Opgave oefening hoofdstuk 3 en 4: BLACKJACK - model

1. De spelregels van Blackjack

Blackjack is een kaartspel en wordt ook eenentwintigen genoemd. Er is één dealer en minstens één speler. Voor de eenvoud gaan we dit spel uitwerken voor 1 speler.

De speler speelt tegen de dealer. Het doel is kaarten te trekken totdat het totaal zo dicht mogelijk de 21 benadert zonder deze waarde te overschrijden. De speler wint als zijn totaal kleiner of gelijk aan 21 is, en hoger dan dat van de dealer. Hij verliest als zijn totaal hoger is dan 21 of lager of gelijk is aan dat van de dealer. In dit laatste geval mag de totale waarde van de dealer de 21 niet overschrijden.

De speler en de dealer beginnen met twee kaarten. Enkel de eerste kaart van de dealer is zichtbaar. De speler begint dan: hij mag kaarten trekken om zo dicht mogelijk 21 te benaderen. Wanneer de speler tevreden is met zijn totaal kan hij passen. Dan is de dealer aan de beurt. De dealer trekt kaarten tot zijn totaal hoger of gelijk aan het totaal van de speler is.

Elke kaart staat voor zijn aangegeven waarde, behalve de prentjes (jack, queen, king) die tellen voor 10. De aas telt voor 1 of 11, wat het best uitkomt. Indien je eerste twee kaarten samen 21 vormen (10 of prentje gecombineerd met aas) dan heb je Blackjack en win je.

Het doel van deze oefening is het domein voor Blackjack uit te werken. We hebben aandacht voor het ontwerp, de implementatie, de unit testen en debuggen. In deze les zullen we het domein gebruiken in een console-applicatie. In een latere les zullen we ditzelfde domein gebruiken in een web-applicatie.

Voorbeeld spelverloop

Start van een nieuw spel…

```
================
-Spades/Two--?/?-
Dealer total = 2

-Diamonds/Ten--Hearts/Five-
Player total = 15
================

Enter your choice:
1. Another card
2. Pass
```

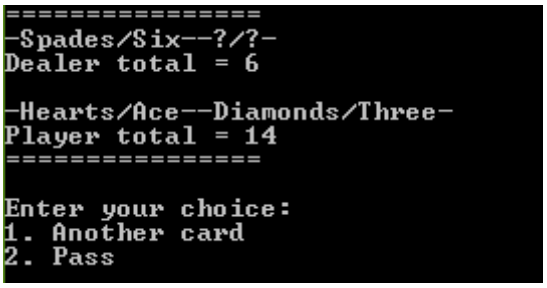
Enkel de eerste kaart van de dealer is zichtbaar, de tweede ligt omgekeerd op tafel en dit is aangegeven als ?/?. Het totaal van de niet omgekeerde kaarten wordt getoond.

Player kiest voor Pass…

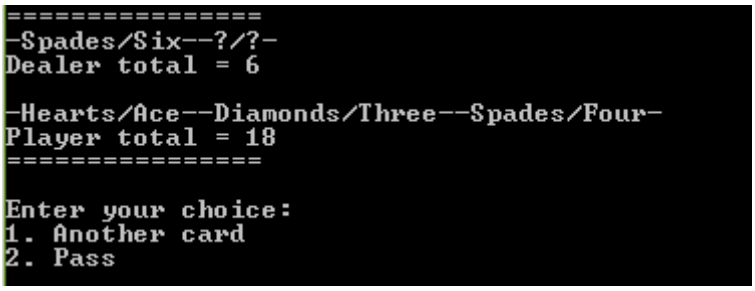
```
================
-Spades/Two--Diamonds/Two--Clubs/Eight--Clubs/Five-
Dealer total = 17

-Diamonds/Ten--Hearts/Five-
Player total = 15
================

Game ends: Dealer wins
Do you want to play again? (y/n)
```

Voorbeeld spelverloop2

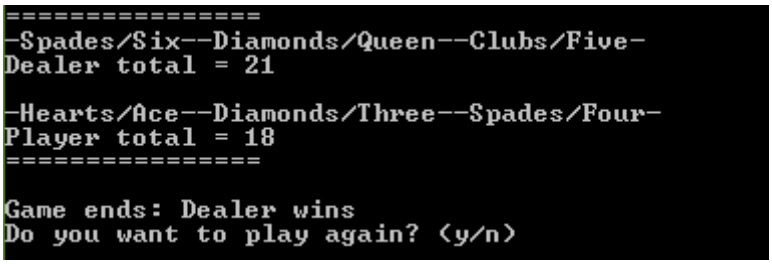
Start van een nieuw spel…



Player kiest voor Another card…



Player kiest voor Pass…

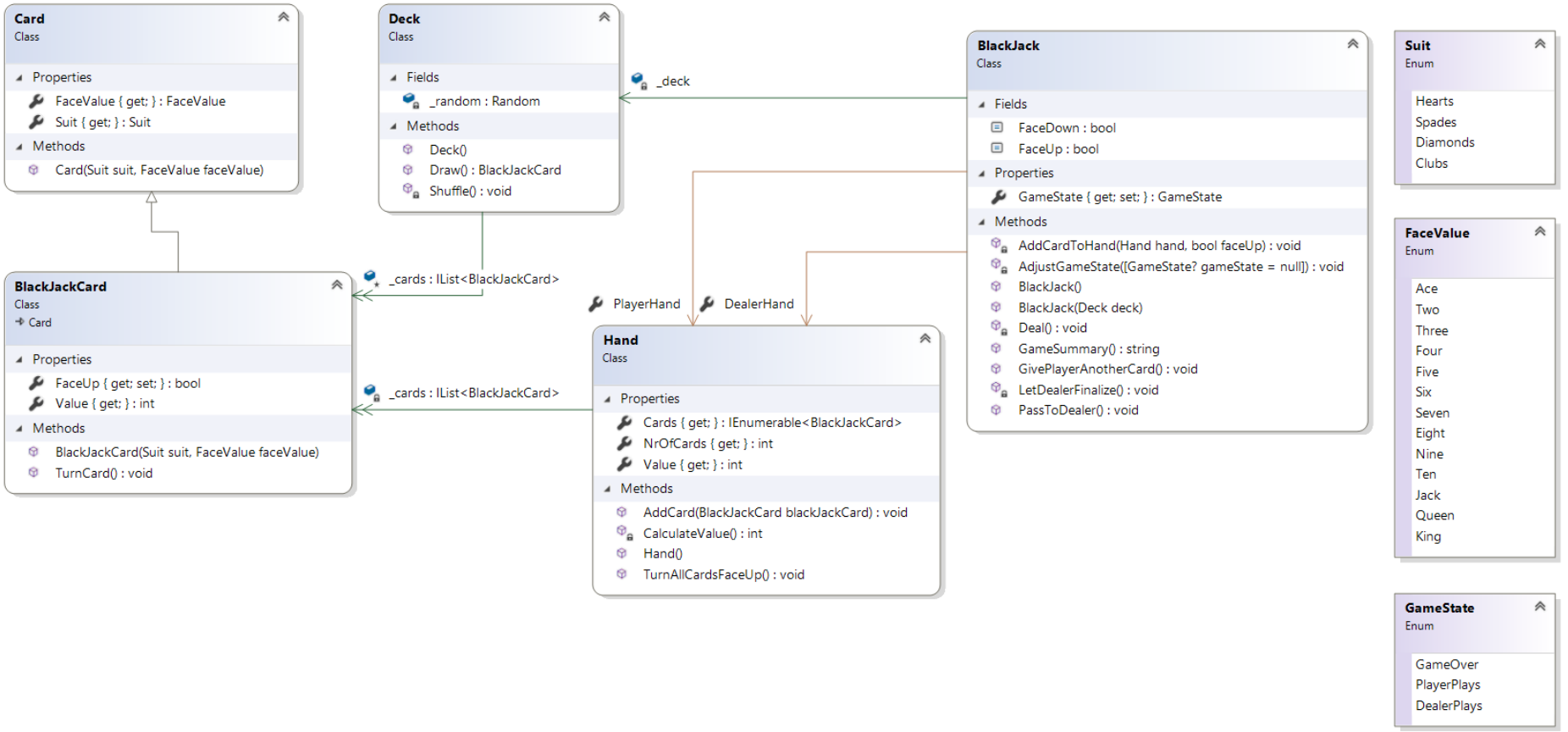


2. Voorbereidend werk

1. Download de folder genaamd **StarterFiles** van chamilo.
2. Maak een nieuwe .Net Core Console applicatie met de naam **BlackJackGame** aan in VS.
3. Voeg het project toe aan git
4. Voeg een nieuw project van het type xUnit Test Project toe aan je solution, noem het **BlackJackGame.Tests.** Verwijder de klasse UnitTest1.cs
5. Voeg in BlackJackGame.Tests een referentie toe naar BlackJackGame (rechtermuisklik op references > Add reference en selecteer in de categorie solution BlackJackGame). Zorg ervoor dat de solution compileert.
6. Commit

3. Het ontwerp

1. Maak een folder genaamd **Models** in het **BlackJackGame** project. Doe hetzelfde in je **BlackJackGame.Tests** project.
2. De folder StarterFiles bevat CardEnums.cs met de enumeraties Suit en FaceValue en GameState.cs met de enumeratie GameState. Voeg deze eerst toe aan het BlackJackGame project in de Models folder.
3. Hieronder kan je het klassendiagram vinden. Maak de klassen aan. De methodes gooien initieel een NotImplementedException. Maak ook het klassendiagram aan.


Card
Class
Properties
FaceValue { get; } : FaceValue
Suit { get; } : Suit
Methods
Card(Suit suit, FaceValue faceValue)
Deck
Class
Fields
_random : Random
Methods
Deck()
Draw() : BlackJackCard
Shuffle() : void
_deck
BlackJack
Class
Fields
FaceDown : bool
FaceUp : bool
Properties
GameState { get; set; } : GameState
Methods
AddCardToHand(Hand hand, bool faceUp) : void
AdjustGameState([GameState? gameState = null]) : void
BlackJack()
BlackJack(Deck deck)
Deal() : void
GameSummary() : string
GivePlayerAnotherCard() : void
LetDealerFinalize() : void
PassToDealer() : void
Suit
Enum
Hearts
Spades
Diamonds
Clubs
FaceValue
Enum
Ace
Two
Three
Four
Five
Six
Seven
Eight
Nine
Ten
Jack
Queen
King
GameState
Enum
GameOver
PlayerPlays
DealerPlays
BlackJackCard
Class
Card
Properties
FaceUp { get; set; } : bool
Value { get; } : int
Methods
BlackJackCard(Suit suit, FaceValue faceValue)
TurnCard() : void
_cards : IList<BlackJackCard>
_cards : IList<BlackJackCard>
PlayerHand
DealerHand
Hand
Class
Properties
Cards { get; } : IEnumerable<BlackJackCard>
NrOfCards { get; } : int
Value { get; } : int
Methods
AddCard(BlackJackCard blackJackCard) : void
CalculateValue() : int
Hand()
TurnAllCardsFaceUp() : void

4. TDD: Unit testen en implementatie

In de filosofie van **Test Driven Development** gaan we nu het domein implementeren. We gaan steeds eerst aandacht hebben voor de testklasse, alvorens we de domeinklasse implementeren. Volgende cyclus gaan we dus meerdere keer doorlopen:

1. Testklasse implementeren
2. Bij test uitvoering falen de testen
3. Domeinklasse implementeren
4. Bij test uitvoering slagen de testen, bij falen kunnen we debuggen.
5. Commit als alle testen slagen

CardTest :

Deze klasse is reeds voorzien in de StarterFiles, voeg ze toe aan je test project.

Card : bevat de gegevens van 1 kaart, nl. **suit** (hearts,...), en **faceValue** (1, 2, ...king)

- bevat de properties Suit en FaceValue, beide van bijhorend enumeratie type
- bevat 1 constructor voor het aanmaken van 1 kaart

Implementeer de klasse Card, zorg dat alle unit testen slagen.

BlackJackCardTest :

- Deze klasse is reeds voorzien in de StarterFiles maar is nog niet volledig, voeg ze toe aan je test project.
- Drie methodes bevatten reeds een Act en Arrange gedeelte maar hebben een “Not yet implemented” fact.

Vervang dit door een correcte Assert.

BlackJackCard : erft van Card. Bevat extra gegevens horend bij een BlackJack kaart

- property FaceUp : beeld op kaart zichtbaar of niet
- property Value : de BlackJack waarde van een kaart. Als het beeldje op de kaart niet zichtbaar is is de waarde 0
- een constructor : bij creatie van een kaart is het beeldje niet zichtbaar
- de methode TurnCard : draait de kaart om

Implementeer de klasse BlackJackCard, zorg dat alle unit testen slagen.

DeckTest : Maak zelf deze testklasse aan en zorg voor volgende testen:

- methode Draw retourneert een object van type BlackJackCard
- constructor levert een deck op met 52 BlackJackCards
- methode Draw werpt een InvalidOperationException wanneer het deck geen kaarten bevat

Deck : een spel kaarten

- _cards : de BlackJack kaarten
- De default constructor maakt een deck van 52 BlackJack kaarten aan, allen met het beeldje naar onder.
- Draw : geeft de bovenste BlackJack kaart uit het deck terug. Als er geen kaarten meer zijn wordt een InvalidOperationException geworpen.
- Shuffle (private methode) : mengen van de kaarten

Implementeer de klasse Deck, zorg dat alle unit testen slagen.

Opm : voor het aanmaken van een deck kan je gebruik maken van de enumeraties

```
foreach (Suit s in Enum.GetValues(typeof(Suit)))
{
}
```

HandTest :

- Deze klasse is reeds voorzien in de Starter maar nog niet volledig, voeg ze toe aan je test project. Implementeer (Arrange/Act/Assert) de testmethodes die nog "not yet implemented" zijn. Groepeer testen, waar mogelijk in Theories

Hand : een speler

- Attribuut _cards (IList) : bevat de BlackJack kaarten van de speler
- Property Cards(IEnumerable) : retourneert de BlackJack kaarten van de speler (enkel getter)
- Property NrOfCards: retourneert aantal kaarten van speler
- Property Value : de totale waarde van de kaarten in de hand van de speler.
- Hand : constructor
- AddCard : voegt een BlackJack kaart toe aan de hand van de speler
- TurnAllCardsFaceUp : draait alle kaarten met beeldje naar boven

Implementeer de klasse Hand, zorg dat alle unit testen slagen.

BlackJackTest :

Deze klasse is reeds voorzien in de Starter en bevat volgende testen. Vooraleer je begint lees je eerst de toelichtingen die achteraan in dit document staan (zie *). Maak eerst de benodigde subklassen aan zodat deze klasse geen foutmeldingen meer geeft.

- bij een nieuw spel (zonder blackjack) bevatten DealerHand en PlayerHand elk 2 kaarten
- bij een nieuw spel (zonder blackjack) bevat de DealerHand een eerste kaart FaceUp en een tweede kaart !FaceUp
- bij een nieuw spel (zonder blackjack) bevat de PlayerHand twee kaarten die FaceUp zijn
- bij een nieuw spel (zonder blackjack) is de GameState gelijk aan PlayerPlays
- GivePlayerAnotherCard voegt een kaart toe aan de playerHand
- GivePlayerAnotherCard moet een InvalidOperationException werpen wanneer de GameState verschillend is van PlayerPlays
- PassToDealer moet leiden tot een GameState gelijk aan GameOver
- GameSummary retourneert null wanneer de GameState niet gelijk aan GameOver is.
- GameSummary bij GameOver is correct
    - wanneer een BlackJack gespeeld wordt
    - wanneer PlayerHand en DealerHand gelijke waarde hebben
    - wanneer PlayerHand boven 21 gaat
    - wanneer PlayerHand onder 21 blijft en DealerHand boven 21 gaat
    - wanneer PlayerHand onder 21 blijft en DealerHand onder 21, maar met een hogere waarde in de DealerHand

BlackJack : het spel

- 2 constanten : FaceDown (waarde false), FaceUp (waarde true)
- DealerHand : hand van de dealer
- PlayerHand : hand van de player
- GameState : status van het spel : player is aan de beurt, de dealer is aan de beurt of game over
- BlackJack : zorgt voor de initialisatie van het spel : nieuw deck, nieuwe hand player en nieuwe hand dealer. Player en dealer krijgen elk 2 kaarten. Bij player beide faceUp, bij dealer 1 faceUp en 1 faceDown. Mogelijk heeft de player nu reeds BlackJack.

- GivePlayerAnotherCard : geeft een nieuwe kaart aan de player, tenminste als de GameState PlayerPlays is. Past daarna eventueel de GameState van het spel aan.
- PassToDealer : De dealer is aan de beurt. Zijn kaarten worden omgedraaid. De GameState wordt DealerPlays. De dealer speelt verder tot GameState GameOver bereikt.
- GameSummary : null indien spel nog niet beëindigd is. De mogelijkheden :
    - Player Burned, Dealer Wins
    - Dealer Burned, Player Wins
    - Equal, Dealer Wins
    - Dealer Wins
    - Player Wins
    - BLACKJACK

Voorstel voor private methodes

- AddCardToHand(Hand hand, bool faceUp) : geeft een kaart uit het deck aan de betreffende speler, al dan niet faceUp.
- AdjustGameState (GameState? gamestate=null): de GameState wordt aangepast. Maakt gebruik van een nullable type GameState en een optional parameter.
  Bij aanroep kan je eventueel een nieuwe gameState doorgeven: bv. wanneer de speler past weet je dat de nieuwe GameState DealerPlays is.
  Verder wordt er in deze methode getest of de GameState moet aangepast worden omdat speler/dealer boven de 21 gaan. Bv. telkens wanneer de dealer een kaart trekt kan je deze methode aanroepen zonder parameter. Er wordt dan gewoon gekeken of het totaal van de dealer boven dat van de speler, of boven 21, uitkomt. In dat geval gaat de methode de GameState op GameOver zetten.
- Deal : Player en dealer krijgen elk 2 kaarten. Bij player beide faceUp, bij dealer 1 faceUp en 1 faceDown. Stelt de GameStatus in. Mogelijk heeft de player nu reeds BJ.
- LetDealerFinalize : dealer speelt verder tot GameOver

Implementeer de klasse BlackJack, zorg dat alle unit testen slagen.

In de StartFiles vind je Program.cs. Deze bevat de applicatielogica. Vervang de bestaande Program.cs door deze versie die je in de startfiles vindt. Veel plezier met het spelen…

(*) Toelichtingen bij BlackJackTest

Voor de meeste van deze testen moet je **controle hebben over de kaarten die zullen uitgedeeld worden**. Hoe kan je bijvoorbeeld testen op Blackjack indien je niet met zekerheid een Blackjack kan uitdelen? Hoe zorg je ervoor dat je kan testen of het spel correct reageert wanneer de dealer wint als je de kaarten niet zo kan uitdelen dat winst door de dealer mogelijk is?

Een mogelijke oplossing is een deck kaarten te **injecteren** in BlackJack. Voeg hiervoor een tweede constructor aan de klasse BlackJack toe die dit toelaat: de constructor kent een parameter van het type Deck, en tijdens constructie wordt de waarde van deze parameter toegekend aan het field deck.

```
public BlackJack(Deck deck)
{
    this.deck = deck;
    // other code
}
```

Nu kunnen we zelf onze deck kaarten voorbereiden door subklassen van de klasse Deck te maken. In deze subklassen kunnen we tijdens constructie de gewenste kaarten in de lijst met kaarten stoppen. Dit veronderstelt wel dat we in die subklassen aan die lijst kunnen. Maak hiervoor het field cards uit de klasse Deck protected ipv private.

```
protected IList<BlackJackCard> _cards;
```

Via collection initializers kunnen we in onze constructor op een mooie manier de gewenste deck aanmaken. Een voorbeeld van een subklasse die een BlackJack bevat en die je ook in de starter vindt:

```
public class PlayerBlackJackWinDeck : Deck {

    public PlayerBlackJackWinDeck() {
        _cards = new List<BlackJackCard>
        {
            //dealer
            new BlackJackCard(Suit.Clubs, FaceValue.Seven),
            new BlackJackCard(Suit.Clubs, FaceValue.Seven),

            //player
            new BlackJackCard(Suit.Clubs, FaceValue.Ace),
            new BlackJackCard(Suit.Clubs, FaceValue.Ten),

            //dealer
            new BlackJackCard(Suit.Clubs, FaceValue.Ten),
        };
    }
}
```

In onze testen kunnen we nu een BlackJack spel aanmaken met de gewenste deck:

```
BlackJack game = new BlackJack(new PlayerBlackJackWinDeck());
```

Maak in het Unit test project een folder Decks aan in de folder Models, en voeg er de subklassen aan toe.