Qualitative Case study: Is the GRASP approach helpful for creating a software design?

Jordan McDonald – 40063974

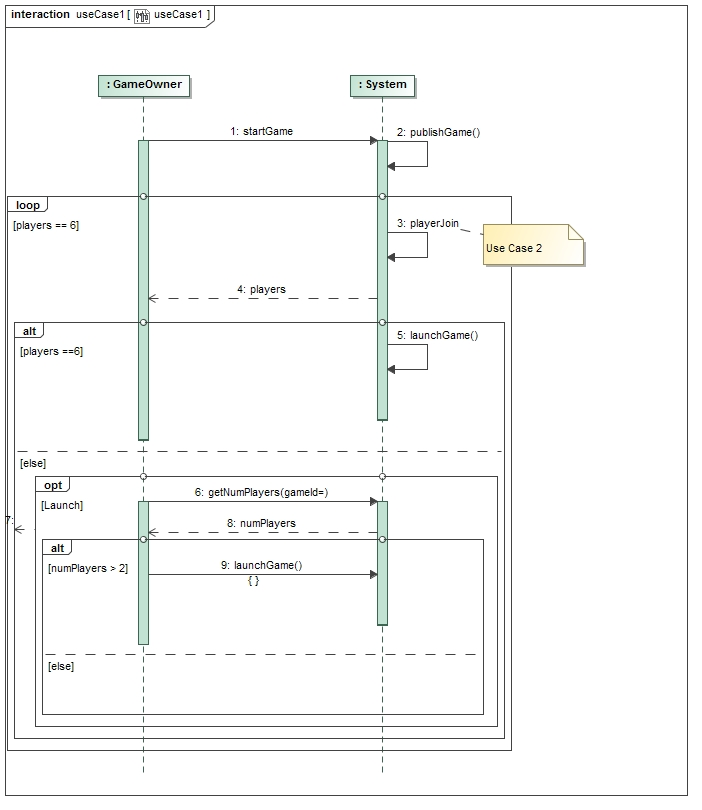
1. **Introduction**

Software modelling techniques such as UML has been utilised for many years in order to assist software developers in planning and realisation source code from a UML model. UML diagrams can take many forms, in this case study the sequence diagram will be evaluated using a qualitative approach based on the opinions of a single subject. The goal is to determine if modelling techniques can be applied to a series of use cases revolving around the board game ‘Risk’ while applying GRASP techniques to determine if they assist the development process. The results show that applying sequence diagrams show a systems processed were very useful and helped apply sound programming principles and patterns at the planning stage.

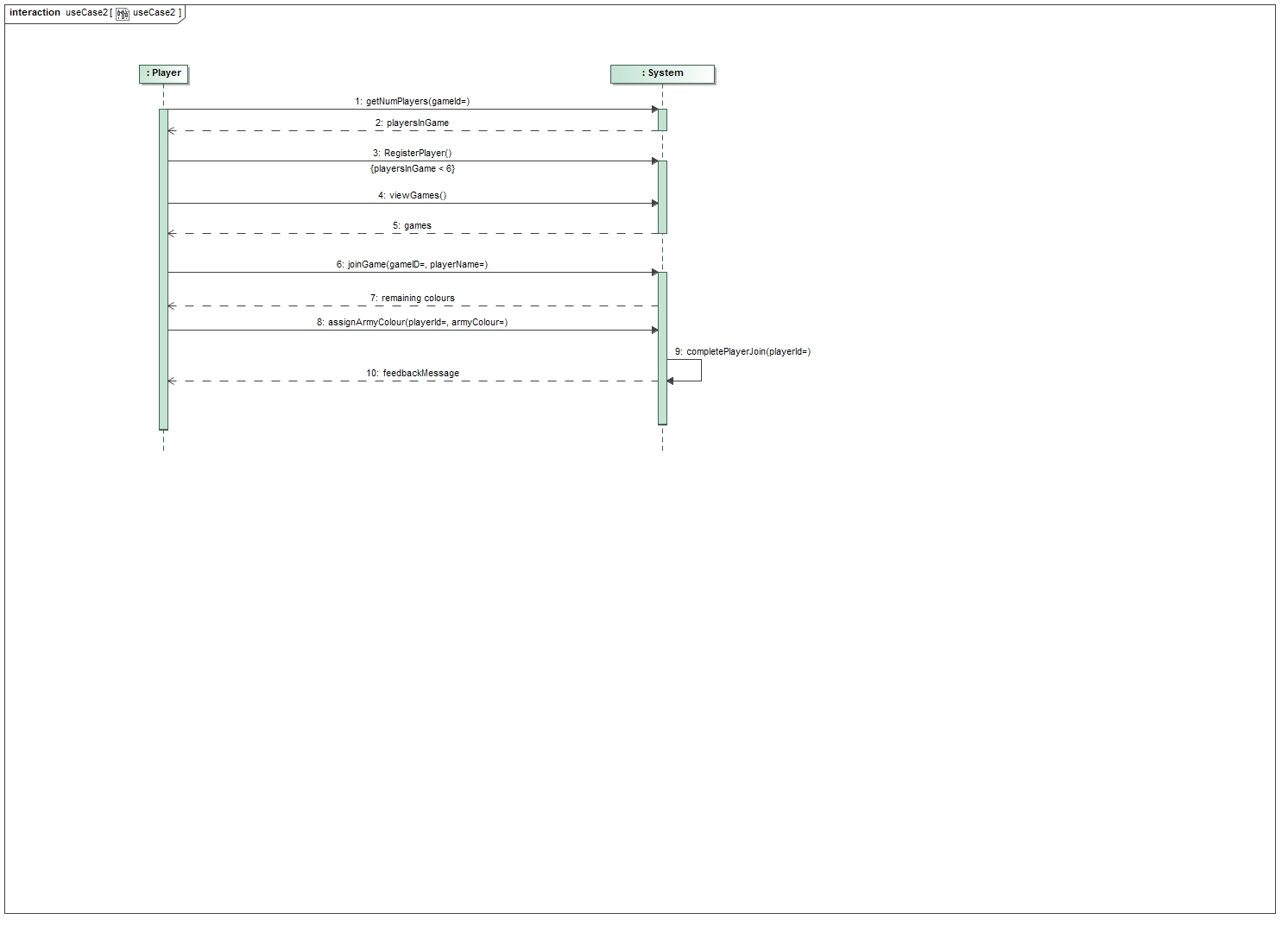
**RQ1** - Is the GRASP approach helpful for creating a software design?

1. **Sequence Diagram Screenshots**

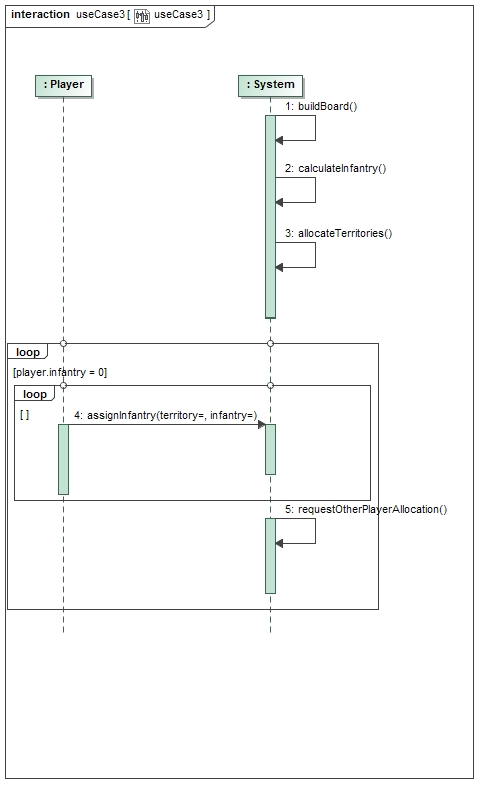
UC1)



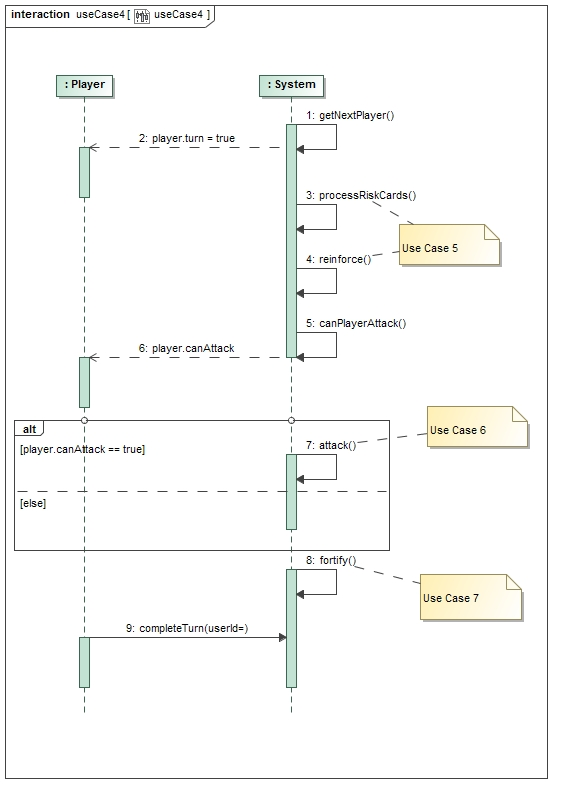
UC2)



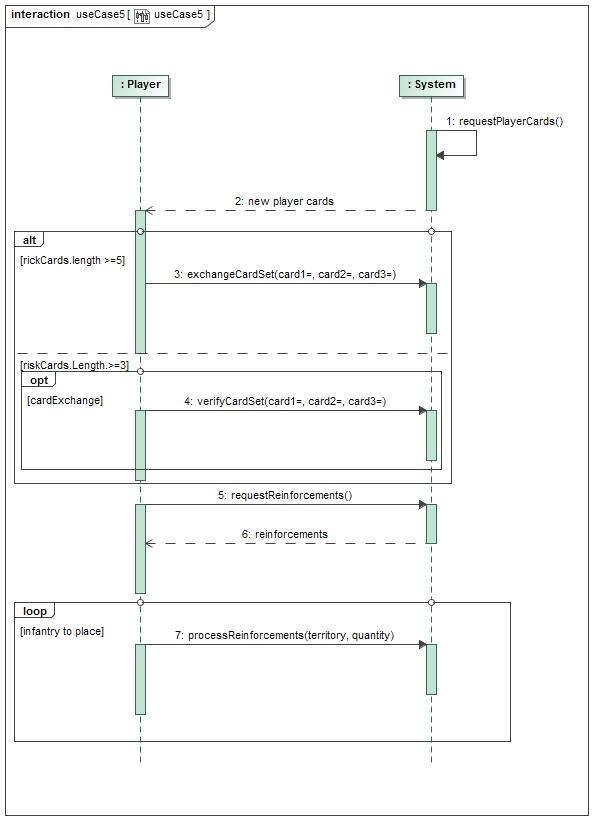
UC3)



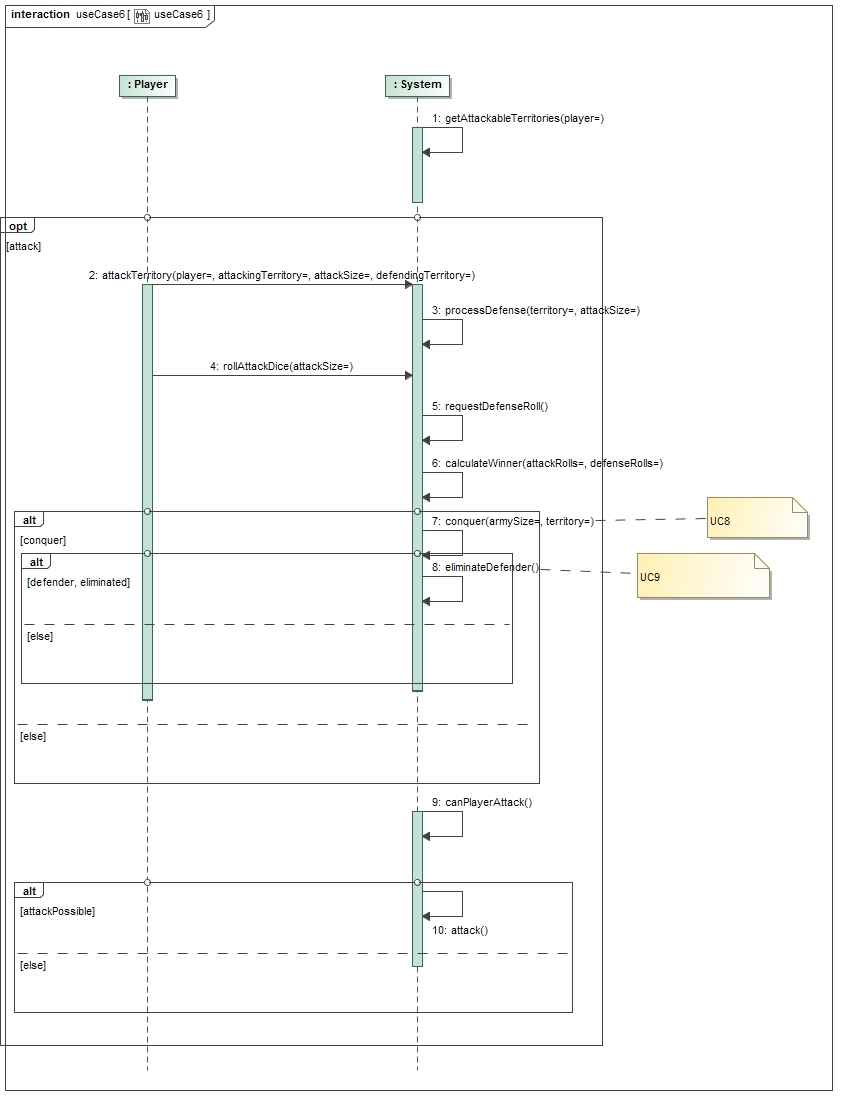
UC4)



UC5)



UC6)



1. **Results**

Applying the grasp principles to realise case studies in the form of sequence diagrams has proven useful, as the subject I have found an awareness of grasp as promoted better planning before developing the diagrams and has resulted in a sufficient representation of code. In order to present this finding I will consider the grasp principles of controller, creator and expert and how they have been applied to the sequence diagrams and help design more manageable code.

**Creator** – The creator example at a stretch is only loosely enforced in the system, player object creating the system is valid when we consider the rule of ‘B closely uses A’ which is true in this context. This is reinforced when we consider that player and system ae going to be linked anyway, so it is not a bad choice, sequence diagram one shows an opportunity to initialise the system , when the ‘startGame’ function is called. It is also a possibility in this situation to consider implementing a initialisation object to create the ‘GameOwner’ and ‘System’. Also implementing a mediator that bridges the gap between the user input and the system could also be advised, but that is out of the scope of this assignment.

**Controller** – In this system we can see that there is no defined controller, this is clear when we consider that on many occasions the player directly polls the system for data, showing that this is the model/domain, for example when ‘getNumPlayers’ is called or ‘requestPlayerCards’ in sequence diagram six. This could have been applied by introducing a controller object possibly named ‘systemController’ (how inventive) which processes the users input and then delegates to the model for data collection, storage or further processing. The model would then return the data to the controller if required and then through the controller to the player, basically this controller handles the use case when the player interacts with the system. This could have had many benefits such as increased opportunities for reuse and pluggable interfaces and opportunity to reason about the state of a use case, these allow low coupling and gives high cohesion.

**Expert** – We can see instances where this is applied on a multiple occasions, for example to find out information about the current game we can see in diagram 1 the player polling the system for the number of players in the game (‘getNumPlayers’). This shows that the system object is the expert in this scenario and is utilised to perform or get information on numerous actions in the system, in use diagram 4 the ‘getNextUser’ method and in diagram 6 ‘canPlayerAttack’. It is good practice to choose an expert which has the information and the system object fulfils this requirement, a downside is the system object has decreased cohesion and is coupled to a large amount of actions in these diagrams. This has helped by information encapsulation – objects use their own info to fulfil tasks and low cohesion is promoted.

1. **OCL**
2. Context RiskSystem::canStartGame() : Boolean

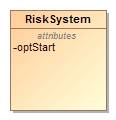
Body:

(Self.optStart == true and player.size() > 2) or player.size() == 6

OR

Context RiskSystem inv (Self.optStart == true and player.size() > 2) or player.size() == 6

/\*function that only starts game if the property opt start is true and there are more than 2 players OR the max player limit is reached, could possible also be expressed as an invariant \*/



3…6



1. Context Player :: infantry : Integer

Derive:

If self..System.getNumPlayers() == 3 then 35

If self. System getNumPlayers() == 4 then 30

If self. System getNumPlayers() == 5 then 25

If self. System getNumPlayers() == 6 then 20

OR

Context System::assignNumInfantry(): Integer

Body:

If self.player.size() == 3 then 35

If self.player.size() == 4 then 30

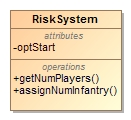
If self.player.size() == 5 then 25

If self.player.size() == 6 then =20

/\*Gets the size of the player collection in an if block then assigns the infantry size\*/



3…6



1. Context Set inv setMakeup: self.card.size() == 3 and car.getAllInstances() -> forAll(c|c<>self implies c.type -> == self.type) or car.getAllInstances() -> forAll(c|c<>self implies c.type -> != self.type) ;

/\*invariant that checks a set is of size 3, then iterates each element in the set and compress against each other to ensure each card is off the same type or they are of different types.\*/



3

1. Context System:howManyDiceRoll(territory: String): Integer

Body:

Self.player.army-> iterate(a|numArmies:Integer|

if(army.currTerritory == territory )

then numArmies = numArmies + 1) <=4

if numArmies == 2 then 1

if numArmies == 3 then 2

if numArmies == 4 then 3



\*

3…6

/\*takes an input of territory being attacked, iterates all players armies and find out who many are in that territory, then assign a roll amount based on the num of armies in that location \*/