Simple Jack - Programmering og Problemløsning

Mads U. Svendsen, Anders F. Jørgensen, Nicolai L. Hargreave, Bo H. Thomsen

4. januar 2016

Indhold

1 Forord

2 Introduktion

Denne opgave er lavet i Programmering og Problemløsning(PoP), på Datalogisk Institut - Københavns Universitet(DIKU). Opgaven har opgavenummeret 10g, målet for opgaven er at udvikle en spilbar version af det beskrevet SimpleJack.

Sådan kompilerer du projektet
I `/src` mappen ligger der en Makefile og hvis man kører den, kompileres game.exe, begge filer kan findes i mappen `/src`. Kommandoen til at kompilerer er make og kan den kan skrives i terminalen.
Derudover kan man kompilerer projektet med dokumentation, ved at bruge kommandoen make withdocs
Tests findes i mappen `/src`, i filen tests.fsx, og kan kompileres med fsharpc og køres med mono.

3 Problemformulering

I dette afsnit vil vi introducere det problem vi ønsker at løse med programmet, og beskrive hvordan vi har forstået den udleveret kravspecifikation.
I dette projekt vil vi gerne udvikle et program der kan spilles af 1-5 spillere i kommandopromten, med mulighed for AI. Der er implementeret i F# og følger reglerne for SimpleJack der er beskrevet i afsnittet Afsnit **??** - **??**.

3.1 Kravspecifikation

Dette forløb har fokus på Klasser og objekter, under emnet Object Orienteret Programmering (OOP). Et af kravene til dette projekt er defor at progammet skal være designet efter OOP paradigmen, og at der skal være et medfølgende UML diagram som programdesignet følger.
Nedenunder følger vores opfattelse af de stillede krav(Opgavebeskrivelsen), altså de krav vi måler vores færdige produkt på.

1. I SimpleJack spilles ikke om penge/jetoner men om sejr/tab, mellem en spiller og dealer
2. SimpleJack består af en dealer og 1-5 spillere
3. Der bruges et normalt kortspil (uden jokere), altså 13x4 = 52 kort
4. Ved spilstart får dealer og hver spiller 2 tilfælde kort fra bunken, med billedsiden opad - så værdien er synlig for alle
5. Hver spiller har en tur, og dealeren har altid tur til sidst

Kortværdierne i spillet er fordel efter følgende princip:

1. Billedkort (knægt, dame og konge) har alle værdien 10
2. Et es kan enten have værdien 1 eller 11
3. Kort mellem 2 og 10 har den påtrygte værdi

Hver spiller, spiller et individuelt spil med dealeren, hvor det gælder om at ende med en sum af kortenes værdier er højere end dealerens sum. Hvis summen overstiger 21 er deltageren "bust"og har tabt. Fordi hver spiller, spiller et individuelt spil kan alle sagtens vinde.
Når en spiller har tur, skal en af de følgende handlinger udføres:

1. "Stand": Spilleren/dealeren vælger at give sin tur videre
2. "Hit": Spilleren/dealeren modtager et kort ad gangen fra bunken, indtil han/hun stopper sin tur

En spiller har vundet hvis ingen af følgende er gældende:

1. Spilleren er "bust"
2. Spillerens hånd har en sum der er lavere eller lig med, summen af dealerens hånd
3. Både spilleren og dealeren har SimpleJack(Es og et billedkort)

En spiller kan enten deltage i spillet via terminalen, eller en spiller kan være en AI som følger følgende regler(Opgavebeskrivelsen):

1. Vælg altid "Hit", medmindre summen af egne kort kan være 17 eller over, ellers vælg "Stand"
2. Vælg tilfældigt mellem "Hit"og "Stand". Hvis "Hit"vælges trækkes et kort og der vælges igen tilfældigt mellem "Hit"og "Stand"osv.

4 Problemanalyse og design

I dette afsnit vil vi beskrive hvilken struktur vi har tænkt os at opbygge vores programefter. Vi vil introducere et UML diagram og beskrive de tanker der ligger bag vores designvalg.

Struktur

I dette afsnit vil vi beskrive den strukturelle opbygning vi har overvejet i vores design.
Vores design bygger op omkring fire kerne klasser, Game, Player, Hand og Card og en enumeration kaldet Suits - der huser de fire forskellige kulører. De forskellige klasser skal mimikke fysiske objekter og funktioner der forekommer i spillet.

Game

Game gemmer Player objekttet for dealeren, en liste af Player objekter af de spillere der deltager og den nuværende stak der kan trækkes fra - denne stak er af typen Hand. Ved siden af det har den en funktion til at returnere antallet af spillere i spillet, det er i princippet bare en funktion der tager længden af players arrayen. Game indeholder også en funktion draw, der trækker et kort fra stack og ligger den ind i den valgte spillers hånd.
Game objectets funktion er at holde styr på alt det nødvendige for spil logikken, så al datahåndtering til et spil foregår i dette objekt.

Hand

Hand gemmer en array af Card objekter, og skal være et objekt der håndtere den funktion vores hånd har i et normalt kortspil. En hånd har en funktion der trækker et kort, altid det øverste(stak) denne har vi valgt at kalde drop, den har en funktion der ligger et kort på stakken denne hedder draw, så kan hele håndens kort erstattets, hånden skal kunne blandes og hånden skal kunne skrives ud som string.

Player

En spiller/person skal have et navn(name) i følge kravspecifikationen, vi bruger også Playerklassen til AI, så derfor har vi også en bool(AI), index er array indexet for Playeren i Players arrayen, eller hvornår spilleren har tur, og hand er et Hand objekt der indeholder spillerens hånd.
Ved siden af de properties har vi en score funktion der beregner spillens score, ved at gå gennem de kort der er i hånden. En isBusted der beregner om scoren er for høj og en scoreboard, der laver en tekstrepræsentaiton af spillerens hånd og score.

Card

Et spillekort har en kulør, se Sektionen **??**, og en værdi/index der repræsentere kortet, hvor 11 - 13 er (bonde, dame, konge) og numrene 2 - 10 har deres respektive værdi og et es har værdien 1. Denne værdi må ikke sammenblandes med den værdi kortet har i Blackjack, da alle billedkort har samme værdi - og derfor vil den værdi ikke være unik. Denne value sammenlædet med suit skaber en unik værdi, der kan bruges til at reprænsetere kortet når spillepladen vises.
Udover suit og value, har kortet en funktion toString() der returnere en tekst repræsentation af kortet.

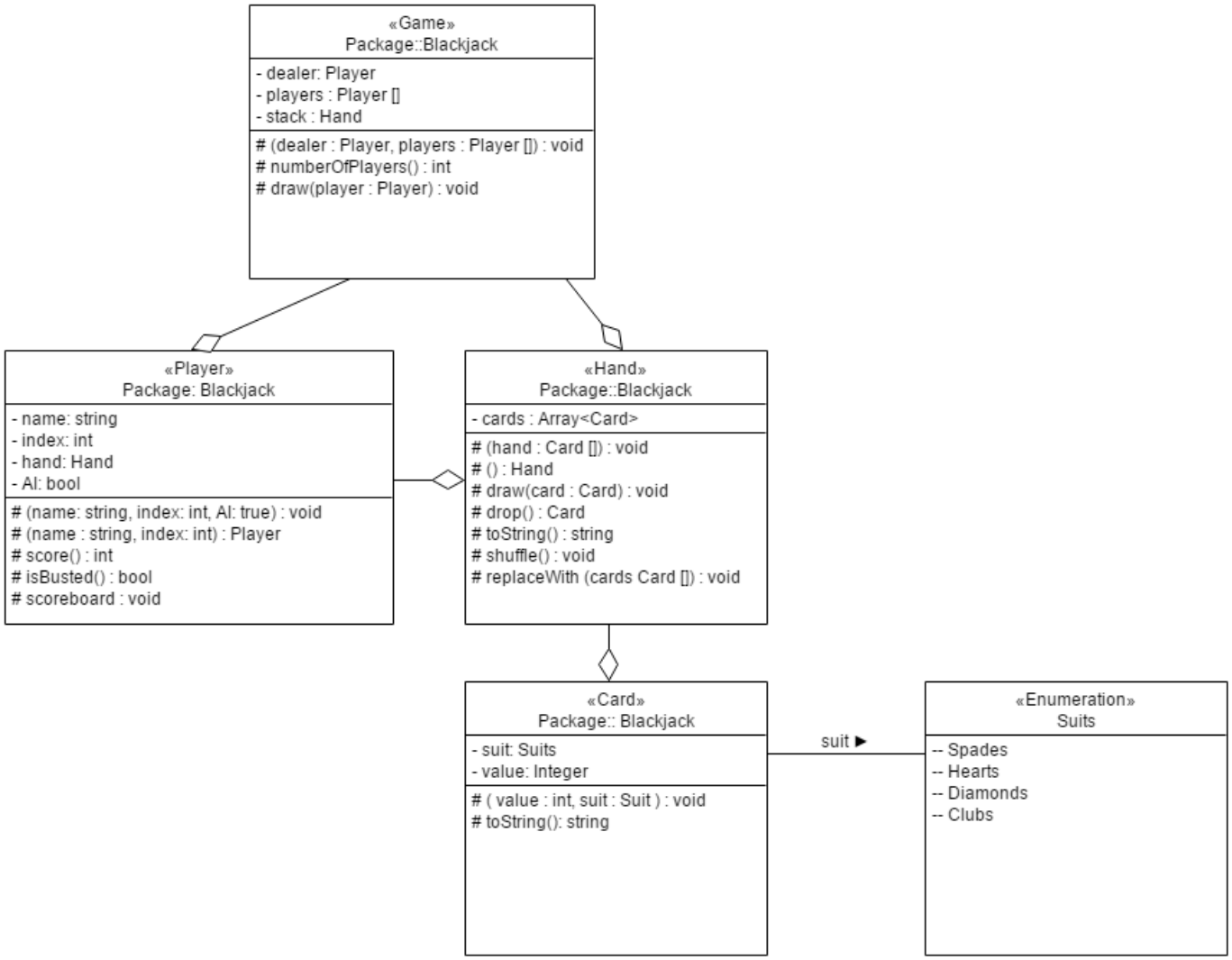
```
member this.toString() =
  let suit =
    match this.suit with
    | Spades -> "spade"
    | Hearts -> "heart"
    | Diamonds -> "diamond"
    | Clubs -> "club"
  let value =
    match this.value with
    | 1 ->  "A"
    | 11 -> "J"
    | 12 -> "Q"
    | 13 -> "K"
```

```
    | x -> sprintf "%d" x                                    14
sprintf "%s%s" suit value                                    15
```

Listing 1: Card's toString metode

Suits

En enumeration er, specielt i OOP, en god måde at repræsentere forskellige værdier for det sammen field, når der kun er de mulige værdier for et field. I vores tilfælde, med Card.suit, har vi kun de fire værdier og derfor ift validering, struktur m.m valgte vi at bruge en enumeration.



Figur 1: UML diagram over vores klasse implementation

5 Programbeskrivelse

Vores programkode er opdelt i fire filer der kan findes i `/src` mappen. Filen `blackjack.fsx` indeholder spillets hjælpefunktioner, enumerations og klasser, der huser de forskellige spilobjekter. Alle disse er beskrevet i Afsnit **??** - **??**
Filen `game.fsx` indeholder spillets logik, main-loop og det er den files der køres. Filen `headers.fsx` indeholder headers/"grafik"der printes i konsollen den indeholder følgende headers:

1. menuHeader der vises over hovedmenuen
2. mainHeader der vises når der skal vælges kommando

3. header der vises på alle andre tidspunkter

Filen `tests.fsx` indeholder unit-tests af programmet.

5.1 Hjælpefunktion

Game.fsx - validate$_name - validate_yn$
Blackjack.fsx - write - writeln - readln - setcursor - clear

6 Afprøvning

7 Diskussion og konlusion

8 Bilag

Dette afsnit indeholder en brugervejledning for brug og spil af SimpleJack og spillets programkode.

8.1 Brugervejledning

8.2 Kildekode

Dette afsnit indeholder alt den kode der er blevet udviklet til projektet, både klasser, spillogik og tests.

```
// Console helperfunctions
let write (str:string) = System.Console.Write str
let writeln (str:string) = System.Console.WriteLine str
let readln() = System.Console.ReadLine()
let setcursor(x,y) = System.Console.SetCursorPosition(x,y)
let clear() = System.Console.Clear()

/// <summary>Enumeration representing card-suits</summary>
type Suits = Spades | Hearts | Diamonds | Clubs

/// <summary>Card is an object representing af card, with value and suit</summary>
/// <param name="value">Integer representing the card value. 1 is A, 11-13 are
/// picturecards</param>
/// <param name="suit">Suits enumeration representing the suit of the card</param>
type Card(value,suit) =
  member this.value:int = value
  member this.suit:Suits = suit
  member this.toString() =
    let suit =
      match this.suit with
      | Spades -> "spade"
      | Hearts -> "heart"
      | Diamonds -> "diamond"
      | Clubs -> "club"
    let value =
      match this.value with
      | 1 -> "A"
      | 11 -> "J"
      | 12 -> "Q"
      | 13 -> "K"
      | x -> sprintf "%d" x
    sprintf "%s%s" suit value

/// <summary>Hand is an object representing a players hand. A Hand-object can
/// draw (Hand.draw card) or drop (Hand.drop) a card. The hand can be shuffled
/// (Hand.shuffle) and replaced by a new Card Array (Hand.replace cards) </summary>
/// <param name="hand">Card Array that represents the cards on the hand (optional)</param>
type Hand(hand) =
  let mutable c:(Card array) = hand
  member this.cards with get() = c
  member this.drop =
    let lastIndex = (Array.length c)-1
    let card = c.[0]
    c <- c.[1..lastIndex]
    card
  member this.draw (card:Card) = c <- Array.append [|card|] c
  member this.toString() =
    let mutable str = ""
    for i=0 to (Array.length c)-1 do
      if i>0 then str <- str + " "
```

```
      str <- str + c.[i].toString()
    str
  member this.shuffle() =
    let len = Array.length c
    let testCard = Card(-1,Spades)
    let newHand = Array.create len testCard
    let test (card:Card) = card.toString()=testCard.toString()
    let rnd = System.Random()
    for i in 0..(len-1) do
      let mutable j = rnd.Next(0,len)
      while test c.[j] do
        j <- rnd.Next(0,len)
      newHand.[i] <- c.[j]
      c.[j] <- testCard
    c <- newHand
  member this.replaceWith cards =
    c <- cards
  new()=
    Hand([|||])

/// <summary>Player is an object representing a Player (AI or human). A Player
/// has a name, index (representing order of game-flow), a hand cards, and a
/// score.
/// The score is updated when called, and are to determine if a player has bust
/// (over 21 points).</summary>
/// <param name="name»String representing the name of the player</param>
/// <param name="index»Integer represebting the index of the position in
/// Player Array in Game object</param>
/// <param name="AI»Boolean representing whether a player is a NPC
/// (Non-Playable Character) or PC (Playable Character)</param>
type Player(name,index,AI) =
  let h = new Hand()
  member this.name:string = name
  member this.index:int = index
  member this.hand = h
  member this.AI:bool = AI
  member this.score =
    let mutable score = 0
    let mutable es = 0
    for card in this.hand.cards do
      if card.value = 1 then es <- es + 1
      if card.value > 10 then
        score <- score + 10
      else
        score <- score + card.value
    while es>0 && floor(float(21-score)/10.)>=1. do
      score <- score+10
      es <- es-1
    score
  member this.isBusted() = (this.score>21)
  member this.scoreboard() =
    let space = if (this.index+1)%3=0 && this.index <> 0 then "" else "  "
    let top =    "" + space
    let middle = "" + space
    let empty =  "" + space
    let bottom = "" + space
    let x = (String.length top - String.length space)*(this.index%3) + (
       this.index)%3
```

```
    let y = (8*(this.index/3)+5)
    System.Console.SetCursorPosition(x,y)
    System.Console.Write top
    System.Console.SetCursorPosition(x,y+1)
    System.Console.Write empty
    let xn = (String.length empty)/2 - (String.length this.name)/2
    System.Console.SetCursorPosition(x+xn,y+1)
    System.Console.Write this.name
    System.Console.SetCursorPosition(x,y+2)
    System.Console.Write middle
    System.Console.SetCursorPosition(x,y+3)
    System.Console.Write empty
    let xc = (String.length empty)/2 - (String.length (h.toString()))/2
    System.Console.SetCursorPosition(x+xc,y+3)
    System.Console.Write (h.toString())
    System.Console.SetCursorPosition(x,y+4)
    System.Console.Write empty
    let score = sprintf "(%d)" this.score
    let xs = (String.length empty)/2 - (String.length (score))/2
    System.Console.SetCursorPosition(x+xs,y+4)
    System.Console.Write score
    System.Console.SetCursorPosition(x,y+5)
    System.Console.Write middle
    System.Console.SetCursorPosition(x,y+6)
    System.Console.Write empty
    System.Console.SetCursorPosition(x,y+7)
    System.Console.Write bottom
  new(name, index) =
    Player(name, index, false)

/// <summary>Game is an object which is used to contain a collection of data,
/// for which is used in-game, like players, a dealer, and a card stack.
/// The Game object is responsible for transfering cards from the stack to the
/// players.</summary>
/// <param name="dealer»A Player object representing a dealer. Player.
/// AI must be set to true.</param>
/// <param name="players»An Array of Player objects.</param>
type Game(dealer,players) =
  let s = new Hand()
  do
    let mutable cards = [||]:(Card array)
    for i=1 to 13 do
      cards <- Array.append cards [|Card(i,Hearts);Card(i,Spades);Card(i,
          Diamonds);Card(i,Clubs)|]
    s.replaceWith cards
    s.shuffle()
  member this.dealer:Player = dealer
  member this.players:(Player array) = players
  member this.numberOfPlayers = Array.length players
  member this.stack = s
  member this.draw (player:Player) =
    if Array.length this.stack.cards > 0 then
      player.hand.draw this.stack.drop
```

Listing 2: Spilklasser

```
#load "./blackjack.fsx"
#load "./headers.fsx"
open Blackjack
```

```
open Headers

///
///
///
let validate_name str = String.length str > 0 && String.length str < 25
let validate_yn str = (str = "y" || str = "n")

///
///
///
let printScoreboard (game:Game) =
  clear()
  write mainHeader
  for player in game.players do
    player.scoreboard()
  game.dealer.scoreboard()
  System.Console.WriteLine ""


///
///
///
let selectPlayer (player:Player) =
  let c = (System.Console.CursorLeft,System.Console.CursorTop)
  let fill = "|XXXXXXXXXXXXXXXXXXXXXXXX|"
  let x = (String.length fill)*(player.index%3) + (player.index)%3
  let y = (8*(player.index/3)+11)
  System.Console.SetCursorPosition(x,y)
  System.Console.Write fill
  System.Console.SetCursorPosition c

///
///
///
let AI (game:Game) (player:Player) =
  let mutable bestValue = 0
  for player in game.players do
    let score = player.score
    if score<22 && score>bestValue then bestValue <- score
  System.Threading.Thread.Sleep(500)
  let mutable IDare = true
  while IDare do
    let diff = max 0 (21 - player.score)
    let es = Array.filter (fun (x:Card)->x.value=1) game.stack.cards |>
       Array.length
    let p = Array.filter (fun (x:Card)->x.value<=diff) game.stack.cards |>
       Array.length
    let pos x = if x < 0 then -x else x
    if p > 40 || es > 0 && p+10 > 20 || p > 25 && pos (bestValue-player.
       score) < 4 then
      System.Threading.Thread.Sleep((52-p)*60)
      game.draw player
    else
      IDare <- false
    printScoreboard game
    selectPlayer player
```

```
///
///
///
let rec main (game:Game) =
  for player in game.players do
    printScoreboard game
    selectPlayer player
    if player.AI=true then
      AI game player
    else
      let mutable command = ""
      while command <> "stand" && player.score < 21 do
        command <- readln()
        if command = "hit" then
          game.draw player
        printScoreboard game
        selectPlayer player
  AI game game.dealer
  let mutable winners = [||]:(Player array)
  for player in game.players do
    if player.isBusted()=false && player.score > game.dealer.score then
      if (player.score=21 && game.dealer.score=21 && Array.length player.
          hand.cards=2
      && Array.length player.hand.cards = Array.length game.dealer.hand.
          cards)=false then

        winners <- Array.append winners [|player|]
  if Array.length winners = 0 && game.dealer.score <= 21 then
    writeln "Dealer was too good!"
  elif Array.length winners = 0 then
    writeln "No winners!"
  else
    writeln "And the winner(s) is:"
  for winner in winners do
    writeln (sprintf " dot %s (%d)" winner.name winner.score)
  write "New round (y/n)?"
  let mutable input = readln()
  while validate_yn input = false do
    clear()
    write header
    write "New round (y/n)?"
    input <- readln()
  if input = "y" then
    for player in game.players do player.hand.replaceWith [||]
    game.dealer.hand.replaceWith [||]
    main(Game(game.dealer,game.players))

///
///
///
let setup() =
  clear()
  write header
  let rec nop() =
    clear()
    write header
    write "Number of players (1-5): "
    let c =
```

```
      try
        readln() |> int
      with
      | _ -> 0
    if c < 1 || c > 5 then nop() else int c
  let numberOfPlayers = nop()
  let mutable players:(Player array) = [||]
  for i=0 to numberOfPlayers-1 do
    let mutable name = ""
    while validate_name name = false do
      clear()
      write header
      write (sprintf "Player %d's name is: " (i+1))
      name <- readln()
      writeln ""
    let mutable input = ""
    while validate_yn input = false do
      clear()
      write header
      write (sprintf "Is player %d a human (y/n): " (i+1))
      input <- readln()
      writeln ""
    let AI = input = "n"
    if AI then name <- name + "(AI)"
    players <- Array.append players [|Player(name,i,AI)|]
  clear()
  let dealer = Player("Dealer",numberOfPlayers,true)
  main(Game(dealer,players))

///
///
///
let rec menu() =
  clear()
  write menuHeader
  let input = System.Console.ReadKey()
  System.Threading.Thread.Sleep(50)
  match input.KeyChar with
  | '1' ->
    setup()
    menu()
  | '2' ->
    clear()
    exit 0
  | _ -> menu()
menu()
```

Listing 3: Spillogikken

Listing 4: Tests