Information Expert
We use this pattern frequently in our design, for example our Stack class is responsible for fulfilling the results of moves, and as the expert on token and board positioning, it fulfills the responsibilities of ensuring tokens are properly captured.

Creator
Stack is also the Creator for the Token class, as it aggregates Token objects in an ArrayList.. As such, it acts as a creator pattern for our purposes.

Low Coupling
We use the low coupling pattern to a reasonable degree, for example, GameAI is coupled to Move, Player, Board and Stack, but in an earlier iteration, CPUPlayer (a now defunct class) was directly coupled to all these classes instead!

High Cohesion
To a great degree we promoted High Cohesion by frequently separating UI elements from their class and taking advantage of UI classes instead, like with SettingsUI and Settings, and GameState being created with the express intention of handling saving and loading of our game, rather than shoving it into an unrelated class.

Controller
We did not make use of this pattern, as we tended to include logic within each UI class. For a project of this size, we did not find it worthwhile to have separated use-case or even proper facade controllers over obvious and direct integration in our UI classes. If the project was larger and we had more time, we would include the pattern for the benefit of BoardUI and SettingsUI.

Pure Fabrication
GameState is an example of us using Pure Fabrication to solve the save/load problem, which has no real-world representation for the game of focus- other than leaving the pieces on the table, which is covered by leaving the program open – and thus meets the Pure Fabrication pattern by definition.

Protected Variations
We did not make use of this pattern, and we absolutely would given an extra iteration. The Move chain for each move type is the most obvious and painful example of code fragility. In general, due to our implementation, accessing details about the players frequently involves two or more chained commands rather than a protected variation pattern.

Singleton
Absolutely could’ve, used but not used for our BoardUI. Doing so would be trivial and an excellent advancement in future proofing the program.

Command/CommandHolder
As referenced in Controller, our UI classes have embedded logic, and in some cases in the aforementioned classes, Command/CommandHolder would be preferable or the only viable option for reducing their size and complexity.