**INHERITANCE**

Many items in the room implement this concept. The inheritance hierarchy works like this:

Interactable

Inspectable

OptionInspectable

PickableInspectable

CandlePickable

KeyPickable

PotionPickable

Interactable is the abstract base class of anything that will pop a balloon when player goes nearby.

Inspectable inherits Interactable, and is the base class of anything that will pop a footer with description when player goes nearby and interact with it. An example is the digits on the floor.

OptionInspectable inherits Inspectable, and is the base class of anything that will pop a footer and a two-button dialog box when player goes nearby and interact with it. Examples are locked door and exit portal

PickableInspectable inherits OptionInspectable, and is the base class of anything that the player will pick up when chosing OK. Classes which inherit from this are CandlePickable, KeyPickable, and PotionPickable.

**POLYMORPHISM**



Increase key

Increase HP

Increase Light

The most obvious example of this is the way that CandlePickable, KeyPickable, and PotionPickable each overrides the method OnPicked of its PickableInspectable base class to provide specific action when it is picked. The dialog button needs not to know which pickable it is dealing with, as it just calls OnPicked when the player clicks it, and the correct behavior will be performed.

**ENCAPSULATION**

Consider this piece of code from PlayerController class:

public enum eInputMode{ MainGameplay,Interacting,Freeze }

private eInputMode inputMode = eInputMode.MainGameplay;

public eInputMode InputMode{

get{ return inputMode; }

set{

if(inputMode != value){

inputMode = value;

...

if(!bPause){

applyInputMode();}

}

}

}

In this game, there are several input modes:

* MainGameplay mode: player can look around, walk, and interacting with items laying around.
* Interacting mode: player cannot look around or walk, but can continue to press interacting button to interact with appearing UI.
* Freeze mode: player cannot do anything. This is used for cutscenes.

PlayerController class keeps track of this input mode in the enum variable named “inputMode”. Depending on player’s action, inputMode may need to change during gameplay. However, switching input mode also requires some work such as disabling player movement and Camera movement code. This work is done via “applyInputMode()” function. PlayerController should make sure that this code is run when input mode is changed.

This is done by not disclosing inputMode as a public variable, but keeping it private and exposing it via InputMode property instead. Outsider can query its state via get accessor, and also set it via set accessor. The set accessor will take responsibility to ensure that “applyInputMode()” is called when appropriate.

Note: I use the convention where variable names start with lowercase and property names start with uppercase letter, which is due to my old habit and is probably not the best.

**ABSTRACTION**

In this game, all the rooms are not created beforehand but only spawn when needed. Spawning a room is a very complicated action, involving Instantiating a prefab, creating floor digits and coloring them according to game logic, checking whether the room has been visited before and place items at the position they used to be if it has, or deciding whether and where to place them according to the predetermined probability table if it has not, and many more. In fact, if you take all the code involved in spawning the room, it will comprise quite a large chunk of the manger class of the main gameplay scene.

However, due to the nature of self-generating map, spawning room is the operation that is done frequently. Hence, the operation is crafted into one function that can be called over and over (in SceneMainManager class):

private GameObject spawnRoom(Vector3 vPosition,int[] aDigit,List<RoomItemData> lgItem)

Even though it is used only within this manager class (hence private), the benefit of abstracting this operation into a function is evident. Callers need not care how the room gets spawn; they just call this function and know that, regardless of how it is done, they will get a room spawned correctly according to their provided function arguments. Furthermore, since spawning room is a complex operation, there were a lot of bugs found in the development stage. When that happens, we only need to direct our attention to this specific function because all codes that spawn the room calls this function. This reduces bug hunt and make the code much more organized than before.