Deadwood™

Yulo Leake

Feburary 27th, 2015

Introduction

The world would be a lackluster puddle if it were not for the infamous Deadwood. This board game lights up the day by letting its players live the dream – making single digit money starring in a multi-million dollar movie! Yes, an actor!

Purpose

The purpose of this document is to organize the logical side of the Deadwood™ board game. It will tie together individual logics into coherent pictures that represent various use cases of the game.

Scope

The scope of this diagram is the logic side of the Deadwood™ board game. The Controller and View (such as User IO, buttons, and graphics) are out of the scope of this document.

Organization

The organization of this document is divided into two parts. First part focuses on the UML diagram and the second part on the sequence diagram focusing on how the logic designs supports each use cases.

UML Diagram

Figure 1 shows all the classes that govern the logical side of the board game.

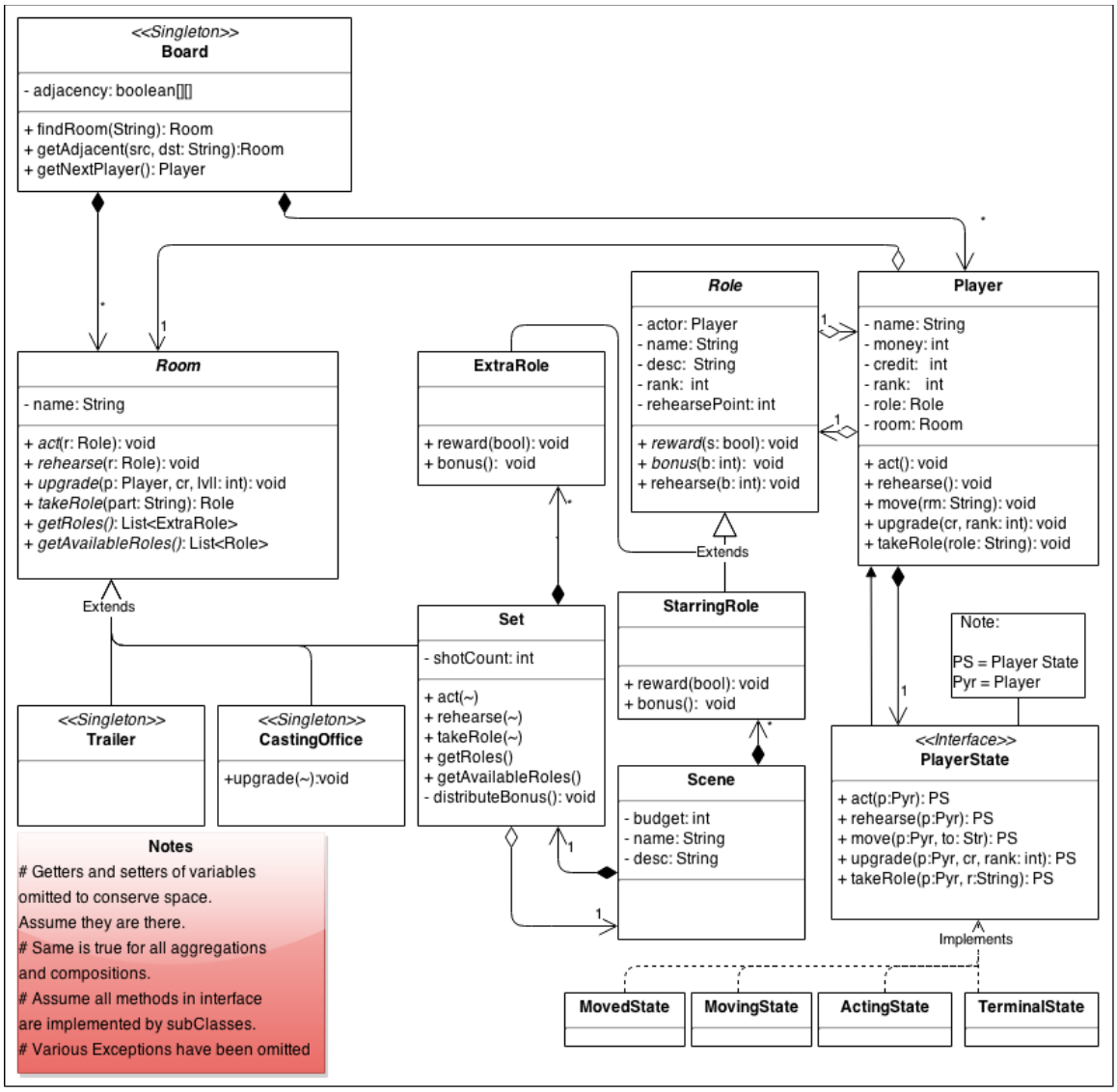

Figure 1 – model classes

Board and Table

The Board and the Table classes are responsible for keeping the game logic running, cycling through the players

and acting as a bridge between the View and Controller side of the game.

Room Classes

The Room class is an abstract class that is extended by various Room subclasses that builds up the overall Board. Each subclass interacts with the Players differently. The Trailer has no real interactions while the CastingOffice governs the upgrade logic for the Player. The Set contains all the logic for Players to act, rehearse, and taking Roles.

Player Class and Interfaces

The Player class stores the data that each user tracks (such as money, credits, and rank). Each Player object is composed of PlayerState, which is a Finite State Machine, and keeps track and limits Player action to make sure he/she follows the rule.

Roles

Roles are divided into two parts: Starring and Extra. The decision to split was made to make implementation easy. Depending on which role the Player has, the implementation of reward and bonuses are different.

Scene

Scene class keeps record of the budget (difficulty of each scene) and whether or not it is revealed.

Sequence Diagram

Move

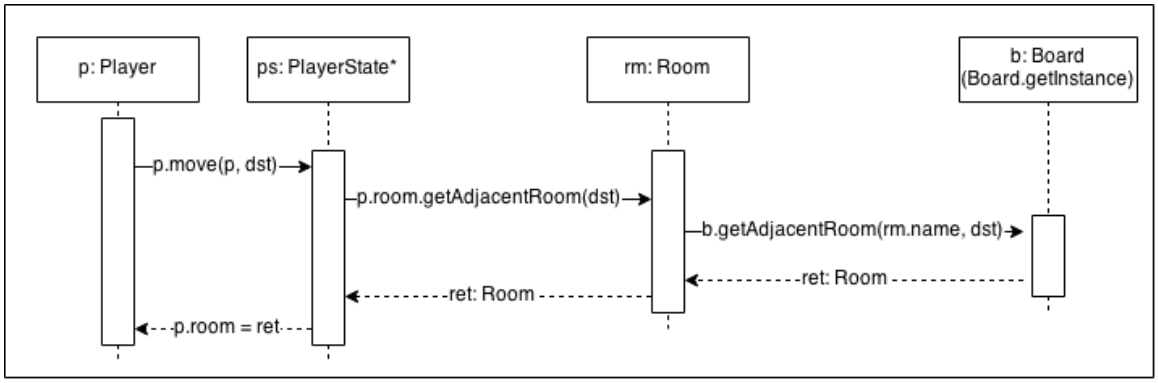

Figure 2: Sequence diagram for Move use case

Move is evoked by the user with “move *dst*”. It will then move the player to *dst* if adjacent to current player room. It will throw an exception if PlayerState is anything but MovingState. It will also throw an exception if *dst* is NOT adjacent to current room or is not available in the game.

Upgrade

Upgrade is evoked by the user with “upgrade *cr rank*” typed into the console. *Cr* is the currency type (supported are “$” and “cr,” while *rank* is the rank to upgrade to [2,6]. It will throw an exception if PlayerState is in Action or Terminal State. Room will throw an exception if it is anything other than CastingOffice. It will further throw an exception if provided *cr* or *rank* are not supported, if player tries to lower its rank, or if player has insufficient amount of currency to upgrade.

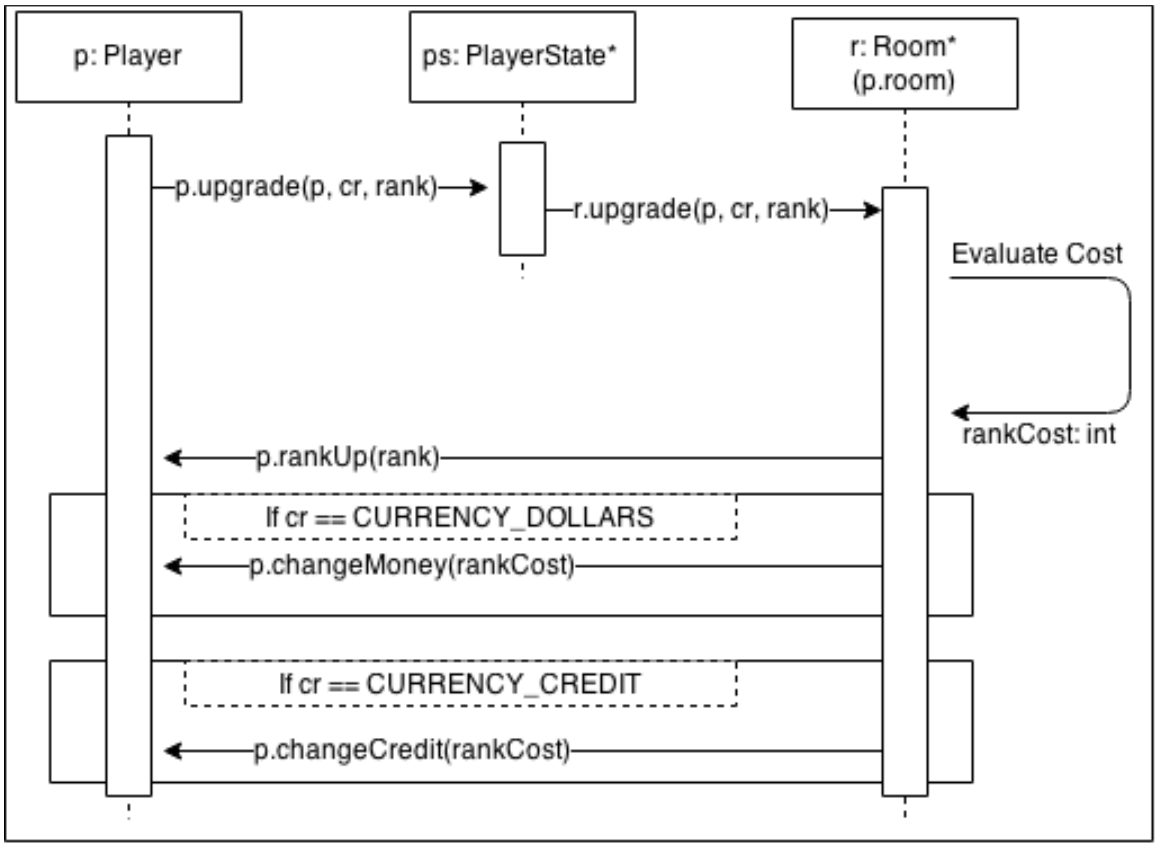

Figure 3: Sequence diagram for Upgrade use case

Take Role

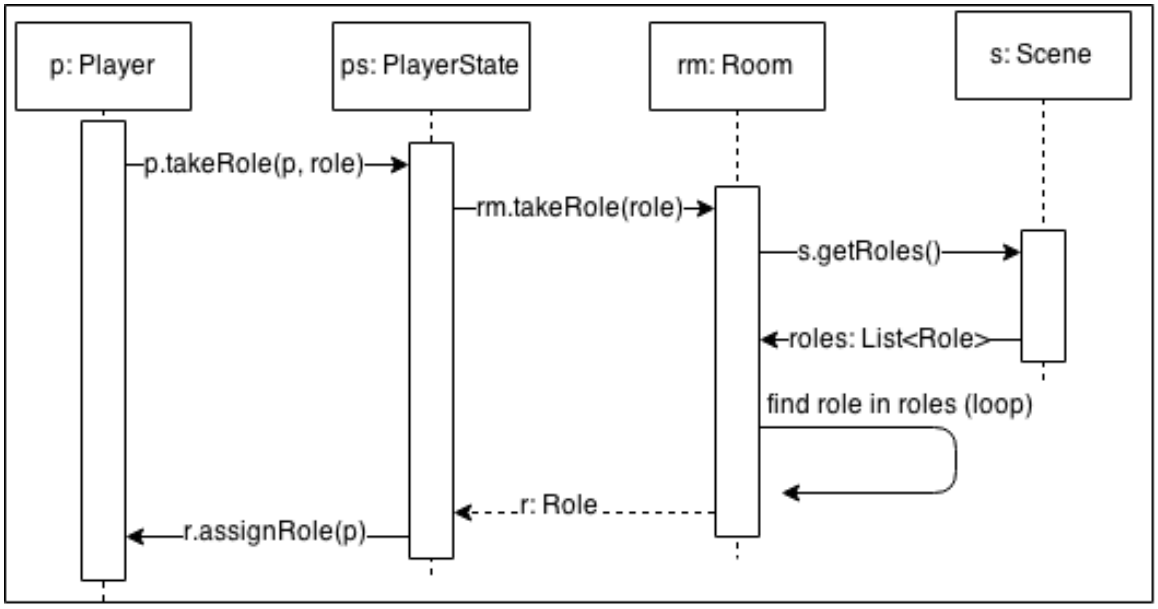

Figure 4: Sequence diagram for Take Role use case

Take role is evoked by the user with “work *role*” typed into the console. *Role* is the String name of Role the user wishes to take up on. PlayerState will thrown an exception if it is anything other than Moving or MovedState. Room will thrown an exception if it is not a Set. A Set will throw an exception if the scene is null (indicating the scene has wrapped), if *role* is taken by another player, or if *role* is not found. Role will then thrown an exception if the user cannot take the role due to insufficient rank requirement.

Act

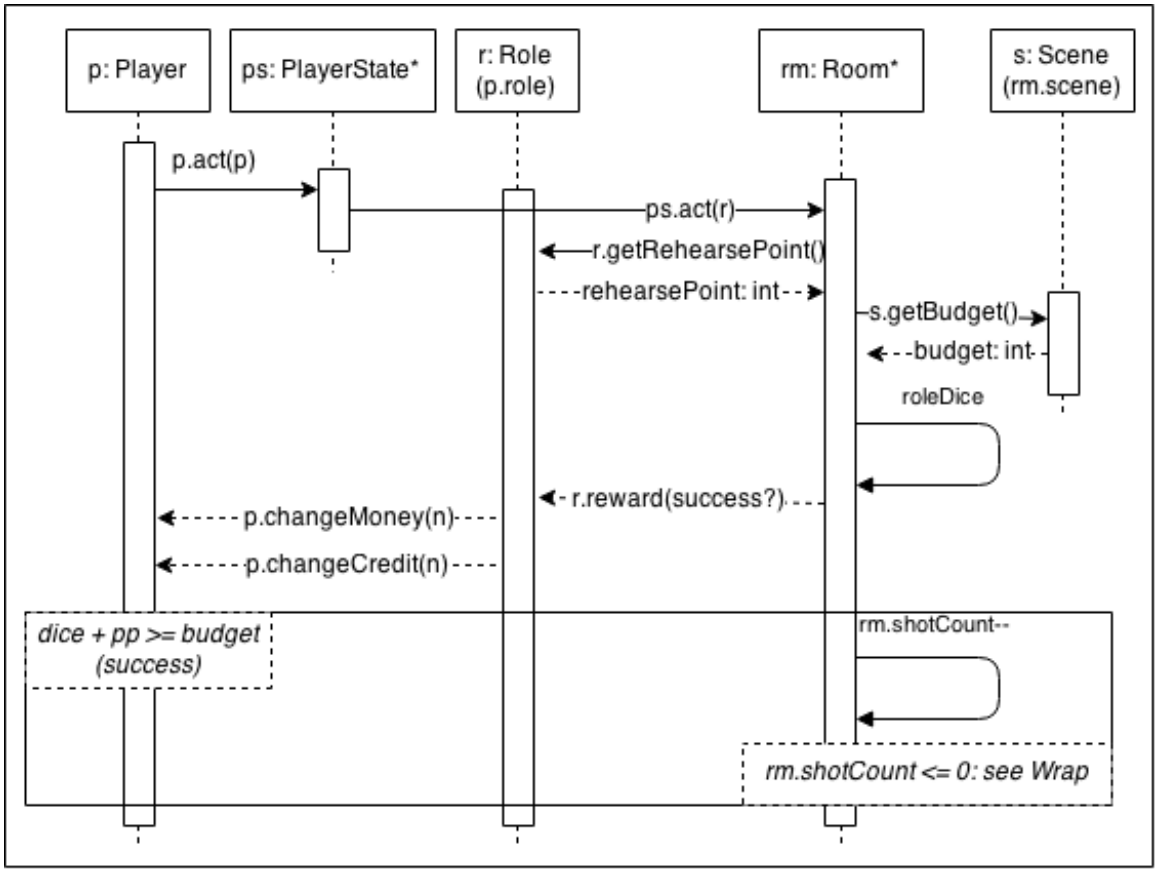

Figure 5: Sequence diagram for Act use case

Act is evoked by user passing “act” into console. The Deadwood class will then call player.act(). The reward method is called no matter what. Each Role subclasses (StarringRole & ExtraRole) implements their own rewarding logic, based on Boolean parameter (whether or not the roll succeeded or not). A “success” is check on *dice* + *rehearsePoint* >= *budget*. PlayerState will throw an exception if it is not in ActionState. Room will throw and exception if it is not a Set subclass.

Rehearse

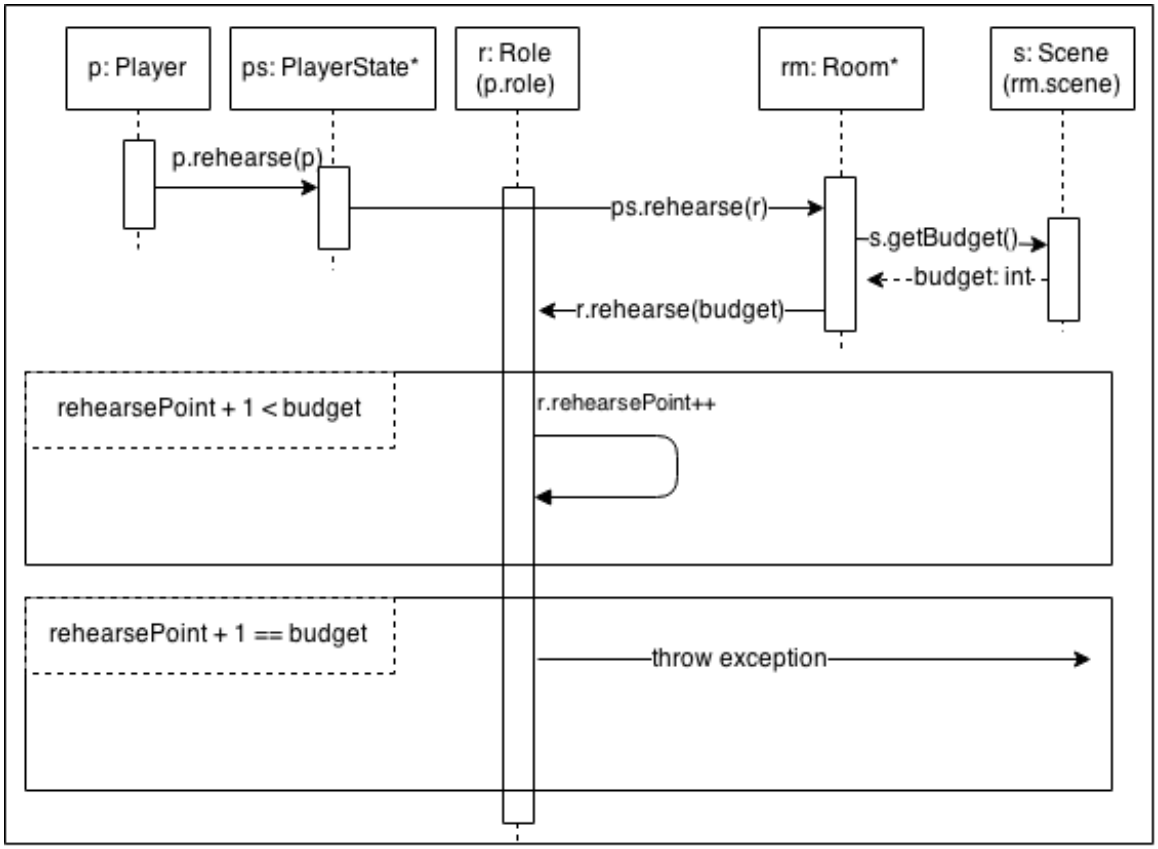

Figure 6: Sequence diagram for Rehearse use case

The rehearse use case makes sure that the rehearsePoint does not exceed the budget -1 of the scene. If

it does, it will throw an exception. Similarly, PlayerState will throw an exception if it's out of ActionState.

Wrap

Wrap is achieved when there is no more shot count in the room. Before it distributes the bonuses, it makes sure that there is at least one person in the starring role. It will first distribute bonuses to Starring Roles, then to Extra Roles.

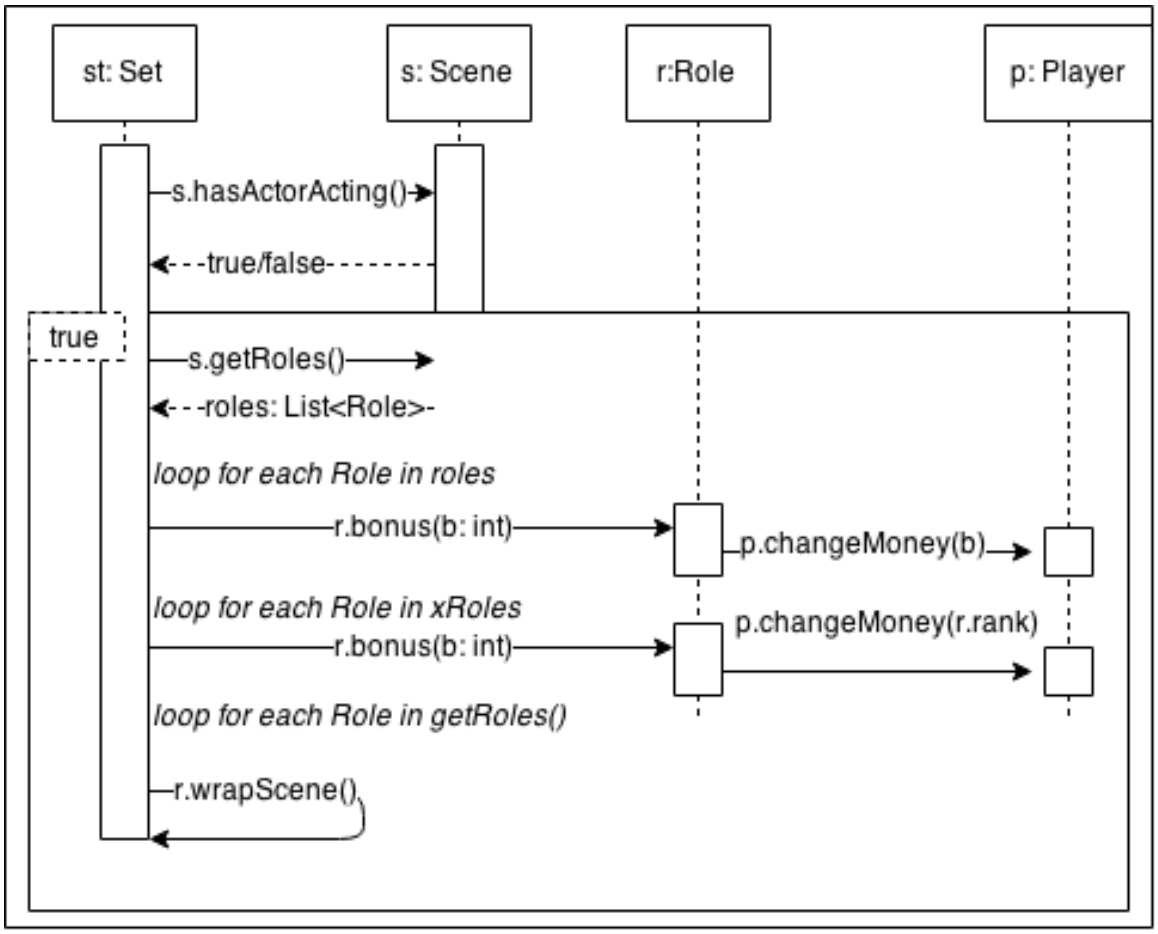

Figure 7: Sequence diagram for Wrap use case