Bulldog: State Pattern and Java Reflection

Author:
Michael Yattaw

May 2, 2025

The final part of the Bulldog program was optional and involved to improving the program by implementing a new improvement. I thought to myself, what better way to improve an object-oriented programming project than by incorporating design patterns that involve complex refactoring to make the code significantly more organized? So, the new proposed significant change is to refactor the Bulldog project to include the state design pattern for console state and swing user interface state. This proposition was likely only to work with my Bulldog program because I decided to keep the old code that allowed Bulldog to be executed through console commands.

So, like usual, I began looking over my code to get an idea what to prompt the Deepseek AI model. The first classes I had to compare were the console main Java class and the BulldogUI main class. These two classes gave me a good idea for a prompt which looked something like this:

> “I have written a game in Java called Bulldog, where players take turns rolling dice until they either choose to stop or roll a 6. The goal of the game is to reach a target score before the other player. Currently, I have two main methods: one for the console command version and another for the Java Swing UI version. I want to implement the State design pattern so that players can choose between the console and Swing UI.”

This was followed by the two classes I examined earlier. One class was called “BulldogApplication” and the other “BulldogUI”.

The Deepseek model had a good approach by creating an interface that both the console and Swing classes were now required to implement, which was called “GameState”. However, in addition to the new “GameState” class, the model generated two new classes called “ConsoleState” and “SwingState”, which extracted code from the two classes I had prompted. After I implemented these, it also generated another class called “GameContext”, which accessed the current “GameState”. Lastly, the model updated my “BulldogApplication” class to select the state at runtime. In general, these new classes only took 10 minutes to generate after prompting the AI.

After spending over 30 minutes implementing and debugging, I asked the AI how to better structure my packages. Here’s the prompt:

> "How should I structure my packages if BulldogApplication is inside me.yattaw.bulldog and BulldogUI is inside me.yattaw.bulldog.ui?"

The AI replied with the following package layout:

```
me.yattaw.bulldog/
├── core/                  # Core game logic
│   ├── model/             # Game models
│   │   ├── Player.java
│   │   ├── HumanPlayer.java
│   │   ├── AIPlayer.java
│   │   └── ... (other player types)
│   ├── GameContext.java   # State pattern context
│   └── GameEngine.java    # Core game mechanics
├── states/                # State pattern implementations
│   ├── GameState.java
│   ├── ConsoleState.java
│   └── SwingState.java
├── BulldogApplication.java # Original console main
├── ui/                    # Swing UI components
│   ├── BulldogUI.java
│   ├── panels/
│   │   ├── MainMenuPanel.java
│   │   └── GamePlayPanel.java
│   ├── model/
│   │   └── PlayerModel.java
│   └── viewer/
│       └── ScoreboardViewer.java
└── Main.java              # New unified entry point
```

Now I noticed a new class called "GameEngine", so I asked Deepseek to update "BulldogApplication" to use it. The following is the updated classes:

```java
/**
 * Michael Yattaw
 * Login ID: michael.yattaw@maine.edu
 * COS 420, Spring 2025
 * Bulldog Project
 * BulldogApplication class: Main class for the Bulldog game
 * Manages matches, players, and gameplay until a player reaches the winning score.
 */
public class BulldogApplication {

    /**
     * Entry point for console version of Bulldog game.
     * @param args Command-line arguments (unused)
     */
    public static void main(String[] args) {
        if (args.length > 0 && args[0].equalsIgnoreCase("console")) {
            GameContext context = new GameContext(new ConsoleState());
            context.initializeGame();
            context.startGame();
        } else {
            SwingUtilities.invokeLater(() -> {
                GameContext context = new GameContext(new SwingState());
                context.initializeGame();
            });
        }
    }

}
```

```java
/**
 * Core game engine that manages matches and player interactions.
 */
public class GameEngine {

    public static final int WINNING_SCORE = 104;
    private final Map<String, List<Player>> matchPlayers = new HashMap<>();

    public void playUntilWinner(String matchName) {
        List<Player> playerList = getMatchPlayers().get(matchName);
        if (playerList == null || playerList.isEmpty()) {
            System.out.println("No players in the match.");
            return;
        }
        while (true) {
            for (Player player : playerList) {
                int roundScore = player.play();
                player.setScore(player.getScore() + roundScore);
                if (player.getScore() >= WINNING_SCORE) {
                    return;
                }
            }
        }
    }

    public boolean createPlayer(String match, String playerType) {
        Player player = createPlayerInstance(playerType);
        if (player == null) return false;
        return matchPlayers.get(match).add(player);
    }

    private Player createPlayerInstance(String playerType) {
        return switch (playerType.toLowerCase()) {
            case "ai" -> new AIUniquePlayer("AICodedPlayer");
            case "fifteen" -> new FifteenPlayer("FifteenPlayer");
            case "human" -> new HumanPlayer("HumanPlayer");
            case "random" -> new RandomPlayer("RandomPlayer");
            case "unique" -> new UniquePlayer("UniquePlayer");
            case "wimp" -> new WimpPlayer("WimpPlayer");
            default -> null;
        };
    }

    public void createMatch(String matchName) {
        matchPlayers.put(matchName, new ArrayList<>());
    }

    public List<Player> getPlayersForMatch(String matchName) {
        return matchPlayers.get(matchName);
    }

    public Map<String, List<Player>> getMatchPlayers() {
        return matchPlayers;
    }

    public int getPlayerScore(String matchName, int playerIndex) {
        List<Player> players = matchPlayers.get(matchName);
        if (players == null || playerIndex < 0 || playerIndex >= players.size()) {
            return -1;
        }
        return players.get(playerIndex).getScore();
    }

    public String getPlayerName(String matchName, int playerIndex) {
        List<Player> players = matchPlayers.get(matchName);
        if (players == null || playerIndex < 0 || playerIndex >= players.size()) {
            return null;
        }
        return players.get(playerIndex).getName();
    }
}
```

This part of the program assignment is now almost complete. I spent another hour implementing and verifying everything worked. The only concern is that the “GameState” interface contained methods “playTurn()” and “displayResults()” which are not yet implemented in “ConsoleState” and “SwingState”, but I’ve left that for future work. Interestingly, the “GameContext” also introduces the Facade pattern, unintentionally applying another design pattern to the project.

Now for the last part, which is essentially repeating another proposed change to improve the program. The final lecture class this semester went over Java reflection, so my second proposed change is to modify the createPlayerInstance method in my GameEngine class to utilize reflection to automatically add player objects from the player types package into the switch statement. Although, because we are using reflections to initialize our Player object, we will no longer need a switch statement but instead check to see if the player type exists, if it does then return the instance of the class.

For the second part of the program assignment, I began to prompt the Deepseek AI model for the code I wanted. The prompt was written as follows:

> “I would like to update my createPlayerInstance method to use reflection instead of the switch statement below. I have a list of player classes in the me.yattaw.bulldog.core.players.types package. Here is my createPlayerInstance method, and here are a few examples of my player classes:”

The prompt was then followed by the createPlayerInstance method and three different player classes, such as FifteenPlayer, WimpPlayer, and RandomPlayer. The AI model gave me a few new methods for working with reflection and initializing a new player object based on a string type. However, I now needed a way to retrieve a list of player types from the package, so I prompted the AI model to give me a method to list all player types from a package. Now I had functioning code that scanned for player classes in a package, retrieved the name of each player by class, and initialized a player by name. Therefore, I created a new ReflectionHelper class and put all these methods inside it.

```
public final class ReflectionHelper {

    /**
     * Dynamically discovers all available Player types in the package.
     * @return List of available player type names (e.g., ["ai", "human", "random"])
     */
    public static List<String> getAvailablePlayerTypes() {
        String packageName = "me.yattaw.bulldog.core.players.types";
        List<String> types = new ArrayList<>();

        try {
            List<Class<?>> playerClasses = getClassesInPackage(packageName).stream()
                    .filter(cls -> Player.class.isAssignableFrom(cls) && !cls.equals(Player.class))
                    .toList();

            for (Class<?> playerClass : playerClasses) {
                String className = playerClass.getSimpleName();
                String typeName = className.replace("Player", "").toLowerCase();
                types.add(typeName);
            }
        } catch (Exception e) {
            System.err.println("Error scanning player types: " + e.getMessage());
        }
        return types;
    }

    /**
     * Creates a Player instance by type name (e.g., "ai", "random").
     * @param playerType The type of player to create (case-insensitive).
     * @return New Player instance, or null if invalid.
     */
    public static Player createPlayerInstance(String playerType) {
        String packageName = "me.yattaw.bulldog.core.players.types";
        try {
            Optional<Class<?>> matchedClass = ReflectionHelper.getClassesInPackage(packageName).stream()
                    .filter(Player.class::isAssignableFrom)
                    .filter(cls -> cls.getSimpleName().equalsIgnoreCase(playerType + "Player"))
                    .findFirst();

            if (!matchedClass.isPresent()) {
                return null;
            }

            Class<?> playerClass = matchedClass.get();
            Constructor<?> constructor = playerClass.getConstructor(String.class);

            String playerName = playerClass.getSimpleName().replace("Player", "");
            return (Player) constructor.newInstance(playerName);
        } catch (Exception e) {
            return null;
        }
    }


    // Helper: Scans package for classes
    public static List<Class<?>> getClassesInPackage(String packageName) throws Exception {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        String path = packageName.replace('.', '/');
        Enumeration<URL> resources = classLoader.getResources(path);

        List<Class<?>> classes = new ArrayList<>();
        while (resources.hasMoreElements()) {
            URL resource = resources.nextElement();
            if (resource.getProtocol().equals("file")) {
                classes.addAll(findClasses(new File(resource.getFile()), packageName));
            }
        }
        return classes;
    }
    // Helper: Recursively finds classes in a directory
    private static List<Class<?>> findClasses(File directory, String packageName)
    throws ClassNotFoundException
    {

        List<Class<?>> classes = new ArrayList<>();
        if (!directory.exists()) return classes;

        for (File file : directory.listFiles()) {
            if (file.isDirectory()) {
                classes.addAll(findClasses(file, packageName + "." + file.getName()));
            } else if (file.getName().endsWith(".class")) {
                String className = packageName + '.' + file.getName().substring(0,
                    file.getName().length() - 6);
                classes.add(Class.forName(className));
            }
        }
        return classes;
    }

}
```

The final step of this assignment was for me to manually implement all of this new code inside both SwingState and ConsoleState classes. For ConsoleState, instead of manually printing each player type, I used a string formatter to input all available players from the new ReflectionHelper class. For the SwingState, I took a similar approach, but for the JComboBox. Overall, I spent about 30 minutes prompting the AI and only 15 minutes implementing and debugging the code.

My final thoughts for this program are that I am pleased with how it turned out, but I am a little disappointed that I stopped updating the console version of Bulldog halfway through the semester. More specifically, I have a lot of overlapping code, such as GameEngine and PlayerModel, because the console version never implemented the new PlayerModel. On the other hand, I am glad that I decided to keep this old code because it allowed me to reuse it for the state design pattern and made the project more object-oriented.