OGO 2.2 spring 2009

Specificatiedocument

Technische informatica, TU/e

Eindhoven, April 14, 2009

Group 2

Etienne van Delden, 0618959

Edin Dudojevic, 0608206

Jeroen Habraken, 0586866

Dion Jansen, 0590077

Stef Louwers, 0590864

Anson van Rooij, 0596312

This page has been left blank.

Contents

1 Introduction **5**

1.1 The game from the player's point of view 5

1.2 The rules 5

1.3 The technology 6

2 Server-Client Interaction **7**

2.1 The socket protocol 7

2.1.1 Introduction 7

2.1.2 Message format 7

2.1.3 Codes for the addressee and sender 7

2.1.4 Message types 8

2.1.5 Message format of the board 10

2.1.6 Role of the controller 10

3 Player-module **11**

3.1 functions and procedures 11

3.1.1 Click 11

3.1.2 Connect 12

3.1.3 Stop 12

3.2 Socketinput 12

3.2.1 Team 12

3.2.2 CanMove 13

3.2.3 Stop 13

4 Boardmodule **15**

4.1 The specification 15

4.1.1 Helpful Functions 16

4.1.2 Initialisation 17

4.1.3 Flood 17

4.1.4 Update Status 18

4.1.5 Initialisation 18

4.1.6 Joining 18

4.1.7 Disconnecting 19

4.1.8 Taking Turns 19

4.1.9 Moving 20

Chapter 1

Introduction

1.1 The game from the player's point of view

This specification concerns a two-player game. The game takes place on an island where foxes live, and dolphins live in the sea surrounding it. Foxes want to eat dolphins, dolphins want to eat foxes. There are tides on the island, so the amount of territory available to both species varies. A dolphin can only catch foxes in the sea and cannot move onto the island. When a dolphin is stranded on the island (because of the tides), it cannot move. Of course, a fox can only catch dolphins on land. When a fox is too far out in the water, it cannot move either. As a player, you control one of the species (your "team"), and your objective is to have your animals eat all members of the other team.

1.2 The rules

The game takes place in real time, on a square board, divided into x by x square tiles, where x is a constant. Some of these tiles contain land, the rest contain water and all land tiles are connected (forming "the island"). Every once in a while, "the water level rises": x land tiles turn into water tiles. After a few of these occurrences, "the water level lowers": x water tiles turn into land tiles.

A tile can contain a fox or a dolphin. When the game starts, there are a certain amount of foxes on the island and an equal amount of dolphins in the water. A player can ask a member of his team to move to a tile adjacent to its current tile ("making a move", or "taking a turn"). After a move, the player must wait for a short time before moving again, to ensure fairness and to encourage thinking about strategy. There is, however, no maximum period of time between a player's turns and the two players' turns can be taken asynchronously.

The animal will do what is asked of it, if it can. Again: a dolphin cannot move onto land, and a fox cannot move too far into the water (more then two steps off the land). Two animals of the same species can never be on the same tile, and a fox can never move into a water tile containing a dolphin. If two animals of different species would land on the same tile in any other case, there are two options but the result will be the same, one of the animals will be eaten and will be removed from the board. On land, the dolphin will be eaten by the fox; in the water, the fox will be the one to go. When the last animal of any team is eaten, the game ends.

1.3 The technology

We have designed a number of programs which facilitate the playing of this game by two human players (there is no AI). There is a design for a server program and one for a client program. These communicate with each other through sockets. The server program administrates the board (including the tides), and admits or rejects moves, through the board module. It also starts the game and passes messages between modules through its controller module. When the game is being played, there will be two instances of the client program involved. Client programs show the board to the player through the viewer module, and pass requests to the server through the player-module.

It will be your job to finish one of these programs. The parts still to be made are the board-module (a pascal unit called “board”), and the player module (a pascal unit called “player”). This specification concerns (mainly) these parts.

Chapter 2

Server-Client Interaction

2.1 The socket protocol

2.1.1 Introduction

This protocol concerns the communication between the server and the clients. We use socket communication in the format defined below.

All communication is sent either by the controller-module (on behalf of the board-module) to a player- or viewer-module, or from a player- or viewer-module to the controller-module (addressing the board-module).

2.1.2 Message format

All messages have the same format: a string consisting of a header and an optional data part, separated by a single space. Messages are ended by a newline character (\n)

The header consists of three characters. The first represents the sender of the message, the second contains the addressee of the message, and the third the type of the message.

The characters used for the sender and the addressee are defined in section 2.1.3, and the message types are defined in section 2.1.4

2.1.3 Codes for the addressee and sender

- B for the board unit
- D for the dolphin player
- E for the dolphin viewer

- F for the fox player
- G for the fox viewer
- P for a new player
- Q for a new viewer

2.1.4 Message types

B

Type: Board update — server to viewer
Data: The board, see section 2.1.5 for details of the encoding.
Sent on behalf of: B
Sent to: E or G

Example message: “BEB WWWDWLFGWLLGDWDW”, if board size would be 4
Description of example: The board module sends a board update to the dolphin viewer.

F

Type: Board update — server to viewer
Data: None
Sent on behalf of: B
Sent to: E or G

Example message: “BEF”
Description of example: The board module tells the dolphin viewer to display a “fail!” message.

J

Type: Request to join the game — player to server
Data: None
Sent by: P
Addressee: B

Example message: “PBJ”
Description of example: This message is sent right after a new client connects to the server.

M

Type: Move unit — player to server
Data: Two coordinates, both written as two integers seperated by a single space.
Sent by: D or F
Addressee: B

Example message: “DBM 4 1 4 2”
Description of example: The dolphin player asks the board to move the unit at the first coordinate [(4,1)] to the second coordinate [(4,2)].

N

Type: Next turn — server to player
Data: None
Sent on behalf of: B
Sent to: (D and E) or (F and G)

Example message: “BDN”
Description of example: The board announces that the dolphin player’s previous move has been processed and that the dolphin player may make another step.

R

Type: View refreshment request — player to server
Data: None
Sent by: D or F
Addressee: B

Example message: “DBR”
Description of example: The dolphin player unit asks the board module to send an updated board to the dolphin viewer.

S

Type: Game end — server to player
Data: The winner: “D” or “F”
Sent on behalf of: B
Addressee: D or F

Example message: “FBS F”
Description of example: The server tells the fox player that the game is over, and that he/she has won.

T

Type: Announce the team — server to (player and viewer)
Data: The team the player will play ("D" for dolphin or "F" for fox), or "R" for rejected if the player can not join the game.
Sent on behalf of: B
Sent to: P and Q

Example message: "BPT D"
Description of example: This message assigns the dolphin team to a joining player.

2.1.5 Message format of the board

In the board update message, the complete board in transmitted to the viewers. In this message the board is encoded as a string where every character represents a tile. The string is of length n^2, where n is the board size. The following characters are allowed:

- W: a water tile
- L: a land tile
- D: a water tile with a dolphin
- E: a land tile with a dolphin
- F: a land tile with a fox
- G: a water tile with a fox

2.1.6 Role of the controller

The controller is the part in the server responsible for the communication with the clients. It receives the messages from the client and forward them to the board module if necessary. The controller also sends the messages from the board to the clients.

When the game is starting, the controller opens the sockets for the client to connect to and it handles the connecting clients by assigning them a team, or rejecting them if for example a third player tries to join.

Chapter 3

Player-module

A player-module handles the communication from a (human) player of the game to the controller- and board-modules. The player-module is instantiated as a pascal unit named "player", which is part of the client program. When the game is being played, two such units are at work, one for each client.

A player-module has no functionality not specified in this document.

Each "player" unit has the following variables:

CanMove: a Boolean, initialized as FALSE. *CanMove* answers the question "Can I send a move request?"

Team: a Char (representing the team), initialized as "N" (No team). Possible values of Team are "N", "D", and "F".

$SelectionX$: an Integer, initialized as 0. Represents a tile recently clicked on.

$SelectionY$: an Integer, initialized as 0. Represents a tile recently clicked on.

Each "player" unit has a set of procedures, we will discuss those below.

3.1 functions and procedures

3.1.1 Click

Informal: When a player clicks on the board in the interface, this is the procedure that formats and sends the generated move request.

Functiondefinition: $procedureClick(Sender : TObject; ACol, ARow : Integer; varcanSelect : Boolean);$

If *canMove* is FALSE, *canSelect* is FALSE and the procedure returns.

canSelect is set to TRUE.

If *SelectionX* and *SelectionY* are both 0, *SelectionX* becomes *ACol* and *SelectionY* becomes *ARow*, then the procedure returns.

If *SelectionX* and *SelectionY* are not 0, the message $Team + “BM''+“ ”+SelectionX+“ ”+SelectionY+“ ”+ACol+“ ”+ARow$ is sent over the socket connection. Then *canMove* is set to FALSE and the procedure returns.

3.1.2 Connect

Informal: This is the procedure that is called when the client is started. It tries to start the communication.

Functiondefinition: $procedureConnect(I : String;\ P : Integer;\ varSuccess : Boolean)$

This procedure creates and initialises a socket and tries to initiate a socket connection to the socket on the IP-Address given in string I and Port Number P. If the connection is made successfully, a thread is made to listen to incoming messages, as defined in section 3.2.

After that, the message $“PCJ''$ is sent over the socket connection.

If everything succeeds, TRUE is returned, if not, FALSE.

3.1.3 Stop

Informal: This is the procedure that is called when the client is stopped. It tries to end the communication neatly.

Functiondefinition: $procedureStop$;

This procedure disconnects the socket, then calls viewer.finish (a procedure in the unit viewer, in the same program), without arguments.

3.2 Socketinput

If and when the “player” unit gets input from the socket, it does the following:

3.2.1 Team

If the socketinput is $“CPT'' + A$, where A is a Char, then Team becomes A. Note the space at the fourth place in the input string.

3.2.2 CanMove

If the socketinput is "B'' + *Team* + "N'', then *CanMove* becomes TRUE.

3.2.3 Stop

If the socketinput is "B'' + *Team* + "S'', then procedure *Stop* (see 3.1.3) is started.

Other socket input than that of the kind specified above should be discarded.

Chapter 4

Boardmodule

4.1 The specification

Board

$Surface == Water \mid Land$
$Occupant == Fox \mid Dolphin \mid Empty$
$Player == Fox \mid Dolphin$
$NumberOfPlayers == 0 \mid 1 \mid 2$
$Status == WaitingForPlayers \mid InGame \mid EndGame$
$Tile : Occupant \times Surface$
$Coordinate : \mathbb{N} \times \mathbb{N}$
$size : BoardSize$
$status : Status$
$numberOfPlayers : NumberOfPlayers$
$floodstate : \mathbb{N}$
$Player1Time : Timer$
$Player2Time : Timer$

$MinBoardSize : \mathbb{N}$

$MinBoardSize = 10$

$MaxBoardSize : \mathbb{N}$

$MaxBoardSize = 50$

$BoardSize : \mathbb{N}$

$MinBoardSize \leq BoardSize \leq MaxBoardSize$

$board : Coordinate \nrightarrow Tile$

$(\forall (x_1, y_1), (x_2, y_2) : \mathrm{dom}\, board \mid$
$\quad tile(x_1, y_1).1 = Land \land tile(x_2, y_2).1 = Land \bullet (x_1, y_1)\mathrm{path}(x_2, y_2))$

Where $(x_1, y_1)\mathit{path}(x_2, y_2)$ is defined as: there exists a path containing only land tiles from (x_1, y_1) to (x_2, y_2). A path is formed by moving only left, right, up or down over the board, moving from one tile to another.

All functions in the board class are specified and explained in sections 4.1.1 to 4.1.9

4.1.1 Helpful Functions

We have added a number of different functions that will only be called by other functions. These are solely used by the board module.

numberOfOccupants

numberOfOccupants

$t : \mathit{Tile}$
$o? : \mathit{Occupant}$
$n! : \mathbb{N}$

$t \in \operatorname{ran} \mathit{board} \land t.1 = o?$
$n! = \#(\mathit{board} \rhd \{t\})$

Returns the number of occupants of the given type.

island

island

$t : \mathit{Tile}$
$i! : \mathit{Coordinate} ⤀ \mathit{Tile}$

$(t \in \operatorname{ran} \mathit{board} \land t.2 = \mathit{Land})$
$i! = \mathit{board} \rhd \{t\}$

Returns the set of coordinates that make up the island.

tile

tile

$x? : \mathbb{N}$
$y? : \mathbb{N}$
$t : Tile$
$o! : Tile$

$(t \in \operatorname{ran}((x, y) \lhd board)$
$o! = t$

Returns the tile for the given coordinate.

4.1.2 Initialisation

INIT

$status' = WaitingForPlayers$
$numberOfPlayers' = 0$
$floodstate' = 0$
$NumberOfOccupants(Dolphin) = NumberOfOccupants(Fox) = BoardSize$
$(\forall (x, y) : \operatorname{dom} board \bullet tile(x, y).1 = Land \Rightarrow tile(x, y).2 \neq Dolphin)$
$(\forall (x, y) : \operatorname{dom} board \bullet tile(x, y).1 = Water \Rightarrow tile(x, y).2 \neq Fox)$

INIT initialises the board by setting all important variables. It guarantees that no Dolphin starts on Land and no Fox in the Water. Initially, there are *BoardSize* Foxes and *BoardSize* Dolphins

4.1.3 Flood

Flood

$\Delta floodstate$
$\Delta island$

$floodstate' = (floodstate + 1) \bmod 6$
$((floodstate' \leq 2 \wedge (\#island' = \#island - boardsize) \vee$
$(floodstate' \geq 3 \wedge (\#island' = \#island + boardsize))$

Flood is the function that deals with changing land to water and vice versa. When *floodstate* is 0,1 or 2 there is flooding. When *floodstate* is 3,4,5 there is ebbing.

4.1.4 Update Status

UpdateStatus

$$
\begin{array}{l}
((\mathit{numberOfPlayers} < 2) \land (\mathit{Status} = \mathit{InGame}) \\
\quad \mathit{Status}' = \mathit{EndGame}) \lor \\
((\mathit{numberOfPlayers} = 2) \land (\mathit{Status} = \mathit{WaitingForPlayers}) \\
\quad \mathit{Status}' = \mathit{InGame}) \lor \\
((\mathit{numberOfPlayers} = 2) \land (\mathit{Status} = \mathit{Ingame}) \land \\
\quad (\mathit{NumberOfOccupants}(\mathit{Dolphin}) = 0 \lor \mathit{NumberOfOccupants}(\mathit{Fox}) = 0) \\
\quad \mathit{Status}' = \mathit{EndGame})
\end{array}
$$

The *UpdateStatus* procedure checks when called if there is need for a state-change. It checks the number of players, the number of elements and in which state the game currently resides.

If the number of players is smaller than 2 but the game is *InGame*, this means that a player has left the game while playing. The game needs to end and *UpdateStatus* will set the *Status* to *EndGame* which means the only player left gets a message the game has ended and the board will no longer process messages from the clients.

If the number of players is equal to 2 and the *Status* is equal to *WaitingForPlayers* this means that the board was waiting for players, but it has them both now. So now the game can start and *UpdateStatus* will set the *Status* to *InGame*. Now the players will get a message the game has started and the board will start processing messages received from the clients.

If the number of players is equal to 2 and the *Status* is *InGame* but one of the players has no animals anymore this means a player has won. Therefore the board will end the game by setting Status to *EndGame*. The clients will receive a message that the game has ended and the board will no longer process messages from the clients.

4.1.5 Initialisation

INIT initialises the board by setting all important variables. It guarantees that no Dolphin starts on Land and no Fox in the Water. Initially, there are *BoardSize* Foxes and *BoardSize* Dolphins.

4.1.6 Joining

Join

$PlayerType! : Char$

$$(NumberOfPlayers = 0) \Rightarrow (PlayerType! = F \wedge NumberOfPlayers' = NumberOfPlayers + 1)$$
$$(NumberOfPlayers = 1) \Rightarrow (PlayerType! = D \wedge NumberOfPlayers' = NumberOfPlayers + 1 \wedge UpdateViews)$$
$$(NumberOfPlayers = 2) \Rightarrow (PlayerType! = R)$$

When a request to join has been filed at the Board, it calls the function *Join*. Join checks the current number of Players and acts accordingly; Assigns Foxes to that player when there are 0 players, assign Dolphins when 1 player has already joined or return Reject when there are 2 players already playing.

4.1.7 Disconnecting

Disconnect

$$NumberOfPlayers' = NumberOfPlayers - 1$$
$$UpdateViews$$

Disconnect disconnects a player. It lowers the number of players and changes the status. When a player leaves during a game, the other player automatically wins.

4.1.8 Taking Turns

Turn

$MessageType? : Char$
$Player? : Char$
$from? : Coordinate$
$to? : Coordinate$
$status! : \mathbb{B}$

$$((Status = InGame) \wedge$$
$$((player? = F) \wedge (MessageType? = M) \wedge (Player1Time = 0)) \Rightarrow$$
$$(TryMove(from?, to?, Fox) \wedge (Player1Time' = 1))$$
$$((player? = D) \wedge (MessageType? = M) \wedge (Player2Time = 0)) \Rightarrow$$
$$(TryMove(from?, to?, Dolphin) \wedge (Player1Time' = 1))$$
$$\vee IllegalMove$$

The function *Turn* handles fairness. It uses 2 timers, *Player1Time* and *Player2Time*. When a player requests a move, the timer is set on a pre-defined value and then TryMove is called. A player's turn ends when his timer has reached 0. The player is notified that he can Move again. Only when a player's time is 0, can a move be done. For taking turns, it does not matter whether a move is valid or not.

The timers are not further specified in Z (because of Z restrictions): All timers should be lowered at specific intervals. When a PlayerTimer reaches 0, a message needs to be sent to the player that he may move again.

4.1.9 Moving

$TryMove \triangleq (CanMove \land (Move \land UpdateViews \land UpdateStatus)) \lor IllegalMove$

with *CanMove* defined as:
$CanMove \triangleq (AdjacentTile \land diffBeast \land (DolphinToSea \lor FoxMovement))$

TryMove is called whenever a player wants to move. It first checks if the move is valid (by calling *CanMove*). If it is, Move is executed (and checks wether there is now a winner), else a message is send to the player that he has made an invalid move.

CanMove consists of several smaller functions, to check if a move is valid or not, by checking the to? coordinate if it is adjacent, if there is a different occupant and if the creature being moved, moves according to it's movement set.

Adjacent Tiles

AdjacentTile

$from? : Coordinate$
$to? : Coordinate$
$report! : \mathbb{B}$

$report! = (\mid from.1 - to.1 \mid + \mid from.2 - to.2 \mid = 1)$

AdjacentTile checks if the movement requested is to an adjacent tile.

Different Beasts

diffBeast

$from? : Coordinate$
$to? : Coordinate$
$report! : \mathbb{B}$

$report! = (tile(from.1, from.2).1 \neq tile(to.1, to.2).1)$

diffBeast checks the occupant of the *to* tile. If it is different than the creature being moved, than it is a valid move. Please note that "*Empty*" is also an occupant and thus, this checks both legal movements to an empty spot and eating a different creature.

Dolphin From Sea To Sea

DolphinToSea

$from? : Coordinate$
$to? : Coordinate$
$player? : Player$
$report! : \mathbb{B}$

$report! = tile(to.1, to.2).2 = Water$
$\wedge tile(from.1, from.2).2 = Water$
$\wedge tile(from.1, from.2).1 = Dolphin$

DolphinToSea looks to the creature that is being moved, checks if it is a Dolphin in the Water and if it is moving to Water. A Dolphin cannot strand itself on land, nor can it move when on land.

Movements of the fox

FoxMovement

$from? : Coordinate$
$to? : Coordinate$
$player? : Player$
$report! : \mathbb{B}$

$Neighbourhood = \{z \in dom(board) : \exists\, x, y(x, y \in \mathbb{N} : ((from.1 - x) + (from.2 - y) \leq 2) \wedge$
$z.1 = x \wedge z.2 = y : z)\}$
$report! = (player? = Fox) \wedge$
$(tile(from.1, from.2).1 = Fox) \wedge$
$\exists\, z(z \in Neighbourhood : (tile(z.1, z.2).2 = Land) \wedge$
$((tile(to.1, to.2).1 \neq Dolphin) \vee (tile(to.1, to.2).2 \neq Water)))$

FoxMovement checks *all* the possible movements of a Fox. A Fox can move on the land, into water, in water and to land. To only acception is when a Fox in the water tries to eat a Dolphin, that is not allowed. FoxMovement checks this by making a new set, with all tiles that this creature can reach if it would make two steps. If there is a land tile within this set, a Fox can move. If the Fox is on the land, this is always true. If it is in the Water, a land tile must be reachable.

Update Views

UpdateViews calls a function in the Controller to send a new version of the Board to the Viewers, according to the specified protocol. *(Note: there is no Z-schema associated with this specification)*

Move

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Move}} \\
\hline
\textit{player}? : \textit{Player} \\
\textit{from}? : \textit{Coordinate} \\
\textit{to}? : \textit{Coordinate} \\
\hline
\textit{tile}(\textit{from}.1, \textit{from}.2)'.1 = \textit{Empty} \land \textit{tile}(\textit{to}.1, \textit{to}.2)'.1 = \textit{tile}(\textit{from}.1, \textit{from}.2).1 \\
\hline
\end{array}
$$

Move makes the requested move.

Illegal Move

IllegalMove calls a function in the Controller to send a message to the corresponding viewer that the player's move is invalid.