SFWR ENG 2AA4 Assignment 4 Model Specification

Albert Zhou

April 2, 2020

This document contains a Module Interface Specification for the model, view, and controller components of the game 'dots'. To improve the interaction between these components, there will be access programs that will make the specification fail to achieve some qualities. The board of dots is represented as a 2D sequence of colours. The user specifies lines of colours to remove in order to achieve a set of objectives within a move limit. Once the objectives are reached the next level begins.

DirectionT, ColourT, PointT, LineT, PathT, and BoardT represent the game board and the moves within. ObjectiveT and LevelT represent the game's objectives and levels. ModelT, ViewT, and ControllerT are the MVC modules that make up the game. DotsT plays the game.

The following are likely changes:

- A more fleshed out and general level generation in the model.
- Replacement of ModelT.hasMoves() with more specific game ending functions over() when there are no moves left and win() when the last level is complete.
- A cleaner view so the user can quickly identify the row and column numbers of colours.
- Waiting for an input instead of immediately moving on to the next level.
- The ability to navigate between levels.
- A more powerful parser in Dots so that the assumption on the input doesn't have to be made.

Direction Module

Module

DirectionT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Directions = {N, E, S, W}

//N stands for north, E for east, S for south, W for west

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new DirectionT	Directions	DirectionT	

Semantics

State Variables

dir: Directions

State Invariant

None

Access Routine Semantics

new DirectionT(d):

- transition: $dir := d$

- output: $out := \text{self}$
- exception: none

Considerations

When implementing in Java, use enums (as shown in Tutorial 06 for ElementT).

Colour Module

Module

ColourT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Colours = {R, G, B, P, O, W}

//R stands for red, G for green, B for blue, P for pink, O for orange, W for white

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new ColourT	Colours	ColourT	
randCol		ColourT	

White is used to denote a removed colour in the board. This was done so that parts of a sequence can be temporarily denoted as missing without changing the size.

Semantics

State Variables

col: Colours

State Invariant

None

Access Routine Semantics

new ColourT(c):

- transition: $col := c$
- output: $out :=$ self
- exception: none

randCol():

- output: $out :=$ a random member of ColourT(except W) where each member has an equal probability of appearing.
- exception: none

Considerations

When implementing in Java, use enums (as shown in Tutorial 06 for ElementT).

Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new PointT	$\mathbb{Z}$, $\mathbb{Z}$	PointT	
row		$\mathbb{Z}$	
col		$\mathbb{Z}$	
equals	PointT	$\mathbb{B}$	

equals() violate essentiality as it can be substituted with row() and col() but, since point equality is used so often, I find it to be convenient and helps with the implementation.

Semantics

State Variables

r: $\mathbb{Z}$
c: $\mathbb{Z}$

State Invariant

None

Assumptions

The constructor PointT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new PointT(row, col):

- transition: $r, c := row, col$
- output: $out := self$
- exception: None

row():

- output: $out := r$
- exception: None

col():

- output: $out := c$
- exception: None

equals(q):

- output: $out := row = q.\text{row}() \wedge col = q.\text{col}()$
- exception: None

Line ADT Module

Template Module

LineT

Uses

DirectionT, PointT

Syntax

Exported Types

LineT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new LineT	PointT, PointT	LineT	IllegalArgumentException
start		PointT	
end		PointT	
mag		$\mathbb{N}$	
points		set of PointT	

Semantics

State Variables

start: PointT
end: PointT
dir: DirectionT

State Invariant

None

Assumptions

The constructor LineT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new LineT(p, q):

- transition: $start, end, dir := p, q, \text{calcDir}(p, q)$
- output: $out := self$
- exception: $exc := (p.\text{row}() = q.\text{row}() \equiv p.\text{col}() = q.\text{col}() \Rightarrow \text{IllegalArgumentException})$

start():

- output: $out := start$
- exception: None

end():

- output: $out := end$
- exception: None

mag():

- output: $out := start.\text{row}() = end.\text{row}() \Rightarrow \text{abs}(start.\text{col}() - end.\text{col}()) + 1 \mid start.\text{col}() = end.\text{col}() \Rightarrow \text{abs}(start.\text{row}() - end.\text{row}()) + 1$
- exception: None

points():

- output: $out := \{i : \mathbb{N} | 0 \leq i < \text{mag}() : \text{PointT}(start.\text{row}() + (dir = N \Rightarrow i \mid dir = S \Rightarrow -i \mid True \Rightarrow 0), start.\text{col}() + (dir = E \Rightarrow i \mid dir = W \Rightarrow -i \mid True \Rightarrow 0))\}$‘
- exception: None

Local Functions

abs: $\mathbb{Z} \rightarrow \mathbb{N}$
abs$(i) = i < 0 \Rightarrow -i \mid True \Rightarrow i$

calcDir: PointT $\times$ PointT $\rightarrow$ DirectionT
calcDir$(p, q) = p.\text{row}() > q.\text{row}() \Rightarrow S \mid p.\text{col}() < q.\text{col}() \Rightarrow E \mid$
    $p.\text{col}() > q.\text{col}() \Rightarrow W \mid True \Rightarrow N$

Path ADT Module

Template Module

PathT

Uses

PointT, LineT

Syntax

Exported Types

PathT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new PathT		PathT	
add	LineT		
valid		$\mathbb{B}$	
mag		$\mathbb{N}$	
points		set of PointT	

valid() is used to identify if a path is valid instead of having a state invariant of validity, an exception to avoid breaking the state invariant in add(), and an access program to check if a line will maintain validity as, when adding lines to a path, it is more convenient to check if the path is valid at the end than to check if every line maintains validity.

Semantics

State Variables

lines: seq of LineT

State Invariant

None

Assumptions

The constructor PathT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new PathT(l):

- transition: $lines := \langle\rangle$
- output: $out := self$
- exception: None

add(l):

- output: $lines := lines || l$
- exception: None

valid():

- output: $out := |lines| = 0 \Rightarrow False \mid |lines| = 1 \Rightarrow True \mid$
  $True \Rightarrow \forall(i : \mathbb{N} | i < |lines| - 1 : lines[i].\text{end}() = lines[i+1].\text{start}())$
- exception: None

mag():

- output: $out := +(l : \text{LineT} | l \in lines : l.\text{mag}()) - (|lines| - 1)$
- exception: None

points():

- output: $out := \cup(l : \text{LineT} | l \in lines : l.\text{points}())$
- exception: None

Board ADT Module

Template Module

BoardT

Uses

ColourT, PointT, PathT

Syntax

Exported Constants

None

Exported Types

BoardT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new BoardT	seq of (seq of ColourT)		IllegalArgumentException
board		seq of (seq of ColourT)	
validPath	PathT	$\mathbb{B}$	IllegalArgumentException
pathCol	PathT	ColourT	IllegalArgumentException
rmPath	PathT		IllegalArgumentException

Semantics

State Variables

b: seq of (seq of ColourT)
$nRow : \mathbb{N}$
$nCol : \mathbb{N}$

State Invariant

None

Assumptions

- The constructor BoardT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.
- The 0th row is at the bottom of the board and the 0th column is at the leftmost side of the board.

Access Routine Semantics

new BoardT($board$):

- transition: $b, nRow, nCol := board, |board|, |board[0]|$
- output: $out := self$
- exception: $exc := (|board| \leq 0 \vee |board[0]| \leq 0 \vee \exists(i : \mathbb{N} | 1 \leq i < |board| : |board[i]| \neq |board[0]|) \vee \exists(i : \mathbb{N} | 0 \leq i < |board| : \exists(j : \mathbb{N} | 0 \leq j < |board[0]| : board[i][j] = W)) \Rightarrow$ IllegalArgumentException)

board():

- output: $out := b$
- exception: None

validPath(p):

- output: $out := \forall(q : \text{PointT} | q \in p.\text{points}() : \text{validPoint}(q) \wedge \text{get}(q) = \text{get}(p.\text{points}()[0]))$
- exception: $exc := (\neg p.\text{valid}() \Rightarrow$ IllegalArgumentException)

pathCol(p):

- output: $out := \text{get}(p.\text{points}()[0])$
- exception: $exc := (\neg \text{validPath}(p) \Rightarrow$ IllegalArgumentException)

rmPath(p):

- transition: Change the state of b as follows:
    1. For every point in path p, set the colour corresponding to the point in b to white. clearOut

2. While there is a non-white colour that is located directly above a white colour, swap them. shiftDown
3. Replace every white colour with a random non-white colour. refill

- exception: $exc := (\neg \text{validPath}(p) \Rightarrow \text{IllegalArgumentException})$

Local Functions

set: PointT $\times$ ColourT
set$(p, c) \equiv b := b[p.\text{row}()][p.\text{col}()] = c$

get: PointT $\rightarrow$ ColourT
get$(p) = out := b[p.\text{row}()][p.\text{col}()]$

validRow: $\mathbb{Z} \rightarrow \mathbb{B}$
validRow$(i) \equiv 0 \leq i < nRow$

validCol: $\mathbb{Z} \rightarrow \mathbb{B}$
validCol$(j) \equiv 0 \leq j < nCol$

validPoint: PointT $\rightarrow \mathbb{B}$
validPoint$(p) \equiv$ validRow$(p.\text{row}()) \wedge$ validCol$(p.\text{col}())$

clearOut: PathT
clearOut$(p) \equiv$ change the state of b such that:
$\forall(q : \text{PoinT} | q \in p.\text{points} : \text{get}(q) = .W)$

shiftDownAble: $\mathbb{B}$
shiftDownAble() $\equiv \exists(i : \mathbb{N} | 0 \leq i < nRow - 1 : \exists(j : \mathbb{N} | 0 \leq j < nCol :$
$b[i][j] = W \wedge b[i+1][j] \neq W))$

shiftDown:
shiftDown() $\equiv$ change the state of b such that:
while shiftDownAble() apply shiftDownRow(i) for all $1 \leq i < nRow$ then Refill()

shiftDownRow: $\mathbb{N}$
shiftDownRow$(i) \equiv$ change the state of b such that:
$\forall(j : \mathbb{N} | 0 \leq j < nCol : \neg(b[i][j] = W \wedge b[i+1][j] \neq W)$

Refill:
Refill() $\equiv$ change the state of b such that:
$\forall(i : \mathbb{N} | 0 \leq i < nRow : \forall(j : \mathbb{N} | 0 \leq j < nCol : b[i][j] \neq W))$

Objective ADT Module

Template Module

ObjectiveT

Uses

ColourT

Syntax

Exported Constants

None

Exported Types

ObjectiveT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new ObjectiveT	ColourT, $\mathbb{N}$	ObjectiveT	IllegalArgumentException
col		ColourT	
goal		$\mathbb{N}$	
complete		$\mathbb{B}$	
attempt	ColourT, $\mathbb{N}$	$\mathbb{B}$	

attempt() takes in the magnitude of a path rather than the path itself because the colour is determined outside of ObjectiveT, so attempt() should not be given one piece of data and told to get the other when both can be given. This may cause some confusion. If given a PathT, the purpose of attempt() becomes clear in relation to the purpose of the overall design. However, as it stands, the abstraction takes away from the understanding.

Semantics

State Variables

col: ColourT
goal: $\mathbb{N}$

complete: $\mathbb{B}$

State Invariant

None

Assumptions

The constructor ObjectiveT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new ObjectiveT(c, g):

- transition: $\mathit{col}, \mathit{goal}, \mathit{complete} := c, g, False$
- output: $\mathit{out} := \mathit{self}$
- exception: $\mathit{exc} := (g < 2 \Rightarrow \text{IllegalArgumentException})$

col():

- output: $\mathit{out} := \mathit{col}$
- exception: None

goal():

- output: $\mathit{out} := \mathit{goal}$
- exception: None

complete():

- output: $\mathit{out} := \mathit{complete}$
- exception: None

attempt(c, x):

- Transition: $\mathit{complete} := \mathit{complete} \Rightarrow True \mid True \Rightarrow c = \mathit{col} \land x >= \mathit{goal}$
- exception: None

Level ADT Module

Template Module

LevelT

Uses

ColourT, ObjectiveT

Syntax

Exported Constants

None

Exported Types

LevelT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new LevelT	Set of ObjectiveT, $\mathbb{N}$	LevelT	IllegalArgumentException
objectives		Set of ObjectiveT	
moves		$\mathbb{N}$	
complete	$\mathbb{Z}$	$\mathbb{B}$	
attempt	ColourT, $\mathbb{N}$	$\mathbb{B}$	

Semantics

State Variables

$objs$: Set of ObjectiveT
$moves$: $\mathbb{N}$

State Invariant

None

Assumptions

The constructor LevelT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new LevelT(o, m):

- transition: $objs, moves := o, m$
- output: $out := self$
- exception: $exc := (m < 5 \Rightarrow \text{IllegalArgumentException})$

objectives():

- output: $out := objs$
- exception: None

moves():

- output: $out := moves$
- exception: None

complete(m):

- output: $out := m \leq moves \land \forall(o : \text{ObjectiveT} | o \in objs : o.\text{complete}())$
- exception: None

attempt(c, x):

- Transition: $objs := \{o : \text{ObjectiveT} | o \in objs : o.\text{attempt}(c, x))\}$
- exception: None

Model ADT Module

Template Module

ModelT

Uses

ColourT, PathT, BoardT, ObjectiveT, LevelT

Syntax

Exported Constants

None

Exported Types

ModelT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new ModelT	$\mathbb{N}, \mathbb{N}$	ModelT	IllegalArgumentException
new ModelT	BoardT, Seq of LevelT	ModelT	
board		BoardT	
level		LevelT	
lv		$\mathbb{N}$	
moves		$\mathbb{N}$	
hasMoves		$\mathbb{B}$	
validMove	PathT	$\mathbb{B}$	IllegalArgumentException
makeMove	PathT		IllegalArgumentException
completeLv			

Semantics

State Variables

b: BoardT
$levels$: Seq of LevelT

lv : $\mathbb{N}$
$moves$: $\mathbb{N}$

State Invariant

None

Assumptions

The constructor ModelT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new ModelT(d, l):

- transition: $b, levels, lv, moves :=$ makeBoard(d), makeLevels$(l), 0, 0$
- output: $out := self$
- exception: $exc := (d \leq 0 \vee l \leq 0 \Rightarrow$ IllegalArgumentException$)$

new ModelT$(board, l)$:

- transition: $b, levels, lv, moves := board, l, 0, 0$
- output: $out := self$
- exception: None

board():

- output: $out := b$
- exception: None

level():

- output: $out := levels[lv]$
- exception: None

lv():

- output: $out := lv$

- exception: None

moves():

- output: $out := levels[lv].\text{moves}() - moves$
- exception: None

hasMoves():

- output: $out := \text{moves}() > 0$
- exception: None

validMove(p):

- output: $out := b.\text{validPath}(p)$
- exception: $exc := (\neg p.\text{valid}() \Rightarrow \text{IllegalArgumentException})$

makeMove(p):

- Transition: $b, levels[lv], moves := b.\text{rmPath}(p), levels[lv].\text{attempt}(b.\text{pathCol}(p), p.\text{mag}()), moves + 1$
- exception: $exc := (\neg \text{validMove}(p) \Rightarrow \text{IllegalArgumentException})$

completeLv():

- Transition: $lv, moves := levels[lv].\text{complete}(moves) \wedge lv < |levels| - 1 \Rightarrow lv + 1, 0 \mid True \Rightarrow lv, moves$
- exception: None

Local Functions

randNum: $\mathbb{N} \rightarrow \mathbb{N}$
randNum(i) $\equiv$ a random number between 0 and i where each member has an equal probability of appearing.

makeRow: $\mathbb{N} \rightarrow$ Seq of ColourT
makeRow(d) $= \langle j : \mathbb{N} : 0 \leq j < d : \text{randCol}() \rangle$

makeBoard: $\mathbb{N} \rightarrow$ BoardT
makeBoard$(d) =$ Board$(\langle i : \mathbb{N} : 0 \leq i < r : \text{makeRow}(d)\rangle)$

makeObj: $\mathbb{N} \rightarrow$ ObjectiveT
makeObj$(i) =$ Objective$(\text{randCol}, i)$

makeLv: $\mathbb{N} \rightarrow$ LevelT
makeLv$(i) =$ LevelT$(\{\text{makeObj}(i)\}, 10)$

makeLevels: $\mathbb{N} \rightarrow$ Seq of LevelT
makeLevels$(l) = \langle i : \mathbb{N} : 0 \leq i < l : \text{makeLv}(3 + \text{randNum(i)})\rangle$

View ADT Module

Template Module

ViewT

Uses

BoardT, ObjectiveT, LevelT

Syntax

Exported Constants

None

Exported Types

ViewT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new ViewT		ViewT	
disp	BoardT, LevelT, $\mathbb{N}$, $\mathbb{N}$		

Semantics

Environment Variables

win: 2D sequence of pixels displayed on a screen

State Variables

None

State Invariant

None

Assumptions

The constructor ViewT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new ViewT():

- transition: None
- output: $out := self$
- exception: None

disp($b, level, lv, moves$):

- Transition: win := Modify window so that "Level: " followed by lv is printed at the top of the screen. Below, the ObjectiveTs in $level$.objecives() are written from left to right, 1 by 1, as the goal followed by the colour if they have not been completed. If all of the objectives are complete print "You win" and nothing else. Below, the 2D sequence of ColourT of b should by displayed starting at the top row. The names of the colours in ColourT should be printed in place of the colour they represent. At the bottom, "Number of moves:" followed by $moves$ should be printed.
- exception: None

Controller ADT Module

Template Module

ControllerT

Uses

PathT, ModelT, ViewT

Syntax

Exported Constants

None

Exported Types

ControllerT = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new ControllerT	ModelT, ViewT	ControllerT	
winnable		$\mathbb{B}$	
eval	PathT	$\mathbb{Z}$	
play	PathT		
completeLv			
updateView			

Semantics

Environment Variables

win: 2D sequence of pixels displayed on a screen

State Variables

m : ModelT
v : ViewT

State Invariant

None

Assumptions

The constructor ControllerT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

new ControllerT($model, view$):

- transition: $m, v := model, view$
- output: $out := self$
- exception: None

winnable():

- output: $out := m.\text{hasMoves}()$
- exception: None

eval(p):

- output: $out := \neg p.\text{valid}() \Rightarrow -1 \mid \neg m.\text{validMove}(p) \Rightarrow -2 \mid True \Rightarrow 0$
- exception: None

play(p):

- transition: $m := m.\text{makeMove}(p)$
- exception: None

completeLv():

- transition: $m := m.\text{completeLv}()$
- exception: None

updateView():

- transition: $win := v.\text{disp}(m.\text{board}(), m.\text{level}(), m.\text{lv}, m.\text{moves}())$
- exception: None

Dots Module

Module

DotsT

Uses

PathT, ModelT, ViewT, ControllerT

Syntax

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
init			
parse	String	PathT	
main			

Semantics

Environment Variables

win: 2D sequence of pixels displayed on a screen
$keyboard$: mapping of buttons to characters

State Variables

c : ControllerT

State Invariant

None

Assumptions

init() is called before any other access program. All input comes in the form "a,b c,d ..." where each pair is a point in row,col form and every pair of points construct a valid line.

Access Routine Semantics

init():

- transition: $c :=$ ControllerT(ModelT$(6, 5)$, ViewT)
- exception: None

parse(s):

- output: $out := p$: PathT such that: points are made from the pairs in s, lines are made from the sequential pairs of points, and lines are added sequentially to p.
- exception: None

main():

- transition: While c.winnable(), play the game. Take an input s from *keyboard* and get a path p from parse(s). Do c.eval(p) and the following: if -1, print to *win* "That path is invalid". if -2, print to *win* "That path is invalid on the board". Else do c.play(p), c.completeLv(), updateView().
- exception: None

Critique of Design

I would consider this a good design. I used an operational specification only when I needed to. The use of formal language makes the specifications clear and unambiguous. DirectionT could have been absorbed in to LineT, but doing so would make calcDir() and points() messy and difficult to read. PointT could have had a ColourT state variable that would have changed BoardT and ObjectiveT. BoardT would need a function to set the colours in a given PathT after it has been verified. ObjectiveT's attempt() would accept a PathT and obtain the needed data from it. I kept them separate to keep the generality of PointT.

The design in consistent in most areas. BoardT and ModelT both have an access program called board() that do different things. In BoardT, it returns a Seq of (Seq of ColourT) whereas, in ModelT, it returns a BoardT. This inconsistent naming may lead to the assumption that ModelT's board() will also return a Seq of (Seq of ColourT). A solution would to rename BoardT's board() to grid(). This will eliminate inconsistency and be more descriptive as it makes more sense to say a board has a grid than to say a board has a board. In contrast, BoardT and ModelT both have functions that deal with a PathT that are named differently. BoardT calls to it as a path while ModelT calls it a move. This is inconsistent as these functions in ModelT mostly use functions in BoardT, so they should be named with the same idea. This naming was done to abstract away from the bigger idea in BoardT. A path is a general path in the board while a move is a path specified by a user. LevelT and ModelT both have moves(). In LevelT it is the maximum number of moves the level can be completed in. In ModelT it is the number of moves remaining. A more appropriate name in LevelT would be maxMoves.

There are some specifications that are not essential. PointT's equals() violate essentiality as it can be substituted with row() and col() but, since point equality is used so often, I find it to be convenient and helps with the implementation. ModelT has 2 constructors, 1 for setting a given board and levels and other for specifying the size of the board and number of levels to be generated. The second constructor can be done by generating the data by itself and setting them, but I believe it is suitable for ModelT to generate its own data rather than have another module to do it and pass it. The non-generating constructor also allows for the testing of other functions with consistent data. ModelT's hasMoves() can be done with by a comparison with moves() but was done for the convenience of a simpler looking while loop in ControllerT. However, this convenience is very minor and not worth losing essentiality so it should be removed.

PointT, LineT, and PathT are mostly general. Changing magnitude so that it returns the length rather than the number of points would make it more general as length is often more important than the number of points, especially in a real number setting where there are an infinite number of points. BoardT is not very general as rmPath() does more than removing the colours in a path. It also affects everything above it. There are many more applications that want to specify a change in only a certain area and no where else. ObjectiveT can become more general by changing the goal to a counter. That way objectives that accumulate the progression of many paths can be made. ModelT, ViewT, ControllerT, and Dots were all made for this specific design.

ModelT's makeMove() and completeLv() are not minimal as they modify more than 1 state variable. I believe this is acceptable as the changes are related to one another. In makeMove(), the board must be changed to remove the path specified. With colours removed the objectives must be updated. Since the user made a move, the moves counter must increment. completeLv() moves to the next level, so the level counter must increment and move counter must be reset to give a fresh start.

All functions in each module are related to the idea of the module.

Information hiding is preserved. The implementations of the modules that make up the model were made after the interfaces, so the interfaces know nothing about the implementation. ViewT, ControllerT, and DotsT were implemented before their interfaces were finalized, however, none of them rely on the implementations of the modules they use.

Questions

1. In all 3 patterns there is a client that interacts with some entity. The client only knows about the service requested, they know nothing about any further interactions.

   The proxy pattern is used when there should not be direct access to the provider. The request do not go directly to the provider. It is sent to a proxy who acts as additional control. The proxy sends the modified request to the provider who then provides the service to the client. It is used if the source requires special permissions or if the requests need to be processed before the provider can receive it.

   The strategy pattern is used when the implementation of a service differs depending on the client. There are multiple implementations of the request. The implementation chosen is dependent on who the client is. It is used when neither inheritance nor an interface sufficiently solves the problem. For a family of modules, some services may not be applicable to all, so they cannot all inherit from a single module. For those who are applicable, it would be wasteful for all of them to implement an interface the same way. The strategy pattern aims to combine the ideas by having a module that the family inherits and using an interface for its services. In this way, all the classes can be given the appropriate implementations while they use the same interface.

   The adapter pattern is used when a module to be used has an unsuitable interface. The request goes through an adapter that modifies it so that it better fits the interface of the provider. It is used to make it look like a module has the appropriate interface. Given a library with all the desired functionality but an awkward interface, the pattern creates an intermediate module with a better interface. The services of this module map to services of the library so that it appears as though the library has a new interface. When a module uses an existing one it sometimes renames an existing function to better fit something like an MIS.

2.