Xpairtise System documentation

Purpose

This short documentation is meant to bridge the gap between Javadoc and the source code. Deeper inside can be gained by looking at the principles underlying this plug-in.

XPairtise System architecture

The salient characteristic of the design is the model-view-controller architecture used for all user interactions and the client/broker/server architecture for network issues.

To aid in this aim, we use interfaces. Thus existing implementations can easily be replaced.

Network design

Basic Network Model

The basic network structure is a typical client-server approach. A single server exchanges data with several clients. No client-to-client connections are made in the current form.

As the underlying transport mechanism, the Java Messaging Service (JMS) was chosen, because it covers many low-level aspects of message exchange that were required for this project (for example, asynchronous 1-to-n relaying of messages). As JMS is only a specification, a concrete implementation had to be chosen. The ActiveMQ library was selected because it is in wide-spread use, and therefore considered mature and stable.

JMS is based around the concept of a "Broker", which serves as a message-relaying hub. (A broker can implement many advanced features, but none of these are used in this project.) It is important to note that the "actual" server (which implements the business logic) does not accept connections itself. Instead, it acts as just another JMS client that connects to the broker (and occupies a reserved queue; see below). For ease-of-use, the server executable starts an embedded broker, and then launches the actual server component that connects to this.

The basic message flow is that the server opens one central message queue, into which clients send their commands (e.g. their authentication request). The server can now either send a direct reply (which is associated by the JMS framework), or send out a separate message. To deliver messages to a specific client, another queue is opened for each client session. The client receives a unique session token in the reply to its initial connection request. This token allows both sides to identify the unique server-to-client queue. It is also used by the client to "tag" each outgoing message, so the server can associate incoming messages with a user session. (This token is also a kind of "shared secret" that provides some simple session authentication.)

In addition to these message queues (which have a single recipient), JMS provides so-called "topics",

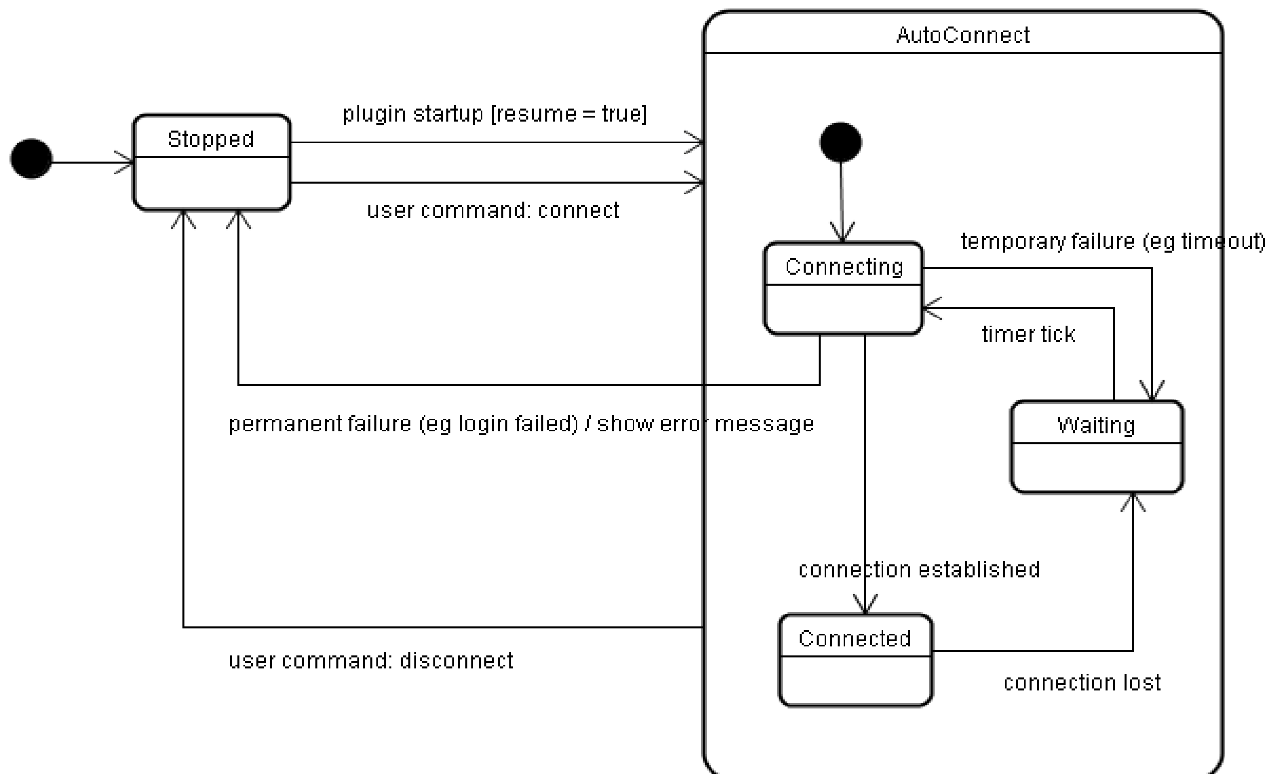
which are m-to-n message multipliers. Several specialized topics are created for various features, where they serve as server-to-client broadcasts. In the current form, only the server application posts messages to topics. All client messages are sent to the server first.

Connection Control

Connecting and disconnecting from a server is done asynchronously in the background. The user controls a boolean "connection desired?" flag with a toggle button in the UI. In regular intervals, a timed event compares the actual with the desired state, and takes action if there's a difference (ie, tries to connect or disconnect).

Connection failures are separated into "permanent" and "non-permanent" failures. A permanent failure is not likely to change if another connection attempt is made; examples are an unknown user id or a wrong password. Non-permanent failures are the opposite, where another attempt could solve the problem (for example, a network timeout). Permanent failures reset the "connection flag" (which is also fed back to the UI), and display a message box with a description of the problem. Non-permanent failures cause the ConnectionManager to go into a wait state, before another attempt is made.

On connection state changes, events are fired which the other components use to initialize, or to modify their state.



State diagram : client startup

Some Involved packages

To gain deeper understanding, you might take a look at the following packages:

```
package de.fuh.xpairtise.common.network.imp.activemq.internal;
package de.fuh.xpairtise.common.network.imp.activemq;
package de.fuh.xpairtise.common.network.data;
```

Involved classes

ActiveMQBrokerConnection.java

```
/**
 * This class provides a ActiveMQ connection and can generate sessions out from
 * this connection. A session is a single threaded context for sending receiving
 * messages.
 *
 */
public class ActiveMQBrokerConnection implements ExceptionListener
{
  // private static ActiveMQBrokerConnection instance;

  private Connection connection;
  private LinkedList<IActiveMQBrokerConnectionStatusListener> listeners = new
LinkedList<IActiveMQBrokerConnectionStatusListener>();
  private boolean connectionEstablished = false;
  [...]
```

Replicated elements

All elements are organized as lists, like the chatlog, the user and the session list. The elements attributes are very simple. Thus, if a list element changes, it will be transmitted as a whole object. Thus an object is completely serialized prior to transmitting.

The purpose of ElementId and SequenceId are as follows:
ElementId stands for a constant entity, e.g. a user or a chat entry.
SequenceId represents a counter for a timeline of editing operations. In all server replicated lists, the SequenceId is assigned by the server. When the client sends changes, they leave this attribute at a value equal to zero.

Replication

Principles

At the core of the replication lies the replicator. Updates are sent to the replicator, which deletes the obsolete element from the replicated list. To this end it uses the ElementId attribute. The replicator inserts the new version into the replicated list, sorted according to the SequenceId attribute.

The update status of each listener is uniquely defined by the highest SeuquenceId it received. Each client listens to the replicator. The replicator uses the method addListenerAndSynchronize() to send the elements of the replicators internal list to the clients via add().

The listener in the client has a lastSequenceId attribute. Every time the listener recieves an update, it sets this attribute to the highest value it received.

By removing old entries from the replicators' lists in the server, we simply have to add() updates to the listener for synchronizing. To generate all network related objects, we created the INetworkFactory interface. The replicated list consists of 2 parts: a master list and a receiver list. The receiver part gets the add, remove and update operations with the corresponding parameters. In other words, the list receiver consumes list updates. In the aforementioned interface theses lists are named as

de.fuh.xpairtise.common.replication.IReplicatedListMaster<T>

de.fuh.xpairtise.common.replication.AbstractReplicatedListReceiver<T> -

The master list is the central model. The clients act as distributed lists. The clients hand the modifications directly to the GUI without storage in between. The GUI must react to the changes of the model.

Server command interface

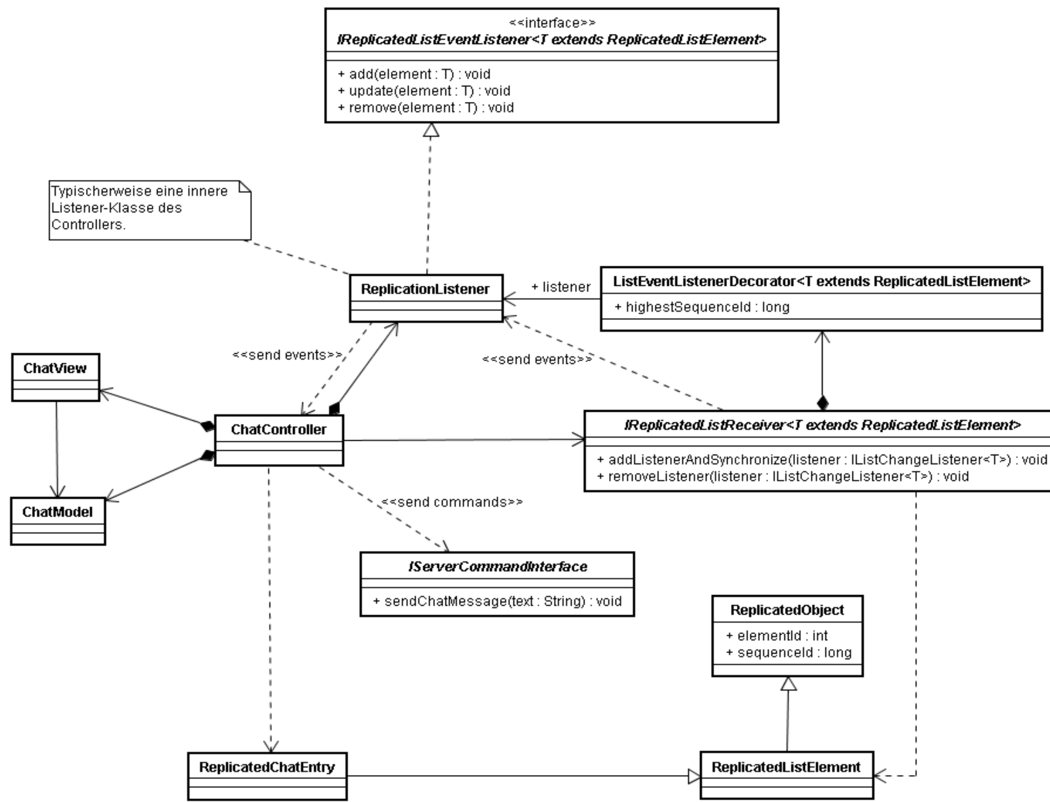
Used for the client to server communication. The server uses a server listener interface, which executes the actions according ro the clients' requests. For instance, the storing of messages in the replicated chat list.

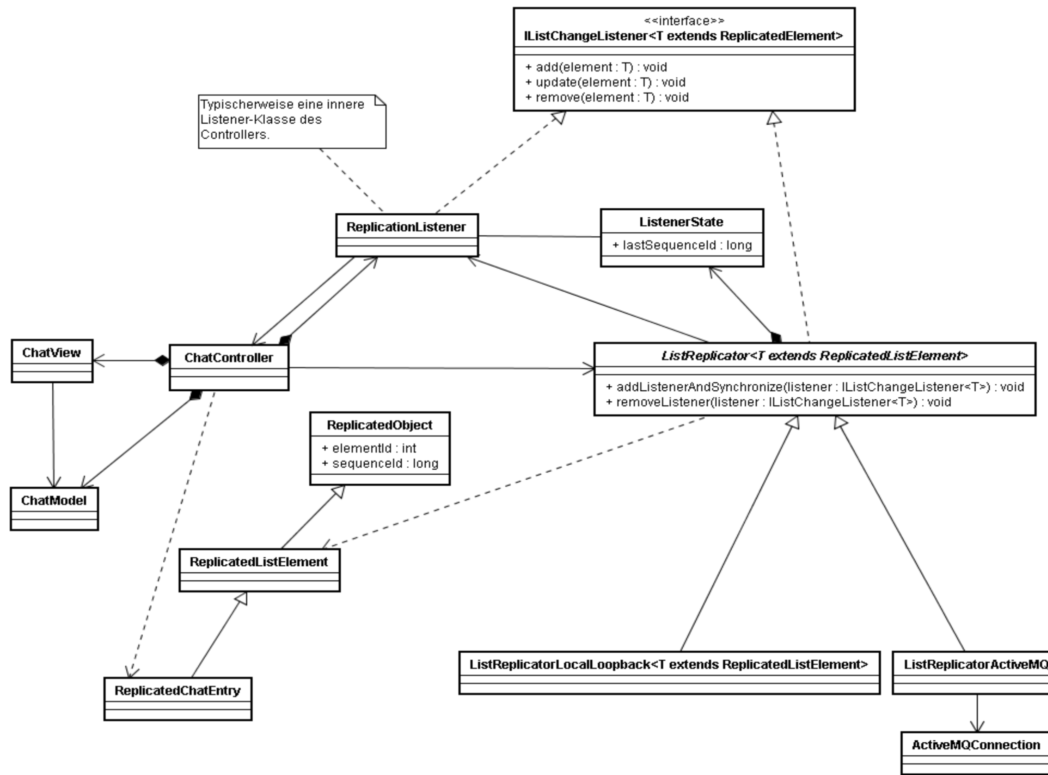
For every replicated list, an amq session is generated. Within each session, several topics or queues are created.

Client initialization

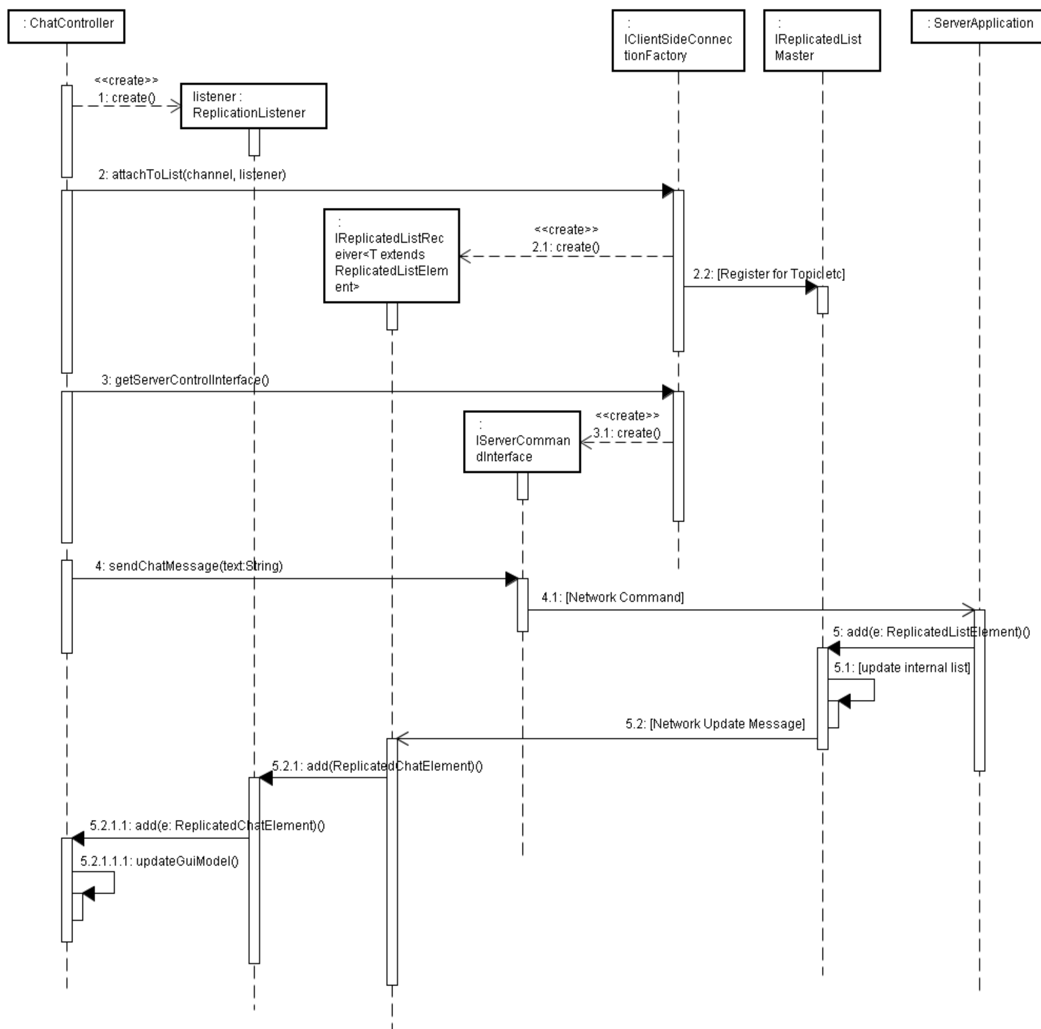
Following steps happen during a clients initialization for every replicated list.

- Client creates a session
- Client creates a topic for receiving an update
- Client creates a single context session
- Client creates a topic for receiving updates
- Client opens a temporary queue for the reply
- Client posts an initialization via the command interface (Queue client -> Server), the temporary queue is handed to the server to reply to
- The client ignores all updates till the reply is received


<<interface>>
IReplicatedListEventListener<T extends ReplicatedListElement>
+ add(element : T) : void
+ update(element : T) : void
+ remove(element : T) : void
Typischerweise eine innere Listener-Klasse des Controllers.
ReplicationListener
+ listener
ListEventListenerDecorator<T extends ReplicatedListElement>
+ highestSequenceId : long
<<send events>>
<<send events>>
ChatView
ChatController
IReplicatedListReceiver<T extends ReplicatedListElement>
+ addListenerAndSynchronize(listener : IListChangeListener<T>) : void
+ removeListener(listener : IListChangeListener<T>) : void
ChatModel
<<send commands>>
IServerCommandInterface
+ sendChatMessage(text : String) : void
ReplicatedObject
+ elementId : int
+ sequenceId : long
ReplicatedChatEntry
ReplicatedListElement



<<interface>>
IListChangeListener<T extends ReplicatedElement>
+ add(element : T) : void
+ update(element : T) : void
+ remove(element : T) : void
Typischerweise eine innere Listener-Klasse des Controllers.
ReplicationListener
ListenerState
+ lastSequenceId : long
ChatView
ChatController
ListReplicator<T extends ReplicatedListElement>
+ addListenerAndSynchronize(listener : IListChangeListener<T>) : void
+ removeListener(listener : IListChangeListener<T>) : void
ReplicatedObject
+ elementId : int
+ sequenceId : long
ChatModel
ReplicatedListElement
ReplicatedChatEntry
ListReplicatorLocalLoopback<T extends ReplicatedListElement>
ListReplicatorActiveMQ
ActiveMQConnection



Workspace synchronizing

When a user creates a new session, he must specify the project for this session. All resources belonging to the specified project are uploaded to the server in a compressed zip file.

The server generates a ReplicatedList for the project. The ReplicatedList stores the names, location and content hash value. The uploaded resources (files) are stored in the servers file system. Changes to these resources are transmitted from the client to the server. With the aid of the aforementioned master list the server takes care to update the resources.

The master list contains the actual state of the resources: name, location, hash. The client detects a

change in the resource via the hash value. To this purpose, the client compares the hashvalues of the local copy of the resource. Upon detecting a difference, the client requests a new version from the server.

Consequently, the client must manage the resources belonging to a project locally. For every local copy of a resource, the corresponding hash value must be available to the client.

Upon joining a session, all local copies of a projects resources are compared to the copies on the server. If a difference is detected, the clients resources are updated from the server.

When the driver changes a resource, the corresponding editor Replicated List records the initial versions hash code (resource sync). Thus late coming clients can compare their version of the resource . The local version of the resource is handled via the local resource management. Every time a resource is saved, the newest version is sent to the server. Changes to the replicated list in the server managing the resources ensue. The replicated list containing the editing actions must dump all recorded actions. The resource synchronization element must be updated. After this sequence of actions, further changes to the resource can ensue.

Some of the classes involved are:

`ClientResourceManager.java`

This class manages the local resources of the currently shared project and keeps them in sync with the version on the server.

```
 Attaches the ClientResourceChangeManager to the serverside list for the
   * given xpSessionId and syncs the project data.
   *
   * @param xpSessionId -
   *           ID of session
   * @param monitor -
   *           Progress monitor instance
   * @throws Exception
   */
  public synchronized void attach(String xpSessionId, IProgressMonitor monitor)
      throws Exception
  {
```

`IServerCommandListener.java`

This interface corresponds to the IServerCommandInterface interface and is the receiver part of the server commands.

`IServerCommandInterface.java`

This interface declares all methods which executes methods on the server side.

Pull-push mechanisms

When joining a session, a client might have to update its resources. In this case, the pull mechanisms come into play: the client sends a request to the server, which responds by sending the resource to the client.

During a session, the driver modifies resources. The modied resources are sent from the client to the server. The server distributes the resource to the clients via an AMQ-topic to all clients which joined

the session in question.

The XPairtise Editor

The core component of the XPairtise editor functionality is the EditorWrapper. Instead of implementing a complete editor, this wrapper is used to integrate existing Eclipse editors into the pair programming environment. It does this by attaching several different listeners to the editor's widget and document in order to be informed about all local changes that need to be distributed. For convenience, these listeners are divided into driver-only listeners, navigator-only listeners and general listeners. The first two groups are attached/detached as needed if the session role of the local user changed, while the general listeners are those that are useful regardless of the current role.

Each EditorWrapper instance is accompanied by an EditorController instance, which is used for the network communication in both directions. It acts as Receiver for the replicated master list used for the updates of the managed editor, forwarding updates it receives via this list to the EditorWrapper which in turn applies those to the wrapped editor. In the opposite direction the EditorWrapper uses the controller to send out the locally occured changes.

Text selections are distributed by both the driver and the navigator of a session. Only the driver's selection really influences the actual selection of remote editors though. The navigator's selections are remotely displayed using a mechanism independent of the main selection, giving it a pure text marker funcion. This feature is provided by the NavigatorSelection class.

The replicated master lists used for the editor functionality are split into two groups. There is one editor list for each currently active XP session and one update list for each currently active editor. An editor list holds ReplicatedEditor elements, each of which represents a single shared editor while an update list is used to distribute all changes belonging to a single editor in the form of AbstractReplicatedEditorCommand elements.

On the server, all of these lists are managed by a central EditorManager, while on the client side, the two list families are handled by different components. The update lists are handled by the aforementioned EditorController instances while the editor list of the XP session the local user is currently participating in is handled by the Singleton ClientEditorManager instance.

This ClientEditorManager is used to manage all local EditorWrapper instances belonging to the current XP session. They are launched/closed upon an add/remove request on the editor list (after launching a new editor for the input file specified in the received ReplicatedEditor element in case of an add-request) or, on the driver side, also for locally launched/closed editors if their input file belongs to the project shared within the current XP session.
Update requests received for existing ReplicatedEditor elements are handled in different ways depending on the fields of the received ReplicatedEditor element.

Possible reactions are: Simply making the local editor visible, refreshing the editor's input or replacing the editor's input with a different file.

To follow editing actions, we need to access the source viewer and widget of the Eclipse Java editor.

This editor is based on the AbstractTextEditor class. The following list summarizes the imports from Eclipse:

```
import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.ui.IEditorPart;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.ide.IDE;
import org.eclipse.ui.part.FileEditorInput;
import org.eclipse.ui.texteditor.AbstractTextEditor;
```

Code snippets

The following code snippets show where and how the connection to the Eclipse environment and resources are made.

Org.eclipse.ui.internal.PartListenerList

```
/**
     * Notifies the listener that a part has been activated.
     */
    public void firePartActivated(final IWorkbenchPart part) {
        Object[] array = getListeners();
        for (int i = 0; i < array.length; i++) {
            final IPartListener l = (IPartListener) array[i];
            fireEvent(new SafeRunnable() {
                public void run() {
                    l.partActivated(part);
                }
            }, l, part, "activated::"); //$NON-NLS-1$
        }
    }
```

ClientEditorManager.java:

```
public void partActivated(IWorkbenchPart part)
  {
    if (part instanceof AbstractTextEditor)
    {
      if (editorManager == null)
      {
        editorManager = ClientEditorManager.getInstance();
      }
      if (editorManager != null && editorManager.isAttached())
```

```
      {
        editorManager.localEditorActivated((AbstractTextEditor) part);
      }
    }
  }



/**
   * handles the local activation of the given editor.
   *
   * @param editor
   *           the editor that was activated
   */
  public void localEditorActivated(AbstractTextEditor editor)
  {
    if (userManager.isDriving())
    {
      String editorId = getIdForEditor(editor);
      if (editorId != null)
        activateEditor(editorId);
      else
          [...]

/*
   * launches an EditorWrapper instance for the given editor.
   */
  private synchronized void wrapEditor(AbstractTextEditor editor,
      String editorId)
  { [..]
```

Detecting resource changes

Classes needed to detect resource changes

```
org.eclipse.core.resources.IResourceChangeEvent
org.eclipse.core.resources.IResourceChangeListener
org.eclipse.core.resources.IResourceDelta
org.eclipse.core.resources.IResourceDeltaVisitor
```

Classes to detect breakpoints and debug events

```
org.eclipse.debug.core.IBreakpointListener
org.eclipse.debug.core.IDebugEventSetListener
```

Classes to launch Run, Debug and other commands:

```
org.eclipse.debug.core.ILaunchManager
org.eclipse.jdt.launching.*
```

Useful links

http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html

http://www.dpunkt.de/leseproben/3-89864-376-X/Kapitel_3.pdf

Help from help:

- Help
  - Help Contents
    - Platform Plug-in Developer Guide
      - Reference
        - API Reference
          - org.eclipse.debug.core
            - DebugPlugin
- JDT Plug-in Developer Guide
  - Programmer's Guide
    - JDT JUnit integration
      - Observing JUnit test runs
- Reference
  - API Reference
    - org.eclipse.jdt.launching
      - JavaLaunchDelegate
      - getLaunchManager()
      - org.eclipse.debug.core.ILaunchManager.addLaunchListener()

Listening to run commands

```
LaunchManager manager = DebugPlugin.getDefault().getLaunchManager();
ILaunchListener listener = new LaunchListener();
manager.addLaunchListener(listener);
```

Eclipse' undo

The Eclipse DefaultUndoManager is linked to a global Undo/Redo mechanism.
It avails itself of the global operations stack, implemented in org.eclipse.core.commands.
Also, projects under Eclipse boast a refactoring history. Thus, refactoring by the driver might be recognized and replicated.

Intercepting editor actions

For every editor view, there is an instance of the Idocument interface. You can register a listener for changes to the document.

```
void IDocument.addDocumentListener(IDocumentListener listener)
void IDocumentListener.documentChanged(DocumentEvent event)
```

class DocumentEvent contains the changes as a Replace-Command.

Session Management

Creating a session (session data)

When users create a session, they must specify the project to be shared. The selected project is packed into a zip-file. Via a stream message the project is sent to the server. If the user chose the option to backup their project, the zip file is also stored on the local drive.

For streaming the project to the server, it is divided into chunks of 64KByte size. The server unpacks the zip-file and stores it in the XPSessionData directory.

For each project, a directory with a unique XPSessionId is generated. At unpacking, each directory and filename is registered in the Resource Master List.

Session persistence

Data pertaining to a session are stored in an xml file. The file XPairtiseSessionDB.xml servers this purpose.

Decorating a project

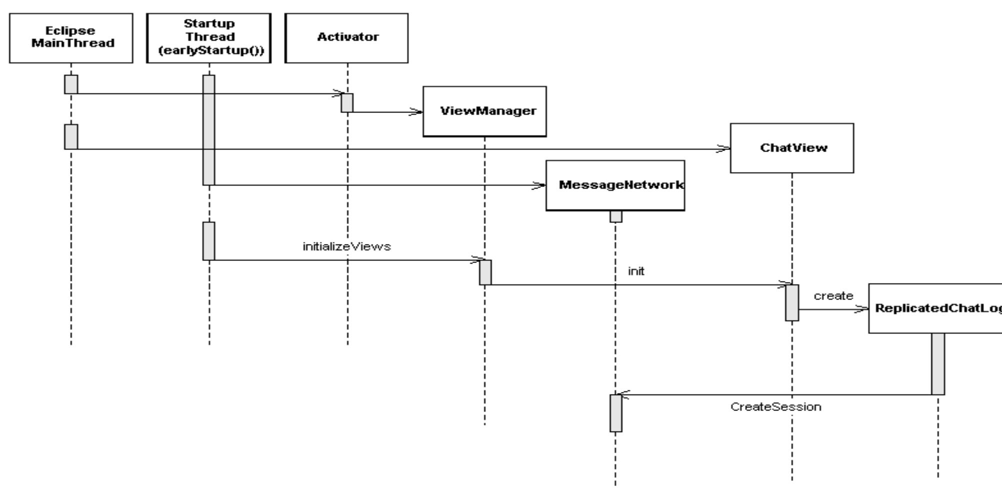
The projects' root directory is decorated by image and text. To this purpose, the XPLabelDecorator method is called when joining or leaving a session.

User gallery

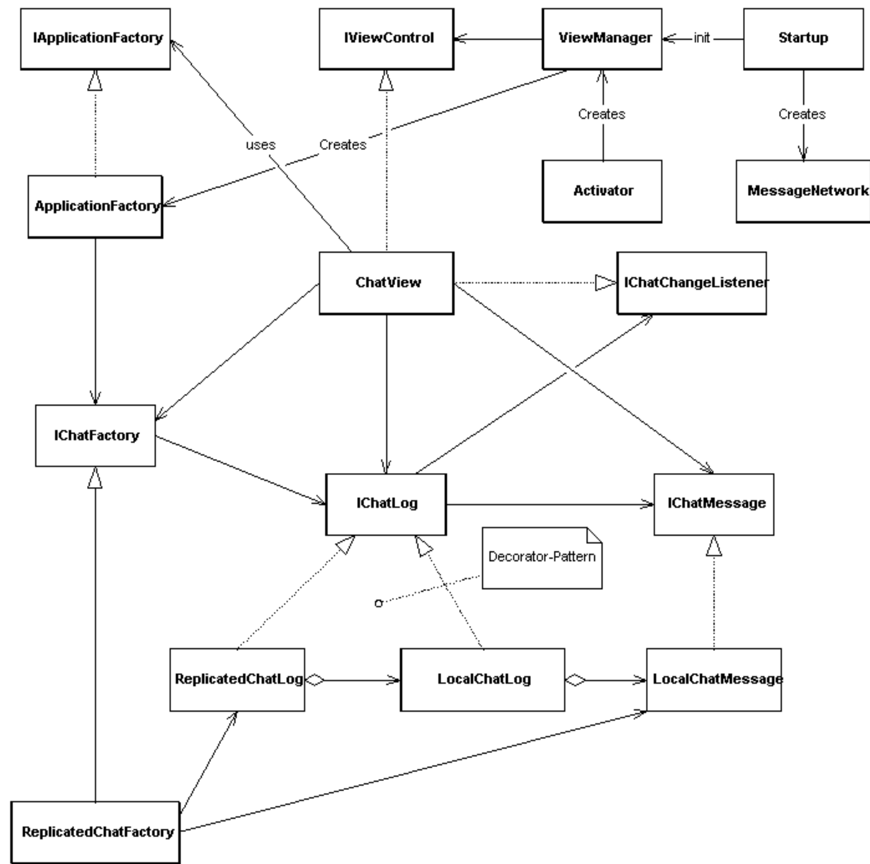
The user gallery is based on the SWT Table class. Users are entered as “TableItems" . Columns 1 and 2 have image entries. The images for the user state are managed globally in the plugins' "ImageRegistry".

Chat functionality

Chat startup



Chat class domain diagram



User Story

The user starts the chat view. The view displays the chat log. Users may enter messages and send them by pressing the send button or the enter key. Soon afterwards the view displays the entered message. The message appears in the chat views of all other connected users.

Involved classes

ChatView,ChatController,GlobalChatView,SessionChatView,IChatView,IChatController

```
public interface IChatView
{
  public void activate();
  public void deactivate();
  public void addEntry(ReplicatedChatEntry element);
}

public abstract class ChatView extends ViewPart
    implements IViewControl, IChatView
{ [...] }
```

Adding a message

A chat log is implemented as a list of entries. A new entry is appended.

```
private Vector<ReplicatedChatEntry> messageList;
```

Changes in the chatlog are notified using callbacks. For the actions an appropriate interface is designed.

```
public void addEntry(ReplicatedChatEntry element)
  {
    final ReplicatedChatEntry e = element;
    PlatformUI.getWorkbench().getDisplay().syncExec(new Runnable()
    {
      public void run()
      {
        showItem(e);
        messageList.add(e);
      }
    });
  }
```

ChatLogChangeListener belongs to the controller part. We also need a callback interface, which is defined as follows:

```
interface ChatLogChangeListener
```

```
{
  public void chatMessageAdded(ChatMessage message);
}
```

Instances of these interfaces operate on local data. For paired programming, these must be replicated. To create instances of these interfaces, we use the AbstractFactory design pattern. We need two methods , one to create chat messages and another, to get a chatlog.

```
interface ChatFactory

{
  public ChatMessage createChatMessage(String chatMessageText, String userName);
  public ChatLog createChatLog();
}
```

ChatView

Callback to create and initialize the viewer

The following method is called by Eclipse.

```
public void createPartControl(Composite parent)
  {
    {
      GridLayout parentLayout = new GridLayout();
      parentLayout.numColumns = 2;
      parentLayout.makeColumnsEqualWidth = false;
      parent.setLayout(parentLayout);
      parent.setSize(new Point(300, 200));
      parent.addControlListener(new ControlListener()
      { [...] }
```

Listener to user action

Either the send or enter button trigger “send”.

```
 private void inputTextKeyPressed(KeyEvent event)
  {
    if (event.keyCode == 13)
    {
      try
      {
        String s = inputText.getText().trim();
        if (!s.equals(""))
        {
          getChatController().enterChatMessage(s);
          inputText.setText("");
        }
      }
      catch (Exception e)
     [....]
```

```
  private void inputTextButtonPressed(Event event)
  {
    try
    {
      String s = inputText.getText().trim();
      if (!s.equals(""))
      {
        getChatController().enterChatMessage(s);
        inputText.setText("");
        inputText.setFocus();
      }
    }
    catch (Exception e)
    [....]
```

Adding the listener ***inputTextKeyPressed***

```
inputText.addKeyListener(new KeyAdapter()
  {
    public void keyPressed(KeyEvent event)
    {
      inputTextKeyPressed(event);
    }
  });
```

Derived Classes of ChatView

There exist two views for chatting: the session chat and the global chat. This fact is mapped to two classes: GlobalChatView and SessionChatView.

```
public class GlobalChatView extends ChatView

public class SessionChatView extends ChatView
```

Chat Controller

The Interface is created in the IClientApplicatioFactory

```
  public IChatController createGlobalChatController(IChatView chatView)
      throws Exception;

  public IChatController createSessionChatController(IChatView chatView)
      throws Exception;
```