DIALOGUE EXPERIMENTAL TOOLKIT

User Manual

DIALOGUE EXPERIMENTAL TOOLKIT

User Manual

P. G. T Healey
ph@eecs.qmul.ac.uk

G.J. Mills
gjmills@stanford.edu

http://www.eecs.qmul.ac.uk/research/imc/diet/about.php

Table of Contents

Before you start! 4
Introduction 4
Equipment required to run the software 4
Possible kinds of experimental intervention are: 5
Recorded data 6
Integration with SPSS / Excel / OpenOffice 6
Naturalistic interventions 6
Interfaces 6
Program versions and download 6
History 6
Getting started 7
Downloading and installing the files: 7
Sorting out the networking and starting the server and client 9
The server ..... **Error! Bookmark not defined.**
The clients ..... **Error! Bookmark not defined.**
Running a single experiment 11
To start the server 11
Running an experiment (from the preinstalled library of interventions) 12
Setting up the client machine 14
To log in a subject 17
During an experiment 19
To terminate an experiment from the server 20
Experimental data 21
During an experiment 22
Turns 22
Turns detailed 22
Contiguous Turns 23
Window policy 23
Parameters 24
To change a parameter value using the GUI 24
Script Console 24

Tutorial 1 – programming the chat tool 26
Download and install the source-code 26
Conversation Controller 27
processChatText(…) 28
Running the intervention 28
Programming a custom intervention – modifying the ConversationController object 29
Transforming turns 29
Insert artificial turns 30
Inserting an artificial turn and blocking the response 30
Final steps 30
Renaming the intervention 30
Tutorial 2 – programming the chat tool 31
Download and install the source-code 31
Creating a ConversationController object 31
Create a Template file 31
Check that it runs 32
Programming the chat tool 33
Responding to pauses in the dialogue 33
Piloting the experiment 34
Programming guidelines 37
Programming and debugging 37
Automatically loading a template you are working on to save time 37
Running the client as an applet 38
Running the client as an applet 38
Basic architecture 40
Participant 40
Client-server communication / Messages 41
Client 41
Server 41
Parameters 42
Integrating Parameters with the GUI 43
To change a parameter value using the GUI 43
Adding your own stimuli 45
Displaying stimuli as a web-page – initializing the window 45
Displaying the next stimulus in the same window 45
Displaying text only 46
Displaying multiple stimuli – avoiding building webpages for each stimulus 46

Using the server to host the images - 47
Saving stimuli information 47
Integrating a new task 49
Kinds of chat tool interface 50
Closing the chat tool windows on the clients 52
Dealing with error messages / java exceptions 53
Server – saving error messages 53
Server – displaying values in the window of the server. 53
Debugging the client – displaying values in the window of the server ...53
Debugging the client – displaying values in the window of the server ...53
Debugging 54
Dealing with crash recovery 55
Server 55
Client 55
Displaying stimuli on the client's computer 56
Libraries used by the chat tool 58
Programming the Character By Character interface 59
Following messages – the nuts and bolts – On the client 63

Before you start!

Remember, if you have any question at all – if anything doesn't make sense, either in the instructions or in the code – please email gjmills@stanford.edu. The program is designed as a general tool for researchers, so any feedback concerning its use will really help us write better documentation.

Introduction

The toolkit is a text-based chat tool for carrying out experiments on dialogue. The basic idea of the chat tool is that it intercepts participants' typed turns and selectively interferes with their content. This allows very fine-grained experimental control over what participants perceive the other participants as having typed.

The chat tool has a constantly expanding library of experiments that can be reconfigured. It also provides an extensive API that allows the programmer to design experimental interventions that are sensitive to the conversational context of the participants.

Equipment required to run the software

The following equipment is required to run experiments using the software:

3 or more computers, which can be any combination of Windows PC, Linux or Apple machines:

- 1 computer runs as **server**, and is used by the experimenter to observe the conversation. This computer runs the server software, which is typically scripted by the experimenter to interfere with participants' turns.

- 2 or more computers run the **client** software. This program runs on each participant's computer. It provides an interface that displays the conversation and allows the participants to type text to each other. There are 2 ways of loading the client software, (1) from a disk or shared folder (2) Directly from a web-page applet that is hosted on the server. This is slightly more complex as it requires setting up the server as a webserver, but makes running batches of experiments much easier.

Network connection. The clients must be able to network with the server. The server has to be able to accept incoming connections on at least one port (you might need to ask systems admin. to "open" a port on the firewall). Possible (tried and tested) setups include:

- WIFI: using the existing infrastructure of the institution
- WIFI: participants bring their own laptops to the experiment (all modern laptops should be able to run the client software and connect to the server without having to install any new software).
- WIFI: The experimenter uses own WIFI Router / switch to create a wireless network for running the experiment. This has proved especially useful, not least because this makes it easy to block access to the wider internet, to ensure that participants can't read emails /or surf the web while taking part in the experiment.
- Ethernet: Participants sit at desktop computers in institution's computer laboratory)
- Mixed: any combination of WIFI / Ethernet should work as long as the Server can accept incoming connections from the clients.

Software required:

- Java 1.5 or higher. Preferably 1.6. (usually installed as default on most computers)
- To program the chat tool, unless you use a different IDE, it is strongly recommended to use Netbeans (v6 or above) as a platform for development. The source code is distributed as a self-contained Netbeans project with all the libraries included.

Possible kinds of experimental intervention are:

- Substitution of synonyms / hypernyms / hyponyms
  - E.g. use WordNet or other resource to selectively substitute words in participants' turns.
- Introducing artificial feedback into the dialogue :
  - Artificial clarification requests, such as "What?" "so you mean?"
  - Acknowledgments, such as "ok" "ok right"
  - Other discourse markers, e.g. "so?"
- Blocking or transforming certain kinds of feedback:
  - Substituting "ok" with "hmm" or "ok, tell me more"
  - Blocking occurrences of "what?"
  - Introducing fake interruptions
- Introducing artificial hesitations and disfluencies
  - Introducing "umm" or "erm" into the dialogue
- Transforming the identity of the other participant
  - Making it appear as if the person is of different gender (e.g. use differently gendered name)
  - Make participants
- Blocking or promoting alignment

Recorded data

The chat tool automatically records the following data of each turn: the producer, the recipient(s), each word (frequency, recency), typing speed, number of edits , turn formulation time and typing overlap. This data is available at runtime to be used for generating experimental interventions that are sensitive to the immediate conversational context. The chat tool also includes the Stanford Parser, and a Java implementation of WordNet.

Integration with SPSS / Excel / OpenOffice

All the data from the experiments is saved to a CSV file (Comma Separated Values) that can be easily loaded into statistical software or Excel/OpenOffice.

Naturalistic interventions

In order to create more plausible, naturalistic interventions, the chat tools allows the interventions to "spoof" participants' typing speeds, and also introduce mistypings (typos) that are sensitive to the keyboard layout.

Interfaces

The chat tool includes a variety of customizable interfaces:

- An interfaces similar to MSN/ICQ/AIM/ Facebook chat: participants type text into a private window, and press SEND or ENTER to send their turn.
- An interface similar to UNIX Talk that displays participants' text in separate windows
- An incremental (character-by-character) interface that is especially useful for investigating incremental phenomena in dialogue.

Program versions and download

The chat tool is programmed entirely in Java (1.5 or above), and runs on all (Linux, Windows, Mac) platforms. It is released under a GPL license, and the latest version is available to download from sourceforge.

History

This is essentially the 4th incarnation of the chat tool. It grew out of the ROSSINI project (P.G.T. Healey Queen Mary University), was used by M. Purver for his thesis, was rebuilt for G.J. Mills's thesis, and then programmed as a stand-alone experimental resource as part of the DiET project (EPSRC). It has been modified and added to as part of the DynDial project (Kings College and Queen Mary University) and also as part of ERIS (FP7 EU Project, G.J Mills.)

Getting started

This section is designed to give you an idea of the chat tool without requiring you to do any programming. The instructions below are for setting up a simple experiment using one of the pre-existing templates from the library of experiments. If you want to try it out on a single computer, simply carry out all the steps listed below on the same computer.

Downloading and installing the files:

Before you can run an experiment using the chat tool, there are several steps that you must take.

- Make sure that java (1.5 or above) is installed on all the machines that are to be used:
    - Run the command prompt (in the windows start menu, select Run…, and then type `cmd`)
    - Type `java -version`
    - If there is an error message, then java is not installed on the machine at all.
    - If the java version is "1.5.x" or above (as shown in the red oval in Figure 1), then this is correct. The chat tool will **NOT** work with earlier versions (e.g. 1.4.x")
    - If java is not installed on your computer (or you have an incompatible version), please get your systems admin to install it from http://java.sun.com/javase/downloads/

Because this is academic software, and is constantly being updated, revised, fixed, etc…. it's a little bit more complex downloading the software - to download the latest version you will have to download it using SVN from sourceforge:

1. Ensure you an SVN client, such as TortoiseSVN (on windows machines). Apple machines now also include an SVN client – please look at your manual on how to use it. For more detailed instructions on how to use SVN with sourceforge, see the following documentation at: http://sourceforge.net/apps/trac/sourceforge/wiki/TortoiseSVN%20instructions

2. Download the full source code of the project by doing an svn checkout from the repository which is located at https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode.
   On a linux machine, do something like the following at the command prompt
   svn co https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode chattool

3. Make yourself some coffee and wait – it can take about 5-10 minutes to download. (it's about 160 Megabytes).

    - The following instructions assume the chattool program is saved to H:\chattool :

Sorting out the networking and starting the server

Sorting out the networking can be quite a headache – it'll depend a lot on what infrastructure you have available at your institution. It might even be easier to use your own cheap WIFI router and get participants to bring in their own laptops.

It's possible to run all the demos and **test the experiments on a single computer** (running both clients and the server on the same machine). This is how the chat tool is typically used when programming and testing interventions.

For the server to be able to connect to the clients it needs to be able to listen for incoming connections on at least one port. Currently, it is set up to listen for incoming connections on port 20000.

There are 2 ways of starting the server. Assuming you have the chattool software installed in a folder called chattool\

Method 1. (The easiest)

Navigate to the folder \chattool\dist

You should see a file there called chattool.jar

Double-click on it.

A window should appear. Click on "START THE SERVER"

An error message 'could not listen on port 20000' means that the firewall settings are too restrictive, and systems admin should be contacted to allow incoming connections on this port. Please look at instructions on the internet on how to open a port.

Importantly, if you know what you are doing and would like to change the port on which the server listens for incoming connections, the setting for this is saved in the following file:

chattool\data\General settings.xml

If you want to change this setting, shut down the chat tool program, open this file in a text editor, and change 20000 to whatever value you wish to use, and then restart the chat tool server.

You might need to play with the settings of the firewall of your computer to let the clients connect.

Method 2.

If for some reason Method 1 doesn't work (this could be for all kinds of reasons – for example you might have a slightly different installation of java)

Open a command-line window (a terminal on UNIX / MAC), navigate to the directory \chattool\
and then at the command-prompt, type the following:

```
java -cp .\dist\chattool.jar; diet.server.experimentmanager.EMStarter SERVER
```

An error message 'could not listen on port 20000' means that the firewall settings are too restrictive, and systems admin should be contacted to allow incoming connections on this port. Please look at instructions on the internet on how to open a port.

If you know what you are doing and would like to change the port on which the server listens for incoming connections, the setting for this is saved in the following file:

Importantly, if you know what you are doing and would like to change the port on which the server listens for incoming connections, the setting for this is saved in the following file:

chattool\data\General settings.xml

If you want to change this setting, shut down the chat tool program, open this file in a text editor, and change 20000 to whatever value you wish to use, and then restart the chat tool server.

You might need to play with the settings of the firewall of your computer to let the clients connect.

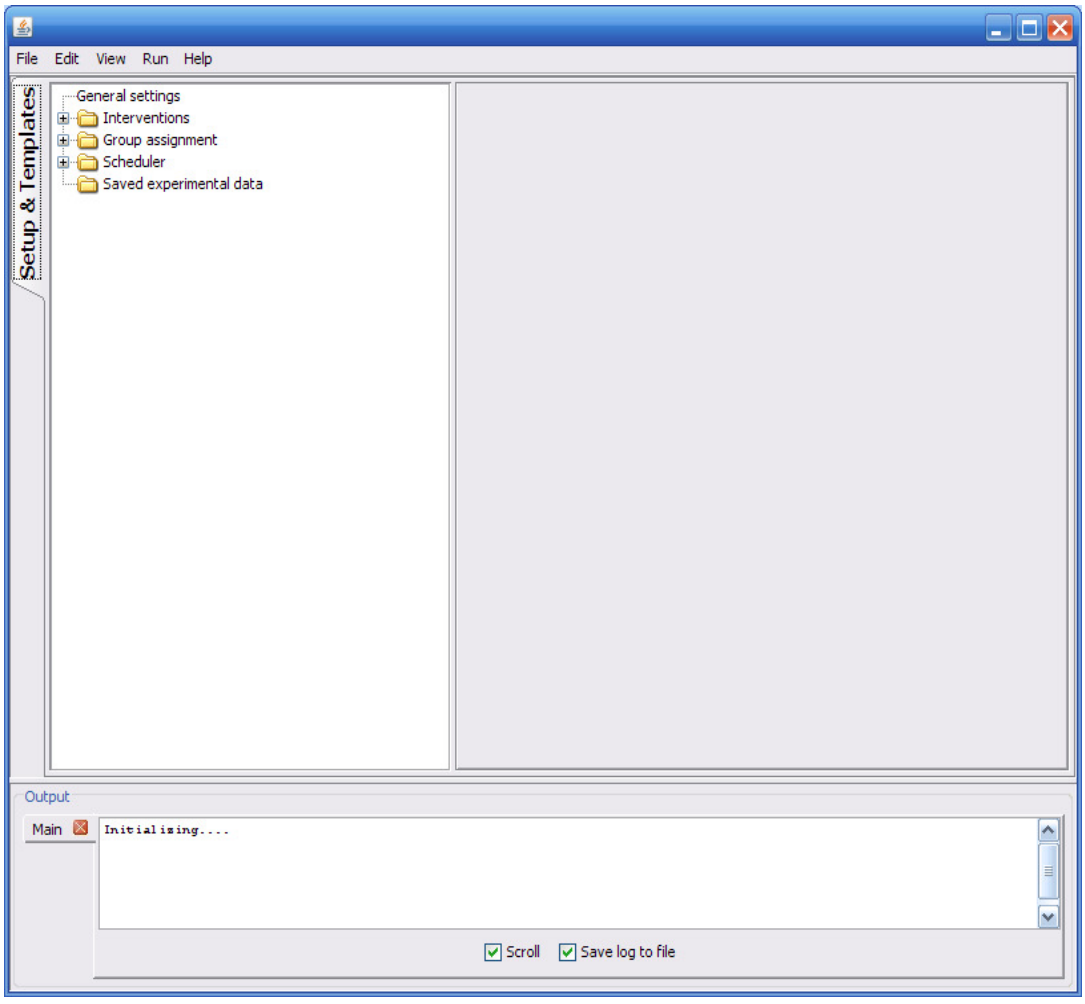
Running a single experiment

There are 2 main steps to running a single experiment. They are, briefly (1), starting the server and selecting the correct experimental settings from the main server console window, and (2) setting up the relevant clients (these are the various computers which the experimental subjects will be using)

To start the server

- **Start the server (see the section on pages** Error! Bookmark not defined.**,** Error! Bookmark not defined. **on starting the server and dealing with networking)**

- When in the relevant directory, type `server` and press `return`. This will run the batch file which launches the server console (as shown in Figure 3, below)

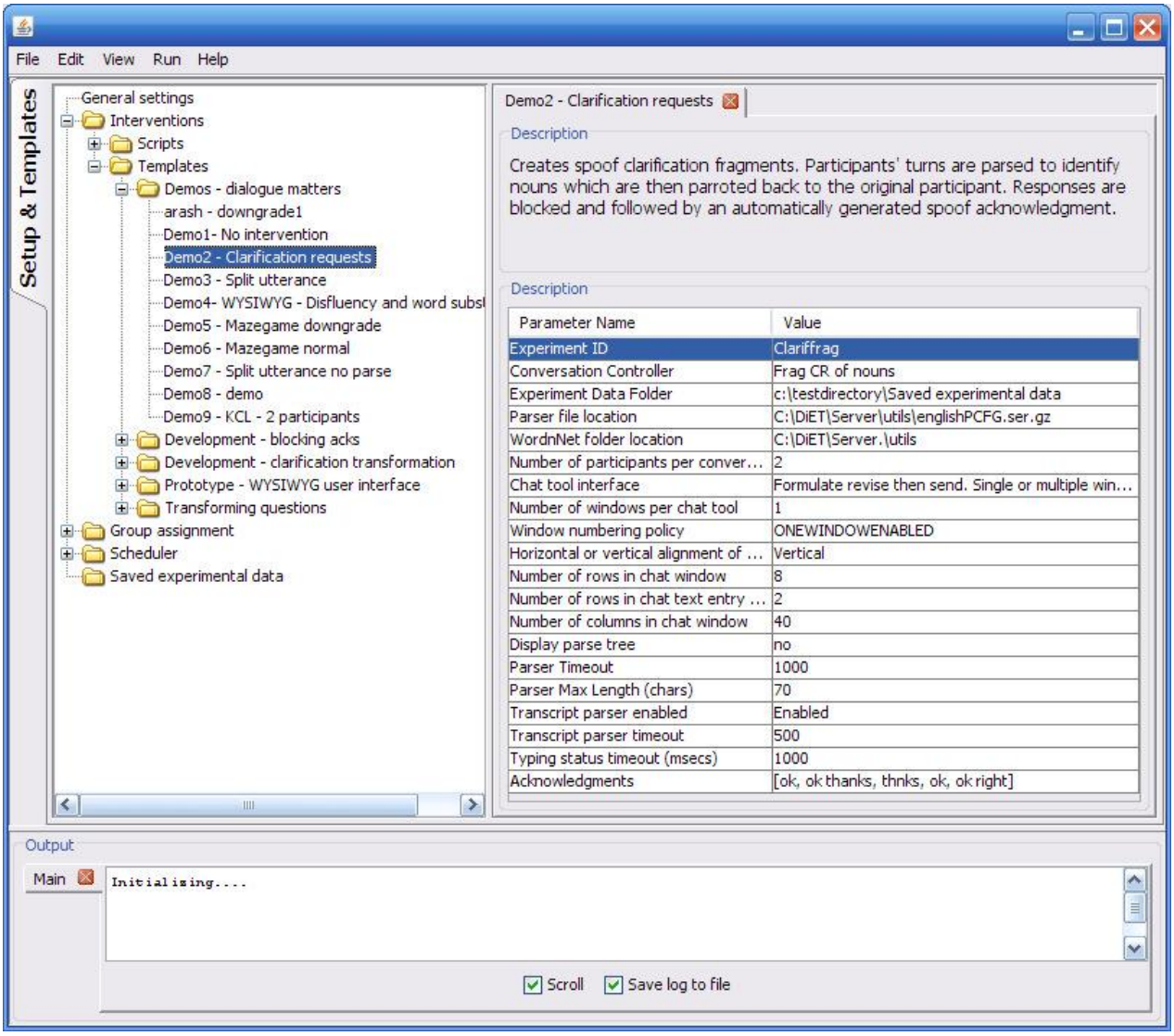


Running an experiment (from the preinstalled library of interventions)

- "Double-click" on the "Interventions" folder (top left), which will bring up a number of subfolders (the folders in the image might be slightly different, as we keep adding/ modifying the library of pre-installed experiments)

- "Double-click" on the "Templates" folder, which will bring up a list of pre-installed experiments. (once again, the folders in the image might be slightly different, as we keep adding/ modifying the library of pre-installed experiments)

- To see the experimental settings for an experiment, double-click on the experiment name. The example shown in Figure 4.
  For the moment, do not edit any of the settings in the table – these will be explained in later sections of the manual….



- To run an experiment using the settings as they are defined in the library, right click on the experiment name (the highlighted text in the left window), which will bring up a submenu.
  You will see two options. The first is to run it as an experiment, the second option is to run everything on the local machine. If you are running the chat tool for the first time, and want to get a sense of how it works, you should probably choose the second option, as it only requires one computer, and therefore won't require you to deal with network / connectivity issues…

  - **Run locally as demo (choose this option if you are trying it out for the first time)**
    This option starts the new experiment from the experimental settings, and in addition, it starts chat clients on the local machine. This is absolutely essential for designing / debugging / testing the chat tool. If you select this window you should expect to see two (or more) client windows pop up, asking for Participant ID and Usernames.

  - **Run as new experiment (this option requires starting the clients separately)**
    This option also starts the new experiment from the experimental settings and waits for clients to connect to the server in order to start the experiment..

Setting up the client machine(s)

Ignore this sub-section if you chose "Run locally as demo" above. This section deals with setting up the clients if they are running on different machines than the server.

Please note, before starting the clients:

1. The server must be up and running
2. The experimenter has to have started a Template with "Run as Experiment" which is currently waiting for clients to connect (see previous section)

Also, the total number of clients required will depend on the number of participants for that particular experimental setup. You will need to do the steps below for each client – so at least twice.

You can see how many clients you need to run, either by

- Running it first as a demo from within the server, and seeing how many client windows pop up.
- Looking at the Template at the parameter value called "Number of participants per conversation"

Also, each client will need to load the client software. To run the clients you only need a single file called \chattool\dist\chattool.jar (you don't need to copy the whole directory).
Depending on your network settings, you could

- Copy this file onto a USB drive. (If possible, don't run it straight from a USB drive, as USB connections are sometimes a bit flaky)
- Send it by email and save in a local folder.
- If the clients can see the same network directory \chattool\ which contains everything, you don't need to copy any files.

There are 4 ways of starting the chat tool (1) From within the server application by selecting "Run as demonstration" (2) By selecting the Jar file (3) By running from the command-line (4) By loading it from a web-page.

For (1), you don't need to configure anything, and (4) requires a fair amount of setting up, so this section will only deal with (2) and (3).

Methods (2) and (3) assume you are trying to connect a chat client to the server and that the chat clients are running on different computers than the server. These methods described *will* work if you're running the chat tool on the same machine, but if you are, you should really be starting the chat clients from within the server, either by right-clicking on the name of the template and selecting "Run as demonstration" or by using the NOGUI argument when starting the server (see the programming chapters for info on how to do this).

Method (2) Selecting the Jar file (Easiest method)

For the clients to connect to the server, they need to know what the server's IP address is, and also what the Port number is. Make sure that the server runs before trying to start the clients.You will need to work out what the Internet address is of the server, and configure the client to use this IP address. To find out the actual IP address of the server is , **on the server computer,** type the command :

ipconfig (in windows)

ifconfig (in apple OS)

Make a note of this number.

On the client computer, navigate to the folder that contains chattool.jar (see the previous section – each client only needs a copy of the file chattool\dist\chattool.jar to run, you don't need to copy the whole folder).

Double-click on chattool.jar

A window should appear.

There are two fields next to the button "START THE CHAT CLIENT".

Replace them with the IP address of the server, and also with the port number. (Make sure you don't leave any whitespace on either side of the numbers)

Click on "START THE CHAT CLIENT"

If this doesn't work, try the next method

Method (3) Command-line parameters

For the clients to connect to the server, they need to know what the server's IP address is, and also what the Port number is. Make sure that the server runs before trying to start the clients.You will need to work out what the Internet address is of the server, and configure the client to use this IP address. To find out the actual IP address of the server is , **on the server computer,** type the command :

ipconfig (in windows)

ifconfig (in apple OS)

Make a note of this number.

On the client computer, navigate to the folder that contains chattool.jar (see the previous section – each client only needs a copy of the file chattool\dist\chattool.jar to run, you don't need to copy the whole folder).

type the following at the command prompt: on the client machine:

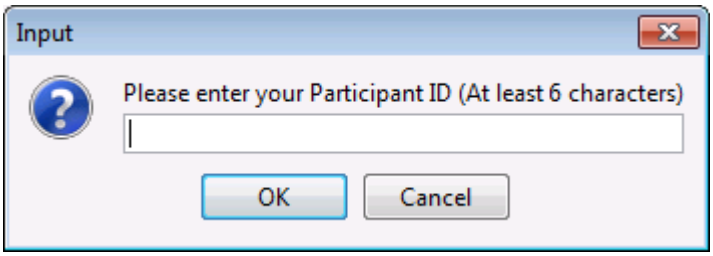
```
java -jar ".\dist\chattool.jar" client XXXXX YYYYYY
```

where

XXXXX is the IP address of the server, if you're running the and YYYYYY is the port number (typically 20000)

Logging in a client

If you have successfully started a client by whichever method, you should see a prompt for a ParticipantID (as shown below)



- The default setting on the chat tool is that it will allow all ParticipantIDs – if you want, though, it is possible to make the server allocate participants to particular experimental conditions.
  If you are trying out the chat tool, simply type any random string, and then select OK

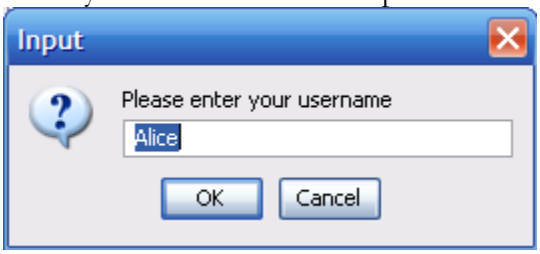
- For each client that you wish to connect to the server you will need to repeat all the steps outlined above in this section (Setting up the client machine)

To log in a subject

- After each subject has entered their participantID at the prompt and pressed "OK" or Return, another prompt will appear, for a username, as shown in Figure 6 below.

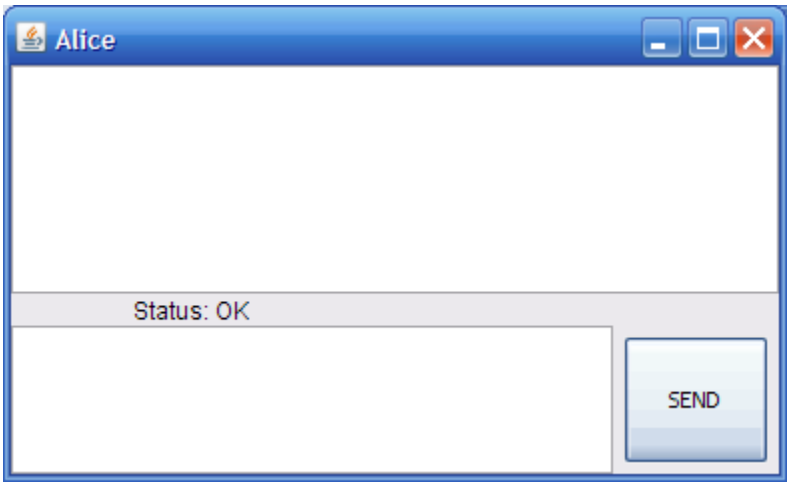
- The name entered here will be the one which their fellow participants see to identify them in the chat window (their nickname). This also ensures a further level of anonymity for the subjects when it comes to analysing the chat logs.

- Once they have entered these two pieces of information (which the server uses to identify them), a chat



  window, as shown in Figure 7, will appear
  (The size of the window is defined in the experimental settings, which is the file that was right-clicked and followed by "Run locally as demo" / "Run as experiment" in the instructions above)

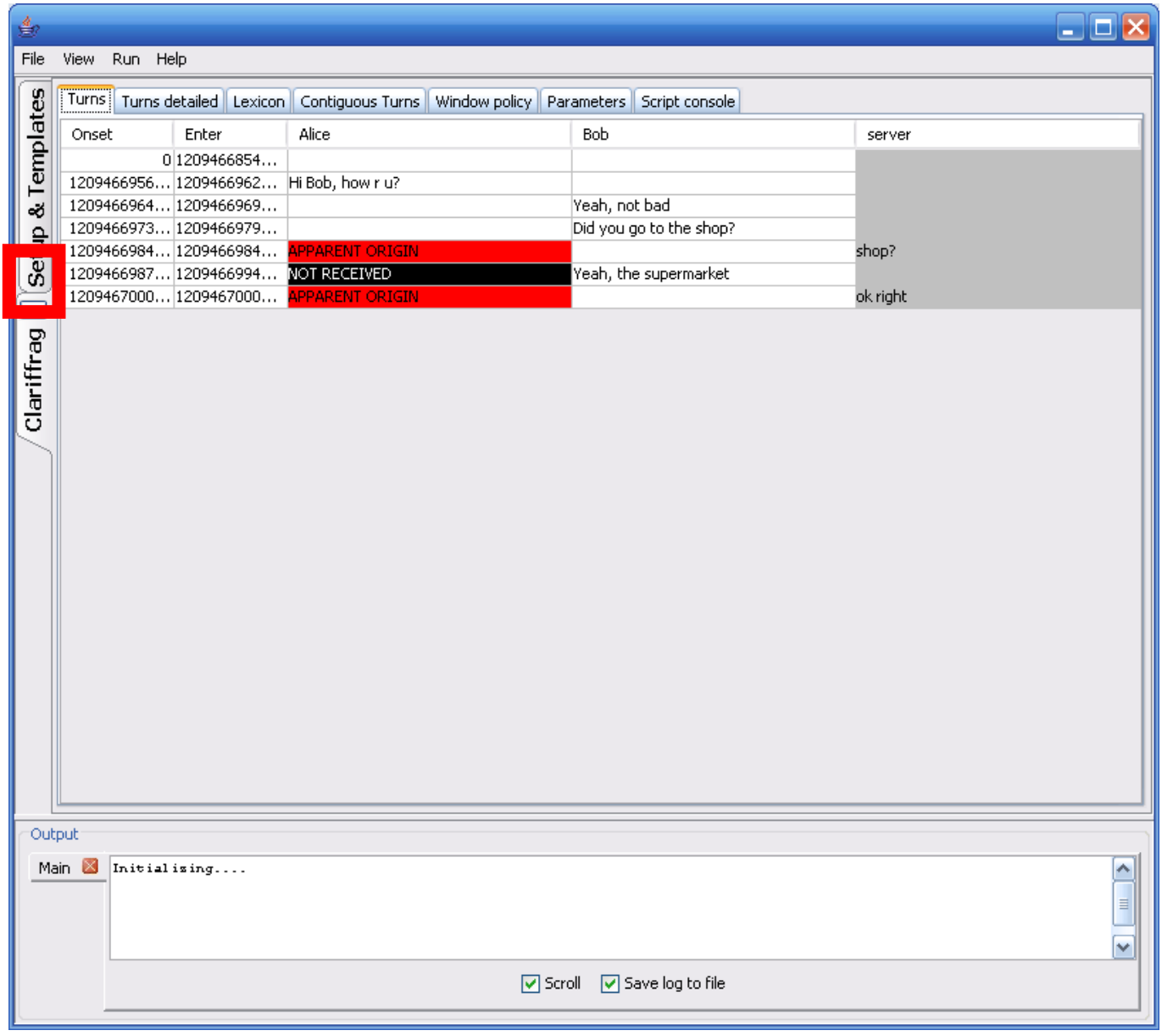
Subjects should be asked to wait until all participants are logged on before they begin typing as other participants will obviously not see any chat if they log on after the text has been transmitted.



During an experiment

On the server machine, there are several factors you can keep an eye on or update whilst an experiment is in progress

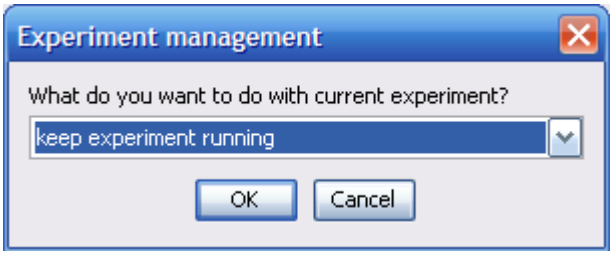
- To look at details of turns and interventions as they occur, select the experiment name from the tabs down the left hand side of the server console, as shown below.
- Other data can also be viewed, by selecting the tabs along the top of the console window.



To terminate an experiment from the server

- Select the tab relating to the experiment (the tab on the left hand side of the screen) In the example above, this would be the tab saying "Clariffrag".

- On the tab is a button with three red dots on it. (On the previous image of the chat tool server window there is a red square showing where to find it). Click on it, this will bring up an experiment management dialogue box (see below)



- Select the required option from the drop down box. Options include retaining the experimental tab open or closing it, and forcing the clients to terminate or not. This is useful if there is a time limit on the experiment, or a task has been completed, as it will log out all clients at exactly the same time.

Experimental data

- All the experimental data is saved to a folder / directory.

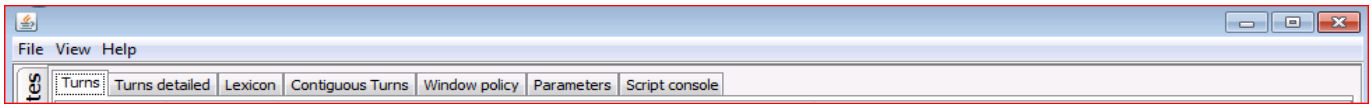
- The directory where it is saved is \chattool\data\Saved experimental data\xxxxname\ where xxxx is a sequential number of the experiment and name is the identifier of the intervention. This name can be defined in the parameters before running the experiment

- This includes all key presses, as well as a more tractable file containing the most regularly used data, for example, the times a person started typing, when they hit the return key, and what they typed

- The most important files is saved as "turns.csv". This file is stored as a CSV file, with the "|" character (ignore the speech marks) used to delimit the categories. This file opens very straightforwardly in openoffice. / SPSS

- Note, also, that the directory contains a copy of the settings to run the experiment. The experiment can be configured to save data here, in case there's an equipment problem and the experiment needs to restart from where it left off (see later sections on how to deal with crashes/restarting)

During an experiment



Each experiment that is running (it is possible to run many experiments simultaneously) is associated with a set of windows – shown above – that can be opened by clicking on their respective tabs. They are

Turns

This window shows the turns of each participant. It also displays who received which turn. All that is shown is the origin of the turn and who received the turn. The participants are listed across the top of the window in order to make it clearer visually who is typing.

Turns detailed

This table shows the information from each turn. The following information is stored:

- **Sender**
  The username of the participant who typed a turn. For interventions generated by the server the Sender is saved as "server".

- **Onset**
  The time of the first keypress of a turn (in milliseconds from the beginning of the experiment)

- **Enter**
  The time of the last keypress (when the participant presses "ENTER" or SEND). Also in milliseconds from the beginning of the experiment)

- **E-O**
  The typing time, which is Onset subtracted from Enter

- **Speed**
  Typing speed (characters per second)- The number of characters of the turn divided by the typing time

- **App. Origin**
  This is the apparent origin of the turn. So for normal turns the apparent origin and the sender will be the same, whereas spoof turns will typically have "server" as origin, but then the username of one of the participants as apparent origin.

- **Text**
  This is the text typed by the participant

- **Recipients**
  This is a list (comma separated) of all the participants who received the turn. Typically this is all other participants of the conversation.

- **Blocked**
  This records whether or not a participant had their interface blocked by the server while they were formulating their turn (this is so that these kinds of turns can be excluded from analysis – as these turns will typically have a lot of noise in them -)

- **KDels**
  This measure records the number of times participants pressed the delete key while formulating their turn.

- **DDels**
  This measure is similar to KDels, except it records the number of actual characters deleted from the text (selecting a block of text and deleting it would be recorded as a single KDel, but would be recorded by this measure as having the size of the text that was deleted)

- **DIns**
  This measure records the number of characters inserted into the text at non-turn-terminal position. Put simply it records how many self-edits (characters inserted) are performed on the text.

- **DDels*N**
  This creates a score summing up the locations of each delete from turn-terminal position

- **Dins*N**
  This creates a score summing up the locations of each insert from turn-terminal position

- **TaggedText**
  This is generally only used for demoing / debugging purposes. It displays output from the Stanford Parser.

Contiguous Turns

This window shows the same info as **Turn**, except that it concatenates multiple turns by the same speaker into a single turn.

Window policy

Ignore this – it was originally used to determine in which window each participant sees each other's text. However nearly all experiments use a single window design, so the values are typically set to zero.

Parameters

See the section on parameters – these parameters allow you to display data/values from the ongoing experiment and then change them while the experiment is running (this can be useful for piloting experiments).

See the section on parameters on how to use them / program them.

To change a parameter value using the GUI

To prevent accidental changing of parameter values during an experiment (e.g. by pressing the wrong button), changing a parameter value takes 4 steps.

1. Select the row of the parameter you want to change
2. Double click the 3rd column (new value). A text entry field should appear
3. Type in the new value into the textfield. Press Enter. In the fourth column, the SET button should become enabled
4. Press the SET button

Script Console

This gives you a BeanShell Interpreter window that allows you to type in Java code and have it executed by the chat tool. This can be essential for controlling aspects of the experiment while it is being piloted, without requiring you to create a user interface of buttons / text entry fields etc.

The conversation controller object of the particular experiment being run is available as an object called cC.

For example,

Say you have a method in your ConversationController object that you would like to invoke during an experiment, e.g. **sendCompliment (String participantname)**

To call this method from the BeanShell window, all you need to do is type the following

cC.sendCompliment("NAMEOFPARTICIPANT");

Tutorial 1 – programming the chat tool

This tutorial is a set of instructions on how to create your own experimental design and then run the experiments.

The chat tool is designed so that you should only have to modify one file to design your experiments: All the messages generated by the clients are routed through a ConversationController object that determines whether the messages are relayed normally, or blocked, transformed or delayed. The ConversationController object can also generate artificial messages that appear to originate from another participant. The ConversationController object also intercepts other information, such as keypresses, from all the clients.

Download and install the source-code

The sourcecode is stored as a netbeans project at sourceforge.net. Although it can be annoying having software that forces you to use a particular IDE – it was decided to use this, as it allows all the directories and paths to be set as project parameters (making it easier to run / install / copy). Though by all means feel free to use your favourite programming envnironment to program the chat tool.

To install the chat-tool

4. Ensure you have netbeans (version 6.0 or higher) installed.
5. Ensure you have java (1.5 or higher). Installing netbeans should also install the correct version of java.
6. Ensure you an SVN client, such as TortoiseSVN (on windows machines)
7. Download the full source code of the project by doing an svn checkout from the repository which is located at https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode.
   On a linux machine, do something like the following at the command prompt
   svn co https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode chattool

This will download everything into a directory called 'chattool'. Note that this is quite a large file transfer that will take a while to download.

This directory is a netbeans project. It should load automatically in netbeans, without requiring you to link up any libraries or paths. You should be able to run the chat server program “out of the box” by selecting “Run”.

Conversation Controller

Go to the diet.server.ConversationController package.

In this package there are five classes called CCCUSTOM1.java, CCCUSTOM2.java, CCCUSTOM3.java... These five classes are basic "stubs" that have the default behaviour of relaying messages between clients, without performing any interventions

Select CCCUSTOM1.java in the editor.

Some of the important methods in objects of this class are:

1. **processChatText(…)** This method is called whenever a message is received from a participant's chat client. It supplies both the individual message and the identity of the origin. In CCUSTOM1, this method simply relays the message to the other participant.

2. **processWYSIWYGTextInserted()**This method is called whenever a participant edits the text in their private turn-formulation window. This allows the experimenter to have access to the contents of the private turn-formulation window **before** the participant presses SEND or presses ENTER. This method is used to transmit to the other participants that that particular participant is typing (As in MSN/AIM/ICQ). Of course, this functionality can also be changed.

3. **processKeyPress()** This method is called whenever a participant presses a key. Typically this happens when they are editing the text in their private turn-formulation window, so it is usually called with **processWYSIWYGTextInserted()**. This method can also be used to transmit to the other participants that that particular participant is typing (As in MSN/AIM/ICQ). Of course, this functionality can also be changed.

4. **processLoop()**This is a method that operates on a separate thread, by default it checks to see whether a participant is no longer typing - and if so then sends a message to the clients that that participant is "idle". This functionality / and the message can also be changed. This method is typically used to trigger interventions after a timeout.

processChatText(...)

The default implementation has the following functionality:

```
pTurnsElapsed.setValue(((Integer)pTurnsElapsed.getValue())+1);
super.expSettings.generateParameterEvent(pTurnsElapsed);
```

Although this looks a little complex, all this does is augment a counter each time a client sends some text - simply counting the number of turns in the dialogue. *pTurnsElapsed* is a Parameter object that is displayed in the user interface of the chat tool (see section on parameters)

```
c.relayTurnToAllowedParticipants(sender,mct);
```

This method takes the turn and sends it to all the other participants, which is what you'd expect from a standard chat tool!

```
c.sendLabelDisplayToAllowedParticipantsFromApparentOrigin(sender,"Status: OK",false);
```

This instantly informs all the clients who received the message that the participant who typed the message is no longer typing.

Running the intervention

In the library of Interventions, there is a folder containing empty templates that are linked up to the ConversationController objects CCUSTOM1, CCUSTOM2, CCUSTOM3…etc

Start the chat-tool server, and navigate to \Interventions\Customizable (in the Setup & Templates server window)

If you double-click on Interventions\Customizable\CCustom1 the second row of the settings should read "CCUSTOM1". This is a parameter that states which ConversationController object the server should load for this experimental template. Note that the ConversationController object has the same name as the java file you looked at in the earlier step.

Underneath this parameter there are other entries for the kind of chat window, its dimensions, and also for parser settings. For the moment leave these unchanged.

Follow the instructions from the section (getting started) on how to run the experiment – to run it as a demo, simply right-click the name of the file and select "Run locally as demo". Test it to see that the clients relay text to each other. If that works, then the next step is to program a custom intervention

Programming a custom intervention – modifying the ConversationController object

The next subsections show how to transform turns, block turns, and also insert artificial turns into the unfolding dialogue. For illustrative purposes, the examples are simply "toy" implementations

Transforming turns

This section shows how to add an intervention to transform participants' turns, in this case appending "...which is great" to every turn.

Return to the *processChattext(..)* method in CCCustom1.java. The method is currently:

```
pTurnsElapsed.setValue(((Integer)pTurnsElapsed.getValue())+1);
super.expSettings.generateParameterEvent(pTurnsElapsed);

c.relayTurnToAllowedParticipants(sender,mct);
c.sendLabelDisplayToAllowedParticipantsFromApparentOrigin(sender,"Status: OK",false);
```

To transform the turns, the line:

```
c.relayTurnToAllowedParticipants(sender,mct);
```

needs to be removed/commented out. In its place, a new command needs to be called that takes the text in the message, appends it, and sends it.

First, the chat tool needs to be told that the incoming turn in its original form will not be relayed to the other participants. **This step is essential for all interventions!!!!!!!**

```
c.setNewTurnBeingConstructedNotRelayingOldTurnAddingOldTurnToHistory(sender, mct);
```

This command is a bit long - and will probably be abbreviated to a shorter command in future versions.

The next command needs to construct the new message, this can be done with straightforward string concatenation.

```
String newTurn = mct.getText()+"...which is great";
```

Finally, the chat tool needs to be told to send the message to the other participants.

```
sendArtificialTurnToAllOtherParticipants(sender,newTurn);
```

The final code in processChatText(..) is:

```
turnsElapsed.setValue(((Integer)pTurnsElapsed.getValue())+1);
super.expSettings.generateParameterEvent(pTurnsElapsed);
c.setNewTurnBeingConstructedNotRelayingOldTurnAddingOldTurnToHistory(sender, mct);
String newTurn = mct.getText()+"...which is great";
sendArtificialTurnToAllOtherParticipants(sender,newTurn);
```

If you run this program by right-clicking on it in the main DiET experiment manager window and choosing "run locally as demo", the chat tool will run the modified intervention.

Insert artificial turns

Essentially, inserting an artificial turn is no different. If the following line:

```
String newTurn = mct.getText()+"...which is great";
```

is replaced with

```
String newTurn = "I don't know";
```

this will replace every single turn with "I don't know".

Of course, you could customize this so that after every X turns an artificial turn is inserted.

Inserting an artificial turn and blocking the response

Have a look at CCFRAG.java in the ConversationController package. This experimental manipulation generates artificial clarification requests, captures participants' responses without relaying them, and then sends an acknowledgment. It shows how to make use of the "is typing" status bar on the chat clients to make more plausible interventions, including the use of artificial error messages to mask the CR recipient's sub-dialogue with the server.

Final steps

Renaming the intervention

Now that you have a fully functioning intervention - it is best to rename it. Netbeans has a really straightforward way of renaming classes - simply right click on CCCUSTOM1.java in netbeans class browser window, choose "Refactor" and then "Rename" and then a name, e.g. "CCToyImplementationDemo.java". To distinguish these classes from other classes in the DiET source code, all ConversationController objects should be prefixed with "CC" (although obviously this is not strictly necessary - it just helps with setting the parameters).

Tutorial 2 – programming the chat tool

This is another tutorial – it covers similar ground to that of the previous tutorial

Download and install the source-code

The sourcecode is stored as a netbeans project at sourceforge.net. Although it can be annoying having software that forces you to use a particular IDE – it was decided to use this, as it allows all the directories and paths to be set as project parameters (making it easier to run / install / copy). Thoughm by all means feel free to use your favourite programming evnironment to program the chat tool.

To install the chat-tool

1. Ensure you have netbeans (version 6.0 or higher) installed.
2. Ensure you have java (1.5 or higher). Installing netbeans should also install the correct version of java.
3. Ensure you an SVN client, such as TortoiseSVN (on windows machines)
4. Download the full source code of the project by doing an svn checkout from the repository which is located at https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode.
   On a linux machine, do something like the following at the command prompt
   svn co https://dietsourcecode.svn.sourceforge.net/svnroot/dietsourcecode chattool

This will download everything into a directory called 'chattool'. Note that this is quite a large file transfer that will take a while to download

Creating a ConversationController object

In the package diet.server.ConversationController, create a new ConversationController object. This is the main object that, as the name suggests, controls the conversation.

The easiest way to do this is to extend DefaultConversationController.

The (non-essential) naming convention is to start the name of the conversation controller with CC

Once you have created the new ConversationController object, e.g. "CCSEQUENCE.java"

Create a Template file

Navigate to the home folder/directory of the chat tool, and then go to the following sub-folder/sub-directory

../data/Interventions/Templates

There you will see a whole list of templates with xml suffixes. These are text files containing basic settings for different experimental setups (as the names suggest). Although they have xml suffixes, the files are not formatted in xml. Instead they are much simpler text files containing parameters that are used as settings for the experiment.

OPEN the 01-No intervention.xml file with notepad (or any other simple text-editor) and replace CCMESSENGERNORMAL with the name of the ConversationController object entered in step 1 above.

Then, SAVE it to a new file, eg. 30 – SEQENCES.xml

Note that this file contains Parameters, some of which are essential for running an experiment, for example to set the number of participants for each condition, or the dimensions of the chat window.

See the section on Parameters.

Check that it runs

Start the chat tool and navigate in the window to Interventions/Templates/

You should see a copy of the template you created in the previous step

Click on it, it should bring up a table – look at the settings – these are parameter objects that get used to customize some aspects of the experiment

(Note, that this means if you make a template for your experiment, you can make it as generic / specific / configurable) as you wish.

Have a look at the section on Paramaters .For the purposes of this tutorial, simply look at the entry "Number of participants per conversation".This is used by the chat tool server to determine how many windows get opened when selecting “Run locally as demo”

Right-click on it and select "run locally as demo". It should start two chat windows – try to see if the chat text gets relayed between the two chat windows.

To close the two chat windows from the server, follow these instructions:

There should be two tabs in on the left-hand side of the screen, with the text at 90 degrees. The first should say "Setup&Templates", and the second should be underneath it. On the second one there's what looks like a button with three red dots on it..Press it. It should bring up a button saying : "What do you want to do with current experiment?"

Choose the last option: "stop experiment and discard window and stop clients"

This option closes down the clients, and also (tries to) close down the interventions running on the chat tool server.

Programming the chat tool

There are two main ways of programming interventions. The usual way is triggering an intervention based upon something that happens in the dialogue, for example by transforming an utterance, inserting a fake question in response to a turn uttered by a speaker. The other way doing something when "nothing" happens. say for example you want to trigger an intervention when there is a pause in the dialogue. For this kind of case, use the processLoop() method, described in the next subsection

Responding to pauses in the dialogue

The first thing to add is the "Is typing" message on the status bar of the chat tool client. Redo step 3 above and have a look between the window where you formulate your text before pressing send, and the window where the text appears. In the middle is a status bar that displays an "is typing" message. Notice that when you type in the text-entry field, it says "is typing" on the other client chat tool window (and vice versa). When you press SEND/Enter, the message disappears.

Notice, though that if you don't press SEND, the message persists. It would be nice to have it so that the message disappears after you haven't typed for set amount of time (e.g. add a timeout of 1 second).

To do this, edit your ConversationController object, and create a new method that overrides processLoop(). This method is called by a separate thread, in a permanent loop (as the name suggests)

```
public void processLoop() {
super.processLoop();
c.resetFromIsTypingtoNormalChatAllAllowedParticipants(super.getIsTypingTimeOut());
}
```

This will add the timeout functionality to the "is typing message"

The reason this is not included by default in the chat tool is that most interventions interfere with what the participants think the other participant has typed – and therefore this "is typing" message will need to be manipulated by the experimenter – i.e. by you – and judicious use will make the interventions you create much more plausible.

Of course you could do something else, say for example you want to set it up so that whenever one of the participants hasn't typed for longer than a fixed time it sends a "prod" to the participant to wake up (that the participant thinks comes from the other participant)

```
public void processLoop(){
    super.processLoop();

    Vector v = c.getParticipants().getAllParticipants(); // Get list of all the participants
    if(v.size()<2)return; // Need to be at least two participants for this to work
    for(int i=0;i < v.size();i++){ // Iterate through lists of participants
        Participant p = (Participant)v.elementAt(i);
        if(!p.isTyping(6000)){ //Found a participant who hasn't typed for 6 seconds
```

```
            int randindex = new Random().nextInt(v.size()-1);
            Participant apparentOrigin =
           (Participant)c.getParticipants().getAllOtherParticipants(p).elementAt(randindex);
            //This selects a random other participant (essential if the group is larger than 2) who
            will be the artificial origin of the turn
        c.sendArtificialTurnFromApparentOriginToRecipient(apparentOrigin, p, "Wake up!");
       //This sends the artificial turn to the participant who hasn't responded
    }
}
}
```

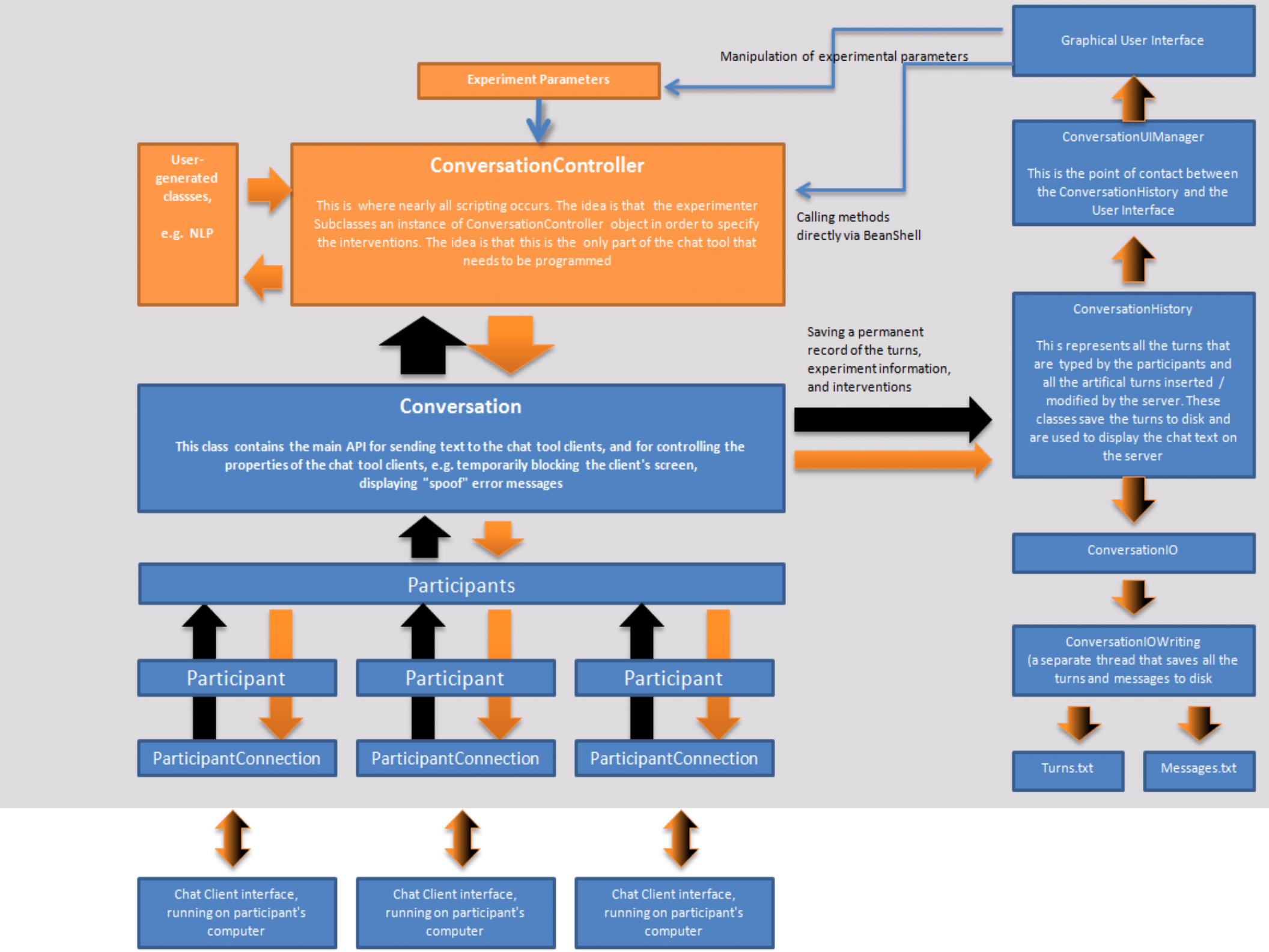
Of course, this method can be used to put in fake interruptions by apparent overhearers – artificial turns – it could be used as the basis for interventions in the task..

Piloting the experiment

See the descriptions / steps of Tutorial 1, as they are identical

Architecture of the server

The next page shows the main classes of the server (these are the essential classes that that are loaded when running an experiment)


Graphical User Interface
Manipulation of experimental parameters
Experiment Parameters
ConversationUIManager
This is the point of contact between the ConversationHistory and the User Interface
User-generated classses,
e.g. NLP
ConversationController
This is where nearly all scripting occurs. The idea is that the experimenter Subclasses an instance of ConversationController object in order to specify the interventions. The idea is that this is the only part of the chat tool that needs to be programmed
Calling methods directly via BeanShell
ConversationHistory
Thi s represents all the turns that are typed by the participants and all the artifical turns inserted / modified by the server. These classes save the turns to disk and are used to display the chat text on the server
Saving a permanent record of the turns, experiment information, and interventions
Conversation
This class contains the main API for sending text to the chat tool clients, and for controlling the properties of the chat tool clients, e.g. temporarily blocking the client's screen, displaying "spoof" error messages
ConversationIO
Participants
ConversationIOWriting (a separate thread that saves all the turns and messages to disk
Participant
Participant
Participant
ParticipantConnection
ParticipantConnection
ParticipantConnection
Turns.txt
Messages.txt
Chat Client interface, running on participant's computer
Chat Client interface, running on participant's computer
Chat Client interface, running on participant's computer

Programming guidelines

Programming and debugging

Automatically loading a template you are working on to save time

As you may have noticed, starting up the chat-tool and debugging an intervention involves

1. Starting the server
2. Navigating to the Template
3. Right-clicking the template and selecting "Run locally as demo"
4. Entering the login details of the clients

To condense the first 3 steps down into a single step, it is possible to start the chat tool so that it automatically starts, you can do this, by using the following arguments

Java –jar "chattool.jar" NOGUI TEMPLATENAME

Remember

- The argument TEMPLATENAME assumes it is a file that is in the Interventions/Templates directory.
- The parameter that determines how many clients to start is loaded from the Template
- The parameter determining on what port to listen on the server / on what port the client should try to connect to the server is loaded from the GeneralSettings.xml file

Netbeans allows you to do this automatically: The netbeans project file comes preloaded with two project settings GUI vs. NOGUI (graphical user interface vs. no graphical user interface). You can switch between them by selecting Run-Select Project Configuration. You would probably want to add your own customized configuration.

Running the client as an applet .

The chat tool can easily be run as an applet. The chat tool includes two webpage templates:

The first is a simple HTML template that loads the applet, and also allows the experimenter to specify (1) the server address (2) the server port number in the HTML file.

The second is a PHP script that automatically detects the IP address of the server. This is very helpful, as networks often change the IP addresses of computers that are on the network – if the clients load this webpage then you won't need to change the webpage each time the IP address of the server changes (on some networks this can be as frequent as occurring each time you reconnect the server to the network).

On the Server, the main applet class is diet.server.experimentmanager.EMStarterApplet.java.

To run the client as an applet from a webpage hosted on the server

- You need to install webserver software on the server. For example, Windows has IIS.
- Setup the directory **`\chattool\webserver`** as the main directory for the website.
- Setup the directory **`\chattool\dist`** as a virtual subdirectory with virtual name `jarfile`
- Make sure that the firewall allows incoming http connections.
- Make sure that your browser can run Java applets – it might be advisable to update to the latest version as well.
- If you're using the PHP template,
    - Make sure that the webserver allows PHP scripting (There are plenty of detailed instructions on the web). If your host institution doesn't allow php, use the HTML template.
    - Make a copy of it to the webserver directory, and and open that file in a text editor:, and edit the port number parameter.
- If you're using loginHTML.HTML,
    - Make a copy of it to the webserver directory, and and open that file in a text editor:, and edit the parameters of the IP address of the server and the port number.
- This will loads EMStarterApplet .java, which should try and connect to the server.
- Note that using the chat tool as an applet requires the IP address of the connection to the chat server and the IP address of the connection to the webserver to be the same – this is because of java network security. (You might get away with using different IP addresses for the webserver and for the actual Chat client server, but Java will probably complain with a Network Security Exception)

BE CAREFUL WHEN DEBUGGING!

One of the problems with debugging the chat tool when you run it as an applet is that the browsers (currently all browsers have this annoying feature) cache the class files…..this means that if you change the source code, recompile it and hit refresh on the browser, the browser will, in all likelihood simply reload the original, unchanged, uncorrected

version. This happens even if you turn off caching (at time of writing this, on Firefox)! This can drive you up the wall! One unwieldy solution to this problem is, each time you make changes (1) rename the filename of the java applet in the dist directory, e.g. change /chatttool/dist/chattool.jar to chattool1.jar (2) change the filename of the webpage, e.g. \webserver\index.php to index2.php (3) edit this file so that it loads chattool2.jar. This can be really unwieldy and annoying…

Basic architecture

The chat tool is a straightforward Server/Multiple clients application. The server constantly listens for incoming connections from a client.

On establishing a connection, the server creates a new ParticipantConnection Thread, while querying the ExperimentManager object whether the participant (email, username) is already associated with a particular experiment.

> If yes, the participant is assigned to the existing experiment.
>
> If not, the ExperimentManager instantiates a new experiment (a Conversation object), and assigns the participant to the experiment.

On being associated with an experiment, every message received from a Client is relayed by the ParticipantConnection object to the Participants object queue.

The Conversation thread associated with the experiment constantly queries this queue for incoming messages..

On retrieving a message, the Conversation method invokes the corresponding method in the ConversationController object.

The ConversationController object determines, on the basis of pre-programmed detection rules, whether to relay the message to the other participants, transform the turn, or send an artificial "spoof" message to the other participants.

After determining what to do with the message, the ConversationController method invokes methods in Conversation and Participants that send (default case) the text to the other participant, or send artificial turns to the participant.

The text from the participants' messages is automatically saved by the ConversationHistory object into the directory / folder associated with the experiment.

Participant

A participant can only be associated with a single conversation object.

Client-server communication / Messages

All the information sent between the client and server is sent packaged in Messages (There is no remote method invocation – If you don't know what that is, don't worry, it's intended to keep programming the chat tool as simple as possible).

To gain a sense of how this works, look at the diet.Message package, which contains all the messages sent between client and server

Client

The client has an object called ConnectionToServer that has a permanent loop that receives messages (commands/text/etc..) from the server, and that also packages information (typically from the chat window) into messages and sends them to the server.

Server

The message pass through the following objects (in order) before they can be manipulated

ParticipantConnection – lowlevel network / connectivity

Participant – The class identifying the participant

Participants – All the participants taking part in that experiment

Conversation – this is the central, main object associated with the interaction between the participants

ConversationController – this is the main object that determines what happens to the messages, how they get relayed between participants. The idea is that in nearly every task you program, you should only have to program this object.

--After this, the messages get passed back down again, via calling methods in Conversation or Participants that send the messages to the clients

Note, there is also a relatively independent set of classes in ConversationHistory, that save all the messages into CSV files that can be read by Excel / OpenOffice / CSV

Parameters

The idea behind parameter objects is to have settings (Strings/lists/numbers) that can be saved as settings, customized for particular experiments, modified by the experiment as it runs, and also displayed in the server window.

Navigate to \chattool\data\Interventions\Templates

This is a directory that contains templates for experiments. Each template contains parameter settings, for example detailing which chat interface to use or other settings. See below for a description of the format of these

When running an experiment, all the parmaters associated with a template are passed to the ConversationController object, which then uses them, say for example as parameters to determine when to trigger interventions, which interventions to choose, how frequently to trigger the interventions, how long to run the experiments for. They are also quite helpful for adjusting settings whilst piloting an experiment. The list of parameters associated with each experiment are stored in an ExperimentSettings file which is passed to the ConversationController object on instantiation.

Each ConversationController object can also add parameters to the list, that you can then see in the table of Parameters whilst running the experiment

Depending on which setup you use, some Parameters are essential. So for example, each Experiment, on initializing and assigning participants to a "Conversation" object, will need to know how to set up the chat tool windows of the clients (width * height).

Some of the settings are slightly obsolete – your best bet is to simply copy a pre-existing template and adapt it for your own use, as shown in the tutorial.

"Number of participants per conversation".This is used by the chat tool server to (1) determine how many windows get opened when selecting "Run locally as demo"

The parameters are stored as text in xml files, e.g. CCUSTOM1.

When initializing an experiment, these parameters are loaded and used by the server to determine things, such as which chat interface the clients use, the size (width*height).

These parameters are also sent to the ConversationController object, where they can be read / manipulated by the program. The parameter objects are also visible in the main window – look at the horizontal tabs of the experiment.

Format of the Template files (that contain the parameter information)

Because we were in a hurry and the XMLEncode / Decode of previous Java versions was notoriously buggy, we decided not to use XML to save the parameters to disk. Load "01 – No intervention..xml" . So that the text doesn't appear garbled, please use a text editor with no Linewrapping.

You should see 16 lines. Each parameter is stored on a new line. The properties of each parameter are separated with vertical "|" symbols). Going from left to right, the properties are:

TYPE (Integer / String / String chosen from a list [STRINGFIXED])

NAME (The actual name of the parameter)

DESCRIPTION (The description of this parameter. We don't really use this so much)

Then, depending on what kind of parameter it is, the following values are either:

VALUE (the value of the parameter)

OR

VALUE | PERMITTEDVALUE1 | PERMITTEDVALUE2 | PERMITTEDVALUE3 | ...list of permitted values

Integrating Parameters with the GUI

If you would like to integrate a parameter so that it is displayed in the GUI of the server (under the Parameters tab of the experiment),

1. Replace one of your variables (Integer, String, Long etc.) in your ConversationController object with its Parameter equivalent, e.g. StringParameter, IntParameter.

2. Initialize this Parameter object in your ConversationController.initialize() method and add it to the expSett file.

3. If your program changes the value of a parameter object during an experiment and you would like to see it updated in the GUI, you must remember to call the following method after changing the value of the parameter.

    - **expSettings.generateParameterEvent(PARAMETER)**

4. This method will inform the GUI, which will subsequently update to show the new value of the parameter.

.

To change a parameter value using the GUI

To prevent accidental changing of parameter values during an experiment (e.g. by pressing the wrong button), changing a parameter value takes 4 steps.

1. Select the row of the parameter you want to change

2. Double click the 3rd column (new value). A text entry field should appear

3. Type in the new value into the textfield. Press Enter. In the fourth column, the SET button should become enabled

4. Press the SET button

Adding your own stimuli

Currently the chat-tool allows you to control which stimuli are displayed on the clients. This is done with the following method:

Conversation. displayNEWWebpage(…..)

This command will open up a new window on the client. This method allows you to specify the size of the window. Also, this window is not closable or resizable by the client.

There are 3 main ways of using this command

Displaying stimuli as a web-page – initializing the window

Simply design your stimuli in HTML as a webpage. Upload the webpage to any server (or even run a webserver on the local server machine – see [ ] for more instructions on how to do this). Suppose you have 3 stimuli you would like to display, and they are stored at the following webaddresses.

http://www.myserver/experiment1/stimuli1.html

http://www.myserver/experiment1/stimuli2.html

To display the first stimulus, in your ConversationController object you would simply need to run the command

c.displayNEWWebpage(p, "ID1", "Stimuli1", http://www.myserver/experiment1/stimuli1.html", 500, 500, false);

"ID1" is a name you give to uniquely identify that window (this is intended for designs where there is more than one window) You can make up your own ID.

"Stimuli1": is the text that will appear on the window header on the client's machine. Again, you can use whatever text you want.

The next parameter is simply the URL of the webpage containing the stimuli.

The next two parameters 500, 500 are the dimensions of the window. You might need to play around with these to get it just right.

The next parameter specifies whether or not scroll bars should be displayed in the window.

Displaying the next stimulus in the same window

To display the next stimulus in the same window, use the following command:

c.changeWebpage(recipient, "ID1", URL);

Where "ID1" is the same unique identier used to initialize the window, and url is the new webpage address, e.g. http://www.myserver/experiment1/stimuli2.html

Displaying text only

If you only need to display text (and now graphics, or complicated layout), you don't need to set it up as a webpage. Do the following:

Initialize an empty window:

c.displayNEWWebpage(p, "ID1", "Stimuli1", "", 500, 500, false);

And then send an instruction to display a new webpage, adapting the following method:

c.changeWebpage(recipient, "ID1", "Text you would like to display", "", "");

Displaying multiple stimuli – avoiding building webpages for each stimulus

One other option allows you to avoid having to design webpages in html. This is especially useful if you would like to display many pictures (e.g.. JPEG) on the client machines, without having to create a webpage for each one.

The chat clients allow you to send HTML code from the server to the clients, instead of the URL. As long as it is preceded with an <html> header, the client knows to treat it as HTML and not as a web-address.

For example, suppose you have, hundreds of images in one folder, so for example

http://www.myexperiment/stimulusset1/img001.jpg

http://www.myexperiment/stimulusset1/img002.jpg

http://www.myexperiment/stimulusset1/img003.jpg

…….

http://www.myexperiment/stimulusset1/img900.jpg

It would be incredibly laborious to create a webpage for each one. Instead of having to do that, you can simply adapt the following code – which will enable you to automate the generation of the URLs of the stimuli, e.g.

String url = http://www.myexperiment/stimulusset1/"

String img = "img001.jpg"

String text = "<html><img src="'+imgsrc+'"></img>";

c.displayNEWWebpage(p, "ID1", "Stimuli1", text, 500, 500, false);

or

c.changeWebpage(sender, "ID1", text);

Using the server to host the images -

It can happen that each time you connect the server to the network, it gets assigned a new IP address. If your script uses a fixed IP address to tell the clients which stimuli to load on the server, this could mean having to change the script each time the server's IP address changes.

To get round this problem, the server API allows you to send a command to the clients that tells them which stimuli to load – without having to specify the IP address of the server. What happens is that the clients prepend the IP address of the server to the subfolder that you specify. For example, suppose the image you have is stored on the server, and the server's IP address is 142.243.125, and it's operating a webserver on port 81.

Your stimulus image is stored at:

142.243.125:81/stimuliset1/image1.jpg

The commands you would need to load this image would be

First, set up the window in which the image will be displayed, e.g. with

c.displayNEWWebpage(p, "ID1", "Some random name you give the client's window", "", 500, 600, false);

Then, use the following command to load the image.

c.changeWebpageImage_OnServer(Recipient, "ID1", **":81/stimuliset2/image1.jpg"**)

IMPORTANT: Note that this method does **not** require you to know the IP address of the server. This method tells the client to use whatever IP address it is currently using for the server. You do, however, have to specify the post number if you're not using the default port 80.

Saving stimuli information

Suppose you use many different stimuli in your experiments, and would like to keep track of them so that later, when you analyse them, the state/value of the stimuli is saved in the spreadsheet.

To save this information (or indeed any other information that you would like to save), simply use the following method in your ConversationController object:

c.saveData(String text)

This will save the text in the spreadsheet associated with the experiment. and will automatically timestamp it.

For a template that does this, look at CCPICTURESTIMULI

Integrating a new task

Currently the chat tool includes the maze game (2 and 3 person versions) and a tangram task. These are all programmed as subclasses of TaskController. These work independently of the ConversationController classes.

To integrate your own task, implement it as a subclass of a TaskController. If you want to save or load data separate from the actual conversational transcripts (i.e. task moves etc..), copy the functionality of the existing SetupIO and SetupIOMazeGameWriting classes.

If you want to save specific information from your task in the turns.txt file, you will need to:

* Create a subclass of ConversationHistory.Turn (see e.g. MAZEGAMETURN)

    o Ensure that it has an empty no argument constructor

    o Make sure it specifies both the header names to be given in the turn.txt file and also that it returns the values you wish to save.

* In your ConversationController object, override the method getTurnTypeForIO(..) to return a copy of your implementation of the turn.

* In ConversationHistory, implement a new Save(YourTaskName)Message(...) method that takes as parameters the additional information you want to save (see e.g. saveMazeGameMessage vs. saveMessage). This method only needs to change the line where the Turn is constructed. Change it so that it instantiates your subclass of ConversationHistory.Turn

Kinds of chat tool interface

Currently there are 3 types of chat window you can use.

(1) Standard chat window. This is the type of chat window which is used by existing commercial implementations. Participants have a text-entry window and a main window. Text entered by participants appears privately in the text-entry window. When the participants press "Enter/Return" or the "Send" button the text appears to them in the main window and is subsequently relayed to the other participants. [See link]

(2) Standard but split: This chat window is similar to (1) except that each participant has multiple main windows, where the text from each participant is displayed in separate windows. See the template "13 – Dual window" for an example of this.

(3) Character by character interface.

In (1) and (2) participants cannot see the other participants typing. This greatly facilitates creating interventions. (2) results in dialogue with reduced sequential coherence. (3) Is still being developed.

Programming interventions with (3) is much more complicated as it involves manipulating incoming and outgoing queues, introducing artificial delays into the dialogue. If at all possible, it is preferable to create interventions with (1).

Selecting the interface:

The interface is chosen via parameters that are stored in the Template associated with the experiment. To explain how it works, this little section explains some of these parameters and what they do:

Navigate to the chattool\Templates\ folder and open the file called "01 – No intervention..xml" with a text editor (ALTHOUGH IT SAYS XML, IT IS NOT AN XML FILE !. See the section on Parameters, on page 42 for more information about the format of this file. In brief, each parameter is stored on a newline – and the values / permitted values / saved values are separated with vertical "|" symbols). Search through this file for the following text which is stored on a single line:

```
STRINGFIXED|Chat tool interface|The different kinds of chat tool.  "Formulate
revise then send" is the default, tpyical of most messenger apps. It allows
single and multiple windows. WYSIWYG Simplex single window displays turns
character-by-character and strictly enforces turn-taking.  WYSIWYG Duplex
multiple windows displays turns character-by-character with each participant's
contributions placed in separate windows. It is similar to UNIX chat|Formulate
revise then send. Single or multiple windows|null|Formulate revise then send.
Single or multiple windows|WYSIWYG Simplex single window|WYSIWYG Duplex multiple
windows|
```

It's a little bit unwieldy, but the name of the standard chat window interface is:

```
Formulate revise then send. Single or multiple windows.
```

Then look at the line in the same file "01 – No intervention..xml" which says:

```
INT|Number of windows per chat tool||1|1|
```

This is the parameter that determines how many windows are displayed one each participant's screen.

Now, close this Template, and open up a new Template called "13 – Dual window.xml", also in the text editor. Also, try running this template locally as a demo. You should see how each participant has two windows. This is determined by the same parameter – notice how in this template, the parameter

```
INT|Number of windows per chat tool||2|1|
```

Has a value of 2, not 1.

Also, look in the other part of this template:

```
INT|Width of main window||400|400|
INT|Height of main window||250|250|
INT|Width of text entry window||220|220|
INT|Height of text entry window||150|150|
INT|Maximum length of textentry||1000|1000|
```

These are the (hopefully self-explanatory!) parameters for the interface.

Default Settings

As you may have noticed, not all of the settings are included in each Template. If some of the parameters are missing, the chat tool tries to load default values – this is all handled in the class called:

diet.message. MessageClientInterfaceSetupParameterFactory.java

This is the class responsible for constructing the Messages that are sent to the client which instruct the client how to configure the interface.

How to restricting participants' access to the conversation history (making it impossible for them to scroll back)

The default setting allows participants to scroll their chat windows to reread the text. If you want to change this, you can use the interface "JChatFrameMultipleWindowsWithSendButtonOneTurnAtATime.java". In the Template directory there is a file called "12 – Limited Turn Display.xml" which uses this interface.

If you open this template in a text editor (remember, no linewrap) you should see the name of this interface stored as:

```
TurnAtATime
```

Which quite literally means that this interface displays a turn at a time. Actually it's a bit of a misnomer, as the default setting is that it displays the 2 most recent turns.

If you want to play with the settings, for example to make it only display 1 turn, play around with the following class:

diet.client.JTextAreaOneTurnAtATime

Closing the chat tool windows on the clients

The default closing behaviour of the chat tool is disabled to prevent participants from inadvertantly closing the chat tool client window. The windows on the client can be closed directly from the server, by pressing the red "..." button on the tab of the experiment. If you need to close the client window on the client machine, on windows press "Ctrl-Alt-Del", select java.exe and press "terminate process".

Dealing with error messages / java exceptions

There chat tool has many different threads, it can be quite tricky keeping track of all the output / debugging messages generated by all these threads. In addition to using your favourite debugging tool, you can use the following methods to automatically capture and display messages.

Server – saving error messages

We've tried to make the server relatively resilient against the occasional bug in your program. The idea is that all Exceptions are captured and saved in an errorlog.txt file in the folder associated with the experiment (i.e. in a subfolder of Saved experimental data)

This should capture most errors, however if you want to capture some errors of your own program, you should use the following method to capture errors.

diet.server.Conversation.saveErrorLog(…)

Calling this method will print an error message in the Main server window

Server – displaying values in the window of the server.

The Server interface has a display in the bottom half that allows you to display messages. By default a single window is displayed called "Main". To send text to this window, use the following method:

Conversation.printWSln("Main",YOURTEXTYOUWOULDLIKETODISPLAY");

This can prove invaluable for tracking the variables of an intervention.

What you can also do is print information to different windows (The server will automatically generate them) by using a variant of the same method

Conversation.printWSln(WINDOWNAME",YOURTEXTYOUWOULDLIKETODISPLAY");

Debugging the client – displaying values in the window of the server

Because most evaluation/debugging tends to involve running the clients "as demo" on the local machine, if you want to debug what's going on on the client, simply use the Conversation.printWSln(….) method as outlined above. Because this method is static, this will only work while running the program on the same machine as "local demo".

Debugging the client – displaying values in the window of the server

The client has a method that sends error messages to the server (Where they are saved in errorlog.txt)

ConnectionToServer.sendErrorMessage(…)

Debugging

If you want to debug the chat tool, and display additional info / parameters while debugging, use the client.Debug.Debug. class to hold your static variables used to determine how it gets debugged.

There is a parameter there called allowCLIENTTO…

If you enable this, you can type commands in the chat window where they will be captured by the ConversationController object, and can be used to test your program without creating complicated test harnesses.

Dealing with crash recovery

There are two types of crash recovery for errors that happen on the server or clients. The first is for experiments where the server needs to be restarted and you would like to restart from where the experiment crashed. The second is dealing with, say, when an individual client disconnects from the server (e.g. a blip on the network)

Server

This functionality needs to be implemented for each experiment. The chat tool provides a way of recovering data from a previous experiment and continuing where it left off.

If, in the server window you go to "Saved experimental data" and right-click on the experiment name, you will see an option – "Attempt to resume experiment…."

This option adds a parameter to the experimentsettings which can be used by the ConversationController object to load in settings that recover the pre-crash state.

Of course, if you want to recover settings, you'd also need to save them – both of which can be done in the ConversationController object

This is used in the maze game experiments to reload the set of mazes in case there's a crash

It can also be really helpful using BeanShell to launch methods in the ConversationController object once the server has restarted.

Client

The chat tool is programmed so that if there's a problem with the network or the client crashes etc…the client can log back in and rejoin the experiment. Obviously, how you implement this depends on the experiment, as it might be the case that you would wish to recover the experiment from a particular game state.

At a lower level, ParticipantCOnnection should handle the intricacies of logging in, followed by

Diet.server.GroupAssignment.reactivateParticipant()

This is the main location in the server that handles these requests to reconnect

Once the reconnect is successful, the following method is called in the ConversationController object:

participantRejoinedConversation(…)

Customize this method to, say, send a custom message to the participant who has rejoined, or to send a message to the other participants that the participant is back in the conversation.

Displaying stimuli on the client's computer

To display stimuli to the clients in a controlled fashion, there are some methods available that can be called from the ConversationController object . For example, once the participants have finished a task you want to present them with new / different task instructions, .

The chat tool currently allows you to:

1. Display a webpage on the client's machine (in a window whose size you can control)
2. Display "normal" unformatted text on the client's machine
3. Display a "progress bar" on the client's machine

Creating more plausible interventions

Although participants are surprisingly forgiving with all kinds of incoherencies/inconsistencies in the dialogue, it is obviously ideal for the interventions to be as "natural" as possible. It helps if the interventions are generated after convincing delays using Thread.sleep() that mimic turn formulation time and are proportional to the length of the interventions. This can be further enhanced by spoofing the "is typing" messages.

Another trick is to use the dynamically produced information produced by the participants – for example using the

Introduce typos into the interventions using TypoGenerator class

If your intervention is usingfake "error messages", make sure that they also occur randomly in the dialogue so that they don't get primed.

Be careful using typing speed as the basis for any interventions, as it can vary quite drastically – e.g. what's the typing speed of a turn that originally contained, say, 20 characters, but then the participant deleted all of them before sending? It's probably best to implement a maximum and minimum value.

Libraries used by the chat tool

The chat tool currently uses:

- SwingUtil1.5, implementation of CloseableTabbedPane.
- Infonode, custom implementation of TabbedPanes.
- Stanford Parser
- JWordNet a Java implementation of WordNet.
- BeanShell scripting, and dynamic interpreting of Java commands]

Programming the Character By Character interface

So far the first part of this user manual has been on programming the turn by turn interface. There is another interface included which allows incremental, character by character relaying of turns between participants. The downside is that it is many orders of magnitude more complicated to program.

Have a look in the Templates directory at the templates with the word CBYC in them – these are templates that contain the

In addition to relaying keypresses, the server needs to use the participants' typing behavior to determine which participant is allowed to hold the conversational floor – this is a quite horrific multithreading problem – the implementation that doesn't allow participants to delete text works, but the implementation with deletes is a little buggy.

The table below shows how the server controls which participant may or may not take the conversational floor.

Client A	Server	Other Clients
Client presses a key, key request gets passed to CBYCDocument. If participant doesn't have the floor, the key pressed is stored locally (As a DocInsert) and a request to gain the floor is sent to the server.		(any requests by this client are also sent to the server, as with Client A)
Any keys pressed by the participant are stored locally	Server receives request for the floor which it sends to the FloorHolder FloorHolder establishes whether it's ok to change speaker. If FloorHolder determines that it's ok to give the participant the floor, it ▪ Sends a message to the Client (In this case Client A)	Any keys pressed by the participant are stored locally, as with Client A)

	that requested it. ▪ If there is a speaker-change then the server automatically prepends the name of the Participant who requested the floor. ▪ Sends message to all the other clients that they can no longer request the floor, and that Client A has gained the floor (SOMEONEELSETYPING) All subsequent requests by any other participants are ignored. It also	
Participant gains control of the floor. All the keypresses stored are displayed on the screen.		Participant receives a message saying that the other participant, ClientA is typing and has the floor. All the stored keypresses are deleted.
All the participants keypresses are immediately displayed on the participant's local screen. The information about what the participant types is sent as a DocChange object to the server. DocChange objects record the String that was typed and the position in the text where the String was typed. Text that is deleted is also sent as a DocChange object that records the position and length of the deleted text. **IMPORTANT:** The position info is transformed from Javas default of indexing the position from the beginning of the text, to an index that counts the position from the turn-terminal position. Therefore typing the letters abc would be recorded as three DocInserts for each of the letters a,b,c and all the	All the messages typed by Client1 are relayed to Sequences -> Sequence, which subsequently relays the DocInserts to the other clients On a separate thread, FloorHolder is constantly waiting for a lull in the typing	Client is blocked from typing anything all the keypresses by ClientA are displayed on this client's screen

DocInserts would have an index of 0.		
Any keypresses typed here will continue to be relayed to the other clients by the server	Floor Holder – after a predetermined timeout (Set in the ConversationController object), the FloorHolder object sends a message to the ClientA, instructing ClientA to relinquish the floor. Note: The timeout can be set to millisecond values or to non	
Client receives instruction from Server saying that it should relinquish floor. Client sends confirmation that the floor has been relinquished		Note: other clients are still blocked from typing
Any keypresses by client are treated as in step 1 of this table and lead to a request being sent for the Floorr		Note: other clients are still blocked from typing
	Server receives request from ClientA and continues to relay all the keypresses from ClientA to the other participants, as above……	Note: other clients are still blocked from typing
	(This is abbreviated description of exactly the same step above) FloorHolder detects lull in the conversation, requests Sequences->Sequence whether the ClientA is permitted to relinquish the Floor If yes, message is sent to A to relinquish the floor	Note: other clients are still blocked from typing
Client receives instruction from Server saying that it should relinquish floor. Client sends confirmation that the floor has been relinquished Client is free to request floor.		Note: other clients are still blocked from typing
Client is free to request floor.	Server receives confirmation that client has relinquished the floor	Note: other clients are still blocked from typing

	Client sends message to all other participants instructing them that they are free to request the floor	
		Client(s) receives message saying client is free to request floor
		Client requests floor (beginning of turn by new speaker)

Some issues

1. When running CBYC experiments, a current bug is that sometimes one of the clients doesn't permit typing at the start. To get round this problem, find one of the clients that does permit typing – after one of them gains the floor,

Following messages – the nuts and bolts – On the client

The main class on the client that deals with user input is JChatFrame. This is an abstract super-class used to build the user interfaces of the chat tool clients. For example, JChatFrameMultipleWindowsWithSendButtonWidthByHeight.

The participant types text, e.g. a single character into their private text entry window.

This is detected by a class in JChatFrameMultipleWindowsWithSendButtonWidthByHeight extends JChatFrame called InputDocumentListener. This listens for any changes made to the text, and is called automatically by the SWING thread, whenever the contents of the text entry window change.

When the InputDocumentListener notices a change to the text-entry window, it calls ClientEventHandler.

	Dealing with User Input on the client	Dealing with messages coming from the server and other participants
Subclasses of JChatFrame e.g.JChatFrameMultipleWindows WithSendButtonWidthByHeight	This class captures all the user's input as it is written to the interface. All this information is automatically captured (E.g. keypresses, timestamps, contents of screen)	
ClientEventHandler	The information from the interface is collated, and sent to ConnectionToServer.	
ConnectionToServer	The information from the clients is packaged into objects of type diet.Message. Each kind of information from the client has its own associated Message. (This might be a little bit overkill/overbloated, but it can really help with debugging!)	