Formal Modeling of the Glovo Application in VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Carolina Azevedo - up201506509

João Mendes - up201505439

Table of Contents

1. Informal system description and list of requirements

1.1 Informal system description

The modeled system represents the basic usage of the Glovo Application. Glovo is the app that allows you to get the best products in your city delivered to your location, not just food but any product.

The main functionality of the app is to order any product from a supplier and get it delivered to you in the least time possible while still guaranteeing the products quality. You can: place an order, check the order status, edit the order or even cancel it (at the cost of a fee) from a range of suppliers whom you can search and all their products.

If you’re a supplier you can add or remove your products and change their price.

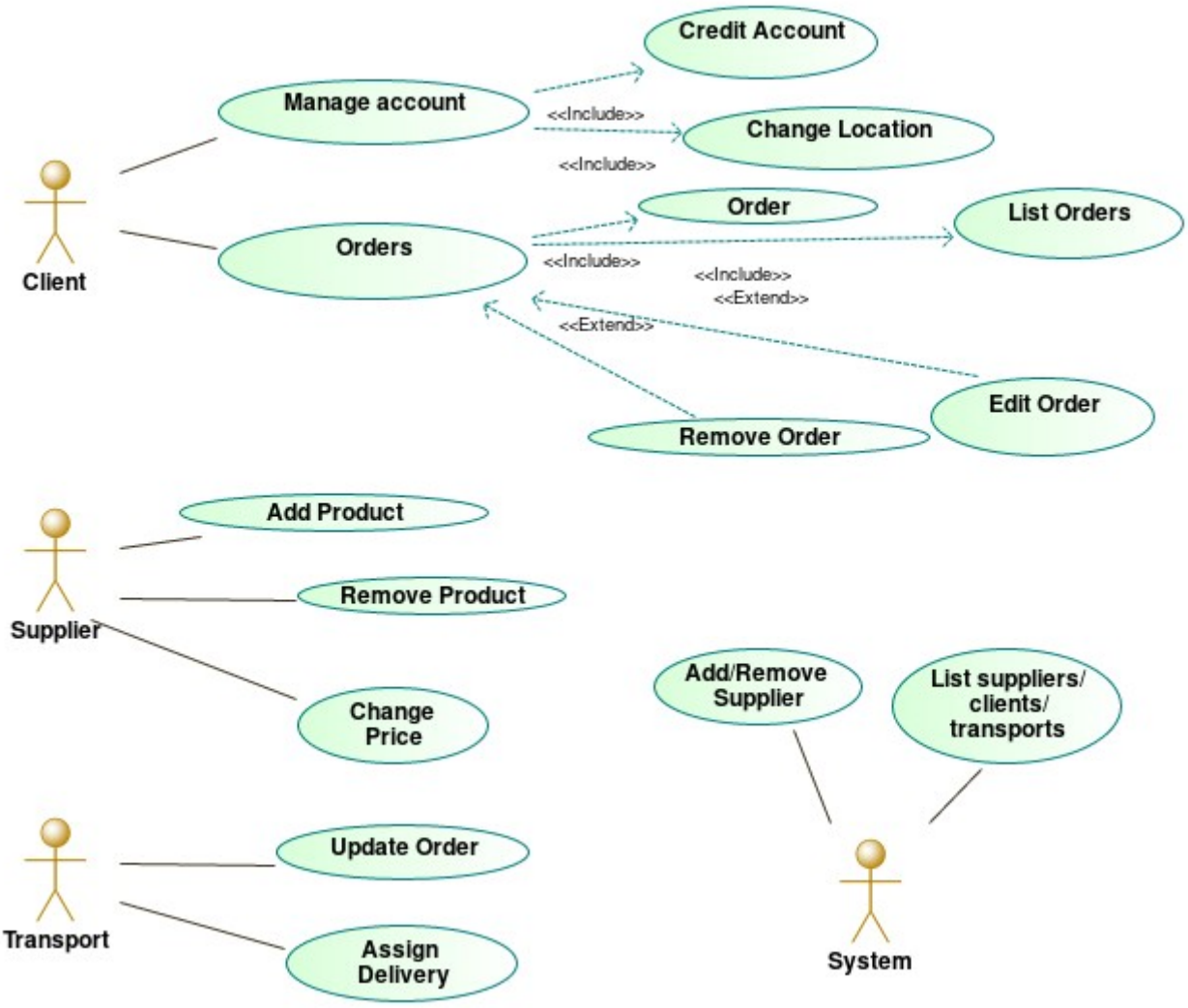
If you’re a transporter you can assign to yourself unredeemed deliveries and update your status.

1.2 List of requirements

Id	Priority	Description
R1	Mandatory	Clients can join the system.
R2	Mandatory	Clients can place orders in the system, choosing the best transport automatically.
R3	Mandatory	Clients can edit their orders, mainly the products involved and their money is refunded and newly charged.
R4	Mandatory	Clients can remove their orders, and their money is refunded.
R5	Mandatory	Clients can change their location.
R6	Mandatory	Clients can credit their accounts to fund their purchases.
R7	Mandatory	Supplier can add/remove products from their menu.
R8	Mandatory	Supplier can edit their products price, for future transactions.
R9	Mandatory	Transporter can assign himself to untaken orders.
R10	Mandatory	Transporter can update a deliveries status
R11	Optional	Client can search for a supplier.
R12	Optional	Client can see his purchase history.
R13	Optional	System operator can list clients, suppliers and transports.
R14	Optional	System operator can add/remove suppliers and transports.

2. Visual UML model

2.1 Use case model



The major use case scenarios are described next.

Scenario	**Manage Account**
Description	Credit Client balance or change its location
Pre-conditions	1. The Client must be part of the system. 2. Credit is positive or location is valid
Post-conditions	1. The Client parameter state has changed
Steps	The Client updates its account information.
Exceptions	The Client isn't in the system.

Scenario	**Order**

Description	The Client makes a Delivery request
Pre-conditions	1. The user has enough balance in its account. 2. The products chosen are from the same supplier.
Post-conditions	1. The users balance has been decreased. 2. A new Delivery has been created.
Steps	1. The client picks a supplier. 2. The client picks products and quantity from the menu. 3. The client orders and pays.
Exceptions	1. The client doesn’t have enough credit.

Scenario	**Remove Order**
Description	The Client makes a delete Delivery request
Pre-conditions	1. The Delivery exists. 2. The Delivery has no Transport assigned.
Post-conditions	1. The users balance has been increased (fee charged). 2. A new Delivery has been removed.
Steps	1. The client picks a delivery. 2. The client cancels the delivery.
Exceptions	1. The delivery is already underway.

Scenario	**Edit Order**
Description	The Client makes an edit Delivery request
Pre-conditions	1. The Delivery exists. 2. The Delivery has no Transport assigned.
Post-conditions	1. The users balance has been changed(fee charged).
Steps	1. The client picks a delivery. 2. The client edits the delivery.
Exceptions	1. The delivery is already underway.

Scenario	**List Orders**
Description	The Client sees their purchase history.
Pre-conditions	1. The Client is in the system.
Post-conditions	(unspecified)
Steps	1. The client checks their orders under Orders.
Exceptions	1. The client is not in the system.

Scenario	**Remove Product**
Description	A Supplier can remove a product from their menu
Pre-conditions	1. The Supplier exists. 2. The Product exists in the suppliers menu.

Post-conditions	1. The product no longer exists in the menu.
Steps	1. The supplier picks a product.
Exceptions	1. The Product doesn’t exist in the suppliers menu.

Scenario	**Add Product**
Description	A Supplier can add a new product to their menu
Pre-conditions	1. The Supplier exists. 2. The Product doesn’t exist in the suppliers menu.
Post-conditions	1. The product now exists in the menu.
Steps	1. The supplier enters a product.
Exceptions	1. The Product exists in the suppliers menu.

Scenario	**Change Price**
Description	A Supplier can a products price on their menu
Pre-conditions	1. The Supplier exists. 2. The Product exists in the suppliers menu. 3. The new price is a valid value.
Post-conditions	1. The price has changed.
Steps	1. The supplier picks a product. 2. The supplier enters a new price.
Exceptions	1. The Product doesn’t exist in the suppliers menu.

Scenario	**Update Order**
Description	A Transport can udate an orders state
Pre-conditions	1. The Transport exists. 2. The Order exists. 3. The Order isn’t finished.
Post-conditions	1. The order status has changed.
Steps	1. The transport picks a order. 2. The transport changes order state.
Exceptions	1. The Transport doesn’t exist. 2. The Order selected isn’t finished.

Scenario	**Assign Delivery**
Description	A Transport can assign a delivery to itself
Pre-conditions	1. The Transport exists. 2. The Order exists. 3. The Order isn’t finished.

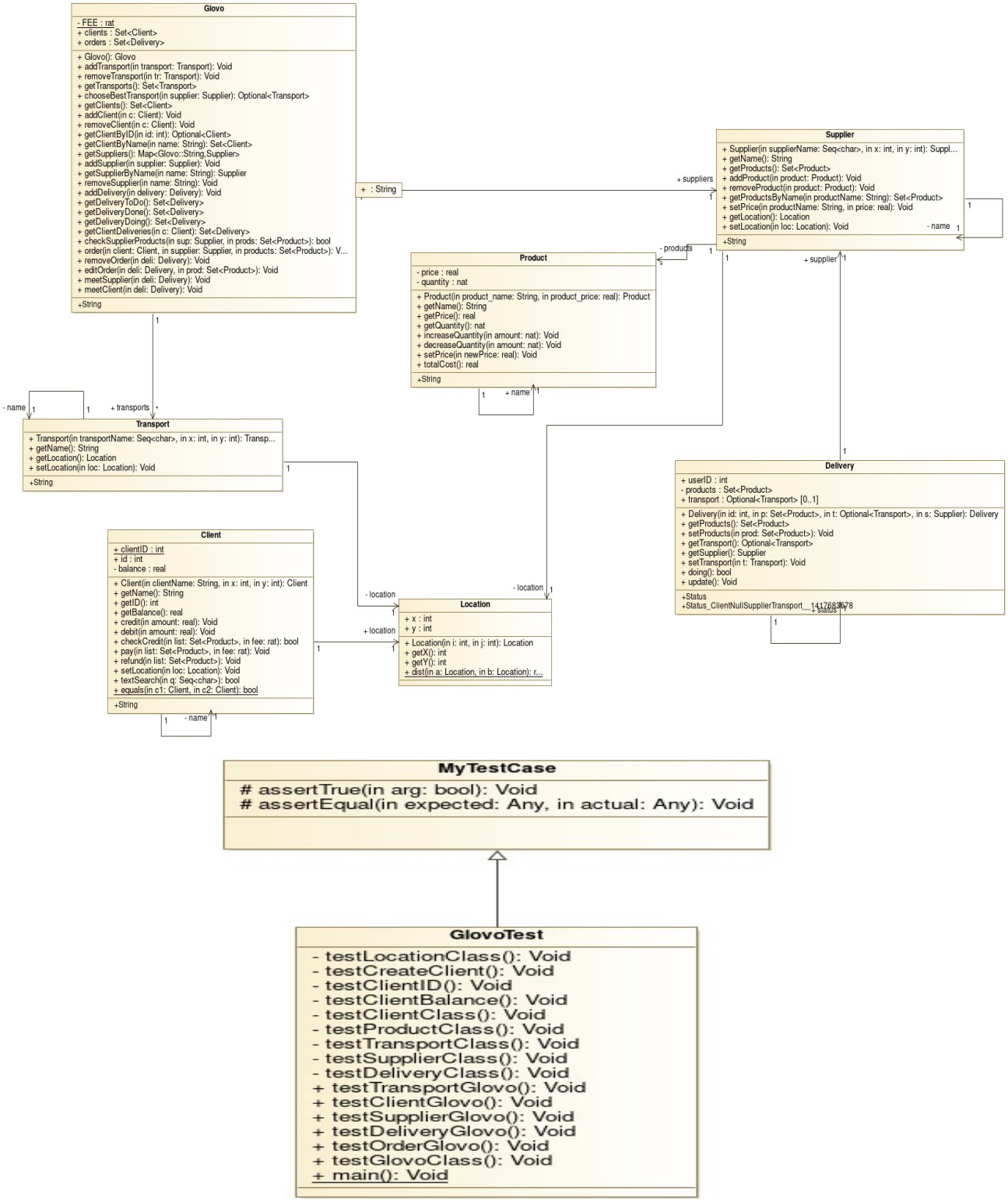
Post-conditions	1. The order transport has changed.
Steps	1. The transport picks a order. 2. The transport changes order transport to itself.
Exceptions	1. The Transport doesn’t exist. 2. The Order selected is finished.

Scenario	**Add Supplier**
Description	System can add new suppliers
Pre-conditions	1. The Supplier doesn’t exist.
Post-conditions	1. The supplier now exists in the system.
Steps	1. The system enters a supplier.
Exceptions	1. The Supplier exists in the suppliers listing.

Scenario	**Remove Supplier**
Description	System can remove a supplier
Pre-conditions	1. The Supplier exists. 2. The Supplier has no pending deliveries.
Post-conditions	1. The product no longer exists in the system.
Steps	1. The system picks a supplier.
Exceptions	1. The Supplier doesn’t exist in the suppliers list.

Scenario	**System Listings**
Description	Credit Client balance or change its location
Pre-conditions	(unspecified)
Post-conditions	(unspecified)
Steps	The System accesses information.
Exceptions	(unspecified)

2.2 Class model



Class	Description
`Location`	Defines operation and functions for location handling, used mostly in other classes.
`Product`	Defines a product at sale in a supplier; can also be found in a Delivery.
`Glovo`	Core model; defines the state variables and operations available to the clients.
`MyTestCase`	Superclass for test classes; defines `assertEquals` and `assertTrue`.
`GlovoTest`	Defines the test cases for the glovo application.
Client	Represents a client in the application with some basic data.
Supplier	Represents a supplier in the application, saving its products.
Transport	Represents a mean of transport for the application; the best one is attributed to each particular Delivery.
Delivery	Represents a delivery in the system, done or to be done, keeping its Transport and maintaining it busy.

3. Formal VDM++ model

3.1 Class Location

```
class Location
instance variables

            public x: int;

            public y: int;

operations

public Location: int * int ==> Location

Location(i,j) == (

            x := i;

            y := j;

            return self

);

-- Returns x

public getX : () ==> int

            getX() ==

            return x;

-- Returns y

public getY : () ==> int
```

```
            getY() ==
            return y;

functions

-- evaluates distance between points
public dist: Location * Location -> real
      dist(a,b) ==
      ((b.x-a.x)*(b.x-a.x))+((b.y-a.y)*(b.y-a.y));
end Location
```

3.2 Class Product

```
class Product
types
public String = seq of char;
instance variables
            public name: String;
            price: real;
            quantity: nat := 0;
operations
      public Product: String * real ==> Product
      Product(product_name,product_price) == (
                  name := product_name;
                  price := product_price;
                  quantity := 1;
                  return self
      );
      -- Returns product name
      public pure getName : ()  ==> String
            getName() ==
            return name;
      -- Returns product price
```

```
	public pure getPrice : () ==> real
		getPrice() ==
		return price;
	-- Returns product quantity
	public pure getQuantity : () ==> nat
		getQuantity() == return quantity;

-- Increases product quantity
public increaseQuantity : nat ==> ()
	increaseQuantity(amount) ==
		quantity := quantity + amount
	pre (quantity + amount) >= 0;

-- Decreases product quantity
public decreaseQuantity : nat ==> ()
	decreaseQuantity(amount) ==
		quantity := quantity - amount
	pre (quantity - amount) >= 0
	post quantity~ = quantity + amount;

-- Change product price
public setPrice : real ==> ()
	setPrice(newPrice) ==
	price := newPrice
	post newPrice = price;

-- Total cost of the product
pure public totalCost : () ==> real
	totalCost() ==
	return quantity * price;
end Product
```

3.3 Class Glovo

```
class Glovo

types

public String = seq of char;


values

FEE = 1.9;


instance variables

            public clients: set of Client :={};

            public orders: set of Delivery :={};

            public suppliers: map String to Supplier :={|->};

            public transports: set of Transport :={};


            inv not exists c1, c2 in set clients &

                                c1 <> c2 and c1.id = c2.id;


operations

public Glovo: () ==> Glovo

            Glovo() == return self;


-- Adds a transport

public addTransport: Transport ==> ()

            addTransport(transport) ==

            transports := transports union {transport}

            pre transport not in set transports

            post transports = transports~ union {transport};


-- Remove a transport

public removeTransport: Transport ==> ()
```

```
            removeTransport(tr) ==
            transports := transports \ {tr}
            pre tr in set transports;

-- Returns the Transports available
public getTransports: () ==> set of Transport
            getTransports() ==
            return transports;

-- Get the best transport
public chooseBestTransport : Supplier ==> [Transport]
      chooseBestTransport(supplier) == (
            dcl best:[Transport] := nil;

            for all transport in set transports do
                              if best = nil or
Location`dist(supplier.getLocation(), transport.getLocation()) >
Location`dist(supplier.getLocation(), best.getLocation())
            then  best := transport;

            return best;
      );

-- Returns clients
pure public getClients: () ==> set of Client
            getClients() ==
            return clients;

-- Adds a client to the system
public addClient :Client ==> ()
```

```
            addClient(c) ==
            clients := clients union {c}
            pre not exists client in set clients & Client`equals(client, c)
            post c in set clients;

-- Removes a client to the system
public removeClient : Client ==> ()
            removeClient(c) ==
            (
                  for all client in set clients do
                        if Client`equals(client, c)
                        then clients := clients \ {client};
            )
            pre exists client in set clients & Client`equals(client, c)
            post not exists client in set clients & Client`equals(client, c);

-- Gets Client by ID
public getClientByID : int ==> [Client]
            getClientByID(id) ==
            (
                        dcl target:[Client] := nil;
                        for all client in set clients do
                                    if client.id = id then target := client;
                        return target;
            );

-- Returns a client by its name
public getClientByName : String ==> set of Client
            getClientByName(name) ==
            return {client | client in set clients & client.textSearch(name)};
```

```
-- Returns the suppliers
public getSuppliers: () ==> map String to Supplier
            getSuppliers() ==
            return suppliers;


-- Adds a supplier to the system
public addSupplier : Supplier ==> ()
      addSupplier(supplier) == suppliers := suppliers munion {supplier.getName()
|-> supplier}
      pre supplier.getName() not in set dom suppliers
      post suppliers = suppliers~ munion {supplier.getName() |-> supplier};


-- Returns a list of suppliers by name
public getSupplierByName: String ==> Supplier
            getSupplierByName(name) ==
            return suppliers(name)
            pre name in set dom suppliers;


-- Remove a supplier
public removeSupplier : String ==> ()
            removeSupplier(name) ==
            (
                        dcl result:bool := false;


                        for all order in set orders do
                              if order.supplier.getName() = name and order.status
<> <Client> then
                                          result := true;


                        if not result then
                              suppliers :=  {name}<-:suppliers;
```

```
            )
            pre name in set dom suppliers;

-- Adds a delivery
public addDelivery: Delivery ==> ()
            addDelivery(delivery) == orders := orders union {delivery}
            pre delivery not in set orders
            post orders = orders~ union {delivery};

-- Returns waiting deliveries
public pure getDeliveryToDo: () ==> set of Delivery
            getDeliveryToDo() ==
            return {order | order in set orders & order.status = <Null>};

-- Returns done deliveries
public getDeliveryDone: () ==> set of Delivery
            getDeliveryDone() ==
            return {order | order in set orders & order.status = <Client>};

-- Returns undergoing deliveries
public getDeliveryDoing: () ==> set of Delivery
            getDeliveryDoing() ==
            return {order | order in set orders & order.doing()};

-- Returns clients deliveries
public getClientDeliveries: Client ==> set of Delivery
            getClientDeliveries(c) ==
            (
                  dcl result:set of Delivery := {};
```

```
                for all order in set orders do
                        if order.userID = c.id
                                then result := result union {order};

                return result;
        )
        pre c in set clients;

-- Checks if products are in supplier
pure public checkSupplierProducts: Supplier * set of Product ==> bool
        checkSupplierProducts(sup, prods) ==
        (
                for all product in set prods do
                        if product not in set sup.getProducts()
                                then return false; -- doesn't get coverage because
breaks the precondition, coverage run would always stop here
                return true;
        )
        pre prods <> {};

-- Place an order
public order : Client * Supplier * set of Product ==> ()
        order(client, supplier, products) == (
                dcl transport:[Transport] := chooseBestTransport(supplier);

                if client.checkCredit(products, FEE) then client.pay(products, FEE);

                addDelivery(new Delivery(client.id, products, transport, supplier));
                if transport <> nil
                        then removeTransport(transport);
        )
```

```
      pre client.checkCredit(products, FEE) and checkSupplierProducts(supplier,
products);


-- Remove order
public removeOrder : Delivery ==> ()
            removeOrder(deli) ==
            (
                  orders := orders \ {deli};


                  getClientByID(deli.userID).refund(deli.getProducts());
            )
            pre deli in set getDeliveryToDo();


-- Edit an order
public editOrder : Delivery * set of Product ==> ()
            editOrder(deli, prod) ==
            (
                  getClientByID(deli.userID).refund(deli.getProducts());


                  if(getClientByID(deli.userID).checkCredit(prod, FEE))
                        then (
                              getClientByID(deli.userID).pay(prod, FEE);
                              deli.setProducts(prod);
                        )
                        else removeOrder(deli);
            )
            pre deli in set getDeliveryToDo();


-- Transport meets supplier
public meetSupplier : Delivery ==> ()
            meetSupplier(deli) ==
```

```
            (
                  deli.transport.setLocation(deli.supplier.getLocation());
                  deli.update();
            )
            pre deli in set {order | order in set orders & order.status =
<Transport>};


-- Transport meets client
public meetClient : Delivery ==> ()
            meetClient(deli) ==
            (

      deli.transport.setLocation(getClientByID(deli.userID).location);
                  deli.update();
                  addTransport(deli.transport);
            )
            pre deli in set {order | order in set orders & order.status =
<Supplier>};
end Glovo
```

3.4 Class Client

```
class Client
types
public String = seq of char;
instance variables
            name: String;
            public static clientID: int := 0;
            public id : int := clientID;
            balance: real;
            public location: Location;
            inv balance >= 0;


operations
```

```
public Client: String * int * int ==> Client
Client(clientName, x, y) == (
            name := clientName;
            id := clientID;
            clientID := clientID + 1;
            balance := 5;
            location := new Location(x,y);
            return self
);

-- Returns user name
public pure getName : () ==> String
            getName() ==
            return name;

-- Returns user ID
public getID: () ==> int
            getID() ==
            return id;

-- Returns user balance
public getBalance: () ==> real
            getBalance() ==
            return balance;

-- User credit
public credit : real ==> ()
            credit(amount) ==
            balance := balance + amount
      pre (balance + amount) >= 0 and amount > 0;
```

```
-- User debit
public debit : real ==> ()
            debit(amount) ==
            balance := balance - amount
      pre (balance - amount) >= 0 and amount > 0
      post balance~ = balance + amount;

-- Checks if user has enough credit
pure public checkCredit : set of Product * rat ==> bool
            checkCredit(list, fee) ==
            (
                  dcl sum: real := 0;

                  for all product in set list do
                              sum:= sum + product.totalCost();

                  return balance > sum + fee;
            )
            pre list <> {};

-- Pay for products
public pay : set of Product * rat ==> ()
            pay(list, fee) ==
            (
                  dcl sum: real := 0;

                  for all product in set list do
                              sum:= sum + product.totalCost();

                  if balance > sum + fee then debit(sum+fee);
```

```
            )
            pre list <> {};

-- Refund for products
public refund : set of Product ==> ()
            refund(list) ==
            (
                  dcl sum: real := 0;

                  for all product in set list do
                              sum:= sum + product.totalCost();

                  credit(sum);
            )
            pre list <> {};

-- Changes Location
public setLocation : Location ==> ()
            setLocation(loc) ==
                        location := loc;

-- Search for name
public textSearch : seq of char ==> bool
textSearch(q) == (
      dcl tmp: seq of char := name;
      dcl match: bool := false;

      while len tmp >= len q and not match do(
            match := true;
```

```
            for index = 1 to len q do
                  if match and q(index) <> tmp(index) then (
                        match := false;
                  );

            if match then
                  return true
            else (
                  tmp := tl tmp;
                  match := false;
            );
      );
      return false;
)
pre len q > 0;

functions
-- Compares 2 clients by ID
public equals : Client * Client -> bool
      equals(c1, c2) ==
      c1.id = c2.id;
end Client
```

3.5 Class Supplier

```
class Supplier
types
public String = seq of char;
instance variables
            products: set of Product :={};
            name: String;
            location: Location;
operations
```

```
public Supplier: seq of char * int * int==> Supplier
Supplier(supplierName, x, y) == (
            name := supplierName;
            products := {};
            location := new Location(x, y);
            return self
);

-- Returns supplier name
public pure getName : () ==> String
            getName() ==
            return name;

-- Returns suppliers products
pure public getProducts: () ==> set of Product
                  getProducts()  ==
return products;

-- Adds a product to suppliers menu
public addProduct : Product ==> ()
            addProduct(product) ==
            products := products union {product}
            pre product not in set products
            post products = products~ union {product};

-- Remove product from suppliers menu
public removeProduct : Product ==> ()
            removeProduct(product) ==
            products := products  \ {product}
            pre product in set products
```

```
            post products~ = products union {product} and product not in set
products;


-- Returns a product by name
public getProductsByName : String ==> set of Product
            getProductsByName(productName) ==
            return {product | product in set products & product.name =
productName};


-- Change product price
public setPrice : String * real ==> ()
      setPrice(productName, price) ==
      for all product in set getProductsByName(productName) do
            product.setPrice(price);


-- Returns transport Location
public pure getLocation : () ==> Location
            getLocation() ==
            return location;


-- Changes Location
public setLocation : Location ==> ()
            setLocation(loc) ==
                        location := loc;
end Supplier
```

3.6 Class Transport

```
class Transport
types
public String = seq of char;
instance variables
            name: String;
```

```
            location: Location;

operations
public Transport: seq of char * int * int ==> Transport
Transport(transportName, x, y) == (
            name := transportName;
            location := new Location(x, y);
            return self
);

-- Returns transport name
public pure getName : () ==> String
            getName() ==
            return name;

-- Returns transport Location
public pure getLocation : () ==> Location
            getLocation() ==
            return location;

-- Changes Location
public setLocation : Location ==> ()
            setLocation(loc) ==
                        location := loc;
end Transport
```

3.7 Class Delivery

```
class Delivery
types
public Status = <Null> | <Transport> | <Supplier> | <Client>
instance variables
            public userID : int;
```

```
        public supplier: Supplier;
        products: set of Product :={};
        public transport: [Transport];
        public status: Status;

        inv status <> nil;
        inv userID > 0;

operations
public Delivery: int * set of Product * [Transport] * Supplier ==> Delivery
        Delivery(id, p,t, s) == (
              userID := id;
              products := p;
              supplier := s;
              transport := t;
              status := <Null>;
              if(t <> nil) then status := <Transport>;
              return self
        );

-- Returns the delivery products
pure public getProducts: () ==> set of Product
        getProducts()  ==
return products;

-- Changes the delivery products
public setProducts: set of Product ==> ()
        setProducts(prod)  ==
              products := prod
        pre prod <> {};
```

```
-- Returns the current transport
pure public getTransport: () ==> [Transport]
            getTransport() ==
return transport;

-- Returns the supplier
pure public getSupplier: () ==> Supplier
            getSupplier() ==
return supplier;

-- Sets a transport for the delivery
public setTransport: Transport ==> ()
            setTransport(t) == (
                  transport := t;
                  status := <Transport>;
            )
            post status = <Transport>;

-- Returns if delivery is underway
public doing: () ==> bool
            doing() ==
            return status = <Supplier> or status = <Transport>;

-- Updates the Delivery State
public update: () ==> ()
            update() ==
            (
                  cases status:
                        <Transport> -> status := <Supplier>,
                        <Supplier> -> status := <Client>
```

```
                end
            );
end Delivery
```

4. Model validation

4.1 Class MyTestCase

```
class MyTestCase
/*
  Superclass for test classes, simpler but more practical than VDMUnit`TestCase.
  For proper use, you have to do: New -> Add VDM Library -> IO.
  JPF, FEUP, MFES, 2014/15.
*/
operations
      -- Simulates assertion checking by reducing it to pre-condition checking.
      -- If 'arg' does not hold, a pre-condition violation will be signaled.
      protected assertTrue: bool ==> ()
      assertTrue(arg) ==
            return
      pre arg;
      -- Simulates assertion checking by reducing it to post-condition checking.
      -- If values are not equal, prints a message in the console and generates
      -- a post-conditions violation.
      protected assertEqual: ? * ? ==> ()
      assertEqual(expected, actual) ==
            if expected <> actual then (
                  IO`print("Actual value (");
                  IO`print(actual);
                  IO`print(") different from expected (");
                  IO`print(expected);
                  IO`println(")\n")
            )
```

```
      post expected = actual
  end MyTestCase
```

4.2 Class TestVendingMachine

```
class GlovoTest is subclass of MyTestCase
instance variables


client1: Client := new Client("carolina", 0, 0);
client2: Client := new Client("joao", 5, 0);
transport1: Transport := new Transport("mota", 0, 5);
transport2: Transport := new Transport("carro", 5, -5);
supplier1: Supplier := new Supplier("mac", -5, 0);
supplier2: Supplier := new Supplier("pizza", -5, 5);

product1: Product := new Product("burguer", 1);
product2: Product := new Product("pizza", 2);


operations
-- Tests related to the Location Class
private testLocationClass: () ==> ()
testLocationClass() ==
(
      dcl loc1: Location := new Location(0,0),
            loc2: Location := new Location(1,0);
            assertTrue(loc1.getX() = 0);
            assertTrue(loc1.getY() = 0);
            assertTrue(loc2.getX() = 1);
            assertTrue(loc2.getY() = 0);

            assertEqual(Location`dist(loc1,loc2),1);
      );
```

```
-- Tests a client Creation
private testCreateClient: () ==> ()
	testCreateClient() ==
	( -- Requirement 1
		dcl client: Client := new Client("x", 0 ,0);
		assertEqual(client.getName(), "x");
		assertEqual(client.getBalance(), 5);
		assertEqual(client.location.getX(), 0);
		assertEqual(client.location.getY(), 0);
		-- Requirement 5
		client.setLocation(new Location(3,4));
		assertEqual(client.location.getX(), 3);
		assertEqual(client.location.getY(), 4);
	);

-- Tests clients different ids
private testClientID: () ==> ()
	testClientID() ==
	(
		dcl x: Client := new Client("carolina", 0, 0),
			y: Client := new Client("joao", 5, 0);
		assertTrue(x.getID() < y.getID());
		assertTrue(not Client`equals(x,y));
	);

-- Tests movements in a clients account
private testClientBalance: () ==> ()
	testClientBalance() == -- Requirement 6
	(
```

```
            dcl client: Client := new Client("carolina", 0, 0),
                        pr: Product := new Product("mota", 1);
            client.credit(15);
            assertEqual(client.getBalance(), 20);
            client.debit(5);
            assertEqual(client.getBalance(), 15);
            pr.increaseQuantity(4);

            assertTrue(client.checkCredit({pr}, 2));

            client.pay({pr}, 2);
            assertEqual(client.getBalance(), 8);

            client.refund({pr});
            assertEqual(client.getBalance(), 13);

            client.refund({pr});
            assertEqual(client.getBalance(), 18);
      );

-- Tests related to the client class
private testClientClass: () ==> ()
      testClientClass() ==
      (
            testCreateClient();
            testClientID();
            testClientBalance();
      );

-- Tests related to the product class
```

```
private testProductClass: () ==> ()
	testProductClass() ==
	(
		dcl pr: Product := new Product("mota", 1);
		assertTrue(pr.getName() = "mota");
		assertEqual(pr.getQuantity(), 1);
		assertEqual(pr.getPrice(), 1);
		pr.increaseQuantity(5);
		assertEqual(pr.getQuantity(), 6);
		pr.decreaseQuantity(3);
		assertEqual(pr.getQuantity(), 3);
		pr.setPrice(5.6);
		assertTrue(pr.getPrice() = 5.6);
		assertEqual(pr.totalCost(), 5.6*3);
	);

-- Tests related to the Transport Class
private testTransportClass: () ==> ()
	testTransportClass() ==
	(
		dcl transport: Transport := new Transport("mota", 1, 1);
		assertTrue(transport.getName() = "mota");
		assertEqual(transport.getLocation().getX(), 1);
		assertEqual(transport.getLocation().getY(), 1);

		transport.setLocation(new Location(3,4));
		assertEqual(transport.getLocation().getX(), 3);
		assertEqual(transport.getLocation().getY(), 4);
	);
```

```
-- Tests related to the Supplier Class
private testSupplierClass: () ==> ()
      testSupplierClass() ==
      (-- Requirement 7
            dcl sr: Supplier := new Supplier("pizza", -5, 5),
            pr1: Product := new Product("burguer", 1),
            pr2: Product := new Product("pizza", 2);

            assertTrue(sr.getName() = "pizza");
            assertEqual(sr.getLocation().getX(), -5);
            assertEqual(sr.getLocation().getY(), 5);
            assertEqual(sr.getProducts(), {});

            assertEqual(sr.getProducts(), {});
            sr.addProduct(pr1);
            assertEqual(sr.getProducts(), {pr1});
            --sr.addProduct(pr1); -- this intentionally breaks precondition
            assertEqual(sr.getProducts(), {pr1});
            sr.addProduct(pr2);
            assertEqual(sr.getProducts(), {pr1, pr2});

            assertEqual(sr.getProductsByName("burguer"), {pr1});
            sr.setPrice("burguer", 5);
            pr1.setPrice(5); -- Requirement 8
            assertEqual(sr.getProductsByName("burguer"), {pr1});
            sr.removeProduct(pr1);
            assertEqual(sr.getProducts(), {pr2});

            sr.setLocation(new Location(3,4));
            assertEqual(sr.getLocation().getX(), 3);
```

```
            assertEqual(sr.getLocation().getY(), 4);
      );

-- Tests related to the Delivery Class
private testDeliveryClass: () ==> ()
      testDeliveryClass() ==
      (
            dcl sr: Supplier := new Supplier("pizza", -5, 5),
                  pr1: Product := new Product("burguer", 1),
                  pr2: Product := new Product("pizza", 2),
                  tr1: Transport := new Transport("mota", 0, 5);

            dcl deli1: Delivery := new Delivery(1, {pr1}, nil, sr),
                  deli2: Delivery := new Delivery(1, {pr1, pr2}, tr1, sr);

            assertEqual(deli1.getSupplier(), sr);
            assertEqual(deli2.getSupplier(), sr);
            assertEqual(deli1.getProducts(), {pr1});
            assertEqual(deli2.getProducts(), {pr1,pr2});
            deli1.setProducts({pr1,pr2});
            assertEqual(deli1.getProducts(), {pr1,pr2});

            assertEqual(deli1.getTransport(), nil);
            assertEqual(deli2.getTransport(), tr1);
            assertEqual(deli1.status, <Null>);
            assertTrue(not deli1.doing());
            assertTrue(deli2.doing());
            deli1.setTransport(tr1);
            assertEqual(deli1.getTransport(), tr1);
            assertEqual(deli1.status, <Transport>);
```

```
            assertTrue(deli1.doing());
            assertTrue(deli2.doing());

            deli1.update();
            assertEqual(deli1.status, <Supplier>);
            assertTrue(deli1.doing());

            deli1.update();
            assertEqual(deli1.status, <Client>);
            assertTrue(not deli1.doing());
      );

-- Tests related to Transport in Glovo
public testTransportGlovo: () ==> ()
            testTransportGlovo() ==
            (
            dcl glovo:Glovo := new Glovo(),
                  a:Transport := new Transport("mota",5,0),
                  b:Transport := new Transport("bicla", 2, 0),
                  sr: Supplier := new Supplier("pizza", 0, 0);

                  assertEqual(glovo.getTransports(), {});
                  glovo.addTransport(a);
                  assertEqual(glovo.getTransports(), {a});
                  glovo.addTransport(b);
                  assertEqual(glovo.getTransports(), {a,b});
                  glovo.removeTransport(a);
                  assertEqual(glovo.getTransports(), {b});
```

```
            glovo.addTransport(a);
            assertEqual(glovo.chooseBestTransport(sr), a);
        );

-- Tests related to Client in Glovo
public testClientGlovo: () ==> ()
        testClientGlovo() ==
        (
        dcl glovo:Glovo := new Glovo(),
            a:Client := new Client("a",5,0),
            b:Client := new Client("b", 2, 0);

        assertEqual(glovo.getClients(), {});
        glovo.addClient(a);
        assertEqual(glovo.getClients(), {a});
        glovo.addClient(b);
        assertEqual(glovo.getClients(), {a,b});
        glovo.removeClient(a);
        assertEqual(glovo.getClients(), {b});

        glovo.addClient(a);
        assertEqual(glovo.getClientByID(1), nil); --static
        assertEqual(glovo.getClientByID(a.getID()), a);
        assertEqual(glovo.getClientByID(b.getID()), b);
        assertEqual(glovo.getClientByName("a"), {a});
    );

-- Tests related to Supplier in Glovo
public testSupplierGlovo: () ==> ()
        testSupplierGlovo() ==
```

```
            (dcl glovo:Glovo := new Glovo(),
                  sr1:Supplier := new Supplier("1", 0, 0),
                  sr2:Supplier := new Supplier("2", 0, 0);

            assertEqual(glovo.getSuppliers(), {|->});
            glovo.addSupplier(sr1);
            assertEqual(glovo.getSuppliers(), {sr1.getName() |-> sr1});
            glovo.addSupplier(sr2);
            assertEqual(glovo.getSuppliers(), {sr1.getName() |->
sr1,sr2.getName() |-> sr2});

            assertEqual(glovo.getSupplierByName("1"), sr1); -- Requirement 11
            assertEqual(glovo.getSupplierByName("2"), sr2);

            glovo.removeSupplier("1");
            assertTrue("1" not in set dom glovo.getSuppliers());
);

-- Tests related to Delivery in Glovo
public testDeliveryGlovo: () ==> ()
            testDeliveryGlovo() ==
            (
            dcl glovo:Glovo := new Glovo(),
                  sr1:Supplier := new Supplier("a",5,0),
                  sr2:Supplier := new Supplier("b", 2, 0),
                  sr3:Supplier := new Supplier("c", 3, 0),
                  pr1:Product := new Product("burguer", 2),
                  cli:Client := new Client("a",0,0),
                  deli:Delivery := new Delivery(cli.getID(),{pr1},nil,sr2),
                  tr1:Transport := new Transport("mota",0 ,0);
```

```
		glovo.addClient(cli);
		assertEqual(glovo.getSuppliers(), {|->});
		glovo.addSupplier(sr1);
		assertEqual(glovo.getSuppliers(), {sr1.getName() |-> sr1});
		glovo.addSupplier(sr2);
		assertEqual(glovo.getSuppliers(), {sr1.getName() |->
sr1,sr2.getName() |-> sr2});

		assertEqual(glovo.getSupplierByName("a"), sr1);
		assertEqual(glovo.getSupplierByName("b"), sr2);

		glovo.removeSupplier("a");
		assertTrue("a" not in set dom glovo.getSuppliers());

		assertEqual(glovo.orders, {});
		glovo.addDelivery(deli);
		assertEqual(glovo.getDeliveryToDo(), {deli});
		assertEqual(glovo.getDeliveryDone(), {});
		assertEqual(glovo.getDeliveryDoing(), {});
		assertEqual(glovo.getClientDeliveries(cli), {deli}); -- Requirement
12

		for all order in set glovo.orders
			do order.setTransport(tr1);

		assertEqual(glovo.getDeliveryDoing(), {deli});
		for all order in set glovo.orders
			do order.update(); --Supplier status
		glovo.removeSupplier("b");
		for all order in set glovo.orders
			do order.update(); --Client status
```

```
            assertEqual(glovo.getDeliveryDone(), {deli});
            --glovo.removeSupplier("c"); breaks pre-condition -> not in set
            glovo.addSupplier(sr3);
            assertTrue("c" in set dom glovo.getSuppliers());
            );
-- Tests related to Order in Glovo
public testOrderGlovo: () ==> ()
            testOrderGlovo() ==
            (
            ( -- Requirement 2
            dcl glovo:Glovo := new Glovo(),
                  sr1:Supplier := new Supplier("a",5,0),
                  pr1:Product := new Product("burguer", 2),
                  cli:Client := new Client("a",0,0),
                  tr1:Transport := new Transport("mota",0,0);

                  cli.credit(50);
                  glovo.addClient(cli);
                  sr1.addProduct(pr1);
                  glovo.addSupplier(sr1);
                  glovo.addTransport(tr1);-- Requirement 9
                  assertEqual(glovo.getClientDeliveries(cli), {});
                  glovo.order(cli, sr1, {pr1});
                  assertTrue(card glovo.getClientDeliveries(cli) = 1);
                  assertEqual(glovo.getTransports(), {});
                  );
                  (
                  dcl glovo:Glovo := new Glovo(),
                  sr1:Supplier := new Supplier("a",5,0),
```

```
		pr1:Product := new Product("burguer", 2),
		cli:Client := new Client("a",0,0),
		tr1:Transport := new Transport("mota",0,0);

		cli.credit(50);
		sr1.addProduct(pr1);
		glovo.addClient(cli);
		glovo.addSupplier(sr1);
		glovo.addTransport(tr1);
		assertEqual(glovo.getClientDeliveries(cli), {});
			--glovo.order(cli, sr1, {pr1});   -- case where
Glovo::line165 breaks precondition, so, not getting coverage
			--assertTrue(card glovo.getClientDeliveries(cli) = 1);
			--assertEqual(glovo.getTransports(), {});
	);
	( -- Requirement 4
		dcl glovo:Glovo := new Glovo(),
		sr1:Supplier := new Supplier("a",5,0),
		pr1:Product := new Product("burguer", 2),
		cli:Client := new Client("a",0,0);
		cli.credit(50);
		sr1.addProduct(pr1);
		glovo.addClient(cli);
		glovo.addSupplier(sr1);
		assertEqual(glovo.getClientDeliveries(cli), {});
		glovo.order(cli, sr1, {pr1});
		assertTrue(card glovo.getClientDeliveries(cli) = 1);
		for all order in set glovo.orders
			do glovo.removeOrder(order);
		assertTrue(card glovo.getClientDeliveries(cli) = 0);
	);
```

```
(-- Requirement 3
      dcl glovo:Glovo := new Glovo(),
            sr2:Supplier := new Supplier("b", 2, 0),
            pr1:Product := new Product("burguer", 2),
            pr2:Product := new Product("pizza", 20),
            cli:Client := new Client("a",0,0),
            deli:Delivery := new Delivery(cli.getID(),{pr1},nil,sr2);
      cli.credit(50);
      glovo.addClient(cli);
      glovo.addSupplier(sr2);
      assertEqual(glovo.orders, {});
      glovo.addDelivery(deli);
      assertEqual(glovo.getDeliveryToDo(), {deli});
      glovo.editOrder(deli,{pr2});
      assertEqual(deli.getProducts(), {pr2});
);
(
dcl glovo:Glovo := new Glovo(),
      sr2:Supplier := new Supplier("b", 2, 0),
      pr1:Product := new Product("burguer", 2),
      pr2:Product := new Product("pizza", 20),
      cli:Client := new Client("a",0,0),
      deli:Delivery := new Delivery(cli.getID(),{pr1},nil,sr2);


      glovo.addClient(cli);
      glovo.addSupplier(sr2);
      assertEqual(glovo.orders, {});
      glovo.addDelivery(deli);
      assertEqual(glovo.getDeliveryToDo(), {deli});
      glovo.editOrder(deli,{pr2});
```

```
            assertEqual(glovo.orders, {});
        );
        (
        dcl glovo:Glovo := new Glovo(),
            sr2:Supplier := new Supplier("b", 2, 0),
            pr1:Product := new Product("burguer", 2),
            cli:Client := new Client("a",0,0),
            tr1:Transport := new Transport("mota",0,0),
            deli:Delivery := new Delivery(cli.getID(),{pr1},tr1,sr2);
            glovo.addClient(cli);
            glovo.addSupplier(sr2);
            sr2.addProduct(pr1);
            glovo.addTransport(tr1);
            assertEqual(glovo.orders, {});
            glovo.addDelivery(deli);
            assertEqual(glovo.getDeliveryDoing(), {deli});
            glovo.meetSupplier(deli);-- Requirement 10
            assertEqual(tr1.getLocation(), sr2.getLocation());
            glovo.removeTransport(tr1);
            assertEqual(deli.status, <Supplier>);
            assertEqual(glovo.getTransports(), {});
            glovo.meetClient(deli);-- Requirement 10
            assertEqual(cli.location, tr1.getLocation());
            assertEqual(deli.status, <Client>);
            assertEqual(card glovo.getTransports(), 1);
        );
);

public testGlovoClass: () ==> ()
        testGlovoClass() ==
        (
```

```
            IO`print("testTransportGlovo -> ");
            testTransportGlovo();
            IO`println("Success");

            IO`print("testClientGlovo -> ");
            testClientGlovo();
            IO`println("Success");

            IO`print("testSupplierGlovo -> ");
            testSupplierGlovo();
            IO`println("Success");

            IO`print("testDeliveryGlovo -> ");
            testDeliveryGlovo();
            IO`println("Success");

            IO`print("testOrderGlovo -> ");
            testOrderGlovo();
            IO`println("Success");
        );

public static main: () ==> ()
main() ==
(
        dcl glovoTest: GlovoTest := new GlovoTest();
        IO`print("testLocationClass -> ");
        glovoTest.testLocationClass();
        IO`println("Success");

        IO`print("testClientClass -> ");
```

```
        glovoTest.testClientClass();
        IO`println("Success");

        IO`print("testProductClass -> ");
        glovoTest.testProductClass();
        IO`println("Success");

        IO`print("testTransportClass -> ");
        glovoTest.testTransportClass();
        IO`println("Success");

        IO`print("testSupplierClass -> ");
        glovoTest.testSupplierClass();
        IO`println("Success");

        IO`print("testDeliveryClass -> ");
        glovoTest.testDeliveryClass();
        IO`println("Success");

        IO`print("testGlovoClass -> ");
        glovoTest.testGlovoClass();
        IO`println("Success");

);
end GlovoTest
```

5. Model verification

5.1 Example of domain verification

One of the proof obligations generated by Overture is:

No.	PO Name	Type
40	Glovo`getSupplierByName	legal map application

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

```
-- Returns a list of suppliers by name

public getSupplierByName: String ==> Supplier

      getSupplierByName(name) == return suppliers(name)

  pre name in set dom suppliers;
```

In this case the proof is trivial because the quantification '**name in set dom suppliers**' assures that the map accesses only inside its domain.

Proof Obligation View:

```
(forall name:Glovo`String & ((name in set (dom suppliers)) => (name in set (dom
suppliers))))
```

5.2 Example of invariant verification

Another proof obligation generated by Overture is:

No.	PO Name	Type
31	Glovo`addClient	state invariant holds

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

```
-- Adds a client to the system
public addClient : Client ==> ()

      addClient(c) == clients := clients union {c}

      pre not exists client in set clients & Client`equals(client, c)

      post c in set clients;
```

The relevant invariant under analysis is:

```
     inv not exists c1, c2 in set clients &
            c1 <> c2 and c1.id = c2.id;
```

The pre-condition checks if there isn't any client in the set with the same ID has the new client:

```
-- Compares 2 clients by ID
public equals : Client * Client -> bool

      equals(c1, c2) ==

  c1.id = c2.id;
```

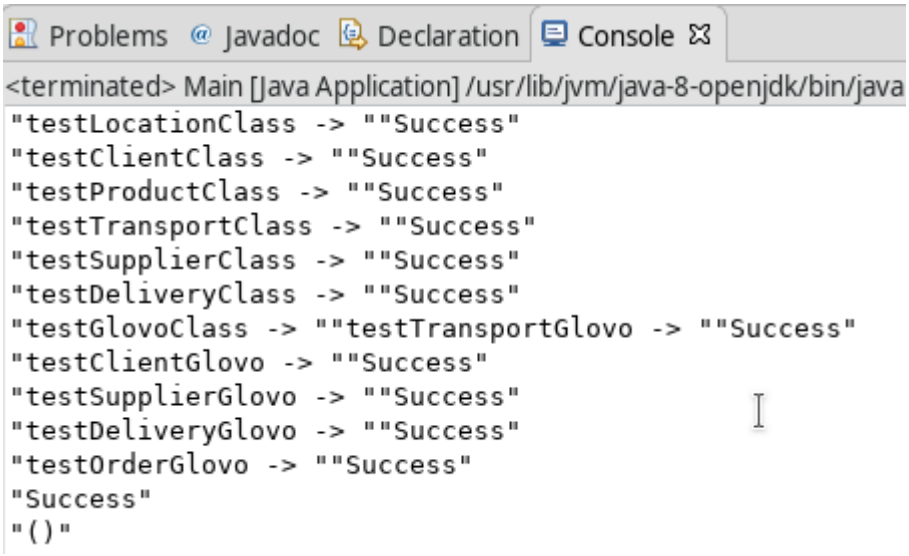
Proof Obligation View:

```
(forall c:Client & ((not (exists client in set clients & Client`equals(client,
c))) => ((not (exists c1, c2 in set clients & ((c1 <> c2) and ((c1.id) =
(c2.id))))) => (not (exists c1, c2 in set clients & ((c1 <> c2) and ((c1.id) =
(c2.id)))))))))

As sides are equivalent, the invariant is held.
```

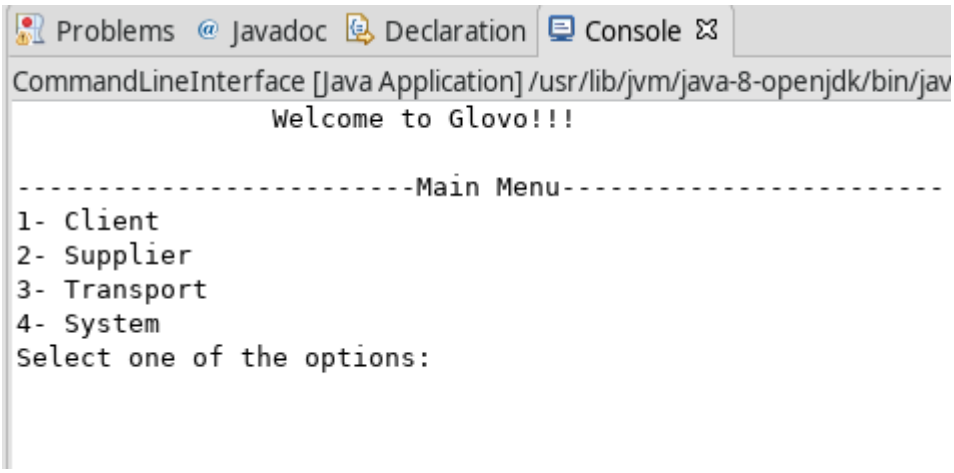
6. Code Generation

To generate Java code, in the VDM Explorer window, with the mouse over the project folder, we selected Code Generation-> Generate Java (Configurarion based). We imported the project generated in the folder “Stack/generated/java” into Eclipse, and there were some errors related to “quotes”, we fixed them and executed the generated entry point (Main.java) from Overture obtaining the following:



Which was our test suit built within Overture, and the tests were successful in Eclipse too.

We noticed that there were some problems with the generation, our pre-conditions, post-conditions and invariants had been thrown away. This was unfortunate, so we created a Command Line Interface to better the programs flow and testability.



7. Conclusions

The team was happy with its job, we fulfilled the requirements we had expected to implement in the first sections and developed a *CLI* to improve the programs readability.

In terms of knowledge, the group was excited with what can be done with the VDMTools process instead of the traditional process, we focused a lot more on Analysis & Design than usual and spent less time coding per se. The learning curve affected our start but VDM has revealed itself to be useful and usable in the future.

If time permitted, as future work, it would be useful to expand a little more and better emulate the features of the real application we tried to model. The user interface is also rudimentary, and we'd like to improve on that too.

This project took approximately 45 hours to develop.

The work was evenly distributed between the group members.

8. References

1. Glovo app web site, https://glovoapp.com
2. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
3. Overture tool web site, http://overturetool.org
4. MFES 2019, MIEIC, Moodle availabe slides