Unit Testing: Personal Software Process & Quality

PSP2, Unit Testing—50 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}-UnitTesting.zip`
(e.g., skbansa2-UnitTesting.zip)

This compressed folder should contain the following:

1. Source Code files:
    a. Source code files from previous assignment (Personal Process 2 or Personal Process 3)
    b. Unit test code (placed in the correct package)
2. A generated code coverage report
3. `UnitTesting.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 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—25 points
Test Results, Code coverage report, Postmortem reflection—12 points
PSP process—13 points
Time log (2), Defect log (2), Estimating Worksheet & Design form (2), Code Review (5), 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 previous assignments (Personal Process 2 or Personal Process 3). You DO NOT need to test UI (JavaFX & TextConsoleUI) classes.

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**.

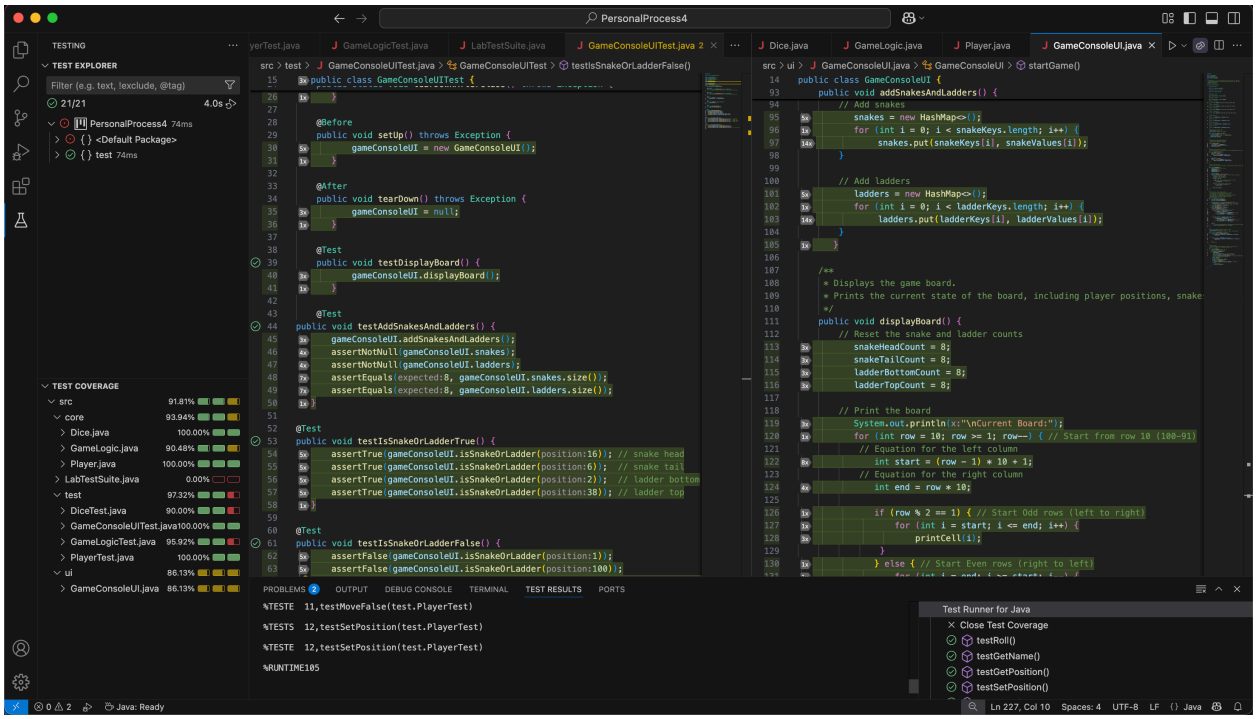
Estimating Worksheet

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)

Nothing to really sketch, just make test files and populate them



Module Estimates

Module Description	Estimated Size
GameConsoleUITest	100 Lines
PlayerTest	50 Lines
DiceTest	50 Lines
GameLogicTest	100 Lines
Total Estimated Size:	**300 Lines**

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.

Test Files to Create:

GameConsoleUITest.java
PlayerTest.java

DiceTest.java
GameLogicTest.java

Code Review Checklist

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

Specification / Design

☒ Is the functionality described in the specification fully implemented by the code?
☒ Is there any excess functionality in the code but not described in the specification?
// Players, snakes, and ladders each have a unique color to help identify them easily

Initialization and Declarations

☒ Are all local and global variables initialized before use?
☒ Are variables and class members of the correct type and appropriate mode
☒ Are variables declared in the proper scope?
☒ Is a constructor called when a new object is desired?
☒ Are all needed import statements included?
☒ Names are simple and if possible short
☒ There are no usages of "magic numbers"

General

☒ Code is easy to understand
// Comments throughout the code are there for aid
☒ Variable and method names are spelt correctly
☒ There is no dead code (i.e., code inaccessible at runtime)
☒ Code is not repeated or duplicated
☒ No empty blocks of code

Method Calls

☒ Are parameters presented in the correct order?
// Assuming this means in the order they are used
☒ Are parameters of the proper type for the method being called?
☒ Is the correct method being called, or should it be a different method with a similar name?
☒ Are method return values used properly? Are they being cast to the needed type?

Arrays/Data structures

☒ Are there any off-by-one errors in array indexing?
// There are no errors, program runs as expected
☒ Can array indexes ever go out-of-bounds?
// No
☒ Is a constructor called when a new array item is desired?

☒ Are your data structures ideal?

// Given a rework of this project, data structures chosen may change. I feel it could have been done more efficiently but due to time constraints, I've ignored this for the moment; program runs as expected so pressing reason.

☒ Collections are initialized with a specific estimated capacity

Object

☒ Are all objects (including Strings) compared with `equals` and not `==`?

☒ No object exists longer than necessary

// Pretty sure Java automatically gets rid of objects not in use anymore or maybe that means "nulling" them out.

☒ Files/Sockets and other resources if used are properly closed even if an exception occurs when using them

// None Used

Output Format

☒ Are there any spelling or grammatical errors in the displayed output?

// There are none

☒ Is the output formatted correctly and consistently in terms of line stepping and spacing?

Computation, Comparisons and Assignments

☒ Check order of

- ☒ computation/evaluation
- ☒ operator precedence and
- ☒ parenthesizing

☒ Can the denominator of any divisions ever be zero?

// No

☒ Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?

☒ Check each condition to be sure the proper relational and conditional operators are used.

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

☒ Does the code rely on any implicit type conversions?

// No

Exceptions

☒ Are all relevant exceptions caught?

☒ Is the appropriate action taken for each catch block?

☒ Are all appropriate exceptions thrown?

// Either that or return to base case in presence of error

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

Flow of Control

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

☒ Do all switch statements have a default branch?

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

☒ Are all loops correctly formed, with the appropriate initialization, increment and termination expressions?
☒ Are open-close parentheses and brace pairs properly situated and matched?

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?
// No files were opened

Documentation

☒ Methods commented in clear language
☒ Most comments should describe rationale or reasons (the *why*); fewer should describe the *what*; few should describe *how*.
☒ Are there any out-of-date comments that no longer match their associated code?
☒ All public methods/interfaces/contracts are commented describing usage
☒ All edge cases are described in comments
// No edge cases
☒ All unusual behavior or edge case handling is commented
☒ Data structures and units of measurement are explained

PSP Time Recording Log

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments
4/29	12:00	12:05	0	5	Design	Setting up IDE, creating project, looking at file, thinking of tests to create
4/29	12:05	2	30	85	Code/Test	Creating Tests and files and testing them
4/30	9:02	9:22	0	20	Code Review	Completing code review
4/30	9:22	9:45	0	23	Post Mortem	Completing document

- **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
							None

- **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	0	0	180	13.8%
Design	5	5	245	18.8%
Code/ Test	60	85	650	49.8%
Code Review	15	20	30	2.3%
Postmortem	60	23	198	15.1%
TOTAL	**140**	**133**	**1303**	**100%**

Defects Injected

Phase	Estimated Defects	Actual Defects	To Date	% of total to Date
Planning	0	0	0	0%
Design	0	0	2	22%
Code/Test	0	0	3	33%
Code Review	0	0	0	0%
Postmortem	0	0	4	44%
TOTAL	**0**	**0**	**9**	**100%**

Final Summary

Metric	Estimated	Actual	To Date
Program Size (Lines of Code—LOC) [1]	300	308	308
Productivity (calculated by LOC/Hour)	130.43	139.36	???
Defect Rate (calculated by Defects/KLOC) [2]	0	0	???

Reflection Questions

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

   My time was around the same I took longer in some parts where I thought I would be shorter and vice-verse

2. **How good was your program size estimate, i.e., was it close to actual?**
   They were pretty accurate. Player and Dice files took less than the GameUI and logic files.

3. **How much code coverage did you achieve for your core classes?**
   93.94%

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