Hanto: A Game Of Strategy for CS4233

Developer's Guide

Gary Pollice

ABSTRACT

Hanto is a game of strategy based upon Hive®. The goal is to encircle your opponent's primary piece (Queen Bee or Butterfly). The game is notable for the ability to create new types of pieces and rules.

This manual provides helpful hints for developers who will produce software implementations of the game for CS4233: Object-Oriented Analysis and Design.

Last Modified: 14-Apr-2016

Table of Contents

Hanto: A Game Of Strategy

1. Introduction

Hanto is a game of strategy that is based strongly on the games of Hive®. Hive was developed by John Yianni and published in 2001 by Gen42 Games [1]. Hanto appeared around 2009 and is published by Agile Route [2]. Hanto is a superset of Hive and has a much more configurable set of variations and rules.

The game is simple. There are two players, Red and Blue. The goal of the game is to surround your opponent's lead piece (the Butterfly). While the game is played on a board consisting of hexagons, the board has no specific dimensions. The active board grows and shrinks as the game progresses. The current size of the board is a function of the number of pieces in the particular version of the game and how they are currently configured on the board.

Each variation has one or more types of pieces specified for the game. Each type of piece has its own characteristics that describe how it can move and its "powers."

2. Game Mechanics

This section describes the board and the notation used to identify tiles on the board. It then describes the pieces that are predefined in the game. Before continuing with this section read the Hanto rules. The material that follows assumes that you already have read those rules.[1]

2.1. The board

The board consists of a contiguous grid of hexagons. They are arranged in an infinite grid as shown in Figure 1. The indexing of hexes for purposes of your Hanto is as shown in that figure. A description of this indexing can be found on the VBForums™ Web site [3].

[1] Two sets of rules for Hanto are provided in the course resources. One set is printed directly from the game in iOS and the other is from a source that I found on an Internet search. These give you a good overview of the game and the rules. This document is the official word on rules and variations in this course.

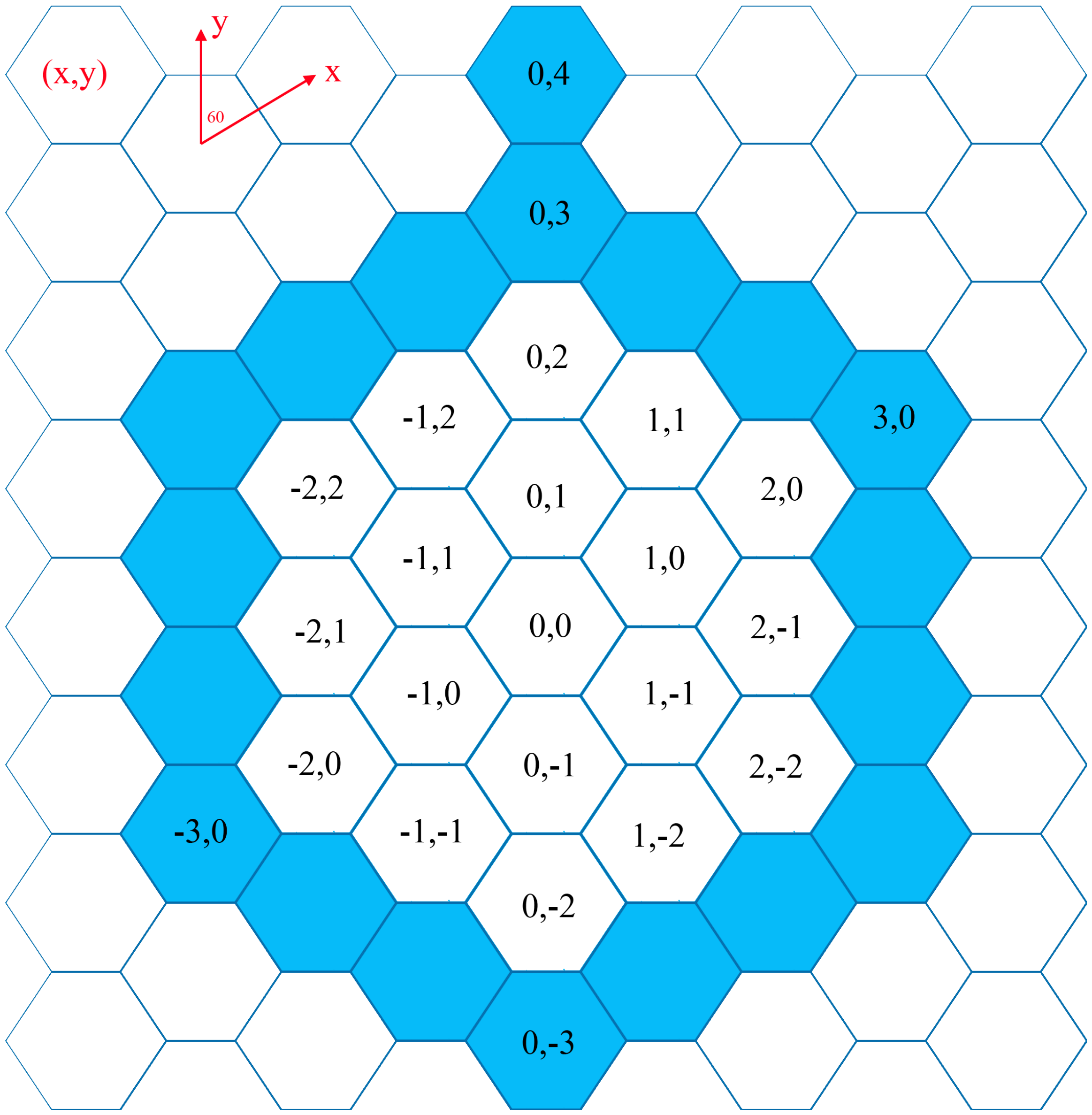


Figure 1. Hexagonal board orientation and notation [3].

2.2. The basic implementation

Students will implement a Hanto game that is capable of supporting several variations of Hanto. In a manner similar to the Backgammon game in *Flexible, Reliable Software Using Patterns and Agile Development* [4]. The variations will be added to this manual as they are defined.

The implementation of Hanto that students develop during the term will support playing a complete game, without any artificial intelligence (AI) player. For the final project deliverable however, the students will develop some sort of AI player that can play a legal game of the various versions of Hanto. The skill of the AI player is not a gradable item, only whether the player plays a legal game. The instructor will provide a Hanto tournament director application that will use student players and have them play against each other. This is explained further in this document.

For this class we have provided the student with a starting Eclipse project that should be used as the basis for code. The started project has mostly interfaces and enumerations. Figure 2 shows the relationship of those elements in the starter project.

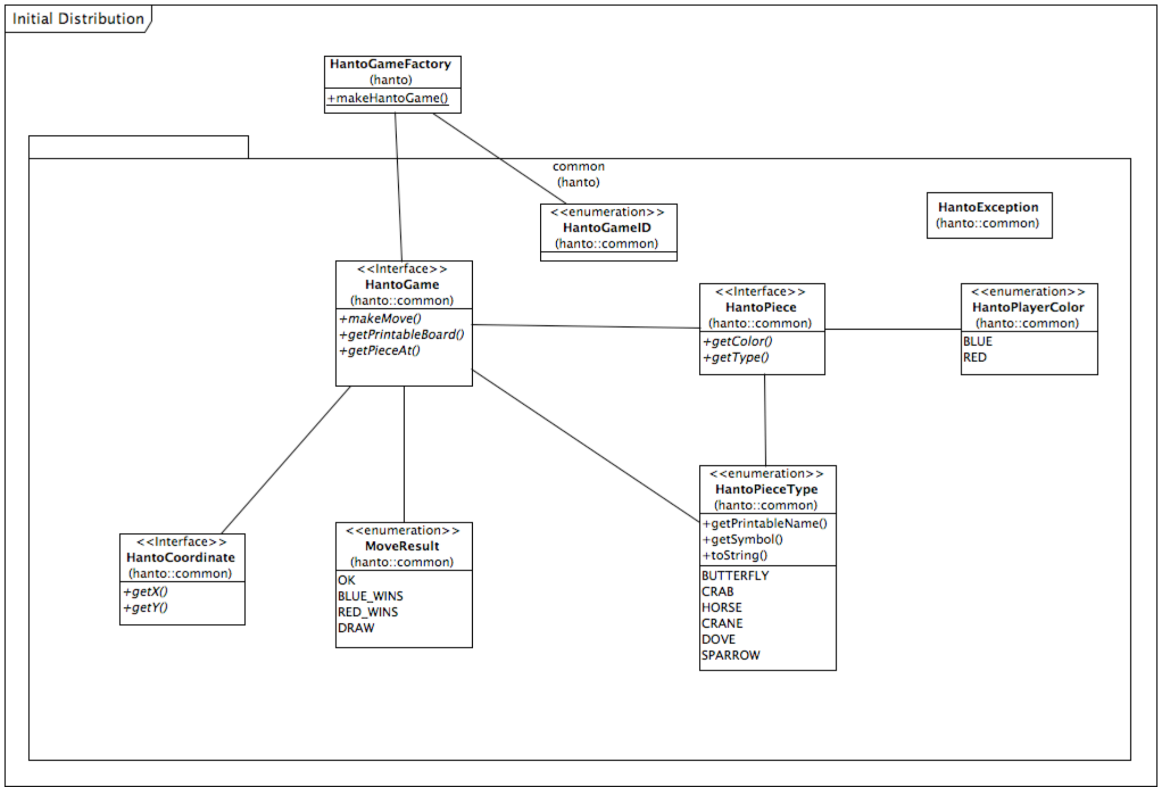


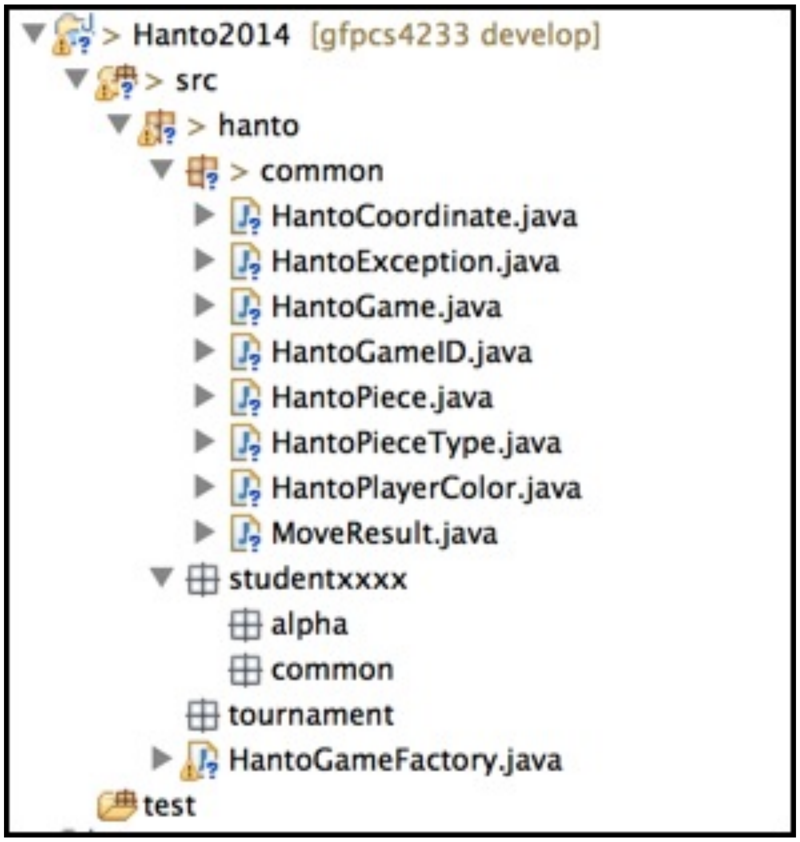
Figure 2. Elements in the Hanto starter project.

Student classes must always support the capabilities required as shown in Figure 2.

2.2.1. Package structure

All student code will be placed in packages that are rooted at a package that is unique to the student, or students. In Figure 3 this is shown as **hanto.studentxxxx**. If you are working on the project alone, you should rename this, replacing **xxxx** with your user name. So, if your user name is jj3927, you would rename the package to **hanto.studentjj3927**. If you are working on this with a partner, you should replace **xxxx** with some unique string that identifies both of the team members (perhaps three initials for each student). For purpose of this documentation, we will just refer to this package as **hanto.studentx**. All student code will be in subdirectories that are in either the **hanto.studentx** package or a package that is a descendent of that package.

Instructor supplied code and tests will always be in the other packages and never include code that is in the **hanto.studentx** package tree.



```
▼ > Hanto2014 [gfpcs4233 develop]
  ▼ > src
    ▼ > hanto
      ▼ > common
        ▶ HantoCoordinate.java
        ▶ HantoException.java
        ▶ HantoGame.java
        ▶ HantoGameID.java
        ▶ HantoPiece.java
        ▶ HantoPieceType.java
        ▶ HantoPlayerColor.java
        ▶ MoveResult.java
      ▼ studentxxxx
          alpha
          common
        tournament
      ▶ HantoGameFactory.java
  test
```

Figure 3. Hanto project package structure.

2.2.2. The initial source code

The initial code base for the Hanto game is shown in Figure 3. The purpose of each file is described here.

HantoException.java

Any exception that is thrown by student code must be a *HantoException* object. This exception class has only two constructors. The first takes just a message and the second takes a message and another *Throwable* (usually another type of *Exception*) object. Some of the required methods in other classes throw *HantoException* objects.

HantoGame.java

This interface describes the interface to a Hanto game controller. All of the logic that supports the playing of Hanto is hidden by this interface. The detailed code that implements this logic is in Java classes in the **hanto.studentx** package. Students will create this code.

The interface is quite simple. The main method is the *makeMove()* method. This is the method that gets called for every move of the game. The realization of this interface will control the game and ensure that a valid game is being played. Two other methods have been added that as convenience methods. They have no impact on the game, but can be used to print out a readable instance of the state of the game by showing the board in some String form, or querying a specific coordinate. The actual format of the board printout is not specified, but you are expected to make it something that anyone who understands the game of Hanto should be able to tell what the current game state is.

HantoCoordinate.java

This interface simply defines a two dimensional coordinate that allows you to get the x and y-coordinates. It has a limited set of capabilities that are a subset of the Java *Point* class.

Note: In the game, a coordinate that is *null* is used to represent a piece that is part of the game but is off the board.

HantoPieceType.java

This enumeration contains values for each piece used in the game. Additionally, there are methods for getting a printable name—such as “Dove” for the DOVE piece—for each enumerated value as well as a method that returns a single character representing that piece.

HantoPlayerColor.java

This is a simple enumeration with two values, BLUE and RED.

MoveResult.java

Another simple enumeration with four values, OK, BLUE_WINS, RED_WINS, and DRAW. One of these values is returned from the *HantoGame. MakeMove()* method as long as the move is valid.

HantoPiece.java

This interface represents a playing piece in the game. It is basically a two field data structure that contains an immutable type and color.

HantoGameID.java

This is an enumeration that has literals that define the versions of Hanto.

HantoGameFactory.java

This is a class that represents a singleton factory object. This is the object that should always be used when creating specific Hanto game instances.

3. General game characteristics and rules

This section describes general game characteristics and rules. Many of the rules and characteristics of the game are described in some detail in the official Hanto rules. The description, however, can be difficult to navigate and some of the finer points of movement capabilities is not always clear. We clarify some of the confusion and possible ambiguity in this section and simplify some of the rules.

3.1. Placing pieces on the board

Unless specified otherwise in a variation's rule set, when a piece is placed on the board it must be placed on an open hex (the destination) that is adjacent to at least one hex that is occupied with a piece the same color as the piece being placed. Additionally, there cannot be a hex adjacent to the destination containing a piece of the opposing player's color. This rule does not apply to the first placement of a piece on the board, which must be made at hex (0, 0).

3.2. Movement

Each piece type has one or more movement capabilities. The movement capabilities may vary from variation to variation. This section identifies each movement characteristic that we currently use in Hanto and provides the information needed to implement it in your Hanto application. This differs from the movement capabilities slightly as they are described in the official Hanto rules. We have simplified them and made them more explicit. Further, there are fewer capabilities than used in the Hanto game for the iPhone and iPod.

Note *that no piece may move until the player's Butterfly has been placed on the board. Until then, pieces may only be placed on the board.*

3.2.1. Walking

A piece that has walking capabilities is able to move from one hex to another by "sliding" it through a sequence of unoccupied hexes, beginning with the hex the piece occupies at the beginning of the movement and ending at the destination. There must be a path of unoccupied hexes from the start to the destination hex.

Sliding means just that. You must be able to slide the piece. The red X in Figure 4 is unable to move. Even though there are open hexes that would seem to allow one to drag the X to them, the piece occupies the complete hex on which it sits. Therefore, two hexes are needed to slide it out, as in Figure 5.

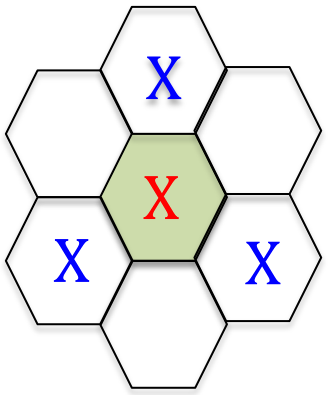


Figure 4. The red X cannot walk.

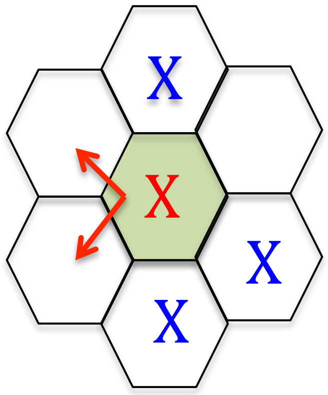


Figure 5. The red X can walk.

If a piece can walk, there must never be a non-connected configuration of pieces during the walk. If the piece can walk more than one hex, the pieces on the board must be connected as one group at each step as the piece walks. This is shown in Figure 6. If the red X is able to walk two or fewer hexes, then it cannot walk to the designated hex because when it takes the first step onto the yellow hex, there would be a disconnected configuration.

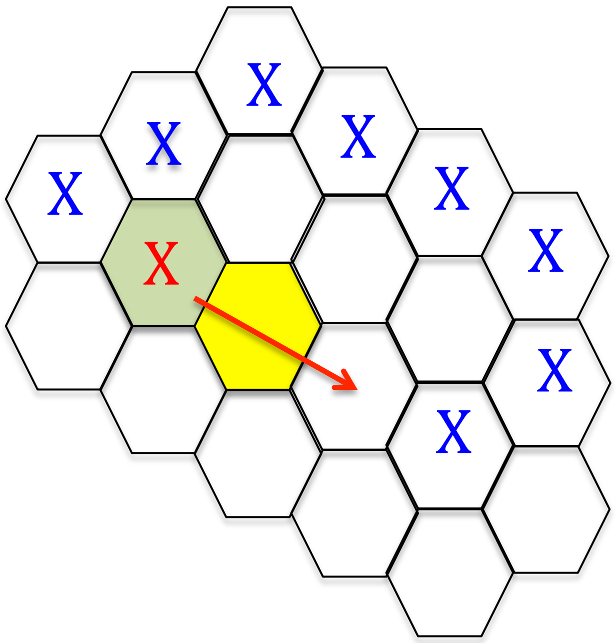


Figure 6. The red X cannot move as shown.

Walking pieces may have a limit to how many hexes they may travel in a turn.

3.2.2. Flying

Pieces that fly simply move from a source hex to a destination hex. They can fly over occupied pieces. During the flight, we do not worry about whether the pieces that are on the board are connected or not (if the move is valid, they will, be connected at the end of the move). The piece that is flying is not considered until it lands on the destination hex. The destination hex must be, unless otherwise specified, unoccupied. Flying pieces often have a maximum number of hexes they can move in a turn.

3.2.3. Running

Running is similar to walking. There are two differences:

1. The piece has a range. This is the number of hexes that it <u>must</u> move during the move.
2. The piece cannot move onto the same hex during a turn's movement. That is, if the red X in Figure 7has a range of 3, it cannot move to the indicated blue hex by first moving onto the blue hex, then back to the starting hex, and then back to the blue hex. While the group connectivity would be maintained during the movement, the piece moves onto the blue hex twice (the first step and the final step).

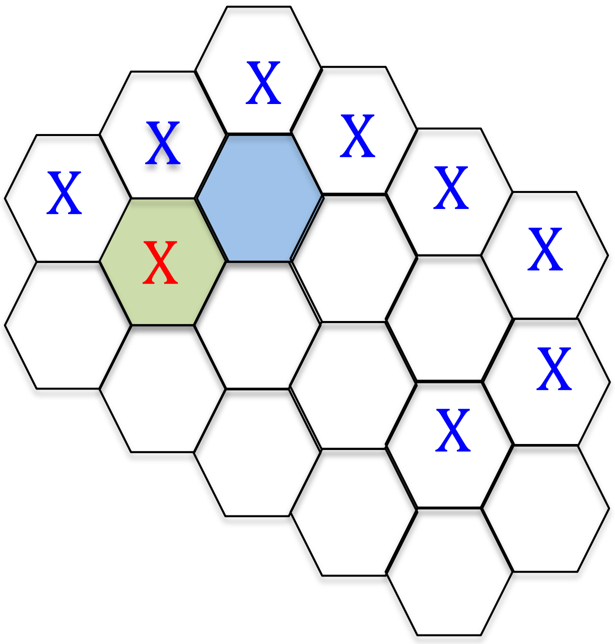


Figure 7. The red X (range 3) cannot run to the blue hex.

3.2.4. Jumping

A piece with jumping capability is able to jump over any number of pieces, in a straight line. In order to move, the piece leaves the source hex and lands on the destination hex. There may be no unoccupied hexes in line of hexes (except for the destination hex).

4. Variations

Students are expected to evolve the design and implementation of Hanto. In order to do this they will implement different variations, or releases, as the course progresses. Variations gain complexity, from very simple—even silly and trivial—versions to complete support for different Hanto versions and rule sets. This section describes the specifics of each version. The final form of this document will contain many more variations of Hanto than students will be able to implement in a single term. The instructor will select the appropriate versions for each term that Hanto is used as the subject of the project.

Unless the rule set specifically states otherwise, the rules of Hanto specified in the official Hanto rules that are applicable to the variation are in effect. This means that the requirement for a single, contiguous formation of pieces is always in effect.

4.1. Alpha Hanto

Alpha Hanto is a version of the game that gets the student started on developing the game and using the supplied code base. Using TDD, the implementation should be quite simple.

4.1.1. Pieces and capabilities

Only one type of piece is used for Alpha Hanto—the Butterfly. This Butterfly has no movement capability. You simply place it on the board.

4.1.2. The rule set

There is exactly one complete move in Alpha Hanto. Blue moves first and places the Butterfly at (0, 0). Next Red places the Butterfly adjacent to the Blue Butterfly. The game then ends in a draw.

Your implementation should do the following:

- Make sure that the correct and/or valid coordinates are passed in as arguments to *makeMove()*.
- Make sure that the correct piece is being moved.
- Return the correct results from invoking *makeMove()*. The first time should return OK and the second should return DRAW, assuming that the moves are legal.

4.1.3. Required deliverables

You will create a class called *AlphaHantoGame* in a **hanto.studentx.alpha** package. It must implement the *HantoGame* interface. Blue will always move first.

You will write the code using TDD and all of your test code will be placed under the **test** source folder. You should have at least 90% code coverage in the implementation code (the **hanto.studentx** package, excepting interfaces and enumerations). You <u>do not need to implement a test game</u> for this version of Hanto.

NOTE: AlphaHanto games are not created with a factory. The purpose of making the AlphaHanto version is to get you familiar with the process. AlphaHanto can be considered a *special* version of the game for developer training purposes.

4.2. Beta Hanto

Beta Hanto adds a little bit more capability to the game. This is still not a very playable game and there will most likely be no winner.

4.2.1. Pieces and capabilities

For this game, each player has one Butterfly and five Sparrows. Each of these have exactly the same capability. They may be placed on the board and not moved.

4.2.2. The rule set

In this version, pieces may be placed anywhere as long as it is adjacent to some piece already on the board (of course the first move is placed at (0,0)).

The Butterfly must be placed on the board by the fourth turn. If a side has not placed the Butterfly in the first three turns, then the only legal move for the fourth turn is to place the Butterfly.

Butterflies and Sparrows have exactly the same capabilities. They may only be placed on the board. Therefore, a game can last no more than six full turns. The game may end before this if a Butterfly is surrounded before the end of the sixth turn. For Beta Hanto, they just need to be placed next to a piece, regardless of color. If six moves are made and there is no winner, the game ends in a draw.

NOTE: *For this and all succeeding versions, if there is any illegality in a player's move, that player loses and the software throws a HantoException with an appropriate message. Also, no moves may be made after the game is over. This too results in a HantoException being thrown.*

4.2.3. Required deliverables

You will create a class called *BetaHantoGame* in a **hanto.studentx.beta** package. It must implement the *HantoGame* interface.

You will write the code using TDD and all of your test code will be placed under the **test** source folder. You should have at least 92% code coverage in the implementation code, as in Alpha Hanto.

4.3. Gamma Hanto

This version of Hanto is an extension to Beta Hanto that introduces the ability of a piece to move.

4.3.1. Pieces and capabilities

Gamma Hanto has exactly the same pieces as in Beta Hanto. The difference is that the pieces have the ability to *walk* one hex during a turn. Walking means that the piece can *slide* to an adjacent hex as described in section 3.2.1.

4.3.2. The rule set

The game will end after 20 turns. If there is no winner at the end of 20 turns, then the game is a draw.

Except for the first two pieces placed on the board (one Blue and one Red), all pieces must be added next to a piece of the same color and not next to a piece of the opposing color, even if it is also adjacent to a piece of the same color.

4.3.3. Required deliverables

You will create a class called *GammaHantoGame* in a **hanto.studentx.gamma** package. It must implement the *HantoGame* interface.

You will write the code using TDD and all of your test code will be placed under the **test** source folder. You should have at least 95% code coverage in the implementation code under **hanto.studentx** (excluding enumerations).

Your *HantoGameFactory* must now return an instance of your Gamma Hanto Game when the *makeHantoGame()* method is called with a *HantoGameID* of GAMMA_HANTO.

4.4. Delta Hanto

This version of Hanto adds a new movement behavior to Hanto and one rule change. You will put your Delta Hanto game in its own package, **hanto.studentx.delta**.

4.4.1. Global rule changes

From this version on, there are rare cases when a player cannot make a valid move or place a piece on the board. In this case, the player must resign. A player resigns by calling the *makeMove()* method with all null arguments. A player may resign at any time, not just when there is no move to be made. You do not have to determine whether a player has a move or not. That is the player's responsibility. All you have to do is accept the resignation. When you receive a resignation, you will return a move result indicating that the opponent wins.

If there is a move after the game has completed—either a win or a draw—you will throw an exception.

There is no move limit from this version moving forward.

4.4.2. Pieces and capabilities

Delta Hanto provides each player with one Butterfly, four Sparrows and four Crabs.

The Butterfly may Walk one hex in a turn.

The Crab may Walk up to three hexes in a turn.

The Sparrow can Fly as described in Section 3.2.2.

4.4.3. Required deliverables

You will create a class called *DeltaHantoGame* in a **hanto.studentx.delta** package. It must implement the *HantoGame* interface.

You will write the code using TDD and all of your test code will be placed under the **test** source folder. You should have at least 95% code coverage in the implementation code under **hanto.studentx** (excluding enumerations).

You also must implement a test Delta Hanto game that will be returned from the *HantoTestGameFactory*. This test game must implement the *TestHantoGame* interface.

4.5. Epsilon Hanto

Epsilon Hanto adds a couple of new movement types to the repertoire of the game capabilities. It also puts a constraint on how resignation is handled. All rules from previous versions apply to Epsilon Hanto, unless specifically stated.

This version implements a full Hanto game controller with the exceptions of some of the more complex movement types.

You will put the Epsilon Hanto game int he **hanto.studentx.epsilon** package.

4.5.1. Rule changes

The only rule change for this version is that if the player resigns and the player has a valid move you will return a *HantoPrematureResignationException*. The code for this follows. It will be placed in the **hanto.common** package under the **src** folder.

```
/*******************************************************************
 * This files was developed for CS4233: Object-Oriented Analysis &
Design.
 * The course was taken at Worcester Polytechnic Institute.
 *
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License
v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************/

package hanto.common;

/**
 * Exception that is only thrown when a player resigns while there
 * is still a valid move available for that player.
 * @version Oct 6, 2014
 */
public class HantoPrematureResignationException
   extends HantoException
{
   public HantoPrematureResignationException()
   {
       super("You resigned when you have a valid move available.");
   }
}
```

4.5.2. Pieces and capabilities

The pieces and capabilities for this game are shown in the following table:

Piece	# Available	Movement
Butterfly	1	Walk 1, must be placed by 4th move.
Sparrow	2	Fly up to 4 hexes distant (the distance between the start and end hex must be ≤ 5).
Crab	6	Walk 1
Horse	4	Jumps as described in section 3.2.4.

4.5.3. Deliverables

You will create a class called *EpsilonHantoGame* in a **hanto.studentx.epsilon** package. It must implement the *HantoGame* interface. **Note:** If you have structured your code such that all games are composed by the factory and there is no need for an **hanto.studentx.epsilon** package, that is acceptable.

You will write the code using TDD and all of your test code will be placed under the **test** source folder. You should have at least 95% code coverage in the implementation code under **hanto.studentx** (excluding enumerations).

The deliverables for Epsilon Hanto will be packaged with the deliverables for the tournament player as described in Section 5 and one submission will be made.

5. The Tournament Player

For the final project, you must produce a player for a Hanto tournament. Your Hanto player must implement the *HantoGamePlayer* interface that is in the **hanto.tournament** package. Your *HantoPlayer* must be named *HantoPlayer* and be in your **hanto.studentx.tournament** package.

Three files are supplied for you, the first two are the *HantoGamePlayer* interface and the *HantoMoveRecord*. These are both in the **hanto.tournament** package. The third is a starting *HantoPlayer* game in your **hanto.studentx.tournament** package. An archive of these will be made available in the course Web pages.

5.1. Requirements

You will produce a Hanto player that plays the game specified for this term. Your player must play a valid game of Hanto. A valid game is defined as such:

- As long as your player has a valid move to make, it must make a move. It cannot resign when there is a move to be made.
- The player plays a correct game. That is, no exceptions are thrown when the moves are entered in a game that follows the rules of Hanto as described in this document.
- Your player does not simply repeat a simple pattern of moves. For example, move from hex A to hex B, move from hex B to hex C, move from hex C to hex A, and repeats this.
- Your player does not intentionally try to lose the game. That is, you don't have a strategy where you intentionally try to surround your own Butterfly.

You will develop and test your test player using TDD. You are expected to have 95% code coverage in your **hanto.studentx** package under your **src** folder when running your tests. This coverage does not include enumerations.

5.2. Deliverables

Your player and your final Hanto game will be due on the day specified in the course Web pages.

References

[1] "Hive (Game)", Wikipedia, 22-Oct-2012, accessed 1-Jan-2013, http://en.wikipedia.org/wiki/Hive_(game).

[2] "Hanto", Agile Route, 2010, accessed 1-Jan-2013, http://www.agileroute.com/hanto/.

[3] "Hexagonal Coordinate System", Nick Thissen, VBForums, 22-Oct-2011, accessed 1-Jan-2013, http://www.vbforums.com/showthread.php?663283-Hexagonal-coordinate-system.

[4] H. B. Christensen, *Flexible, Reliable Software Using Patterns and Agile Development*. Boca Raton: Chapman& Hall/CRC, 2010, p. 496.

[5] "Simply Singleton", David Geary, JavaWorld, 25-Apr-2003, accessed 30-Jan-2013, http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html

[6] "Hexagon grids: coordinate systems and distance calculations", keekerdc blog, accessed 16-Feb-2013, http://keekerdc.com/2011/03/hexagon-grids-coordinate-systems-and-distance-calculations/.

[7] "How to Play Hanto," YouTube video, accessed 1-Sep-2014, https://www.youtube.com/watch?v=npjSWPm6gvg.

[8] Boardgame Arena Hive pages, accessed 17-Feb-2016, http://en.boardgamearena.com/#!gamepanel?game=hive.

[9] Gen42 games Hivepage, accessed 17-Feb-2016, http://gen42.com/hive.