Shoppers Drug Mart Calculator Spec

Aditya Sharma

May 31, 2020

Module Interface Specification (MIS) document contains modules, types and methods for implementing an online web application.

The purpose of this software product is to calculate the total price of customer's cart items as they shop along in Shoppers Drug Mart (SDM).

1 Overview

2 Domain

2.1 Relevant Facts and Assumptions

2.2 Purpose

2.3 Stakeholders

2.3.1 Intended Audience

The primary audience will be all SDM Customers.

2.3.2 Impacted Groups/ Individuals

The corporation known as Shoppers Drug Mart is a group that could be impacted by this application. The impact on the corporation is currently unknown; however, this conclusion was drawn because of the potential impact that this application could have on a SDM Customer. If the SDM Customer is impacted, it would only be logical for the impact to stretch onto the company serving the customer.

2.4 Product Scope

The purpose of this application would be to track the total price of the all items that a shopper would want to purchase from SDM. As a deliverable to the customer, the application will be providing the customer with a editable cart with a total price.

3 Functional Requirements

- The application needs to be geared towards predominantly mobile use
- The UI should be "card" based
- All items should be immediately visible
- The user should be able to add and delete items
- The items should have names, descriptions, price, applied discounts
- All the fields on the items should be editable
- The user should be able to apply particular discounts to particular items
- The user should be able to add as many items to their cart
- THe total price of all the items should be displayed

Types Module

3.1 Item

Property	Data Type
itemName	string
itemDescription	string
itemInitialPrice	number
itemQuantity	number
itemDiscountType	string
itemDiscount	number
itemFinalPrice	number
deleteItem	boolean

3.2 CardState

Property	Data Type
itemArray	Array<Item>

ItemADT Module

3.3 Module

ItemADT

3.4 Uses

3.4.1 Imported Constants

None

3.4.2 Imported Data Types

Item

3.4.3 Imported Access Programs

None

3.5 Syntax

3.5.1 Exported Constants

None

3.5.2 Exported Types

None

3.5.3 Exported Functions

None

3.5.4 Exported Classes

ItemADT

3.5.5 Access Routine Interface

Routine name	In	Out	Exceptions
constructor	Item (props)	ItemADT	
editItem	Item		
deleteItem	Item		

3.6 Semantics

3.6.1 State Variables

item: Item

3.6.2 State Invariants

- All properties of *item* must have a value except for *item.itemDescription*
- All properties of *item* must be editable/ changeable (must respond to user input)

3.6.3 Assumptions

None

3.6.4 Access Routine Semantics

constructor(props):

- transition: bind *this* to each routine
- output: *out* := self
- exception: none

editItem():

- transition: none
- output: *out* := causes modal to pop-up
- exception: none

deleteItem():

- transition: delete item from itemArray

- output: *out* := none
- exception: none

Card Module

3.7 Module

Card

3.8 Uses

3.8.1 Imported Constants

None

3.8.2 Imported Data Types

Item ItemArray

3.8.3 Imported Access Programs

ItemADT

3.9 Syntax

3.9.1 Exported Constants

None

3.9.2 Exported Types

None

3.9.3 Exported Functions

None

3.9.4 Exported Classes

None

3.9.5 Access Routine Interface

Routine name	In	Out	Exceptions
constructor	any	Card	

3.10 Semantics

3.10.1 State Variables

3.10.2 State Invariants

-

3.10.3 Assumptions

None

3.10.4 Access Routine Semantics

None

3.10.5

Land Use Type Module

Module

LanduseT

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Landtypes = {R, T, A, C}

//R stands for Recreational, T for Transport, A for Agricultural, C for Commercial

Exported Access Programs

Routine name	In	Out	Exceptions
new LanduseT	Landtypes	LanduseT	

Semantics

State Variables

landuse: Landtypes

State Invariant

None

Access Routine Semantics

new LandUseT(t):

- transition: *landuse* := t

- output: *out* := self
- exception: none

Considerations

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

Point ADT Module

Template Module inherits Equality(PointT)

PointT

Uses

N/A

Syntax

Exported Types

[What should be written here? —SS] PointT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PointT	$\mathbb{Z}$, $\mathbb{Z}$	PointT	
row		$\mathbb{Z}$	
col		$\mathbb{Z}$	
translate	$\mathbb{Z}$, $\mathbb{Z}$	PointT	

Semantics

State Variables

r: [What is the type of the state variables? —SS] $\mathbb{Z}$
c: [What is the type of the state variables? —SS] $\mathbb{Z}$

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

PointT(row, col):

- transition: [What should the state transition be for the constructor? —SS] r, c := row, col
- output: $out := self$
- exception: None

row():

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

col():

- [What should go here? —SS] output: $out := c$
- exception: None

translate(Δr, Δc):

- [What should go here? —SS] output: $out :=$ PointT(r + Δr, c + Δc)
- exception: [What should go here? —SS] None

Generic Seq2D Module

Generic Template Module

Seq2D(T)

Uses

PointT

Syntax

Exported Types

Seq2D(T) = ?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Seq2D	seq of (seq of T), $\mathbb{R}$	Seq2D	IllegalArgumentException
set	PointT, T		IndexOutOfBoundsException
get	PointT	T	IndexOutOfBoundsException
getNumRow		$\mathbb{N}$	
getNumCol		$\mathbb{N}$	
getScale		$\mathbb{R}$	
count	T	$\mathbb{N}$	
countRow	T, $\mathbb{N}$	$\mathbb{N}$	
area	T	$\mathbb{R}$	

Semantics

State Variables

s: seq of (seq of T)
scale: $\mathbb{R}$
nRow: $\mathbb{N}$
nCol: $\mathbb{N}$

State Invariant

None

Assumptions

- The Seq2D(T) constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Assume that the input to the constructor is a sequence of rows, where each row is a sequence of elements of type T. The number of columns (number of elements) in each row is assumed to be equal. That is each row of the grid has the same number of entries. $s[i][j]$ means the ith row and the jth column. The 0th row is at the top of the grid and the 0th column is at the leftmost side of the grid.

Access Routine Semantics

Seq2D(S, scl):

- transition: [Fill in the transition. —SS] s, scale, nRow, nCol = $S, scl, |S|, |S[0]|$
- output: $out := self$
- exception: [Fill in the exception. One should be generated if the scale is less than zero, or the input sequence is empty, or the number of columns is zero in the first row, or the number of columns in any row is different from the number of columns in the first row. —SS] $(scale < 0) \vee (s = \emptyset) \vee (s[0] = \emptyset) \vee (\exists i : \mathbb{N} | i \in [1..|s| - 1] : (|s[i]| \neq |s[0]|)$

set(p, v):

- transition: [? —SS] s[$p.r$][$p.c$] := v
- exception: [Generate an exception if the point lies outside of the map. —SS] (¬validPoint(p) ⇒ IndexOutOfBoundsException)

get(p):

- output: [? —SS] out := s[$p.r$][$p.c$]
- exception: [Generate an exception if the point lies outside of the map. —SS] (¬validPoint(p) ⇒ IndexOutOfBoundsException)

getNumRow():

- output: out := nRow
- exception: None

getNumCol():

- output: out := nCol
- exception: None

getScale():

- output: out := scale
- exception: None

count(t: T):

- output: [Count the number of times the value t occurs in the 2D sequence. —SS] $out := (+i, j : \mathbb{N} | \text{validRow}(i) \wedge \text{validCol}(j) \wedge s[i][j] = t : 1)$
- exception: None

countRow(t: T, $i : \mathbb{N}$):

- output: [Count the number of times the value t occurs in row i. —SS] $out := +(elem : T | \text{validRow}(i) \wedge elem \in s[i] \wedge elem = t : 1)$
- exception: [Generate an exception if the index is not a valid row. —SS] $\neg$validRow(i) $\Rightarrow$ $IndexOutOfBoundsException$

area(t: T):

- output: [Return the total area in the grid taken up by cell value t. The length of each side of each cell in the grid is scale. —SS] $out := \text{count(T)} \cdot \text{scale}^2$
- exception: None

Local Functions

validRow: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid row number. —SS]
validRow(i) $\equiv (i \geq 0 \wedge i <$ nRow $\Rightarrow True | i < 0 \vee i \geq$ nRow $\Rightarrow False)$

validCol: $\mathbb{N} \rightarrow \mathbb{B}$
[returns true if the given natural number is a valid column number. —SS]
(validCol(j) $\equiv (j \geq 0 \wedge j <$ nCol $\Rightarrow True | j < 0 \vee j \geq$ nCol $\Rightarrow False)$

validPoint: PointT $\rightarrow \mathbb{B}$
[Returns true if the given point lies within the boundaries of the map. —SS]
validPoint(T) $\equiv ()$validRow(T.row()) $\wedge$ validCol(T.col()) $\Rightarrow True | \neg$validRow(T.row()) $\vee$
$\neg$validCol(T.col()) $\Rightarrow False)$

LanduseMap Module

Template Module

[Instantiate the generic ADT Seq2D(T) with the type LanduseT —SS]
LanduseMap is Seq2D(LanduseT)

DEM Module

Template Module

DemT is Seq2D($\mathbb{Z}$)

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
total		$\mathbb{Z}$	
max		$\mathbb{Z}$	
ascendingRows		$\mathbb{B}$	

Semantics

Access Routine Semantics

total():

- output: [Total of all the values in all of the cells. —SS]
  $out := (+i, j : \mathbb{N} | \text{validRow}(i) \wedge \text{validCol}(j) : s[i][j])$
- exception: None

max():

- output: [Find the maximum value in the 2d grid of integers —SS]
  $out := (m : \mathbb{Z} | \forall (i, j : \mathbb{N} | \text{validRow}(i) \wedge \text{validCol}(j) \wedge m \geq s[i][j]) \wedge \exists (i, j : \mathbb{N} | s[i][j] = m) : s[i][j])$
- exception: None

ascendingRows():

- output: [Returns True if the sum of all values in each row increases as the row number increases, otherwise, returns False. —SS]
  $out := (\forall i : \mathbb{N} | \text{validRow}(i) \wedge \text{validRow}(i+1) : \text{sumRow}(i) < \text{sumRow}(i+1))$
- exception: None

Local Functions

validRow: $\mathbb{N} \to \mathbb{B}$
[returns true if the given natural number is a valid row number. —SS]
validRow(i) $\equiv (i \geq 0 \wedge i < \text{nRow} \Rightarrow True | i < 0 \vee i \geq \text{nRow} \Rightarrow False)$

validCol: $\mathbb{N} \to \mathbb{B}$
[returns true if the given natural number is a valid column number. —SS]
(validCol(j) $\equiv (j \geq 0 \wedge j < \text{nCol} \Rightarrow True | j < 0 \vee j \geq \text{nCol} \Rightarrow False)$

sumRow: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
[returns the sum of a particular row. —SS]
sumRow(i) $\equiv (+j : \mathbb{N} | \text{validRow}(i) \wedge \text{validCol}(j) \wedge s[i][j] \in s[i] : s[i][j])$

Critique of Design

[Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why? One thing you could discuss is that the Java implementation, following the notes given in the assignment description, will expose the use of ArrayList for Seq2D. How might you change this? There are repeated local functions in two modules. What could you do about this? —SS]

No, I don't believe that there is anything missing regarding the interface for the modules with regards to the methods. The reason is because each module interface had a particular goal and each interface succinctly described the methods that would be used to achieve the goal - no more were needed. With regards to the Software Engineering Principles, I believe that majority of the Software Engineering Principles were followed.

- Consistency: The module interfaces were consistent because they followed the same naming conventions. Also, if I were given just a little bit of information regarding the modules, I would be able to piece the rest of them together.
- Essential: The module interfaces satisfied the criteria of being essential because they included useful features. They didn't include any over-the-top, creative/ somewhat-useless features. Each module had a particular goal and they had methods that worked togehter to achieve that goal/ purpose.
- Minimal: The module interface satisfied the criteria of being minimal because they didn't offer methods that provided uniquely independent/ seperate access routines. Each of the routines were somewhat related to each other and would be utilized together to achieve a particular goal.
- Opaque: The module interfaces were opaque and this supported information hiding/ encapsulation. This is because the modules were designed in a way such that if the implementation of the particular methods changed, then the overall action would not change. However, this is assuming that the specifications were followed exactly.
- Low Coupling and High Cohesion: The module interfaces did satisfiy Low Coupling because they weren't strongly dependent on each other. They were created to work with each other but they aren't overly dependent on each other. However, at the same time, they aren't enough modules for me low coupling to be completely valid. At the same time, the module interfaces satisfied High Cohesion because of how the methods within modules could be used within other methods for other tasks. It had a high cohesivenesss.

- General: The module interfaces were general because they didn't require super specific data types or particular implementation details. Seq2D used generic types and PointT established generic types too. They weren't super specific and that was why it satisfies the quality of being general. It allows them to be used for multiple situations.

As mentioned in the assignment description, ArrayLists will be used in the Java Implementation. From a design standpoint, I would change this because this is the opposite of information hiding. In information hiding, we should hide the implementation details behind an interface. However, by exposing the fact that ArrayLists should and will be used in the implementation, we are exposing this implementation detail to the developer. In order to enforce information hiding, the description and MIS should not have disclosed this fact and should not be enforcing it. If I were to make a change, the first change I would do is make the necessary changes to enforce information hiding. Then, I would specify that a particular data type is used to store the information (making things as abstract/ generic as possible).

I also noticed that there were repeated local functions in the two modules. This is bad practice because the exact same code is being re-written. Instead, it should be reused. Since those local functions were used in the exceptions, I would create a Module Interface where those local functions are methods. This would allow for reuse by the other modules that leveraged exceptions that rely on the local functions. This in turn improves the quality of the software system.

In addition to your critique, please address the following questions:

1. The original version of the assignment had an Equality interface defined as for A2, but this idea was dropped. In the original version Seq2D inherited the Equality interface. Although this works in Java with the LanduseMapT, it is problematic for DemT. Why is it problematic? (Hint: DEMT is instantiated with the Java type Integer.)
   Its problematic because there is a difference in types. So, while the equality interface is used to handle generic types and comparisons between such types, the DemT is using numbers. We can't perform comparisons the exact same way because integers use a different form of comparison than generic types. Assuming that Generic Types is referring to ADTs, these ADTs will have references which is why we can't perform comparisons via "==" the way integers do. As a result, there needs to be a different type of way to determine if the Generic Types are equal. But since DemT uses an integer, the way we determine equality with generic types will be different than the way we determine equality for integers. That is why it would be problematic.

2. Although Java has several interfaces as part of the standard language, such as the Comparable interface, there is no Equality interface. Instead equals is provided through inheritance from Object. Why do you think the Java language designers decided to use inheritance for equality, instead of providing an interface?

   I think that Java language designers decided to use inheritance for equality instead of an interface because with inheritance for equality, we can perform overloading or overriding. This allows us to handle different types and numbers of parameters for our equality functions. If an interface would have been used, then we would need to implement the interface exactly. And an interface specifies the particular types for the variables we need to implement, but with inheritance, we can change the data types of variables to match our needs. We wouldn't be able to design for change if we used an interface. In addition, it allows for reuse. For example: If I design an interface to allow for equality for Grid objects. I won't be able to use that same interface to determine if Map objects are equal. I can only implement it for Grid objects. However, with inheritance, I can quickly inherit key properties and simply change the data types.

3. The qualities of good module interface push the design of the interface in different directions. Why is it rarely possible to achieve a module interface that simultaneously is essential, minimal and general?

   A module interface is said to be essential when it contains methods that are essential to accomplishing its purpose; when it does not include any unnecessary access routines. A module interface is said to be general when it is able to satisfy unpredictable usecases. A module interface is said to be minimal if it does not have 2 access routines that provide independent services.

   The issue arises when we want to make a module general. By doing so, in order to make it useful in unpredictable cases, we need to include unique, creative/ abstract methods. However, by doing so, these routines will most likely be unnecessary for the key functionality of the module. Thus, by making it general, we would not make it essential. In addition, by making it general, we would have unique routines that would most likely provide independent services and this would make the module not minimal. This showcases it becomes difficult to make a module general, minimal and essential.