Abstract
This document contains the specification for the board module for our game.

1 The specification

1.1 The Board

Board

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

$$MinBoardSize : \mathbb{N}$$
$$MinBoardSize = 10$$

$$MaxBoardSize : \mathbb{N}$$
$$MaxBoardSize = 50$$

$$BoardSize : \mathbb{N}$$
$$MinBoardSize \leq BoardSize \leq MaxBoardSize$$

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

Where $(x_1, y_1)path(x_2, y_2)$ 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 **??** to **??**

1.2 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.

1.2.1 numberOfOccupants

numberOfOccupants

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

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

Returns the number of occupants of the given type.

1.2.2 numberOfSurfaces

numberOfSurfaces

$t : Tile$
$s? : Surface$
$n! : \mathbb{N}$

$t \in \operatorname{ran} board \land t.2 = s?$
$n! = \#(board \rhd \{t\})$

Returns the number of surfaces of the given type.

1.2.3 island

island

$t : Tile$
$i! : Coordinate \pfun Tile$

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

Returns the set of coordinates that make up the island.

1.2.4 tile

tile

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

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

Returns the tile for the given coordinate.

1.3 Initialisation

INIT

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

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 10 Foxes and 10 Dolphins

1.4 Flood

Flood

$\Delta floodstate$
$\Delta island$

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

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

1.5 Update Status

UpdateStatus

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

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.

1.6 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 10 Foxes and 10 Dolphins

1.7 Joining

Join

$$
\begin{array}{l}
PlayerType! : Char \\
\hline
(NumberOfPlayers = 0) \Rightarrow (PlayerType! = F \land NumberOfPlayers' = NumberOfPlayers + 1) \\
(NumberOfPlayers = 1) \Rightarrow (PlayerType! = D \land NumberOfPlayers' = NumberOfPlayers + 1 \land \\
UpdateViews) \\
(NumberOfPlayers = 2) \Rightarrow (PlayerType! = R)
\end{array}
$$

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.

1.8 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.

1.9 Taking Turns

Turn

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

$$((Status = InGame) \land$$
$$((player? = F) \land (MessageType? = M) \land (Player1\,Time = 0)) \Rightarrow$$
$$(TryMove(from?, to?, Fox) \land (Player1\,Time' = 1))$$
$$((player? = D) \land (MessageType? = M) \land (Player2\,Time = 0)) \Rightarrow$$
$$(TryMove(from?, to?, Dolphin) \land (Player1\,Time' = 1))$$
$$\lor IllegalMove$$

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.

1.10 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.

1.10.1 Adjacent Tiles

AdjacentTile

$$
\begin{array}{l}
from? : Coordinate \\
to? : Coordinate \\
report! : \mathbb{B} \\
\hline
report! = (\mid from.1 - to.1 \mid + \mid from.2 - to.2 \mid = 1)
\end{array}
$$

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

1.10.2 Different Beasts

diffBeast

$$
\begin{array}{l}
from? : Coordinate \\
to? : Coordinate \\
report! : \mathbb{B} \\
\hline
report! = (tile(from.1, from.2).1 \neq tile(to.1, to.2).1)
\end{array}
$$

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.

1.10.3 Dolphin From Sea To Sea

DolphinToSea

$$
\begin{array}{l}
from? : Coordinate \\
to? : Coordinate \\
player? : Player \\
report! : \mathbb{B} \\
\hline
report! = tile(to.1, to.2).2 = Water \\
\land tile(from.1, from.2).2 = Water \\
\land tile(from.1, from.2).1 = Dolphin
\end{array}
$$

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.

1.10.4 Movements of the fox

FoxMovement

$$
\begin{array}{l}
from? : Coordinate \\
to? : Coordinate \\
player? : Player \\
report! : \mathbb{B} \\
\hline
Neighbourhood = \{z \in dom(board) : \exists\, x, y (x, y \in \mathbb{N} : ((from.1 - x) + (from.2 - y) \leq 2) \land \\
z.1 = x \land z.2 = y : z)\} \\
report! = (player? = Fox) \land \\
(tile(from.1, from.2).1 = Fox) \land \\
\exists\, z (z \in Neighbourhood : (tile(z.1, z.2).2 = Land) \land \\
((tile(to.1, to.2).1 \neq Dolphin) \lor (tile(to.1, to.2).2 \neq Water)))
\end{array}
$$

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.

1.10.5 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)*

1.10.6 Move

Move

$$
\begin{array}{l}
player? : Player \\
from? : Coordinate \\
to? : Coordinate \\
status! : \mathbb{B} \\
\hline
x \in board(\!|\ to?\ |\!) \bullet y \in board(\!|\ from?\ |\!) \bullet (x'.1 = y.1 \land y'.1 = Empty)
\end{array}
$$

Move makes the requested move.

1.10.7 Check on Winner

CheckWinner

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

$report' = \forall\, x, y((x, y) \in \mathbb{N} \land (x, y) \leq BoardSize : k \in board(\!|\ (x, y)\ |\!) \bullet k.1 = Empty)$

CheckWinner does a check to see if the move we just made results in 1 player winning the game.

1.10.8 Illegal Move

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

1.11 Flooding

The *Flood* procedure takes care of the dynamic waterlevel in the game. At a specific interval the procedure is called and it raises the *floodstate* by 1. This means that the number of water tiles increases/decreases by *BoardSize*, as a result the number of land tiles will decrease/increase by *BoardSize*,