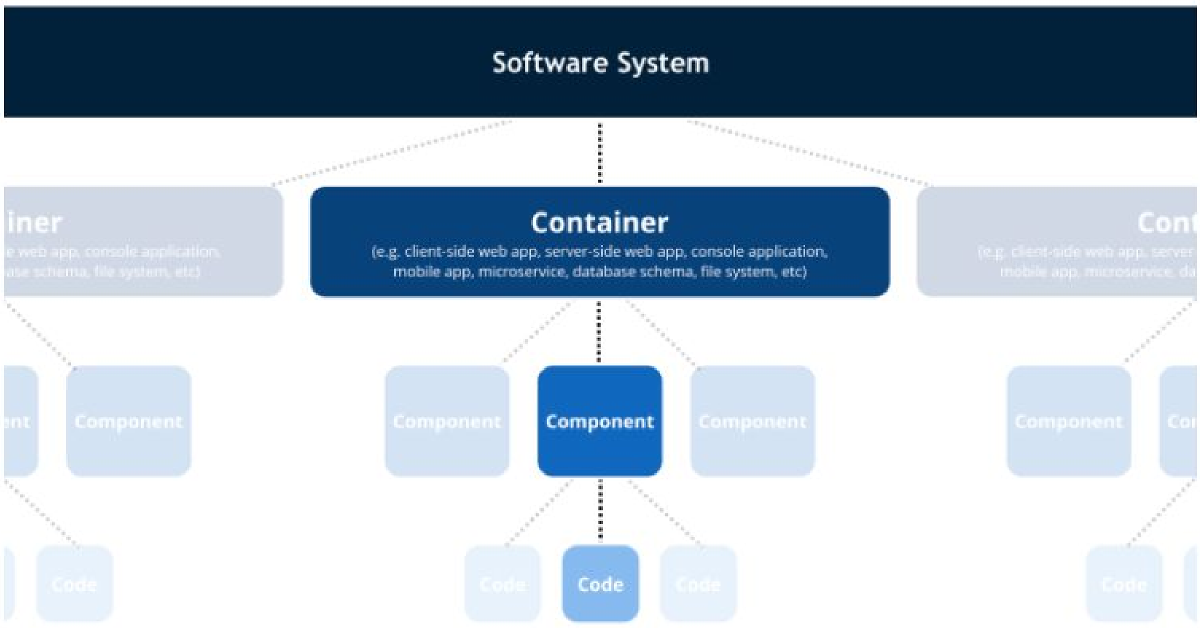
TCI course group deliverable 2

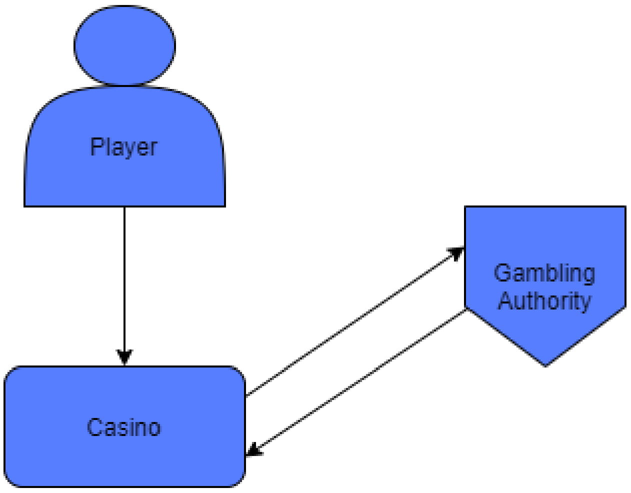
Contents:

1.
Project explanation using C4 model:


Software System
iner
Container
(e.g. client-side web app, server-side web app, console application, mobile app, microservice, database schema, file system, etc)
Con
Component
Component
Component
Component
Component
Code
Code
Code
Code
Code

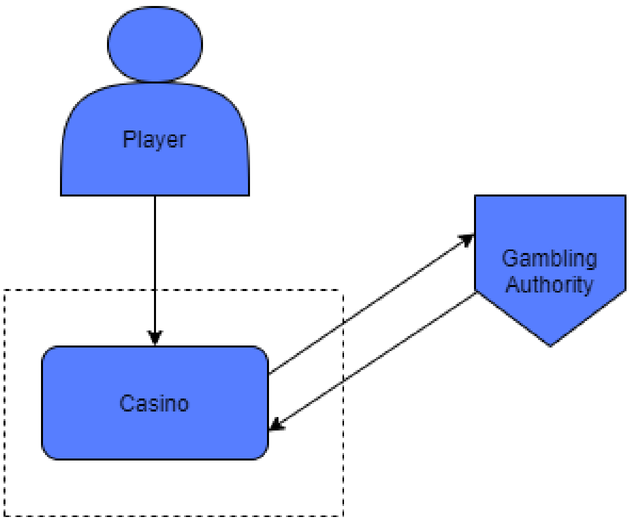
2.
Expanded behavioral testing list

1. Software System



Highest level overview - system context diagram. System consists of three party interactions, as can be seen on the left. A responsible player can use casino services, and such fair services are ensured with cyclic relationship between casino and gambling authority (GA), where in casino provides information to GA on two occasions and receives one response. Most importantly GA logs placed bets within a time frame and provides a truly random outcome. And additionally collects overall gambler activity.

2. Container

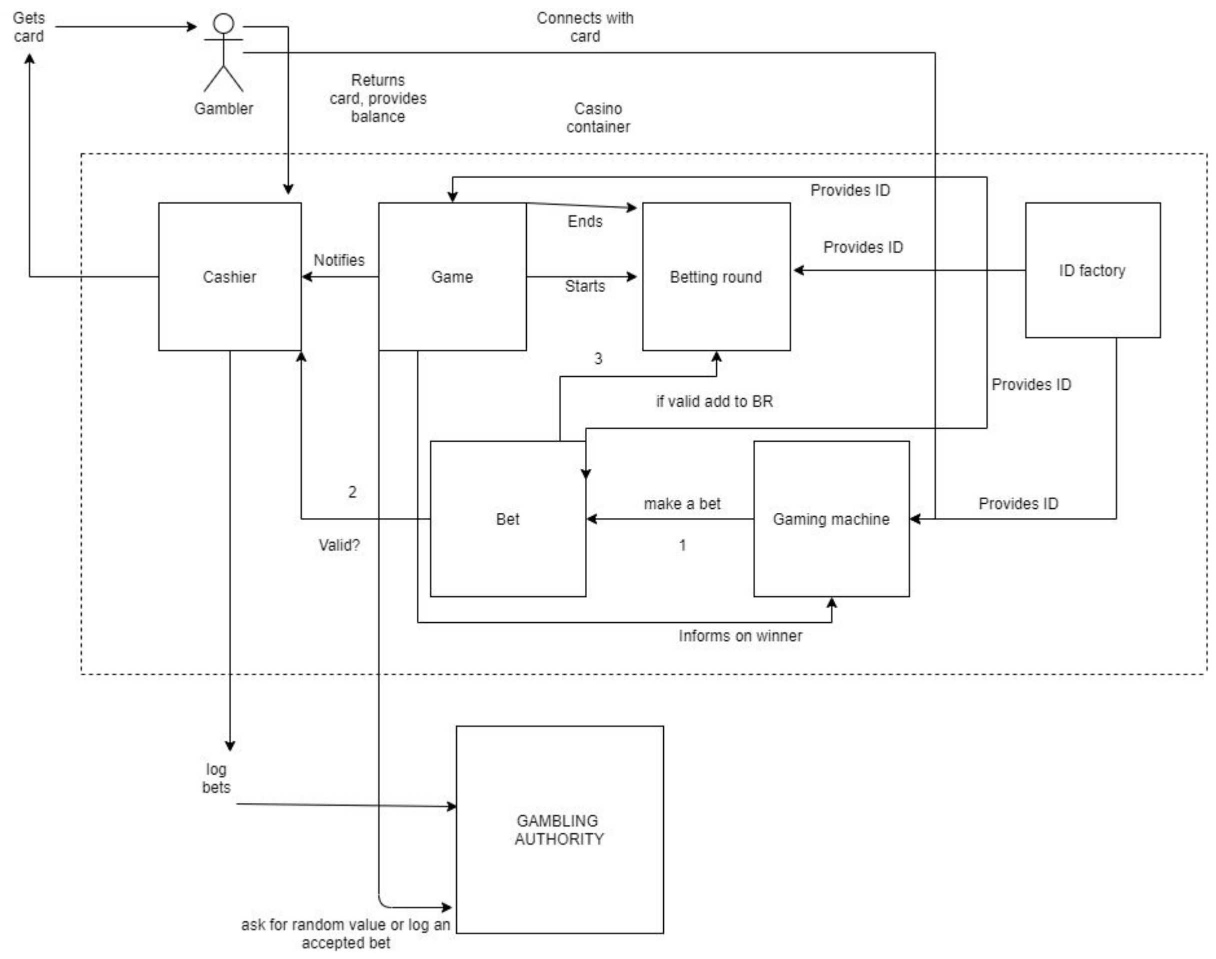


More concrete dive into independent components - container diagram. Which in this case luckily is only one - casino. Player and gambling authority are not at the focus. Normally under the C4 model during this step composite elements of the main container would be exposed, for example API, APP, DB, and else. Casino is a sole and undividable container.

3. Component

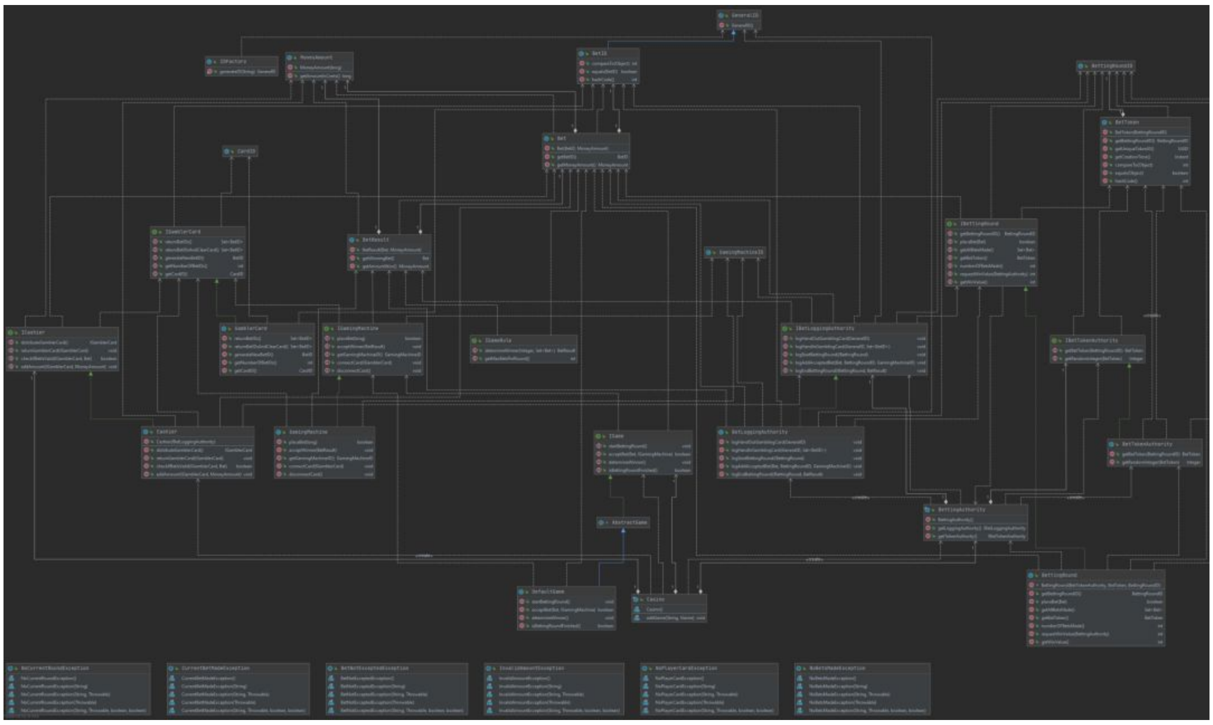
Yet more concrete dive into container contents - component diagram.
Players begin their interaction with a casino through a human cashier. Cashiers enable players to begin participating in one or few games, out of casino owned collection of games. After receiving permission the player interacts with one or few games. Both games and cashiers are using the casino's API to fulfill their procedures. Namely games are providing GA with placed bets, and GA responds with confirmation whether the bet was placed timely. After a time expires GA informs the game on the winner. Cashier interacts with the API, when permission is collected and wins are given out.

Not good enough.
In C4, Components are inside a container, and typically execute in the same process space. Therefore components are not separately deployable units. Though ideally probably there should not be heavy dependence between components, in general, but practically they must enable proper functioning of the container.



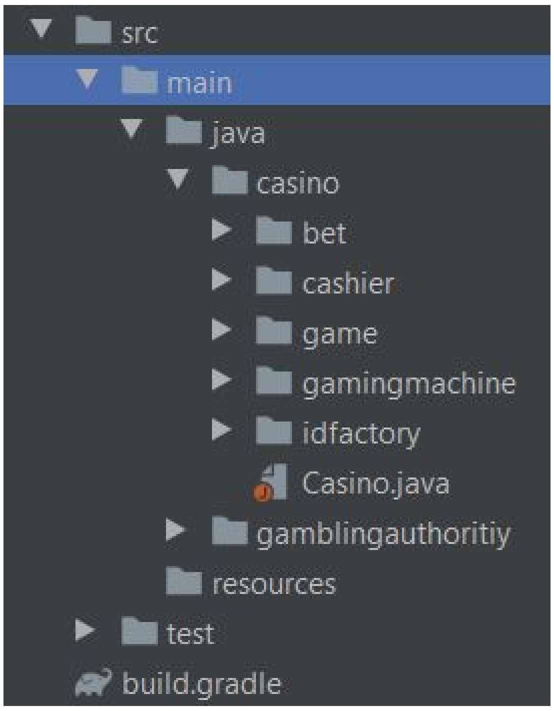
4. Class diagram

To update once git complete



Group deliverable - General Description - Classes as provided at the project startup

CASINO contains the following packages:



```
▼ src
    ▼ main
        ▼ java
            ▼ casino
                ▶ bet
                ▶ cashier
                ▶ game
                ▶ gamingmachine
                ▶ idfactory
                  Casino.java
            ▶ gamblingauthoritiy
          resources
    ▶ test
  build.gradle
```

Classes in the packages were distributed as follows:

- Student A: cashier, bettinground, abstractgame, BetID
- Student B: defaultgame, gamblercard, BettingRoundID, GamingMachineID
- Student C: gamingmachine, gamerule, idfactory, CardID

Student A - Yoanna Borisova
Student B - Dongdong Ke
Student C - Danas Jusys

Package	Behavior	Tests
Bet	All behaviors are immutable	Not testing immutable classes
Cashier	1. Cashier hands out and collects gambler cards to and from players 2. Cashier adds or retracts stored money balance from a card administration 3. Cashier keeps track of active gambler cards given out and their balances 4. Cashier validates whether the bet is valid and deduct balance	1. **TestCashierHandsCards()** - Goal: Test if Player has received a card and if gambling authority is informed when a card is being given out? Mocked - yes. Must use: `void logHandOutGamblingCard(GeneralID card);` 2. **TestCashierCollectsCards()** - Goal if the cashier can collect cards and gambling authority is informed when a card is being returned? Mocked - yes. Must use: `void logHandInGamblingCard(GeneralID`

	5. Cashier is notified how much money to add to the winner balance 6. Cashier removes stored bet id's from a returned gambler card 7. Return all betID by this card 8. Returns all generated betID's by this card, and clears all betID's from the card. 9. The card generates a unique betID for every bet made by the gambler on the machine. 10. Return number of betID's generated on this card. 11. Return the card id	`card, Set<BetID> betsMade);` 3. **TestCashierCanRetractMoneyBalance( )** - Goal to test if the cashier is able to deduce stored money balance from administration? Non mocked - use class method 4. **TestCashierCanAddMoneyBalance()** - Goal to test if the cashier is able to add stored money balance to administration? Non mocked - use class method 5. **TestCashierIsAwareOfCards()** - Goal is to test if the cashier is aware of currently active gambler cards? Non mocked - use class method 6. **TestCashierValidatesBet()** - Goal is to test if the cashier is able to check whether the gambler card contains enough balance for a bet to be placed? Mocked - yes. Bet, gaming machine 7. **TestCashierIsNotifiedOfWinner()** - goal is to test if the cashier is notified of the winner and how much balance to add? Mocked - yes. Game 8. **TestCashierRemovesBetIDs()** - Goal is to test if bet history is removed? Non mocked - use class method **9.** **TestReturnAllBetID()-** Non Mocked -use class method 10. **TestReturnBetIDAndClearCard()-** Non mocked - use class method 11. **TestGenerateNewBetID()-** Non mocked - use class method 12. **TestGetNumberOfBetID()-** Non mocked- use class method 13. **TestReturnCardID()-** Non mocked -use class method
Game	1. Game consists of multiple sequential betting rounds 2. Game consists of one betting round at a moment 3. Game creates betting round by informing GA with unique ID and receives response Gambling token 4. A player is able to place a bet during a betting round 5. Game uses rules to determine maximum number of bets for a betting round 6. Bet is accepted by game's instance of betting round and is stored	1. **TestGameConsistsOfMultipleRounds()** - Does the game consist of multiple betting rounds? Mocked - no. 2. **TestGameDoesNotSupportMoreThanOneRoundAtTheSameTime()** - Does the game support more than one betting round at the moment? Mocked - no 3. **TestGameCanCreateNewBettingRound ()** - Does the game provide GA with a unique new betting round ID and receive gamblig token? Mocked - yes with GA 4. **TestBettingRoundPlaceBet()** - Goal is to test if the betting round betting round successfully adds a bet to the current round. Mocked - yes (Bet). 5. **TestBettingRoundReturnsCorrectNum**

	7. Betting round asks for a random value from the GA 8. Game uses rules to determine winner from GA response, after a betting round 9. Game informs cashier on winner balance change 10. Game notifies connected machines to display win result 11. Betting round sends stored bet log to the GA through casino API, after a betting round 12. Betting round checks if it is finished after every placed and stored bet 13. New betting round is created one current expires 14. Checking the current bettinground and accept bet and check placeBet from current betting round with right amount 15. Throw exception NoCurrentRoundException when the bet is not valid 16. Return the current betting ID 17. When placing the bet successfully, return true 18. Return set of all bets made in this betting round 19. Return betToken from this betting round 20. Return number of bets made in the betting round	**berOfBets()** - Goal is to test if the correct number of bets are returned for the current betting round. Mocked - no. 6. **TestBettingRoundStoresBets()** - Goal is to test if bets are being stored during a betting round in a list? Mocked - no. 7. **TestBettingRoundGetsWinValue()** - Goal is to test if the betting round receives a random win value from the GA. Mocked - yes (GA) 8. **TestBettingRoundFinishedShouldReturnTrue()-** After all bets are finished, return true. Mocked- GamingMachine 9. **TestBettingRoundReturnFalse()-** If not all finished, return false. Mocked - GamingMachine 10. **TestAcceptBetSuccessfully()-** Non mocked - use method 11. **TestAccpetBetThrowException()-** Mocked - Bet 12. **TestbettingRoundID()-** Non mocked -Use method 13. **TestPlaceBetSuccessfully()-** Non mocked - Use method 14. **TestReturnAllBetsDuringThisBet() -**Non mocked - Use method 15. **TestGetToken()-** Non mocked - Use method 16. **TestNumberOfBetsMade()-** Non mocked -Use method 17. **TestGameRuleWorksForMaxNumber()** - Is the game rule working for a maximum number of allowed bets? Mocked - no 18. **TestGameDetermineGameWinner()** - Is game rule determining a winner? Mocked - no 19. **TestGameNotifiesCashierOfWinner()** - Is the game notifying the cashier how to add to the winner balance? Mocked - 20. **TestGameProvidesGAWithBets()** - Is a game providing GA with a set of bets made during a round? Mocked - yes with BettingRound 21. **TestBettingRoundChecksIfItsFinished( )** - Is betting round checking whether it is finished, after each placed bet? Mocked - no. 22. **TestGameCreateNewbettingRound()** - Is a new betting round created once, one expires? Mocked - no Tests for game rule: 1. Does the game rule inform the betting round whether maximum bet capacity has

		been reached? Mocked - yes, bet and betting round 2. Does the game rule determine from a GA winner number response which bet is a winner? Mocked - yes, gaming authority, betting round
Gaming machine	1. Several gaming machines can be connected to one game 2. Gaming machine has unique ID 3. Gambling machine reads from a connected gambler card 4. Maximum of one bet can be submitted per one betting round on a gaming machine 5. Gaming machine checks whether bet is valid 6. Several gaming machines can be connected to one game 7. After accepting the winner, clear all open bets on this machine 8. After placing bet, game machine can't place the bet anymore 9. When the winner has made his bet, update the amount of winner	Tests for gaming machine: 1. **OneGameCanConnectSeveralGameMachine()-**Can several machines connect to one game? Mocked - Game 2. **GamemachineHasUniqueGamingMachineID() -**Do gaming machines have unique IDs? Non mocked - use class constructors 3. **GameMachineShouldReadFromCard()-** Can a gaming machine read from a gamblers card? Mocked - gambler card 4. **CheckIfBetAmountValid -**Can gaming machines check whether the bet is valid? Mocked - cashier **5. CheckBetAmountValid()-** If the bet amount is valid, return true. Mocked- cashier 6. **CheckPlayerHasEnoughMoeny()-**Can a gaming machine accept more than one bet per round from the same gambler card? Mocked - cashier. 7. **TestNoOpenBets()-** All bets should be cleared on this machine 8. **GameMachineOnlyHaveOneBet()-** The game machine only accepts 1 bet. Non mocked - user class 9. **ShouldUpdateWinnerAmount()-** Mocked - cashier
ID Factory	1. Produces various types of identifications for gambler cards, gaming machines, betting rounds, bets under the general id form 2. General id form contains unique identifier and timestamp	Tests for ID factory 1. Is requested ID string case sensitive? Non mocked 2. Is the request presented as a type string? Non mocked Tests for ID's: 1. Test to see if two ID's are equal? Require compareTo and hashCode method overload. 2. Test to see if each given instance type is of supported inheriting type. 3. Applies to BetID, BettingRoundID, GamingMachineID, GamblerCardID

Note:
Test name is constructed from a question.
Exceptions are not mentioned, but each exception will have few tests.
Getters (like returnID etc) will also be tested.