Assignment 1.0 - Chess Library

Overview

This week, we will be focusing on unit testing and object oriented design by implementing the core objects and data structures for a chess game application. This will be the first of several weeks in building this application. By the end of the week, you should have implemented the tests and the corresponding implementation for the pieces and board of a chess game.

For this assignment, you are *required* to use either Eclipse or IntelliJ IDEA. Both are free and have powerful refactoring tools available.

Eclipse vs. IntelliJ IDEA

You are likely already familiar with Eclipse from earlier programming courses here at UIUC. A few of our course staff, prefer IntelliJ. If you've never tried it out, consider using it for this project, besides, it's free for UIUC students when you register with your @illinois email!

Assignment Format

This course is likely very different previous courses you have taken, in that we typically **reuse** your code from the previous week for each assignment. As such, don't waste your time with messy code: focus on maintainability.

Refer to https://wiki.illinois.edu/wiki/display/cs242/Style+Conventions for style and conventions.

Read this entire page before beginning your assignment, and post on Piazza if anything is still unclear.

What am I actually turning in?

Many students find our concrete expectations for this first assignment confusing, and that's perfectly normal. There are a few key things to note here:

- Your task is to design, test, and implement (more or less in that order) a chess library from scratch. This is intentionally vague, and we expect you to spend some time making design decisions for your library.
- For this assignment, your *deliverables* are simply your unit test suite and your implementation. It is not part of the requirements to add a GUI or main game loop for this week.

If something is still unclear, ask on Piazza, chances are someone else is also confused. In general, we are flexible with interpretations of the assignment, **as long as it does not trivialize any component of the assignment**.

Background

For this assignment, we assume that you have some knowledge of the chess board game. We will assume a simple 8 x 8 board, with two players, and the standard board pieces. Each player alternately moves and strategically captures the others' pieces, until the game ends when one player is in checkmate (the king is about to be captured, and cannot move), or there is a stalemate (neither player can move).

From Wikipedia:

Each chess piece has its own style of moving.

- The king moves *one* square in any direction.
- The rook can move any number of squares along any rank or file, but may not leap over other pieces.
- The bishop can move any number of squares diagonally, but may not leap over other pieces.
- The queen combines the power of the rook and bishop and can move any number of squares along rank, file, or diagonal, but it may not leap over other pieces.
- The knight moves to any of the closest squares that are *not* on the same rank, file, or diagonal, thus the move forms an "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the only piece that can leap over other pieces.
- The pawn may move forward to the unoccupied square immediately in front of it on the same file; or on its first move it may advance two squares along the same file provided both squares are unoccupied; or it may move to a square occupied by an opponent's piece which is diagonally in front of it on an adjacent file, capturing that piece.

We do not expect you to deal with special moves like *castling* for the king & rooks, or *en passant* or *promotion* for the pawn, however, if you want to have a complete Chess game at the end of the project, you might want to implement them for playability's sake.

Part I: Game Logic

First, select data structures and test and implement logic for the chess board itself. Since there are many types of chess pieces that will be involved in the game, we suggest you deal with this first, dealing with a general chess piece.

Specifically, you should implement the logic for components such as:

- The board shape / size
- Piece movement on the board
- Pieces capturing/killing other pieces
- Putting the king in check
- Game ending conditions (detecting when a player has won or lost)

You can safely assume that your game will be played with two players, and that this number of players will not change in future weeks. You do not need to implement the game loop, but a single for loop handling logic for turns and a GUI should be the only missing pieces for a working chess game in this week's implementation.

Extensibility

How much effort would it require you to change your library to support a rectangular (not necessarily square) board? How about one in the shape of a hexagon?

Design your game logic with considerations such as this in mind. Remember, we will be asking you to further extend your code next week.

Part II: Chess Pieces

Your library should be able handle movement for all of the standard chess pieces. This includes where pieces can and cannot move. Imagine that some other component of your code (the players / game loop) need to play the game. What methods do you need to implement for your chess pieces?

Logic for all of the following pieces should be implemented this week:

- Rook
- Bishop
- Knight

- King
- Queen
- Pawn

Again, you only need to worry about the most basic movements for each of these pieces, namely (1) moving across the board and (2) capturing another piece. You do not need to consider special cases such as *castling* for the king & rooks, or *en passant* or *promotion* for the pawn (unless you want to).

Extensibility

How much effort would it require you to add new types of chess pieces to your library? Design your chess with considerations such as this in mind. Remember, we will be asking you to further extend your code next week.

Object-Oriented Design in Java

One of the reasons we start CS 242 off with an assignment in Java is that it is fairly easy to follow good object-oriented design practices in Java, such as inheritance and polymorphism. We will expect you to understand and use these concepts, and they will make you life *much* easier for assignments like this.

If you are unfamiliar with these concepts, or could use a refresher, here is a relatively simple tutorial for using polymorphism in Java: http://www.artima.com/objectsandjava/webuscript/PolymorphismInterfaces1.html. We also suggest you familiarize yourself with abstract classes and interfaces in Java if you have never used them before.

Unit Testing

Unit testing is a great way to ensure that, given an implementation, the code works correctly. This is enforced by creating small (unit) tests for each function. Each test should be able to call the function it is testing, and determine if the function worked according to its specification.

Introduction

This can be a strange concept at first, but it is really quite simple: If you know the input to a function, and you know how the function works, you should know exactly what the output of the function is. Since you know what the output should be before even calling the function, you can compare the output you get from making the function call to the expected output. If they match up, then the test has succeeded. Otherwise, it fails.

One test is rarely sufficient to be confident in the correctness of a function. A complete test suite would try all corner-cases, as well as the common case. For instance, if a function operates on positive integers, and it is called with a negative parameter, does it do the right thing? The success and failure of tests are completely defined by the person designing the application. Therefore, it is necessary to clearly understand what a function should do in order to test it.

Each test should be named in a fashion that clearly says what it does. Each test should be written in such a way that it is immediately apparent from the prose of its statements:

- the objects it works on,
- specific methods it tests,
- an exceptions it calls or handles,
- any preconditions and postconditions that must be met.

If it doesn't, the test must include comments that fully explain what is missing.

Example

An example of a valid jUnit test would look like (for an imaginary Phone object):

```
public class PhoneTests {
        //note these are jUnit 4 style tests, the naming convention
is different for jUnit 3 style tests

        /**
    * The constructor is supplied with valid values so no exceptions
should be thrown.
    */
    @Test
    public void ValidConstructor1() throws Exception {
                String number = "1-555-555-5555"
                String name = "John Smith"
        Phone phone = new Phone(name,number);
        assertEquals(number, phone.getNumber());
                assertEquals(name, phone.getName());

    }

    /**
    * This test checks the case where the constructor is supplied a
```

```
phone number with too many dashes.
    */
        @Test(expected=InvalidPhoneNumberException.class)
    public void TooManyDashes() throws Exception {
                String number = "9-9-999-999-9999";
                String name = "John Smith";
        Phone phone = new Phone(name,number);
    }

        //more tests below...
}
```

Technique

Good testing requires you to test for the unexpected. Every function should have *at least* one test. However to get full credit, you should write tests which cover all corner cases and other possibilities that your library may eventually encounter. You do not need to write tests for extremely trivial, small functions (5 lines or less).

A common misconception is that more tests is always better. While good test coverage of your code is important, you can make the most of your time when unit testing by strategically selecting the test cases to implement. For example, if we are testing a function that adds two positive integers, we could test inputs -5, 0, 5, 6, 7, INT_MAX, etc., but testing all of 5, 6, and 7 may be redundant, since they are all fairly small positive numbers and so are very similar, likely redundant test cases.

This assignment

For your assignment, we expect you to thoroughly test both the game logic and chess pieces for your library. This is extremely important because the errors will propagate into future assignments!

For example, you may want to test:

- What happens when a player tries to move a piece to an empty space on the board?
- What happens when a player tries to move a piece to an invalid space (off the board)?
- What happens when a player captures another piece? Does the captured piece disappear?

- What happens when a player tries to move to a space already containing one of his/her pieces?
- What happens when the player's king is put in "check"?
- Which player moves first?

Remember, **test-driven development means writing your tests before you write their implementation!**

When writing your functions always keep in mind writing them so that they are easy to test. The more focused each individual function is, the easier it will be to test, and the more correct your code will be. Modularity is key to unit-testing. It may be a good idea to split your library into multiple classes based on the data structure(s) you use as an internal representation for the chessboard.

There are some resources which aid in the unit testing process, such as JUnit for Eclipse and IntelliJ. You are free to use these if you would like. Eclipse and JUnit are available on the CSIL Linux computers, and an introduction to using JUnit with Eclipse is provided in this assignment description.

jUnit in Eclipse and IntelliJ IDEA

One of the reasons we require you to use either Eclipse of IntelliJ for this assignment is jUnit integration. Both IDEs have had support for jUnit integration for some time.

Eclipse

There is a thorough tutorial for using jUnit + Eclipse here: http://www.vogella.com/articles/JUnit/article.html#eclipse_usingjunit

IntelliJ IDEA

There is some documentation for using jUnit with IntelliJ here as well:

- Super Simple Example
- Basic Configuration
- Fancier Use

Introduction to JUnit

An example of jUnit 4 tests was shown above. Generally, developers put all unit tests for one type in one file, with multiple files of tests composing a test suite for the library or application. It will be much easier for you to keep things organized if you follow a similar pattern. Although Eclipse and

IntelliJ will let you run a single file containing jUnit tests directly, it takes a bit more effort to run all tests in all files at once (your test suite).
The following examples will work for both Eclipse and IntelliJ, but are unnecessary for IntelliJ. It is possible to simply create "Run Configurations" in IntelliJ IDEA to accomplish this without adding any code, rather than created additional classes as we describe below.

For example, say I had tests for a Phone object in a file called PhoneTests and tests for an Address object in a file called AddressTests, I could have the IDE jUnit runner run them both by creating a file called something like ContactInfoTests and have it contain:

```
package contactInfo.tests;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

/**
 * Runs all tests for classes in the <code>contactInfo</code> package
 * @author Jerome Bell
 *
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({AddressTests.class, PhoneTests.class})
public class ContactInfoTests {
}
```

If I was to then instruct my IDE to run ContactInfoTests as a jUnit test, it would run all the tests in both the PhoneTests class and the AddressTests class. You might not need to, but you can chain it up even further if you divide your tests into subcategories and have a master suite file like:

```
package allTests;
import junit.framework.TestSuite;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import contactInfo.tests.ContactInfoTests;
import persistence.tests.DatabaseInterfaceTests;

@RunWith(Suite.class)
@Suite.SuiteClasses({ContactInfoTests.class,
DatabaseInterfaceTests.class})
public class RunAllTests extends TestSuite{
}
```

Last bit of help

When we solve these assignments, we *do* in fact write *all* of the test *before* the methods they test. After writing the tests, we write stub implementations of the tests that call them. The code then compiles without errors, but the tests fail since they didn't do anything yet. **This is normal.** Then one by one, we write the implementation for each of the methods until all of the tests pass.

Sample code and tests

To assist you in your efforts, here are two sample files **SquareTest.java** and **Square.java**. You do not need to use this class, but feel free to take portions of it to use in your chess data structures.

Still confused or need help?

First, ask questions on Piazza. If you have a question, there is a pretty good chance someone else has the some one and an even better chance that someone else in the class or one of the TAs will be able to answer it for you. If you are still having a problem, email your moderator or one of the TAs to get advice. Remember, its best to ask questions early on so they have time to be answered. Don't wait until the last second to get started then realize that you are confused.

Submission

This assignment is due at the **beginning of your discussion section the week of September 18th, 2017**. Please be sure to submit in SVN, and ask your moderator or TA before the deadline if you have any questions. Also, make sure to place your work in a folder called **Assignment1.0**, matching this spelling exactly.

For example, you should be able to bring up https://subversion.ews.illinois.edu/svn/fa17-cs242/NETID/Assignment1.0/ in a web browser and see all of your code (if you replace NETID with your actual netid)

Objectives

- Practice designing a library
- Learn how to write unit tests

Resources

- Eclipse vs. IntelliJ IDEA
- How is IntelliJ better than Eclipse?
- Eclipse vs. IntelliJ - Verdict is IntelliJ

- Point-by-point Comparison
- Another Point-by-point Comparison
- Object-Oriented Design
- Java Polymorphism Tutorial
- Polymorphism
- Java Abstract Classes
- Java Interfaces
- Testing
- JUnit API JavaDoc
- JUnit Assert
- JUnit (Wikipedia)
- JUnit Homepage
- Test-Driven development (Wikipedia)
- jUnit Integration
- Eclipse: Tutorial
- IntelliJ: Super Simple Example
- IntelliJ: Basic Configuration
- IntelliJ: Fancier Use
- Workshop Slides
- **Workshop1.pptx**

Grading

We will bias clarity over cleverness in evaluating your code. You should adopt a *"teaching"* mindset, always asking yourself, *"How would I change what I have written to make it as easy as possible for someone else to understand my code without my direct assistance?"*

Refer to the standard Grading rubric if any portion of this rubric is unclear. Note that the standard rubric assumes a scale of 0-2 for each category.

Category	Weight	Scoring	Notes
Basic Preparation	2	0-1	Ready to go at the start of section
Cleverness	2	0-2	The hardest points on the rubric
Code Submission	4	0-2	Submitted correct content on time and to the correct location in the repository
Decomposition	4	0-2	Project is adequately decomposed to different classes and methods

Category	Weight	Scoring	Notes
Documentation	4	0-2	Comments for each class and each function are clear and are following style guides
Effort	2	0-2	Perform considerable amount of work
Naming	2	0-2	Variable names and method names are readable and are following Java conventions
Overall Design	4	0-2	Have nice approaches and structures in overall
Participation	5	0-2.5	Interact with the group 2 times (ask a question, make a comment, help answer a question, etc.)
Presentation	4	0-2	Present the code clearly
Requirements - Chess Pieces	5	0-2.5	2 points for implementing basic movement rules and capture behaviors for all chess pieces 2.5 points for special movements or extra effort
Requirements - Data Structures	4	0-2	2 points for a basic chess game structure, including a board and two players etc.
Requirements - Game End Conditions	4	0-2	2 points for implementing game end conditions like checkmate and stalemate detection
Testing - Chess Pieces, Game Logic	5	0-2.5	2 points when junit tests for all chess behaviors implemented have coverage above 80% 2.5 points when code coverage is 95% or more
Testing - Data Structures	5	0-2.5	2 points when junit tests for implemented game structure have coverage above 80% 2.5 points when code coverge is 95% or more
Full Score	56		