EZ Intranet Messenger

Abhishek Appadoo - 6540635
David Di Paola - 6654983
Alyssa Gangai - 6678661
David Paparo - 6664334
Jean-Michel Rodrigue - 6021212

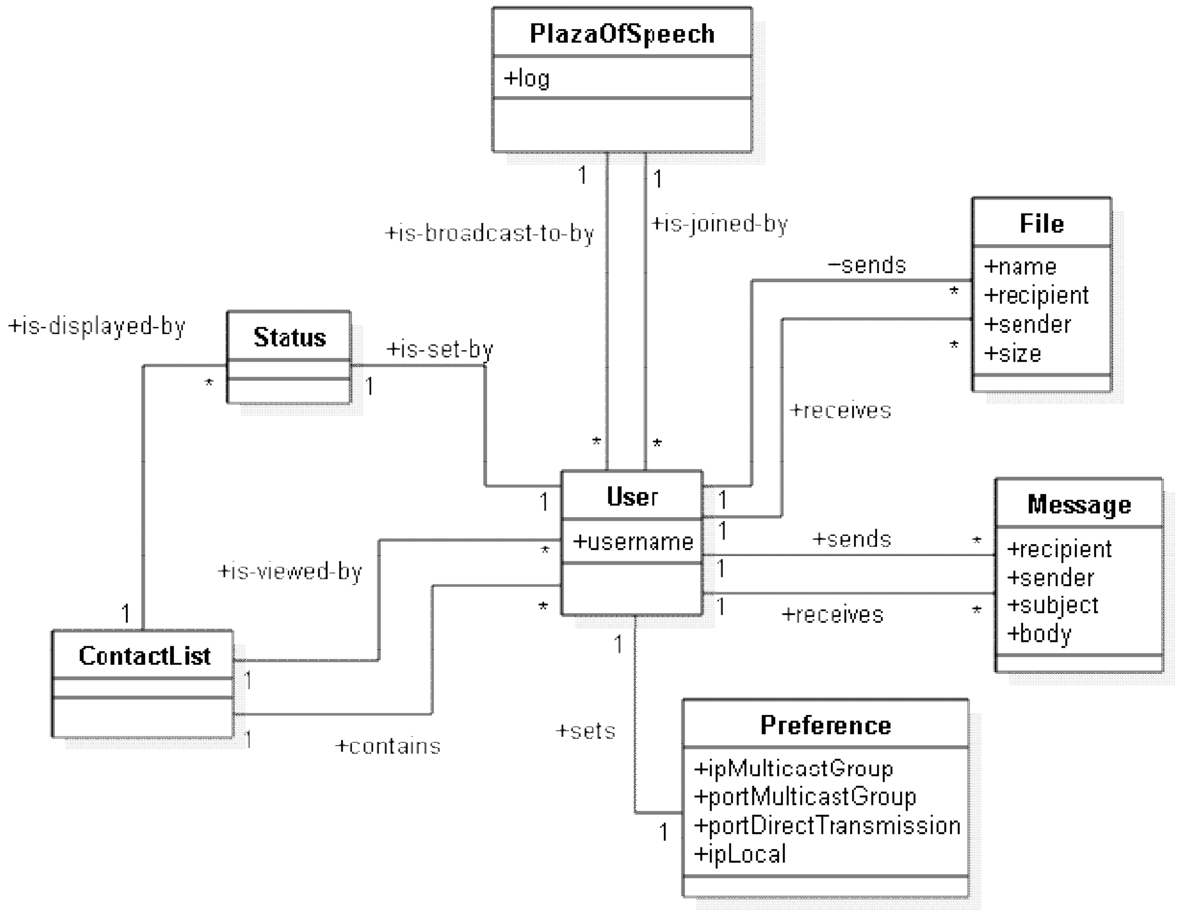
https://github.com/LdTrigger/soen343-ezim

Summary of Project

The application is a simple intranet messaging service that allows instant communication and file transfer between users. This functionality is available without the need for a centralized server within a local area network. Like many instant messengers, the program will list all online contacts and allows users to send private messages. Additionally, the program allows users to send broadcast messages to all users.

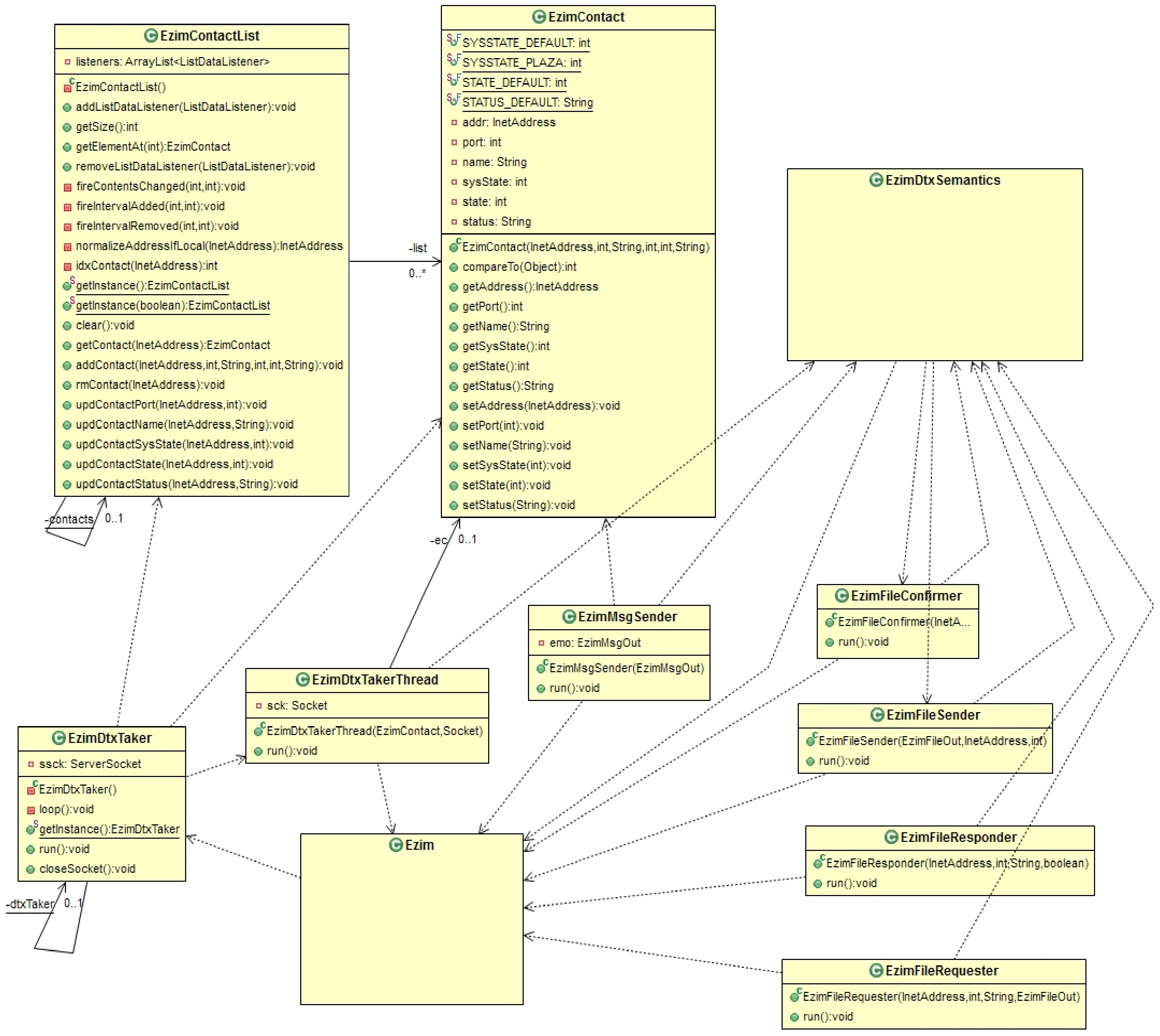
Class Diagram of Actual System

Conceptual Diagram

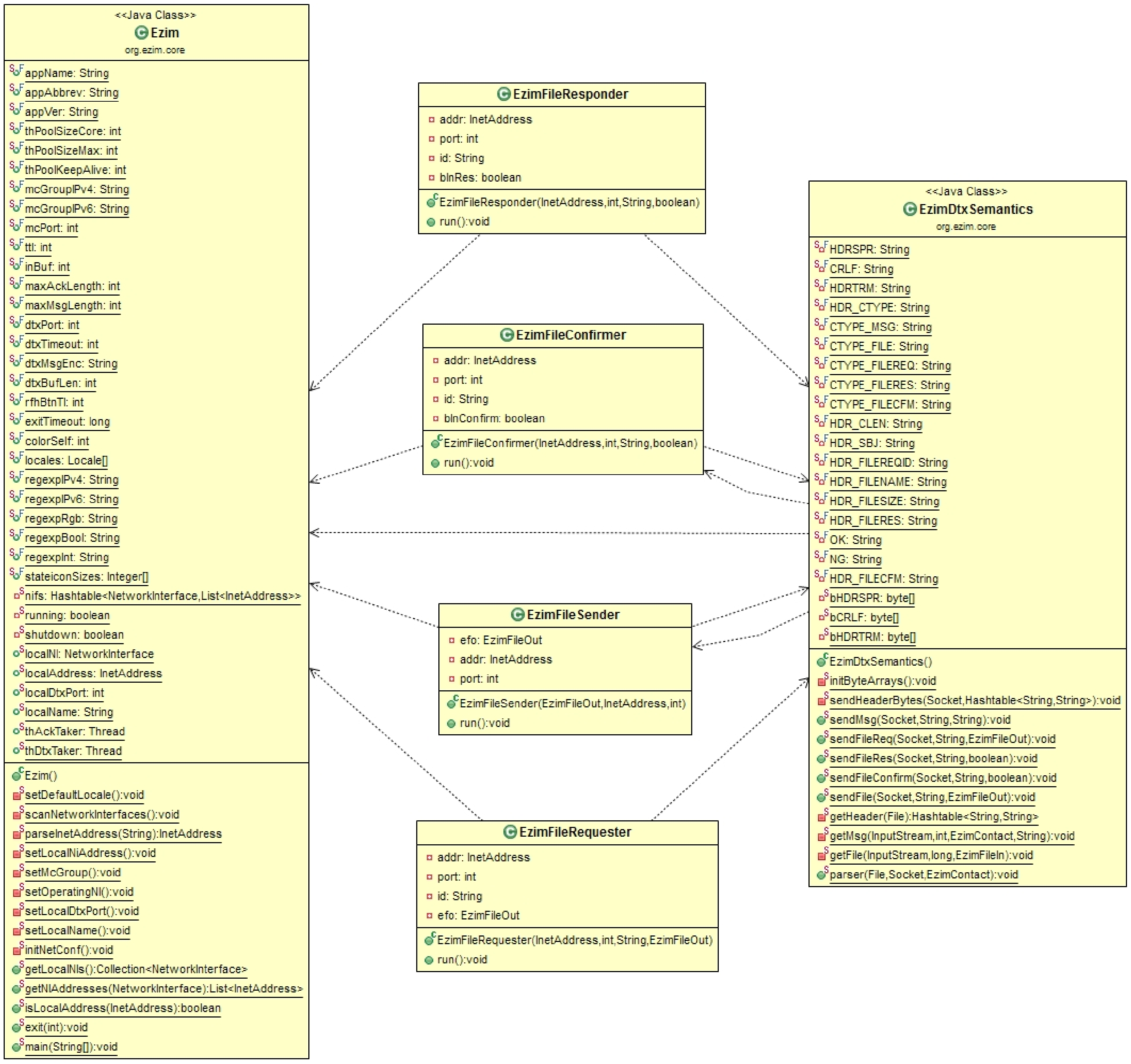


Class Diagram

ObjectAid UML Explorer was used to easily generate the class diagrams.


EzimContactList
listeners: ArrayList<ListDataListener>
EzimContactList()
addListDataListener(ListDataListener):void
getSize():int
getElementAt(int):EzimContact
removeListDataListener(ListDataListener):void
fireContentsChanged(int,int):void
fireIntervalAdded(int,int):void
fireIntervalRemoved(int,int):void
normalizeAddressIfLocal(InetAddress):InetAddress
idxContact(InetAddress):int
getInstance():EzimContactList
getInstance(boolean):EzimContactList
clear():void
getContact(InetAddress):EzimContact
addContact(InetAddress,int,String,int,int,String):void
rmContact(InetAddress):void
updContactPort(InetAddress,int):void
updContactName(InetAddress,String):void
updContactSysState(InetAddress,int):void
updContactState(InetAddress,int):void
updContactStatus(InetAddress,String):void
-contacts
0..1
-list
0..*
EzimContact
SYSSTATE_DEFAULT: int
SYSSTATE_PLAZA: int
STATE_DEFAULT: int
STATUS_DEFAULT: String
addr: InetAddress
port: int
name: String
sysState: int
state: int
status: String
EzimContact(InetAddress,int,String,int,int,String)
compareTo(Object):int
getAddress():InetAddress
getPort():int
getName():String
getSysState():int
getState():int
getStatus():String
setAddress(InetAddress):void
setPort(int):void
setName(String):void
setSysState(int):void
setState(int):void
setStatus(String):void
-ec
0..1
EzimDtxSemantics
EzimFileConfirmer
EzimFileConfirmer(InetA...
run():void
EzimMsgSender
emo: EzimMsgOut
EzimMsgSender(EzimMsgOut)
run():void
EzimDtxTakerThread
sck: Socket
EzimDtxTakerThread(EzimContact,Socket)
run():void
EzimFileSender
EzimFileSender(EzimFileOut,InetAddress,int)
run():void
EzimDtxTaker
ssck: ServerSocket
EzimDtxTaker()
loop():void
getInstance():EzimDtxTaker
run():void
closeSocket():void
-dtxTaker
0..1
Ezim
EzimFileResponder
EzimFileResponder(InetAddress,int,String,boolean)
run():void
EzimFileRequester
EzimFileRequester(InetAddress,int,String,EzimFileOut)
run():void

A Closer Look at Ezim and EzimDtxSemantics



Class Descriptions and Relationships

All relationships between classes are dependency relationships.

Ezim
The Ezim.java defines all default constants the application needs including the application name and version, thread pool sizes, maximum message lengths, and other attributes relating to the networking process involved in sending messages. It contains the main class which will instantiate and start a EzimDtxTaker thread. This class' primary role is to initialize the network configurations by scanning the network interface and instantiating an IP address and applying the network settings. Upon program exit, this class is responsible for initiating the closing of the socket through which data travels by calling closeSocket() EzimDtxTaker upon program exit.

Dependencies:
1. EzimDtxTaker: An EzimDtxTaker is instantiated in the main method of the Ezim Class.

EzimContact
The Contact class contains attributes pertaining to an individual contact. This includes name, IP address, status, port. This class contains getters and setters for the private Contact attributes and has no dependencies.

EzimContactList
The EzimContactList is a Singleton class that contains an Array List of EzimContacts. It is also an observable class that notifies listeners whenever a contact is added, removed or updated.

Dependencies:
1. EzimContact: An ArrayList of EzimContacts is an attribute of EzimContactList
2. EzimContactList: Self dependency as it is a singleton class.

EzimDtxSemantics
The EzimDtxSemantics class that is responsible for all incoming and outgoing messaging and file transfers. This class's responsibilities can be separated into those for handling tasks related to incoming transmission and those for outgoing transmission.

Incoming Transmission
EzimDtxSemantics's main responsibility is to parse all incoming direct transmissions and then react accordingly. It does this by first retrieving the header from the incoming socket and basing its subsequent actions on the header's content type. The following actions can then be performed:

- Receive incoming messages from the socket using a getMsg() function.
- Receive incoming file using a getFile() function
- Receive incoming file request (calls the GUI),
- Receive incoming file confirmation (sets the system to receive messages),
- Receive incoming file response (if the user has accepted the request it creates file confirmer and file sender).
- Handle cases where the file doesn't exist, the request is refused by the remote user, or the outgoing file window is close.

Dependencies:
1. EzimFileConfirmation: instantiated in parser when a file request is accepted.
2. EzimFileSender: instantiated in parser when a file request is accepted.
3. EzimContact: A method parameter for parsing incoming transmissions.
4. Ezim: Ezim system constants are used as parameters and variables.

Outgoing Transmission
In terms of outgoing transmission, EzimDtxSemantics is responsible for the following actions:

- Sending the message header. (Note that a message header is sent for each message, file, file request, file response, and file confirmation)
- Sending the outgoing message
- Sending the file request
- Sending the file response through the socket.
- Sending the file confirmation
- Sending the file

Dependencies:
1. Ezim: Ezim system constants are used as parameters and variables.

<u>EzimDtxTaker</u>
This class initializes the socket which receives transmission data. The EzimDtxTaker is a singleton class which is made of a server socket. It binds the socket to the local IP address and port and waits for a connection to be made from a remote user. Once a connection is made, it searches for the EzimContact making the connection in the EzimContactList. Only if this Contact is in the Contact List, will it create an instance of a EzimDtxTakerThread and execute it.

Dependencies:
1. EzimContactList: The EzimContactList is retrieved in order to know which Contacts messages can be received from.
2. EzimContact: The EzimContacts found in the EzimContactList are used by this class to retrieve the remote sender's network information and set up the connection between the two contacts.
3. EzimDtxTakerThread: An EzimDtxTakerThread is instantiated by the EzimDtxTaker class.
4. EzimTakerThread: Self dependency since it is a singleton class.

<u>EzimDtxTakerThread</u>
The DtxTakerThread is a Runnable class that is made of an EzimContact and a socket. It is responsible for accepting the input from the socket and passing it to EzimDtxSemantics for interpretation along with the socket and contact. EzimDtxSemantics will then act accordingly.

Dependencies:
1. EzimContact: The remote contact is an attribute of EzimDtxTakerThread.
2. EzimDtxSemantics: EzimDtxSemantics' parser() function is called upon to handle the parsing of the input from the incoming data transmission.
3. Ezim: Ezim system constants are used as parameters.

<u>EzimFileConfirmer, EzimFileRequester, EzimFileResponder, EzimFileSender</u>
Each of these classes are Runnable classes that set up the socket and then call upon their respective functions in EzimDtxSemantics**.**

Dependencies:

1. EzimDtxSemantics: sendFileConfirm(), sendFileReq(), sendFileRes() are all functions of EzimDtxSemantics that are called by their respective classes to handle the actual sending of these messages.

EzimMsgSender

EzimMsgSender is tightly coupled with the UI's associated outgoing message window. It is a Runnable class that sets up the socket and then calls upon

Dependencies:

1. Ezim: Ezim's network settings are used as parameters.
2. EzimDtxSemantics: EzimDtxSemantics.sendMsg() is called upon to perform the actual message sending.
3. EzimContact: A list of contacts is created to store the contacts for which the message is intended.

Mapping of conceptual classes

Conceptual Class	Actual Class
ContactList	EzimContactList
User	EzimContact
Preferences	Ezim
Status	EzimContact
Message	EzimMsgSender EzimDtxSemantics
File	EzimDtxSemantics EzimFileConfirmer EzimFileRequester EzimFileResponder EzimFileSender
PlazaOfSpeech	EzimMsgSender, EzimDtxSemantics

EzimDtxSemantics and *EzimFileSender* Relationship

```
public class EzimFileSender implements Runnable{
   private EzimFileOut efo;
   private InetAddress addr;
   private int port;


   public EzimFileSender(EzimFileOut efoIn, InetAddress iaIn, int iPort){...}


   public void run(){
       Socket sckOut = null;
       InetSocketAddress isaTmp = null;
       try{
           // Some code for setting up the socket (...)
           EzimDtxSemantics.sendFile(sckOut, this.efo.getId(), this.efo);
       } catch(Exception e){...
       } finally{...}
   }
}


public class EzimDtxSemantics{
   private static Hashtable<String, String> getHeader(File fIn) throws Exception {...}
   private static void getFile(InputStream isIn, long lCLen, EzimFileIn efiIn) throws Exception{...}


   public static void parser(File fIn, Socket sckIn, EzimContact ecIn)    {
       Hashtable<String, String> htHdr = null;
       String strCType = null;
       String strCLen = null;
       try{
           htHdr = EzimDtxSemantics.getHeader(fIn);
           strCType = htHdr.get(EzimDtxSemantics.HDR_CTYPE);
           strCLen = htHdr.get(EzimDtxSemantics.HDR_CLEN);
           if (strCType == null){...
       } else if (strCLen == null){...
           } else{
               if (strCType.equals(EzimDtxSemantics.CTYPE_MSG)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILE)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILEREQ)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILECFM)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILERES)){...
                   String strFileReqId = htHdr.get(EzimDtxSemantics.HDR_FILEREQID);
                   String strFileRes = htHdr.get(EzimDtxSemantics.HDR_FILERES);
                   EzimFileOut efoTmp = EzimFtxList.getInstance().get(strFileReqId);
                   EzimThreadPool etpTmp = EzimThreadPool.getInstance();
                   if (efoTmp != null){
                       if (strFileRes.equals(EzimDtxSemantics.OK)){
                           if (efoTmp.getFile().exists()){
                               efoTmp.setSysMsg(EzimLang.Sending);
                               EzimFileConfirmer efcTmp = new EzimFileConfirmer(ecIn.getAddress(),
ecIn.getPort(), strFileReqId, true);
                               etpTmp.execute(efcTmp);
                               EzimFileSender efsTmp = new EzimFileSender(efoTmp, ecIn.getAddress(),
ecIn.getPort());
                               etpTmp.execute(efsTmp);
                           } else{...}
                   } else{...}
               } else if (strFileRes.equals(EzimDtxSemantics.OK)){...}
           }
       }
       } catch(EzimException ee){ ...
       } catch(Exception e){ ...
       } finally{...}
   }
}
```

Code Smells and System Level Refactorings

The most important code smells in the project are high coupling and low cohesion. The class *EzimDtxSemantics* contains logic for sending and receiving files and messages, through a parsing method called *parser*(). This method is tightly coupled to four classes called *EzimFileSender*, *EzimFileConfirmer*, *EzimFileRequester*, and *EzimFileResponder*. These four classes are guilty of another code smell called *Duplicated Code*. This is dangerous as it makes the system less maintainable; when portions of the code need to be modified, certain sections meant to share functionality might be missed. Furthermore, copying functionalities may lead to undesirable results, such as omitting slight differences between the copied sections or adding unnecessary functionalities. In this case, each *run*() method of these four classes have the exact same code except for their very last method call which is respective to the class name. The code has clearly been copied between classes since some of the comments were not changed to reflect the different class names.

The best way to refactor this is to use the *Extract Class* refactoring. This can be done by creating a new class that will handle connecting the socket to the IP address and port, called *SocketBinder*. The IP address and port number parameters originally in the constructor of each of these 4 classes will be assigned as attributes to the new *SocketBinder* class along with Socket attribute. At this point *Extract Method* will be used on the duplicated portion of code that sets up the socket and injected into a new method called *connectSocket*() belonging to *SocketBinder*. As a result of this refactoring *FileSender*, *FileResponder*, *FileRequester*, *FileConfirmer* become *Middle Men* classes, since they call instances of the *SocketBinder* and then delegate their responsibilities back to *EzimDtxSemantics* which holds the methods that actually take action. Furthermore, they will also be *Lazy Classes* since they will not hold many responsibilities. To counter these code smells, it will be best to use *Remove Middle Man*, and have *EzimDtxSemantics* create instances of *SocketBinder* directly instead of delegating this responsibility to these four classes. The four classes can then be deleted and *EzimDtxSemantics* will be responsible for calling its own *sendFileReq*(), *sendFileRes*(), *sendFileConfirm*() and *sendFile*() methods. It should be noted that *EzimMsgSender* also contains this duplicated code however it is tightly coupled with the GUI which is out of the scope of this project.

Afterwards, to correct the low cohesion, it is best to *Extract Method*s with similar functionality. This can be achieved by creating two new classes, *Receiver* and *Sender*. They then extract the relevant methods from *EzimDtxSemantics*, thus separating the sending and receiving functionality and increasing cohesion. The *Receiver* class will now also be responsible for parsing incoming messages. The remaining attributes in *EzimDtxSemantics* will remain, making it a *Data Class*.

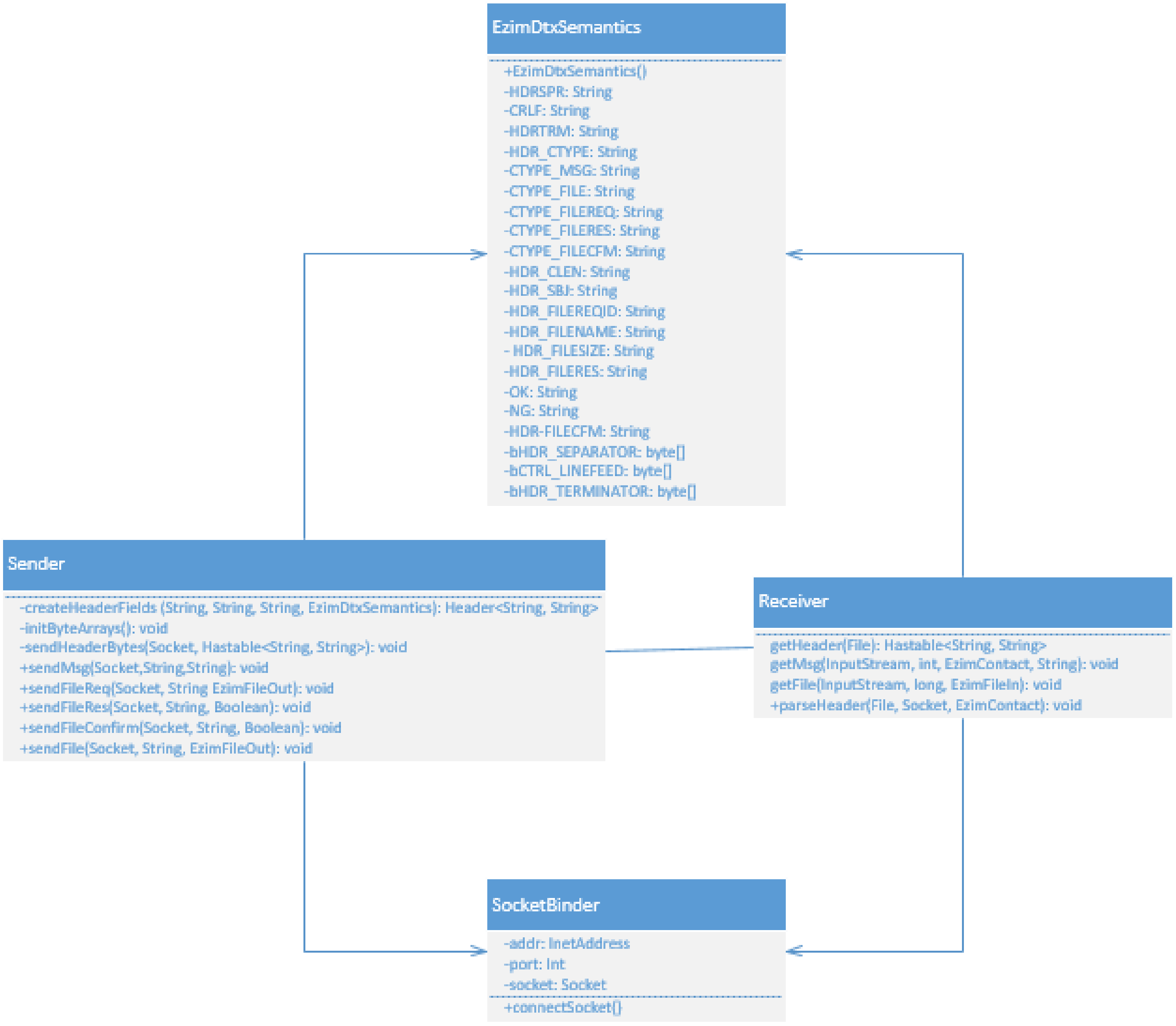
Once again, *Duplicated Code* appears as a code smell as it is an issue in *EzimDtxSemantics*. The methods *sendMsg*(), *sendFileReq*(), *sendFileRes*(), *sendFileConfirm*() and *sendFile*() are a clear case of this. They each contain a section devoted to "[creating] necessary header fields", placing everything into a hash table, which is then sent along. Due to its nature, it would be best to extract the creation of the hash table into a method called *createHeaderFields*(), passing it the few varying values as parameters: a string used to identify the type of data being sent, already stored in global variables.

Additionally, the sheer amount of comments being used to describe methods, parameters and attributes, whose names tell you very little about their functionality. Although this is a basic issue, it is so prevalent throughout the code that it becomes difficult to understand what a class is responsible for. Newcomers to this project would be hard-pressed to understand what they are looking at, requiring many hours of digging to obtain the logic behind the system. Refactorings that can be used to fix this code smell are *Rename Metho*d and *Rename Parameter*. By renaming methods and parameters with more appropriate names unnecessary comments can be avoided, the purpose of the method calls will be more clear and readability will be improved. Unfortunately, due to its abundance throughout the code, there is no single class that needs this change. *EzimDtxSemantics,* even as a *Data Class,* is particularly guilty of this code smell. For example, there are byte array attributes that have names such as bHDRSPR, bCRLF, bHDRTRM. These say very little about the attributes, which actually represent "header separator", "control line feed", and "header terminator" respectively. These should be renamed to bHDR_SEPARATOR, bCTRL_LINEFEED, and bHDR_TERMINATOR. The *parser*() method, now living within *Receiver*, is also very vague, and will be renamed to *parseHeader*() to reflect its actual functionality.

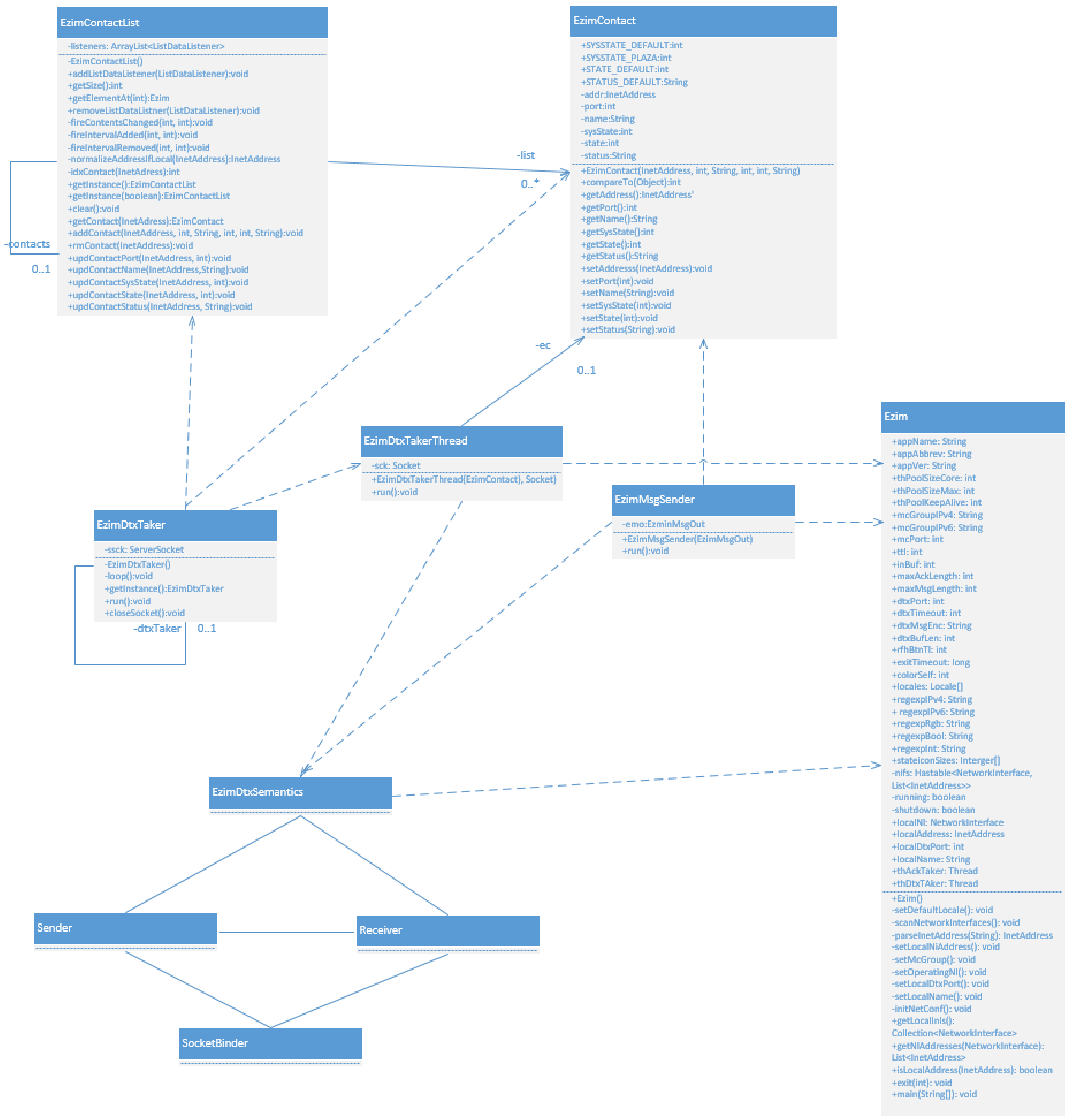
EzimDtxSemantics also suffered from relying on too many *Switch Statements* and *Long Methods*. As *parseHeader*() now belongs *Receiver*, these will need to be looked at more closely. There is a set of conditionals within *parseHeader*() determining the method's behaviour based on what type of header it receives. This is what causes the *Long Method* code smell since there are multiple cases to account for.

A *Strategy* pattern can be implemented here to help guide the creation of the *Sender* object, using a "type" attribute to account for its behaviour.

Class diagram of the EzimDtxSemantics Refactorings


EzimDtxSemantics
+EzimDtxSemantics()
-HDRSPR: String
-CRLF: String
-HDRTRM: String
-HDR_CTYPE: String
-CTYPE_MSG: String
-CTYPE_FILE: String
-CTYPE_FILEREQ: String
-CTYPE_FILERES: String
-CTYPE_FILECFM: String
-HDR_CLEN: String
-HDR_SBJ: String
-HDR_FILEREQID: String
-HDR_FILENAME: String
- HDR_FILESIZE: String
-HDR_FILERES: String
-OK: String
-NG: String
-HDR-FILECFM: String
-bHDR_SEPARATOR: byte[]
-bCTRL_LINEFEED: byte[]
-bHDR_TERMINATOR: byte[]
Sender
-createHeaderFields (String, String, String, EzimDtxSemantics): Header<String, String>
-initByteArrays(): void
-sendHeaderBytes(Socket, Hastable<String, String>): void
+sendMsg(Socket,String,String): void
+sendFileReq(Socket, String EzimFileOut): void
+sendFileRes(Socket, String, Boolean): void
+sendFileConfirm(Socket, String, Boolean): void
+sendFile(Socket, String, EzimFileOut): void
Receiver
getHeader(File): Hastable<String, String>
getMsg(InputStream, int, EzimContact, String): void
getFile(InputStream, long, EzimFileIn): void
+parseHeader(File, Socket, EzimContact): void
SocketBinder
-addr: InetAddress
-port: Int
-socket: Socket
+connectSocket()

Class Diagram of the Refactored System



Specific Refactorings to be implemented in Milestone 4

1. The first refactoring to be performed will be the creation of SocketBinder and the extraction of the duplicated code in EzimFileResponder, EzimFileRequester, EzimFileSender, and EzimFileConfirmer. All four classes will delegate their socket binding responsibilities to SocketBinder. Once this is complete, we will remove the delegation by replacing all instances of the four original classes with the instantiation of a SocketBinder object and then making the appropriate send() method call.

2. The next refactoring will be to reduce coupling and increase cohesion in the *EzimDtxSemantics* class by extracting out two new classes, Sender and Receiver, based on the two main responsibilities, sending and receiving All send methods, as well as the corresponding private method *initByteArrays*() will be extracted into Sender and all get methods as well as parser() will be extracted into Receiver. The single call to parser() by DtxThreadTaker will be changed to reflect the fact that parser is now in the Receiver class. EzimDtxSemantics will be left as a Data Class to house string constants related to direct transmission.

3. The final set of refactorings will be the minor changes within the classes themselves. This includes the removal of duplicate code and renaming of attributes. Duplicate code creating the header fields found with the send methods of the new Sender class will be extracted into a new method *createHeaderFields*(), removing a large portion of code in each of the send methods, and placing it under a single location in order to increase maintainability.

Here is the code related to the second refactoring:

```
public class EzimDtxSemantics
{
  // final static string variables left out for readability. Refer to UML diagrams

   public static void sendMsg(Socket sckIn, String strSbj, String strMsg)throws Exception {...}
   public static void sendFileReq(Socket sckIn, String strId, EzimFileOut efoIn)throws Exception {...}
   public static void sendFileRes(Socket sckIn, String strId, boolean blnRes)throws Exception {...}
   public static void sendFileConfirm(Socket sckIn, String strId, boolean blnConfirm)throws Exception {...}
   public static void sendFile(Socket sckIn, String strId, EzimFileOut efoIn)throws Exception {...}

   private static Hashtable<String, String> getHeader(File fIn)throws Exception {...}
   private static void getMsg(InputStream isIn, int iCLen, EzimContact ecIn, String strSbj)throws Exception {...}
   private static void getFile(InputStream isIn, long lCLen, EzimFileIn efiIn)throws Exception {...}

   public static void parser(File fIn, Socket sckIn, EzimContact ecIn)    {
       Hashtable<String, String> htHdr = null;
       InputStream isData = null;
       String strCType = null;
       String strCLen = null;
       long lCLen = 0;


       try{
           isData = sckIn.getInputStream();
           htHdr = EzimDtxSemantics.getHeader(fIn);
           strCType = htHdr.get(EzimDtxSemantics.HDR_CTYPE);
           strCLen = htHdr.get(EzimDtxSemantics.HDR_CLEN);
           if (strCType == null){...
           } else if (strCLen == null){...
           } else{
               if (strCType.equals(EzimDtxSemantics.CTYPE_MSG)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILE)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILEREQ)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILECFM)){...
               } else if (strCType.equals(EzimDtxSemantics.CTYPE_FILERES)){...
```

```
                   String strFileReqId = htHdr.get(EzimDtxSemantics.HDR_FILEREQID);
                   String strFileRes = htHdr.get(EzimDtxSemantics.HDR_FILERES);
                   EzimFileOut efoTmp = EzimFtxList.getInstance().get(strFileReqId);
                   EzimThreadPool etpTmp = EzimThreadPool.getInstance();
                   if (efoTmp != null){
                       if (strFileRes.equals(EzimDtxSemantics.OK)){
                           if (efoTmp.getFile().exists()){
                               efoTmp.setSysMsg(EzimLang.Sending);
                               EzimFileConfirmer efcTmp = new EzimFileConfirmer(ecIn.getAddress(),
                                   ecIn.getPort(), strFileReqId, true);
                               etpTmp.execute(efcTmp);
                               EzimFileSender efsTmp = new EzimFileSender(efoTmp, ecIn.getAddress(),
                                   ecIn.getPort());
                               etpTmp.execute(efsTmp);
                           } else{...}
                   } else{...}
               } else if (strFileRes.equals(EzimDtxSemantics.OK)){...}
           }
          }
       } catch(EzimException ee){ ...
       } catch(Exception e){ ...
       } finally{...}
    }
  }
}
```