

”BO” (Blue and Orange)

1 Objectives

- Use recursion
- Use lists
- Design and execute a test plan
- Use version control (SVN)
- Write documentation and use Javadoc
- Revise your work
- Comment your code

2 Assignment

Your team is to develop a design for a computer program to play the game BO, or “Blue and Orange,” which is remarkably similar to the game Go. See description of the game below. As before, your grade depends mostly on good design, good coding technique and a correctly working game.

You will be given a working GUI and an interface for a game which uses the GUI. You will also be given Javadoc files for the GUI, similar to those found in the Java API.

1. **You must display your game using the provided GUI.**
2. **You must use a 2D array in your implementation of your board.**
3. **You must use a linked list to create marker chains.**
4. **You must use recursion to remove markers from the board.** ¹

Why? Because the purpose of this project is to practice and demonstrate your knowledge of lists and recursion. The 2D array is the simplest way to implement the board and works with the provided GUI. The GUI will give you a visual way to spot errors in your algorithms and make the game actually playable when you are done.

3 BO – The Blue and Orange Game

The board on the screen has a grid of 9 by 9 points (or intersections) that are originally all empty. The two players are assigned a *marker color*. The players alternate selecting an empty point. A marker in the player’s color is placed on a selected point during the player’s turn. If a player places her marker next to another of her markers, this forms a *chain of markers* that are treated as one big marker. (A single marker is simply a one element chain.) When a player plays a marker between two chains, the two chains become one larger chain that includes the joining marker.

If a player places her marker at a point such that her markers *surround* an opponent’s chain of markers, all the opponent’s markers in the chain are *removed from the board*.

For example, if the board had the following configuration (where B is one marker and O is the other).

```
BBB
B000_
BBB
```

and player B selected the underlined square, all three Os would be removed from the board. The above configuration consists of three B chains and one O chain.

Notice that a move may cause several chains to become surrounded, such as in the following configuration:

```
BB BB
BOO_OOB
BB BB
```

A B in the underlined location will surround the two O chains and all four Os will be removed.

If a player puts her marker in a position where the chain becomes totally surrounded, then it *must* also cause an opponents chain to be surrounded. Otherwise, the move is not allowed. Likewise, a player is **not** allowed to place a marker in a position that will recreate a board layout that was previously seen (to prevent cycles). For example, imagine the following move given this initial board layout:

```
BO
BO O
BO
```

B plays in the middle of the O markers (where it is surrounded). In this case, the O marker in the middle of the Bs gets removed.

```
BO
B BO
BO
```

The orange player might want to play her marker where the previous marker was removed, but this is not allowed as it recreates the previously seen board layout.

Players have the option of *passing their turn* if they do not wish to play a marker.

A game of BO ends when either both players have passed or there are no official moves left to make. The winner will be the player with the most markers on the board.

4 Implementation Expectations

1. Every chain should be represented as a linked list. Don't use a Java `List` class for the chains; implement your own. Why do you need to implement your own? Because each chain should keep track of the vacant points around it as well. When there are no vacant points left, then the chain needs to be removed from the board.
2. When a chain is surrounded, use a recursive method that starts at the head of the linked list and removes each part of the chain. Why do this recursively? Because you should think of a chain as an individual unit that is removing itself one piece at a time. When the first marker is removed, it leaves a smaller chain to be removed.
3. I have provided a `Colour` enum class for you to use for the blue and orange markers. (Note that this is spelled different than the standard `Color` class for differentiation) You should look up Java enums to understand more about how they work. <http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html>
4. As always, try to be as object-oriented as possible. Classes should contain only enough knowledge and responsibility as they require. For this project, you need to at least have the following classes: `Marker`, `Chain`, `Board`, and `BlueOrange`. (If you are really clever, you might not need the `Chain` class, but it is easier to solve the problem if you have one.) All of your classes should implement a `toString` method for testing purposes.
5. I have provided three interfaces for you: `IMarker`, `IBoard`, and `IBlueOrange`. You must use them when implementing the `Marker`, `Board`, and `BlueOrange` classes, respectively.
6. When the game ends, you should throw a `BOException` with a message including the game ending condition and players' scores. `BOException` has been provided for you. You only need to use it.

5 The GUI

As with your previous projects, I am giving you a GUI. However, this time it is *not* optional to use it. Carefully read the provided Javadocs to understand how it works. **You must use the IMarker, IBoard, and IBlueOrange interfaces in your implementation!** Why? The GUI uses these interfaces. The GUI is provided in a jar file on the class website called `BO.jar`. Look up jar files on the internet to find out what they are and how to use them. You can also use the command `man jar` to learn about the jar command in a terminal window.

5.1 Look over the Javadoc for the GUI

You should see a GUI docs link on the class website. If you follow that link, you will find a complete technical specification of the GUI. Carefully read the class comments for all of the classes and the method comments for the interfaces. This will tell you some important implementation details.

5.2 What the GUI expects

To use the GUI, your `BlueOrange` class must have/do the following items:

- call the `BOVizualizer` GUI's `run` method in its main method. For example:

```
BlueOrange game = new BlueOrange(...);
BOVizualizer.run(game);
```

- implement the `IBlueOrange` interface
- use the provided `Colour` enum class to switch between the players
- use `JOptionPane` methods to display messages and warnings to the players

You can see more details in the JavaDocs that come with the GUI.

6 Testing

You must **develop** and **execute** a testplan which shows that the board is correctly updated after each move. Remember that there are two players and numerous ways to create marker chains and surround them. At least cover as many of the simple scenarios as you can involving 6 pieces or less. You should also test for situations involving a player passing his turn. You may change the default board size if you wish for the tests. If you do so, *document it*.

Develop a testplan to test all cases. Execute it and save your results. You will find it much easier to use a text version of the board (via `toString`) rather than the GUI for this.

Carefully explain all testplans and show the actual results. Your test plan must cover updates in all eight directions, both players, alternating turns, pieces that should not be updated, etc. Examples of what you expect to see are a very good idea.

Testplans are worth more in this assignment, so do not skip them.

7 Other details

You may use any existing classes and objects that you worked with in the labs or in the previous projects. You will need to produce Javadoc for this assignment. As always you need to provide detailed comments on your classes, methods and fields (at least as much as I provide in the Javadoc). In software engineering, providing detailed documentation is a good practice which makes code more understandable.

You must use Subversion for this project.

8 Important Deadlines

8.1 Phase 0 - The day it is assigned

You must work in a team of two.

One of the goals for this course is to introduce you to good software engineering practice, which includes teamwork.

You may not work with someone who has already been your partner for an assignment. If there are an odd number of students in the class, I will allow one team of three people. The person left without a partner gets first dibs on being in that team of three. Make sure your name and your partners name(s) are included (typed) with all submitted documents.

Send me one email with the usernames of all people in your team and your team name. If you do not provide a team name, I will make one up. This time, the made-up team names will likely be embarrassing.

8.2 Phase 1

Hand in the following items via Subversion. They are numbered in the order you should produce them in.

1. Problem Statement, class descriptions, and CRC cards.
2. The UML diagram must show all of the classes involved in this project, including the GUI. As a result, this diagram should include at least 10 classes.
3. A BlueOrange class which implements the IBlueOrange interface and compiles and runs but does **not** have to actually play BO yet. This will require you to put in method headers but as little code as possible. However, you need to be able to get the GUI to show up.
4. All member data must be implemented (coded) with sharing tags (public, private, etc..) as needed to match your UML.
5. All methods must have headers to match your UML (and satisfy the required interfaces).
6. Your algorithms *must* be seen in Javadoc-style comments at the top of each method. Your recursive method must have the base cases and recursive cases clearly explained (preferably in pseudo code rather than full english). You should **not** write any more code in this method yet than is absolutely necessary to get it to compile (perhaps a return value of false).
7. A Javadoc technical specification for your classes (remember to check the private option to get a tech spec).
8. Include a definition **in your own words** of what a jar file is. Remember to cite any sources you used to find this information.
9. Your testplans are expected to be complete and very detailed.
10. You must be able to add pieces to the board (although you do not have to remove them).
11. You must have `toString` methods that can print out a text version of the board. The Javadoc comment should give an example of what prints. Since you cannot print in blue and orange, find some other obvious way to represent them.
12. You must have completed and commented a method to count the number of pieces each player has. It is better style and code-reuse to have one counting method that works for either player than it is to have two very similar counting methods. You can call this method from `add` or `pass` in your `BlueOrange` class every time a piece is placed to test it. For now, have the `getChainCount` method return this information.
13. You must have completed and commented the `reset` method. We can test it using the reset button on the GUI.
14. How will you know when the game is over? Figure out exactly how you will know and put a comment about how in the method which will detect the end and do something about it. (One method is obvious involving two players passing. But how do you determine if there are no valid moves left?)

I will run your (not working yet) game from your main method to see that you can click on the board and count pieces correctly.

Your code should follow the Style Guide posted on the CSCI 204 web site.

Do not wait for my input before beginning the next phase.

8.3 Phase 2

Complete the game and test it.

Handin via Subversion

- User manual
- UML (updated if necessary)
- Technical specification (Javadoc on private mode)
- Testplan and results of executing your testcases.
- Code

In Phase 2, everything must be implemented correctly. In particular, you need to return the number of chains for each player from the `getChainCount` method, and have marker chains remove themselves recursively.

8.4 Phase 3

If you worked in a team, you must individually email me an assessment of your and your partner's contribution to the assignment. The assessment should be a percentage of how the work was split. It should add to 100. In a perfect team, you would both score a 50. If you are not writing me 50, Bob 50, I'd like a sentence or two saying why. If I decide the work split was very unfair, the grades will be adjusted accordingly. Your email must have the subject 204 team: OrangeBlue. This email must arrive within 1 class day of the phase 3 deadline.

If you send me an email with a subject other than the one seen here, I may choose not to count it (and I may not find it in the many incoming emails until after projects are graded). If you do not email me, I will assume you agree with your partner(s) evaluation of you. If nobody on your team emails me, I will assume everything went fine.

8.5 Extra Credit: Official Go Scoring (Group or Individual)

In the actual game of Go, the rules for scoring are more complex than those here. For extra credit, you must look up from a credible source (i.e., not Wikipedia or random Google page) a set of official scoring rules and implement the scoring rules. For credit, you must show me your source and explain the scoring rules. Then show me how they were integrated into your project. The rules must also be documented in your User's Manual.