Escape: a Family of Games for CS4233 Developer's Guide

Gary F. Pollice

Version Alpha.1, April 11, 2020

Abstract

Escape is a board game, actually a family of board games, designed for term projects in CS4233: Object-Oriented Analysis & Design. The game has several possible variations that allow the instructor to choose an appropriate subset of the rules for implementation during any given term. Variations are usually played by two players. Each player starts with a set of pieces on the board. Each piece has a point value and specific movement and battle characteristics. The player who exits the most points off the board by the end of the game wins. Variations modify the rules in several ways such as the number and type of pieces, valid exit squares, different movement and battle rules, and so on. During any given term, students will typically implement software to manage a game according to one or more variations.

This manual provides the description of the game, the rules, and helpful hints for developers who will produce the software implementations of the game.

Contents

1 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 has shown 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. Figure 1 shows the two components and how they interact.

2 Game Mechanics

Escape was designed specifically for this course. It is a game that admits a rich amount of variability to which students may apply several design patterns and principles in implementing the game. As the name suggests, the goal Escape is for players to have their pieces "escape" from the board while preventing opponents' pieces from escaping. Depending upon the game variation, there

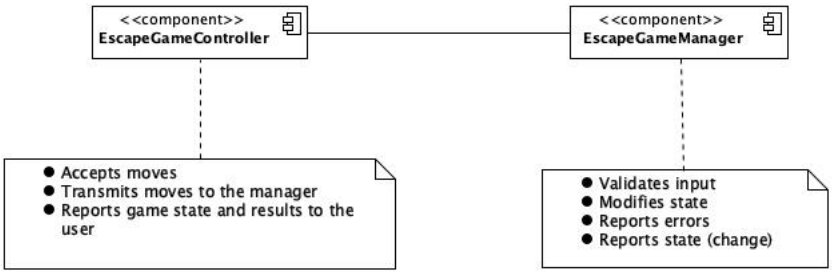

Figure 1: **Controller (external) and Manager.**

are different means of escaping, determining the value of escaped pieces, and so on. This section describes the basic game mechanics that apply to all games. Specific variations are described in other sections or in separate documents.

Escape is played on a board that consists of locations (we will sometimes them "squares" for simplicity). The locations can have different shapes, such as square, cube, hexagon, triangle, and so on. There are ways to combine boards and locations that significantly change the game play.

The Escape pieces vary among variations. Each piece has a number of attributes, or properties, that determine the piece's capabilities during any single game. Each game begins with a set of pieces, some set of properties assigned to each piece, and rules that guide the game play, including the initial configuration and victory conditions.

2.1 Basic game boards

The most common two game boards are a board of squares (Figure 2) and a board of hexagons (Figure 3). These boards may be finite or infinite; square boards are usually finite and hex boards may be either. For this game, the square board (Figure 2) has squares that are represented by coordinates of the form (*row*, *square*). The coordinate (1, 1) represents the square in the lower left location from a canonical viewpoint. 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. In Escape, the square boards are always finite.

Hexagonal boards are quite different than square boards in the way the hexes are referenced. Regardless of whether a board is finite or infinite, there is one *origin hex* that is denoted as (0, 0) as shown in Figure **??**. All other hexes are represented by coordinates with x and y components that are relational to (0, 0).

Any Escape game versions that you might implement in this course will have boards that are one of the basic game boards. There may be differences in the way the boards are laid out, their dimensions, how distance might be calculated, and the properties of each square or hex on the board (i.e., whether a piece may travel through the location, escape if it lands on it, and so on).

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	18

Figure 2: **Square board, orientation, and notation.**

2.2 Game pieces

Every variation of Escape uses named pieces that move, escape, and possibly form other actions such as engaging opponents. The names are simply labels used to distinguish one type of piece from another. The pieces are named after animals, but any label such as `Piece1, Piece2`, etc. would work just as well. In one game, the `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.

Pieces also have an arbitrary number of attributes that contribute to the game play. 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, strength when engaging opponents, and so on. The attributes work in conjunction with variation rules to determine the game play.

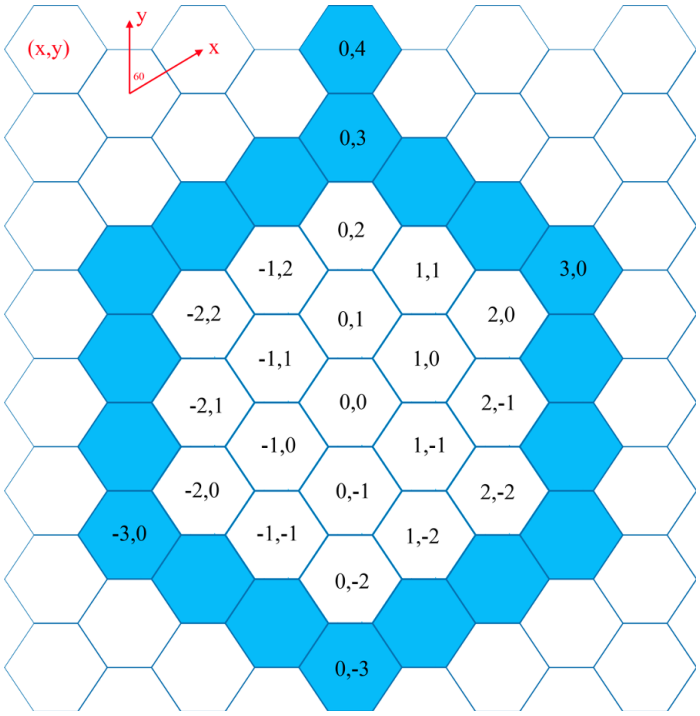

Figure 3: **Hexagon board, orientation, and notation.**

2.3 Game rules

Each Escape variant has a unique set of rules that determine the game play. There are different types of rules such as

- *Movement rules* that define movement patterns (e.g., orthogonal only), whether a piece is able to jump over another piece, if a piece must rest for a specific number of turns, and so on.
- *Engagement rules* if the game supports the sides engaging in some form of combat between pieces; the rules of engagement and calculations to determine the outcome of an engagement are defined by rules for the specific variation.
- *Initialization rules* specify any constraints or requirements on how the game may be setup before play begins.
- *Victory or end of game rules* describe how the game might end and how to determine a winner.

2.4 Specifying game variations

From the previous sections, one can imagine practically any number of variations of the Escape game are possible. This suggests that each game variation must be individually composed. In order to do this, we will use configuration files to describe the boards, pieces, and other game characteristics that the student-developed software will use to compose the game. Rules will be described textually. Students will determine how to implement them, organize them into appropriate sets for the variations, and represent them in the software. The configuration file will only provide the game version identifier. Section 3.4 describes these files in more detail.

3 Starting code base

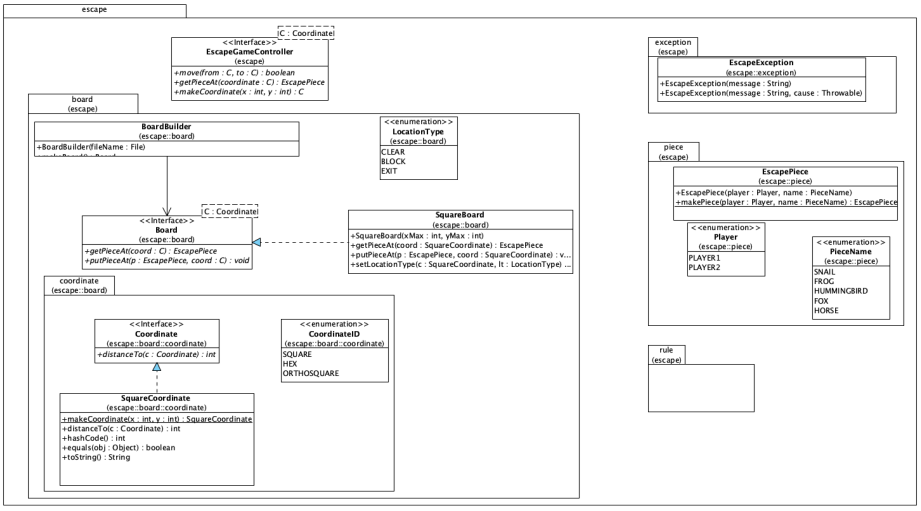

Figure 4: General structure of the starting code.

The starting code is set up for the first deliverable. The overall layout is shown in Figure 4. The code has been divided into several packages. The packages contain Java classes and interfaces to satisfy specific behavioral components. The code is skeletal and the student is expected to modify it according to their particular design and implementation. Table 1 describes the packages and their intended purpose.

Table 1: Starting code packages.

Package	Contents
`escape`	This is the root package for the classes that implement the Escape game. Classes and interfaces in this package implement the complete game controller and provide the public interface to the game to any clients.
`escape.board`	All of the classes and interfaces that pertain to the board are in this package or its subpackages.
`escape.board.coordinate`	All classes that implement the different coordinate types are located in this package. It is a subpackage to *eclipse.board* since the coordinates reference locations on the board.
`escape.piece`	All classes that are necessary for implementing the game pieces are in this package. For the starting code base, this package has only what is necessary for a minimal piece implementation.
`escape.rule`	This is an empty package in the starting code. It will be populated for future deliverables. There is a nackage.info file in the starting code. It may be removed at any time.
`escape.exception`	This package contains a single EscapeException.java file that implements a runtime exception. It is there for whatever use a student may desire for catching internal errors and error messages that can be displayed.
`util`	This package contains utility classes and example code. The utility classes are used for initializing the game and its components.

The starting code is just that, a beginning. Some of the code is exemplar; that is, it is one way of doing things. It is not necessarily the best way of doing things, and certainly not the only reasonable way of satisfying the requirements. Students should not simply accept the code and continue developing in the style it is written. My expectations are that you will take what you think is appropriate and fits your design and re-implement the other things, as long as they do not break the contracts set up. The contracts are interfaces that describe expected behavior; they may be actual Java interfaces or methods that concrete classes must implement. I have tried to keep these to a minimum in order to give students the most freedom in developing a flexible, maintainable design and implementation. I have tried to identify the elements that must not be changed with comments in the code and in the descriptions in this specification. There

is always the possibility that some of my early decisions must be revisited and changes made. This is often a consequence of evolutionary development.

For this course, there are three or four deliverables. The initial plan is for three deliverables, each adding a major component, until the complete game manager is ready. The deliverables are

1. The first deliverable (alpha) will implement the different boards and coordinates that will be needed for any of the supported games. This release also will require the implementation of a *board initializer* that can instantiate the appropriate board and coordinate types. Students will `BoardBuilder` object that takes as input a board specification (XML) file to return an initialized board. Testing of this release will focus only on the board and coordinates.

2. The second deliverable (beta) implements the pieces and movement constraints on pieces. Students will implement a factory that can produce pieces. The factory will be built by a `PieceFactoryBuilder` object that takes as input a piece specification (XML) file to return the appropriate piece factory.

3. The third deliverable (delta) implements the complete game, with the rules and public interface for the client. In this case, the games are built by a `EscapeGameBuilder` class that take in a game specification (XML) file with all of the information necessary to create the game. This file contains the names of the the board specification and piece specification files. Any rules specific to the game will be constructed based upon the game ID, which is a unique label given to a game variation.

The following sections describe the contents of each package of the starting code base in some detail—enough to help orient the student on how to organize the code and work with the starting code.

3.1 `escape.board`

Most of the work for the *alpha* release will be done in this package and its subpackage(s). While a complete game framework for Escape would include support for N dimensions, we limit ourselves to two dimensions. The initial structure of the `escape.board` and the `escape.board.coordinate` packages is shown in Figure 5.

The main class in `escape.board` is the `Board` interface. Any type of board must implement this interface. There are only two methods that describe the behavior of all boards: `putPieceAt()` and `getPieceAt()`. Each of these has a piece and a coordinate as parameters. Coordinates can differ so the `Board` interface has a generic type, `C` which takes a specific type of coordinate. Even though there are only two required methods for any `Board` implementation, one would expect several other methods to be added by developers according to their design.

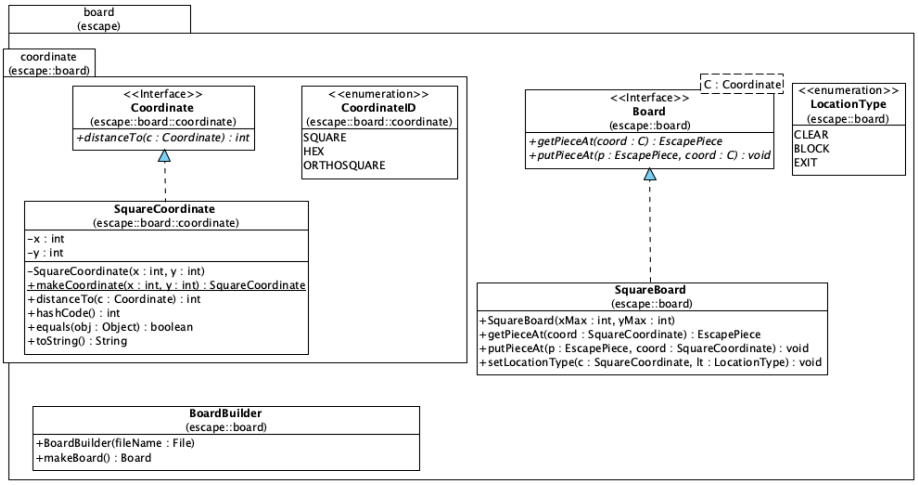

Figure 5: Initial `escape.board` and subpackages.

Any board is made up of locations. There are different types of locations in any game. In the starting code, three types of locations are identified in the `LocationType` interface. These are `CLEAR`, `BLOCK`, and `EXIT`.`CLEAR` locations are the normal location where a piece may reside or travel through. `BLOCK` locations are ones where a piece may not reside nor pass through. `EXIT` locations are those whee pieces may exit the board, in order to score points for the owning player.

A sample board implementation is provided in the `escape.board` package in the SquareBoard.java file. While not a complete implementation, it illustrates how a concrete `Board` implementation might look. The class implements locations and pieces that are on the board as two maps with coordinates as the key for each of them.

The third example file is BoardBuilder.java. This is a builder class that creates a SquareBoard and initializes it based upon information contained in a board configuration file. The information from the configuration file is marshaled into a `BoardInitializer` object. This object is a JavaBean. The example of creating the `BoardInitializer` from the configuration file is in the `BoardBuilder` constructor. **You will need to have a `BoardBuilder` class implementation for the first release.**

3.1.1 `escape.board.coordinate`

We currently only deal with 2-dimension coordinates. Adding a third dimension is not trivial, but should not be difficult; but that's for another term maybe. There is one key interface in the `escape.board.coordinate` package and that is `Coordinate`. Any coordinate implementation must implement this interface. There is only one method in the `Coordinate` interface, `distanceTo()`. This method calculates the distance from the `Coordinate` instance to another coor-

dinate. Read the javadoc documentation on the `Coordinate` interface and you will see that you must also implement a static `makeCoordinate()` method that takes an appropriate number of integers and creates an instance of the concrete coordinate. In addition, the javadoc specifies that you must override `equals()` and `hashcode`. The specification of implementing the required methods in any concrete implementation of `Coordinate` are there for some of the tests that we will run for early releases. The effort required to implement these methods should be quite minimal.

The other two files in the `escape.board.coordinate` package are an enumeration, CoordinateID.java, and an example coordinate implementation, SquareCoordinate.jave. The `coordinateID` has three elements that define three types of coordinates that we might use in the games versions that you will implement. Each of these is described in the documentation. The `CoordinateID` value will be in the board configuration file so that you know what type of coordinates the board must support. The `SquareCoordinate` class is just an example of what one of the implementations might look like.

3.2 `escape.util`

This package contains utility classes and examples that can be used to initialize the game and other components. Some of the classes might be moved to other packages in a final release, but for this class, we will keep them here. In the starting code base there is

- BoardInitializer.java which is used by the `BoardBuilder`.
- LocationInitializer.java that describes one location on a board for initialization purposes.
- InitializerTest.java which is an example of how to use the JAXB classes for binding XML files to Java classes.

3.3 Other packages and classes under `escape`

For the first release, most of the other code in the starting code base are not needed. These consist of the `escape.exception` package and the `escape.rule` package. There is nothing in the latter package and s single runtime exception that you can use as it stands or extend it for specific purposes. This is there as a helper and may be more useful in future releases.

There is one class in the `escape` package. This is EscapeGameManager.java. This is not used in the first release, but it defines the main interface for the final project release. The contents of this interface will most likely change from the initial code base before the second release.

3.4 Configuration files

Configuration files are XML files that describe the contents of a typical JavaBean. That is, it contains the initial data that gets bound to the instance

variables using the getter and setter methods of the bean. This is similar to dependency injection. Some frameworks, like the Spring Framework use configuration files as part of their dependency injection capabilities.

You can use the code in the InitializerTest.java file to see how to create a configuration file using the *marshaling* capabilities of JAXB and how to create a JavaBean instance. You can find a simple JAXB tutorial here.

A sample board initialization configuration file is boardConfig1.xml that is in the config/board folder in the starting Eclipse project. The file looks like this:

```
content<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<boardInitializer>
  <coordinateId>SQUARE</coordinateId>
  <locationInitializers>
    <x>2</x>
    <y>2</y>
    <locationType>CLEAR</locationType>
    <player>PLAYER1</player>
    <pieceName>HORSE</pieceName>
  </locationInitializers>
    <locationInitializers>
    <x>3</x>
    <y>5</y>
    <locationType>BLOCK</locationType>
  </locationInitializers>
  <xMax>8</xMax>
<yMax>8</yMax>
</boardInitializer>...
```

Notice that in the array of LocationInitializers, not all fields are specified in the second item. There is no `playr` or `pieceName`. When a field has no value, it is set to the Java default value for the class, which is `null` in these two cases.

Configuration files will be used extensively in the tests and final release.

4 Releases

This section contains the description of all of the releases that you will deliver for this project. Initially, only the Alpha release is specified. Additional releases will be specified in subsequent versions of this document.

4.1 Alpha release

The alpha release of Escape focuses on the various boards. In order to successfully implement this, you will need to do the following:

- Modify the `BoardBuilder` class so that it can create the desired type of board, with the correct type of coordinates, given a file name that is a path to a valid configuration file. See the BoardTest.java file in the test

directory for an example. The tests will create a new `BoardBuilder` and call the `makeBoard()` method. They will then exercise the appropriate behavior of each board.

- Implement different coordinate types for the three types of coordinates identified in `CoordinateID`. You must name these classes `SquareCoordinate`, `HexCoordinate`, and `OrthoSquareCoordinate` that each have a static `makeCoordinate()` method. This is necessary for my tests; although you may find it useful when constructing the board and performing other operations as the code evolves. You should be calculating the distance to another coordinate of the same type should be correct. The `SQUARE` and `ORTHOSQUARE` should be quite straightforward. Distance on a hex board is more complicated. There are several references for ways to do this. You can find them with a simple Web search. *Remember if you use someone's code for the algorithm, or just use their algorithm, you need to put a citation in your code as a comment. Failure to do so is plagiarism.*

For this release, there is nothing you have to do with the `EscapePiece` class. You will need to have a convenient way of creating pieces from the configuration file (e.g., from the information in the `pieceName` and `player` instance variables in the `locationInitializers`).