**A page (at most) explaining the use of patterns in your design, or where you would use patterns if you had more time.**

In our Blockus game development, we decide to GRASP pattern. Which provided us with a foundation for our object-orientated game design. Creator: Which helps us answer to which class creates a given Object. For instance, In our case to understand who creates the object **Blockuspiece**. We identified that because object **Blokus** is closely related to **Blockuspieces**, therefore, we were able to conclude that blokus is the creator. Information expert: In our case because the player marker needs to find the blokus pieces to which it is to move and the options pertaining to the blokus piece. The **Blockus** object aggregates all the blokus pieces, so the blokus has the Information needed to fulfill this responsibility. Low Coupling: Another major GRASP pattern that we followed was low Coupling. It is a measure of how strongly one object is connect to or has knowledge of other objects. So using the information expert principle it guided us to a choice that supports low coupling. Referring to Information expert it helped us find the object that has most of the information required for the responsibility therefore in our case **Blockus** and assign responsibility there. Had we placed the responsibility anywhere else say for instance **MainMenu**, the overall coupling would have been higher because more information or object must be shared away from there actual source, as the **Blockuspiece** had to be shared with **Blockusplayer**, away from the actual source on the **Blokus** itself. Controller: It’s a simple layer architecture that has a User interface and a domain layer. In our case Player in the game blokus, generates and UI events such as selecting and placing a piece/ taking a move and the UI software object such as JFrame window and JButton processed the event and cause the game to play. So when whenever an Object in the UI layer picks up an event, it delegates the request to an object in the domain layer. High Cohesion: Using this principle helped us to measure how functionality related to the operation of software elements are. It also helps us understand how much work an object is doing. So for instance, it was better for us to different create different classes to performs different methods. Polymorphism: In our case when related alternatives vary based on classes, we assigned responsibility for such alternatives using polymorphic operation- to the types for which its behavior varied. Pure fabrication: It helps us assign a highly cohesive set of responsibilities to a class that doesn’t represent a domain concept. So when a player takes place a piece/ takes a turn: pieces are used in many games and by assigning placing a piece and flip or rotating a piece responsibility in player makes it impossible to generalize this move. Also, It becomes impossible to simply rotate or flip a piece without placing a piece again. So, therefore, we attempted to create a class that holds the pieces, so that the player can place them, flip or rotate.