Escape Introduction

Every offering of CS4233 presents techniques for designing and implementing systems using primarily an object-oriented approach. This requires the use of the following skills and techniques:

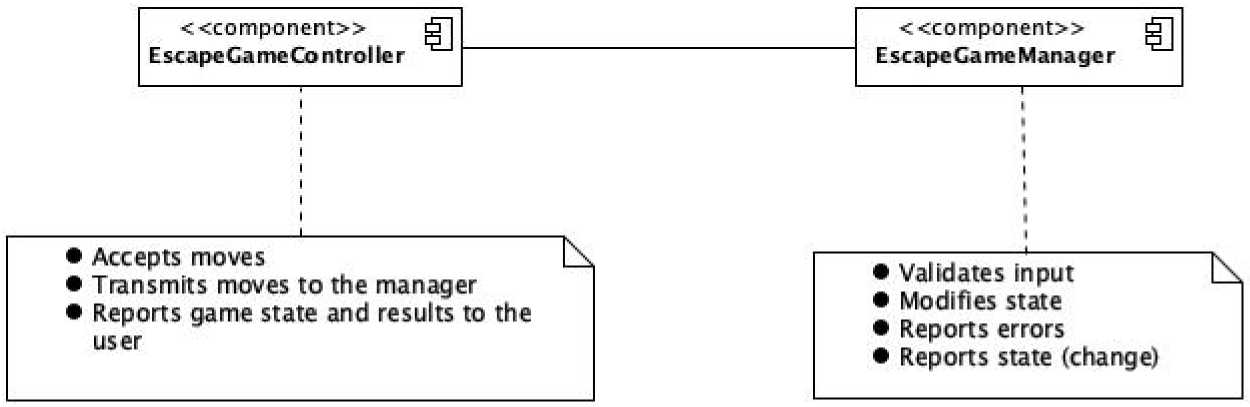
- developing an object model from requirements
- identifying relationships between objects
- assigning behavior to classes that supports loose coupling and high cohesion
- applying established design principles and patterns
- thorough testing
- evolutionary design

Finding a non-trivial project that can be completed in a seven week course is challenging. Experience indicates that games, especially board games, offer a reasonable blend of interest, complexity, and variety for students to exhibit competence in the above skills and techniques.

The game for this term is Escape, a game designed specifically for this course. Escape has the following features:

- variable board dimensions
- variable board location shapes (e.g. squares, hexes, etc.)
- rule variations such as game length, victory conditions, and so on
- different types of pieces with various attributes

An ideal scenario for this course is one where a completely playable game with user interface and human-player interaction results from the project. At this time, the necessary infrastructure does not exist. Therefore, students are responsible for developing a game manager that initializes a game variation, maintains the game state, accepts moves in the appropriate order from an external player or controller, validates the move, reports on the results of moves, and determines victory situations. The following image shows the two components and how they interact.



If you have not already encountered incomplete program specifications, then you have not experienced real-world software development. Ambiguity and incomplete specifications are a fact of life in most situations, especially in the rapid or continual release cycles that we encounter today. Just as you design and develop software iteratively, refining your design and implementation in each iteration, requirements also evolve. Even small software projects evolve and requirements change due to changes in business environment or incomplete or ambiguous requirements. Prior to iterative development methods companies and government agencies expended tremendous efforts to attempt to get exact requirement specifications written before development started---almost all of these efforts failed.

If I tried to get every facet of Escape described completely and unambiguously, it would require much more for you to read, and I would still miss some things. This means that you, as the software developer, have a responsibility to ask questions when you are not sure of the meaning of a statement. If you find errors you need to make me and the rest of the class aware of these (Slack is the best way).

Escape: Game Mechanics

Escape was designed specifically for this course. It is a game that offers rich variability to which students may apply several design patterns and principles in implementing the game software. As the name suggests, the goal of Escape is for players to have their pieces "escape" from the board while preventing opponents' pieces from escaping. Depending upon the game variation, there are different means of escaping, determining the value of escaped pieces, and so on. This section describes the basic game mechanics. Each course offering has different requirements for implementing game variations. Each variation has some combinations of the features described in these notebooks. Each game begins with a set of pieces, some set of properties assigned to each piece type, a board consisting of some type of tiles or shape, and rules that guide the game play. It also includes the initial board configuration, and victory conditions.

Game play

The game play for Escape is simple. Two or more players take turns moving one or more of their pieces each turn until one player achieves victory conditions or the game results in a draw.

Before the game begins, players place all of their pieces on the game board. How the pieces are placed and the board configuration occur outside of the game manager software that you will develop. A game configuration file specifies the initial setup to your game manager (see the notebook on configuring games). Once the game configuration is established, play begins.

The game is played in turns. A "turn" is when a player makes one or more moves, depending upon the rules specified in the game configuration.[1] When all players have moved, this is a "round." Games may be limited to a specific number of rounds.

Move sequence

A "move" consists of the following steps.[2]

1. Game controller (the client) calls the `move()` method in the game manager (your component). Parameters are:
   - Source location of the moving piece
   - Destination location of the moving piece.
2. The game manager verifies that the move is legal. Legal depends upon the universal rules (e.g. no move if the game is over) and the specific rules in the game configuration. If the move is illegal, the game manager responds with an error response. In this case, the state of the game does not change and the game controller can submit another command for the moving player. As of this writing, however, the following **always** represent an illegal move:
   - There is no piece on the source location.
   - The piece on the source location, but it does not belong to the moving player.
   - There is a piece on the destination location, but it belongs to the moving player.
   - The source and target locations represent the same location on the board (i.e. the piece is not moving).
3. The game manager makes the move, updating the state of the game; such as updating the locations of pieces on the board, removing pieces, resolving engagements, identifying a winner, and so on. This prepares the game for the next move.
4. The game manager returns a move response object to the game controller.

✓ You do not have to worry about the locations passed in being `null` or off the board. That is the responsibility of the game controller, which is outside of your project.

---

[1] If the game configuration does not specify a number of moves per turn, the default is one per turn for each player.

[2] Depending upon the game variation in effect, these may change somewhat.

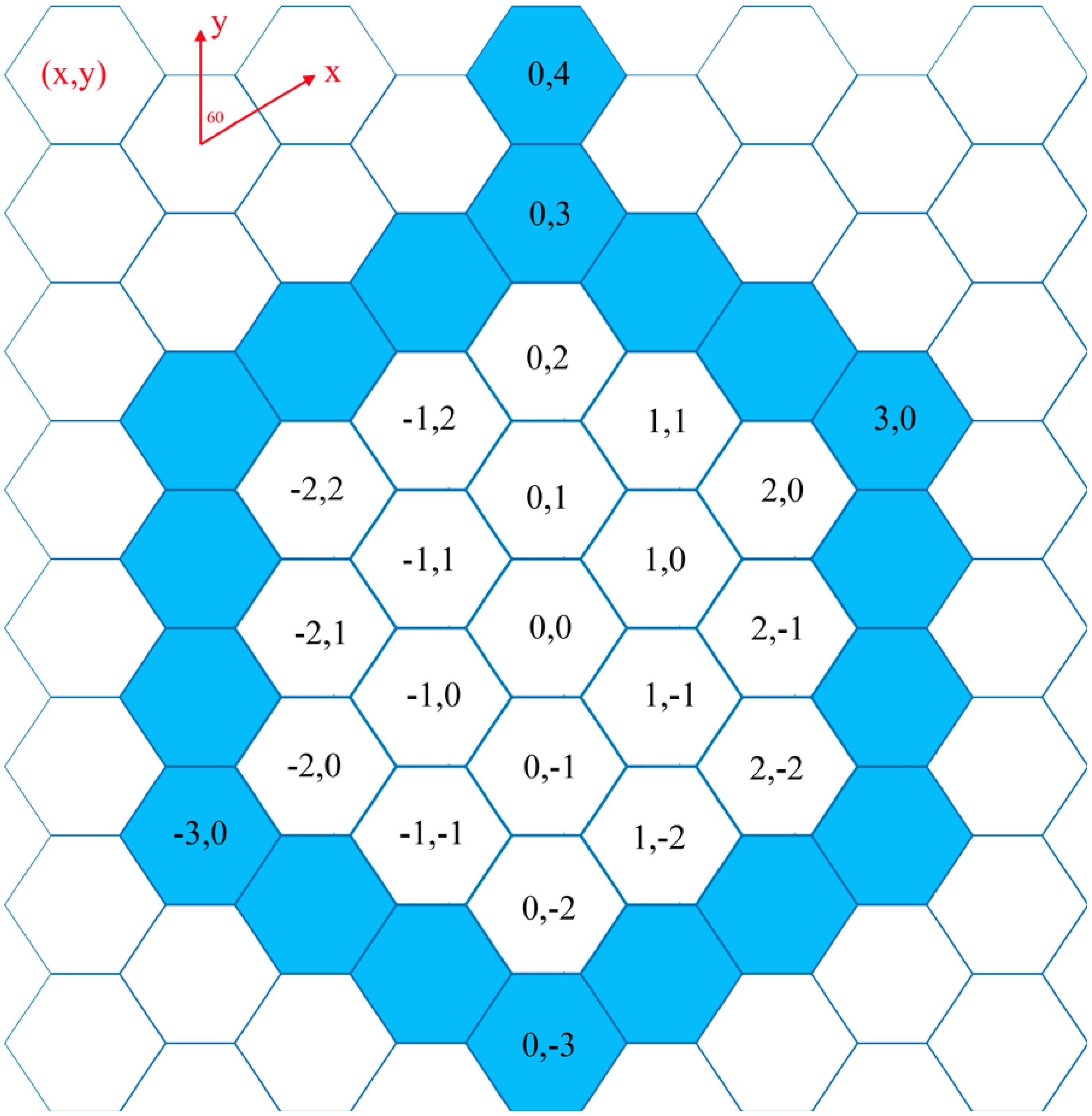
Escape: Game Boards

Escape is played on a board that consists of locations. Most boards are 2-dimensional; however in some offerings of the course you may see boards that are not. The locations can have different shapes: square, hexagon, and so on. Each location is referenced by a coordinate that identifies. Coordinates are discussed in detail in the notebook on coordinates.

The following figures show the most common 2D boards:

8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8
7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8
6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
2,1	2,2	2,3	2.4	2,5	2,6	2,7	2,8
1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8

Square Board



Hexagonal Board

In the current offering we are only dealing with 2D boards so the documentation from here on assumes that there are just two values describing the coordinate (x, y).

The locations also have an attribute that indicates their type. There are three types of locations:

- `CLEAR`: This is the default for any location. It indicates that this is a plain location with no special characteristics.
- `EXIT`: This is an exit location. Score is accumulated by moving pieces onto an `EXIT` location where they exit the board. The player's score is adjusted by the rules specified in the game configuration. A

piece *may not move over an `EXIT` location* unless they are able to fly over it, or jump over it. Any attempt to cross an `EXIT` by a piece without `FLY` or `JUMP` capability simply falls through the `EXIT` and is removed from the game.

- `BLOCK`: This type of location does not allow a piece to land on it. A piece may not cross over a `BLOCK` unless they have either `FLY` or `UNBLOCK` capabilities. A piece may not `JUMP` over a block.

Boards may be finite or infinite. Locations are identified by their coordinates, which are expressed at a pair of integers. The two integers are called x and y but do not necessarily correspond to the x-y coordinate scheme for the standard Cartesian grid (i.e., the x-component refers to the horizontal axis and the y-component refers to the vertical axis). The square board shown above has squares that are represented by coordinates of the form (row, column); opposite of the standard representation in geometry.[1]

✓ Finite boards always have the coordinate (1, 1) located in lower left location from a canonical viewpoint of the player that moves first. If there are two players seated at opposite sides of the board, one player would have square (1, 1) at their lower left and the other would have (1, 1) at their upper right location on the board. **Finite boards always have coordinate components that are positive integers**.

Infinite boards may have coordinate components that are any integral value. There is a reference location at (0, 0) and all other locations are relative to that (see the hexagonal board above). Notice the way the locations are related to each other in the boards. This is critical in order to determine the distance and direction from one location to another and finding paths between them.[2]

---

[1] If you think a little bit about whether x corresponds to a row or a column and y corresponds to the other , you will realize that it really does not matter.

[2] One thing I've observed in the many times we have used Escape as the term project is that students get hung up on trying visualize the boards. It is best to think of them more abstractly and not worry about visual cues unless there is some specific reason (usually there isn't).

Escape: Coordinates

A coordinate is more than just a pair of integers. Coordinates are types that have properties and behavior. How much behavior a coordinate has is one of the many design decisions that you will make as you design and implement your project. The game controller only sees the Coordinate as an opaque object. You may extend the `Coordinate` interface and/or implement one or more classes that implement the interface. The `Coordinate` interface has no required methods; it is a *marker interface*.

One of the decisions that you will need to make early on is how many classes you need, or want, in order to define coordinates and boards.

There are two types of coordinates in the basic board types described in the previous notebook.

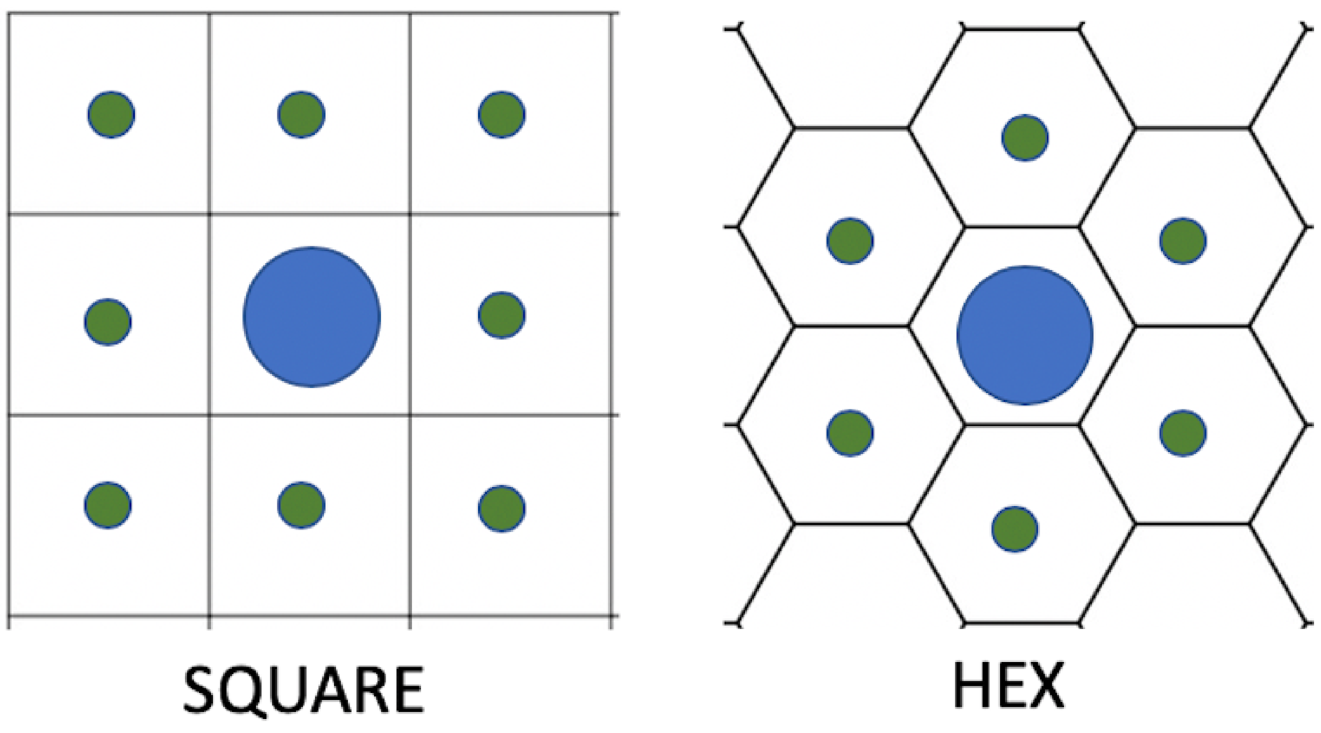
1. **SQUARE**: A coordinate that is used on square boards. The coordinate represents squares that may allow movement in any one of eight directions, horizontally, vertically, and diagonally. Certain game

variations may restrict the movement of certain piece types to a subset of the eight possibilities.

2. **HEX**: A coordinate that has six neighboring coordinates to which a piece may move. Boards with this type of coordinate do not recognize the concepts of orthogonality or diagonality.

The starting code base contains an enumeration, `CoordinateID` that has constants for each of the coordinate types you might encounter.

When pieces move in any game they move along paths that consist of a sequence of individual moves from one location to one of its neighbors. Each coordinate type has a specific set of neighbors. The figure below illustrates the neighbor sets for each basic coordinate type. The large blue circle represents the piece and the smaller green circles represent the neighbors.



Neighbors

You will also find out that you need to compute other properties, for example the distance between two coordinates. As you develop your solution, you will incrementally add what you need, based upon your design.

Escape: Game Pieces

The Escape pieces have a piece type. The type has a name, movement pattern, and attributes that determine the piece's capabilities for a specific game variation. The actual pieces do not have names, just types. From an object-oriented point of view, their identity would the the `Object` identity of their memory address. This identity would rarely be used. When a piece needs to be obtained or examined, it is accessed by using the piece's location on the board. For example, a client can ask for the piece at location (5, 4).

In every variation of Escape, the configuration defines each piece type and the capabilities of the types that are used in that variation So, for example in one variation he `HORSE` might be able to ¸gallop from one end of the board to the other, and in a different variation, it might make moves only like the Knight piece in standard chess. In Escape, piece types are given the names of animals, but any label would do.

Pieces have an arbitrary number of attributes that define their capabilities for a specific game version. For example, in one version pieces might have a numeric value that is used in calculating how much that piece adds to the score when it escapes from the board. Other attributes might include movement limitations, special abilities, and so on. The attributes work in conjunction with game variation rules to determine the game play.

The following sections describe the different movement patterns and attributes. Each game variation implements some or all of these. For simplicity, each of these is defined by enumeration constants in the `EscapePiece` interface in the starting code base.

Movement patterns

Movement patterns describe how a piece type may move during the game play. For example, in the game of checkers, the pieces may only move diagonally; in chess the Rook moves orthogonally, and so on. The pattern is just that — a pattern of movement. There is nothing in the pattern that indicates any minimum or maximum number of spaces, or any other attributes of the piece. **Every piece type must have exactly one movement pattern associated with it**. Each movement pattern has an enumeration constant in the `MovementPatternID` enumeration in the `EscapePiece` interface in the starting code base.

During a game, pieces move according to the movement pattern specified by the movement pattern attribute of their piece type, and any other constraints or advantages that might be specified for that type. In general, a piece moves along a path that is a sequence of individual "steps." Each step involves a move from a location to one of it's neighbors. The movement pattern identifies which subset of the neighbors a piece may move to.

The movement patterns and their meanings for Escape are:

- `ORTHOGONAL`: Movement is only allowed along the horizontal or vertical axis (or the z-axis if a Cube was used for the board). This pattern is not valid for boards with `HEX` or other coordinates that may be added in the future that do not admit orthogonality.
- `DIAGONAL`: Movement is only allowed along the diagonal in boards with `SQUARE` coordinate types. It is not valid to use this with a board that has other types of coordinates (i.e. `HEX`).
- `LINEAR`: Movement must be in a straight line. For square boards, this means either orthogonally or diagonally. For hex boards, it means in a line along one of the six directions of the sides.
- `OMNI`: Piece may move to any location (not its own location though) using any path that is not restricted by other movement rules or attributes. The path must consist of individual steps from one tile to neighbors of that tile as defined by the coordinate type.

The following table summarizes the valid combinations of coordinate types for game versions and the movement patterns that are allowed for those coordinate types. `DIAGONAL` and `ORTHOGONAL` have been abbreviated to adhere to the page margins.

	ORTHOGONAL	DIAGONAL	LINEAR	OMNI
SQUARE	YES	YES	YES	YES
HEX	NO	NO	YES	YES

Allowable movement patterns for coordinate types

✓ Every piece type **must** have a movement pattern. There are no defaults.

Piece type attributes

Each Escape piece type has a set of attributes associated with it during a game. All units with the same type have exactly the same attributes when a game starts. There are two types of attributes, *boolean attributes* and *valued attributes*. Boolean attributes are described only by name. If the attribute is defined, then it has a `true` value; otherwise it is `false`. Valued attributes have an associated integer value. The attributes vary by piece type and game variation. Attributes *must be explicitly specified* in the game configuration file for any game variation unless a default is specified.

The currently defined attributes are shown in this table along with their type and a brief description.

Attribute	Type	Brief description
VALUE	valued	The points associated with the piece
DISTANCE	valued	The maximum distance that this piece may move
FLY	boolean	This piece may fly over any location regardless of the location type or if it has a piece on it
JUMP	boolean	This piece may jump over pieces or exits, but not obstacles (`BLOCK` locations)
UNBLOCK	boolean	This piece may cross over a BLOCK location

✓ Every piece type must have the `DISTANCE` attribute defined.

The following subsections in this notebook describe the attributes in more detail.

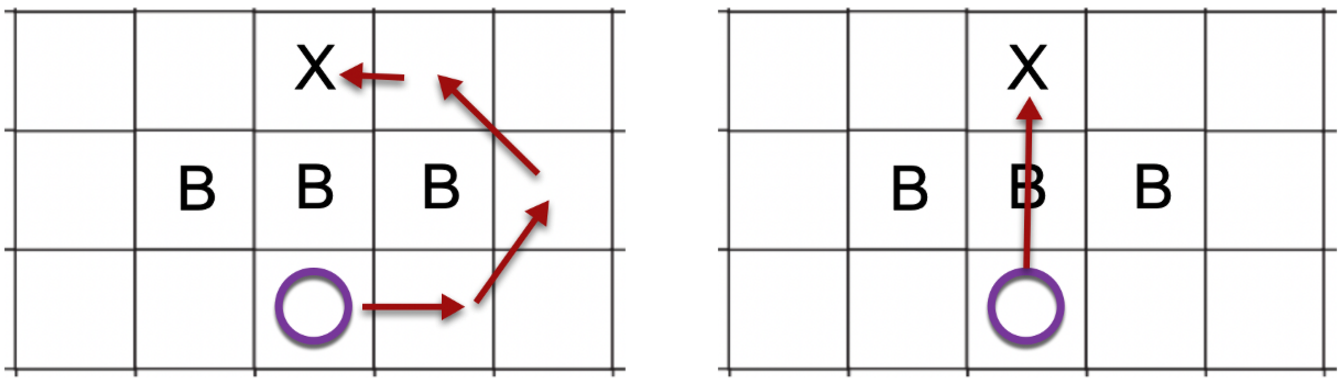
The VALUE attribute

This is a valued attribute. It is a positive number that indicates the points a piece is worth when the game begins. This is used in calculating the score when pieces escape from the board. This value may change, depending upon the rule set selected for a game. For example, a piece might gain value points if it defeats an opponent's piece, or lose value points if it loses an encounter with an opponent.

NOTE: If this attribute is not present, then the default value for a starting piece is 1.

The DISTANCE attribute

The valued attribute defines the maximum distance that a piece with this piece type may travel during a turn. This distance is the maximum distance of a **legal path** from the piece’s starting point to ending point of a move. A legal path is the number of individual moves that a piece makes from the starting to ending location. The following diagram illustrates the difference between a path and the distance between two coordinates in a `SQUARE` board and a piece that has `OMNI` movement pattern. Each board/movement pattern combination might influence what a legal path might be.



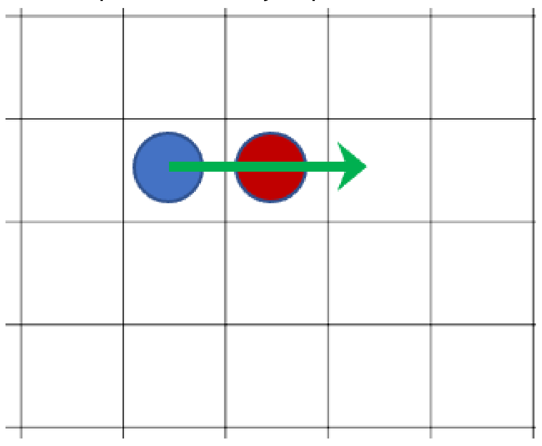
SQUARE path distance = 4 SQUARE coordinate distance = 2

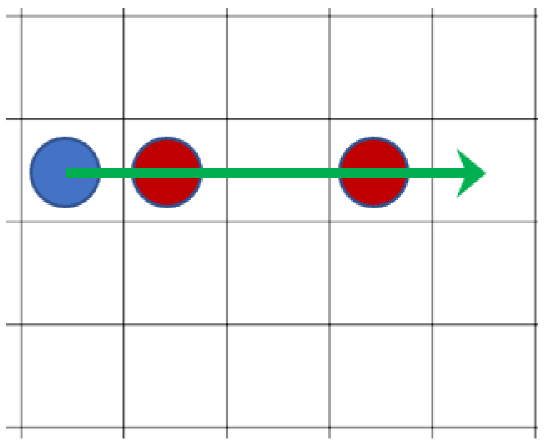
The FLY attribute

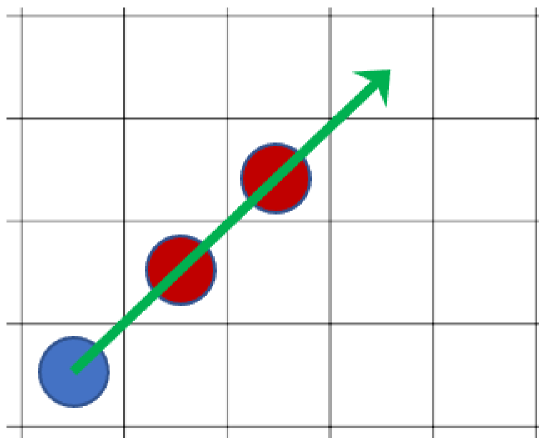
Like the `DISTANCE` attribute, `FLY` is a value attribute. Any unit that has a `FLY` attribute can simply move from it starting location to the end point, as long as no other restrictions prevent it from doing so, without worrying about `BLOCK` or `EXIT` locations or pieces in the way. A piece with this attribute may move from the starting location to the ending location, as long as there is a *valid path that satisfies the movement pattern for the piece* that is less than or equal to the integer value of the `DISTANCE` attribute for the piece type. For example, if the piece has a `ORTHOGONAL` movement pattern, the legal path must include only horizontal or vertical steps and the ≤ the value of the `DISTANCE` attribute.

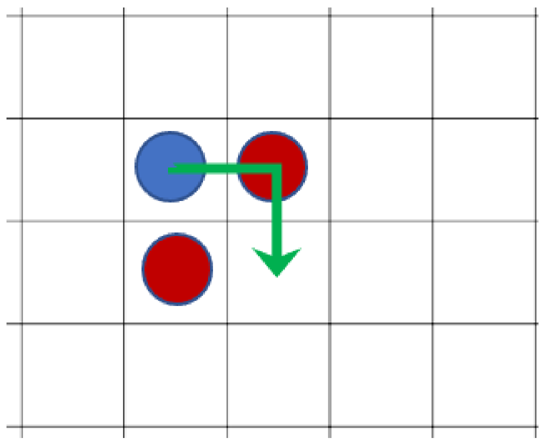
The JUMP attribute

This is a simple attribute. A piece with this attribute can jump over other pieces or exits that may be on the path from the piece's starting point to ending point of a move. A piece with `JUMP` may jump over just one piece or exit at a time. If, for instance the desired path has two other pieces in consecutive locations along the path, then the `JUMP` would be invalid. If, however, two or more pieces are in the path and there is at least one empty location between pieces, then the `JUMP` is valid. The piece may not change directions during the jump. The figure below shows examples of valid and invalid jumps. The blue circle is the moving piece and the red circles represent other pieces (regardless of which player owns them). A piece with `JUMP` **may not** jump over a `BLOCK` location. It may jump over a piece and land on an `EXIT` location. Jumping over a piece takes two locations along the path. So, if a piece only has one location left when it attempts to jump, it cannot perform that jump.


Valid


Valid


Invalid


Invalid

The UNBLOCK attribute

A piece with this attribute may move across locations that have the BLOCK attribute. The piece may **not** end the move on the BLOCK location, but otherwise, the BLOCK location is just like a CLEAR location for the piece with this attribute.

Escape: Game Rules

Each Escape variant has a set of rules that affect the game play. These rules describe game characteristics such as the number of turns, how to determine the score, victory conditions, rules of engagement, and so on. This section describes the predefined game rules and how they apply to games. Not all of these may apply for specific variants. For your current course offering, there may be additional rules in place. If so, they will be defined in the assignment or variation documentation.

Each rule has a label and an optional integer value. The rule labels are in the `Rule` interface in the starting code. The next subsections describe the rules in detail.

TURN_LIMIT

This rule indicates that the game ends after some finite number of turns (one turn consists of each player making a move) have been made. When the number of turns indicated by the turn limit have been made, the game ends. The winner is the player that has removed the most points from the board through exit locations, Other rules may influence game play in conjunction with this rule, but a game with this rule will never last longer than the number of terns specified by this rule.

SCORE

If this rule is present, then the winner is the player that first removes the number of points from the board. The starting number of points for any piece are determined by the `VALUE` piece attribute.

POINT_CONFLICT

This indicates that when a player moves a piece onto a location containing an opposing player's piece, the piece with the lower current point value is removed and the other piece remains on the square. The remaining piece has its value reduced by the value of the removed piece. For example, if a `FROG` worth 5 points lands on a location with a `SNAIL` worth 2 points, then the `SNAIL` is removed and the `FROG` remains on the location with a value of 3 points. The two points for the removed `SNAIL` are not counted when determining the winner. If both pieces have the same value, then both are removed from the board and neither is counted in determining the winner.

Example: Assume a `HORSE` with a value of 10 encounters a stack of a `SNAIL` worth 1 point and two `FROG`s worth 5 points each, then the `HORSE` is removed and the `SNAIL` and one `FROG` is removed (total of 6 points) and the second `FROG` has 4 points removed from its value, leaving it with a `VALUE` of 1 point.

If there are two pieces with the same lowest value when reduction needs to be made, the pieces are chosen by sorting them by piece type alphabetically and taking the first one (e.g. `SNAIL` and `HORSE` would reduce the `HORSE`).

Several constraints on the game rules and situations that may arise during a game that are not explicitly described in the game rules. This list attempts to describe those that have appeared in previous terms and describe the way to handle these situations. Some of these describe actions that the game manager implementation must adhere to.

1. If `TURN_LIMIT` and `SCORE` are present, then the game ends when *either* one team achieves the score condition of removing points equal to or greater than the number specified in the `SCORE` rule, *or* the turn limit has been reached.

2. In all game versions, your game should keep track of which player is, and should be making the move. Players move in the order they are defined in the configuration file.

3. If a player cannot make a move in the current game state and has not met the victory conditions, then the opposing player wins. The opposing player wins, even if the opposing player could not make a

valid move in the current game state.

4. All other things being equal, if both players have removed the same number of points and an end-of-game condition has been met, then the game is considered a draw.

5. If an invalid move is attempted, the moving player loses the game.

Move results are returned to the client in a `MoveResult` object and additional information is conveyed, if appropriate through messages that are pushed to an observer, if that option has been implemented.

Escape: Configuring Games

One can imagine many variations of the Escape game. This suggests that each game variation must be individually composed. In order to do this, we use configuration files to describe the boards, pieces, and other game characteristics that the student-developed software will use to implement the game manager. A configuration file is a structured file. In previous terms we have used XML files to describe the game configuration. While XML is readable, it is quite easy to make a mistake. XML only describes structure. Other technologies like XML Schemas are necessary to ensure that an XML file is correct. Creating the XML Schema file can be quite complicated.

In order to simplify the creation of the XML configuration file and reduce the possibility of errors, I created a small language that is simpler than XML, designed to allow you to create configuration files more easily. There is an EscapeConfigurator tool that you can use to convert a simpler Escape Game Configuration file (.ecg) into an XML configuration file. The tool checks for consistency where possible and reports any errors it finds. The appendices contain the information about how to write the simple files and run the tool. The tool is also provided as a library in the starting code so that you may invoke it from tests or other classes. You give it the name of the .ecg file and it returns the appropriate XML file as a String. If there are errors, they appear on `stderr` and an exception is thrown.

The following listing shows a very simple example of an Escape Game Configuration file (the extension of such files is `.egc`. Notice that in many cases, you can use uppercase or lowercase names. This listing does not use them consistently. You should try to be consistent when writing your own configurations.

```
Coordinate type : SQUARE
xMax : 25
yMax : 20

Locations :
    (3, 5) block
    (4, 4) clear player1 snail
    (5, 12) exit

Piece descriptors :
    SNAIL omni [distance 1]
    DOG linear [distance 5]
    HORSE diagonal [distance 7, jump]
    bird linear [fly, distance 5]
    frog omni [distance 3, unblock]
```

```
Rules :
    SCORE 10
    TURN_LIMIT 20
```

When this file is input to the *EscapeConfigurator* tool it converts the configuration into an XML representation that is much more complex and tedious to create. The configurator tool also checks for several possible errors, but not all of them yet.

The `EscapeConfigurator.jar` is in the starting Escape code base. You can copy it to any directory and execute it from a terminal. If you just run the command:

```
java -jar <path to directory/EGC.jar -h
```

You will see this output:

```
Usage: EscapeConfigurator [ options ] [ sourcefiles ]
Options:
    -d <output directory>
    -s Display results on stdout
    -h Display this message
```

While you do not need the JAR file in your Escape project, you might want to keep it there so that you can generate appropriate XML files for your tests.

What do you do with the XML representation of a game configuration? There is a class in the starting code base, `EscapeGameBuilder` that takes the XML file and creates an instance of the `EscapeGameInitializer` class that is also in the starting code base. This class is basically a data structure that contains predefined objects created from the game description file.

You will implement the `makeGameManager()` method in the builder by using the data in the initializer to build a specific `EscapeGameManager` implementation. This is illustrated in the notebooks on the starting code base.

Structure of a configuration file

In general, a configuration file has the following structure:

```
    Coordinate type : <CoordinateID>
    xMax : <integer>
    yMax : <integer>

    Locations :
    <(x, y) other-attributes>
    ...
    Piece descriptors :
    <PieceName MovementPattern attributes>
    ...

    Rules :
    <RuleName optional-value>
    ...
```

The full grammar used by the parser can be found in the *Escape configuration tool* project in my public repository.

Below I've tried to make a simple regular expression grammar that describes the structure of the file without going into the details of context-free grammars and ANTLR.

- '*' means zero or more occurrences
- '+' means one or more occurrences
- '?' means zero or one occurrence
- parentheses are used to group items
- '[' and ']' indicate ranges
- '|' indicates alternatives
- A word in all uppercase letters is a lexical element (i.e. a word or symbol). I have not included the lexical grammar. You should be able to figure out that something like "COORDINATE" might be written as "Coordinate" or "coordinate." Also INTEGER and LETTER would have the meaning you would expect.
- Literals are enclosed in single quotes like this `':'`.

```

configuration = coordinateType dimensions
        (locationInitializers | pieceDescriptors | rules)*;

coordinateType = COORDINATE TYPE? ':' coordinateName;

coordinateName = HEX | SQUARE | TRIANGLE ;

dimensions = dimension+ ;  // accepts any number, but if > 2 it will give
an error

dimension = ( 'xMax' | 'yMax) ':' INTEGER ;

locationInitializers = LOCATIONS ':' location+ ;

location = coordinate locationType? piece?;
```

```
coordinate = '(' coord ',' coord ')' ;

coord = '-'? INTEGER ;

locationType = BLOCK | CLEAR | EXIT ;

pieceDescriptors
:
    PIECE DESCRIPTORS ':' pieceDescriptor+ ;

pieceDescriptor =
:
    pieceName movementPattern attributes
;

pieceName = LETTER+ ;   // Any word, but right now only BIRD | DOG | FROG
| HORSE | SNAIL

movementPattern = DIAGONAL | LINEAR | OMNI | ORTHOGONAL | SKEW  ;

attributes = '[' attribute (',' attribute)* ']' ;

attribute = attributeName INTEGER? ;

attributeName = DISTANCE | FLY | JUMP | STACK | UNBLOCK | VALUE ;

rules = RULES ':' (ruleName INTEGER?)+ ;

ruleName = DISTANCE | FLY | JUMP | STACK | UNBLOCK | VALUE ;
```

Some things to remember about the configuration:

- There must be a coordinate type and (right now) two coordinate axes. The tool will accept one, but you should have two.
- The axis value must be non-negative. **A value of 0 means the axis is infinite.** That is, that any integer value will do for that axis.
- After the board's dimensions are entered, the `locationInitializers`, `pieceDescriptors`, and `rules` can be entered in any order and are optional. Also, you can have multiple sections of these in your configuration file. A better practice might be to format the configuration as in the example above.
- You don't have to put each element on separate lines. Whitespace is ignored.

**Also note: ** The file format and the grammar used may actually be a little different, specifically in the values in the categories due to changes in the game specification for your course offering.