Project Deliverable 6—Individual: Personal Software Process & Quality

PSP2, Unit Testing—30 points

Instructions:

- The is an individual assignment. ***No collaboration is allowed.***
- Use of generative AI tools (such as ChatGPT, etc.) is NOT allowed.

Late submissions will NOT be accepted (please note course policies in Syllabus).

Submission Instructions:

Submit a zipped folder named: `{YourASURiteUserID}-ProjectDeliverable6.zip`

(e.g., skbansa2-ProjectDeliverable6.zip)

This compressed folder should contain the following:

1. Source Code files:
    a. Source code files from Deliverable 4 or 3
    b. Unit test code (placed in the correct package)
2. A generated code coverage report
3. `ProjectDeliverable6.docx` (or pdf) with Completed Time Log, Estimation worksheet, Design form, Defect Log, Personal Code Review and Project Summary provided at the end of this assignment description.
    a. Make sure to provide responses to the reflection questions listed in ProjectDeliverable6 file (this document).
4. A few screen shots showing
    a. results of your JUnit testing,
    b. code coverage report.
5. A readme file (optional; submit if you have any special instructions for testing).

Grading Rubric:

Unit Testing—15 points

Test Results, Code coverage report, Postmortem reflection question responses—5 points

PSP process—10 points

Time log (2), Defect log (2), Estimating Worksheet & Design form (2), Code Review (2), Project Summary (2)

Program Requirements:

Using JUnit, conduct unit testing for the game logic class(es) and computer player class in the `core` package. Create a separate package called `test` for your test class(es).

Use your code from Deliverable 3 or 4. You do not need to test JavaFX class.

Provide test methods for all important methods of the game logic and computer player modules; use equivalence portioning to pick test cases. You should have both success and failure cases. Use a code coverage tool, such as EclEmma, to generate a code coverage report; generate this report for only the core classes you are placing under test. You must achieve at least 90% code coverage of your game logic and computer player classes. Include the code coverage report in your submission.

Personal Process:

Follow a good personal process for implementing this game. You will be using PSP2 in this assignment. Estimate and track your effort and defects for the unit testing code that you write. Don't forget to conduct a personal code review.

PSP Forms

- Please use the **estimating worksheet** contained herein to estimate how big your program might be.
- Please include in the **design form** any materials you create during your design process. It's at the end of this document.
- Please use the **code review checklist** contained herein to statically analyze your code for common mistakes.
- Please use the **time log** (provided at the end of this document) to keep track of time spent in each phase of development.
- Please use the **defect log** (provided at the end of this document) to keep track of defects found and fixed in each phase of development.
- When you are done implementing and testing your program, complete the **project summary** form to summarize your effort and defects. Also answer the reflection questions.

Phases

Follow these steps in developing the game:

Plan—understand the program specification and get any clarifications needed.

1. **Estimate** the **time** you are expecting to spend on unit testing.

2. **Estimate** the **defects** you are expecting to inject in each phase.
3. **Estimate** the **size** of the program (only for **new code** that you will be adding).
4. Enter this information in the **estimation columns** of the project summary form. Use your best guess based on your previous programming experience.
5. Use the provided **estimating worksheet** to show how you are breaking up code into smaller modules and estimating.

Design—design the test classes and methods. Test case generation for various test methods needs to be done. Keep track of time spent in this phase and log it. Also keep track of any defects found and log them. Use this phase to design various test methods and test cases.

Code—implement the tests. Keep track of **time** spent in this phase and log. Also keep track of any **defects** found and log them.

Code Review—use the **code** review **guidelines** provided later in the document to conduct a personal review of your code and fix any issues found. Provide comments in the checklist about your findings. There should be a minimum of 4 comments.

Test—test your program thoroughly and fix any bugs found. The goal here is to test all core classes thoroughly. Keep track of **time** spent in this phase and log. Also keep track of any **defects** found and log them.

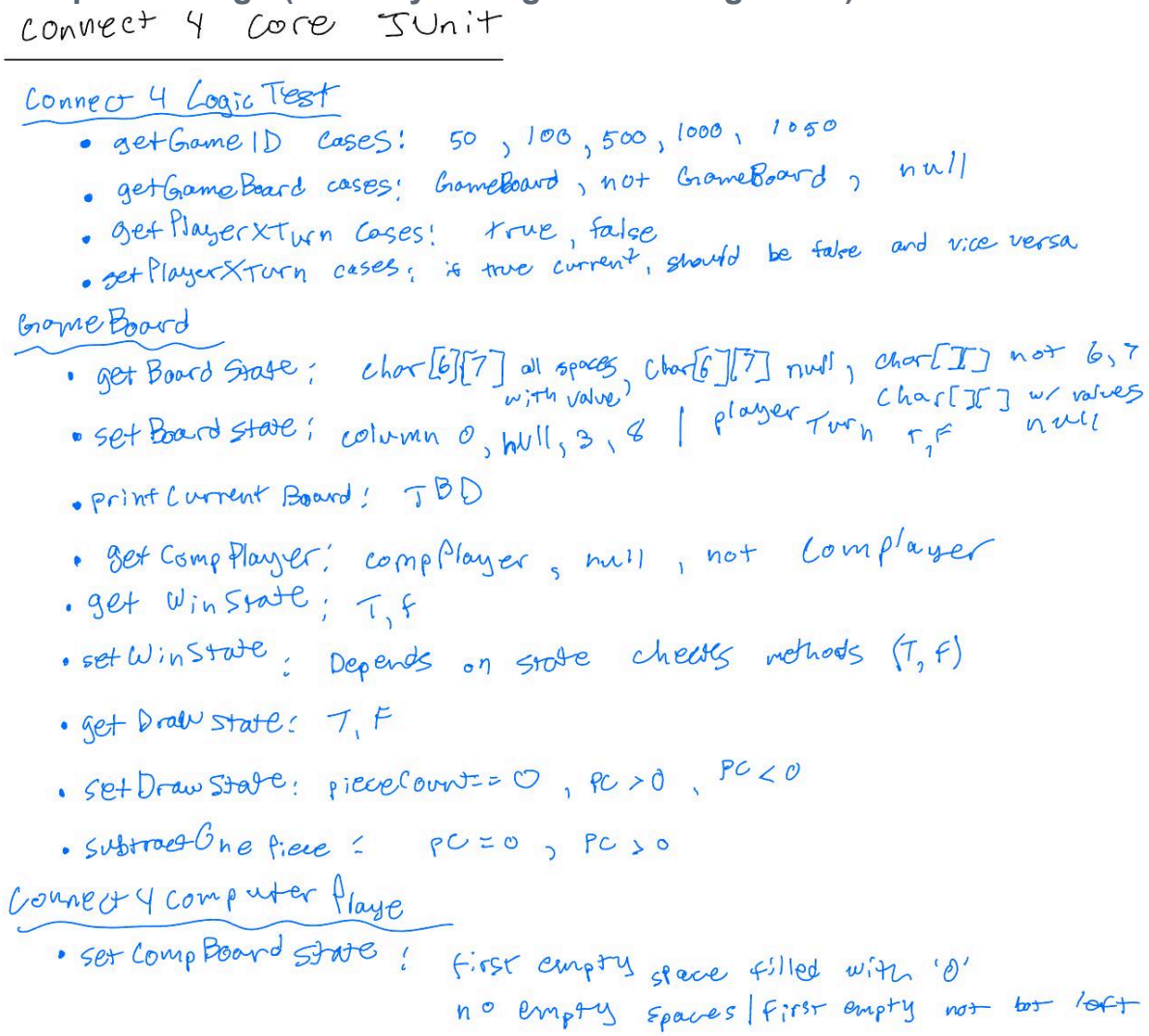
Postmortem—complete the actual, and to date **columns** of the **project summary** form and answer the **reflection questions**.

Estimating Worksheet

PSP2 Informal Size Estimating Procedure

1. Study the requirements.
2. Sketch out a crude design.
3. Decompose the design into “estimatable” chunks.
4. Make a size estimate for each chunk, using a combination of:
    a. visualization.
    b. recollection of similar chunks that you’ve previously written
    c. intuition.
5. Add the sizes of the individual chunks to get a total.

Conceptual Design (sketch your high-level design here)



Module Estimates

Module Description	Estimated Size
Connect4LogicTest class buildout	30 mins; 30 LOC
GameBoardTest class buildout	60 mins; 50 LOC
Connect4ComputerPlayerTest class buildout	20 mins; 20 LOC
Total Estimated Size:	**110 mins; 100 LOC**

Code Review Checklist

Add comments. Boxes are checkable (☒+👆= ☒).

Specification / Design

☒ Is the functionality described in the specification fully implemented by the code?
Yes, fully implemented test cases
☒ Is there any excess functionality in the code but not described in the specification?
No, nothing further

Initialization and Declarations

☒ Are all local and global variables initialized before use?
Yes
☒ Are variables and class members of the correct type and appropriate mode
Yes
☒ Are variables declared in the proper scope?
Yes
☒ Is a constructor called when a new object is desired?
Yes
☒ Are all needed import statements included?
Yes and none extra
☒ Names are simple and if possible short
☒ There are no usages of “magic numbers”
No magic numbers present

General

☒ Code is easy to understand
Yes, clear test cases
☒ Variable and method names are spelt correctly
Yes, no spelling errors
☒ There is no dead code (i.e., code inaccessible at runtime)
No
☒ Code is not repeated or duplicated
Nothing duplicated
☒ No empty blocks of code
No empty code

Method Calls

☒ Are parameters presented in the correct order?
yes
☒ Are parameters of the proper type for the method being called?
Yes
☒ Is the correct method being called, or should it be a different method with a similar name?
Yes
☒ Are method return values used properly? Are they being cast to the needed type?
Yes, no casting needed/happening

Arrays/Data structures

☒ Are there any off-by-one errors in array indexing?
None
☒ Can array indexes ever go out-of-bounds?
There is one spot that throws an ArrayOB exception so all areas are handled
☒ Is a constructor called when a new array item is desired?
Yes
☒ Are your data structures ideal?
I believe so
☒ Collections are initialized with a specific estimated capacity
Yes

Object

☒ Are all objects (including Strings) compared with `equals` and not `==`?
NA
☒ No object exists longer than necessary
Yes
☒ Files/Sockets and other resources if used are properly closed even if an exception occurs when using them
NA

Output Format

☒ Are there any spelling or grammatical errors in the displayed output?
No
☒ Is the output formatted correctly and consistently in terms of line stepping and spacing?
Yes

Computation, Comparisons and Assignments

☒ Check order of
☒ computation/evaluation
good
☒ operator precedence and
good
☒ parenthesizing
good
☒ Can the denominator of any divisions ever be zero?
NA
☒ Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?
NA
☒ Check each condition to be sure the proper relational and conditional operators are used.

Good

☒ If the test is an error-check, can the error condition actually be legitimate in some cases?

Yes

☒ Does the code rely on any implicit type conversions?

No

Exceptions

☒ Are all relevant exceptions caught?

Yes

☒ Is the appropriate action taken for each catch block?

Yes

☒ Are all appropriate exceptions thrown?

Yes

☒ Are catch clauses fine-grained and catching specific exceptions?

Yes

Flow of Control

☒ In switch statements, is every case terminated by `break` or `return`?

No switches, but yes in for loop once

☒ Do all switch statements have a default branch?

NA

☒ Check that nested if statements don't have "dangling else" problems.

No

☒ Are all loops correctly formed, with the appropriate initialization, increment and termination expressions?

Yes

☒ Are open-close parentheses and brace pairs properly situated and matched?

Yes

Files

☒ Are all files properly declared and opened?

☒ Are all files closed properly, even in the case of an error?

☒ Are EOF conditions detected and handled correctly?

☒ Are all file exceptions caught?

NA for all

Documentation

☒ Methods commented in clear language

Yes

☒ Most comments should describe rationale or reasons (the *why*); fewer should describe the *what*; few should describe *how*.

Yes

☒ Are there any out-of-date comments that no longer match their associated code?
No
☒ All public methods/interfaces/contracts are commented describing usage
yes
☒ All edge cases are described in comments
Yes
☒ All unusual behavior or edge case handling is commented
Yes
☒ Data structures and units of measurement are explained
Yes

PSP Time Recording Log

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments
4/17/24	8:30PM	9:15PM	0	45	Plan + Design	Created estimates and rough sketch of plan
4/18/24	6:35PM	10:35PM	60	180	Code	Built the test classes with all of the different test case options
4/19/24	3:30PM	4:10PM	0	40	Code Review	Went through the list of checks and added any defects found
4/19/24	4:10PM	4:30PM	0	20	Test	Confirmed test cases and tried to think of any missing
4/19/24	4:30PM	4:45PM	0	15	Postmortem	Answered reflections and filled in statistics

- **Interruption time**: Record any interruption time that was not spent on the task. Write the reason for the interruption in the "Comment" column. If you have several interruptions, record them with plus signs (to remind you to total them).
- **Delta Time**: Enter the clock time you spent on the task, less the interrupt time.
- **Phase**: Enter the name or other designation of the programming phase being worked on. Example: Design or Code.
- **Comments**: Enter any other pertinent comments that might later remind you of any details or specifics regarding this activity.

PSP Defect Recording Log

Serial No.	Date	Defect Type No.	Defect Inject Phase	Defect Removal Phase	Fix Time (duration)	Fix Ref	Description
100	4/18/24	80	Code	Code	15	NA	Was checking the pieceCount variable in a test case as it went to zero and it wasn't getting all the way to zero because my loop was incorrect

- **Serial No.**: The unique id you associate with the defect; allows you to reference it later.
- **Defect Type No.**: The type number of the type—see the PSP Defect Type Standard table below and use your best judgement.
- **Defect Inject Phase**: Enter the phase (plan, design, code, etc.) when this defect was injected using your best judgment.
- **Defect Removal Phase**: Enter the phase during which you fixed the defect.
- **Fix Time**: Enter the amount of time that you took to find and fix the defect.
- **Fix Ref**: If you or someone else injected this defect while fixing another defect, record the number of the improperly fixed defect. If you cannot identify the defect number, enter an X. If it is not related to any other defect, enter N/A.
- **Description**: Write a succinct description of the defect that is clear enough to later remind you about the error and help you to remember why you made it.

PSP Defect Type Standard

Type Number	Type Name	Description
10	Documentation	Comments, messages
20	Syntax	Spelling, punctuation, typos, instruction formats
30	Build, Package	Change management, library, version control
40	Assignment	Declaration, duplicate names, scope, limits
50	Interface	Procedure calls and references, I/O, user formats
60	Checking	Error messages, inadequate checks
70	Data	Structure, content
80	Function	Logic, loops, recursion, computation, function defects

90	System	Configuration, timing, memory
100	Environment	Design, compile, test, or other support system problems

PSP2 Project Summary

Time in Phase

Phase	Estimated time (in minutes)	Actual time (in minutes)	To Date	% of total time to Date
Planning	30	30	153	6%
Design	20	15	255	10.1%
Code	110	180	1655	65.3%
Code Review	30	40	135	5.3%
Test	30	20	200	8%
Postmortem	30	15	135	5.3%
TOTAL	**250**	**300**	**2533**	**100%**

Defects Injected

Phase	Estimated Defects	Actual Defects	To Date	% of total to Date
Planning	0	0	0	0%
Design	0	0	15	51.7%
Code	2	1	11	37.9%
Code Review	1	0	2	7%
Test	1	0	1	3.4%
Postmortem	0	0	0	0%
TOTAL	**4**	**1**	**29**	**100%**

Final Summary

Metric	Estimated	Actual	To Date
Program Size (Lines of Code—LOC) [1]	100	433	1452
Productivity (calculated by LOC/Hour)	24	86.6	34.4
Defect Rate (calculated by Defects/KLOC) [2]	40	2.3	20

Reflection Questions

1. How good was your time *and defect* estimate for various phases of software development?

My time estimate was pretty good compared to previous estimates. I still missed the mark slightly though. My defect estimate was also pretty good. This wasn't a huge defect deliverable.

2. How good was your program size estimate, i.e., was it close to actual?

[1] LOC stands for lines of code.
[2] KLOC stands for kilo lines of code (1000 lines)

My size estimate was not very close. I was close on 2 of the 3 test classes I built, but I wasn't quite sure how many test cases I would end up writing when I estimated so I missed that mark.

3. How much code coverage did you achieve for your core classes?

Total coverage was about 98% which I was happy with being that it's my first attempt at unit testing.

PSP Design Form

Use this form to record whatever you do during the design phase of development. Include notes, class diagrams, flowcharts, formal design notation, or anything else you consider to be part of designing a solution that happens BEFORE you write program source code. Attach additional pages if necessary.

Connect 4 Core JUnit

Connect 4 Logic Test

- getGameID cases: 50, 100, 500, 1000, 1050
- getGameBoard cases: GameBoard, not GameBoard, null
- getPlayerXTurn cases: true, false
- setPlayerXTurn cases: is true current, should be false and vice versa

GameBoard

- getBoardState: char[6][7] all spaces with value, char[6][7] null, char[I] not 6,7
- setBoardState: column 0, null, 3, 8 | playerTurn T, F | char[I] w/ values null
- printCurrentBoard: TBD
- getCompPlayer: compPlayer, null, not CompPlayer
- getWinState: T, F
- setWinState: Depends on state checks methods (T, F)
- getDrawState: T, F
- setDrawState: pieceCount == 0, PC > 0, PC < 0
- subtractOnePiece: PC = 0, PC > 0

Connect 4 Computer Playe

- setCompBoardState: first empty space filled with '0'
  no empty spaces | first empty not ~~for left~~