Configuration Class How-To

This document is an introduction on how to implement a game in the Colored Trails (CT) system.

To maximize the flexibility of the Colored Trails system to allow various kinds of games, we separate the game logic from the rest of the system. Thus, the game designer needs only concentrate on the game logic, rather than infrastructure issues such as networking and communicating state changes.

To implement a game, one creates a *game configuration class*. Such a class is what defines the logic of the game; this class will communicate with the server through an established API that will allow control over nearly all aspects of the game. Indeed, the design philosophy of the CT3 system is to allow the game configuration class to customize as much of the game as possible. As a result, we cannot hope to illustrate all possible ways of using the features made available to the game designer, but we will illustrate how all components of the API are used.

To implement a game, one begins by creating a Java class that extends the class coloredtrails.shared.types.GameConfigDetailsRunnable. For illustrative purposes, let us call our subclass MyGame. The source file MyGame.java should not belong to any package and should be saved in the gameconfigs/ subdirectory. At runtime, the game configuration file will be dynamically loaded by the Colored Trails server via the Administrator's "add configuration" command. Thus, the game configuration class executes on the server's JVM.

This document presents the most useful methods of the game configuration class and describes typical usage.

Basic Run Method

When a game begins, the CT3 server will invoke the run() method of the configuration class. This is the only method of GameConfigDetailsRunnable that ***must be*** overridden. In the run() method, we setup the game state, which includes the state of the game board and the states of the players. To set the game state, we first need to obtain the ServerGameStatus instance for the game we are about to start:

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

DEFINING GAME COLORS

We begin by defining the colors for the game. These are the colors that may appear on the board as well as the colors of the chips that players may have:

```
// set game palette
gs.addColorToGamePalette("RGBRed");
gs.addColorToGamePalette("RGBGreen");
gs.addColorToGamePalette("purple1");
gs.addColorToGamePalette("orange1");
```

A full list of colors that can be used is found in the class shared.GlobalColorMap.java

BUILDING THE GAME BOARD

We next set the dimensions of the game board:

```
gs.makeNewBoard(4, 4);  // this makes 4 rows of 4 columns each
```

Then, we assign colors to the board's squares. We create an array of Square objects the same size as the board, and fill each element of the array with a new Square instance:

```
board = gs.getBoard();
Square[][] squares =
```

```
            new Square[board.getRows()][board.getCols()];

for (int i = 0; i < board.getRows(); i++)
      for (int j = 0; j < board.getCols(); j++)
            squares[i][j] = new Square();
```

Then, we set the color of each square in the array. For example, if we want to use colors selected at random from our game palette, we can do the following:

```
String colorname =
            GamePalette.getRandomColorName(gs.getGamePalette());
squares[i][j].setColor(colorname);  // for each i and j
```

Then, we assign the squares to our board instance:

```
board.setSquares(squares);
```

PLACING GOALS ON THE GAME BOARD

We first create Goal instances, one for each goal we wish to place on the board. All goals have a *location*; they may each also have an integer *type* and/or a String *id*. These latter properties are useful when your game has multiple goals, particularly if different goals are intended for different players (e.g., Player 1 must reach Goal 1, while Player 2 must reach Goal 2). We will use the Goal constructor that allows us to specify the goal's type and location:

```
Goal g0 = new Goal(0, new RowCol(1, 1));  // goal 0 has type 0
Goal g1 = new Goal(1, new RowCol(3, 2));  // goal 1 has type 1
```

Goal g0 has type zero and is positioned on row 1, column 1 (rows and columns are numbered starting at zero); goal g1 has type 1 and is positioned at row 3, column 2. **The goal's type is used by the GUI to determine what goal icon to display.** Currently, we have two different goal icons, corresponding to types 0 and 1.

We then place these goals on the board. We use the board object's setGoal() method:

```
gs.getBoard().setGoal(g0, true);
gs.getBoard().setGoal(g1, true);
```

The above calls will place our two goals on the board at the locations specified when we created the goal objects.

We can also remove goals from the board with the setGoal() method:

```
gs.getBoard().setGoal(g0, false);
gs.getBoard().setGoal(g1, false);
```

If we want to move a goal, then we use the board object's moveGoal() method:

```
gs.getBoard().moveGoal(g0, new RowCol(2, 2));  // moves the goal
```

The above call will move goal g0. We can get the location of a goal with the goal object's getLocation() method:

```
RowCol rc = g0.getLocation();  // returns current location
```

The above call will return the goal's current location, even after we move it with Board.moveGoal().

PLACING PLAYER PIECES ON THE BOARD

Each player in the game has a unique *PerGameID*, starting with zero up to the number of players - 1. If the players in your game have different roles, you can use this ID number to define the different roles in your game. For example, the player with ID 0 may be the one that has to move first in your game.

The GameStatus object includes PlayerStatus objects, which represent the location of the player on the board as well as the chips the player possesses. We can obtain a player's PlayerStatus by specifying the player's PerGameID:

```
PlayerStatus ps = gs.getPlayerByPerGameId(0);
```

We specify the location of a player with the setPosition() method of the PlayerStatus class:

```
gs.getPlayerByPerGameId(0).setPosition(new RowCol(1, 1));
gs.getPlayerByPerGameId(1).setPosition(new RowCol(2, 1));
```

Allocating Chips to Players

Each player also possesses a set of colored chips, which the player can use to move on the board. We first create a ChipSet instance, and specify how many chips of each color exist in the set:

```
ChipSet playerChipset = new ChipSet();
playerChipset.set(“RGBRed”, 2);  // two red chips
playerChipset.set(“purple1”, 3);  // three purple chips
```

Note that we are not required to specify a chip count for all colors that are defined for the game; not specifying a chip count for a color is equivalent to specifying that we have zero chips of that color.

We then assign the chip set to a player:

```
gs.getPlayerByPerGameId(0).setChips(playerChipset);
```

Note that we can enumerate the colors that are defined for the game with the GamePalette:

```
for (String color : gs.getGamePalette().getColorListArray())
     playerChipset.set(color, 3);  // three chips of each color
```

GAME PHASES

A game may have any number of phases. A phase is a period of time during which players have certain permissions and prohibitions applied to them. We can define a sequence of phases such that the actions that are allowed change from one phase to the next. A phase terminates when it times out, or it can be advanced by the game configuration class when some event of interest occurs. The sequence of phases will progress linearly in the order in which they are defined, unless the game configuration class specifies the phase that should follow. The phase sequence can also be made to automatically loop.

We first create a ServerPhases object and pass the game configuration class as an argument to the constructor:

```
ServerPhases ph = new ServerPhases(this);
```

We then define a sequence of phases in the order they should occur. The game will begin with the first phase we define. For each phase, we specify a phase name (to be displayed in the GUI) and the length of the phase in seconds:

```
ph.addPhase("Communication Phase", 30);
ph.addPhase("Exchange Phase", 10);
```

```
ph.addPhase("Movement Phase", 10);
ph.addPhase("Feedback Phase", 10);
```

We then indicate whether we wish this sequence to automatically loop or not:

```
ph.setLoop(false);
```

Finally, we pass the phases object to our GameStatus instance:

```
gs.setPhases(ph);
```

The ServerPhases object does not define the permissions and prohibitions applied to the players; it only defines phase names, phase lengths, and the sequence. The permissions and prohibitions are set elsewhere in the game configuration class, as described below.

beginPhase(String phaseName) Method

The beginPhase() method of the configuration class is called by the CT3 server at the beginning of each game phase. The argument passed to this method is a string giving the name of the game phase that is about to start (see Defining Phases, above). The beginPhase() method allows you to perform any tasks that are appropriate for your game at the beginning of each phase. For example, it may be that your game has one phase in which players are only allowed to communicate with each other and transfer chips, and another phase in which players are only allowed to move on the board. Such player actions can be allowed or disallowed by the game configuration class. As another example, it might be that you want to automatically provide additional chips to certain players at the beginning of certain game phases. Yet another example might be that goals are to move at the beginning of certain phases. You can of course add functionality beyond what is described here.

Indefinite Phase lengths

Game phases may last indefinitely. This is useful for purposes of having practice phases (e.g., in a tutorial) or if for synchronization points during a game. The configuration file “lib/adminconfig/PhaseTest.txt” provides an example for how to use this functionality. It includes two phases with indefinite length. In the Exchange phase, the system waits for one of the players to accept a proposal before moving to the next phase. This next phases allows players to move their pieces by hand. As soon each player has moved her piece at least one square,

this phase will end. The final phase is timed; player scores are calculated and displayed here. The game ends after this last phase.

SETTING PLAYER PERMISSIONS

The CT3 system allows the game configuration class to control what actions players are permitted to take. Currently, there are three types of action that can be permitted or prohibited: 1) a player may communicate with another agent by sending that agent a DiscourseMessage (see below and sample code for agents for further details), 2) an agent may be allowed to transfer chips to another agent, 3) an agent may be allowed to move on the board. We allow or disallow these actions by sending a boolean argument to the following PlayerStatus methods:

```
PlayerStatus ps = gs.getPlayerByPerGameId(0);
ps.setCommunicationAllowed(true);
ps.setTransfersAllowed(false);
ps.setMovesAllowed(false);
```

Note that we can give different permissions to different players in each game phase.

AUTOMATICALLY GIVING CHIPS TO PLAYERS

To add chips of a certain color to a player's chip set, we can do the following:

```
PlayerStatus ps = gs.getPlayerByPerGameId(0);  // get Player 0
ChipSet cs = ps.getChips();  // get Player 0's current chips
cs.add("purple1", 5);  // add five purple chips
cs.set("RGBRed", 2);  // player now has two red chips
ps.setChips(cs);
```

We first get the PlayerStatus instance for the player we are interested in. Then we get that player's chip set. Next, we add some number of chips with the add() method. We can use this method to add chips of a color not previously possessed by the player; we can also use this method to take away chips of a certain color by passing a negative integer argument. Note that the add() method does not require the result of adding to be non-negative. We can also set the number of chips of a certain color with the set() method. Once we have modified the chip set as desired, we must update the player's state with the setChips() method; this method will cause the CT3 server to inform other players about this player's new state.

endPhase(String phaseName) Method

The endPhase() method of the game configuration class is called by the CT3 server at the end of each game phase; a phase ends when its timer reaches zero or when some event is used to trigger the end of the phase (see below). The String argument passed to the endPhase() method is the name of the phased that has just ended. Just like the beginPhase() method, we can use the endPhase() method to perform any tasks that are appropriate for our game.

ENDING THE GAME

One important operation that we will likely perform in the endPhase() method is ending the game. We will likely end the game when a particular phases has completed, or when we have looped through the sequence of phases a certain number of times (if we are looping then we are responsible for keeping track of the number of times we have looped; the CT3 system does not currently track this). In the example below, the game phase "Feedback Phase" is the last phase of the game:

```
if(phaseName.equals("Feedback Phase"))
      gs.setEnded();  // tell the CT3 server that the game ended
```

AUTOMATIC MOVEMENT

Another type of operation we might want to perform at the end of a phase is to automate the movement of player pieces. In the example below, we move each player's piece to a board location that will maximize the player's score given the chips the player possesses. This example makes use of two new classes. The Scoring class is used to compute a player's score given the game state; we discuss this more below. The BestUse class is used to compute the best move sequence of a player given the game state; it requires the Scoring instance to determine which move sequence maximizes the player's score.

SCORING

The Scoring constructor currently takes three integer arguments. The first argument is the number of points awarded to a player if it lands on a goal square. The second argument is a penalty for each square the player is away from the goal square; in the example below, if the player can only get to within two squares of the goal, it gets a penalty of 20 points (i.e., -20). The third argument is

the value of each chip the player doesn't use for moving.

```
Scoring s = new Scoring(100, -10, 5);
```

Note that the Scoring class does not take into account the states of other players in the game; of course, you can define new scoring rules to do so.

To get the score of a player in a certain state, we use the Scoring instance as follows. The example below gets the score of Player 0 given its current location, current set of chips, and the location of Goal g0 (see above on defining goals):

```
PlayerStatus ps = gs.getPlayerByPerGameId(0);  // get Player 0
double thescore = s.score(ps, g0.getLocation());
```

BEST USE OF CHIPS

The BestUse class can be used to calculate the best sequence of moves of a player, given the player's current location and chips, and the location of the goal it needs to reach.

```
Scoring s = new Scoring(100, -10, 5);
PlayerStatus ps = gs.getPlayerByPerGameId(0);  // get Player 0
int goaltype = 0;  // Player 0 wants to reach Goal 0

BestUse bu = new BestUse(gs, ps, s, goaltype);
Path p = bu.getPaths().get(0);
```

We use the getPaths() method to obtain an ArrayList<Path> instance, which is a list of optimal paths for the player. A Path instance represents a sequence of RowCol objects; see shared.types.Path java code for details.

AUTOMATIC MOVEMENT

Using the Scoring and BestUse classes, we can achieve automatic movement as follows. The example below moves players after the "Communication Phase" ends:

```
if (phasename.equals("Communication Phase"))
{
      // 100 for goal, -10 per unit distance, 5 for each chip
     Scoring s = new Scoring(100, -10, 5);

      ArrayList<PlayerStatus> playersInGame = gs.getPlayers();
```

```
      for (PlayerStatus ps : playersInGame)
      {
            ps.setMovesAllowed(true);  // allow moving on board

            // calculate best use of chips possessed
            BestUse bu = new BestUse(gs, ps, s, 0);
            Path p = bu.getPaths().get(0);
            gs.doPathMove(ps.getPerGameId(), p);  // move player
            // set the player's score now that it's moved
            ps.setScore(getPlayerScore(i));
      }
}
```

The doPathMove() method will cause the CT3 server to move the specified player along the specified path. The CT3 server checks to make sure that the player has the chips required to complete the entire path. **If the entire path cannot be traversed, then the player is not moved at all**; an alternative policy that might be implemented in the future would be to allow the player to move the longest feasible prefix of the path. The getPlayerScore() method is described below.

Discourse Methods

The doDiscourse() method of the configuration class is invoked by the CT3 when a player has sent a discourse message, addressed either to another player (e.g., to propose exchanging chips) or to the CT3 server itself (e.g., to request information of some sort—note that you can create your own DiscourseMessage subclasses to enable transmission of information to, from, and between players).

The doDiscourse() method provides a way for the configuration class to intercept messages and examine them. Such examination allows the configuration class to prohibit certain messages from being relayed to a recipient player, or take certain actions, depending upon the message type or message content.

The DiscourseMessage class is the super class of all message classes; different subclasses represent different types of messages between players. The type of the message is held in "msgType" field in the message class. The content of the message is flexible and represented by a Hashtable named "contents".

The example below shows how we can examine the DiscourseMessage and implement an automatic transfer of chips when the message indicates that a player has accepted an offer made by another player to exchange chips. The

example uses the doTransfer() method to make the CT3 server actually transfer chips from one player to another. The example also uses the Phases.advancePhase() method to move the game to the next phase, regardless of how much time remains in the phase; this is an example of an event-based phase advance.

```
public boolean doDiscourse(DiscourseMessage dm)
{
  ServerGameStatus gs =
       ServerData.getInstance().getGameStatusById(getGameId());

  // go ahead and relay the message
  boolean result = gs.doDiscourse(dm);

  // automatic exchange of chips upon acceptance of a proposal
  // is this message type a response to a proposal?
  if (dm.getMsgType().equals("basicproposaldiscussion"))
  {
    // make a copy of the response message
    BasicProposalDiscussionDiscourseMessage bpddm =
              new BasicProposalDiscussionDiscourseMessage(dm);

    // did responder accept the proposal?
    if (bpddm.getCommentary().equals("accept"))
    {
      // retrieve original proposal from response
      BasicProposalDiscourseMessage pdm =
            new BasicProposalDiscourseMessage((Hashtable)bpddm.
                                getDataValue("proposalMessage"));

      // Get the sender and recipient chips from proposal
      ChipSet senderChips = pdm.getSenderChips();
      ChipSet recipientChips = pdm.getRecipientChips();

      // Check that sender and recipient have chips to transfer
      if (gs.getPlayerByPerGameId(pdm.getFromPerGameId()).
                              getChips().contains(senderChips) &&
          gs.getPlayerByPerGameId(pdm.getToPerGameId()).
                              getChips().contains(recipientChips))
      {
        System.out.
            println("#### EXECUTING AUTOMATIC CHIP TRANSFER");
```

```
        // transfer the chips from the sender
        super.doTransfer(pdm.getFromPerGameId(),
                         pdm.getToPerGameId(), senderChips);

        // transfer the chips from the recipient
        super.doTransfer(pdm.getToPerGameId(),
                         pdm.getFromPerGameId(), recipientChips);

        // automatically advance the game's phase
        gs.getServerPhases().advancePhase();
      }
      else
      {
        System.out.println("Not enough chips to transfer");
      }
    }
  }

  return result;
}
```

Automatic Exchange

A chip exchange between players A and B consists of two chip transfers, one from A to B and another from B to A. Although the chip-transfer code checks whether the sending player possesses the chips specified by the transfer, we would like a chip exchange to occur only if both players possess the specified chips; that is, if A possesses the chips to be sent to B, but B lacks chips to be sent to A, then we do not want A to send chips to B. The information about the chips to be transferred and the sender and recipient id is held in the discourse message. To check for chip availability, use the method contains() in the ChipSet class.

int getPlayerScore(int perGameId) Method

The getPlayerScore() method of the configuration class is called by the CT3 server when the server needs to know the current score of the player specified by the player's perGameId. The return value is an integer. Future versions of CT3 may change the return type to a double. Typical code for this method would be

the following:

```
public int getPlayerScore(int perGameId)
{
      ServerGameStatus gs =
      ServerData.getInstance().getGameStatusById(getGameId());

      PlayerStatus ps = gs.getPlayerByPerGameId(perGameId);

      // assume g0 is a Goal object we've defined elsewhere
      return (int)Math.floor(s.score(ps, g0.getLocation()));
}
```

Of course, we would change the above code if different players need to get to different goals.

Adding and Changing Functionally

While overriding GameConfigDetailsRunnable.run() is the only requirement for a game configuration class, there are also a number of handler methods for different actions that occur throughout the course of a game. The following table lists these handler methods and briefly discusses their default operation if they are not overridden.

Method Name	**Default operation**
doMove()	Execute a move if the player involved has the chips and the new position is a neighbor of the current position and remove the requisite chip
doTransfer	Transfer chips from one player to another as requested if the players have the necessary chips. See below for more detail.
doDiscourse	Relay a discourse action between players. See below for more detail.
beginPhase	Invoked before each game phase begins; you may use this method to change permissions based upon which game phase is beginning, for example.
endPhase	Invoked after each game phase ends

The above handler methods provide the foundation for implementing player roles and player commitments (e.g., enforcement of trade proposals). The configuration class must supply additional code to track the actions taken by players if, for example, the game designer wishes to implement commitments.

Useful Get Methods

Class Name	Method Name	Returns
ServerData	getInstance()	ServerData
GameConfigDetailsRunnable	getGameId()	gameId
ServerData	getGameStatusById()	ServerGameStatus
ServerGameStatus	getPlayerByPerGameId	PlayerStatus
ServerGameStatus	getBoard	Board
ServerGameStatus	getGamePalette	GamePalette
Phases	getCurrentPhase	Current phase name
ServerGameStatus	getScore	Score obtained by this player in its current game
BestUse	getPaths()	Best path to get to the goal
DiscourseMessage	getMsgType()	Message type
DiscourseMessage	getDataValue(key)	Value by key in the content of the message

Examples

Examples of a working configuration files are in the gameconfigs/ directory of the source distribution.

Partial Visibility Functionality

By default, players in Colored Trails have access to the entire game state, e.g., the colors on board, locations of player pieces, locations of goals, and chips possessed by each player. Nevertheless, it may be the case that an experiment calls for one or more player to have uncertainty about the game state. For example, we might want a player to lack information about what chips another player possesses, or have uncertainty about the location of a goal. This document discusses the facility in Colored Trails that is used to introduce uncertainty about game state.

Approach

Colored Trails represents game state using several classes. These classes are Square, Goal, Board, ChipSet, PlayerStatus, GameStatus, GamePhase, and Phases. ColoredTrails currently allows the experimenter to introduce uncertainty in all of these representation classes except for GamePhase and Phases (these two classes will be addressed in a future release). The APIs for these representation classes are designed to operate under the default assumption for full information.

All of the representation classes are subclasses of CTStateContainer. In this superclass, all game state information is represented with feature-value pairs. For example, by default, ChipSet is encoded with feature-value pairs where each feature is a chip color and the corresponding value indicates the number of chips of that color. The ChipSet API hides this approach to encoding game state. To introduce uncertainty into a Colored Trails game, the experimenter needs to use the more general API of CTStateContainer, where feature-value pairs can be accessed and manipulated directly.

By allowing direct access to the feature-value pairs of a representation class, the experimenter is free to modify information in arbitrary ways. Thus, an experimenter may not only remove information, but may re-represent the information. For example, it may be the we do not want Player 1 to know anything about what chips Player 2 possesses; this is a case where we simply remove data from the ChipSet. Alternatively, it may be that we want Player 1 to know something short of full information; for example, Player 1 may not be told the exact ChipSet that Player 2 has, but may be told that Player 2 has at least two blue chips and no more than one red chip. By encoding state with feature-value pairs, the experimenter has freedom to represent game state in any way that is appropriate for the experiment. The only requirement is that the GUI code, and any computer agents, all understand the feature-value convention used by the experimenter. Note that a value for a feature may be any arbitrary object. Thus, an experimenter may devise a new classes to assist with re-representation of game state.

Filter Methods

Before the Colored Trail server sends game state information to players, it gives the game configuration class an opportunity to modify or filter game state data. This is where the experimenter can introduce uncertainty into the game. There are currently three filter methods defined in GameConfigDetailsRunnable. To introduce uncertainty, the experimenter needs merely to override the appropriate method in the game configuration subclass. The default

The filter methods are:

```
public Board filterBoard(Board fromserver, int perGameId)

public Set<PlayerStatus> filterPlayerStatus(Set<PlayerStatus> fromserver,
                                            int perGameId)

public GamePalette filterGamePalette(GamePalette fromserver, int perGameId)
```

The first parameter to each filter method is a representation class instance that is to be modified in some way. Note that this class instance is a copy of the game state data held internally by the server. Thus, you are free to modify the instance without fear of corrupting the server. The second parameter to each filter method is the ID of the player to which the modified game state is to be addressed. Thus, the server will call each filter method once for each player in the game. Note that the server broadcasts game state information to each player regardless of to whom the data is addressed. This provides the opportunity for a player to learn another player's view of the game state. In future versions of Colored Trails, we will introduce permissions that can be used to prevent a player from seeing what another players knows about the game state.

filterBoard

Override this method to modify what a player is told about the state of the game board. The Board object is composed of several Square instances, which themselves may contain Goal instances. Here is one example of introducing uncertainty:

```
// Player 0 doesn't know the colors of squares that contain goals
public Board filterBoard(Board serverboard, int perGameId)
{
        if (perGameId == 0) // we filter information for this player only
                for (RowCol rc : serverboard.getGoalLocations();)
                        serverboard.getSquare(rc).remove("color");

        return serverboard;
}
```

The above example removes, for Player 0, information about the color of squares that contain goals. Thus, Player 0 will know where goals are, but will not know what color chip it needs to land on a goal square; instead, perhaps, Player 0 will need to negotiate with or observe other players to obtain this information. This example code shows a mixture of calls to the higher-level standard API (the getSquare method of Board) and the lower-level API that is common to all representation classes (the remove method of CTStateContainer). Because we have removed the "color" feature of goal squares, the GUI code must first check

whether a square has a color feature before it attempts to render the square's color (otherwise a null pointer exception will occur). When a square lacks a color feature, the square will instead show some background color (which should not be a chip color) or perhaps the GUI can render a special background pattern to signify that the color of the square is unknown. Similarly, when a computer agent tries to reason about what chips it might need to reach a goal square, it will need to take into account the possibility that it will lack knowledge about a square's color and therefore need to reason under uncertainty.

filterPlayerStatus

Override this method to modify what a player is told about the state of all players in the game, including itself. PlayerStatus instances hold information about where a player is located on the board and what chips a player possesses. The first argument is a collection of PlayerStatus instances, one instance for each player in the game. Because this filter method is called once for each player in the game, the experimenter can provide each player with a completely different view of each player's game state. For example, Player 0 may not know where Player 1 is located on the board, but know Player 1's chips; at the same time, Player 1 may know Player 0's location but not Player 0's chips.

```
public Set<PlayerStatus> filterPlayerStatus(Set<PlayerStatus> serverps,
                                            int perGameId)
{
        if (perGameId == 0)
                for (PlayerStatus ps : serverps)
                        if (ps.getPerGameId() == 1)
                                ps.remove("position");
        else if (perGameId == 1)
                for (PlayerStatus ps : serverps)
                        if (ps.getPerGameId() == 0)
                                ps.remove("chips");
}
```

filterGamePalette

Override this method to modify what a player knows about the colors defined for the game. This option is perhaps the least useful of the three filter methods, but is supplied for completeness.

Agent-Side Code

The filter methods allow the experiment designer to control what each agent learns about the game state. Here we discuss how agent-side code processes data when it is being filtered.

The class coloredtrails.agent.ColoredTrailsClientImpl has a method called onMessage that is called each time the server sends game state information (as well as other kinds of messages). This method checks to see if the perGameId of the intended recipient matches the perGameId of the agent receiving the message; if the IDs match, then the agent's state information is updated according to what is received in the message, otherwise the message is ignored. For many situations that involve uncertainty, this default behavior is appropriate. In some cases, however, an experimenter may wish for an agent to examine state information addressed to another player. For example, if a player needs to reason about what another player knows about the game state, then the player should read messages addressed to that other player. Code for doing this might look something like the following:

```
...

else if (cmd.equals(ColoredTrailsServer.BOARD))
{
        try
        {
                ObjectMessage obj = (ObjectMessage)msg;
                // find out who this game state data is addressed to
                int forPerGameId =
                        obj.getIntProperty(ColoredTrailsServer.SPECIFICMSG);
                // is it addressed to me?
                if (forPerGameId == perGameId)
                {
                        Board board = (Board)obj.getObject();
                        getGameStatus().setBoard(board);
                        boardUpdated(board);
                        ...
                }
                // is this message for Player 3 (who I need to reason about)?
                else if (forPerGameId == 3)
                {
                        // update a different Board field representing what Player 3
knows
                        // don't update your own Board instance!
                }
        }
        ...
}
```