Plataforma "*Ugly Fruit*"

Métodos Formais em Engenharia de Software

Mestrado Integrado em Engenharia Informática e Computação

Trabalho efetuado por:

Bruno Monteiro Marques - up201405781

Cláudia Margarida da Rocha Marinho - up201404493

Janeiro de 2018

Descrição do Sistema e Lista de Requisitos **1**
1.1 Descrição do Sistema 1
1.2 Lista de Requisitos 2
Modelo visual do UML **4**
2.1 Diagrama de Use Cases 4
2.2 Diagrama de classes 7
Modelo formal de VDM++ **8**
3.1 Classe Basket 8
3.3 Classe Producer 8
3.4 Classe Product 8
3.5 Classe User 8
3.6 Classe PlatformUtils 8
Validação do modelo **8**
Verificação do modelo **8**
5.1 Exemplo de verificação de domínio 8
5.2 Exemplo de verificação de invariante 9
Geração do código **9**
Conclusões **9**

1. Descrição do Sistema e Lista de Requisitos

1.1 Descrição do Sistema

A plataforma Fruta Feia é uma plataforma criada no âmbito de evitar o desperdício causado pelo atual sistema de normalização de frutas e legumes que leva a que um número elevado destes produtos sejam completamente desperdiçados.

A plataforma gira à volta da interação entre três partes: os produtores, os moderadores das delegações e os consumidores. Assim sendo, esta plataforma permite aos produtores que normalmente desperdiçam estes produtos que encontrem um consumidor para os mesmos. Para facilitar tanto o acesso aos produtos como o fornecimento destes, foram criadas delegações que são locais apropriados para a recolha dos produtos e que são geridas por moderadores.

Cada uma das delegações da plataforma devem ser capazes de satisfazer os pedidos dos consumidores inscritos. Esta satisfação dos consumidores depende do fornecimento de frutas/legumes por parte dos produtores que se devem inscrever numa

delegação, indicando as frutas/legumes que vão fornecer a essa mesma delegação, além da quantidade que se comprometem a fornecer semanalmente.

Também é importante referir que um utilizador só se pode inscrever numa delegação quando é garantido que esta vai ser capaz de disponibilizar cestas ao utilizador com quantidade e variedade suficientes. Quando tal não se verifica (ou quando é atingido o limite do número de utilizadores inscritos numa delegação), o utilizador entra numa lista de espera. Quando houver vagas e capacidade de satisfação dos pedidos do consumidor por parte da delegação então o utilizador será registado na delegação pretendida e sairá da lista de espera.

Quando um utilizador se tenta registar numa delegação como um consumidor, este deve indicar o seu nome e o tamanho pretendido da cesta. Caso o registo seja bem-sucedido, o consumidor deverá levantar a sua cesta de frutas e legumes semanalmente ou, caso não possa proceder a este levantamento, enviar uma mensagem de cancelamento aos moderadores da delegação. Um utilizador também pode mudar de delegação a qualquer altura, sendo que as cestas passam a ser fornecidas no novo ponto de recolhida escolhido.

As cestas podem ser ou pequenas ou grandes. As cestas pequenas custam três euros e meio e contêm entre 3 e 4 kg de frutas/legumes de 7 variedades diferentes. Já as cestas grandes custam sete euros e contêm entre 6 e 8 kg de frutas/legumes de 8 variedades diferentes.

Por último, é importante mencionar que tanto produtores como consumidores não se podem inscrever em mais do que uma delegação simultaneamente.

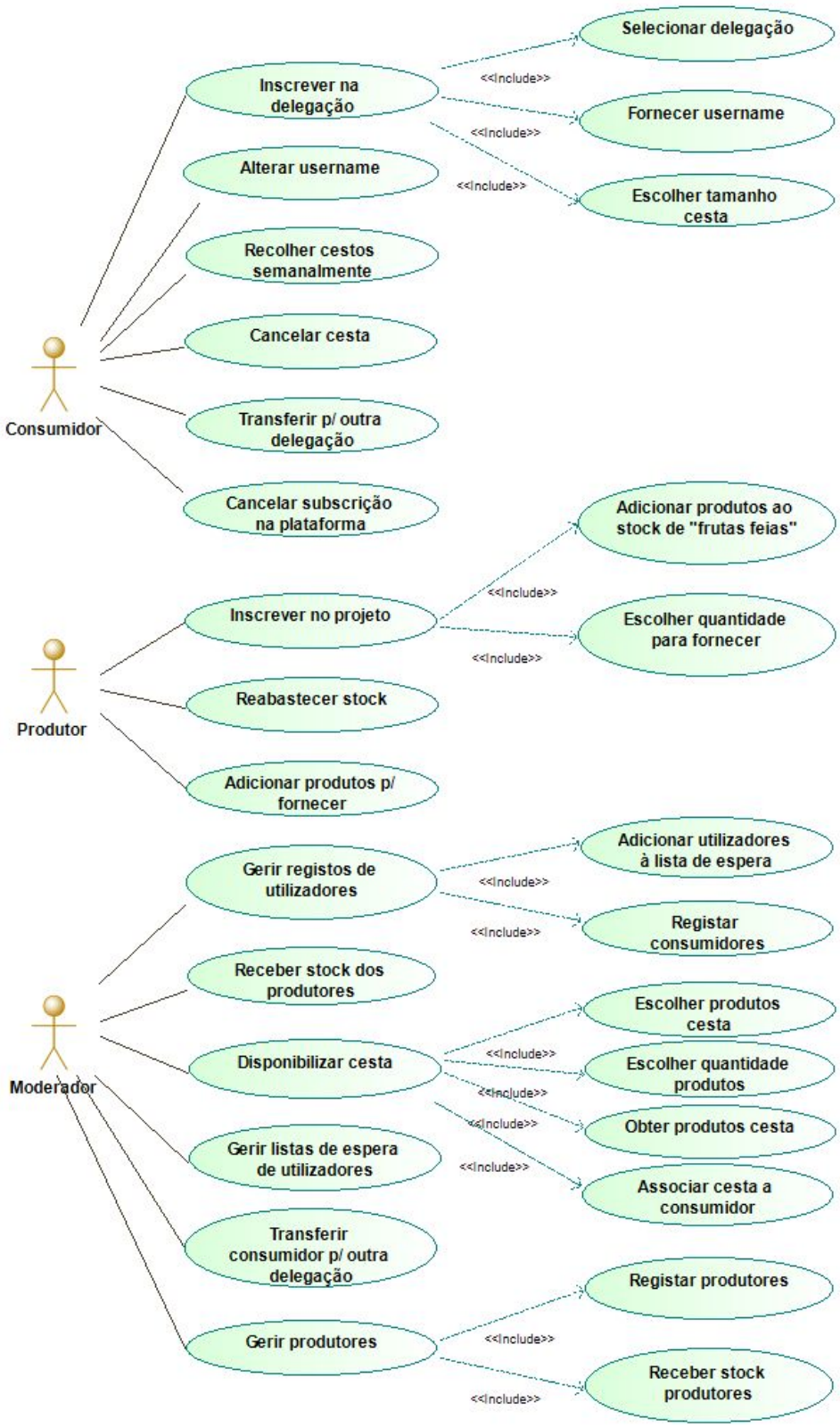
1.2 Lista de Requisitos

ID	Prioridade	Descrição
R1	Obrigatório	O consumidor deve poder registar-se numa delegação na plataforma *Fruta Feia*.
R2	Obrigatório	O consumidor deve poder escolher o tamanho da cesta a lhe ser entregue, entre os tamanhos disponíveis.
R3	Obrigatório	O consumidor deve poder informar os moderadores da sua delegação que pretende cancelar a entrega de uma cesta na semana correspondente.
R4	Obrigatório	O consumidor deve poder cancelar a sua subscrição à plataforma Fruta Feia.
R5	Obrigatório	O consumidor pode escolher mudar de delegação.
R6	Opcional	O consumidor deve poder mudar o seu username.
R7	Obrigatório	O produtor deve poder registar-se numa delegação na plataforma

		Fruta Feia.
R8	Obrigatório	O produtor deve escolher o tipo e quantidade de produtos a fornecer à sua delegação semanalmente.
R9	Obrigatório	O produtor deve ser capaz de reabastecer o seu *stock* de “frutas feias” semanalmente.
R10	Opcional	O produtor deve poder aumentar a quantidade ou variedade de produtos fornecidos à sua delegação semanalmente.
R11	Obrigatório	O moderador deve gerir os pedidos de registo de novos utilizadores, rejeitando ou aceitando estes.
R12	Obrigatório	O moderador deve gerir a lista de utilizadores em espera, inscrevendo estes utilizadores na delegação quando possível.
R14	Obrigatório	O moderador deve poder efetuar transferências de consumidores entre delegações.
R15	Obrigatório	O moderador deve criar e fornecer o número correto de cestas aos consumidores inscritos na delegação.
R16	Obrigatório	O moderador deve poder verificar se cada pedido de cesta foi cancelado ou não por parte do consumidor.
R17	Obrigatório	O moderador deve ser capaz de reabastecer a delegação com os produtos fornecidos pelos produtores.

2. Modelo visual do UML

2.1 Diagrama de *Use Cases*



Os *Use Cases* mais relevantes vão ser descritos seguidamente:

Use Case	**Inscrever utilizador numa delegação**
Descrição	Este *Use Case* descreve a capacidade de um consumidor se inscrever numa delegação à sua escolha de tal forma que este possa fazer parte do projeto.
Pré-condições	1. O utilizador não deve estar inscrito numa outra delegação. 2. O utilizador não deve estar inscrito na delegação em que se tenta registar.
Pós-condições	1. O consumidor que se tentou registar ou se encontra na lista de espera de utilizadores ou na lista de utilizadores da delegação. 2. O número máximo de utilizadores na delegação não é ultrapassado. 3. É possível satisfazer todos os pedidos de cestas dos utilizadores inscritos na delegação, em termos de peso das cestas e de variedade de produtos nestas.
Passos	1. Criar utilizador, fornecendo o nome de utilizador necessário para o registo na plataforma. 2. O utilizador escolhe o tamanho da cesta pretendida na delegação em que se pretende inscrever. 3. O utilizador seleciona a delegação para a qual se pretende inscrever. 4. Se for possível inscrever o utilizador na delegação, este é registado nessa mesma delegação como consumidor. 5. Se não for possível, o utilizador é adicionado à lista de espera dessa mesma delegação.
Exceções	5. Se o utilizador já estiver na lista de espera, nada acontece.

Use Case	**Transferir utilizador para outra delegação**
Descrição	Este *Use Case* descreve a possibilidade de transferir um utilizador da delegação em que este se encontra atualmente inscrito para outra.
Pré-condições	1. O utilizador deve estar inscrito numa delegação.
Pós-condições	1. O utilizador não se encontra mais na lista de utilizadores da delegação anterior. 2. O utilizador ou é inscrito na nova delegação ou entra na lista de espera dessa mesma delegação.
Passos	1. Remover utilizador da delegação em que este se encontra inscrito atualmente. 2. Tentar registar o utilizador numa nova delegação.

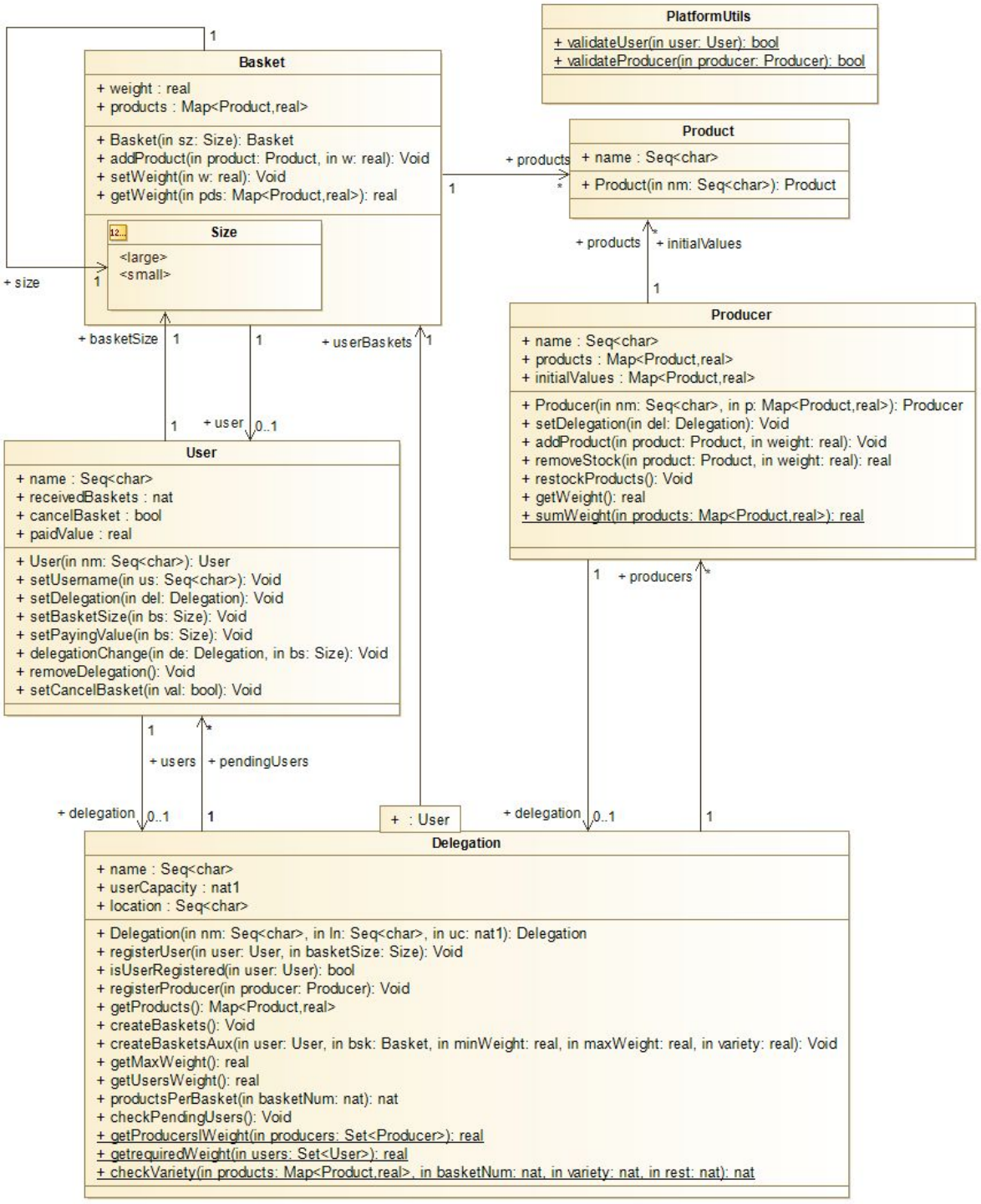
	3. Se registo foi bem-sucedido, o utilizador é adicionado à lista de consumidores da delegação, senão é adicionado à lista de espera.
Exceções	(nenhumas)

Use Case	**Inscrever produtor numa delegação**
Descrição	Este *Use Case* descreve a capacidade de um produtor se inscrever numa delegação à sua escolha de tal forma que este possa fazer parte do projeto.
Pré-condições	1. O produtor não deve estar inscrito numa outra delegação. 2. O produtor não deve estar inscrito na delegação em que se tenta registar.
Pós-condições	1. A lista de produtores inscritos na delegação não deve estar vazia.
Passos	1. Criar produtor, indicando o seu nome. 2. O produtor deve indicar os produtos e respetivas quantidades que este está disposto a fornecer semanalmente à delegação em que se vai inscrever. 3. Registar o produtor na delegação. 4. Percorrer lista de espera e verificar se é possível registar os utilizadores nessa lista.
Exceções	(nenhumas)

Use Case	**Criar cestas para consumidores**
Descrição	Este *Use Case* descreve como o moderador de uma delegação pode criar cestas para todos os utilizadores inscritos nessa mesma delegação.
Pré-condições	1. A lista de consumidores inscritos na delegação não pode estar vazia. 2. A lista de produtores registados na delegação não pode estar vazia. 3. É possível satisfazer todos os pedidos de cestas dos utilizadores inscritos na delegação, em termos de peso das cestas e de variedade de produtos nestas.
Pós-condições	1. A lista de cestas criadas nessa semana não pode estar vazia.
Passos	1. Obter o tamanho da cesta de cada um dos utilizadores inscritos na delegação. 2. Verificar se o consumidor cancelou a recolha da cesta e caso tenha, não criar cesta para esse utilizador. 3. Obter pequenas quantidades de produtos dos produtores inscritos na delegação até que a quantidade mínima e variedade requerida

	para a cesta seja atingida. 4. Criar cestas válidas e associá-las a cada um dos utilizadores que não cancelaram os pedidos semanais das cestas. 5. Reabastecer stock com produtos fornecidos pelos produtores.
Exceções	(nenhumas)

2.2 Diagrama de classes



Classe	Descrição
Basket	Representa uma cesta de produtos que vai ser entregue a um utilizador.
Delegation	Representa uma delegação em que utilizadores e produtores se podem inscrever ; as cestas também são criadas aqui.
Producer	Representa um produtor que vai fornecer os produtos a delegações.
Product	Representa um produto que vai compor as cestas.
User	Representa um utilizador que se vai tornar num consumidor quando se registar numa delegação.
PlatformUtils	Valida utilizadores e produtores de forma a garantir que tanto os utilizadores e os produtores só se registam numa delegação.

3. Modelo formal de VDM++

3.1 Classe *Basket*

A classe basket é a classe representativa da cesta de legumes e frutas a ser entregue semanalmente aos utilizadores da plataforma, o seu peso e lista de produtos nela contida são variáveis importantes pois são processadas cuidadosamente pela lógica do programa de forma a respeitar as condições da plataforma (o número de legumes/ frutas diferentes e a sua quantidade). As funções da mesma prendem-se principalmente à volta do conteúdo da cesta.

```
class Basket
types
	public Size = <small> | <large>;
	public Weight = real;
	public ProductSet = map Product to Weight;
instance variables
	public size: Size;
	public weight : Weight := 0;
	public products: ProductSet := { |-> };
	public user: [User] := nil;
operations
	--Constructor of class Basket
	--@params sz -> size of basket either small or large
	public Basket : Size ==> Basket
		Basket(sz)==(
			size := sz;
			return self;
		)
		post size = sz;

	--Operation responsible for adding a product to a Basket
	--@params product -> product to add to basket
	--@params w -> weight of product in Kg to add to basket
	public addProduct : Product*real ==> ()
		addProduct(product, w)==(
			dcl pds : ProductSet := products;

			if(product not in set dom products)
			then pds := pds munion {product |-> w}
			else pds(product) := pds(product) + w;

			if (size = <small>)
			then (
				if (getWeight(pds) <= 4)
				then (
					if(product in set dom products)
					then products(product) := products(product) + w
					else products := pds;
				);
			) elseif (size = <large>)
			then (
				if (getWeight(pds) <= 8)
```

```
                        then(
                                if(product in set dom products)
                                then products(product) := products(product) + w
                                else products := pds;
                        );
                );

                setWeight(getWeight(products));
        )
        post products <> { |-> };

    --Operation responsible for setting the weight of a basket
    --@params w -> weight of basket in Kg
        public setWeight : real ==> ()
        setWeight(w) == (
                weight := w;
                )
                post weight = w;

    --Operation responsible for getting the weight of a set of products
    --@params pds -> set of products whose weight should be calculated
    --@return real -> weight of product set in Kg
        public getWeight : ProductSet ==> real
                getWeight(pds)==(
                        dcl tmp : real := 0;

                        for all p in set dom pds do (
                                tmp := tmp + pds(p);
                        );

                        return tmp;
                )
                pre pds <> { |-> };
end Basket
```

3.2 Classe *Delegation*

A classe delegation é a classe que contém a lógica de operação da plataforma, representa o local de recolha das cestas por parte dos utilizadores assim como agrega nela os produtores e utilizadores do website de forma organizada podendo desta forma controlar stocks e criação de cestas corretamente.

Contém também a função createBaskets() que é a responsável pela criação das cestas para os utilizadores.

Esta função percorre a lista de utilizadores na delegação que pretendem receber uma cesta semanal, e até atingir o intervalo de peso aceitável na cesta irá a cada produtor buscar uma quantidade pré determinada dos seus produtos normalmente 0.5Kg e adicionar à cesta até que se verifica a condição da variedade de produtos e peso.

É também de notar que

```
class Delegation
types
        public DelegationName = seq1 of char;
        public LocationName = seq1 of char;
        public UserSet = set of User;
        public Weight = real;
        public ProductSet = map Product to Weight;
        public ProducerSet = set of Producer;
        public UserBaskets = map User to Basket;

instance variables
        public name: DelegationName;
        public userCapacity: nat1;
        public location: LocationName;
        public users: UserSet := {};
        public producers: ProducerSet := {};
        public pendingUsers: UserSet := {};
        public userBaskets: UserBaskets := { |-> };
        inv getUsersWeight() <= getMaxWeight();
        inv card users <= userCapacity;
        inv card users > 0 => card producers > 0;

operations
        --Constructor of class Delegation
        --@params nm -> Name of delegation
        --@params ln -> Name of location
        --@params uc -> Physical capatiy of delegation to support users
        public Delegation : DelegationName*LocationName*nat1 ==> Delegation
                Delegation(nm,ln,uc)==(
                        name := nm;
                        location := ln;
                        userCapacity := uc;
                        return self;
                )
                post name = nm and location = ln and userCapacity = uc;


        --Operation responsible for registering a user to the delegation
```

```
        --@params user -> User object to add to the delegation
        --@params basketSize -> Size of the basket small or large

        public registerUser : User*Basket`Size ==> ()
                registerUser(user, basketSize)==(
                        dcl tmp : User := new User("tmp");
                        tmp.setBasketSize(<large>);
        if ((card users >= userCapacity) or (getrequiredWeight(users union {tmp}) >
        getMaxWeight()) or productsPerBasket(card users + 1) < 8)
                then (
        If user not in set pendingUsers
                then (
                user.setBasketSize(basketSize);
                pendingUsers := pendingUsers union {user};

                IO`print("Pending user ");
                IO`println(user.name);
                );

        )
        else (
        if(user in set pendingUsers)
        then pendingUsers := pendingUsers \ {user};
                user.setBasketSize(basketSize);
                user.setDelegation(self);
                users := users union {user};
                IO`print("Registered user ");
                IO`print(user.name);
                IO`print(" on delegation ");
                IO`println(self.name);
                        );
                )
        pre PlatformUtils`validateUser(user) and user not in set users
        post card users <= userCapacity and (user in set users or user in set
pendingUsers) and getUsersWeight() <= getMaxWeight() and if card users > 0 then
(card producers > 0 and productsPerBasket(card users) >= 8) else card producers
>= 0;

        --Operation responsible for removing a user from the delegation
        --@params user -> User object to remove from the delegation

        public removeUser : User ==> ()
        removeUser(user)==(
         user.removeDelegation();
         checkPendingUsers();
        users := users \ {user}
        )
        pre users <> {} and user in set users
        post user not in set users;

        --Operation responsible for registering a producer to the delegation
        --@params producer -> producer object to add to the delegation
        public registerProducer : Producer ==> ()
                registerProducer(producer)==(
                        producer.setDelegation(self);
                        producers := producers union {producer};
```

```
                        IO`print("Registered producer ");
                        IO`print(producer.name);
                        IO`print(" on delegation ");
                        IO`println(self.name);

                        checkPendingUsers();
                )
        pre producer not in set producers and
PlatformUtils`validateProducer(producer)
        post producers <> {};

        --Operation responsible for getting the products from all the producers in the
delegation
        public pure getProducts : () ==> ProductSet
        getProducts()==(
                dcl P : ProductSet := { |-> };
        for all producer in set producers do (
                for all product in set dom producer.products do (
                if product in set dom P
                then P(product) := P(product) + producer.products(product)
                else P := P munion { product |-> producer.products(product)};
                        );
                );
        return P;
        );

        --Operation responsible for creating the baskets to deliver to each user
registered in the delegation
        public createBaskets : () ==> ()
        createBaskets()==(
        for all user in set users do (
                dcl bsk : Basket := new Basket(user.basketSize);
                dcl minWeight : real := 0;
                dcl maxWeight : real := 0;
                dcl variety : real := 0;

        if (user.basketSize = <small>)
                then (minWeight := 3; maxWeight := 4; variety := 7)
                else (minWeight := 6; maxWeight := 8; variety := 8);
                        if user.cancelBasket = false
                        then createBasketsAux(user, bsk, minWeight, maxWeight,
variety)
                        else user.setCancelBasket(false);
                        );

        if ((getrequiredWeight(users) < getMaxWeight()) and productsPerBasket(card
users) >= 8)
                then   IO`println("No need to restock")
                else (IO`println("Restocking delegation stock"); restock())
                )
        pre users <> {} and producers <> {} and getUsersWeight() <=
getMaxWeight() and productsPerBasket(card users) >= 8
        post userBaskets <> { |-> };

        --Auxiliary operation to createBasket
        --@params user -> Target user of basket
        --@params bsk -> Basket to deliver
```

```
        --@params minWeight -> Minimum weight for basket
        --@params maxWeight -> Maxinmum weight for basket
        --@params variety -> Required number of different products

public createBasketsAux : User*Basket*real*real*real ==> ()
        createBasketsAux(user, bsk, minWeight, maxWeight, variety)==(
                dcl v : real := 0;
                dcl variedProducts : set of Product := {};
                dcl varietyAchieved : bool := false;

                while(bsk.weight < minWeight) do (
                for all producer in set producers do (
                        for all product in set dom producer.products do (
                        dcl w : real := 0.5;
                        dcl deletedWeight : real := 0;

                        if (product not in set variedProducts and v < variety)
                        then (
                                variedProducts := variedProducts union {product};
                                v := v + 1;
                        );

                        if(product in set variedProducts)
                        then (
                                deletedWeight := producer.removeStock(product, w);
                                if (deletedWeight > 0)
                                then (
                                        bsk.addProduct(product, deletedWeight)
                                );
                        );

                        if(v >= variety)
                        then (varietyAchieved := true;);

                        if(bsk.weight >= minWeight and bsk.weight <= maxWeight and
varietyAchieved = true)
                        then (
                                if (user in set dom userBaskets)
                                then userBaskets(user) := bsk
                                else userBaskets := userBaskets munion {user |->
bsk};

                                if user.basketSize = <small>
                                then IO`print("Small ")
                                else IO`print("Large ");

                                IO`print("basket created for user ");
                                IO`print(user.name);
                                IO`print(" on delegation ");
                                IO`println(self.name);

                                IO`println("Basket contents: ");
                                IO`println(bsk);
                                IO`println("");

                                return
                        );
```

```
                        );
                );
                );
        )
        pre minWeight > 0 and maxWeight > 0 and variety > 0;

        --Operation responsible for restocking the products of the delegation after the
baskets were create (week passed)
        public restock : () ==> ()
        restock() ==(
                        for all producer in set producers do(
                                producer.restockProducts();
                                );
                );
        --Operation responsible for getting the maximum weight of products to sustain
the user base
        public pure getMaxWeight : () ==> real
        getMaxWeight()==
                getProducerslWeight(producers);

        --Operation responsible for getting the weight of products to satisfy the current
user base
        public pure getUsersWeight : () ==> real
        getUsersWeight()==
                getrequiredWeight(users);

        --Operation responsible for getting the weight of products to satisfy the current
user base
        --@params basketNum -> Number of baskets to create
        public pure productsPerBasket: nat ==> nat
                productsPerBasket(basketNum)==(
                        checkVariety(getProducts(), basketNum, 0);
                )
                pre basketNum > 0;

        --Operation responsible for trying to register users in the pending user list

        public checkPendingUsers: () ==> ()
                checkPendingUsers()==(
                        for all user in set pendingUsers do (
                                registerUser(user, user.basketSize);
                        );
                );
functions
        --Operation responsible for getting the weight of all the producer's in the
delegation
        --@params producers -> Producers in delegation
        public getProducerslWeight : ProducerSet -> real
        getProducerslWeight(producers)==(
         if producers = {} then 0
                else let p in set producers in p.getWeight() +
getProducerslWeight(producers\{p} )
                );

        --Operation responsible for getting the required weight to sustain all the users
in the delegation
```

```
        --@params users -> Users in delegation

        public getrequiredWeight : UserSet -> real
        getrequiredWeight(users)==(
         if users = {} then 0.0
                else let u in set users in
                if u.basketSize = <small> then 4.0 + getrequiredWeight(users\{u})
                else 8 + getrequiredWeight(users\{u})
                );









        --Operation responsible for calculating the variety of products in the required
baskets
        --@params products -> Products in delegation
        --@params basketNum -> Number of baskets required
        --@params variety -> Number of diferent products per basket

        --@return variety -> Number of diferent products per basket

        public checkVariety : ProductSet*nat*nat -> nat
                checkVariety(products, basketNum, variety)==(
                 if products = {|->} then variety
                        else let p in set dom products in
                        if products(p) >= basketNum then checkVariety({p} <-:
products, basketNum, variety + 1)
                        else checkVariety({p} <-: products, basketNum, variety)
                        )
                        pre basketNum > 0;

end Delegation
```

3.3 Classe *Producer*

```
class Producer
types
	public ProducerName = seq1 of char;
	public Weight = real;
	public ProductSet = map Product to Weight;

instance variables
	public name: ProducerName;
	public products: ProductSet := { |-> };
	public initialValues : ProductSet := { |-> };
	public delegation:[Delegation] := nil;

operations

	--Constructor of class Producer
	--@params nm -> name of producer
	public Producer : ProducerName*ProductSet ==> Producer
	Producer(nm,p)==(
		name := nm;
		products := p;
		initialValues := p;
		return self;
	)
	post name = nm;

	--Operation responsible for setting a delegation to the producer
	--@params del -> delegation object producer is joining
		public setDelegation : Delegation ==> ()
		setDelegation(del)==(
			delegation := del;
		)
		post delegation = del;

	--Operation that allows to remove a delegation from a producer
	 public removeDelegation: () ==> ()
	 removeDelegation() ==(
		delegation := nil;
	 )
	 pre delegation <> nil
	 post delegation = nil;
```

```
        --Operation responsible for adding a product to a producer's stock
        --@params product -> product to add to producer stock
        --@params weight -> weight of product in Kg to add to stock
        public addProduct : Product*real ==> ()
        addProduct(product, weight)==(
                if product in set dom products
                then (
                        products(product) := products(product) + weight;
                        initialValues(product) := initialValues(product) + weight)
                else (
                        products := products munion { product |-> weight };
                        initialValues := initialValues munion { product |-> weight }
                );

                if (delegation <> nil)
                then delegation.checkPendingUsers();
        )
        pre weight >= 0.5
        post product in set dom products;

        --Operation responsible for removing a product from producer's stock
        --@params product -> product to remove from producer's stock
        --@params weight -> weight of product in Kg to remove
        public removeStock : Product*real ==> real
        removeStock(product,weight)==(
                dcl ret : Weight := 0;
                if (product not in set dom products)
                then return ret
                else
                        if ((products(product)-weight) < 0)
                        then ((ret := products(product)); (products(product):=0);
return ret)
                        else ((products(product):= (products(product) - weight));
 return weight)
        )
        pre weight > 0 and products <> { |-> };

        --Operation responsible for restocking the products on a producer's stock after
the baskets were create (week passed)
        public restockProducts : () ==> ()
                restockProducts()==(
                        products := initialValues;
                );

        --Operation responsible for getting the weight of the producer's stock
        public pure getWeight : () ==> real
        getWeight()==
                sumWeight(products);


functions
        --Function responsible for adding the weight of the producer's stock
        --@params products -> set of products whose weight should be calculated
        --@return real -> weight of product set in Kg
        public sumWeight : ProductSet -> real
        sumWeight(products)==(
```

```
        if products = { |-> } then 0
                else let p in set dom products in products(p) + sumWeight({p} <-:
products)
        );

end Producer
```

3.4 Classe *Product*

```
class Product
types
        public ProductName = seq1 of char;

instance variables
        public name: ProductName;

Operations

        --Constructor of class Product
        --@params nm -> name of product
        public Product : ProductName ==> Product
        Product(nm)==(
                name := nm;
                return self;
        )
        post name = nm;

end Product
```

3.5 Classe *User*

```
class User

types
	public Username = seq1 of char;

instance variables
	public name: Username;
	public delegation: [Delegation] := nil;
	public basketSize: Basket`Size;
	public receivedBaskets: nat := 0;
	public cancelBasket: bool := false;
	public paidValue: real := 0;

operations

	--Constructor of class User
	--@params nm -> name of user
	public User : Username ==> User
		User(nm)==(
			name := nm;
			return self;
		)
		pre len nm <= 15   -- username nao pode ter mais que 15 caracteres
		post name = nm;

	--Operation responsible for changing the username
	--@params us -> name of user
	public setUsername : Username ==> ()
		setUsername(us)==(
			name := us;
		)
		pre len us <= 15
		post name = us;

	--Operation responsible for setting a delegation to the user
	--@params del -> delegation object user is joining
	public setDelegation : Delegation ==> ()
		setDelegation(del)==(
			delegation := del;
		)
		post delegation = del;


	--Operation responsible for setting the size for the user's basket
```

```
        --@params bs -> Size of the basket small or large

        public setBasketSize : Basket`Size ==> ()
                setBasketSize(bs)==(
                        basketSize := bs;
                )
                pre bs = <small> or bs = <large>
                post basketSize = bs;

        --Operation updating the value user paid for the baskets
        --@params bs -> Size of the basket small or large
        public setPayingValue : Basket`Size ==> ()
                setPayingValue(bs)==(
                receivedBaskets:= receivedBaskets + 1;
                if(bs = <small>) then paidValue := paidValue + 3.5
                else paidValue := paidValue + 7
                )
                pre bs = <small> or bs = <large>;

        --Operation that allows for the user to switch delegations
        --@params d3 -> New delegation object
        --@params bs -> Size of the basket small or large

        public delegationChange : Delegation*Basket`Size ==> ()
                delegationChange(de,bs) ==(
                delegation.removeUser(self);
                de.registerUser(self, bs);
         )
         pre delegation <> nil;

        --Operation that allows to remove a delegation from a user
         public removeDelegation : () ==> ()
         removeDelegation() ==(
                delegation := nil;
         )
         pre delegation <> nil
         post delegation = nil;


        --Operation that updates the cancel basket option for the user
         public setCancelBasket : bool ==> ()
         setCancelBasket(val)==(
                cancelBasket := val;
         )
         post cancelBasket = val;
end User
```

3.6 Classe *PlatformUtils*

```
class PlatformUtils
types

functions
        --Function that verifies if user already belongs to a delegation
        --@params user -> User to add to delegation
        public validateUser : User -> bool
        validateUser(user)==(
         if (user.delegation = nil) then true
                else false
                );

        --Function that verifies if producer already belongs to a delegation
        --@params producer -> producer to add to delegation
        public validateProducer : Producer -> bool
        validateProducer(producer)==(
         if (producer.delegation = nil) then true
                else false
                );
end PlatformUtils
```

4. Validação do modelo

5. Verificação do modelo

5.1 Exemplo de verificação de domínio

Duas das *proof obligations* geradas pelo Overture é:

4	Basket`addProduct(Product, real)	legal map application
5	Basket`addProduct(Product, real)	legal map application

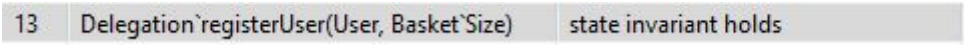
O código em análise (com os *map application* relevantes sublinhados) é:

```
public addProduct : Product*real ==> ()
        addProduct(product, w)==(
                dcl pds : ProductSet := products;

                if(product not in set dom products)
                then pds := pds munion {product |-> w}
                else pds(product) := pds(product) + w;

                if (size = <small>)
                then (
                        if (getWeight(pds) <= 4)
                        then (
                                if(product in set dom products)
                                then products(product) := products(product) + w
                                else products := pds;
                        );
                ) elseif (size = <large>)
                then (
                        if (getWeight(pds) <= 8)
                        then(
                                if(product in set dom products)
                                then products(product) := products(product) + w
                                else products := pds;
                        );
                );

                setWeight(getWeight(products));
        )
        post products <> { |-> };
```

A condição '**product in set dom products**' assegura que tanto no primeiro como no segundo exemplo, o mapa *products* é acedido apenas dentro do seu domínio, sendo por isso a prova trivial.

5.2 Exemplo de verificação de invariante

Outra *proof obligation* gerada pelo Overture é:



O código em análise é:

```
public registerUser : User*Basket`Size ==> ()
        registerUser(user, basketSize)==(
                dcl tmp : User := new User("tmp");
                tmp.setBasketSize(<large>);
if ((card users >= userCapacity) or (getrequiredWeight(users union {tmp}) >
getMaxWeight()) or productsPerBasket(card users + 1) < 8)
        then (
If user not in set pendingUsers
        then (
        user.setBasketSize(basketSize);
        pendingUsers := pendingUsers union {user};

        IO`print("Pending user ");
        IO`println(user.name);
        );

)
else (
if(user in set pendingUsers)
then pendingUsers := pendingUsers \ {user};
        user.setBasketSize(basketSize);
        user.setDelegation(self);
        users := users union {user};
        IO`print("Registered user ");
        IO`print(user.name);
        IO`print(" on delegation ");
        IO`println(self.name);
                );
        )
        pre PlatformUtils`validateUser(user) and user not in set users
        post card users <= userCapacity and (user in set users or user in set
pendingUsers) and getUsersWeight() <= getMaxWeight() and if card users > 0 then
(card producers > 0 and productsPerBasket(card users) >= 8) else card producers >=
0;
```

Os enxertos de código selecionados são a linha em que a *proof obligation* se incide principalmente e algumas das condições que se devem verificar após executar registerUser e que dependem do registo (ou não-registo) do utilizador que tentou-se inscrever na delegação. Com isto tudo, pretende-se verificar que as seguintes invariantes verificam-se após a execução do código:

```
inv card users <= userCapacity;
inv getUsersWeight() <= getMaxWeight();
inv card users > 0 => card producers > 0;
```

Tem-se de provar que a invariante se verifica após a execução de registerUser e assim sendo deve-se verificar que:

(card users <= userCapacity

and getUsersWeight() <= getMaxWeight()
and if card users > 0 then card producers > 0)
=>
(card users <= userCapacity
and getUsersWeight() <= getMaxWeight()
and card users > 0 => card producers > 0)

A primeira e segunda pós-condições obviamente implicam a primeira e segunda invariante (já que são exatamente iguais), por isso pode-se simplificar a prova significativamente na seguinte forma:

(if card users > 0 then card producers > 0) => (card users > 0 => card producers > 0)

Esta prova pode ser reescrita como:

(if card users > 0 then card producers > 0) => (if card users > 0 then card producers > 0)

O que é verdadeiro, já que tanto a pós-condição como a invariantes são exatamente iguais.

6. Geração do código

Para que o código Java seja gerado com sucesso é necessário que a definição das operações esteja no formato **op: A * B * … ==> R op(a,b,…) == bodystmt** e também que a definição das funções esteja no formato **f: A * B * … * Z -> R1 * R2 * … * Rn f(a,b,…,z) == bodyexpr**. Note-se que tanto apesar de as pré e pós-condições não estarem presentes nestes dois formatos, também se pode especificá-las (apesar de estas serem ignoradas pelo Java na prática). Caso as operações/funções sejam especificadas no outro formato, o Overture não vai conseguir traduzir o conteúdo delas, não sendo então possível testar o código em Java.

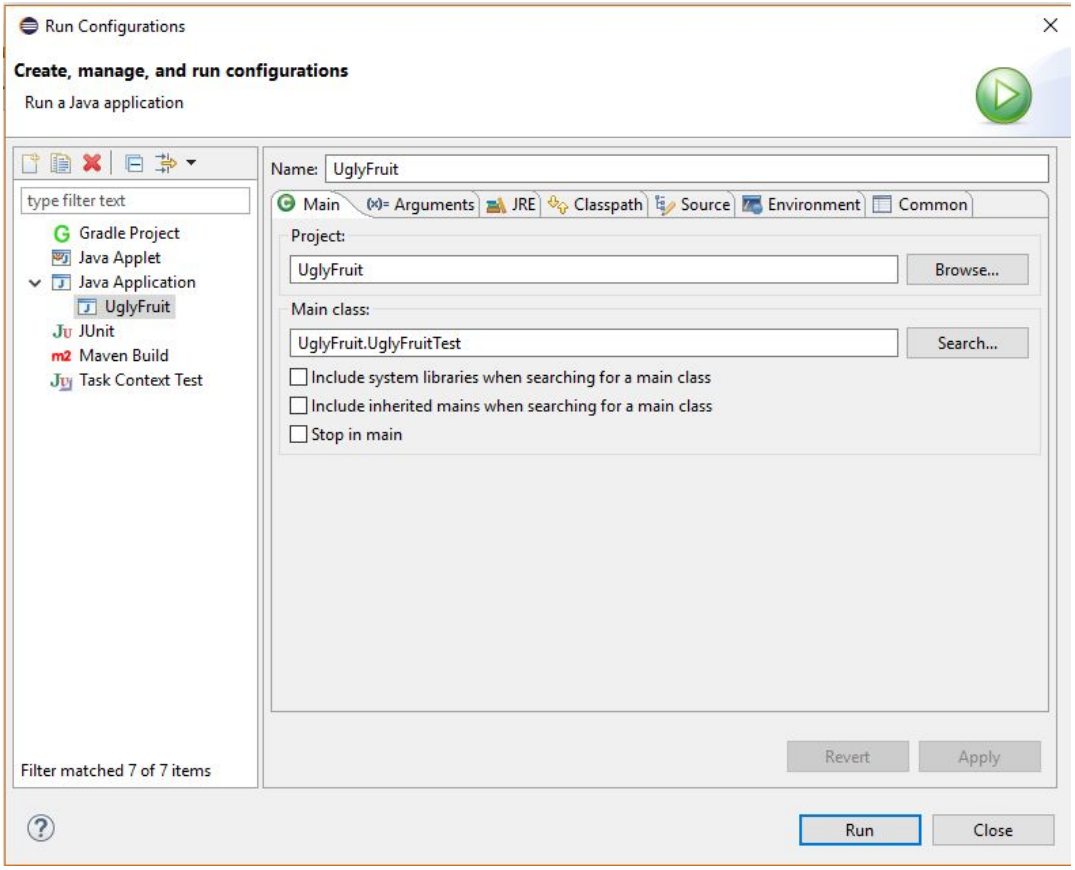
Para gerar o código em Java, basta só clicar na pasta do projeto e escolher a opção **Code Generation** > **Generate Java**. Isto irá criar uma pasta chamada “java” dentro de uma pasta chamada “generated” (que por sua vez se encontra na pasta do projeto). Dentro dessa pasta, vai-se encontrar um projeto java com os ficheiros com o código fonte, as livrarias e as configurações do projeto.

Para abrir esse projeto no eclipse, basta só abrir o IDE e ir a **File** > **Open Projects from File System…** e depois clica-se em **Directory…** no campo **Project Source** e escolhe-se a pasta de código java que foi gerada pelo Overture (carregando depois no botão **Finish**).

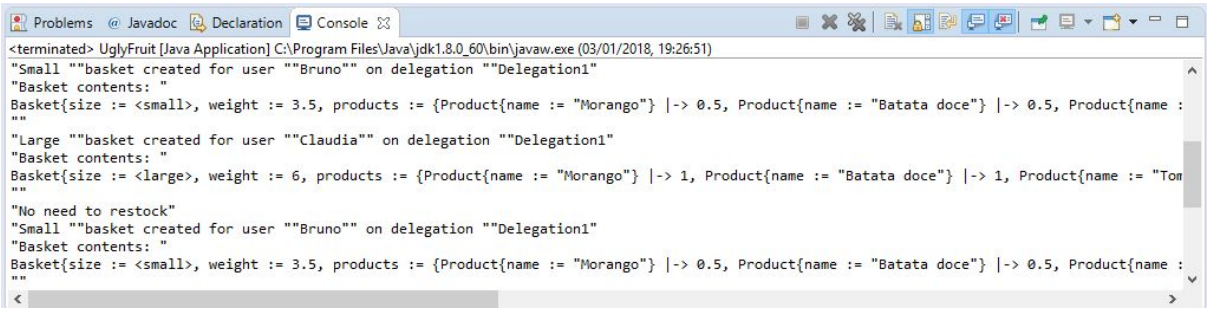
Para correr o projeto, é preciso fazer um pequeno ajuste no ficheiro **UglyFruitTest.java** que se encontra no package **UglyFruit** que é acrescentar um argumento (String[] args) ao main nesse ficheiro. O main deve estar definido na seguinte forma: **public static void main(String[] args)**. Depois deste ajuste, já é possível correr o projeto. Para isso, só basta clicar na pasta do projeto e escolher a opção **Run As** > **Run**

Configurations… e na janela que abriu, deve-se criar uma instância em **Java Application**. Depois de criar a instância, deve-se mudar o nome desta no campo **Name** para **UglyFruit** e deve-se especificar a main class que vai ser **UglyFruit.UglyFruitTest** no nosso projeto.

Depois de aplicar estas mudanças (clicando em **Apply**), a janela de configurações da instância para correr o programa deve ser semelhante a:



Depois basta só clicar no botão “Run” para correr o projeto. Deve-se conseguir correr o projeto sem problemas, sendo possível verificar isto analisando o que foi imprimido na consola. Uma parte do que é imprimido na consola é:



O único “problema” que se verificou ao correr o projeto é que algumas das impressões são menos legíveis, devido ao excesso de aspas. Mesmo assim, isto é um problema insignificante, podendo-se chegar à conclusão que o código java pode ser gerado e testado com sucesso para o nosso projeto.

7. Conclusões

Pensámos que com este projeto conseguimos modelar a plataforma *UglyFruit* com sucesso, já que implementamos as funcionalidades principais que a caracterizam. Também tivemos o devido cuidado em restringir as várias operações e funções implementadas de tal modo que estas fossem modeladas de uma forma que faz sentido no âmbito do projeto *UglyFruit*. Também conseguimos obter coverage máxima com os testes que efetuamos. Esta abundância de testes foi muito útil pois ajudou-nos a detetar algumas inconsistências na modelação de algumas utilidades que foram subsequentemente corrigidos.

Quanto ao que pode ser melhorado no nosso projeto é que talvez devíamos ter criado uma classe plataforma para tornar os testes ainda mais fáceis (apesar de já serem fáceis no estado atual do projeto).

No que toca à distribuição de trabalho esta foi 50/50, já que a maioria do trabalho foi efetuado conjuntamente.

8. Referências