Pesten

Izaak van Dongen

April 27, 2018

Contents

Listings

1 Introduction

Pesten is a classic Dutch card game, similar to Uno but played with playing cards. The name translates as something like 'bothering'. The objective is to annoy your fellow players as much as possible. The suits function as colours, and various cards have special functions. There is also a fine tradition of introducing house rules. People can feel very strongly about these, and we're still finding special cases that require and

appeal to the van Dongen jury, so while this implementation aims to codify the van Dongen house rules, it makes no guarantee of absolute accuracy.

It is played with one to two packs of cards, but this is entirely variable depending on how many players there are.

2 Rules

2.1 Basic play

The basic functioning of the game is, as mentioned, very similar to Uno. There is a discard pile and a pickup pile. The top card of the discard pile determines the current player's allowable discards. A player is allowed to play any card of either the same suit or the same rank as the current card, or a joker, unless there is currently a special effect in play...

If a player is able to play, they must. However, if they can't, they pick up one card from the draw pile. If they are able to play this card, they may, and this card takes effect as normal.

2.2 Special cards

Some cards have effects, which generally apply to the next player. Due to this, sometimes players opposite each other form 'teams' as they never obstruct each other. This mode of play also allows everyone to win more often. All effects are listed here:

Card	Effect
Joker	This card has no suit so can be played on any card (unless this table specifies otherwise, see 2). The following player must take 5 cards, or play their own joker, which increases the count to 10 and moves to the next player. The player who ends up taking the cards may decide on the initial suit after the joker, but may not play. Play goes to the next player after the suit has been decided.
2	The following player must take 2 cards. If the following player has a 2, they may play it and then the total number of cards to be taken is 4, by the next player, and so on. A player may also 'escalate' by playing a 3 of the same rank. This increases the payload by three. A following player must then play a 4 of the same rank or a 3, and so on. Any cards played in this mode are exempt from their normal special effects. A joker cannot be played while a 2-stack is in play. The player who ends up taking the cards may not play, and play goes to the next player from them.
7	A 7 allows the player to take another turn. Their next card must fit on the seven, or they will have to pick up a card.
8	An 8 skips the next player. This action cannot be stopped by any card, as the next player simply doesn't get a turn, so cannot do anything like play their own 8.
10	A 10 means the player before the current player now has their turn, but play goes on as normal.
Jack	A jack lets the player choose the suit to go on with. The next player must play a jack, or a card of the declared suit, or a joker.
King	Changes the direction of play. The next turn goes the player originally 'before' the current player.

Any unmentioned cards are not special.

2.3 Card sets

A player may also play a 'set' of cards. A set of cards is either three or more of the same rank, or three or more adjacent cards of the same suit in ascending or descending order. For example, one might play 6 ♣, 6 ♢, 6 ♠, or 6 ♣, 5 ♣, 4 ♣. NB for the purpose of these sets, aces are considered to be both before 2 and after the King.

A set must contain at least three cards following the pattern. However, the play of two cards is permitted if they form a set with the top card on the discard pile, eg you can play 6 ♢, 6 ♠ if there is a 6 ♣ at the top of the discard pile.

If there are special effect cards, only the top card has its effect. This means that it's a better idea to play 9 ♣, 8 ♣, 7 ♣ than the other way round, as this gives the player another turn.

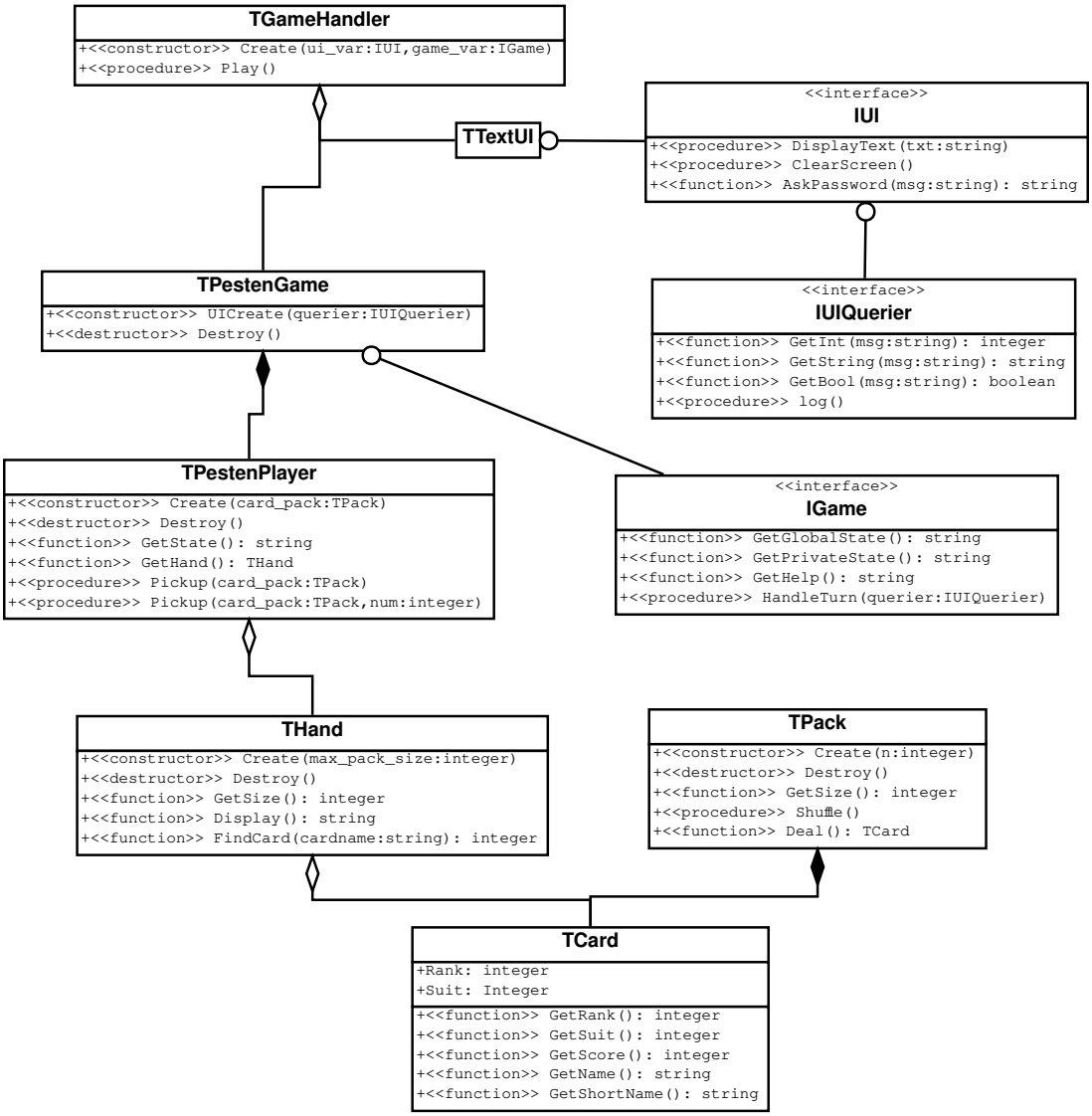
If a card set is played while a 2-stack is building, only the last card of the set contributes to the number to be picked up.

2.4 Last (card), but not least

As soon as a player reaches their last card, they must tap the table twice and declare 'last card' with appropriate volume and enthusiasm. If they don't, a two-card penalty is inflicted. If the player is playing a series of cards (using 7s, or sets) and they forget last card, they take back their last card and two penalty cards. They may not continue to play.

3 Class hierarchy

Here is my wonky UML:


TGameHandler
+<<constructor>> Create(ui_var:IUI,game_var:IGame)
+<<procedure>> Play()
TTextUI
<<interface>>
IUI
+<<procedure>> DisplayText(txt:string)
+<<procedure>> ClearScreen()
+<<function>> AskPassword(msg:string): string
TPestenGame
+<<constructor>> UICreate(querier:IUIQuerier)
+<<destructor>> Destroy()
<<interface>>
IUIQuerier
+<<function>> GetInt(msg:string): integer
+<<function>> GetString(msg:string): string
+<<function>> GetBool(msg:string): boolean
+<<procedure>> log()
TPestenPlayer
+<<constructor>> Create(card_pack:TPack)
+<<destructor>> Destroy()
+<<function>> GetState(): string
+<<function>> GetHand(): THand
+<<procedure>> Pickup(card_pack:TPack)
+<<procedure>> Pickup(card_pack:TPack,num:integer)
<<interface>>
IGame
+<<function>> GetGlobalState(): string
+<<function>> GetPrivateState(): string
+<<function>> GetHelp(): string
+<<procedure>> HandleTurn(querier:IUIQuerier)
THand
+<<constructor>> Create(max_pack_size:integer)
+<<destructor>> Destroy()
+<<function>> GetSize(): integer
+<<function>> Display(): string
+<<function>> FindCard(cardname:string): integer
TPack
+<<constructor>> Create(n:integer)
+<<destructor>> Destroy()
+<<function>> GetSize(): integer
+<<procedure>> Shuffle()
+<<function>> Deal(): TCard
TCard
+Rank: integer
+Suit: Integer
+<<function>> GetRank(): integer
+<<function>> GetSuit(): integer
+<<function>> GetScore(): integer
+<<function>> GetName(): string
+<<function>> GetShortName(): string

4 Implementation

All source code in Pascal and LaTeX can be found at `github.com`.

This assignment's total source clocs [1] in at 747 loc (this TeX document is an extra 237 loc).

```
{$MODE OBJFPC}

unit UCard;

interface

uses SysUtils;

const
    suits: array[0..3] of string = ('Spades', 'Clubs', 'Hearts', 'Diamonds');
    char_suits: array[0..3] of string = ('S', 'C', 'H', 'D');
    ranks: array[0..12] of string =
                        ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven',
                         'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King');
    char_ranks: array[0..12] of char = ('A', '2', '3', '4', '5', '6', '7',
                                        '8', '9', 'T', 'J', 'Q', 'K');

type
    TCard = class
        protected
            Rank, Suit: Integer;
        public
            constructor Create(r, s: integer);
            function GetRank: integer;
            function GetSuit: integer;
            function GetScore: integer;
            function GetAltScore: integer;
            function GetName: string;
            function GetShortName: string;
    end;

    TCardArray = array of TCard;
    TCardKeyFunc = function(card: TCard): integer;

function proper_mod(a, b: integer): integer;

implementation

{global functions}

function proper_mod(a, b: integer): integer;
begin
    proper_mod := a mod b;
    if proper_mod < 0 then
        proper_mod := proper_mod + b;
end;

{methods}

constructor TCard.Create(r, s: integer);
begin
    Rank := r;
    Suit := s;
end;

function TCard.GetRank: integer;
```

[1] Count Lines of Code `https://github.com/AlDanial/cloc`. This waffle was included for the pun and because I wanted to see if I could load the loc count into a LaTeX engine at compile time. Turns out I can.

```
begin
    result := Rank;
end;

function TCard.GetSuit: integer;
begin
    result := Suit;
end;

function TCard.GetScore: integer;
begin
    result := Rank * 4 + Suit;
end;

function TCard.GetAltScore: integer;
begin
    result := Suit * 4 + Rank;
end;

function TCard.GetName: string;
begin
    result := Format('%s of %s', [ranks[Rank], suits[Suit]]);
end;

function TCard.GetShortName: string;
begin
    result := Format('%s%s', [char_ranks[Rank], char_suits[Suit]]);
end;

end.
```

Listing 1: UCard.pas

```
{$MODE OBJFPC}

unit UPack;

interface

uses SysUtils, UCard;

type
    TPack = class
    protected
        cards, all_cards: TCardArray;
        bottom, ncards, num_packs: integer;
        procedure Populate;
    public
        constructor Create(n: integer);
        destructor Destroy; override;
        function GetSize: integer;
        function GetMaxSize: integer;
        procedure Shuffle;
        function Deal: TCard;
        procedure ReturnCard(card: TCard);
    end;

implementation

constructor TPack.Create(n: integer);
begin
    bottom := 0;
    ncards := 52 * n;
    num_packs := n;
    setlength(cards, ncards);
```

```
33      setlength(all_cards, ncards);
34      Populate;
35      Shuffle;
36  end;
37
38  destructor TPack.Destroy;
39  var
40      i: integer;
41  begin
42      for i := 0 to 51 do
43          all_cards[i].free;
44  end;
45
46  function TPack.GetSize: integer;
47  begin
48      result := ncards;
49  end;
50
51  function TPack.GetMaxSize: integer;
52  begin
53      result := length(cards);
54  end;
55
56  procedure TPack.Populate;
57  var
58      i, j: integer;
59  begin
60      for j := 0 to num_packs - 1 do
61          for i := 0 to 51 do begin
62              cards[j * 52 + i] := TCard.create(i mod 13, i div 13);
63              all_cards[j * 52 + i] := cards[j * 52 + i];
64          end;
65  end;
66
67  procedure TPack.Shuffle;
68  var
69      i, ind_a, ind_b: integer;
70      temp: TCard;
71  begin
72      for i := ncards - 1 downto 1 do begin
73          ind_a := proper_mod(random(i) + bottom, length(cards));
74          ind_b := proper_mod(i, length(cards));
75          temp := cards[ind_b];
76          cards[ind_b] := cards[ind_a];
77          cards[ind_a] := temp;
78      end;
79  end;
80
81  function TPack.Deal: TCard;
82  begin
83      result := cards[proper_mod(bottom + ncards - 1, length(cards))];
84      dec(ncards);
85  end;
86
87  procedure TPack.ReturnCard(card: TCard);
88  begin
89      cards[bottom] := card;
90      bottom := proper_mod(bottom - 1, length(cards));
91      inc(ncards)
92  end;
93
94  initialization
95
96  begin
97      randomize;
```

```
end;

end.
```

Listing 2: UPack.pas

```
{$MODE OBJFPC}

unit UHand;

interface

uses SysUtils, UCard;

type
    TKeyArray = array of integer;

    type THand = class
        protected
            cards: TCardArray;
            size: integer;
            procedure Sort(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, upper:
    ↪ integer);
            procedure Merge(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, mid, upper:
    ↪  integer);
        public
            constructor Create(max_pack_size: integer);
            function GetSize: integer;
            function Display: string;
            procedure PushCard(card: TCard);
            procedure InsertCard(card: TCard; i: integer);
            function RemoveCard(i: integer): TCard;
            function FindCard(cardscore: integer): integer;
            function FindCard(cardname: string): integer;
            function PopCard: TCard;
            function PopCard(cardscore: integer): TCard;
            function PopCard(cardstring: string): TCard;
            procedure ClearHand;
            function ViewCard(i: integer): TCard;
            function ViewCard(cardname: string): TCard;
            function TopCard: TCard;
            procedure SwapCards(i, j: integer);
            procedure Sort(keyfunc: TCardKeyFunc);
            procedure SortByRank;
            procedure SortBySuit;
    end;

implementation

constructor THand.Create(max_pack_size: integer);
begin
    size := 0;
    setlength(cards, max_pack_size);
end;

function THand.GetSize: integer;
begin
    result := size;
end;

function THand.Display: string;
var
    i: integer;
begin
    result := 'Hand(';
```

```
58      if size > 0 then
59          result := result + cards[0].GetShortName;
60      for i := 1 to size - 1 do
61          result := result + ', ' + cards[i].GetShortName;
62      result := result + ')';
63  end;
64
65  procedure THand.PushCard(card: TCard);
66  begin
67      cards[size] := card;
68      inc(size);
69  end;
70
71  function THand.PopCard: TCard;
72  begin
73      result := cards[size - 1];
74      dec(size);
75  end;
76
77  function THand.FindCard(cardscore: integer): integer;
78  var
79      i: integer;
80  begin
81      result := -1;
82      for i := 0 to size - 1 do
83          if cards[i].GetScore = cardscore then
84              result := 1;
85  end;
86
87  function THand.FindCard(cardname: string): integer;
88  var
89      i: integer;
90  begin
91      result := -1;
92      for i := 0 to size - 1 do
93          if cards[i].GetShortName = cardname then
94              result := 1;
95  end;
96
97  function THand.PopCard(cardscore: integer): TCard;
98  begin
99      result := RemoveCard(FindCard(cardscore));
100 end;
101
102 function THand.PopCard(cardstring: string): TCard;
103 begin
104     result := RemoveCard(FindCard(cardstring));
105 end;
106
107 procedure THand.ClearHand;
108 begin
109     size := 0;
110 end;
111
112 procedure THand.InsertCard(card: TCard; i: integer);
113 var
114     j: integer;
115 begin
116     for j := size downto i + 1 do
117         cards[j] := cards[j - 1];
118     cards[i] := card;
119     inc(size);
120 end;
121
122 function THand.RemoveCard(i: integer): TCard;
```

```
begin
    result := cards[i];
    for i := i to size - 2 do
        cards[i] := cards[i + 1];
    dec(size);
end;

function THand.ViewCard(i: integer): TCard;
begin
    result := cards[i];
end;

function THand.ViewCard(cardname: string): TCard;
begin
    result := cards[FindCard(cardname)];
end;

function THand.TopCard: TCard;
begin
    result := ViewCard(size - 1);
end;

procedure THand.SwapCards(i, j: integer);
var
    tmp_card: TCard;
begin
    tmp_card := cards[i];
    cards[i] := cards[j];
    cards[j] := tmp_card;
end;

procedure THand.Sort(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, upper: integer);
var
    mid: integer;
begin
    if upper - lower > 1 then begin
        mid := (lower + upper) div 2;
        Sort(cardbuf, keybuf, keys, lower, mid);
        Sort(cardbuf, keybuf, keys, mid, upper);
        Merge(cardbuf, keybuf, keys, lower, mid, upper);
    end;
end;

procedure THand.Merge(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, mid, upper:
    ↪ integer);
var
    i, j, k: integer;

begin
    i := lower;
    j := mid;
    k := 0;
    while (i < mid) and (j < upper) do
        if keys[i] <= keys[j] then begin
            keybuf[k] := keys[i];
            cardbuf[k] := cards[i];
            inc(i);
            inc(k);
        end else begin
            keybuf[k] := keys[j];
            cardbuf[k] := cards[j];
            inc(j);
            inc(k);
        end;

```

```
187     for i := i to mid - 1 do begin
188         keybuf[k] := keys[i];
189         cardbuf[k] := cards[i];
190         inc(k);
191     end;
192 
193     for j := j to upper - 1 do begin
194         keybuf[k] := keys[j];
195         cardbuf[k] := cards[j];
196         inc(k);
197     end;
198 
199     for i := 0 to k do begin
200         keys[lower + i] := keybuf[i];
201         cards[lower + i] := cardbuf[i];
202     end;
203 end;
204 
205 procedure THand.Sort(keyfunc: TCardKeyFunc);
206 var
207     cardbuf: TCardArray;
208     keybuf, keys: TKeyArray;
209     i: integer;
210 begin
211     setlength(keys, length(cards));
212     setlength(keybuf, length(cards));
213     setlength(cardbuf, length(cards));
214     for i := 0 to length(cards) - 1 do
215         keys[i] := keyfunc(cards[i]);
216     Sort(cardbuf, keybuf, keys, 0, length(cards));
217 end;
218 
219 function _GetScore(card: TCard): integer;
220 begin
221     result := card.GetScore;
222 end;
223 
224 procedure THand.SortByRank;
225 begin
226     Sort(@_GetScore);
227 end;
228 
229 function _GetAltScore(card: TCard): integer;
230 begin
231     result := card.GetAltScore;
232 end;
233 
234 procedure THand.SortBySuit;
235 begin
236     Sort(@_GetAltScore);
237 end;
238 
239 end.
```

Listing 3: UHand.pas

```
1 {$MODE OBJFPC}
2 
3 unit UUIQuerier;
4 
5 interface
6 
7 type
8     IUIQuerier = interface
9         function GetInt(msg: string): integer;
```

```
        function GetString(msg: string): string;
        function GetBool(msg: string): boolean;
        procedure log(msg: string);
    end;

implementation

end.
```

Listing 4: UUIQuerier.pas

```
{$MODE OBJFPC}

unit UGame;

interface

uses UPlayer, UCard, UUIQuerier, UPack, SysUtils;

type
    IGame = interface
        function GetGlobalState: string;
        function GetPrivateState: string;
        function GetHelp: string;
        procedure HandleTurn(querier: IUIQuerier);
    end;

    EGameStop = class(Exception);

    TPestenGame = class(TInterfacedObject, IGame)
    protected
        players: array of TPestenPlayer;
        num_packs: integer;
        card_pack: TPack;
        top_discard: TCard;
        suit_exemption: integer;
        curr_player_no, original_game_start: integer;
        history: array of string;
        history_start: integer;
        two_in_play: boolean;
        cur_two_rank, cur_two_suit, cur_two_acc: integer;
        curr_direction: integer;
        procedure WriteHistory(s: string);
        procedure HandleNormal(card: TCard; querier: IUIQuerier);
        procedure HandleTwo(card: TCard);
        procedure AdvanceSteps(steps: integer);
        procedure HandlePickup;
        procedure HandleCardPlay(card: TCard; querier: IUIQuerier);
        function CardValid(card: TCard): boolean;
    public
        function GetGlobalState: string;
        function GetHelp: string;
        function GetPrivateState: string;
        procedure HandleTurn(querier: IUIQuerier);
        constructor Create(n_players, start_player, n_packs: integer; querier: IUIQuerier);
        constructor UICreate(querier: IUIQuerier);
        destructor Destroy; override;
    end;

implementation

constructor TPestenGame.UICreate(querier: IUIQuerier);
begin
    Create(querier.GetInt('How many players?'),
           querier.GetInt('Which player number deals?'),
```

```
55              querier.GetInt('How many packs?'),
56              querier);
57  end;
58
59  constructor TPestenGame.Create(n_players, start_player, n_packs: integer; querier:
        ↪ IUIQuerier);
60  var
61      i: integer;
62  begin
63      two_in_play := false;
64      SetLength(players, n_players);
65      SetLength(history, n_players * 2);
66      history_start := 0;
67      curr_direction := 1;
68      for i := 0 to length(history) - 1 do
69          WriteHistory('Game start');
70      num_packs := n_packs;
71
72      card_pack := TPack.Create(n_packs);
73
74      for i := 0 to length(players) - 1 do
75          players[i] := TPestenPlayer.Create(card_pack);
76
77      curr_player_no := start_player;
78      original_game_start := start_player;
79      HandleCardPlay(card_pack.Deal, querier);
80  end;
81
82  destructor TPestenGame.Destroy;
83  var
84      i: integer;
85  begin
86      card_pack.Destroy;
87      for i := 0 to length(players) - 1 do
88          players[i].Destroy;
89  end;
90
91  function TPestenGame.GetGlobalState: string;
92  var
93      i: integer;
94  begin
95      result := 'history:' + #10;
96      for i := history_start to history_start + length(history) - 1 do
97          result := result + history[i mod length(history)] + #10;
98      result := result + 'Top of discard is '
99                       + top_discard.GetName
100                      + #10;
101 end;
102
103 function TPestenGame.GetPrivateState: string;
104 begin
105     result := players[curr_player_no].GetState;
106 end;
107
108 function TPestenGame.GetHelp: string;
109 begin
110     result := 'This is pesten, see the pdf. Cards denoted as '
111             + '([23456789TJQKA][SCHD]|take)';
112 end;
113
114 procedure TPestenGame.HandleTurn(querier: IUIQuerier);
115 var
116     user_card: string;
117 begin
118     user_card := querier.GetString('What card would you like to play?');
```

```
119
120     if user_card = 'take' then
121         HandlePickup
122     else begin
123         if players[curr_player_no].GetHand.FindCard(user_card) = -1 then begin
124             querier.log('This card is not in your hand');
125             HandleTurn(querier);
126         end else if not CardValid(players[curr_player_no].GetHand.ViewCard(user_card)) then
    ↪ begin
127             querier.log('This card is not valid to play');
128             HandleTurn(querier);
129         end else
130             HandleCardPlay(players[curr_player_no].GetHand.PopCard(user_card), querier);
131     end;
132 end;
133
134 function TPestenGame.CardValid(card: TCard): boolean;
135 begin
136     if two_in_play then
137         result := (card.GetRank = cur_two_rank)
138                 or ((card.GetRank = cur_two_rank + 1)
139                 and (card.GetSuit = cur_two_suit))
140     else begin
141         if top_discard.GetRank = 10 then
142             result := (card.GetRank = 10) or (card.GetSuit = suit_exemption)
143         else
144             result := (card.GetRank = top_discard.GetRank)
145                   or (card.GetSuit = top_discard.GetSuit);
146     end;
147 end;
148
149 procedure TPestenGame.WriteHistory(s: string);
150 begin
151     history[history_start] := s;
152     history_start := (history_start + 1) mod length(history);
153 end;
154
155 procedure TPestenGame.HandlePickup;
156 begin
157     if two_in_play then begin
158         WriteHistory(Format('Player %d picks up %d cards', [curr_player_no, cur_two_acc]));
159         players[curr_player_no].pickup(card_pack, cur_two_acc);
160         two_in_play := false;
161     end else begin
162         WriteHistory(Format('Player %d picks up a card', [curr_player_no]));
163         players[curr_player_no].pickup(card_pack);
164     end;
165 end;
166
167 procedure TPestenGame.HandleCardPlay(card: TCard; querier: IUIQuerier);
168 var
169     nsize, i: integer;
170 begin
171     if players[curr_player_no].GetHand.GetSize = 1 then begin
172         WriteHistory(Format('Player %d wins', [curr_player_no]));
173         if querier.GetBool('Do you want to continue playing?') then begin
174             nsize := card_pack.GetMaxSize;
175             card_pack.Destroy;
176             for i := 0 to length(players) - 1 do
177                 players[i].Destroy;
178             Create(length(players), original_game_start, nsize div 52, querier)
179         end else
180             raise EGameStop.Create('Game is over');
181     end;
182     WriteHistory(Format('Player %d plays a %s', [curr_player_no, card.GetName]));
```

```
183
184     if two_in_play then
185         HandleTwo(card)
186     else
187         HandleNormal(card, querier);
188 end;
189
190 procedure TPestenGame.AdvanceSteps(steps: integer);
191 begin
192     curr_player_no := proper_mod(curr_player_no + curr_direction * steps, length(players));
193 end;
194
195 procedure TPestenGame.HandleTwo(card: TCard);
196 begin
197     if card.GetRank = cur_two_rank then
198         cur_two_acc := cur_two_acc + cur_two_rank + 1
199     else if (card.GetRank = cur_two_rank + 1)
200         and (card.GetSuit = cur_two_suit) then begin
201         inc(cur_two_rank);
202         cur_two_acc := cur_two_acc + cur_two_rank + 1;
203     end;
204     AdvanceSteps(1);
205     top_discard := card;
206 end;
207
208 procedure TPestenGame.HandleNormal(card: TCard; querier: IUIQuerier);
209 begin
210     case card.GetRank of
211         1: begin
212             two_in_play := true;
213             cur_two_rank := 1;
214             cur_two_suit := card.GetSuit;
215             cur_two_acc := 2;
216             AdvanceSteps(1);
217         end;
218         6: begin
219             WriteHistory(Format('Player %d gets another turn', [curr_player_no]));
220         end;
221         7: begin
222             WriteHistory(Format('Player %d skips a turn', [(curr_player_no + 1) mod length(
    ↪ players)]));
223             AdvanceSteps(2);
224         end;
225         9: begin
226             WriteHistory(Format('Play goes back one turn', [(curr_player_no + 1) mod length(
    ↪ players)]));
227             AdvanceSteps(-1);
228         end;
229         10: begin
230             repeat
231                 suit_exemption := querier.GetInt(
232                         'What suit do you want to make it (ref:SCHD)');
233             until suit_exemption in [0..3];
234             WriteHistory(Format('Player %d sets suit to %s', [curr_player_no, suits[
    ↪ suit_exemption]]));
235         end;
236         12: begin
237             curr_direction := -curr_direction;
238             AdvanceSteps(1);
239         end;
240         else
241             AdvanceSteps(1);
242     end;
243     top_discard := card;
244 end;
```

```
245
246 end.
```

Listing 5: UGame.pas

```
1  {$MODE OBJFPC}
2
3  unit UPlayer;
4
5  interface
6
7  uses UHand, UPack, UCard;
8
9  type
10     TPestenPlayer = class
11     protected
12         hand: THand;
13     public
14         constructor Create(card_pack: TPack);
15         destructor Destroy; override;
16         function GetState: string;
17         function GetHand: THand;
18         procedure Pickup(card_pack: TPack);
19         procedure Pickup(card_pack: TPack; num: integer);
20     end;
21
22 implementation
23
24 constructor TPestenPlayer.Create(card_pack: TPack);
25 begin
26     hand := THand.Create(card_pack.GetMaxSize);
27     Pickup(card_pack, 7);
28 end;
29
30 destructor TPestenPlayer.Destroy;
31 begin
32     hand.Destroy;
33 end;
34
35 procedure TPestenPlayer.Pickup(card_pack: TPack);
36 begin
37     hand.PushCard(card_pack.deal);
38 end;
39
40 procedure TPestenPlayer.Pickup(card_pack: TPack; num: integer);
41 var
42     i: integer;
43 begin
44     for i := 1 to num do
45         pickup(card_pack);
46 end;
47
48 function TPestenPlayer.GetState: string;
49 begin
50     result := 'Your hand is: ' + hand.Display;
51 end;
52
53 function TPestenPlayer.GetHand: THand;
54 begin
55     result := hand;
56 end;
57
58 end.
```

Listing 6: UPlayer.pas

```
{$MODE OBJFPC}

unit UUI;

interface

uses UGame, UUIQuerier, SysUtils, StrUtils;

type
    {User interface interface}
    IUI = interface(IUIQUerier)
        procedure DisplayText(txt: string);
        procedure ClearScreen;
        function AskPassword(msg: string): string;
    end;

    {Plain ansi terminal implementation of a UI}
    TTextUI = class(TInterfacedObject, IUI, IUIQuerier)
    public
        function GetInt(msg: string): integer;
        function GetString(msg: string): string;
        function GetBool(msg: string): boolean;
        function AskPassword(msg: string): string;
        procedure log(msg: string);
        procedure DisplayText(txt: string);
        procedure ClearScreen;
    end;

implementation

function TTextUI.GetInt(msg: string): integer;
var
    response: string;
begin
    writeln(msg);
    write('Enter integer > '); readln(response);
    try
        result := strtoint(response);
    except
        on E: EConvertError do
            result := GetInt(msg);
    end;
end;

function TTextUI.GetString(msg: string): string;
var
    response: string;
begin
    writeln(msg);
    write('Enter text > '); readln(response);
    result := response;
end;

function TTextUI.GetBool(msg: string): boolean;
var
    response: string;
begin
    writeln(msg);
    write('Enter message containing ''y'' to confirm > '); readln(response);
    result := AnsiContainsStr(LowerCase(response), 'y');
end;

procedure TTextUI.log(msg: string);
begin
    writeln('(game engine) ' + msg);
```

```
end;

procedure TTextUI.DisplayText(txt: string);
begin
    writeln(txt);
end;

procedure TTextUI.ClearScreen;
begin
    {Ansi escape code to clear terminal}
    write(#27 + '[1;1H');
end;

function TTextUI.AskPassword(msg: string): string;
begin
    writeln(msg);
    write('Enter password > '); readln(result);
end;

end.
```

Listing 7: UUI.pas

```
{$MODE OBJFPC}

unit UGameHandler;

interface

uses UUI, UGame;

type
    TGameHandler = class
    protected
        UI: IUI;
        game_engine: IGame;
    public
        constructor Create(ui_var: IUI; game_var: IGame);
        procedure Play;
    end;

implementation

constructor TGameHandler.Create(ui_var: IUI; game_var: IGame);
begin
    UI := ui_var;
    game_engine := game_var;
end;

procedure TGameHandler.Play;
begin
    while True do begin
        UI.ClearScreen;
        UI.DisplayText(game_engine.GetHelp);
        UI.DisplayText(game_engine.GetGlobalState);
        UI.DisplayText(game_engine.GetPrivateState);
        game_engine.HandleTurn(UI);
    end;
end;

end.
```

Listing 8: UGameHandler.pas

```
{$MODE OBJFPC}
```

```
2
3  program PPesten;
4
5  uses UGameHandler, UUI, UGame;
6
7  var
8      UI: TTextUI;
9      game: TPestenGame;
10     handler: TGameHandler;
11 begin
12     UI := TTextUI.Create;
13     game := TPestenGame.UICreate(UI);
14     handler := TGameHandler.Create(UI, game);
15     handler.play;
16
17     handler.destroy;
18     UI.destroy;
19     game.destroy;
20 end.
```

Listing 9: PPesten.pas

5 Usage

```
1 How many packs?
2 Enter integer > 2
3 Which player number deals?
4 Enter integer > 1
5 How many players?
6 Enter integer > 3
```

Listing 10: Initialisation routine

After this point the program sends the terminal the escape code for screen clearance, so an empty screen appears:

```
1  This is pesten, see the pdf. Cards denoted as ([23456789TJQKA][SCHD]|take)
2  history:
3  Game start
4  Game start
5  Game start
6  Game start
7  Game start
8  Player 1 plays a Six of Hearts
9  Top of discard is Six of Hearts
10
11 Your hand is: Hand(6C, 3H, 2D, 9H, KD, TH, 7D)
12 What card would you like to play?
13 Enter text > 3H
14 ...
```

Listing 11: User taking a turn

6 Regrets/todo

This assignment is absolutely incomplete, for no particularly good reason other than time constraints and other things competing for my attention.

- User interface wise:

  - I would have liked to implement a nicer UI (IUI interface) with ncurses or a GUI toolkit like GTK+
  - This would go hand in hand with a better communication protocol between game engine and IUI, likely using XML.
  - I would also liked to have implemented proper user security using a password scheme, with cryptographically secure hashing algorithms. A user would only be allowed to see their private information with this password, resulting in a genuinely fair solution for a one-screen card game.
  - It would also have been cool for the program to automatically generate possible moves, or at least detect when the user can't play and pick up a card for them.
- The program is functional, but edge cases remain largely unaccounted for:
  - The pack running out of cards just results in an access violation.
  - Entering a negative user index results in catastrophic failure.
  - Entering a maliciously large number of packs results in either conversion error or dumb memory consumption likely resulting in kernel killing.

  The list goes on.
- Various rules remain unimplemented:
  - There isn't a joker card, let alone a handling mechanism for the rule (see 2.2).
  - Sets of cards aren't implemented (see 2.3).
  - I haven't even thought of way to capure the "last card" rule (see 2.4).