CS619
Project 1

Instructor: Karen JIN

September 19, 2012

The purpose of this project is to refresh your Java programming skills and to start thinking about object oriented design. You will learn to work in teams and also familiarize yourself with Eclipse IDE and other development tools.

1 Background

A classical D&D computer game consists of a dungeon (maze), where the purpose of the game is to make one's way through the dungeon, entering at the "entrance" of the maze and exiting at its "exit". To make things more challenging, the layout of the dungeon is not known a priori and there are obstacles (zombies, tar-pits, and doors) that must be overcome by using objects (flares, ladders, and keys) that are found along the way.

1.1 The Game System

In this project, your team will design and implement a simple game player for a Dungeons & Dragons (D&D) maze game. Please download the zip file `MazeGame.zip` from the course website. The archive (`MazeGame.zip`) contains a classical D&D game system. The system comprises three parts: i) a game server, ii) a game viewer, and iii) a game player. The game server implements the actual game, enforces all the rules, advances the game based on the player's moves, and provides updates to the game viewer. The game viewer client allows humans to observe the progress of the game. The game player client is responsible for deciding what the player should do next. Typically, the game player is a human, but in our case the game player will be a program. Both the game viewer and the game player clients connect to the game server via the network. Figure 1 provides an overview of the system.

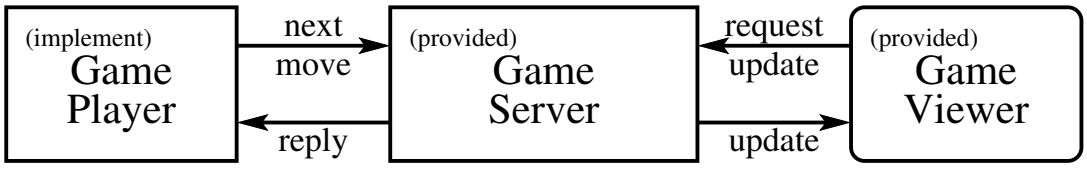

Figure 1: The Game Components

In theory, only the game server and the game player need to be running in order to play the game. However, in practice, it is also useful to watch the game being played via the game viewer.

The game viewer is provided as an applet that can be executed in a web browser or as a Java application. To load the applet locally, simply unzip the `MazeGame.zip` archive into a directory on the local machine and open the HTML file `GameViewer.html` in the web browser or run the application at the command line:

```
java -cp GameViewerApplet.jar GameViewerApplet
```

Once the applet is loaded it is ready to connect to the game server. To do this, four pieces of information are needed:

Host : the location of the game server
Port : the address of the game server on the host
Game : the name of the game being viewed
Password : the password associated with the game

The defaults provided in the first three fields assume that the server is being run locally (on the same machine as the browser), as described below. There is presently only one type of the game that the server supports, namely `Maze`. Thus, unless additional parameters must be passed to the game, the *Game* field need not change. The default port for the server is `4242`, which need not change as well. Lastly, the *Password* field is used to uniquely identify an instance of the game being played. This password must be at least 8 characters long and must not contain spaces. This is the same password that is specified when invoking the game player.

To connect to the server click on the "Connect" button. If the game, identified by the *Game* and *Password* fields is in progress, the viewer will display the game and update the screen as the game proceeds. If the game is not in progress, or the server is not running, the viewer will attempt to reconnect every couple seconds. Clicking on the "Disconnect" button will disconnect the viewer from the server as well as ceasing attempts to connect to the server. Note, the viewer simply displays the game in progress, it does not allow interaction with the game. The server is solely responsible for managing the game(s).

The server can either be used locally or remotely. The benefits of using a local server is that no Internet connection is needed. However, if to use the remote server, obviates the need for running one locally. To run the server locally, open a terminal window on the local machine, change to the directory containing the unzipped archive, and use the command

```
java -cp BasicGame.jar GameServer
```

which will run the server on port `4242`.

To initiate a game, the game player connects to the game server and specifies the game name and password. The game server instantiates the game and then waits for the game player to issues moves that move the player through the game. Once the game is instantiated the game viewer will also be able to connect and view the game. If the game player disconnects, the game is terminated.

Just like the game viewer, the game player requires the same four parameters (*Host*, *Port*, *Game*, and *Password*) to be specified in order to connect to the server and instantiate the game. These are specified as command line arguments to the game player.

Within `BasicGame.jar`, a sample game player (`SamplePlayer`) is provided. For example, to run the sample game player so that it connects to a local game server, and starts the `Maze` game with password `coolness` use the command:

```
java -cp BasicGame.jar SamplePlayer localhost 4242 Maze coolness
```

The `SamplePlayer` will start up, connect to the server, which should already be running, and then start playing the instantiated game. Once the game viewer connects to the same server with the same password, the game can be viewed. Try this out to ensure that the system is working. Running the `SamplePlayer` will also serve as a live illustration of the discussion in the next section.

Useful Tip The server generates a random maze every time a game is instantiated. However, for debugging purposes it is often useful to use the same maze. This can be done by passing a parameter to the game sever specifying a specific board number. This is done by appending the board number to the *Game* parameter, separated by a colon (`:`). The board number is displayed at the start of each game in the message box in the right bottom section of the game viewer. For example, to specify board `123456789`, the *Game* parameter would be `Maze:123456789`, e.g.,

```
java -cp BasicGame.jar SamplePlayer localhost 4242 Maze:123456789 coolness
```

Also, specify `Maze:123456789` in the *Game* field of the game viewer. This is because the *Game* parameter of the game player must match the *Game* parameter of the game viewer.

Useful Tip Have the server and the game viewer running before starting the game player. First start up the server. Second, start up the viewer. Third, set the parameters in the viewer and click on the "Connect" button. The viewer will start attempting to connect. Fourth, run the game player. This minimizes the risk of missing any of the game player's moves.

1.2 The Rules of the Game

The rules of the game are relatively simple. The maze is situated on a grid that is 50 spaces wide and 40 spaces high. Each of the grid elements represents a wall, a passage way, or a passage way with an object. A player cannot move through walls, but can move through passage ways. However, objects in the passage ways may obstruct the player. The entrance to the maze is on the left wall, and the exit is on the right wall. The maze is 1-connected, meaning that there is exactly one path between any two passage ways in the maze. Thus, there exists a path between the entrance and the exit.

The game player has to move the heroine from the entrance, through the maze, to the exit, and stop when the exit is reached. Since the maze is based on a grid, the player can move in the four cardinal directions: up, down, left, and right. Apart from walls, players may be obstructed by three different kinds of objects: zombies, tar-pits, or doors. However, unlike walls, these objects can be overcome. In addition to zombies, tar-pits, and doors, there are also flares, ladders, and keys scattered throughout the maze. A flare can be used to vapourize two zombies; a ladder can be used to cross a tar-pit; and a key will open one door. Note there are two kinds of doors, horizontal and vertical, but they all have the same lock, so a key will open any door in the maze.

To pick-up a flare, ladder, or key, the heroine must be adjacent to the object. The heroine may carry an unlimited number of flares and keys but only one ladder at a time (ladders are clunky). To apply a flare to a zombie, a ladder to the tar-pit, or a key to a door, the heroine must also be adjacent to the obstruction. A flare is good for two applications, and a ladder or key is good for one application. I.e., the flare will be used up after it is applied to two zombies, a ladder will be used-up once it is applied to a tar-pit, and a key will be used-up once it is applied to a door. The

system keeps track of how many flares, ladders, and keys the heroine is holding and will not allow the player to apply an object that it does not hold.

Unlike the game viewer, which shows the entire maze, the heroine only sees the eight adjacent grid elements. Consequently, the heroine will need to explore the maze in order to locate the exit.

In one move the heroine may modify the state of one its adjacent squares. This includes:

1. picking up a flare, key, or ladder from an adjacent square,
2. applying a flare, key, or ladder to an obstruction such as a zombie, door, or tar-pit in an adjacent square, or
3. moving to the adjacent square if it does not contain an obstruction.

The heroine can only apply an object to an obstruction if she is carrying the object, i.e., a flare to a zombie, a ladder to a tar-pit, and a key to a door. The game ignores invalid moves such as attempts to apply an object that is not available, or to move off the grid, through a wall, through an obstruction, or more than one square. That is, the heroine's current state does not change. On a valid move, the game will update the heroine's state.

1.3 Technical Details

The game player communicates with the server using the *GameConnection* class. Additionally, two container classes, *GameUpdate* and *GameMove* encapsulate the communication that takes place between the game player and game server. Specifically, the game player sends game moves to the server and receives game updates.

1.3.1 The *GameConnection* Class

The *GameConnection* class has many methods, but only five of them, including one constructor, are needed. The five methods are:

`public GameConnection( String host, int port ) throws Exception`
: Creates a game connection to the server.

`public void setUpPlayerClient( String game, String pw ) throws Exception`
: Sets up the connection for a game player.

`public Dimension getBoardSize() throws Exception`
: Determines the size of the maze.

`public GameUpdate sendMove( GameMove gm ) throws Exception`
: Sends a move to the server and returns a game update.

`public void disconnect()`
: Disconnects from the server.

`public GameConnection( String host, int port ) throws Exception`

This constructor is used to create a new connection to the game server. It takes two parameters *Host* and *Port*, which specify the location of the server. These are provided by the user on the command line (see examples in the previous section). If the connection cannot be made an exception is raised. For example, the following code creates a connection to a local server on port `4242`:

```
GameConnection c = new GameConnection( "localhost", 4242 );
```

`public void setUpPlayerClient( String game, String pw ) throws Exception`

This method configures the connection to the game server for player use. It takes two parameters, the *Game* and the *Password*, both of which are also specified by the user (see examples in the previous section). If the configuration fails, an exception is raised. The configuration should be done immediately after the connection to the server is instantiated. E.g.,

```
c.setUpPlayerClient( gameId, password );
```

`public Dimension getBoardSize() throws Exception`

This method returns a *Dimension*[1] object containing the size of the grid (maze). This is useful to determine the entrance and exit of the maze, and can be called any time after the connection has been configured. If the connection fails, an exception is raised. E.g.,

```
Dimension bd = c.getBoardSize();
```

`public GameUpdate sendMove( GameMove gm ) throws Exception`

This method sends a game move to the server and returns a response from the server in the form of a game update. The update represents the position of the player after the move. If the move was invalid, the state of the player will be unchanged. The update contains the player's position and the visible portion of the board, i.e., comprising the eight squares surrounding the current position. It is possible to query the current location and surroundings of a player by sending the nilpotent move (described below). This should be done at least once at the beginning of the game to ascertain the starting position. If the connection breaks during the move, an exception is raised. The following is an example of how to query the current location using the nilpotent move, followed by an example of a "next" move.

```
GameUpdate u = c.sendMove( new GameMove( (byte)0, new Point( 0, 0 ) ) );
...
u = c.sendMove( new GameMove( b, newPos ) );
```

`public void disconnect()`

The method closes the connection. The server will terminate the game as soon as the connection is closed. E.g.,

```
c.disconnect();
```

To understand the *GameMove* and *GameUpdate* classes a brief word about the structure of the game board is needed.

[1] imported from `java.awt`

1.3.2 The Game Board

The game board is represented by a 2-Dimension grid of fixed size, which can be determined using the `getBoardSize()` method. Each grid element (square) can contain a game piece, which is identified by a *byte* value $[0 \ldots 255]$. In the case of the maze, game pieces include walls, zombies, tar-pits, doors, flares, ladders, keys, and most importantly, the heroine. Each type of game piece has a unique ID, which is the byte value stored in the square that contains the game piece. These IDs are found in the class *MazeIDs*:

```
public class MazeIDs {
  public static final byte WAY        = 0x0;
  public static final byte WALL       = 0x1;
  public static final byte PLAYER     = 0x2;
  public static final byte KNOWN_WALL = 0x3;
  public static final byte KEY        = 0x5;
  public static final byte V_DOOR     = 0x6;
  public static final byte H_DOOR     = 0x7;
  public static final byte ZOMBIE     = 0x8;
  public static final byte FLARE      = 0x9;
  public static final byte TAR_PIT    = 0x0A;
  public static final byte LADDER     = 0x0B;
  public static final byte COV_PIT    = 0x0C;
}
```

where

`WAY` denotes an empty passageway.
`WALL` denotes a maze wall that has not yet been passed.
`PLAYER` denotes the heroine.
`KNOWN_WALL` denotes a maze wall, that has been passed in the past.
`KEY` denotes a key to open a door.
`V_DOOR` denotes a door in a vertical passageway that can be opened with a key.
`H_DOOR` denotes a door in a horizontal passageway that can be opened with a key.
`ZOMBIE` denotes a zombie in a passageway that can be toasted with a flare.
`FLARE` denotes a flare in a passageway.
`TAR_PIT` denotes an uncovered tar-pit that can be covered with a ladder.
`LADDER` denotes a ladder in a passageway.
`COV_PIT` denotes a tar-pit covered with a ladder, and hence passable.

Consequently, a move simply consists applying a board piece to a position on the board. Specifically, to pick up or apply an object such as a flare, ladder, or key, the ID of the object and the destination is specified. To move the player, the player's ID and destination are specified.

1.3.3 The *GameMove* Class

The *GameMove* class is used to encode the next move in the game. For each move the game player instantiates a *GameMove* object, and calls the `sendMove()` method, passing the *GameMove* object as a parameter. The constructor for the *GameMove* object takes two parameters: i) a byte ID of the game piece being applied to a board location, and ii) a *Point*[2] object representing the location

[2]Imported from `java.awt`.

on the board (see example in the previous section). As mentioned before, the nilpotent move, comprising ID 0, and location $(0, 0)$, can be used to query the current state of the player.

The response to a move is a *GameUpdate* object.

1.3.4 The *GameUpdate* Object

The *GameUpdate* object is returned after each move and contains a "snapshot" of what is visible to the player. The *GameUpdate* class has three useful methods:

`public Point location()`
: Returns the location of the chunk of the board visible to the player.

`public Dimension size()`
: Returns the size of the chunk of the board visible to the player.

`public byte value( int x, int y )`
: Returns the byte value in a square within the visible chunk of the board.

As mentioned earlier, the game board is a grid, of size $W \times H$. However, only a small chunk of the board is visible to the player at any one time, specifically, the adjacent squares. An update provides access to this small chunk as illustrated in Figure 2.

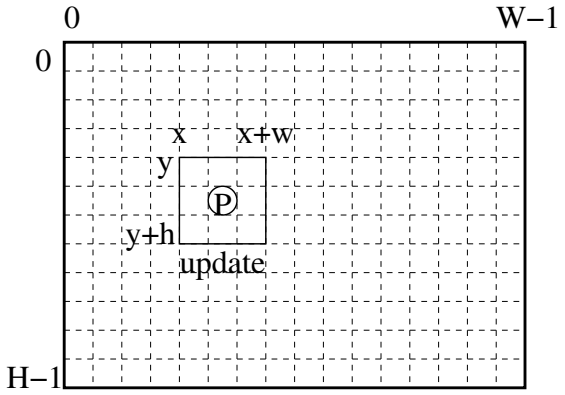

Figure 2: An example of a $W \times H$ board and a 3×3 update on it.

`public Point location()`

This method returns a *Point* object containing the (x, y) coordinates of the top left corner of the chunk.

`public Dimension size()`

This method returns a *Dimension* object containing the size (w, h) of the chunk—in Figure 2, the chunk is of size 3×3.

`public byte value( int x, int y )`

This method is used to access the values of the squares in the chunk. Note that the coordinates are relative to the board, not the chunk, so the invocation

```
byte b = u.value( x + 1, y + 1 );
```

would yield the value of the square in the center of the chunk depicted in Figure 2. Think of the update as beam of light falling onto a portion of the board. This portion is visible but the coordinates are still relative to the entire board.

Useful Tip The player will typically be in the center of the chunk. The exception is when the player is at the entrance or exit of the maze. After performing the initial nilpotent move, scan the update to locate the initial location of the player.

2 The Programming Task

The task is to implement a game player, called *MyGamePlayer* that plays and solves the Maze game. The game player must be implemented in Java and have the same functionality as the sample game player, specifically:

1. The program takes four command-line arguments: *Host*, *Port*, *Game*, and *Password*. For example, the game player can be invoked using the command:

   ```
   java MyGamePlayer localhost 4242 Maze coolness
   ```

   The program should perform basic error checking, e.g., too few parameters. If an error occurs, the program should report it and abort.
2. The program connects to the game server, instantiates a game, using the parameters specified on the command-line. If the instantiation fails, the program should report an error and abort.
3. The program should then play and solve the Maze game. It is strongly recommended that the program sleep for 50 milliseconds between moves to allow the game viewer time to keep up. To put the program to sleep for 50 milliseconds invoke the statement

   ```
   Thread.currentThread().sleep( 50 );
   ```

   before sending the next move.
4. The program should try to solve the maze in as few moves as possible.
5. The program should stop once the exit of the maze is reached.

Additionally, the program should be designed in an object oriented manner. Consider what will happen if more objects are added to the maze, or the rules change.

3 Grading Scheme

Weekly Project Reports (5%): First report due Thursday Sept 6, due each Thursday thereafter

The weekly project status reports are a way for your ten and your TA and instructor to make sure that your team is on track and that individual team members are contributing as expected. If your weekly meeting minutes include all the below information, they may serve as your project status report. Otherwise, produce a separate document which contains:

1. Overall team issues or concerns
2. Development status (tasks completed, in progress)
3. Planning status (new plans made, old plans adjusted)
4. Quality reports (results of reviews, testing, etc.)
5. Any other significant items

Also you need to include reports from each team member which contains:

1. Progress against previous goals
2. Hours worked
3. Any changes to the plan
4. Goals for the upcoming week

Milestone 1 (5%): Due September 6, 11:59pm Set up your groups website and development tools.

1. The Group website should contain on its front page at least the following information:
   - Group Name
   - List of group members (may include each member's picture and a short bio)
   - A brief project definition
   - Link to the meeting minutes and weekly project reports
2. Set up Subversion clients. http://tortoisesvn.net/downloads.html is said to be a good SVN client for windows. http://svnbook.red-bean.com/ is a good resource for learning to use Subversion.
3. Set up Eclipse and import the jar file provided. Run the program to see how the game server, game client and game viewer work.

Milestone 2 (5%): Due September 13, 11:59pm

1. Implement a simple player so that the player is able to move about the maze. You may use a simple algorithm like the right-hand rule for traversing the maze. You may ignore the obstacles at this stage.
2. Use Eclipse Junit testing framework and provide test cases to test for your current code.
3. Provide an initial design of the final version of your game player. Use UML diagrams to illustrate your design.

Final submission (10%): Due September 23, 11:59pm

Along with your source code, you need submit a team report (in pdf format) that describes the overall structure and operation of the program. The report must be well-formatted and include the following sections:

1. Cover page (document name, project title, team member, department and university, date, etc)
2. Project overview
3. Design details (Discussion, design decision, description of classes and diagrams.)
4. Test (Approach, feature tested, tools and environment)
5. Reference

The description should include

1. what classes the program consists of,
2. the purpose of each class,
3. how these classes interact, and
4. justification for the program's design and organization.
5. what are the important abstractions in the program.
6. what are the hierarchies in the program design? (Draw a diagram of each hierarchy.)

You also need to include in your report a full UML class diagram showing the class design and an interaction diagram showing the dynamic behavior of all objects in your program.

To submit your work, you must upload your project code and report to your SVN repository. Each member also need to submit a peer evaluation form by email to our TA by the due day. (Include your team number and name at the subject line of the email.)

The project grading scheme is included on the next page.

Implementation (8 pts)	Exceptional	Acceptable	Substandard	Unacceptable
Functionality (40%)	The program meets and exceeds functionality requirements in an exceptional manner. The program solves the maze and uses a faster (typically) non-oblivious algorithm.	The program meets functionality requirements. The program solves the maze using a simple oblivious algorithm like the "Right Hand Rule".	The program meets some functionality requirements. The program moves the player through the maze, but does not always solve it.	The program meets few or no functionality requirements. The program does not move the player through the maze or the program crashes with out doing anything.
Design (25%)	The program is exceptionally well designed, with an appropriate correspondence between entities and classes. The organization is intuitive. The program is easily extendible.	The program is competently designed. There is a reasonable correspondence between entities and classes. The program is understandable but not easy to extend.	The program is decomposed into classes that bear some correspondence to the entities. An attempt has been made at an object oriented design.	No attempt has been made at object oriented design.
Code (20%) organization, commenting, readability, etc	Code is very well organized, commented, readable, and maintainable. Code looks professional.	Code is competently organized, commented and understandable. Requires minor tuning to bring it up to professional looking code.	Code is poorly organized, not commented, or confusing. Requires significant effort to bring it up to professional looking code.	Code is unintelligible.
Test (15%) organization, commenting, readability, use of unit testing	Test code is very well organized, readable, and maintainable. Code looks professional. Demonstrate excellent usage of Junit testing environment.	Test code is competently organized, commented and understandable. Requires minor tuning to bring it up to professional looking code. Demonstrate adaquate usage of Junit testing environment.	Test code is poorly organized, not commented, or confusing. Requires significant effort to bring it up to professional looking code. Show very limited usage of Junit testing environment	Test code is unintelligible.
Report (2 pts)	Exceptional	Acceptable	Substandard	Unacceptable
Content (40%)	Addresses all points. Shows superior understanding of the issues.	Addresses most of the points. Shows some understanding of the issues.	Addresses some of the points. Shows partial familiarity with the issues.	Addresses few of the points. Shows little or no familiarity with the issues.
Analysis (30%)	Justifies all design decisions. Discusses criteria used in justifications. Considers the issues from multiple points of view.	Justifies most design decisions. Identifies criteria used in justifications.	Justifies some design decisions. Identifies some criteria in the analysis.	Little or no justification and no identification of the criteria used.
Presentation (30%) grammar, spelling, citations, headings, paragraphs, figures, and tables	Always uses standard conventions. The document looks professional.	Mostly uses standard conventions. The document could use some editing.	Does not consistently use standard conventions. The document requires significant editing.	Standard conventions are flouted. Document is unreadable.