CT^3 Framework

I. Usage

A. Building and Running CT^3

To acquire the most recent code from cvs, changing `user` and `drdoom.eecs.harvard.edu` as appropriate:

```
$ cvs co -d ':ext:user@drdoom.eecs.harvard.edu:/home/lair/coloredtrails/cvs' ct3
Password: <type eecs password>
```

To build it, run the included `antbuild.sh` script. This is primarily a wrapper to increase the classpath of `ant` while building the code to include some libraries from the `jar/` directory.

```
$ cd ct3/
$ ./antbuild.sh
```

Building the code generates a jar file, `ct3.jar`, which contains both coloredtrails and its required Java libraries.

CT^3 consists of three distinct applications, each run through the same interface. These consist of:

Server Runs games and communicates with clients.

Admin Client Used by an experimenter to communicate with the server allowing games to be run, configurations to be uploaded, and server data to be accessed.

GUI Client Used by players to connect to the server and be matched to games.

All three applications are run using the `ct3.jar` jar file, each using different command line arguments. The server is run as:

```
$ java -jar ct3.jar -s [--server_localport <int>]
```

... where `server_localport` describes the port which the server will listen on (if this is not specified, port 8080 is the default). The GUI client is run as:

```
$ java -jar ct3.jar -c --pin <int> [--client_hostip <int>] [--client_localport <int>]
```

... where `pin` is the personal identification number which will be associated with the given player, `client_hostip` is the server IP to be connected to (if not set, it can be specified using the UI), and `client_localport` is the port on which the client will listen for server messages (if not specified, it will default to 8888). The default client and server ports are defined in `coloredtrails.shared.Utility`. The admin client is run as:

```
$ java -jar ct3.jar -a
```

... and takes no additional command line options at present.

CT³ requires Java 1.5 as it uses new language features like generics and boxing/unboxing. Most versions of Java 1.5 are available from Sun at `http://java.sun.com/`. Apple makes the only version of Java 1.5 for Mac OS X (named the MRJ rather than JDK or JRE), and it is available at `http://developer.apple.com/java/`. CT³ has been tested with Java 1.5 Update 1 on Mac OS X and Linux. The Java binary on Mac OS X is located in an odd place:

```
$ /System/Library/Frameworks/JavaVM.framework/Versions/1.5/Commands/java
```

But on most other platforms, Java 1.5 will be located in a more obvious place (usually `/opt/jdk*/bin/java` or `/usr/local/jdk*/bin/java` on *nix).

B. Server Usage

The server is the first application that the experimenter needs to start, but it should not require any intervention after being started. It may occasionally print some debugging output.

C. GUI Client Usage

The GUI client consists of three different windows:

Taskbar The initial and main window for the player to perform actions.

Waiting for Players Window After connecting, this window pops up with a (non-)progress bar while waiting for a game to start.

Game Display This window displays information about a current game, including the board state, current and upcoming phases, and current chip allocations among players.

The taskbar consists of a series of buttons: a connect button used to choose a server host IP to connect to (assuming that this has not been specified on the command line), zero or more discourse buttons to permit a variety of discourse communications among players, a transfer button to permit the unilateral transfer of chips by a player, and an exit button to exit the GUI client. The game display primarily updates itself from the state of the server, though it also permits movement by dragging the player's icon to a new square.

D. Admin Client Usage

On startup, the admin client will display a brief introductory message and then will present the user with a prompt:

```
$ java -jar ct3.jar -a
Colored Trails Admin Interface v1.0
Type 'help' for more information.
user:/home/user/ct3/%
```

The admin client allows you to set state with a `csh`-like `setenv` command. In order to do anything useful with the client, the experimenter must set `HOST` to the server that was started previously:

```
user:/home/user/ct3/% setenv HOST http://host:port/
```

In order to upload configurations, the client must also have its `CONFIGDIR` set, which defines the directory where runnable configuration classes (discussed later) reside:

```
user:/home/user/ct3/% setenv CONFIGDIR /home/user/configdir/
```

Once `HOST` is set and the server is running, a variety of commands become available, including commands to list available or running players, configurations, and games:

```
user:/home/user/ct3/% list players
<list of all registered players follows>
user:/home/user/ct3/% list configurations
<list of all uploaded configurations follows>
user:/home/user/ct3/% list games
<list of all running or completed games follows>
```

Aside from these purely informative commands, there are also two action commands. The first is `add configuration`, which adds a configuration file which consists of compiled Java code (this file must exist in `CONFIGDIR`):

```
user:/home/user/ct3/% add configuration <config class>
<success or failure follows>
```

The second command is `new game`, which starts a new thread defined by a selected configuration using several specified players:

```
user:/home/user/ct3/% new game config <config class> players <pin_1 ... pin_n>
<assigned id of the game or -1 for failure follows>
```

A sample usage of `add configuration` and `new game` using the configuration class `KaleidoscopeConfigDetails` and players with PINs 333 and 400 follows:

```
user:/home/user/ct3/% add configuration KaleidoscopeConfigDetails
true
user:/home/user/ct3/% new game config KaleidoscopeConfigDetails players 333 400
3
```

The client also supports a variety of commands which are unrelated to CT^3 (though most of these are likely of little use), as well as output redirection and pipes if desired (e.g., the output of a CT^3 command could be piped through a simplified version of `grep` or output to a file using the client).

II. General Architecture

For definitions of software engineering terms used in this section, see the Glossary.

A. Front End

As noted previously, all three primary applications which make up CT³ are accessed through a command line front end. These options are parsed in `coloredtrails.shared.app.FrontEnd` using the Java version of GNU Getopt. There is a great deal of documentation for how to use GNU Getopt, though the code itself should be fairly clear—Getopt can take long (–foo) and short options (-f) with or without arguments (or with optional arguments). The front end can also spawn more than one of the primary applications, and the command line flags of each are designed not to conflict in order to allow for this.

B. Shared Types and Data

There is a great deal of code which is shared between the server and its clients. The first major place where this happens is in communicating game and server state between the server and clients. Most game data is stored as a type in the `coloredtrails.shared.types` package, descending from `coloredtrails.shared.types.XmlRpcHashSerializable`. Descending from this class means that the type can be turned into a `Hashtable` and then sent over the wire as an argument to an XML-RPC method. The other big advantage of this design is that client and server can have different versions of the same type, with different methods and data, but with the same core data which is sent over the line and parsed by the core type. For example, `coloredtrails.types.PlayerStatus` is extended in the GUI client to also hold a `Sprite` object for display on the board, but it is still compatible with the `XmlRpcHashSerializable` implementation of itself on the server:

```
abstract public Hashtable toHash();
abstract public String toString();
```

```
abstract public void fromHash(Hashtable h);
```

While most of the types are what you would expect in terms of sharing game state (e.g., `Board`, `ChipSet`, `Path`, `RowCol`), there are two major exceptions. The first is the `HistoryLog` which is made up of a series of `HistoryEntry`s. The log is meant to function as a fairly complete log of the game for experimenters. The second is `GameConfigDetailsRunnable`, which is the configuration for a particular game which is run directly as code on the server. Rather than discussing that class here, there is a section dedicated to it later in this document (Configuration Class HOWTO).

There are several other shared classes of note. `coloredtrails.shared.exceptions` consists of classes implementing `ExceptionWriter`. `ExceptionWriter`s can be passed to the more general `coloredtrails.shared.Utility.makeXmlRpcRequest()` method, such that when an XML-RPC request fails, the particular appropriate output for the application (a GUI dialog, a printed message, output to an outstream) is performed. `ExceptionWriter`s for the GUI, admin client, and server exist. `DiscourseMessage`s are used to permit various types of discourse in a similar way to shared types (these are discussed further in the GUI client section). Finally, colors are handled by the `GlobalColorMap`, a class which defines `String` to `Color` mappings as well as providing a list of available colors. All client-server communication is conducted using the `String` representation of colors.

C. Admin Client

The admin client is based off of JShell: a former class project which provides an extensible shell-like interpreter. The core of JShell is a read-eval-print loop (REPL). A prompt is printed, the user inputs a line, and then the interpreter reads one line of input. The shell has a list of `Command` objects which can each signal themselves as the correct command for the user's input. Once the input is accepted by a command, the command is instantiated,

can reject incorrect input, and then is given the input to run itself (this is roughly the [**factory??**] object-oriented design pattern). The command writes it output, and the loop begins again.

All `Command` classes extend `Command` and need to implement the following methods:

```
public void doCommand() {
  /* output the command's result to the out stream */
}

public Command getCommand(String[] argv, StateHolder stater,
                          PrintWriter out, BufferedReader in) {
  /* return an instance of the command with the given state */
  /* argv is the argv of the command as per unix argv       */
  /* stater is a stateholder which describes jshell state   */
  /* in and out describe the input and output streams...    */
  /*                    ...into the command, for piping.    */
}

public void horribleDeath() throws BadShellInputException {
  /* throw an exception if the improper input has been given */
}

public boolean isProperCommand(String command) {
  /* return whether command as described in the string... */
  /* ...should be handled by this command object.         */
}
```

Any `Command` can be added to the admin client by adding an instance of it to the interpreter. This is done currently by calling the `addCommand` method in `AdminClientJShell`, which extends the normal `JShell`. Rather than ex-

tending `Command` directly, most `Command`s in the admin client instead extend `AdminClientCommand` which also grants access to a number of useful utility methods (including "**public** Object[] makeXmlRpcRequest(String method, Vector params)" which is the method for making XML-RPC requests to the server from the admin client).

Environmental variables are added and changed within the JShell (e.g., when `setenv` and `unsetenv` are run) by making a call to a `StateHolder` object which all `Command`s are given at instantiation time. Though `StateHolder` has a variety of methods in its interface, the primary methods of interest to `Command` writers are "**public void** setVariable(String varname, String value)" and "**public** String getVariable(String varname)," both of which should be fairly self-explanatory. There may be some bugs in JShell—it does parsing in some odd ways, and is primarily suited to easier parsing tasks (e.g., it does not use a parser generated by some sort of reasonable grammar because most of the commands it accepts are very simple). Excessive use of variable interpolation should be avoided if possible (or should at the very least be assumed not to automatically work).

D. Server

The server is started by starting two threads: the `ServerQueryResponderThread` and the `IpAddressGathererThread`. The latter is optionally used in order to detect the client's IP address (the server needs to know the client's IP address in order to initiate XML-RPC method calls on the client). The client can also try to auto-detect or manually set its IP and then send this directly to the server using the XML-RPC method `client.informServerOfIpAddress`. `ServerQueryResponderThread` starts an XML-RPC web server which handles two sets of commands: those in `AdminCommands` (under the name `admin.*`, e.g., `admin.newGame`) and those in `ClientCommands` (under the name `client.*`).

XML-RPC automatically translates commands sent over the wire into calls to the proper methods in `AdminCommands` and `ClientCommands`.

Most commands handled by the `AdminCommands` or `ClientCommands` classes translate into a modification or reading of data from `ServerData` which holds all of the server information including all game states of running games. `ServerData` is a [**singleton??**] accessible throughout the server. The most complex XML-RPC command is the `admin.newGame` command. This command does the following:

1. Create game state for a new game, with a new game ID.
2. Set `GameStatusSender` to observe the new game state (as per the [**observer??**] design pattern), so that when the state changes in the future, all players are properly notified. `GameStatusSender` places `GameStatusUpdateMessage`s on queues corresponding to each client.
3. Create new threads for each player in charge of sending the assigned player pending updates as scheduled by the `GameStatusSender`. Each thread is a `SinglePlayerMessageSenderThread`.
4. Create a new thread running the `GameConfigDetailsRunnable` which was specified by the admin client. This runnable is in charge of changing the game state and handling game messages as appropriate for the style of the game required in the experiment (see Configuration Class HOWTO below).

After a new game has been started in this way, it is only modified as provided for by the `GameConfigDetailsRunnable`.

E. GUI Client

Most of the architecture of the GUI client is fairly uninteresting. The client consists of two main threads: one Swing thread, with a hierarchy of GUI actions descending from the

`Taskbar`, and one networking thread, which receives updates from the server. The client's XML-RPC handler is named `ServerCommands` (the convention in CT[3] is to name an XML-RPC handler the remote client accessing the handler and then "Commands"). Messages sent from the client to the server are all currently routed through the `ClientCommunication` class.

By default, the client permits the user to connect to the server, transfer chips, and exit. However, this is the bare minimum: one can add to the set of discourse messages which can be sent and handled by the server and client by creating classes descending from `coloredtrails.client.ui.discourse.DiscourseHandler` (for the UI handling of the discourse message) and `coloredtrails.shared.discourse.DiscourseMessage` (for the message sent between clients and relayed by the server). Any number of `DiscourseHandler`s can be added to the GUI client. When a discourse message is received from the server, the client will cycle through all of the `DiscourseHandler`s that it knows about and call the `onReceipt()` method of whichever class' `getType()` matches the `getMsgType()` output of the `DiscourseMessage`. Similar methods also exist for when a `DiscourseMessage` is sent (`onSend()`), or when it is clicked in the `Taskbar` (if the `DiscourseHandler` is one which should be available for initiation by the user).

III. Configuration Class HOWTO

Game configuration is done by creating a compiled Java class of type `GameConfigDetailsRunnable` and then uploading it to the server using the admin client's `add configuration` command. The only method that must be overridden is `run()`:

```
abstract public void run();
```

This code can get the `gameid` by calling "**public int** getGameId()". It is then possible to leverage this into a `ServerGameStatus` object by creating an instance of `ServerData` and getting the status by `gameid`:

```
ServerGameStatus gs = ServerData.getInstance().getGameStatusById(getGameId());
```

The game `Board`, `PlayerStatus`es, and other assorted game state can be acquired similarly. All changes to the state of these game objects will be observed by the `GameStatusSender` and then sent to clients. Thus, the writer of a configuration runnable needs only concentrate on the game logic rather than communication or client behavior.

While overriding `run()` is the only requirement for a configuration runnable, there are also a number of handler methods for different actions that occur throughout the lifetime of a game:

```
public boolean doMove(int perGameId, RowCol newpos) {
  /* perform a move: the default is to perform the move and */
  /* remove the requisite chip                              */
}

public boolean doTransfer(int perGameId, int toPerGameId,
                          ChipSet chips) {
  /* perform a transfer: the default is to perform the      */
  /* requested transfer, assuming that the player has the   */
  /* requisite chips                                        */
}

public boolean doDiscourse(DiscourseMessage dm) {
  /* relay a discourse action between players: the default  */
  /* is to relay the message without understanding it       */
}

public int getPlayerScore(int perGameId) {
  /* determine the current score for a given player given   */
  /* the game state                                         */
```

}

These handler ([**callback??**]) methods should provide for a wide variety of behavior: various role and commitment protocols should be easy to enact using this system of methods, intercepting unacceptable communication or enforcing contracts previously communicated. The configuration runnable can keep its own state, and thus can keep track of data like commitments.

In order to ease the creation of phases, a `Phase` type exists which makes it easy to create a game configuration with phases occurring at set times and with actions occurring at the beginning or end of phases (or restricting other actions based on the current phase). A standard phase setup might look like:

```
ServerPhases ph = new ServerPhases(this);
ph.setStartPhaseName("Monkey Phase");
ph.addPhase("Monkey Phase", "Badger Phase", 10);
ph.addPhase("Badger Phase", "Train Phase", 10);
ph.addPhase("Train Phase", "", 10);
ph.setFirstPhaseTime(10);
ph.setLoop(false);
gamestatus.setPhases(ph);
```

Each phase is defined by its phase name, the following phase name, and the time in seconds that it lasts. For now, the duration of the first phase is also required (due to an implementation side-effect which may eventually be removed). The total phase cycle may be looping or non-looping (if non-looping, the configuration runnable needs to be in charge of its own end of game change). `ServerPhases` takes a `PhaseChangeHandler` as an argument: this phase change handler can simply be the configuration runnable, or can be another object that implements:

```
public interface PhaseChangeHandler {
    public void beginPhase(String phaseName);
    public void endPhase(String phaseName);
```

}

These are called at the beginning and end of a phase, respectively, with the name of the phase being ended. Phase timing is currently handled by decrementing an int which holds the seconds count—this is likely only accurate to within a few seconds, and currently no attempt at sychronization is made except for at the beginnings and ends of phases. Configurations that may cause clients to try to perform last minute requests will likely require some slight modification of the core code to permit sub-second synchronization.

For an example of a working configuration file, one should look in the `misc/gameconfigs/` directory of the source distribution. `KaleidoscopeConfigDetails` implements much of the functionality that a game can have, randomly setting goals, square colors, and player locations.

`GameConfigDetailsRunnable` classes should not be in a package, and should be compiled at the base of the source distribution. A working `javac` command to compile `KaleidoscopeConfigDetails` at the base of the source distribution is:

```
javac -cp .:jars/xmlrpc-2.0-beta.jar KaleidoscopeConfigDetails.java
```

... though you will want to change `KaleidoscopeConfigDetails.java` to your own configuration details runnable.

IV. Client-Server Communication

CT³ uses Apache XML-RPC (`http://ws.apache.org/xmlrpc/`) for client-server communication. All of this communication is funnelled through `makeXmlRpcRequest()` methods however, so the choice of XML-RPC implementation as well as transport protocol in general would be relatively easy to change if desired. The protocol is platform independent in the sense that any client written in a language that supports XML-RPC (most languages have an implementation because the protocol is relatively simple) can send and receive every mes-

sage that is communicated, with the exception of the Java bytecode which defines a game configuration (sent by the experimenter to the server).

The server responds to the following administrator commands over XML-RPC:

admin.listGames Return a vector of `GameStatus` hashes describing games in progress.

admin.listConfigurations Return a vector of `GameConfigDetailsRunnable` hashes which have been uploaded to the server, and their names.

admin.listPlayers Return a list of players who have connected to the server (returns `PlayerConnection` hash vector).

admin.addConfiguration Add a particular configuration for use when starting new games. Takes the bytecode of the configuration and its name. Returns success or failure.

admin.newGame Start a new game (as discussed above in the server architecture), returning success or failure. May fail if configuration is invalid, players are invalid, or for other reasons.

It also responds to the following client commands:

client.register Register that this client would like to be assigned to a game, and give a pin assigned to the client and a port which the client would like to be communicated to on.

client.informServerOfIpAddress If the client can figure out its own IP address, then register the IP address with the server (do this or use the IP address gathering thread before calling client.register).

client.messageIdStart Get an integer to start with as a base for messageid ints when sending messages. Each message sent by the client has attached messageid for communication purposes (e.g., rejection, acceptance).

client.move Request that the server move the player's token to a new specified location.

client.transfer Request that the server transfer a set of the player's chips to another player.

client.discourse Request that a message be relayed through the server to another player (and possibly be interpreted by the server en route).

Lastly, the client understands the following commands received from the server:

server.gameStarted The game started (includes initial game data).

server.gameEnded The game ended.

server.boardUpdated Something caused the board to change to a new state (includes new state).

server.phasesUpdated The set of current phases changed (includes new state).

server.phaseAdvanced The state of the phases changed (includes new state).

server.playersUpdated The state of the players changed (includes new state).

server.logUpdated The state of the server history log changed (includes new state).

server.discourse Another player has sent the given **[discoursemessage??]**.

These commands are not currently designed to be optimal—if necessary, one could likely substantially reduce bandwidth consumption by only sending changed data during updates (rather than entire data structures in which one part has been changed). However, for the current purpose this is not a problem.

Communication Note: Harvard's wireless FAS network blocks incoming non-initiated communication on upper (and possibly lower) level ports, so one should only run experiments on the wired network.

V. Descriptions of Prerequisite Libraries

There are several libraries which add functionality to CT^3. These are:

OOGA and TransparencyMap These libraries provide for the sprites shown on the `BoardPanel` and allow them to be clicked and dragged.

JShell Discussed above in the admin client section, handles command line parsing and creates a pseudo-Java shell.

GNU Getopt Discussed above in the Front End section, handles command line options.

Licenses of these libraries are discussed in the file `LICENSES` included in the source distribution.

VI. Short Term Future Work

The following things should be completed in the short term to make CT³ cover most of the needs of researchers using it.

A. Additional Logging

More information should be put in the log. This is a relatively trivial task, but currently only a few actions trigger adding information into the history log, whereas experimenters will likely want much more data to analyze after an experiment. Definitely add game start, score change, and game end history entries at the very least.

B. Commitments

There should be higher level abstractions (perhaps a class or two) to ease the creation of game configurations which involve commitments. This will also likely involve creating a few `DiscourseMessage`s and `DiscourseHandler`s which have special meaning in these games (e.g., a `ForcedProposalDiscourseMessage`). This machinery should be able to track commitments that have been made, and then enforce them at predefined points (probably at the beginning and ends of certain phases).

C. Roles

There should be higher level abstractions for creating configurations where certain players have certain roles. Currently, the ordering of players added to a new game determines their per game id, and this can be used to define particular roles within the game (i.e., stop player 1 from sending chips), but this is currently difficult to do and should be better abstracted.

D. Per-Player Goals

Currently a goal is defined as a goal for all players. Experimenters would like to be able to create per-player goals. This is a relatively easy modification to the core (modify `Board` most likely) but a relatively difficult HCI challenge.

E. Shortest Path GUI

Previously, the CT2 client enabled a human player to see all shortest paths to the goal visually. This functionality should be fairly easy to transfer from the old code to the new code (though per-player goals may make it more difficult).

F. Chip Visibility

The game configuration should be able to modify which players can see one another's chips, and possibly whether they can see other things about other players (previous actions and so forth). This should likely be thought of in a total context including Board Visibility as well—each player should have a view of the game state, and updates should only be sent when this view changes.

G. Miscellaneous UI Cleanup

Unavailable taskbar buttons should be disabled (e.g., do not allow transfer while the client is connecting or disconnected). Failed requests to move, transfer, or perform another action should be displayed more cleary in the client, perhaps with better error handling or error messaging. The game end should be made more clear with some GUI actions. ChipSetInputPanel's should not allow invalid chip values to be input (greater than the player's allocated chips for a transfer, for example).

H. Miscellaneous Code Cleanup

Remove the bug where `currentSecsLeft` in the `Phases` class must be manually set. The code will make this more clear.

`ClientData` defines its `DiscourseHandler` objects in the constructor, which can lead to infinite loops if the `DiscourseHandler` requests data from `ClientData` improperly when constructed. This should be changed so that `DiscourseHandler` authors do not need to be as careful when using `ClientData` instances.

Switch `AllPlayerChipsDisplay` to a `ListTable` to eliminate annoying occasional Swing exception about array out of bounds.

MessageID is used in two different contexts in `DiscourseMessage`s and regular XML-RPC methods. Fix this.

I. Race Conditions

Check that there are no race conditions between threads accessing `ServerData` and other shared state.

J. Improve Robustness of Client-Server Communication

Allow the server to detect which clients if any are not responding to updates. Give an interface to the experimenter to handle these cases.

K. Add Client's Server Port Selection

Create a command line option for selecting the server port to connect to. Currently only the local port and the server's IP can be set on the client, and while the server can select any port to listen on, currently the client can only connect on port 8080.

L. Direct Server Messaging

Provide a mechanism for a configuration details runnable to send mesages directly to clients to be displayed to the user (rule or phase explanations for example).

M. Better Specification of Allowable Acts

Create data in `GameStatus` to better specify to clients (and thus improve the HCI and reasoning abilities of these clients) what actions are allowed at a given moment. This should also restrict which discourse actions are allowed dynamically (rather than recompiling the client for each experiment or silently rejecting invalid discourse messages).

N. Test Multiple Games

Test both whether multiple games occuring on the server at the same time lead to any concurrency bugs, and whether clients can handle multiple games in sequence (in other words, when a game ends, the player can be reassigned by the admin client to a new game). Both of these features should work in theory, but have not been tested.

O. Add Admin Client Command to Display Log

Add an admin client command which prints the history log of a particular game, by gameid.

VII. Long Term Future Work / On the Horizon

These areas are long term features which would be ideal if they are handled, but are not pressing.

A. Message/Action History

Provide a history of past actions by the player or in general in the UI. This could be based on the history log.

B. Teams

Currently the configuration language allows one to define scores based on arbitrary groups of players, one must simply do this manually. One could imagine wanting to do more advanced tasks with coalition formation or wanting to ease the task of calculating team scores of a particular kind, but this task is as yet undefined (see also teams and board visibility below). The question here is in essence whether teams should be a first class notion in the game and in turn whether it should have additional abstractions used in the configuration file and game status as well as modifications to the HCI to display the information.

C. Board Visibility

Eventually, configurations should be able to modify the visibility of given squares on the board for particular players. It is not immediately clear how to handle this in the context of the HCI or of teams.

D. Protective Code

Possibly modify some of the client side versions of game types to throw exceptions if/when the agent code tries to modify something that should be set in another way, for example, trying to directly change board state rather than requesting a move from the server.

VIII. Coding Conventions

There are certain coding conventions which should be followed while coding on CT³. These are:

1. All position coordinates are in terms of (row,col) and in particular `RowCol` objects. The one exception is the interface to OOGA, which is in terms of (x,y) to match that library.
2. I don't have a particular preference, but "message" or "msg" should be standardized on, and "recipient/sender" or "from/to" should be as well (these became competing standards before I had realized it).
3. Handlers for XML-RPC are named *Requestor*`Commands`, where *Requestor* is the requesting client, i.e., Server, Client, or Admin.

Appendices

A. HOWTO Add Common Items to the CT³ Framework

A. Add a New Discourse Action

All communication between clients occurs through discourse messages. Discourse messages are some data sent from one client addressed to another client and relayed through the server. There are two major parts to creating a new communication primitive which can be sent between players: creating a `DiscourseMessage`, the data which is sent between the

players, and creating a `DiscourseHandler`, the element which permits the creation, display, and future actions upon the communication being sent.

A.1 Creating a DiscourseMessage

First, create a new class which extends `DiscourseMessage`. This involves implementing the following methods:

```
public String getMsgType() {
  /* return a string identifying this message type            */
}

public HistoryEntry toHistoryEntry(String phaseName, int phaseNum,
                                   int secondsIntoPhase) {
  /* create a history entry from this message suitable for... */
  /* ...adding to the history log on the server               */
}

public DiscourseMessage newInstance(DiscourseMessage dm) {
  /* create a new instance of this message from a given ...   */
  /* ... input discourse message                              */
}
```

`getMsgType()` will later be matched up with the type of a `DiscourseHandler` for display. The format of the `HistoryEntry` is not particularly important (see the `HistoryEntry` class for how to create an appropriate entry). Implementing `newInstance()` will likely only require returning a new object of the type with the data from the `DiscourseMessage`—all of the data will likely be the same, but the new type will override certain methods. All data carried by the `DiscourseMessage` should be added to the `contents` hash in a format which is acceptable for transmission over XML-RPC. It may ease usage of your

`DiscourseMessage` to make convenience functions to access the data types added to the hash for your message: for instance, `BasicProposalDiscourseMessage` has methods to get and set sender and recipient `ChipSets`, which are really just facades to the `addDataValue()` and `getDataValue()` methods in the `DiscourseMessage` base class which in turn modify or retrieve data from the `contents` hash.

A.2 Creating a DiscourseHandler

Creating a `DiscourseHandler` requires implementing the `DiscourseHandler` interface:

```
public interface DiscourseHandler {
    public void onReceipt(DiscourseMessage dm);
    public void onSend(DiscourseMessage dm);
    public String getType();
    public String getTaskbarName();
    public boolean inTaskbar();
    public ActionListener getDiscourseActionListener();
}
```

`getType()` is an arbitrary `String` which should be the same name as the `DiscourseMessage`'s `getMsgType()` which this [**handler??**]purports to handle. `inTaskbar()` determines if the `DiscourseHandler` is shown as a button in the taskbar, and `getTaskbarName()` determines the name of the button if it is. The associated button on the `Taskbar`, if any, will be associated with the `ActionListener` returned by `getDiscourseActionListener()` allowing an action to be performed when the user initiated the discourse action by clicking the button. `onSend()` and `onReceipt()` are called when a `DiscourseMessage` associated with this handler is sent or received by the client, respectively.

A.3 Completing the Process

After writing the appropriate `DiscourseMessage`s and associated `DiscourseHandler`s for the clients, the client and server need to be informed about them. `DiscourseMessage`s are added to the server by adding an instance of the new `DiscourseMessage` to the array generated at `coloredtrails.server.ServerData.getDiscourseMessageTypes()`. `DiscourseHandler`s are added to the client by adding an instance of the handler to the `discourseHandlers` list in `coloredtrails.client.ClientData`. The client will now auto-generate the proper buttons on startup and will search the new handlers on receipt and sending of messages. Likewise, the server will forward the new messages and add history notices to the log as appropriate.

B. Add a New Admin Client Command

This is discussed above in the Admin Client architecture section, briefly, however, one will want to extend `AdminClientCommand`, implement the following methods:

```
public void doCommand() {
  /* output the command's result to the out stream */
}

public Command getCommand(String[] argv, StateHolder stater,
                          PrintWriter out, BufferedReader in) {
  /* return an instance of the command with the given state */
  /* argv is the argv of the command as per unix argv       */
  /* stater is a stateholder which describes jshell state   */
  /* in and out describe the input and output streams...    */
  /*                      ...into the command, for piping.  */
}
```

```
public void horribleDeath() throws BadShellInputException {
  /* throw an exception if the improper input has been given */
}

public boolean isProperCommand(String command) {
  /* return whether command as described in the string... */
  /* ...should be handled by this command object.         */
}
```

... and then add the `Command` to the list of commands using `addCommand` in the constructor of `AdminClientJShell`. One will usually want to use `checkArgs` for the `horribleDeath` implementation, which can die with an appropriate notice if the wrong number of arguments is given (or `minArgs` or `maxArgs`); `String.startsWith` for the `isProperCommand` method, and the combination of `failIfXmlRpcHostUndefined` (which checks if the server to connect to has been defined using `setenv`, dying otherwies) and `makeXmlRpcRequest` (the admin client version). All output should be sent to `out` rather than `stdout` such that pipes and other features work (e.g., `out.println()` rather than `System.out.println()` in `doCommand()`). The `in` and `out` of a `Command` are created as pipes within Java and allow communication between each `Command` thread while running (though for all commands in the Admin Client, implementers should only need to know that they should write to `out` rather than any of the low level details).

C. Add a New ExceptionWriter

`Utility.makeXmlRpcRequest()` will do its best to make an XML-RPC request, but sometimes something will go wrong, throwing an error. The natural notification mechanism for this varies by the usage, whether the call is happening from a graphical user interface, a command line one, or from the server which has no interface at all. `ExceptionWriters`

allow the request method to handle all of these cases without code duplication—instead, simply write a `write` method which displays an error using the string error and an exception object, and this method will be called if something goes wrong. Finally, though the general `makeXmlRpcRequest()` method takes an `ExceptionWriter`, the GUI client, server, and admin client have their own `makeXmlRpcRequest()` methods which call the `Utility` method with the appropriate writer for their context.

D. Add a New Transferrable Type

D.1 Implementing XmlRpcHashSerializable

The first challenge in creating a new transferrable type is to implement `XmlRpcHashSerializable`:

```
public interface XmlRpcHashSerializable {
    abstract public Hashtable toHash();
    abstract public void fromHash(Hashtable h);
    abstract public String toString();
}
```

`toHash` and `fromHash` being defined for all types means that its easy to make more advanced types which have less advanced types embedded in them: for instance, a `toHash()` method of an object which contains a `RowCol` class can add the full hash generated by `RowCol.toHash()` to an entry in the output hash. `toString()` is only for display purposes, and is primarily for ease of display on the admin client and for debugging.

If the only purpose of the type is to be shared between the admin client and server, like the `PlayerConnection` type (because player clients are not interested in the IP and port details of connected players), then this is the end of the process. However, if the type contains game data which needs to be sent to players when changed, then additional work must be done to allow the server to watch the type for changes.

D.2 Allowing the Type to be Watched by the Server

Generally, a new game type will be associated with a single game. Therefore, a game type will usually extend `Observable`, and when the type is added to the `GameStatus` of which it is part (by setting the attribute defined by the type) then the `GameStatus` will be made an `Observer` of the object. For instance, when the `Phases` of an object are set in `GameStatus`, the `GameStatus` object is made an `Observer`, using the `addObserver` method, of the new `Phases` object. When the new type is changed (e.g., for each method in the new type which changes its state), it will need to call two methods in succession: `setChanged()` and `notifyObservers()`. This notifies observers (almost always simply the `GameStatus` object) that the object has changed. Then one must modify the `update` method of `GameStatus` to notify *its* observers that its state has changed, with an appropriate message, given the message which was sent as a notification by the original type. Finally, one must modify the `update` method in `coloredtrails.server.GameStatusSender` to message the clients the appropriate new state when it receives updated information from `GameStatus`. Often, if the state is already transferred when a particular high level change happens, then the last two steps will not be necessary—in other words, adding a new type to `PlayerStatus` would only require that `PlayerStatus` include the new type as part of its state, and be aware when that state changes, since `PlayerStatus` changes are already communicated by `GameStatusSender` when a `PLAYER_CHANGED` notification occurs.

Note: Never send a `GAME_START` change code unless a new game is really starting for the players involved, since otherwise the old `GameStatus` of the players will be clobbered by the new `GameStatus`, which may break the `Observer` relations on the GUI client.

E. Add a New Color

Open `coloredtrails.shared.GlobalColorMap`. The static declaration contains all of the colors known to the server. Adding a new color requires giving it a name and RGB value. This is done by adding to the list using a new `addColor()` call:

```
addColor("colorName", rval_int, gval_int, bval_int);
```

F. Add a New FrontEnd Command Line Flag

See the excellent Getopt documentation which is included in `gnu.getopt` in the source distribution and edit the self-contained `coloredtrails.shared.app.FrontEnd` file.

B. Notes on IntelliJ IDEA

Much of the GUI is constructed using IntelliJ IDEA. While this is generally painless and much easier and satisfying then coding Swing directly, there is one bug / misfeature which I encountered. Occasionally when adding a Swing component not managed by IntelliJ's Swing layout, a GridLayout exception of some sort will be thrown. Usually the fix was simply to add an additional panel with none of its attributes set to the original panel that one was trying to add the non-IntelliJ Swing component to.

C. Additional CVS Notes

The CT³ documentation is under a separate cvs project: `ct3-design` rather than `ct3` (perhaps they should be merged?). This can be acquired as so:

```
$ cvs co -d ':ext:user@drdoom.eecs.harvard.edu:/home/lair/coloredtrails/cvs' ct3-design
Password: <type eecs password>
```

... and includes the CT³ specification as well as this document.