Bonus Lab: GitHub Copilot in VS Code

Contents

[Some words of wisdom before we begin… 2](#_Toc198279656)

[Time limit: 5 hours 2](#_Toc198279657)

[General Approach: *Distrustfully* Pair-Program with AI 2](#_Toc198279658)

[The Design: AI-implement your domain logic’s interface 3](#_Toc198279659)

[Machine setup – disable Copilot Autocomplete 4](#_Toc198279660)

[Stage 1: initial game state 5](#_Toc198279661)

[Write the test first 5](#_Toc198279662)

[Copy/paste the rules of the game into a block comment 6](#_Toc198279663)

[Prompt Copilot Edits to implement these rules 6](#_Toc198279664)

[Record what happened in log.txt 7](#_Toc198279665)

[Commit your completed Stage 7](#_Toc198279666)

[Stage 2, 3, 4: add more rules to the game 8](#_Toc198279667)

[Some of the rules below seem like a lot of work for a bonus lab 8](#_Toc198279668)

[Do not let the LLM modify you test cases 8](#_Toc198279669)

[Do not trust LLM-generated test cases more generally 8](#_Toc198279670)

[Stage 2 rules: (normal play rules) 9](#_Toc198279671)

[Stage 3 rules: (new color) 10](#_Toc198279672)

[Stage 4 rules: (monopoly) 11](#_Toc198279673)

[Reflection 12](#_Toc198279674)

In this bonus lab, we are going to experiment with using generative AI to incrementally implement the rules of a game for us. We will carefully design an AI-friendly and test-friendly API. We will use TDD to first manually write exhaustive JUnit tests based on the rules of the game. Then we will feed those tests, the rules, and the design to the AI to have it provide the implementation for us. If any tests fail, we will tell the AI what it did wrong and ask it to correct itself.

We will iterate a few times to simulate real-life changes to the requirements over time, and gauge their impact on the correctness of the AI-generated code. This experiment will ultimately simulate how one would safely use AI in industry to write code. While the technology changes all the time, the hope is that the principles outlined in this Lab will inform your use of generative AI in your future work.

# Some words of wisdom before we begin…

## Time limit: 5 hours

While this exercise is very exciting and new, Buffalo asks that you please spend no more than five hours on it. We are exploring what GitHub Copilot’s various settings can and can’t do for us, but this exploration could go on forever, and the answer changes every few weeks.

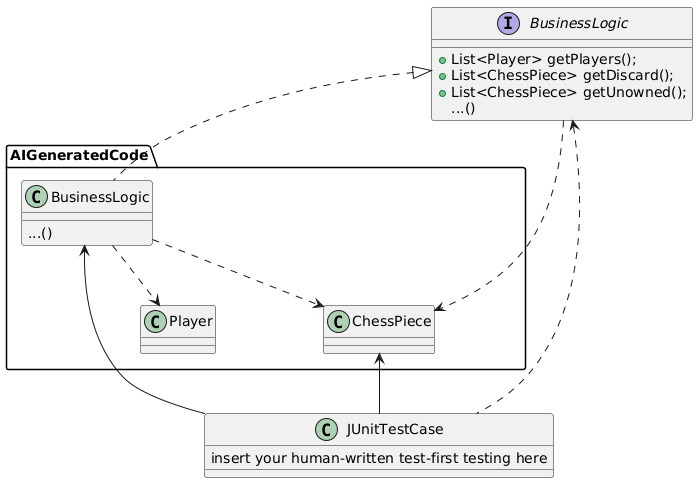
## General Approach: *Distrustfully* Pair-Program with AI

As we implement this game, we will always abide by the principle of **never blindly trusting the output of generative AI.** Generative AI is known to hallucinate, and hallucinations are the enemy of everything that Software Quality Assurance values. This is why we are going to use **AI-assisted pair programming** as follows:

1. **Disable AI Autocompletions.** Remember how we never blindly trust the AI? We especially do not trust AI Autocompletion. We will only allow the AI to edit our code when we prompt it to do so.
2. **Write the JUnit tests ourselves first**, based on our human interpretation of the game's rules.
3. **Let the AI edit our production code to make the tests pass**, using the whole codebase and its test cases as input.
4. **Run the tests** and correct the AI output with manual edits when something goes wrong.

## The Design: AI-implement your domain logic’s interface

From the outset, we are going to tightly constrain the AI to only generate implementations of a Java interface that we give it. The interface will look vaguely like a finite state machine: its methods will represent each of the possible inputs into the system, and calling those methods will advance the program to the next state. It will look something like this:



Throughout the Lab, we will add these methods to the BusinessLogic interface, then let the AI implement them inside the concrete class AIGeneratedCode.BusinessLogic. Along the way, we will want to make assertions about the state of the game. We will represent the state with at least two data structures: ChessPiece and Player, which we will populate over time.

# Machine setup – disable Copilot Autocomplete

We are going to set up GitHub Copilot in a very specific way that is probably not how you are used to interacting with LLMs. While we will encourage you to experiment with your setup, we find that this one specific setup generally produces better results.

1. Install Visual Studio Code.
2. Fork and clone the git repo for this Lab from the link on Moodle.
3. Open this project in VS Code and set up Java + Maven for the first time if you have not already done so in VS Code:
   1. In the bottom right, it may prompt you to install the Extension Pack for Java; do so.
   2. When you do that, VS Code will open a **Get Started With Java Deployment** tab asking you to follow some further set up steps like install the JDK; **ignore this screen** because you already have a JDK.
   3. In the Extension search bar, search for Maven.
   4. Verify that **Maven for Java** is already installed; install it if not.
4. In VS Code, find the Copilot button on the topmost bar: .
5. Click the **icon** on that button to open Copilot on the right bar. Follow the prompts to set up Free access to Copilot – it is basically a button press at this point.

But now you need to set up Copilot in theway that is least likely to introduce errors into your code:

1. When your Copilot license is set up, click the **arrow** next to the Copilot button, then select **Configure** **Code Completions**.
2. In the drop down menu that appears in the Search bar, select the fourth option,   
   **Disable Completions.**
3. Once more, click the **arrow** next to the Copilot button, then select **Open Copilot Edits**. (Ctrl-Shift-I on Windows).

If you ever want the AI to do something for you: we find it works best if you prompt it in the **Copilot Edits** pane, review the proposed changes carefully, then decide whether to accept or reject the changes. The autocompletions are not nearly as good as Edits and are especially prone to mistakes as you type out test cases – the very time you don’t want AI involved!!

# Stage 1: initial game state

We will use GitHub Copilot to help us implement the following rules of the game: (they are initially incomplete and very vague; we will expound upon them incrementally)

* In a game, there are four players, and each player has a pile of chess pieces in front of them.
* Players start with five random chess pieces in their piles.
* There is a massive Unowned pile of chess pieces in the middle of all players.
* In the game, there are a total of 64 chess pieces, of equal amounts of the colors red, blue, green, and yellow.
* There is a single chess piece drawn from these 64 pieces in the Discard pile.

You’ve probably made similar setups in your own term projects before. However, since we do a lot of TDD in this course, it only makes sense we would use a variation of TDD here.

## Write the test first

We first want to write a test (in src/test/java) that creates a new instance of the AIGeneratedCode.BusinessLogic class (in src/main/java/AIGeneratedCode/BusinessLogic.java) from the diagram. In this test, we want to assert that the initial state of the game is correct. Along the way, we want to set up all the classes, interfaces, and methods you see in the previous UML diagram, so that the AI understands the design.

Go ahead and write this test according to the diagram. When you come across missing classes, fields, and methods: fix them in the normal non-AI way by generating placeholders for them. Remember: **don’t trust the AI**. Write this test yourself.

Here are the assertions you should make about the returned State object:

* Test that getPlayers() returns four players. Each player has a pile of chess pieces containing five chess pieces.
* Test that getDiscard() returns one non-null chess piece.
* Test that getUnowned returns (64 - 4\*5 – 1) pieces left in it.
* Across all piles in total, each color has: 8 pawns, 2 rooks, 2 knights, 2 bishops, 1 queen, and 1 king. (observe that this last set of assertions was **not** in the stated rules of the game)

Run this test. View the test results in the TEST RESULTS tab at the bottom of VS Code, especially the rightmost column of that tab. You should have written just enough code for it to compile, but it should still fail to pass the tests because we haven’t written the real code.

## Copy/paste the rules of the game into a block comment

Open your **AIGeneratedCode.BusinessLogic** class. Add the following block comment above the class declaration to describe the initial rules of the game:

/\*  
Stage 1 rules:  
- In a game, there are four players, and each player has a pile of chess pieces in front of them.

- Players start with five random chess pieces in their piles.

- There is a massive Unowned pile of chess pieces in the middle of all players.

- In the game, there are a total of 64 chess pieces, of equal amounts of the colors red, blue, green, and yellow.

- There is a single chess piece drawn from these 64 pieces in the Discard pile.  
  
Stage 2 rules:  
WIP  
  
Stage 3 rules:  
WIP  
  
Stage 4 rules:  
WIP  
\*/

**Best Practice**: When you want a generative AI to write code for you, we’ve found that **writing your business logic as a comment inside your code** seems to generate the best results.

## Prompt Copilot Edits to implement these rules

1. Find the **Copilot Edits** tab on the right pane.
   1. If you lost it: click the arrow next to the Copilot Icon -> Open Copilot Edits.
2. In the bottom right of the chat box that appears, select an LLM; I used **gpt-4o** since it worked out of the box, but realize that this list changes all the time.
3. Run this prompt: Implement the Stage 1 rules of the game in AIGeneratedCode/BusinessLogic.java so that it passes all tests. #codebase.
   1. In the version of Copilot used as of this writing, the **#codebase** tag at the end signals to Copilot to add your project to its context.
4. Copilot will briefly churn on your codebase and then give you a diff to review and accept; accept the changes.
5. Run your test case to see whether it passes now.
   1. **If your test failed**: try to figure out why it failed, then **fix the code manually**.

## Record what happened in log.txt

Since this is more of a Lab and not a Tutorial, we want to record what happens as we experiment with Copilot’s features.

Create a file called **log.txt.** In that file, write down how your initial Copilot Edit went. I suggest that you write each entry with this template:

Stage: 1

Copilot Edits Prompt:

LLM:

Test Results:

Changes I had to make to the AI-generated code:

## Commit your completed Stage

When those initial tests pass, git add all your .java files as well your log.txt. Commit your initial work: "Completed Stage 1."

# Stage 2, 3, 4: add more rules to the game

Each Stage on the subsequent pages describes another iteration of rules that we want to add to the game. To simulate changes to a system's requirements in real life, we are going to implement them just one Stage at a time. Our LLM-friendly workflow will be:

1. Write the tests first.
2. Copy/paste the new rules into the block comment.
3. Prompt Copilot Edits to implement Stage X.
4. Run the tests and correct any AI-generated mistakes.
5. Record what happened in log.txt.
6. Commit: “Completed Stage X.”

## Some of the rules below seem like a lot of work for a bonus lab

Some of these rules will seem very daunting and complex. But remember we are experimenting with having the AI implement them for us. Feel free to introduce \*many\* interface methods and public fields to make your test cases easier for you to write. Experiment with the AI's ability to infer your intent and implement your ideas.

## Do not let the LLM modify your test cases

Sometimes, the LLM thinks it knows better than you about your own test cases. It does not. **Reject** any changes that involve modifying your test cases and try re-prompting it.

## Do not trust LLM-generated test cases more generally

Some of these tests will be tedious to write. You might be tempted to try further experimenting with having Copilot generate test cases for you. We welcome you to try, but we caution against it until you get the hang of the rules you are implementing.

When Copilot fails to generate test cases, it does so rather spectacularly. If you then let Copilot generate the code implementing those same bad tests, you create what is called a *degenerate feedback loop*. Copilot’s default autocompletion, as of this writing, is especially bad about amplifying degenerate feedback loops, which is why we recommended earlier that you disable it entirely.

**Do not enter degenerate feedback loops**. Write your test cases yourself. You can do it; keep the test code and interface methods as simple as possible.

# Stage 2 rules: (normal play rules)

Replace the Stage 2 WIP lines in the block comment with these:

Stage 2 rules:

- Each turn, a player can take only one of three actions: Discard a piece, Draw a piece, or Forfeit the game.

- To Discard a piece, the chess piece discarded must either match the color of the last chess piece added to the Discard pile, or match the kind of chess piece that it is. Otherwise, it cannot be discarded, and the player must discard a different piece.

- If a player cannot discard a piece according to any rules of the game, then they must instead select a new piece from the Unowned pile to draw, with their eyes open.

- A player must always Discard a piece if they can; they cannot simply Draw if they just feel like it.

- If a player ever needs to Draw from the Unowned pile and that pile is empty, then that player can only Forfeit the game.

- A player who Forfeits is removed from the player rotation.

- When a player successfully discards all their pieces from their pile according to the rules of the game, then they win the game.

- If, after a Forfeit, only one player is left in the game, then that last remaining player automatically wins, even if their only option would be to Forfeit.

Follow our AI-friendly TDD process to implement these rules and log how it went.

**Tips**:

* As you write your test cases, add intuitive methods to the interface. I recommend that you outline each of the three player actions as their own methods in BusinessLogic.
* Think about how to handle invalid moves and what that will look like in your test cases. (How did you handle invalid input in your project?)
  + Whatever design you settle on: write the tests first so that you demonstrate the desired behavior to the LLM.
* Practice evolutionary design. Use your tests to outline as simple of a design as you can imagine. Let the LLM fill in the implementation and see how it does.

# Stage 3 rules: (new color)

The challenge with these new rules is that they affect multiple existing places in the code. How does the AI handle that?

Stage 3 rules:

- There is a new color of chess piece: rainbow, and there are now 80 chess pieces at the start of a new game.

- Rainbow-colored pieces can always be discarded, regardless of the color of the last discarded chess piece.

- Whenever a Rainbow-colored piece is discarded, the turn order of the game reverses.

- Rainbow-colored pieces are their own color for all other intents and purposes.

**Tip**: go back and modify your very first test case to expect 80 pieces now, in equal amounts of the five colors.

# Stage 4 rules: (monopoly)

What is interesting about this set of rules is that it has non-local effects on the game state.

Stage 4 rules:

- If you have at least five pieces of the same color in your pile and hold more pieces of that color than any other player, you gain a Monopoly over that color.

- While you hold a Monopoly over a color, you can always elect to Draw an available Pawn of the Monopolized color from the Unowned pile, in lieu of any other action you would normally be forced to take.

- You can only hold one Monopoly; it is always over the color that you hold the most pieces of.

- If another player ever gains more pieces of the Monopolized color in their pile than you have in your pile, you lose your Monopoly to the other player.

# Reflection

In your log.txt, make one last commit answering the following questions:

1. How did the LLM end up designing the game's logic? Did it use a general OOP design, a design pattern you recognize, a mess of if statements, or something else entirely?

2. What tasks did the LLM struggle with?

3. Were there times where you successfully re-prompted the LLM to fix an error? Go back through your chat history and briefly summarize the instances where this occurred.

4. Were there times you needed to manually fix the code? What was the LLM getting wrong, and how did re-prompting fail you?