

Fruta Feia in VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

*Diogo Amorim Cepa - up201403367*

*Nuno Jorge Dias Carneiro Martins - up201405079*

**Contents**

[1. Informal system description and list of requirements 3](#_30j0zll)

[1.1 Informal system description 3](#_1fob9te)

[1.2 List of requirements 3](#_3znysh7)

[2. Visual UML model 4](#_2et92p0)

[2.1 Use case model 4](#_tyjcwt)

[2.2 Class model 6](#_3dy6vkm)

[3. Formal VDM++ model 7](#_1t3h5sf)

[3.1 Class FrutaFeia](#_4d34og8) 7

[3.2 Class Products](#_2s8eyo1) 10

[3.3 Class Producer](#_17dp8vu) 11

3.4 Class Delegation 12

3.5 Class Consumer 15

3.6 Class Basket 16

[4. Model validation 12](#_3rdcrjn)

[4.1 Class Test](#_26in1rg)s [12](#_lnxbz9)

[5. Model verification 16](#_35nkun2)

[5.1 Example of domain verification 16](#_1ksv4uv)

[5.2 Example of invariant verification 16](#_44sinio)

6. Java Code Generation

7[. Conclusions 17](#_2jxsxqh)

8[. References 17](#_z337ya)

# 1. Informal system description and list of requirements

## 1.1 Informal system description

The Fruta Feia system will have three possible roles: the client, the producer and the delegation. A client registers in the system and chooses the type of basket that he wants to receive, a 3 kg with 4 types of products or a 8 kg with 4 types. After he’s assigned to a delegation where he will buy his ordered basket. The delegation is the entity that sells the baskets to the clients, having a set of which clients are assigned to it. The producers are responsible of providing the products of the baskets. Each producer has a set of delegations that he produces to, and a set of products and the quantity produced of each product. The Fruta Feia system sees how much products the producers have and then sorts them to have balanced baskets based on how many clients and the location of the producer and delegation.

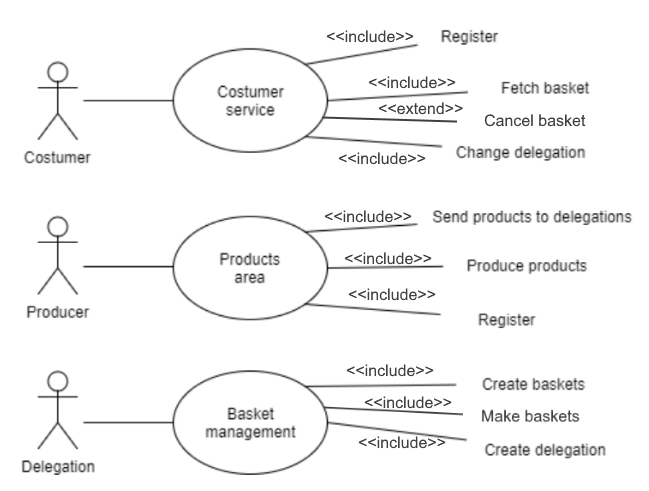
## 1.2 List of requirements

|  |  |  |
| --- | --- | --- |
| **Id** | **Priority** | **Description** |
| P0 | Mandatory | The producer can register itself |
| P1 | Mandatory | The producer can add and remove products from its list |
| P2 | Mandatory | The producer can add product stocks to its products |
| P3 | Mandatory | The producer can associate itself with a delegation |
| P4 | Mandatory | The producer can send its products to associated delegations |
| D0 | Mandatory | The delegation can register itself |
| D1 | Mandatory | The delegation can associate itself with a customer |
| D2 | Mandatory | The delegation can create and fill baskets of fruit |
| C0 | Mandatory | The customer can register himself |
| C1 | Mandatory | The customer can fetch his basket when ready |
| C2 | Low | The customer can cancel his basket if he doesn’t want it |

These requirements are directly translated onto use cases as shown next.

# 2. Visual UML model [[1]](#footnote-2)

## 2.1 Use case model [[2]](#footnote-3)



The major use case scenarios (to be used later as test scenarios) are described next.

|  |  |
| --- | --- |
| **Scenario** | **Producer Register** |
| **Description** | A producer wants to register the system |
| **Pre-conditions** | 1. There is no other producer with the same name |
| **Post-conditions** | 1. There is a producer with the choosen name |
| **Steps** | 1. Choose username  2. Create user with said username  3. Insert user in set of users |
| **Exceptions** | 1. A producer with that name exists already |
| **Scenario** | **Producer Add a product** |
| **Description** | Add a product that a producer makes |
| **Pre-conditions** | 1. The product is not already in it’s set of products |
| **Post-conditions** | 1. The product is in the set of products |
| **Steps** | 1. Choose product name  2. Insert product in map of products(map of product to stock) |
| **Exceptions** | 1. The product is already in the list of products |

|  |  |
| --- | --- |
| **Scenario** | **Producer Increase product stock** |
| **Description** | Increase the stock of a product a producer produces |
| **Pre-conditions** | 1. The product is in the list of products it produces  2. The ammount to be increased is greater than 0 |
| **Post-conditions** | 1. The stock of the product is greather than the value it was increased by |
| **Steps** | 1. Select product to increase stock  2. Increase stock in map of products |
| **Exceptions** | 1. The product is not in the list of products the producer makes  2. The value to increase is negative |

|  |  |
| --- | --- |
| **Scenario** | **Producer Send products to delegation** |
| **Description** | The producer sends it’s products to the delegations |
| **Pre-conditions** | 1. The producer has any products to send  2. The delegation and the producer have products in common |
| **Post-conditions** | (none) |
| **Steps** | 1. Select the producer that wants to send its products  2. For each product the producer has, distribute the product equally to all delegations that seek it |
| **Exceptions** | 1. The producer doesn’t have any products |

|  |  |
| --- | --- |
| **Scenario** | **Delegation Add customer** |
| **Description** | The delegation adds a new customer |
| **Pre-conditions** | 1. The customer is not already in the list of the delegation’s customers  2. The customer exists |
| **Post-conditions** | 1. The customer is in the list of customers |
| **Steps** | 1. Select the customer to insert |
| **Exceptions** | 1. The customer does not exist  2. The customer is already in the list of customers |

|  |  |
| --- | --- |
| **Scenario** | **Delegation Prepare baskets** |
| **Description** | The delegation makes the necessary baskets and fills them with products |
| **Pre-conditions** | 1. The delegation has at least 1 product |
| **Post-conditions** | 1. The number of baskets made is equal to the number of customers |
| **Steps** | 1. For each customer, create an empty basket  2. Fill basket with the N products with biggest stock(depending if it’s a big or small basket), being the proportion equal to the stock of the product |
| **Exceptions** | 1. The producer doesn’t have any products |

|  |  |
| --- | --- |
| **Scenario** | **Consumer Register** |
| **Description** | The consumer registers itself in the system |
| **Pre-conditions** | 1. There is no other consumer with this name |
| **Post-conditions** | 2. There is a producer with this name |
| **Steps** | 1. Choose a name  2. Insert user in list of consumers |
| **Exceptions** | 1. The consumer already exists with that name |

## 

|  |  |
| --- | --- |
| **Scenario** | **Consumer Join Delegation** |
| **Description** | The consumer becomes client of a delegation |
| **Pre-conditions** | 1. The consumer exists  2. The consumer is not a client already |
| **Post-conditions** | 1. The consumer is a client |
| **Steps** | 1. Select the delegation  2. Insert the consumer in the list of clients of the delegation |
| **Exceptions** | 1. The consumer doesn’t exist  2. The consumer is already a client |

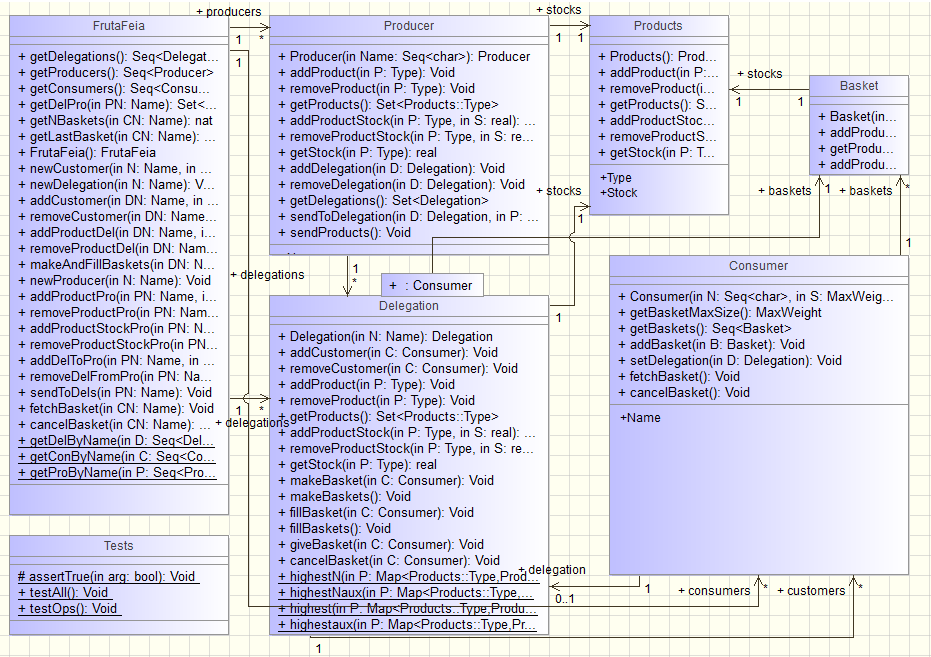
## 

|  |  |
| --- | --- |
| **Scenario** | **Consumer fetch basket** |
| **Description** | The consumer fetches his basket |
| **Pre-conditions** | 1. The consumer is a client of the delegation  2. The delegation has the client’s basket ready |
| **Post-conditions** | 1. The client received his basket |
| **Steps** | 1. Remove the basket from the delegation  2. Add the basket to the list of baskets of the client |
| **Exceptions** | 1. The consumer is not a client of the delegation  2. The basket is not ready to be delievered |

## 

|  |  |
| --- | --- |
| **Scenario** | **Consumer cancel basket** |
| **Description** | The consumer cancels his basket |
| **Pre-conditions** | 1. The consumer is a client of the delegation  2. The delegation has the client’s basket ready |
| **Post-conditions** | 1. The client’s basket was removed and is not in the client’s hands |
| **Steps** | 1. Remove the basket from the delegation |
| **Exceptions** | 1. The consumer is not a client of the delegation  2. The basket is not ready to be delievered |

## 2.2 Class model



|  |  |
| --- | --- |
| **Class** | **Description** |
| FrutaFeia | Represents the entire FrutaFeia network |
| Products | Saves a list of products and its stocks |
| Producer | A producer that makes products to send to delegations |
| Delegation | An entity that makes baskets of fruits for its customers |
| Consumer | A client that will get baskets from the associated delegation |
| Basket | An object that contains a limited amount of fruits with a limited weight |
| Tests | A series of tests to the operations of FrutaFeia |

# 3. Formal VDM++ model

## 3.1 Class FrutaFeia

**class** FrutaFeia

**instance variables**

**public** delegations : **seq** **of** Delegation;

**public** producers : **seq** **of** Producer;

**public** consumers : **seq** **of** Consumer;

**inv** forall d **in** **seq** delegations & d.customers subset elems consumers;

**inv** forall p **in** **seq** producers & p.delegations subset elems delegations;

**operations**

--List delegations

**public** getDelegations : () ==> **seq** **of** Delegation

getDelegations() == (return delegations);

--List producers

**public** getProducers : () ==> **seq** **of** Producer

getProducers() == (return producers);

--List delegations

**public** getConsumers : () ==> **seq** **of** Consumer

getConsumers() == (return consumers);

--List of delegations of a producer

**public** getDelPro: Producer`Name ==> **set** **of** Delegation

getDelPro(PN) == (

return getProByName(producers,PN).getDelegations()

);

--N baskets a customer has received

**public** getNBaskets: Consumer`Name ==> **nat**

getNBaskets(CN) == (

return len getConByName(consumers,CN).getBaskets()

);

--Get last basket

**public** getLastBasket: Consumer`Name ==> Basket

getLastBasket(CN) == (

return hd getConByName(consumers,CN).getBaskets()

);

--Constructor

**public** FrutaFeia : () ==> FrutaFeia

FrutaFeia() == (delegations := [];producers := [];consumers := []; return self;);

--new Client

**public** newCustomer : Consumer`Name \* Basket`MaxWeight ==> ()

newCustomer(N,M) == (

**dcl** customer : Consumer := new Consumer(N,M);

consumers := [customer] ^ consumers;

)

**pre** forall c **in** **seq** consumers & c.name <> N

**post** exists c **in** **seq** consumers & c.name = N;

--new Delegation

**public** newDelegation: Delegation`Name ==> ()

newDelegation(N) == (

**dcl** delegation : Delegation := new Delegation(N);

delegations := [delegation] ^delegations;

)

**pre** forall d **in** **seq** delegations & d.name <> N

**post** exists d **in** **seq** delegations & d.name = N;

--add Customer to delegation

**public** addCustomer: Delegation`Name \* Consumer`Name ==> ()

addCustomer(DN,CN) == (

getDelByName(delegations,+DN).addCustomer(getConByName(consumers,CN));

);

--remove Customer from delegation

**public** removeCustomer: Delegation`Name \* Consumer`Name ==> ()

removeCustomer(DN,CN) == (

getDelByName(delegations,DN).removeCustomer(getConByName(consumers,CN));

);

--add Product type to delegation

**public** addProductDel: Delegation`Name \* Products`Type ==> ()

addProductDel(DN,PT) == (

getDelByName(delegations,DN).addProduct(PT);

);

--remove Product type from delegation

**public** removeProductDel: Delegation`Name \* Products`Type ==> ()

removeProductDel(DN,PT) == (

getDelByName(delegations,DN).removeProduct(PT);

);

--make baskets all fill them

**public** makeAndFillBaskets : Delegation`Name ==> ()

makeAndFillBaskets(DN) == (

getDelByName(delegations,DN).makeBaskets();

getDelByName(delegations,DN).fillBaskets();

);

--New producer

**public** newProducer : Producer`Name ==> ()

newProducer(N) == (

**dcl** producer : Producer := new Producer(N);

producers := [producer] ^ producers;

);

--add Product type to producer

**public** addProductPro: Delegation`Name \* Products`Type ==> ()

addProductPro(PN,PT) == (

getProByName(producers,PN).addProduct(PT);

);

--remove Product type from producer

**public** removeProductPro: Delegation`Name \* Products`Type ==> ()

removeProductPro(PN,PT) == (

getProByName(producers,PN).removeProduct(PT);

);

--add Product type to producer

**public** addProductStockPro: Delegation`Name \* Products`Type \* Products`Stock ==> ()

addProductStockPro(PN,PT,Q) == (

getProByName(producers,PN).addProductStock(PT,Q);

);

--remove Product type from producer

**public** removeProductStockPro: Delegation`Name \* Products`Type \* Products`Stock ==> ()

removeProductStockPro(PN,PT,Q) == (

getProByName(producers,PN).removeProductStock(PT,Q);

);

--add delegation to producer

**public** addDelToPro : Producer`Name \* Delegation`Name ==> ()

addDelToPro(PN,DN) == (

getProByName(producers,PN).addDelegation(getDelByName(delegations,DN));

);

--remove delegation from producer

**public** removeDelFromPro : Producer`Name \* Delegation`Name ==> ()

removeDelFromPro(PN,DN) == (

getProByName(producers,PN).removeDelegation(getDelByName(delegations,DN));

);

--send prods from producer to delegations

**public** sendToDels : Producer`Name ==> ()

sendToDels(PN) == (getProByName(producers,PN).sendProducts());

--fetch basket

**public** fetchBasket : Consumer`Name ==> ()

fetchBasket(CN) == (getConByName(consumers,CN).fetchBasket());

--cancel basket

**public** cancelBasket : Consumer`Name ==> ()

cancelBasket(CN) == (getConByName(consumers,CN).cancelBasket());

**functions**

**public** getDelByName: **seq** **of** Delegation \* Delegation`Name -> [Delegation]

getDelByName(D,N)==(if D = [] **then** nil **else** if (hd D).name = N **then** (hd D) **else** getDelByName(tl D, N))

**pre** exists d **in** **seq** D & d.name = N;

**public** getConByName: **seq** **of** Consumer \* Consumer`Name -> [Consumer]

getConByName(C,N)==(if C = [] **then** nil **else** if (hd C).name = N **then** (hd C) **else** getConByName(tl C, N))

**pre** exists c **in** **seq** C & c.name = N;

**public** getProByName: **seq** **of** Producer \* Producer`Name -> [Producer]

getProByName(P,N)==(if P = [] **then** nil **else** if (hd P).name = N **then** (hd P) **else** getProByName(tl P, N))

**pre** exists p **in** **seq** P & p.name = N;

**end** FrutaFeia

## 3.2 Class Products

**class** Products

**types**

**public** **static** Type = **seq** **of** **char** **inv** t == len t > 0;

**public** **static** Stock = **real** **inv** s == s >= 0;

**instance variables**

**public** products : **map** Type **to** Stock;

**operations**

**public** Products: () ==> Products

Products() == (products := { |-> };return self);

--Add a product

**public** addProduct: Type ==> ()

addProduct(P) == (products := products munion {P |-> 0};)

**pre** P not **in** **set**(dom products)

**post** P in **set**(dom products) and products(P) = 0;

--Remove a product

**public** removeProduct: Type ==> ()

removeProduct(P) == (products := {P} <-: products)

**pre** P in **set**(dom products)

**post** P not **in** **set**(dom products);

--Get product list

**public** getProducts: () ==> **set** **of** Type

getProducts() == (return dom products);

--Add stock to a product

**public** addProductStock: Type \* **real** ==> ()

addProductStock(P,S) == (products := products ++ {P |-> products(P) + S})

**pre** P in **set**(dom products) and S >= 0

**post** products(P) >= S;

--Remove stock from a product

**public** removeProductStock: Type \* **real** ==> ()

removeProductStock(P,S) == (products := products ++ {P |-> products(P) - S})

**pre** P in **set**(dom products) and S >= 0 and S <= products(P);

--Get stock of a product

**public** getStock: Type ==> **real**

getStock(P) == (return products(P))

**pre** P in **set**(dom products);

**end** Products

## 3.3 Class Producer

**class** Producer

**types**

**public** Name = **seq** **of** **char** **inv** n == len n > 0;

**instance variables**

**public** name : Name;

**public** stocks : Products;

**public** delegations : **set** **of** Delegation;

**operations**

**public** Producer: **seq** **of** **char** ==> Producer

Producer(Name) == (name := Name;stocks := new Products();delegations := {} ; return self);

--Add a product

**public** addProduct: Products`Type ==> ()

addProduct(P) == (stocks.addProduct(P););

--Remove a product

**public** removeProduct: Products`Type ==> ()

removeProduct(P) == (stocks.removeProduct(P));

--Get product list

**public** getProducts: () ==> **set** **of** Products`Type

getProducts() == (return stocks.getProducts());

--Add stock to a product

**public** addProductStock: Products`Type \* **real** ==> ()

addProductStock(P,S) == (stocks.addProductStock(P,S));

--Remove stock from a product

**public** removeProductStock: Products`Type \* **real** ==> ()

removeProductStock(P,S) == (stocks.removeProductStock(P,S));

--Get stock of a product

**public** getStock: Products`Type ==> **real**

getStock(P) == (return stocks.getStock(P));

--Add a new delegation

**public** addDelegation: Delegation ==> ()

addDelegation(D) == (delegations := delegations union {D})

**pre** D not **in set** delegations

**post** D in **set** delegations;

--Remove a delegation

**public** removeDelegation: Delegation ==> ()

removeDelegation(D) == (delegations := delegations \ {D};)

**pre** D in **set** delegations

**post** D not **in set** delegations;

--Get list of delegations

**public** getDelegations: () ==> **set** **of** Delegation

getDelegations() == (return delegations);

--Send product to delegation

**public** sendToDelegation: Delegation \* Products`Type \* **real** ==> ()

sendToDelegation(D,P,Q) == (D.addProductStock(P,Q);removeProductStock(P,Q);)

**pre** Q <= stocks.products(P) and P in **set** (dom stocks.products inter dom D.stocks.products);

--For each delegation associated with the producer, send products that they need, being equally distributed among all delegations that seek a certain product

**public** sendProducts: () ==> ()

sendProducts() == (

**dcl** n : **nat**;--number of delegations that sell the current product

**dcl** s : **nat**;

for **all** product **in set** (dom stocks.products) **do**(

n := 0;

for **all** delegation **in set** delegations **do** (

if(product in **set** delegation.getProducts()) **then** n:= n+1;

);

if(getStock(product) > 0) **then**(

s := getStock(product);

for **all** delegation **in set** delegations **do**(

if(product in **set** delegation.getProducts()) **then** sendToDelegation(delegation,product,s / n);

);

removeProductStock(product,getStock(product));

);

);

)**pre** exists p **in set** dom stocks.products & stocks.products(p) > 0;

**end** Producer

## 3.4 Class Delegation

**class** Delegation

**types**

**public** Name = **seq** **of** **char** **inv** n == len n > 0;

**instance variables**

**public** name: Name;

**public** stocks : Products;

**public** customers : **set** **of** Consumer;

**public** baskets : **map** Consumer **to** Basket;

--Baskets can only be sent to customers

**inv** dom baskets subset customers;

--Number of baskets is equal or less than number of customers

**inv** card dom baskets <= card customers;

--A basket can't be in the hands of the customer and in the delegation at the same time

**inv** forall c **in set** customers & elems c.baskets inter rng baskets = {};

**operations**

**public** Delegation: Name ==> Delegation

Delegation(N) == (name := N;stocks := new Products();customers := {};baskets := { |-> };return self);

--Add customer

**public** addCustomer: Consumer ==> ()

addCustomer(C) == (customers := customers union {C};C.setDelegation(self))

**pre** C not **in set** customers

**post** C in **set** customers;

--Remove customer

**public** removeCustomer: Consumer ==> ()

removeCustomer(C) == (customers := customers \ {C})

**pre** C in **set** customers

**post** C not **in set** customers;

--Add a product

**public** addProduct: Products`Type ==> ()

addProduct(P) == (stocks.addProduct(P););

--Remove a product

**public** removeProduct: Products`Type ==> ()

removeProduct(P) == (stocks.removeProduct(P));

--Get product list

**public** getProducts: () ==> **set** **of** Products`Type

getProducts() == (return stocks.getProducts());

--Add stock to a product

**public** addProductStock: Products`Type \* **real** ==> ()

addProductStock(P,S) == (stocks.addProductStock(P,S));

--Remove stock from a product

**public** removeProductStock: Products`Type \* **real** ==> ()

removeProductStock(P,S) == (stocks.removeProductStock(P,S));

--Get stock of a product

**public** getStock: Products`Type ==> **real**

getStock(P) == (return stocks.getStock(P));

--Make a new empty basket for a certain customer(requires that a basket does not already exist for this customer)

**public** makeBasket: Consumer ==> ()

makeBasket(C) == (

**dcl** b : Basket := new Basket(C.getBasketMaxSize());

baskets := baskets munion {C |-> b}

)

**pre** C in **set** customers and C not **in set** dom baskets

**post** C in **set** dom baskets;

--Make baskets for every customer

**public** makeBaskets: () ==> ()

makeBaskets() == (

for **all** customer **in set** customers **do**

makeBasket(customer)

);

--Fill the basket of a customer with the (3 or 4) highest stock products in the company, being proportionally distributed

**public** fillBasket: Consumer ==> ()

fillBasket(C) == (

**dcl** products : **seq** **of** Products`Type := highestN(stocks.products,Basket`maxNProducts(baskets(C).maxWeight));

**dcl** sum : **real** := 0;

for p **in** products **do**(

sum := sum + stocks.getStock(p);

);

for p **in** products **do**(

baskets(C).addProduct(p);

removeProductStock(p,stocks.getStock(p)\*baskets(C).getMaxWeight()/sum);

baskets(C).addProductStock(p,stocks.getStock(p)\*baskets(C).getMaxWeight()/sum);

);

)

**pre** C in **set** dom baskets and card dom baskets(C).stocks.products = 0 and card dom stocks.products > 0--No products are in the basket initially

**post** card dom baskets(C).stocks.products > 0;

--Fill baskets

**public** fillBaskets: () ==> ()

fillBaskets() == (

for **all** c **in set** customers **do**

fillBasket(c);

);

--Give basket to consumer and remove it from delegation

**public** giveBasket:Consumer ==> ()

giveBasket(C) == (

**dcl** B: Basket;

B := baskets(C);

baskets := {C} <-: baskets;

C.addBasket(B);

)

**pre** C in **set** dom baskets;

--Cancel basket from consumer(just remove from delegation)

**public** cancelBasket:Consumer ==> ()

cancelBasket(C) == (

baskets := {C} <-: baskets;

)

**pre** C in **set** dom baskets;

**functions**

--Returns the N products with highest stock with ascending order

**public** highestN: **map** Products`Type **to** Products`Stock \* **nat** -> **seq** **of** Products`Type

highestN(P,N) == (highestNaux(P,N,[]));

**public** highestNaux: **map** Products`Type **to** Products`Stock \* **nat** \* **seq** **of** Products`Type -> **seq** **of** Products`Type

highestNaux(P,N,L) == if(N = 0 or P = { |-> }) **then** L **else** let c = highest(P) **in** highestNaux({c} <-: P, N-1, L^[c]);

**public** highest: **map** Products`Type **to** Products`Stock -> Products`Type

highest(P) == highestaux(P,"None",-1)

**pre** card dom P > 0;

**public** highestaux: **map** Products`Type **to** Products`Stock \* Products`Type \* **real** -> Products`Type

highestaux(P,Biggest,Max) == if(P = { |-> }) **then** Biggest **else** (let p **in set** dom P **in** if(P(p) > Max) **then** highestaux({p} <-: P,p,P(p)) **else** highestaux({p} <-: P,Biggest,Max));

**end** Delegation

## 3.5 Class Consumer

**class** Consumer

**types**

**public** Name = **seq** **of** **char** **inv** n == len n > 0;

**instance variables**

**public** name : Name;--Name of the consumer

**public** maxbasketsize : Basket`MaxWeight;--Size of the baskets the consumer should receive

**public** baskets : **seq** **of** Basket;--Baskets the consumer has retrieved

**public** delegation : [Delegation];--Delegation

**operations**

**public** Consumer: **seq** **of** **char** \* Basket`MaxWeight ==> Consumer

Consumer(N,S) == (name := N; maxbasketsize := S;baskets := []; delegation := nil; return self);

--Get max size of the baskets he shall receive

**public** getBasketMaxSize: () ==> Basket`MaxWeight

getBasketMaxSize() == (return maxbasketsize);

--Get seq of baskets the customer has already received

**public** getBaskets: () ==> **seq** **of** Basket

getBaskets() == (return baskets);

--Add a new basket

**public** addBasket: Basket ==> ()

addBasket(B) == (baskets := [B]^baskets)

**pre** B not **in set** elems baskets

**post** B in **set** elems baskets;

--Define delegation

**public** setDelegation : Delegation ==> ()

setDelegation(D) == (delegation := D);

--Fetch the basket in case its available

**public** fetchBasket : () ==> ()

fetchBasket() == (

delegation.giveBasket(self)

);

--Cancel the current basket

**public** cancelBasket : () ==> ()

cancelBasket() == (

delegation.cancelBasket(self)

);

**end** Consumer

## 3.6 Class Basket

**class** Basket

**types**

**public** MaxWeight = **<Grande>** | **<Pequena>**;--Cesta grande ou pequena

**instance variables**

**public** maxWeight : MaxWeight;

**public** stocks : Products;

--O peso maximo da cesta nao pode ser excedido

**inv** totalWeight(stocks.products) <= maxWeightW(maxWeight);

--O número máximo de tipos de produtos não pode ser excedido(se for cesta de 4 quilos entao o maximo e 3, se for cesta de 8 quilos é 4)

**inv** card(dom stocks.products) <= maxNProducts(maxWeight);

**operations**

**public** Basket: MaxWeight ==> Basket

Basket(M) == (maxWeight := M ; stocks := new Products();return self);

--Add a product

**public** addProduct: Products`Type ==> ()

addProduct(P) == (stocks.addProduct(P);)

**pre** card(dom stocks.products) + 1 <= maxNProducts(maxWeight);

--Get product list

**public** getProducts: () ==> **set** **of** Products`Type

getProducts() == (return stocks.getProducts());

--Add stock to a product

--Add stock to a product

**public** addProductStock: Products`Type \* **real** ==> ()

addProductStock(P,S) == (stocks.addProductStock(P,S))

**pre** totalWeight(stocks.products) + S <= maxWeightW(maxWeight);

--Get stock of a product

**public** getStock: Products`Type ==> **real**

getStock(P) == (return stocks.getStock(P));

--get max weight

**public** getMaxWeight: () ==> **real**

getMaxWeight() == (return maxWeightW(maxWeight));

**functions**

--Total weight of products

**public** totalWeight: **map** Products`Type **to** Products`Stock -> **real**

totalWeight(Products) == (

if Products = { |-> } **then** 0 **else** let s **in set** dom Products **in** Products(s) + totalWeight({s} <-: Products)

);

--Max number of products

**public** maxNProducts: MaxWeight -> **nat**

maxNProducts(P) == (if P = <Pequena> **then** 3 **else** if P = <Grande> **then** 4 **else** 0);

--max weight

**public** maxWeightW: MaxWeight -> **real**

maxWeightW(P) == (if P = <Pequena> **then** 4 **else** if P = <Grande> **then** 8 **else** 0);

**end** Basket

# 4. Model validation

## 4.1 Class Tests

**class** Tests

**operations**

**protected** **static** assertTrue: **bool** ==> ()

assertTrue(arg) == (IO`print("Success\n"); return;)

**pre** arg;

**public** **static** testAll: () ==> ()

testAll() == (

IO`print("\n");

testOps();

);

**public** **static** testOps : () ==> ()

testOps() == (

**dcl** fruta:FrutaFeia := new FrutaFeia();

**dcl** p : Producer;

**dcl** d1: Delegation;

**dcl** d2:Delegation;

**dcl** c1:Consumer;

**dcl** c2:Consumer;

**dcl** c3:Consumer;

--Create a new Producer and add products to it

fruta.newProducer("p1");

fruta.addProductPro("p1","Banana");

fruta.addProductPro("p1","Manga");

fruta.addProductPro("p1","Pera");

p := fruta.getProByName(fruta.getProducers(),"p1");

--Assert that the products are correct

Tests`assertTrue(p.getProducts() = {"Banana","Manga","Pera"});

--Remove a product

fruta.removeProductPro("p1","Pera");

--Assert that the products are correct

Tests`assertTrue(p.getProducts() = {"Banana","Manga"});

--Add stock to the products and assert the stock is correct

fruta.addProductStockPro("p1","Banana",10);

Tests`assertTrue(p.getStock("Banana") = 10);

fruta.addProductStockPro("p1","Manga",15);

fruta.removeProductStockPro("p1","Banana",2);

Tests`assertTrue(p.getStock("Banana") = 8);

--Create new delegations

fruta.newDelegation("d1");

fruta.newDelegation("d2");

d1 := fruta.getDelByName(fruta.getDelegations(),"d1");

d2 := fruta.getDelByName(fruta.getDelegations(),"d2");

--Add products that the delegation wants

fruta.addProductDel("d1","Banana");

fruta.addProductDel("d1","Manga");

fruta.addProductDel("d2","Banana");

fruta.addProductDel("d2","Manga");

--Remove a product

fruta.removeProductDel("d2","Manga");

--Assert that the list of products is correct

Tests`assertTrue(d2.getProducts() = {"Banana"});

--Add delegation to list of delegations of the producer p1

fruta.addDelToPro("p1","d1");

fruta.addDelToPro("p1","d2");

--Assert that the delecation set is correct

Tests`assertTrue(fruta.getDelPro("p1") = {d1,d2});

--Remove delegation from producer

fruta.removeDelFromPro("p1","d2");

fruta.addDelToPro("p1","d2");

--Send producer products to its client delegations

fruta.sendToDels("p1");

--Assert that, having 2 delegations that want bananas, half the bananas were sent to each

Tests`assertTrue(d1.getStock("Banana") = 4);

--Create new customer

fruta.newCustomer("c1",<Grande>);

fruta.newCustomer("c2",<Pequena>);

fruta.newCustomer("c3",<Pequena>);

--Add customers to delegations

fruta.addCustomer("d1","c1");

fruta.addCustomer("d1","c2");

fruta.addCustomer("d1","c3");

c1 := fruta.getConByName(fruta.getConsumers(),"c1");

c2 := fruta.getConByName(fruta.getConsumers(),"c2");

c3 := fruta.getConByName(fruta.getConsumers(),"c3");

--Removea customer

fruta.removeCustomer("d1","c3");

--Assert that the list of customers of the delegation is correct

Tests`assertTrue(d1.customers = {c1,c2});

--Make baskets and fill them

fruta.makeAndFillBaskets("d1");

--Assert that 2 baskets were made

Tests`assertTrue(card dom d1.baskets = 2);

--c1 Fetch basket

fruta.fetchBasket("c1");

--Assert that the client got the basket

Tests`assertTrue(fruta.getNBaskets("c1") = 1);

--Cancel basket

fruta.cancelBasket("c2");

--Assert that the products in the basket are the products in the delegation(Banana and Manga)

Tests`assertTrue(fruta.getLastBasket("c1").getProducts() = {"Banana","Manga"});

--Assert that the basket has more Manga than Banana, because the delegation had more Manga than Banana(15 vs 4)

Tests`assertTrue(fruta.getLastBasket("c1").getStock("Manga") > fruta.getLastBasket("c1").getStock("Banana"));

);

**end** Tests

# 5. Model verification

## 5.1 Example of domain verification

One of the proof obligations generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 17 | Delegation`fillBasket(Consumer) | legal map application |

The code under analysis (with the relevant map application underlined) is:

**public** fillBasket: Consumer ==> ()

fillBasket(C) == (

**dcl** products : **seq** **of** Products`Type := highestN(stocks.products,Basket`maxNProducts(baskets(C).maxWeight));

**dcl** sum : **real** := 0;

**for** p **in** products **do**(

sum := sum + stocks.getStock(p);

);

**for** p **in** products **do**(

baskets(C).addProduct(p)

removeProductStock(p,stocks.getStock(p)\*baskets(C).getMaxWeight()/sum);

baskets(C).addProductStock(p,stocks.getStock(p)\*baskets(C).getMaxWeight()/sum);

);

)

and the generated proof obligation is:

(forall C:Consumer & (((C in set (dom baskets)) and (((card (dom ((baskets(C).stocks).products))) = 0) and ((card (dom (stocks.products))) > 0))) => ((C in set (dom baskets)) => (C in set (dom baskets)))))

In this case, the proof indicates that for every Customer of the delegation, the Cust omer must have an empty basket :

“(C in set (dom baskets)) and (((card (dom ((baskets(C).stocks).products))) = 0)”

And there must be enough products to distribute to every customer’s baskets:

“((card (dom (stocks.products))) > 0))) => ((C in set (dom baskets)) => (C in set (dom baskets))))”

## 5.2 Example of invariant verification

Another proof obligation generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 1 | Basket`Basket(Basket`MaxWeight) | state invariant holds |

The code under analysis (with the relevant state changes underlined) is:

**public** addProductStock: Products`Type \* **real** ==> ()

addProductStock(P,S) == (stocks.addProductStock(P,S))

**pre** totalWeight(stocks.products) + S <= maxWeightW(maxWeight);

The relevant invariant under analysis is:

**inv** totalWeight(stocks.products) <= maxWeightW(maxWeight);

The pre-condition implies that the total weight that the basket already has together with the stock to be increased must be smaller than the max-weight of the basket.

This is the only operation that can increase the stock of the product in the basket.

Therefore, if the pre-condition is not broken, the invariant that states that the total weight of the basket must be smaller or equal to the max weight can never be broken.

# 6. Java Code Generation

The overture tool offers the possibility of generating Java Code from the VDM++ model that was developed.

After doing the necessary steps to generate the code, a few aspects were observed.

The first and most important is that all the pre-conditions, post-conditions and invariants are missing. This is a limitation of the Overture Ide. Because of this, the Java program that uses an interface will crash in case any operation that would break the pre-conditions is executed.

The second aspect is that for Enums, instead of using the Enum types that Java offers, the Overture Ide generated a different Class for each enum value.

Nonetheless, the execution of the program was as expected, as long as the pre-conditions are not broken, where the program might crash.

It was also developed a console interface that uses the defined requirements of the project to manipulate the data of FrutaFeia, but as mentioned, the pre-conditions and invariants can be broken because they were not generated in Java.

# 7. Conclusions

The model that was developed covers all the requirements.

Among the possible improvements to the product, the biggest ones would be:

1- Make an interface that allows more flexibility with the operations

2- Add a set of prefered products to the customers, and the baskets would be made preferably with these products.

3- Make more extensive tests to more possibilities and with invalid inputs

This project took approximately 70 hours to develop.

The end result program was a lot bigger than anticipated, and proved hard to analyse proof obligations due to the generated ones being slightly complex.

# 8. References

1. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
2. Overture tool web site, http://overturetool.org

1. [↑](#footnote-ref-2)
2. [↑](#footnote-ref-3)