Architectural Design

Team 28
Comp 3004
April 8th 2014

Table of Contents

Client/Server Integration

Overall Behavior:

Client/server interaction is intended to follow the Observer design pattern, where multiple clients will register interest in the state of the game playing out on the server, and, upon changing the state of the game, the server will send notifications back to any connected clients.

Classes involved:

server.logic.ConnectionLobby deals with incoming connections which represent client/observer subscriptions, and spawns new server.logic.PlayerConnection threads for each subscribed client. When the state of the game is changed on the server side code, it dispatches notifications to subscribed clients by dispatching an event class in the common.event.network package, all of which inherit from AbstractNetworkEvent. These events are transmitted across the network to the subscribed clients

Motivation:

The primary motivation for using this pattern was the decoupling of the logic for changing the state of the game, which is housed on the server side code, from the logic for viewing the state of the game and receiving user input which is on the client side.

Server Handling of Client Requests

Overall Behavior:

Handling of client requests to, for example, move a creature is handled on the server side as an event based system. Clients will send a message to the server that indicates what they would like the server to do, and the server will dispatch an event on it's own internal bus, to be handled by different classes. (client will receive an update on the results of the request after it is handled)

Classes Involved:

Commands that the user wants to perform are represented on the server side as objects in the server.event.internal package. These objects are constructed in response to some message received from the client representing what they want the server to do, and dispatched as events, which get consumed by classes in the server.logic.game.handlers package.

Motivation:

Abstraction of things the user can do (move a creature) from the input method (push button 5, or button 3...) as well as separation of the logic for handling the changing of the game state into multiple classes (one for movement, one for recruitment etc...) as opposed to having one 'God class'(our first implementation) that manages all user input.

Server Handling of Game Rules

Overall Behavior:

Before handling a request from the client to do a particular action (represented as a command object) the command handler will call a special method to validate the request and ensure that game rules are being respected, if not it will throw an exception that bubbles back up to the requesting client.

Classes Involved:

All of the validation logic is contained in public static methods in the classes in the server.logic.game.validators package. Objects that want to validate a particular command, pass the command parameters along with the current game state to these methods.

Motivation:

Separation of the logic for validating a command from it's handling, so that it can be referenced elsewhere in the system (by the clients themselves for example). Reduces the volume of code in the command handler classes.

Server Management of Game State

Overall Behavior:

There is a single game state shared and modified by all of the command handler classes.

Classes Involved:

There is a GameState class that acts as container of all the state information of the game. This class references many other classes that are involved in specifying the current state of a game. The GameState class itself does not contain any logic for handling or validating commands, it merely acts as a container, providing getter and setter methods for different elements of the game state.

Motivation:

Simplify saving and loading of games to the saving and loading of a single object (GameState). Reduce volume of code in handler classes by placing state information elsewhere, and reduce complexity of connections between classes by having a single container one can reference to obtain any state related information they require.

Client Handling of User Input

Overall Behavior:

General user input is captured by the main window which contains a number of controls for placing things on the main board, more specialized user input is handled by dedicated windows that pop up when needed.

Classes Involved:

Input for the main interface panel is represented by client.gui.Board, handling of input into this panel is done by the nested class client.gui.Board.Controller. More specialized panels are defined in client.gui.components, handling of input into these classes typically takes the form of ActionListeners on the interface controls that directly send messages to the server.

Motivation:

Consolidation of main input handling to a single place, reduction in volume of code by using specialized panels for more complex input.

Representation of Tiles

Overall Behavior:

We made use of the Adapter design pattern to represent the different kinds of tiles in the game (forts, vs special characters).

Classes Involved:

ITileProperties is the adapter interface shared by all the different types of tiles in the game. The two main implementing classes are TileProperties and TwoSidedTileProperties representing things with nothing on their reverse side, and things like SpecialCharacters or Buildings which have something special on their reverse side.

Motivation:

Allow the bulk of the code to treat game tiles generically as in most cases, these two types of tiles are identical, but maintain some special handling for operations on tiles with something special on their reverse side.