection<List<InetAddress>> cTmp = null;
        if (null == Ezim.localNI) {
                cTmp = Ezim.nifs.values();
        }else{
                cTmp = new ArrayList<List<InetAddress>>();
                ((ArrayList<List<InetAddress>>) cTmp).add (Ezim.nifs.get(Ezim.localNI));
        }
        // try to pick an IPv6 non-loopback and non-link-locale address
        if (Ezim.localAddress == null) {
                for(List<InetAddress> lTmp: cTmp) {
                        for(InetAddress iaTmp: lTmp) {
                                if (iaTmp instanceof Inet6Address  && ! iaTmp.isLoopbackAddress()
                                        && ! iaTmp.isLinkLocalAddress(){
                                        Ezim.localAddress = iaTmp;
                                        break;
        }}}}
        // try to pick a non-loopback and non-link-locale address
        if (Ezim.localAddress == null) {
                for(List<InetAddress> lTmp: cTmp) {
                        for(InetAddress iaTmp: lTmp) {
                                if (!iaTmp.isLoopbackAddress() && !iaTmp.isLinkLocalAddress()){
                                        Ezim.localAddress = iaTmp;
                                        break;
        }}}}
        // try to pick an IPv6 non-loopback address
        if (Ezim.localAddress == null) {
                for(List<InetAddress> lTmp: cTmp) {
                        for(InetAddress iaTmp: lTmp){
                                if (iaTmp instanceof Inet6Address && ! iaTmp.isLoopbackAddress()){
                                        Ezim.localAddress = iaTmp;
                                        break;
        }}}}
        // try to pick a non-loopback address
        if (Ezim.localAddress == null) {
                for(List<InetAddress> lTmp: cTmp) {
                        for(InetAddress iaTmp: lTmp) {
                                if (! iaTmp.isLoopbackAddress()){
                                        Ezim.localAddress = iaTmp;
                                        break;
        }}}}
        // try to pick an IPv6 address
        if (Ezim.localAddress == null) {
                for(List<InetAddress> lTmp: cTmp) {
                        for(InetAddress iaTmp: lTmp) {
                                if (iaTmp instanceof Inet6Address) {
                                        Ezim.localAddress = iaTmp;
                                        break;
        }}}}
        // pick the first available address when all failed
        if (Ezim.localAddress == null) {
                Ezim.localAddress = Ezim.nifs.elements().nextElement().get(0);
        }
        ……
}
```