**SOFTWARE ARCHITECTURE DOCUMENT**

* Table of Contents
  + Architectural representation: Page 1
  + Architectural Decisions Page 2 -10
    - Low Coupling Page 3- 4
    - High Cohesion Page 5-6
    - Information Expert Page 7
    - Controller Page 8
    - Creator Page 9
    - Polymorphism Page 16-18
    - Factory Page 19-21
  + Logical View Page 21-24

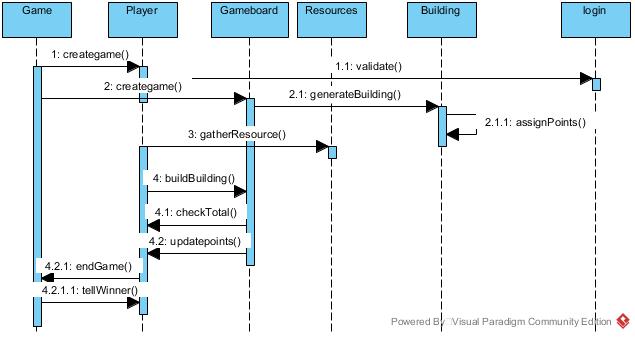
Architectural Representation

In this document we will be discussing the different choices that we have made when designing our architecture. We have chosen a few different key aspects, namely low coupling and high cohesion, Creator, Information Expert, and Controller. In Low coupling and High cohesion, we highlight choices where we simplified a class to split into more so that it was not bloated, and one where we combined classes that were similar and could be. In creator we showed how the game class created an instance of the login class. In Information Expert we show how the game class controls the information being transferred in our system. In Controller we describe how the game class controls the general flow of the entire system. We had made a variety of choices throughout describing the architecture, and were able to describe what choices we made when starting the initial designs and how they varied from the final design. We used the different GRASP Patterns to help us design a better architecture that follows these standards. As described in class, we wanted to follow the key motto of skillful responsibility assignment. Appended to the end of the document is the Logical view, which reflects how our choices are linked together in a package diagram.

Architectural Decisions

Choice 1:

Figure 1.1



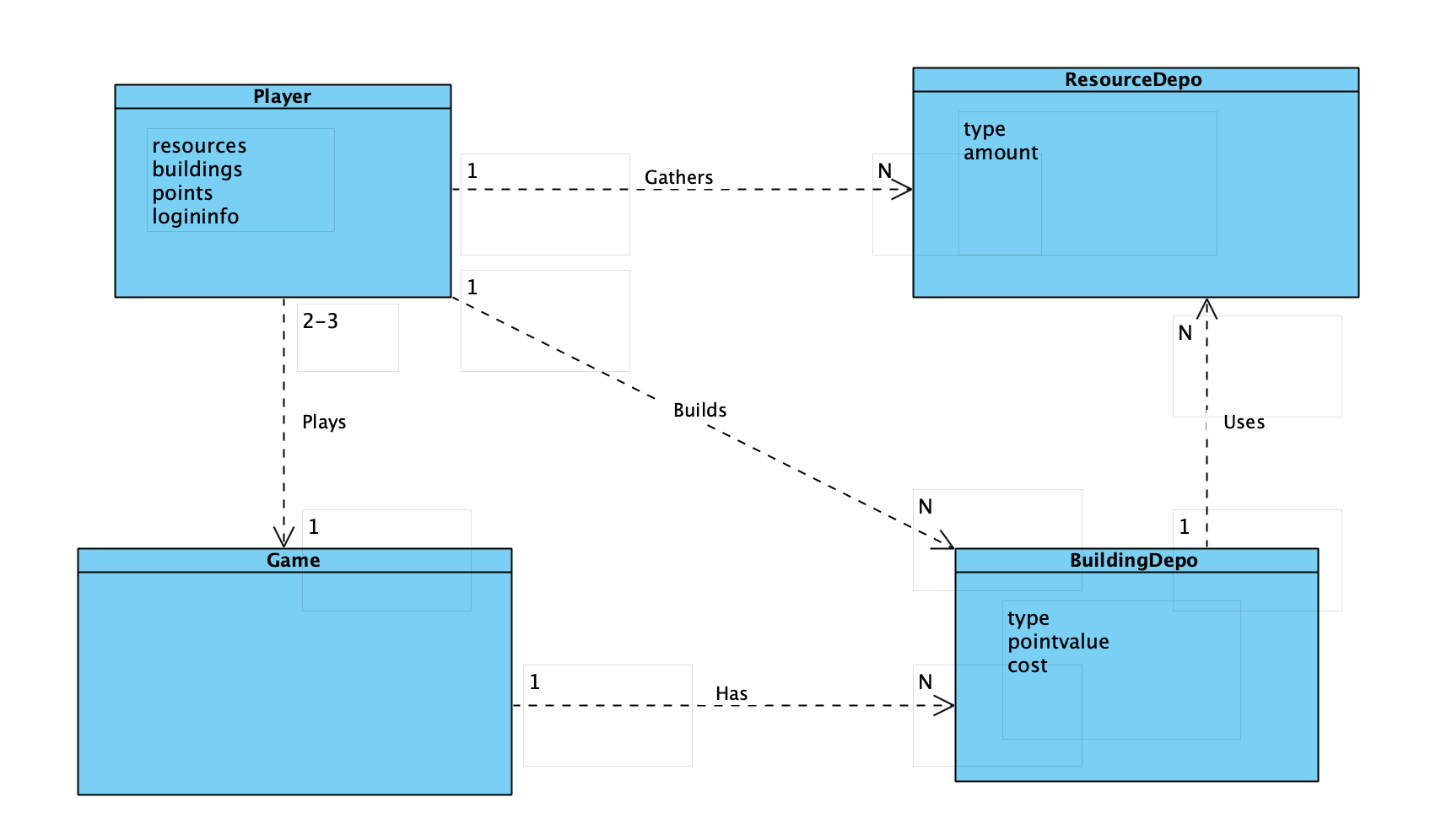
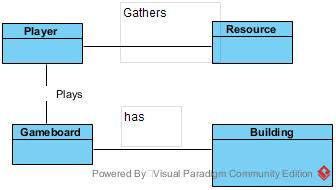
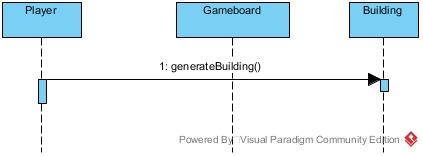
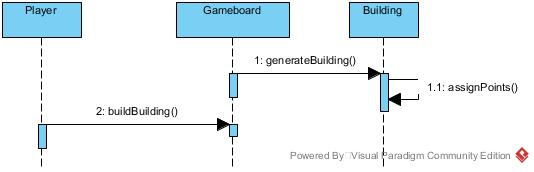
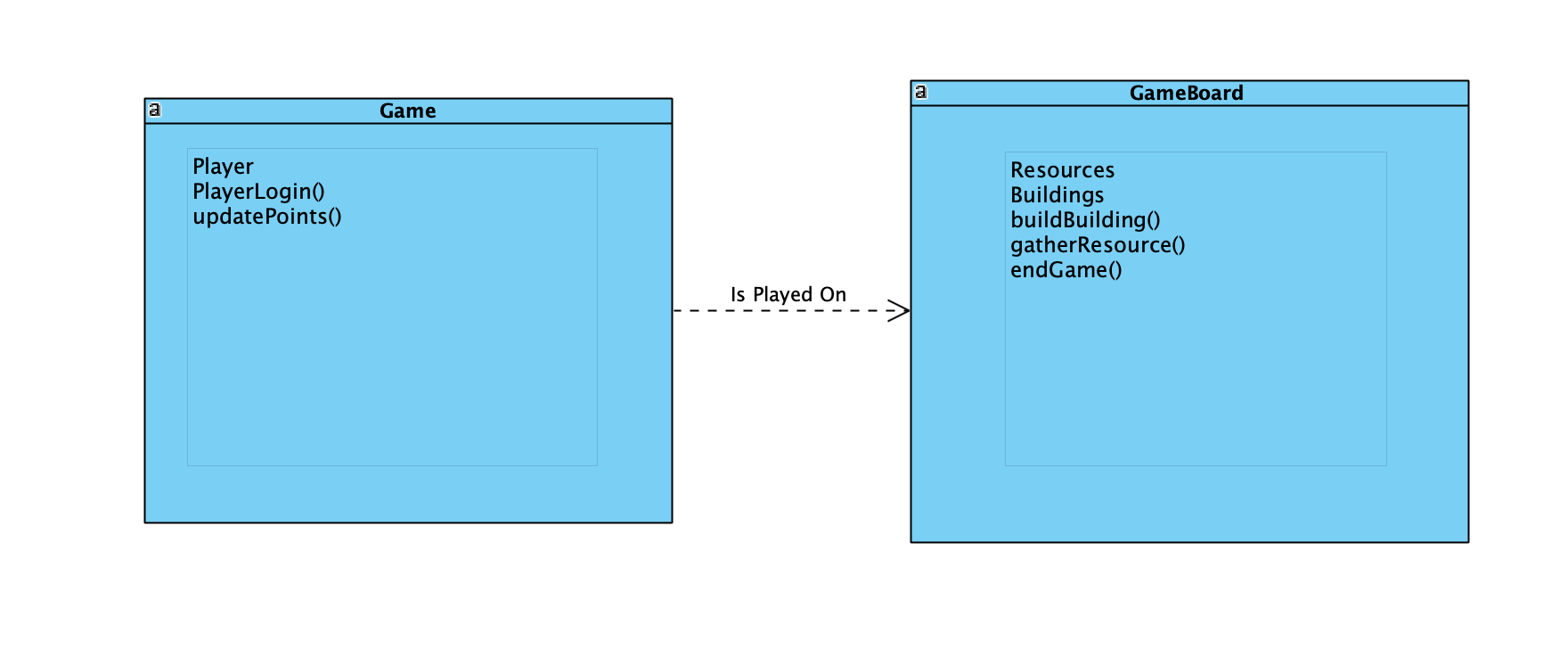
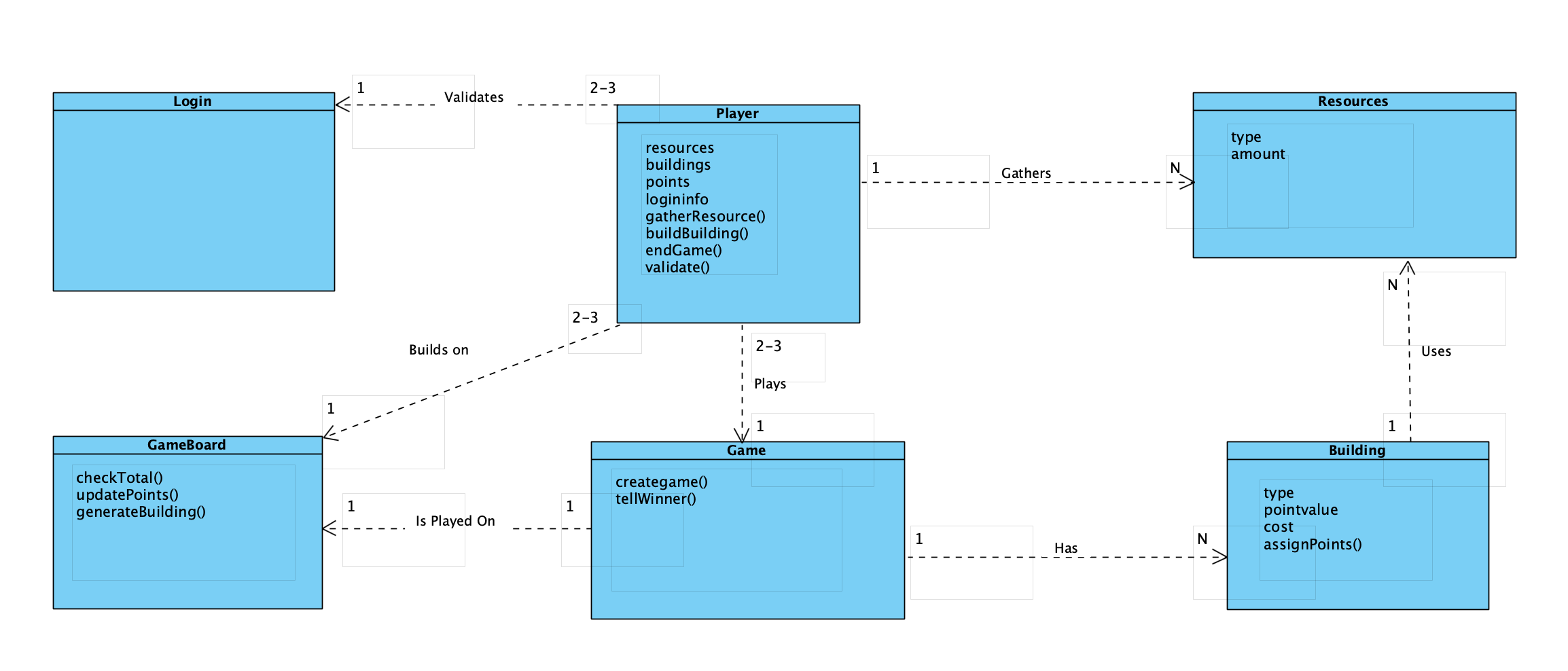
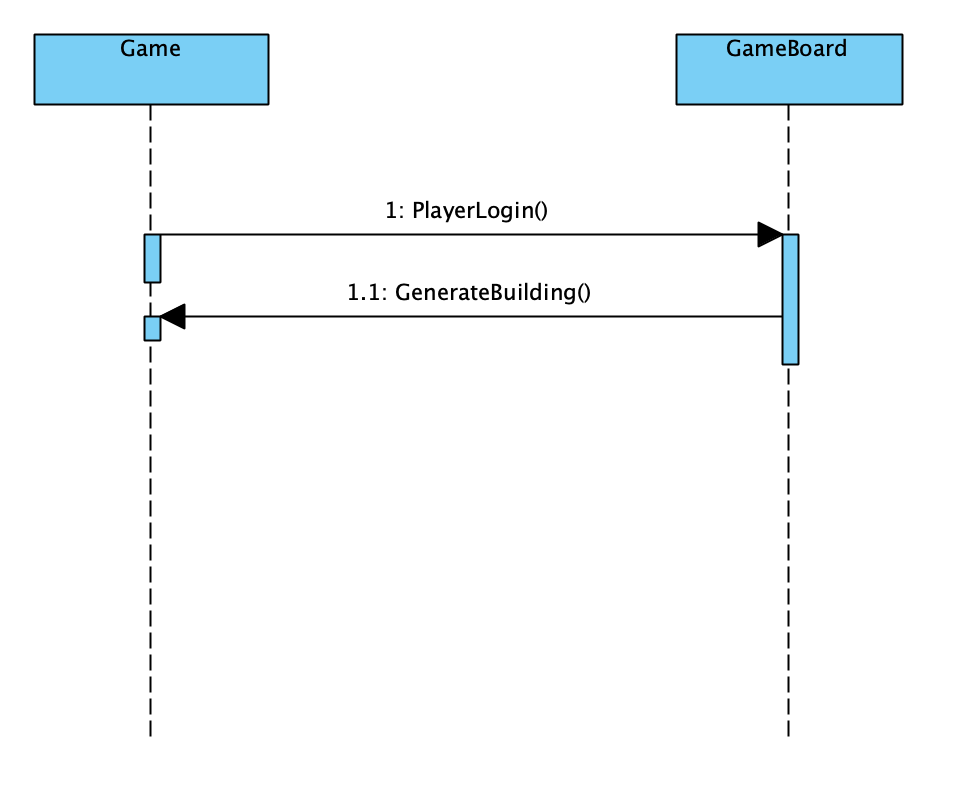
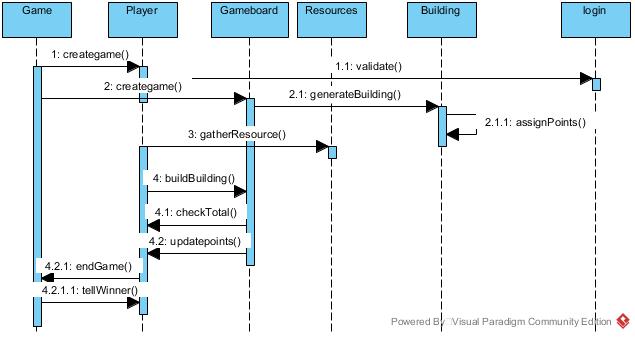
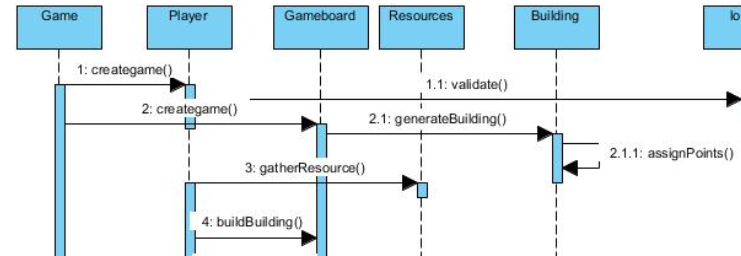
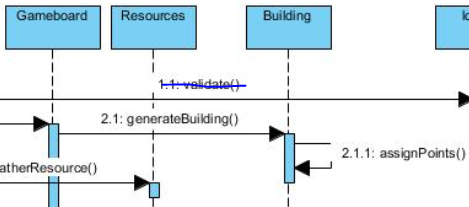
* Decision of Low/High Coupling
  + 
    - Choice 1 (alternative choice)
  + 
    - Choice 2 (selection) (class diagram)
  + 
    - Choice 1 (alternative) (SSD)
  + 
    - Choice 2 (selection) (SSD)
  + Originally the decision was to have the player interact with the individual buildings (choice 1). We decided this would be far too much coupling because of the static design of the buildings (their point values and “score” don’t change during runtime). We could have done the same for resources, but depending on where a player is in the game, they will have to gather a different amount of resources. So now the decision is to make the gameboard create the building and have its values stored so the player only interacts with the gameboard and the resources (choice 2).
* Decision of High Cohesion
  + 
    - Choice 1 (alternative choice)
  + 
    - Choice 2 (Selection)
  + 
    - Choice 1 (alternative choice) (SSD)
  + 
    - Choice 2 (Selection) (SSD)
  + Originally we had designed our system to have a gameboard that handled all of the individual interactions on the board. It contained the resources, player, and building classes and functions. We realized that this would be far too much for one class to handle, so we split it into the respective classes. This is an example of High cohesion because these new classes do much less but do a very specific thing. While this did make higher coupling, it made more sense in our system to split the actions.
* Decision of Information Expert

Figure 1.3

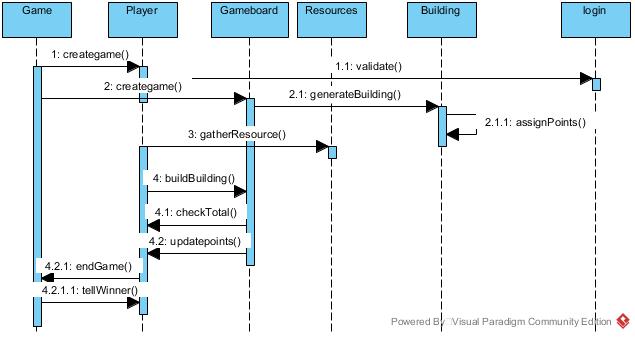
Choice 1 (Sequence)



Choice 2 (Class Diagram)



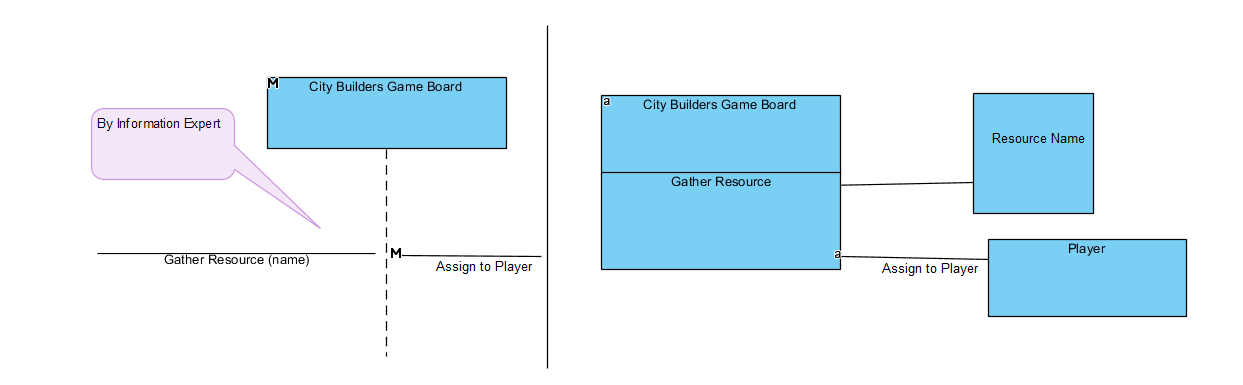
* Originally the responsibilities for each object pertaining to resources, buildings, and points were assigned following a sequence of events according to what the player wants to do with it. Meaning, that the game acknowledges the responsibility for the object whenever the player performs an action and only allows if game parameters allow it to be fully achieved. However, in this class diagram, we went for the option to break up every responsibility for each object as its own class entity. Therefore the information expert for the resources, and buildings are assigned to the game board instead of the player.



* + The information expert assigns the basic responsibilities for all the objects in the game. To do that, it will assign a responsibility to the classes that has the information needed to fulfill those jobs. As for example, a problem would be assigning a job for the resources. The class that has the info needed to fulfill that responsibility is the building class. The information expert assigns the resources (as shown in figure 1.3) for the responsibility to assign the resources to be used to build the building.
  + The information expert is also assigning a responsibility to the player class, once it first assigns a responsibility to the gameboard to gather resources for the player. If the player wants to gather a resource only the game board has the information needed to fulfill the request for the resources. The information expert assigns the gameboard to the job of retrieving the resources, and using it on the players command to gather resources. The class object for the resources are not filled with the information to be assigned to a players inventory. As shown in the figure above (1.3), the player commands for “gather resource”, which passes through the gameboard class that has the responsibility to grab the resource, and assign it.

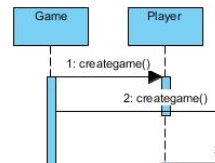
**APPLYING EXPERT**

* A responsibility for an object needs the info to be fulfilled. Since the resources for an object such as resources is used to be assigned to a player, and assigned to a specific building, the gameboard assigns the appropriate responsibility for that resource object. In this case we retrieve the resource for the action of the player gathering a resource. The information expert (game board) assigns the responsibility for the object to be assigned and defined as a given resource name and assigned to the player.

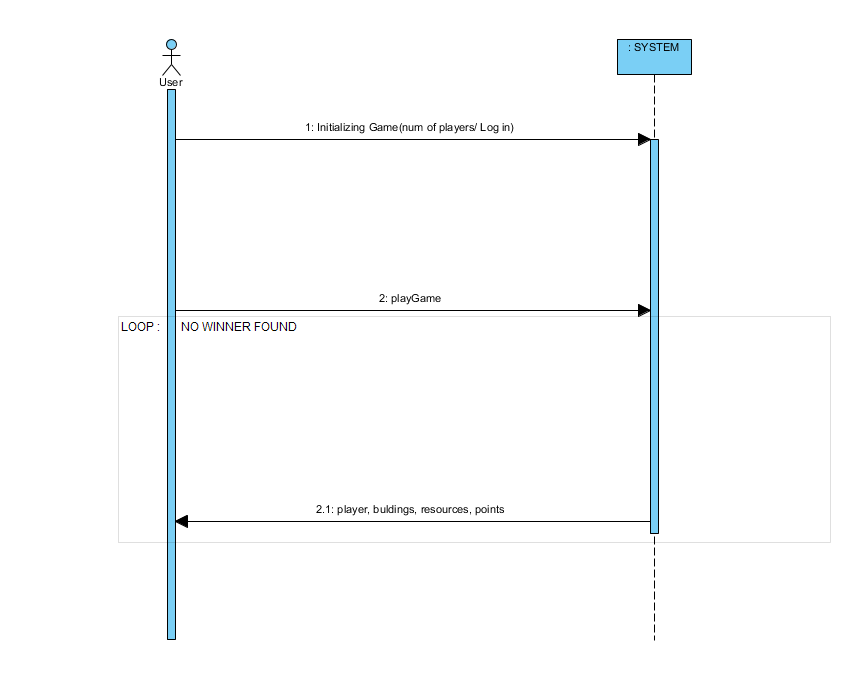
****

* Decision of Controller GRASP principal

Choice 1 (Class Diagram)

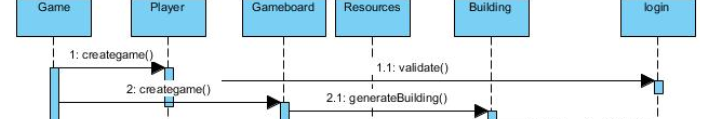


* Game class controller holds responsibility to create game board for player to play in.

(option 2: root object System)

* + What first object beyond the UI layer receives and coordinates a system operation? The answer is the game assigns responsibility to either the overall system, or running software, or it assigns a responsibility to the use case of administrator starting the game.
  + City Builders as a game running on a laptop, desktop, phone, or tablet represents the device that runs as the software’s system operator. This controller assigns the responsibility to the games functioning objects such as administrator overseeing the game, and player logging in to play game. The system the game runs on upholds the responsibility the task is kept alive.

Choice 2 Sequence (SSD)

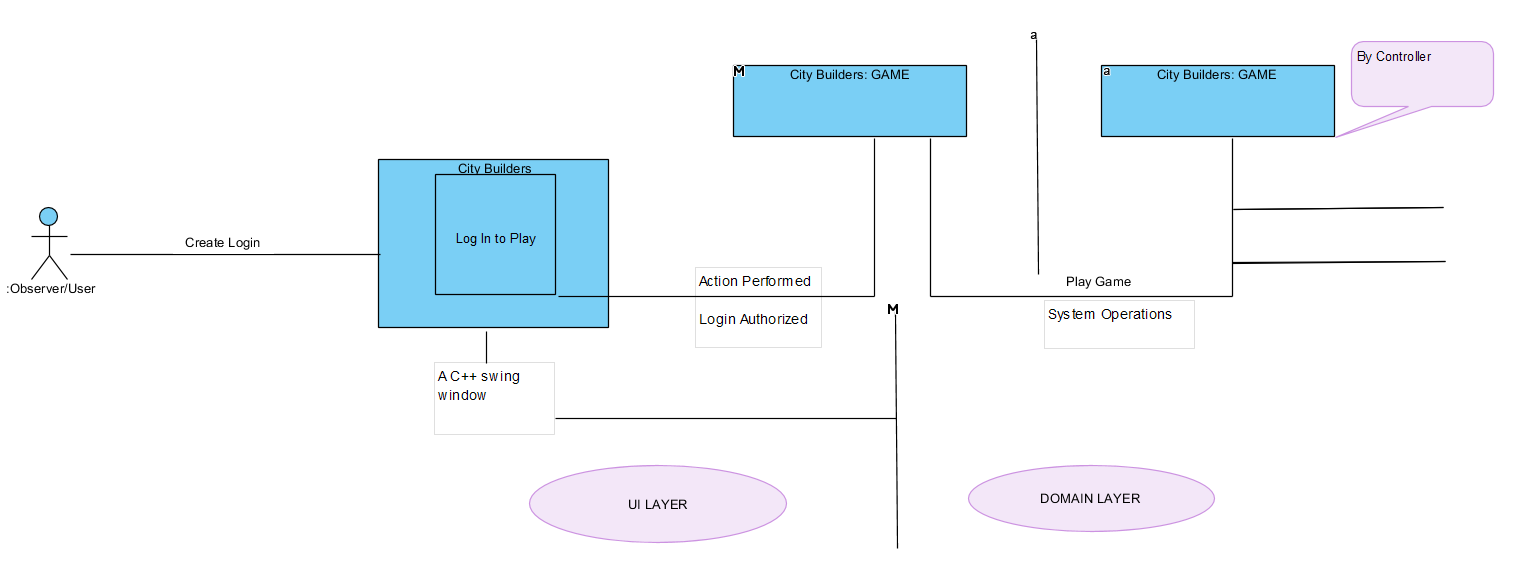


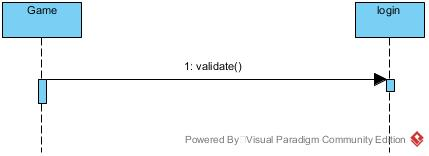
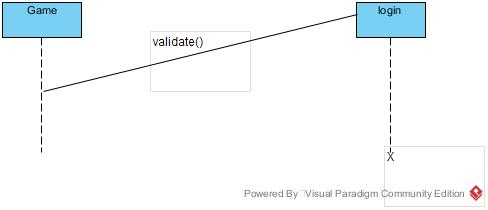
(Option 2: Session Controller) Chosen Scenario

Session Controller

* Represents the use case, or session controller. When player activates login, and start up a game, the systems use case “startGame” occurs within the City Builders game. The figure below represents the connection between the UI layer to the domain layer of software objects, and how the controller pattern is applied to City Builders.
* We ended up choosing this principle within the controller GRASP principles due to having the separation of classes, and the controller only dealing with the Domain layer of the game, rather than scenario 1 where the system that the game runs on controlling the game. With the users having an identifiable account, it is not up to the system to decide the conditions of the game for that player. Every user logged in has its own controller session that takes part in the City Builders game class within the domain layer.

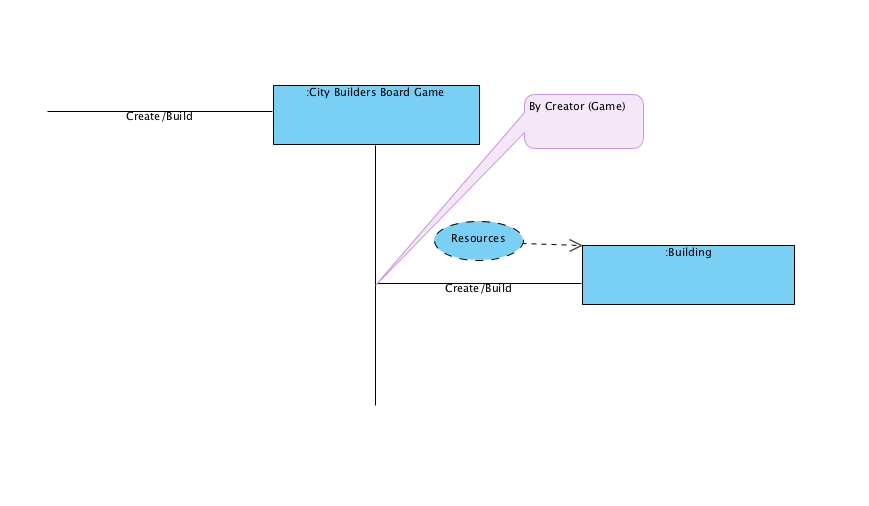
Applying Controller Pattern using City Builders: Game



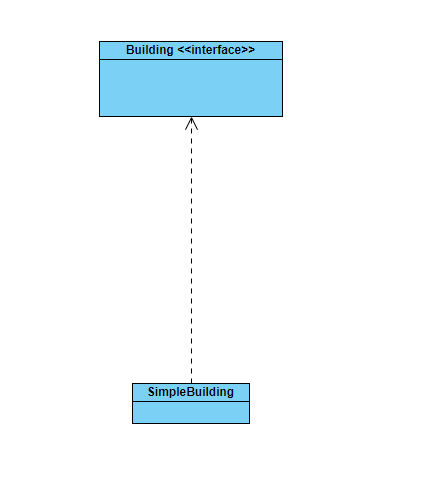
* Creator
  + 
    - Choice 1 (alternative choice) (class diagram)
  + 
* In choice 1 we have to ask who creates the instance of the login, and what creates the game once a user is logged in. What is the responsibilities assigned, and who does the game utilize to operate.
  + The Solution to this problem is when the User creates a login it creates a vessel that the system can assign a responsibility for that user. When the game is loaded the creator needs a logged in player, so the user creates a login so the system can administer an assignment to the user to play the game. This instance occurs when one of these situations becomes true:
    - * Game requires players
      * System certifies player
      * System has information and data assigned to game to incorporate user.
    - Choice 2 (selected choice) (class diagram)
  + 
    - In our choice 2, selected choice, the game simply requires a validates user. The only step for the creator is the system to select a verified user and implement them into the game.
    - Choice 1 (alternative choice) (ssd)
  + 
    - Choice 2 (selected choice) (ssd)
  + We have another choice to create any amount of users for one player to play against and those extra users being used by the system known as CPU. We decided to scrap this system in favor of having live users being able to make decisions on their own, rather than manipulating the game to be aware of the users actions whenever they are in their turn.
* APPLYING CREATOR
  + We apply Creator by looking for software objects that satisfy the role of playing the game, or the system assigning a game. When looking at the figures above for our choices, the common theme between them is not having a designed software object, the user, to assign to the game. Without it, the role of the system doesn’t start to starting the game. Within the game, the creator assigns responsibility to the individual domain model objects such as resources, and buildings.

Applying the Creator pattern in a dynamic Model

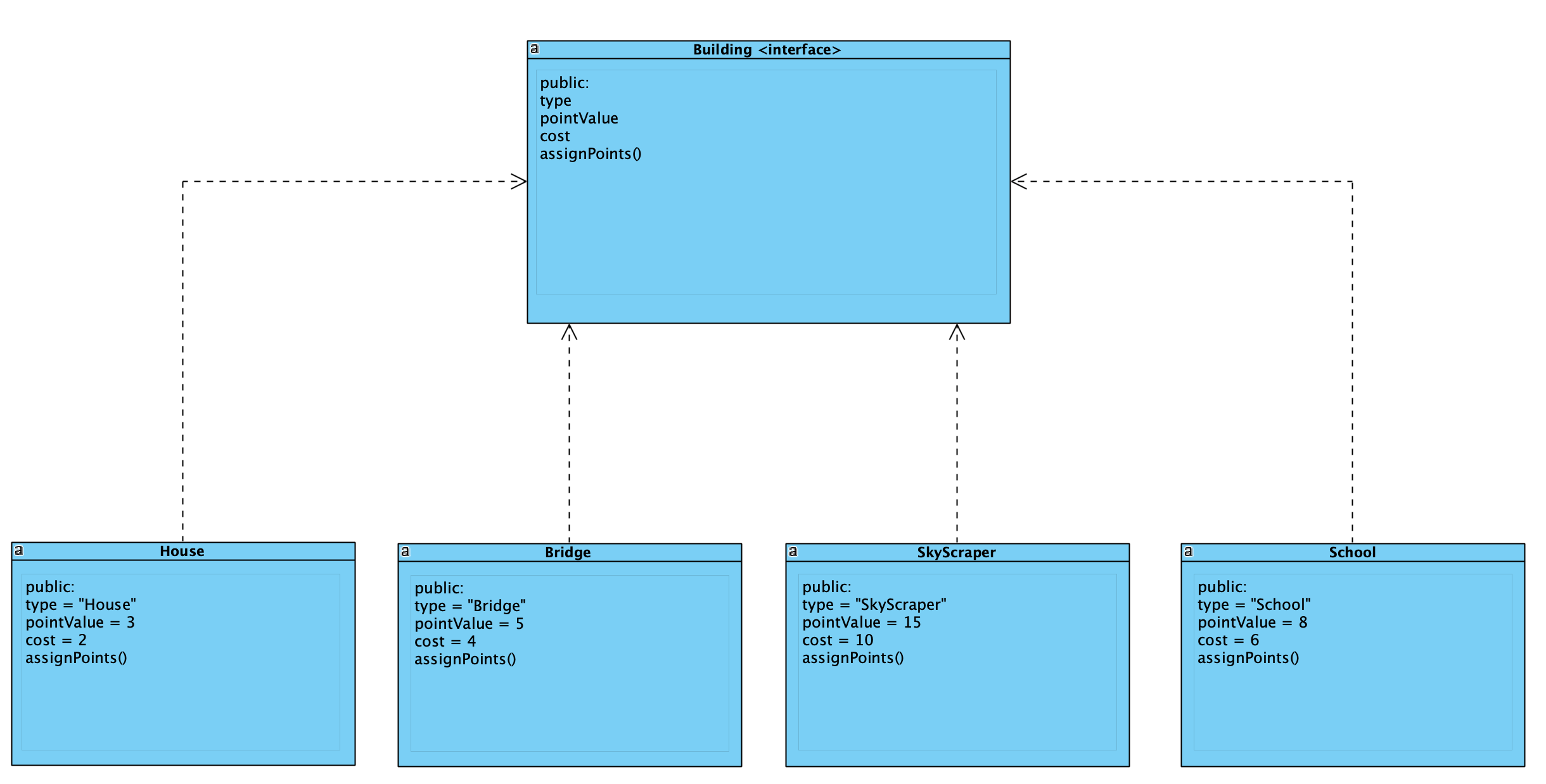
* + In the figure below, when the game world is created, the player has an action to modify the board by adding buildings onto it. When the player calls for the action to build a structure, the board is modified by the creator that activates the responsibility of the board to add the structure, dependent on resources.



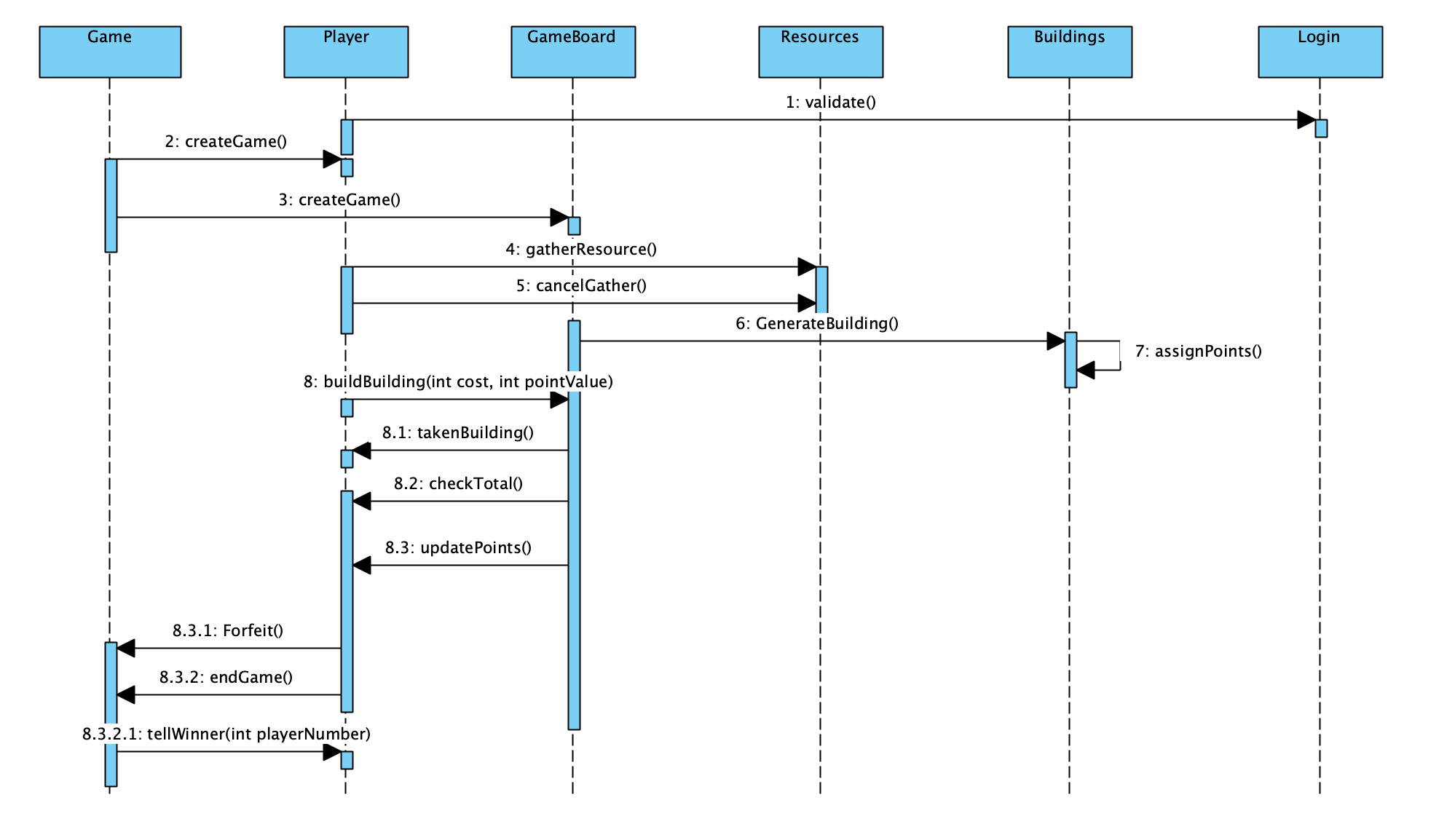
* Polymorphism Pattern (REVISION)
  + Diagrams
    - Choice 1 Class Diagram (Alternative)



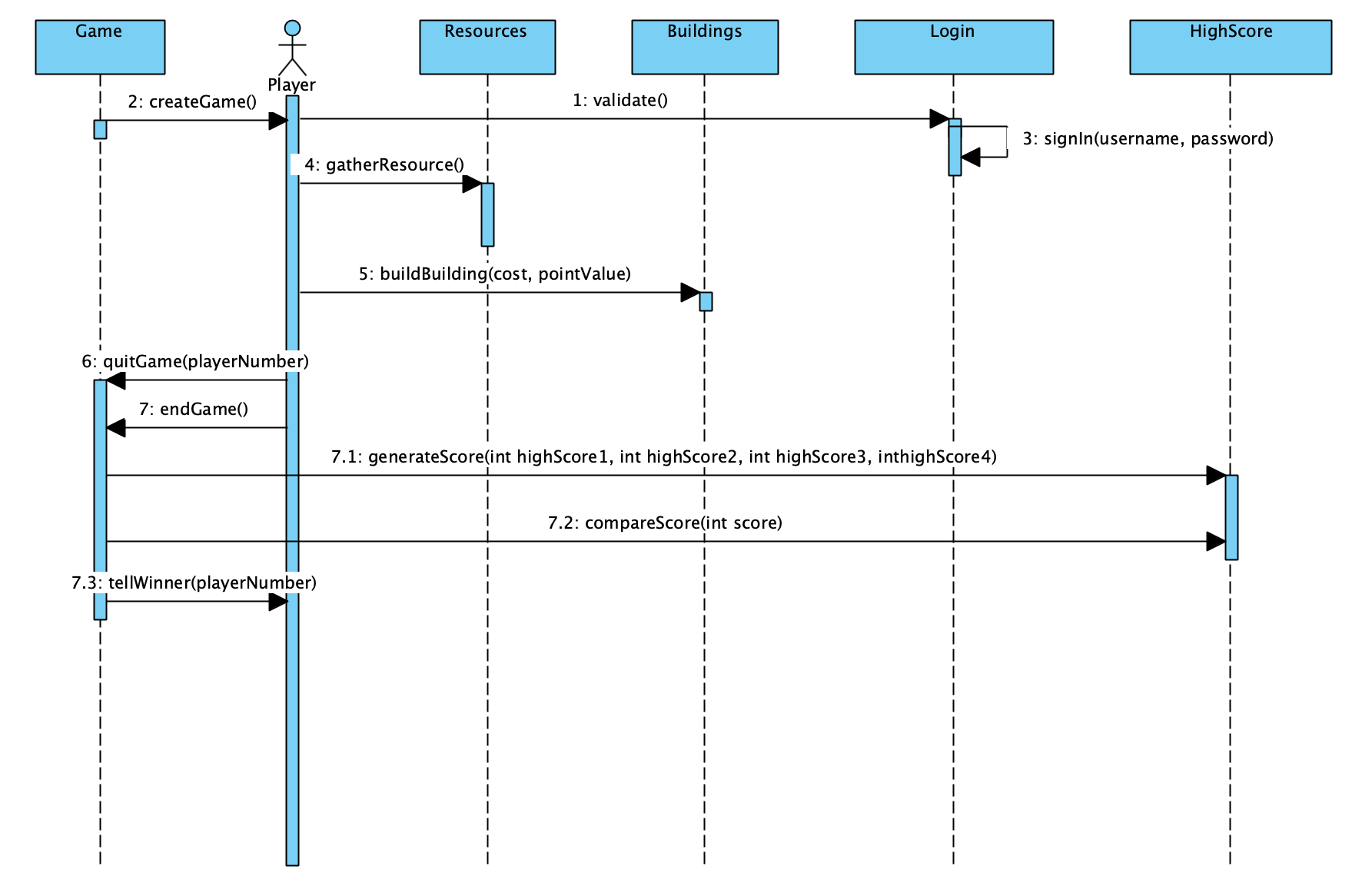
* + - Choice 2 Class Diagram (Selection)



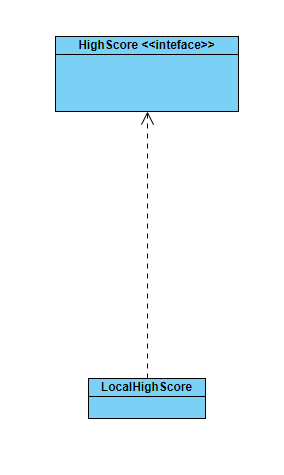
* + - Choice 1 SSD (Alternative)



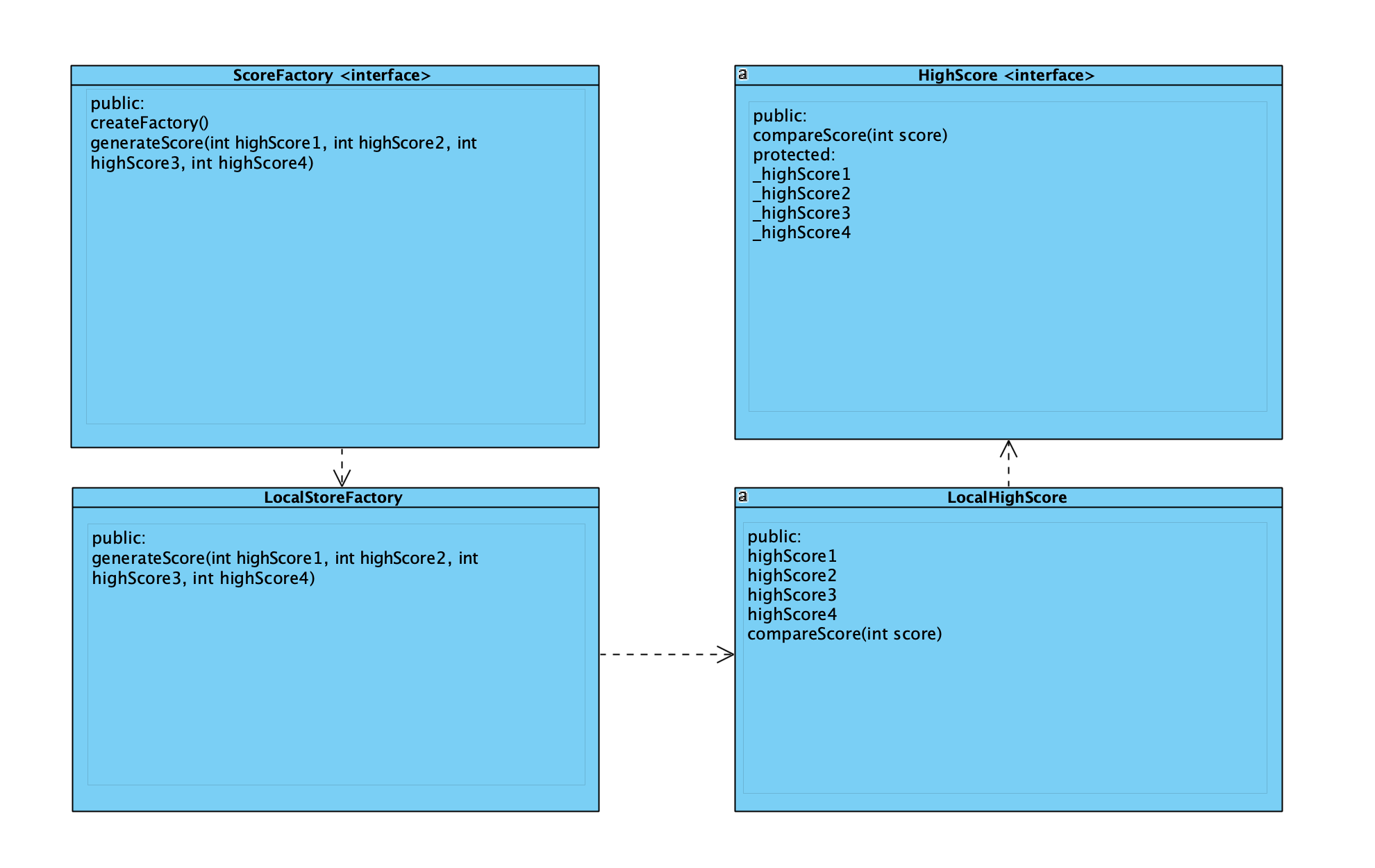
* + - Choice 2 SSD (Selection)



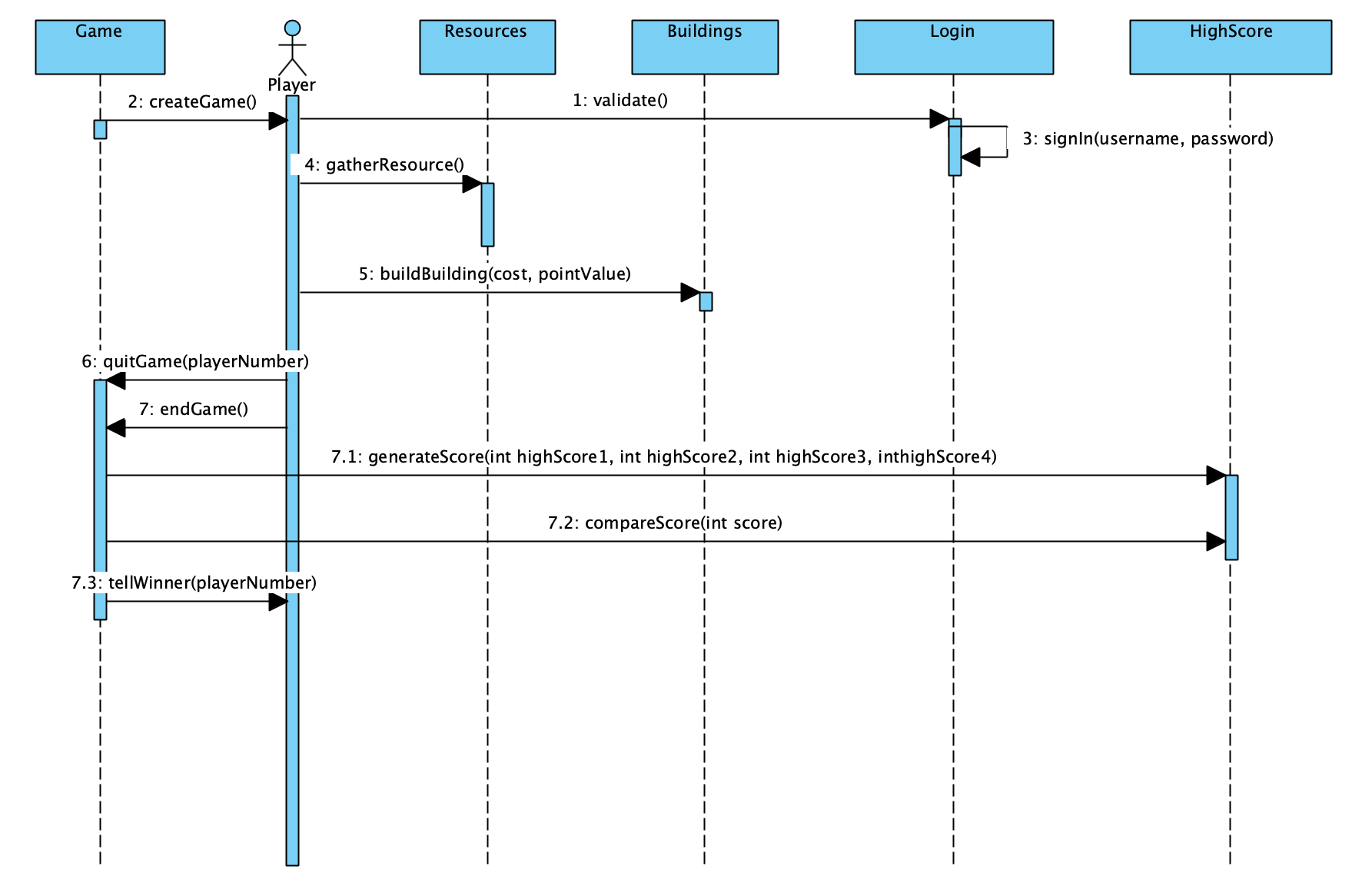
* + Explanation
    - In the original version of our system, we had decided to use only a simple building class as an instance our our building interface (choice 1). However, we quickly realized that as our system changed, we would need to have different buildings. So instead of changing how the simple building class worked, we decided to polymorphously implement the building interface through the bridge, house, school, and skyscraper classes (choice 2). This allows us to change the values and properties of the individual buildings within their own class definition without affecting the other buildings or the system as a whole. It also allows us to add or remove buildings from the system with very little change to the rest of the system. Another side effect of this is that the Gameboard class is now obsolete, as we can define how the different buildings will be defined in their own interfaces and classes.
* Protected Variation/ Factory Pattern (REVISION)
  + Diagrams
    - Choice 1 Class Diagram (Alternative)



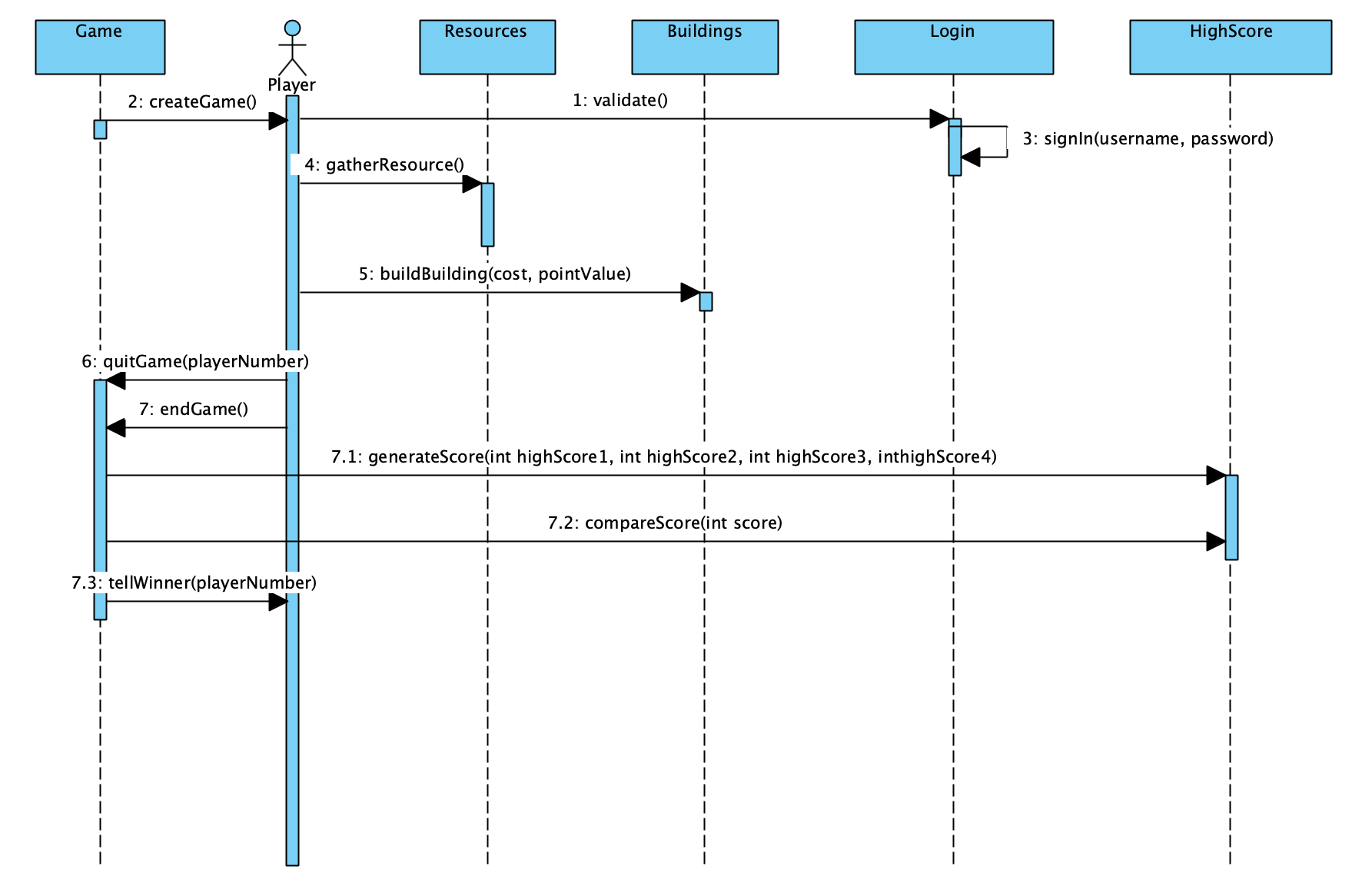
* + - Choice 2 Class Diagram (Selection)



* + - Choice 1 SSD (Alternative)



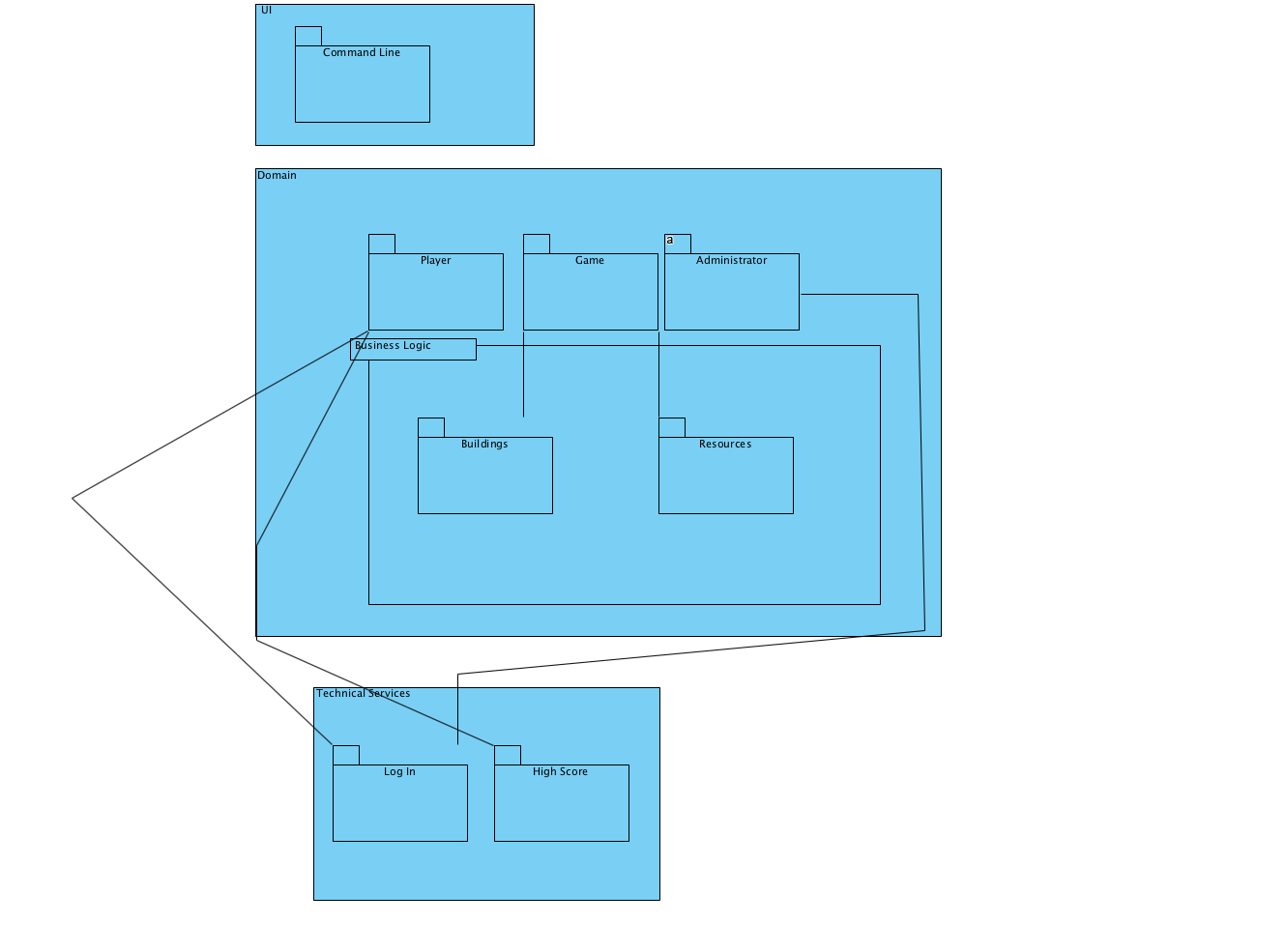
* + - Choice 2 SSD (Selection)



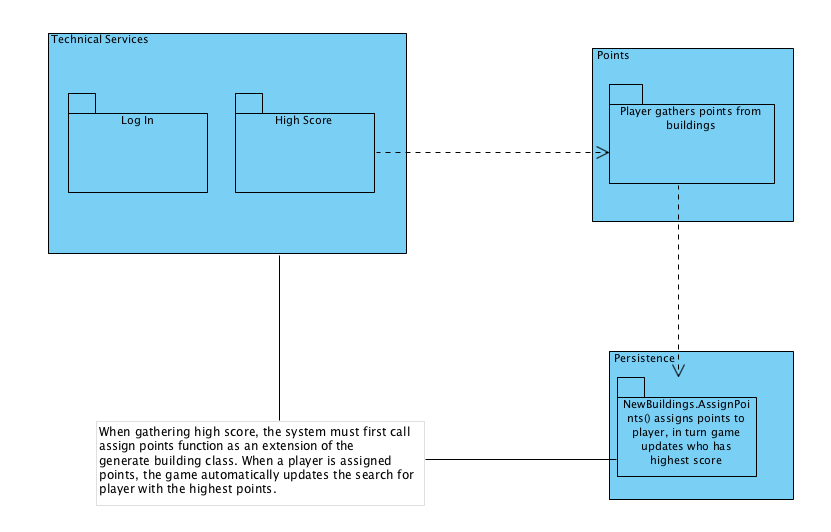
* + Explanation
    - Our primary external system is implemented through the high score interface. It is currently just a stub implementation, but in a future iteration it will allow the system to be able to take high scores from various different places. As it is currently implemented, there are four hard coded high scores that are compared to the winner’s score at the end of the game. If we had chosen to not use a factory pattern (Choice 1), we would have to change multiple sections of code throughout the system when we wanted to alternate what the high scores were or where we would be retrieving them (file or online). However, through the use of the factory pattern (Choice 2), we are able to modify the values for our high scores to be anything we want without having to modify the code anywhere else. This is also useful because we don’t know what kind of properties our final version of LocalScore class will be, but the Score Factory will still be able to create our Local Score class no matter what changes. This means that we do not need to alter the lines of code where the score factory is declared when we want to make a change to local score.

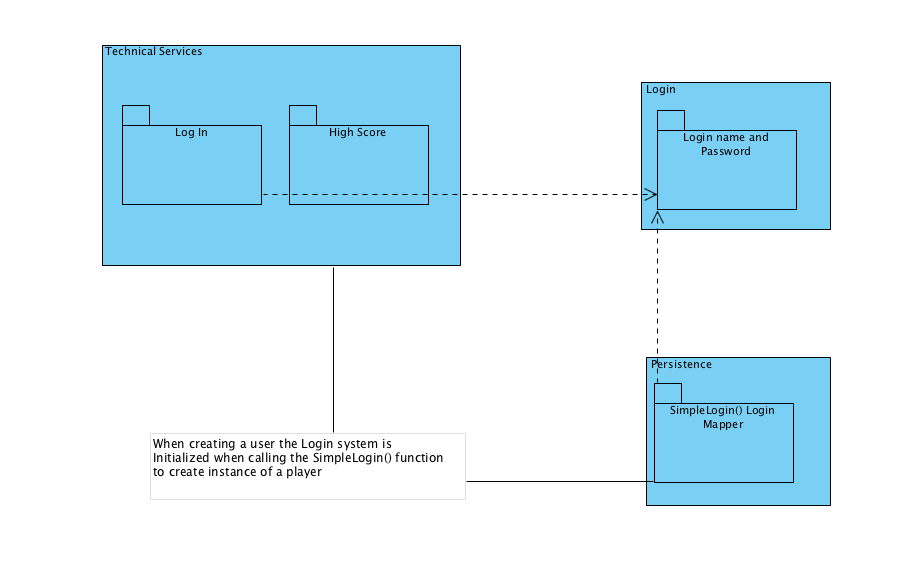
Logical View

Package Diagram of the logical view:

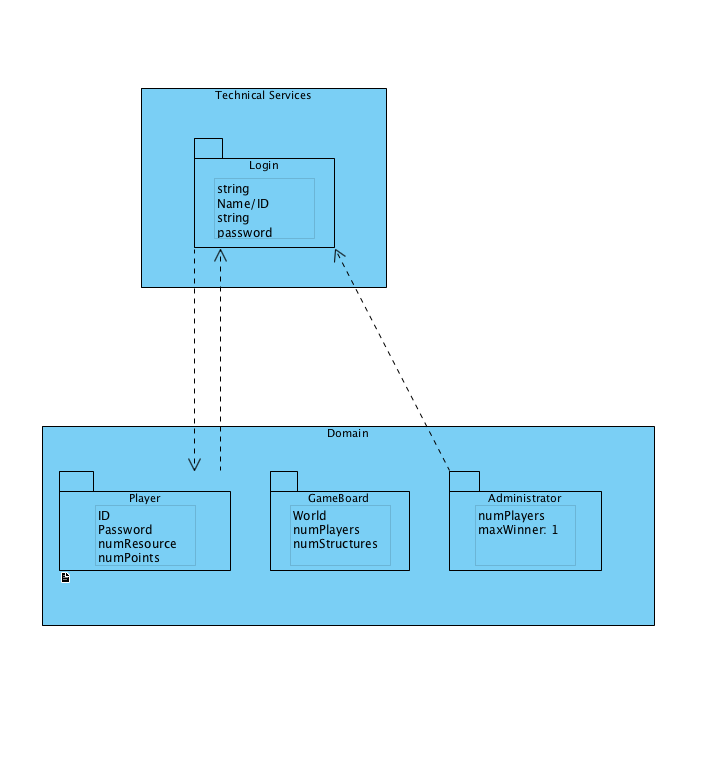


* A classic layered architecture is modeled showcasing the layers of classes and system operations. The primary controller for the use of the creator comes from the game class, which is part of the domain layer that holds the board game, and the player. Both coordinate to act dependant on one another.
* The technical services layer holds the Login service that identifies the user, and the administrator takes that and validates it.
  + Technical Services with Persistence Package and Logging Package





Class Diagram of Domain And Technical Services



* The Domain Layer maps the login class with the player class, as the player provides info for the login. In return the Administrator class validates the login data, after which activates the GameBoard class.
* EXAMPLE CODE:

void createGame()

{

//instance of creator: creates the player classes

SimplePlayer player1;

player1.validate();

//GameBoard currentGameboard;

//currentGameboard.GenerateBuilding();

bool foundWinner= false;

int currentPlayer =0;

bool firstPlayer = true;

int totalPlayers= 3;

char input;

while(!foundWinner) { //loop UI layer code to retrieve action and utilize Domain layer packages}

Function Signatures

* What have we provided is the services provided by the GRASP principles. Our UI output relies on the sequence of the classes as shown in our graphs, such as the the Creator. The classes all depend on the info provided by another class, which in return outputs to the UI of the game board.

|  |  |  |
| --- | --- | --- |
| Date | Phase | Info |
| 10/01/18 | Inception | n/a |
| 10/29/18 | Elaboration 1 | Added Initial SAD |
| 12/10/18 | Elaboration 2 | Updated controller and creator GRASP patterns. Added sections for polymorphism implementation, and added Factor Patterns. |