Object-Oriented Software Engineering

Requirements for Hare and Hounds

The purpose of this assignment is to refresh your Java coding skills and to provide you with some basic knowledge of how to program web services in Java.

For this assignment we will be building a web-based variant of the classic game hare and hounds. This is a two player board game where the hounds try to trap a hare as it attempts to escape beyond them.

The figure below depicts the board on which the game is played, set to its starting position

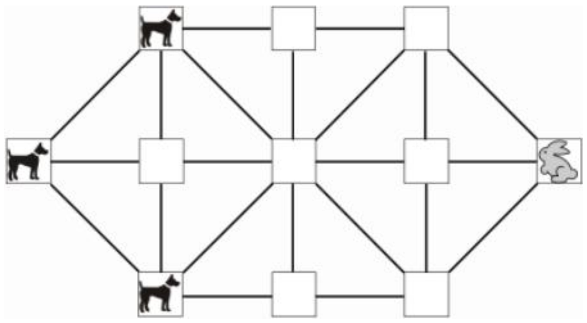


Figure 1: The board and coordinates

Here is a summary of the the rules:

- The game is turn based with the hound moving first.
- The player with the turn can only move one of his pieces per turn. A piece can only move one step at a time and it can only move to an empty location connected to its current location.
- Hounds cannot move backwards while the hare has no such restriction.
- The turns continue until one of the following (win) conditions occur:
  - The hare is trapped such that it has no valid move. The hounds win in this case.
  - The hare manages to sneak past the hounds. i.e. it moves to a square such that there are no hounds to left of it. In this case the hare wins.
  - The same board position occurs three times over the course of the game. In this case the hounds are considered to be stalling and the hare wins.

Here is a link to an online version of the game that you can play. Note that this does not implement the last winning condition. (TODO: Maybe we shouldn't have this)

The Assignment

Your task for this assignment will be to build a RESTful web server backend for the game using the Spark-Java micro web framework. You will integrate this with our front-end implementation located here. The front-end code will both display the board and manage user input, and will invoke your backend API. In other words, we are providing the view and controller, and you are to write the model of the web app.

Your game backend must be capable of hosting multiple games at the same time. However for simplicity we do not require that your server persist games across restarts. In other words you do not have to use a database to hold all your game data. You can do it all in memory. (Extra Credit: Make the games persist across server restarts using a sql database backend. Approximately 5 extra points)

We also don't officially require your code to have automated tests. However you are strongly encouraged to write a few. Extra Credit: We are offering upto another 5 points of extra credit for automated tests for your code. These tests must be acceptance/integration tests at a level similar to those in the Todo app; i.e. they test the overall functionality of the game backend. The number of points you get would be contingent upon the coverage and quality of your tests.

One of the key lessons of this assignment is programming to an interface. The split between model and view/controller is the point where that interface is defined in this application. We give a clear specification of that interface below that your backend must obey.

Your final "product" should be a maven-buildable standalone application, similar to our sample Todo app, that when started will make the game playable via a web browser pointed to http://localhost:8080/.

Backend Specification

General

- In the following specification, `<foo>` indicates a placeholder. When sending responses, you should send an actual value of the appropriate type (which will be indicated in the specification). When receiving data, you can expect some actual value to be present. (See also the note on error handling)
- Positions on the board are indicated by an `x, y` coordinate pair. The picture below shows the numbering scheme.

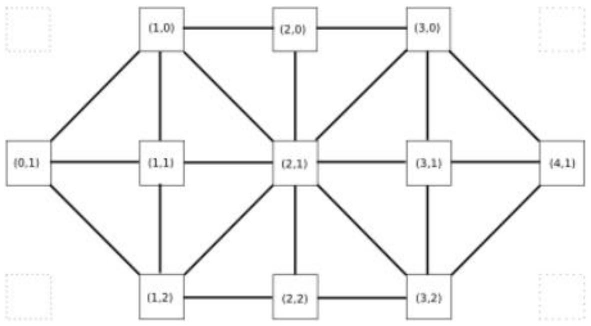
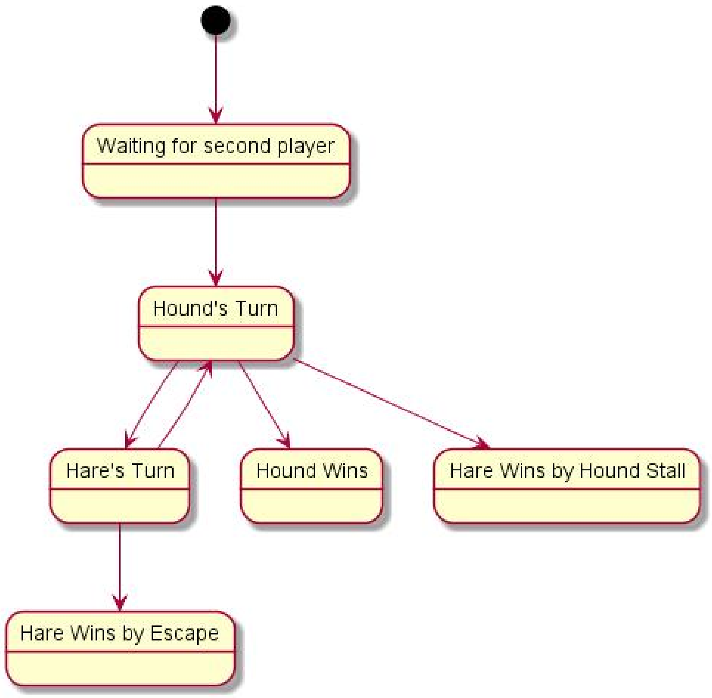


Figure 2: The board and coordinates

- The flow of a single game after creation can be described by the state diagram below. Keep it in mind while reading the specification:



The Web Api

A note on error handling:

- In general it is good practice to always return a standard error payload as the content of your response when there is an error (in addition to setting the HTTP status code). But for this assignment you are not required to do so except in the cases explicitly specified below. It is a point to keep in mind for your course project.
- Sometimes the request you receive is itself malformed. For example, the request to create a new game is missing the `pieceType` (see the specs below). In such cases, you should respond with a HTTP status code of `400` (which means "Bad Request"). In addition, each endpoint's description specifies responses for various semantic errors.

Start a game

Method	POST
URL	`/hareandhounds/api/games`
Content	`{ pieceType: <type> }`

Success Response	Code: 201
	Content: `{ gameId: <gameId>, playerId: <playerId>, pieceType: <type> }`

`pieceType` will be a string with a value of `HARE` or `HOUND`. `gameId` and `playerId` should both be strings. The identifiers can be in any format. It is required that `gameId` is unique across all games created by this server and `playerId` should be unique within the context of a single game.

Join a game

Method	PUT
URL	`/hareandhounds/api/games/<gameId>`
Success Response	Code: 200
	Content: `{ gameId: <id>, playerId: <id>, pieceType: <type> }`
Failure Response	
Invalid game id	Code: 404
Second player already joined	Code: 410

The `gameId` should refer to a previously created game; if it does not it is an error. It is also possible that another player has already joined the game by the time this request is received. You must respond with status code `410` to indicate that the game is no longer available.

On success the implementation must return a new player id for the player as well as the type of pieces the new player gets to play. As before the `playerId` should be a string which is unique within the context of this specific game and `pieceType` must be one of `HARE` or `HOUND`.

Play a game

Method	POST
URL	`/hareandhounds/api/games/<gameId>/turns`
Content	`{ playerId: <id>, fromX: <x>, fromY: <y>, toX: <x>, toY: <y> }`
Success Response	Code: 200
	Content: `{ playerId: <id> }`
Failure Response	
Invalid game id	Code: 404
	Content: `{ reason: "INVALID_GAME_ID" }`
Invalid player id	Code: 404
	Content: `{ reason: "INVALID_PLAYER_ID" }`
Incorrect turn (i.e. the turn is not this player's)	Code: 422
	Content: `{ reason: "INCORRECT_TURN" }`
Illegal move	Code: 422
	Content: `{ reason: "ILLEGAL_MOVE" }`

Once again if `gameId` and `playerId` do not refer to a previously created game and a player in that game, it is an error. To prevent player's from attempting to move pieces out-of-turn, you must check whether the specified player has the turn and respond with an error if not. Finally you must ensure that the move being made is legal as per the rules of the game.

Observe that this particular endpoint is required to return a reason for the error in response body.

Descibe the game board

The board in this game is fixed and the number of pieces are constant. So it is simply described in JSON as a list of four objects: three representing the hounds and one for the hare. Each object has the following structure: `{ pieceType: <type>, x: <x>, y: <y> }`. As usual `pieceType` is a string - either `HARE` or `HOUND`. Both `x` and `y` are integers.

Method	GET

URL	`/hareandhounds/api/games/<gameId>/board`
Success Response	Code: 200
	Content: a board description like above
Failure Response	
Invalid game id	Code: 404

Describe the game state

The current game state is described by one of the following strings: `WAITING_FOR_SECOND_PLAYER`, `TURN_HARE`, `TURN_HOUND`, `WIN_HARE_BY_ESCAPE`, `WIN_HARE_BY_STALLING`, `WIN_HOUND`.

Method	GET
URL	`/hareandhounds/api/games/<gameId>/state`
Success Response	Code: 200
	Content: `{ state: <current state> }`
Failure Response	
Invalid game id	Code: 404

Implementation Instructions

- All code you write must go under the Java package `com.oose2015.<your jhed id>.hareandhounds`. Feel free to create sub-packages under this if you feel the need.
- Your `main` method must be inside a class `com.oose2015.<your jhed id>.hareandhounds.Bootstrap`
- Your code must work against Java SDK version 8 and have a fully functioning Maven build file (Maven phases compile and package in particular must properly function)
- A good way to get going is to copy the structure of the Todo app, remove the application specific Java and front-end code from `src/main/java` and `src/main/resources/public` respectively and update the latter with the front-end code from the github repository. You can then put your own Java package under `src/main/java`. We strongly recommend this approach. That said, if it builds in Maven, uses our front-end without modification, has code in the right package, and runs on http://localhost:8080, you can use any app structure you want.

Submission Instructions

Assignments will be submitted via Blackboard. The assignment submission should upload an archive containing your work that is a fully encapsulated Java project with a Maven build file; we should be able to unzip this file to a directory and run The Maven package phase, `mvn package` if the command line is used, to build a fully functioning jar for your application. If your application is not properly packaged you will get a 0 and be asked to resubmit.

Grading Criteria

Score Calculation

The assignment is graded 75% on correctness and 25% on style.

Style

Your work throughout this course will be graded in accordance with industry development standards and practices. If you have no experience with industry practices (and most students will not), do not worry; the grading of the first assignment is quite lenient in this regard and you will receive feedback indicating what you need to improve. That said, this assignment is a good opportunity to develop the right habits. Except for the most egregious offenses (naming a variable with a dollar sign or a complete lack of commentation, for example), a warning is typically given to a student or group and points are only deducted if the next assignment does not improve. The following will be considered style requirements for the course.

- All code must be properly commented. All methods and classes must contain at least some Javadoc. Large methods with complex operations should feature inline comments. (Implicit use of `{@inheritDoc}` for inherited members is permissible.)
- All classes should be declared within packages, such as `com.oose2015.<your jhed id>.hareandhounds`. Use of the default package is unacceptable in any large software project.
- All identifiers must meet Java naming convention standards. Highlights of these standards appear below. All non-final fields and local variables should be named in lower camel case. Examples of lower camel case include: `helloWorld`, `variableName`, `testOneTwoThree`. All types should be named in upper camel case. Examples of upper camel case include: `ArrayList`, `MyVeryOwnClass`, `HtmlReader`. All final fields should be named in underscore-separated caps. Examples include: `SINGLETON`, `SPECIAL_VALUE`, `LOCAL_TAX_RATE`. No identifier should contain the $ character under any circumstances.
- Give methods and variables meaningful names. The definition of a meaningful name is subjective, of course, but it should

be obvious that `connectionMapping` is a more descriptive name than `temp42`. Variables should also adhere to Java naming conventions (classes in upper camel case, etc.).

- Empty catch blocks are bad practice. If a method throws an exception, it is trying to tell you something. Simply ignoring it is just as bad as ignoring a method’s return value or any other information at your disposal. If the exception is truly meaningless to you, at least throw some form of runtime exception wrapping it and include a comment indicating that the code cannot be reached for whatever logical reason.
- Program to interfaces, not to classes. For variables, always use the least specific interface which supports your needs. For instance, use `List` instead of `ArrayList`.
- Use generics when appropriate. For instance, declaring a variable of type `List<String>` is better practice than declaring a variable of type `List`. Using generic type variables drastically reduces the amount of time you spend debugging dyanmic type errors.

Again, remember that these are all very leniently approached in the first assignment. Style is an important part of project work; it makes your code more readable and more maintainable. With that said, you should definitely choose functionality if your time constraints force you to pick between the two. If you have any difficulty whatsoever with understanding or implementing these or any other requirements, feel free to contact the teaching assistant or instructor.