```
-- A client of a Kid2Kid store.
-- A client may as well be a supplier of products.
class Client
types
	public NotEmptyString = seq1 of char;
values
instance variables
	static idCounter: nat := 0;
	id: nat1;
	name: NotEmptyString;
	boughtProducts: set of Product := {};
	soldProducts: set of Product := {};
	boughtGiftCards: set of GiftCard := {};
operations
	public Client: NotEmptyString ==> Client
	Client(nameSeq) ==
			(
			name:=nameSeq;
			idCounter := idCounter + 1;
			id := idCounter;
			return self)
			 pre len nameSeq > 0
			 post len name > 0 and
			 id = idCounter~ + 1 and
			 boughtProducts = {} and
			 soldProducts = {} and
			 boughtGiftCards = {} and
			 RESULT = self;

	-- Update name of the client
	public setName: NotEmptyString ==> ()
		setName(newName) ==
			name := newName
			pre true
			post name = newName;

	-- Get name of the client
	public getName: () ==> NotEmptyString
		getName() ==
			return name
			pre true
			post RESULT = name;

	-- Get id of the client
	public pure getId: () ==> nat1
		getId() ==
			return id
			pre true
			post RESULT = id;

	-- Register that the client bought a product.
	public buyProduct: Product ==> ()
		buyProduct(product) ==
			(
			boughtProducts := boughtProducts union {product};
			)
			pre true
			post
				card boughtProducts = card boughtProducts~ + 1 -- Has one
more product bought than previously.
				and product in set boughtProducts;
```

```
	-- Register that the client sold a product
	public sellProduct: Product ==> ()
	sellProduct(product) ==
		soldProducts := soldProducts union {product}
			pre true
			post
				card soldProducts = card soldProducts~ + 1  -- Has one more
product sold than previously.
				and product in set soldProducts;

	-- Register that the client bought a gift card
	public buyGiftCard: GiftCard ==> ()
		buyGiftCard(giftCard) ==
			boughtGiftCards := boughtGiftCards union {giftCard}
			pre true
			post card boughtGiftCards = card boughtGiftCards~ + 1 and
			giftCard in set boughtGiftCards;

	-- Get the set of products sold by the client
	public pure getProductsSold: () ==> set of Product
		getProductsSold() ==
			return soldProducts
			pre true
			post RESULT = soldProducts;

	-- Get the set of products bought by the client
	public pure getProductsBought: () ==> set of Product
		getProductsBought() ==
			return boughtProducts
			pre true
			post RESULT = boughtProducts;

	-- Get the set of gift cards bought by the client
	public pure getGiftCardsBought: () ==> set of GiftCard
		getGiftCardsBought() ==
			return boughtGiftCards
			pre true
			post RESULT = boughtGiftCards;
end Client
```

```
-- Cloths sold in Kid2Kid stores
class Clothing is subclass of Product
types
	public ClothingCat = <Jeans> | <Dresses> | <Pajamas>;
instance variables
	maxAge: nat;
	minAge: nat;
	subCategory: ClothingCat;
	inv minAge > 0 and minAge < 13; -- Kid2Kid only has cloths for children less than
13 years old.
	inv maxAge > 0 and maxAge < 13;
	inv maxAge >= minAge;
operations
public Clothing: ProductState * [CustomString] * nat * nat * ClothingCat ==> Clothing
	Clothing(productState, desc, minAgeInput, maxAgeInput, subCategoryInput) ==
		(
		idCounter := idCounter + 1;
		id := idCounter;
		state := productState;
		description := desc;
		minAge := minAgeInput;
		maxAge := maxAgeInput;
		subCategory := subCategoryInput;
		setPrices();
		return self;
		)
		pre true
		post
			minAge = minAgeInput and
			maxAge = maxAgeInput and
			subCategory = subCategoryInput and
			state = productState and
			RESULT = self;
end Clothing
```

```
-- A simple date class with only day, year and month
class Date
instance variables
	year : nat1;
	month: nat1;
	day : nat1;
	inv day <= 31;
	inv year >= 2000;
	inv month <= 12;

operations
	public Date: nat1 * nat1 * nat1 ==> Date
	Date(d,m,y) ==
		(
		day := d;
		month := m;
		year := y;
		return self;
		)
		pre d <= 31 and y >= 2000 and m <= 12
		post day=d and month=m and year = y and RESULT=self;
end Date
```

```
-- Footwear sold in Kid2Kid stores
class Footwear is subclass of Product
types
	public FootwearCat = <Boots> | <Party> | <Sports>;
instance variables
	size: nat;
	subCategory: FootwearCat;
	inv size > 15 and size < 45;
operations
	public Footwear: ProductState * [CustomString] * nat * FootwearCat ==> Footwear
		Footwear(productState, desc, sizeInput, footwearCat) ==
		(
			size := sizeInput;
			state := productState;
			description := desc;
			subCategory := footwearCat;
			setPrices();
			return self;
		)
		pre true
		post size = sizeInput and
		subCategory = footwearCat and
		state = productState and
		RESULT=self;
end Footwear
```

```
-- Furniture sold in Kid2Kid stores
class Furniture is subclass of Product
types
	public FurnitureCat = <Cribs> | <Beds> | <ToyBoxes>;
instance variables
	subCategory: FurnitureCat;
operations
	public Furniture: ProductState * [CustomString] * FurnitureCat ==> Furniture
		Furniture(productState, desc, furnitureCat) ==
		(
			idCounter := idCounter + 1;
			id := idCounter;
			state := productState;
			description := desc;
			subCategory := furnitureCat;
			setPrices();
			return self;
		)
		pre true
		post subCategory = furnitureCat and state = productState and RESULT=self;
end Furniture
```

```
-- Gear sold in Kid2Kid stores
class Gear is subclass of Product
types
	public GearCat = <Swings> | <Bathtubs> | <Trolleys>;
instance variables
	maxAge: [nat] := nil; -- defaults to nil
	minAge: nat;
	subCategory: GearCat;
	inv minAge < 13 and minAge > 0
operations
	public Gear: ProductState * [CustomString] * nat * [nat] * GearCat ==> Gear
	Gear(productState, desc, minAgeInput, maxAgeInput, subCategoryInput) ==
		(
		idCounter := idCounter + 1;
		id := idCounter;
		state := productState;
		description := desc;
		minAge := minAgeInput;
		maxAge := maxAgeInput;
		subCategory := subCategoryInput;
		setPrices();
		return self;
		)
		pre minAgeInput > 0 and minAgeInput < 13
		post minAge = minAgeInput and
			subCategory = subCategoryInput and
			RESULT = self;
end Gear
```

```
/*
* GiftCards can be used to buy products in a store, but they can onle be used once.
* GiftCards can be bought in one store and used in any other.
*/
class GiftCard
types
values
	private static GiftValues : set of nat1 = {10, 20, 40}; -- Set of possible
GiftCard values
instance variables
	static idCounter: nat := 0;
	id: nat1;
	value: nat1;
	inv value in set GiftValues;

operations
	public GiftCard: nat1 ==> GiftCard
	GiftCard(v) ==
		(idCounter := idCounter + 1; id := idCounter; value := v; return self;)
		pre v in set GiftValues
		post id = idCounter~+1 and value = v and RESULT=self;

	-- Get value of the GiftCard
	public pure getValue: () ==> nat1
	getValue() ==
		return value
		pre true
		post RESULT = value;

	-- Get possible values of any GiftCard
	public static pure getPossibleValues: () ==> set of nat1
	getPossibleValues() ==
		return GiftValues
		pre true
		post RESULT = GiftValues;
end GiftCard
```

```
/* Main class where all interactions will pass through.
*/
class Kid2Kid
types
	public NotEmptyString = seq1 of char;
	public UserType = <Cashier> | <Admin> | <LoggedOut>;

values

instance variables
	clients: set of Client := {};
	stores: set of Store := {};
	activeGiftCards: set of GiftCard := {};
	transactions: set of Transaction := {};

	loggedInType: UserType := <LoggedOut>;
	loggedInUsername: NotEmptyString;
	loggedInStore: NotEmptyString; -- for logged in cashiers

operations

	public Kid2Kid: () ==> Kid2Kid
	Kid2Kid() ==
		return self
		pre true
		post clients = {} and stores = {} and activeGiftCards = {} and
transactions = {} and
		loggedInType = <LoggedOut> and RESULT = self;

	-- Login as admin
	public login: NotEmptyString ==> UserType
	login(name) ==
	(
		if (name = "Admin")
		then (loggedInType := <Admin>; loggedInUsername := "Admin")
		else (loggedInType := <LoggedOut>; loggedInUsername := " ");

		return loggedInType
	)
		pre true
		post RESULT = loggedInType;

	-- Login as cashier
	public login: NotEmptyString * NotEmptyString ==> UserType
	login(storeLocation, cashierName) ==
		(
		if (storeLocation in set getStoreLocationsInternal() and cashierName in
set getStore(storeLocation).getCashierNames())
		then (loggedInType := <Cashier>; loggedInUsername := cashierName;
loggedInStore := storeLocation)
		else (loggedInType := <LoggedOut>; loggedInUsername := " ";
loggedInStore := " ");

		return loggedInType
		)
		pre true
		post RESULT = loggedInType;

	-- Get a store using its name
	private pure getStore: NotEmptyString ==> Store
	getStore(name) ==
		(
```

```
		dcl store: Store;
		for all s in set stores do
			if (name = s.getLocation()) then store := s;
		return store
		)
		pre true
		post RESULT in set stores and RESULT.getLocation() = name;

	-- Get the cashier that is loggedin
	public pure getLoggedInCashier: () ==> StoreCashier
	getLoggedInCashier() ==
		(
		dcl cashier: StoreCashier;
		for all c in set getCashiersInternal() do
			if (loggedInUsername = c.getName() and loggedInStore =
c.getStore().getLocation())
			then cashier := c;
		return cashier
		)
		pre loggedInType = <Cashier>
		post RESULT in set getCashiersInternal() and RESULT.getName() =
loggedInUsername and RESULT.getStore().getLocation() = loggedInStore;

	-- Remove a cashier from the system
	public removeCashier: StoreCashier ==> ()
	removeCashier(c) ==
		c.getStore().removeCashier(c)
		pre hasAdminPerms() and c in set c.getStore().getCashiers()
		post c not in set c.getStore().getCashiers();

	-- Get all the transactions created
	public pure getTransactions: () ==> set of Transaction
	getTransactions() ==
		return transactions
		pre hasAdminPerms()
		post RESULT = transactions;

	-- Get all the transactions authrozied by the logged in cashier
	public pure getTransactionsOfLoggedInCashier: () ==> set of Transaction
	getTransactionsOfLoggedInCashier() ==
		return getTransactionsOfCashierInternal(getLoggedInCashier())
		pre loggedInType = <Cashier>
		post RESULT = getTransactionsOfCashierInternal(getLoggedInCashier());

	-- Get all the transactions of a given cashier
	public pure getTransactionsOfCashier: StoreCashier ==> set of Transaction
	getTransactionsOfCashier(cashier) ==
		return getTransactionsOfCashierInternal(cashier)
		pre hasAdminPerms()
		post RESULT = getTransactionsOfCashierInternal(cashier);

	-- Get transactions of a given cashier. Bypasses permissions.
	private pure getTransactionsOfCashierInternal: StoreCashier ==> set of
Transaction
	getTransactionsOfCashierInternal(cashier) ==
		return {x | x in set transactions & x.getCashier() = cashier}
		pre true
		post true;

	-- Get the set of active giftcards
	public pure getActiveGiftCards: () ==> set of GiftCard
	getActiveGiftCards() ==
```

```
            return activeGiftCards
            pre hasCashierPerms()
            post RESULT = activeGiftCards;

        -- Get names of all cashiers. Bypasses login permissions.
        private pure getCashierNamesInternal: () ==> set of NotEmptyString
        getCashierNamesInternal() ==
            (
            dcl names: set of NotEmptyString := {};
            for all cashier in set getCashiersInternal() do names := names union
{cashier.getName()};
            return names;
            )
            pre true
            post true;

        -- Get set of locations of all stores. Bypasses login permissions.
        private pure getStoreLocationsInternal: () ==> set of NotEmptyString
        getStoreLocationsInternal() ==
            (
            dcl names: set of NotEmptyString := {};
            for all store in set stores do names := names union {store.getLocation()};
            return names;
            )
            pre true
            post true;

        -- Get all cashiers. Bypasses login permissions.
        private pure getCashiersInternal: () ==> set of StoreCashier
        getCashiersInternal() ==
            (
            dcl cashiers: set of StoreCashier := {};
            for all store in set stores do cashiers := cashiers union
store.getCashiers();
            return cashiers;
            )
            pre true
            post true;

        -- Get all cashiers and check for admin permissions.
        public pure getCashiers: () ==> set of StoreCashier
        getCashiers() ==
            return getCashiersInternal()
            pre hasAdminPerms()
            post true;

        -- Get names of all cashiers and check for admin permissions.
        public pure getCashierNames: () ==> set of NotEmptyString
        getCashierNames() ==
            return getCashierNamesInternal()
            pre hasAdminPerms()
            post true;

        -- Add a client to the system
        public addClient: Client ==> ()
        addClient(c) ==
            clients := clients union {c}
            pre hasAdminPerms()
            post c in set clients; -- New client is added to the system

        -- Remove client from system.
        public removeClient: Client ==> ()
```

```
	removeClient(c) ==
		clients := clients \ {c}
		pre hasAdminPerms() and c in set clients -- Client is part of the system
		post c not in set clients; -- Client isn't part of the system

	-- Add a store to the system and check for admin permissions.
	public addStore: Store ==> ()
	addStore(s) ==
		stores := stores union {s}
		pre hasAdminPerms() -- New store is added to the system
		post s in set stores;

	-- Get all the clients of the system and check for cashier permissions.
	public pure getClients: () ==> set of Client
	getClients() ==
		return clients
		pre hasCashierPerms()
		post RESULT = clients;

	-- Get all the stores of the system and check for admin permissions.
	public pure getStores: () ==> set of Store
	getStores() ==
		return stores
		pre hasAdminPerms()
		post RESULT = stores;

	-- Get all the products at a given store.
	public pure getProductsAtStore: Store ==> set of Product
	getProductsAtStore(store) ==
		return store.getProductsAvailable()
		pre store in set stores
		post RESULT = store.getProductsAvailable();

	-- Buy product of a client at a store with a specific cashier.
	public buyProductInCash: Product* Client * StoreCashier * Date ==> ()
	buyProductInCash(product, client, cashier, date) ==
		(
		cashier.buyProductInCash(product);
		client.sellProduct(product);
		addPurchaseTransaction(date, client, {product}, cashier);
		)
		pre hasCashierPerms() and client in set clients and cashier in set
getCashiersInternal()
		post true;

	-- Buy product of a client at a store with a specific cashier and pay in credit
notes
	public buyProductInCreditNotes: Product * Client * StoreCashier * Date ==> ()
	buyProductInCreditNotes(product, client, cashier, date) ==
		(
		cashier.buyProductInCreditNotes(product, client);
		client.sellProduct(product);
		addPurchaseTransaction(date, client, {product}, cashier);
		)
		pre hasCashierPerms() and client in set clients and cashier in set
getCashiersInternal()
		post true;

	-- Sell product to a client at a store with a specific cashier.
	public sellProductInCash: Product * Client * StoreCashier * Date ==> ()
	sellProductInCash(p,c,sc,d) ==
		(
```

```
            sc.sellProduct(p);
            c.buyProduct(p);
            addSaleProductTransaction(d, c, {p}, {}, sc);
            )
            pre hasCashierPerms() and c in set clients and sc in set
getCashiersInternal()
            post true;

      -- Sell product to a client at a store and receive in credit notes.
      public sellProductInCreditNotes: Product * Client * StoreCashier * Date ==> ()
      sellProductInCreditNotes(p, c, sc, d) ==
            (
            sc.sellProductInCreditNotes(p, c);
            c.buyProduct(p);
            addSaleProductTransaction(d, c, {p}, {}, sc);
            )
            pre hasCashierPerms() and c in set clients and sc in set
getCashiersInternal()
            post card transactions = card transactions~ + 1;

      -- Sell product to a client at a store and receive in gift cards.
      public sellProductInGiftCards: Product * Client * StoreCashier * set of GiftCard
* Date ==> ()
      sellProductInGiftCards(p, c, sc, gc, d) ==
            (
            sc.sellProduct(p);
            c.buyProduct(p);
            addSaleProductTransaction(d, c, {p}, gc, sc);
            for all giftCard in set gc do useGiftCard(giftCard);
            )
            pre hasCashierPerms() and card gc > 0 and gc subset activeGiftCards and
                  sumGiftCardValues(gc) >= p.getSellPrice() and
                  c in set clients and
                  sc in set getCashiersInternal()
            post card transactions = card transactions~ + 1 and
                  gc inter activeGiftCards = {};

      -- Sell a giftcard to a client at a store
      public sellGiftCard: Client * StoreCashier * Date * nat1 ==> GiftCard
      sellGiftCard(c, sc, d, value) ==
            (
            dcl gc: GiftCard := sc.sellGiftCard(value);
            c.buyGiftCard(gc);
            addSaleGCTransaction(d, c, {gc}, sc);
            activateGiftCard(gc);
            return gc;
            )
            pre hasCashierPerms() and c in set clients and sc in set
getCashiersInternal()
            post card activeGiftCards = card activeGiftCards~ + 1;

      -- Activate a given giftcard
      private activateGiftCard: GiftCard ==> ()
      activateGiftCard(giftCard) ==
            activeGiftCards := activeGiftCards union {giftCard}
            pre hasCashierPerms()
            post activeGiftCards = activeGiftCards union {giftCard};

      -- Use a given giftcard, making it unusable
      private useGiftCard: GiftCard ==> ()
      useGiftCard(giftCard) ==
            activeGiftCards := activeGiftCards \ {giftCard}
```

```
		pre hasCashierPerms() and giftCard in set activeGiftCards
		post activeGiftCards = activeGiftCards~ \ {giftCard};

	-- Create and save the transaction that sells a set of products
	private addSaleProductTransaction: Date * Client * set of Product * set of
GiftCard * StoreCashier ==> ()
	addSaleProductTransaction(d,c,sp,gcs,sc) ==
		(
		dcl transaction: Transaction := new Sale(d,c,sp,gcs,sc);
		transactions := transactions union {transaction};
		)
		pre hasCashierPerms() and card sp >= 1 and
		c in set clients and sc in set getCashiersInternal()
		post card transactions = card transactions~ + 1;

	-- Create and save the transaction that sells a set of giftcards
	private addSaleGCTransaction: Date * Client * set of GiftCard * StoreCashier ==>
()
	addSaleGCTransaction(d,c,gcs,sc) ==
	(
		dcl transaction: Transaction := new Sale(d,c,gcs,sc);
		transactions := transactions union {transaction};
		)
		pre hasCashierPerms() and card gcs >= 1 and
		c in set clients and sc in set getCashiersInternal()
		post card transactions = card transactions~ + 1;

	-- Create and save the transaction that purchases a set of products
	private addPurchaseTransaction: Date * Client * set of Product * StoreCashier ==>
()
	addPurchaseTransaction(d,c,sp,sc) ==
	(
		dcl transaction: Transaction := new Purchase(d,c,sp,sc);
		transactions := transactions union {transaction};
		)
		pre hasCashierPerms() and card sp >= 1
		and c in set clients and sc in set getCashiersInternal()
		post card transactions = card transactions~ + 1;

	-- Check if the logged in user has cashier permissions
	private pure hasCashierPerms: () ==> bool
	hasCashierPerms() ==
		return loggedInType in set {<Admin>, <Cashier>}
		pre true
		post RESULT = loggedInType in set {<Admin>, <Cashier>};

	-- Check if the logged in user has admin permissions
	private pure hasAdminPerms: () ==> bool
	hasAdminPerms() ==
		return loggedInType = <Admin>
		pre true
		post RESULT = (loggedInType = <Admin>);

	-- Sum the values of a set of giftcards
	public pure sumGiftCardValues: (set of GiftCard) ==> nat1
	sumGiftCardValues(giftCards) ==
		(
		dcl res: nat := 0;
		for all gc in set giftCards do res := res + gc.getValue();
		return res;
		)
		pre card giftCards > 0
```

```
            post RESULT > 0;
end Kid2Kid
```

```
-- Abstract class that represents any product sold in Kid2Kid stores, except for
GiftCards.
class Product
types
	public CustomString = seq1 of char;
	public ProductState = <New> | <Low_Use> | <High_Use>;
values
	public CREDIT_NOTE_MULTIPLIER: real = 1.2;
instance variables
	protected static idCounter: nat := 0;
	protected id: nat;
	protected buyPrice: real := 0;
	protected sellPrice: real := 9999999;
	protected state: ProductState;
	protected description: [CustomString] := nil;
	inv sellPrice >= buyPrice;
operations

	-- Get the sell price of the Product
	public pure getSellPrice: () ==> real
	getSellPrice() ==
		return sellPrice
		pre true
		post RESULT = sellPrice;

	-- Get the buy price of the Product
	public pure getBuyPrice: () ==> real
	getBuyPrice() ==
		return buyPrice
		pre true
		post RESULT = buyPrice;

	-- Get the credit notes value of the Product
	public pure getCreditNotesValue: () ==> real
	getCreditNotesValue() ==
		return buyPrice * CREDIT_NOTE_MULTIPLIER
		pre true
		post RESULT = buyPrice * CREDIT_NOTE_MULTIPLIER;

	-- Update the description of the Product.
	public setDescription: CustomString ==> ()
	setDescription(newDescription) ==
		description := newDescription
		pre true
		post description = newDescription;

	-- Algorithm that sets the buy price of the Product
	protected setPrices: () ==> ()
	setPrices() ==
		(
		buyPrice :=  10 * getStateValue();
		sellPrice := 1.3 * buyPrice
		)
	pre state <> undefined
	post buyPrice > 0 and sellPrice > buyPrice;

	-- Set sell price manually.
	public setSellPrice: real ==> ()
	setSellPrice(newPrice) ==
		sellPrice := newPrice
	pre buyPrice <> undefined and newPrice > buyPrice
	post sellPrice > buyPrice and sellPrice = newPrice;
```

```
	-- Get the state of the product as a number to be used in the algorithm that
evaluates the Product value.
	public pure getStateValue: () ==> real
	getStateValue() ==
	(
	if state = <New> then return 1.0;
	if state = <Low_Use> then return 0.8;
	return 0.5;
	)
	pre state = <New> or state = <Low_Use> or state = <High_Use>
	post RESULT = 0.5 or RESULT = 0.8 or RESULT = 1.0;

	-- Get the description of the Product
	public pure getDescription: () ==> CustomString
	getDescription() ==
		return description
		pre true
		post RESULT = description;

end Product
```

```
-- A purchase transaction
class Purchase is subclass of Transaction
instance variables
operations
	public Purchase: Date * Client * set of Product * StoreCashier ==> Purchase
	Purchase(d, c, ps, sc) ==
			(
			date := d;
			client := c;
			productsTransacted := ps;
			storeAuthorizer := sc;
			value := sumProductValues(productsTransacted);
			return self;
			)
			pre card ps >= 1
			post RESULT = self;


	-- Sum the buy prices of the products in the transaction
	protected pure sumProductValues: set of Product ==> nat
	sumProductValues(pSet) ==
		(
		dcl result: nat := 0;
		for all p in set pSet do
			(
			result := result + p.getBuyPrice();
			);
		return result;
		)
		pre true
		post true; -- same as body

traces
-- TODO Define Combinatorial Test Traces here
end Purchase


-- Represents a collection of items that were sold by a store
```

```
class Sale is subclass of Transaction
instance variables
	giftCardsUsed: set of GiftCard := {};
	giftCardsSold: set of GiftCard := {};

operations

	-- Store sells products. Client possibly uses active gift cards.
	public Sale: Date * Client * set of Product * set of GiftCard * StoreCashier ==>
Sale
	Sale(d, c, p, cardsUsed, sc) ==
		(
		date:=d;
		client:=c;
		productsTransacted:=p;
		giftCardsUsed:=cardsUsed;
		storeAuthorizer:= sc;
		value:=sumProductValues(p);
		return self
		)
		pre card p >= 1 -- there is at least on product being sold
		post
			date = d and
			client = c and
			productsTransacted = p and
			giftCardsUsed = cardsUsed and
			storeAuthorizer = sc and
			value = sumProductValues(productsTransacted) and
			RESULT = self;

	-- Store sells gift cards.
	public Sale: Date * Client * set of GiftCard * StoreCashier ==> Sale
	Sale(d, c, g, sc) ==
		(
		date := d;
		client := c;
		giftCardsSold := g;
		storeAuthorizer := sc;
		value := sumGCValues(g);
		return self;
		)
		pre card g >= 1
		post value = sumGCValues(g) and
		date = d and
		client = c and
		giftCardsSold = g and
		storeAuthorizer = sc and
		RESULT = self;

	-- Get gift cards that were used in the transaction
	public pure getGiftCardsUsed: () ==> set of GiftCard
	getGiftCardsUsed() ==
		return giftCardsUsed
		pre true
		post RESULT = giftCardsUsed;

	-- Get gift cards that were sold in the transaction
	public pure getGiftCardsSold: () ==> set of GiftCard
	getGiftCardsSold() ==
		return giftCardsSold
		pre true
		post RESULT = giftCardsSold;
```

```
	-- Sum the sell prices of the products
	protected pure sumProductValues: set of Product ==> real
	sumProductValues(pSet) ==
		(
		dcl result: real := 0;
		for all p in set pSet do
			(
			result := result + p.getSellPrice();
			);
		return result;
		)
		pre true
		post true; -- same as body

	-- Sum the values of a set of giftcards
	private pure sumGCValues: set of GiftCard ==> nat
	sumGCValues(gcSet) ==
		(
		dcl result: nat := 0;
		for all g in set gcSet do
			(
			result := result + g.getValue();
			);
		return result;
		)
		pre true
		post true; -- same as body

end Sale


-- A physical Kid2Kid store
```

```
class Store
types
	public NotEmptyString = seq1 of char;
	public ClientsToCredits = map nat to real;
values
instance variables
	location: NotEmptyString; -- geographic place, like the city
	productsAvailable: set of Product := {}; -- products that the store has in stock
	productsSold: set of Product := {}; -- products that the store already sold
	giftCardsSold: set of GiftCard := {}; -- gift cards that the store already sold
	clientsCreditNotes: ClientsToCredits := { |-> }; -- map ids of clients to its
credit notes
	cashiers: set of StoreCashier := {}; -- set of store cashiers that work here

operations
	public Store: NotEmptyString ==> Store
	Store(storeLocation) ==
		(location := storeLocation;
		return self;)
		pre len storeLocation > 0
		post
			location = storeLocation and
			productsAvailable = {} and
			productsSold = {} and
			clientsCreditNotes = { |-> } and
			cashiers = {};

	-- Add a cashier to the store
	public addCashier: StoreCashier ==> ()
		addCashier(cashier) ==
		cashiers := cashiers union {cashier}
		pre true
		post cashiers = cashiers~ union {cashier};

	-- Add a product to the store and pay in cash
	public buyProduct: Product ==> ()
		buyProduct(p) ==
		productsAvailable := productsAvailable union {p}
		pre true
		post productsAvailable = productsAvailable~ union {p};

	-- Add a product to the store and pay with credit notes
	public buyProductInCreditNotes: Product * nat1 ==> ()
	buyProductInCreditNotes(p, clientId) ==
		(
		buyProduct(p);
		addCreditNote(clientId, p);
		)
		pre true
		post productsAvailable = productsAvailable~ union {p} and
		checkCreditNotes(clientId, p, clientsCreditNotes, clientsCreditNotes~);

	-- Sell a giftcard
	public sellGiftCard: (nat1) ==> GiftCard
	sellGiftCard(value) ==
		(
		dcl newGiftCard:GiftCard := new GiftCard(value);
		giftCardsSold := giftCardsSold union {newGiftCard};
		return newGiftCard;
		)
		pre true
		post card giftCardsSold = card giftCardsSold~ + 1;
```

```
	-- Sell a product that is available and receive in cash
	public sellProduct: Product ==> ()
	sellProduct(p) ==
		(
		productsAvailable := productsAvailable \ {p};
		productsSold := productsSold union {p};
		)
		pre hasProduct(p)
		post productsAvailable = productsAvailable~ \ {p} and productsSold =
productsSold~ union {p};

	-- Sell a product that is available and receive in credit notes
	public sellProductInCreditNotes: Product * nat ==> ()
	sellProductInCreditNotes(p, clientId) ==
		(
		sellProduct(p);
		spendCreditNote(clientId, p.getSellPrice());
		)
		pre hasProduct(p) and clientsCreditNotes(clientId) >= p.getSellPrice()
		post productsAvailable = productsAvailable~ \ {p} and
				productsSold = productsSold~ union {p}; -- should also
check that creditNotes of client have decreased

	-- Get creditnotes of a client. Returns 0 if client has no credit notes yet.
	public getCreditNotesOfClient: nat ==> real
	getCreditNotesOfClient(clientId) ==
		if clientId in set dom clientsCreditNotes then
			(return clientsCreditNotes(clientId))
		else (
				clientsCreditNotes := clientsCreditNotes ++ {clientId |-> 0};
				return 0
			)
		pre true
		post RESULT = clientsCreditNotes(clientId);

	-- Get credit notes of all clients
	public pure getClientsCreditNotes: () ==> ClientsToCredits
	getClientsCreditNotes() ==
		return clientsCreditNotes
		pre true
		post RESULT = clientsCreditNotes;

	-- Get location of the store
	public pure getLocation: () ==> NotEmptyString
	getLocation() ==
		return location
		pre true
		post RESULT = location;

	-- Get products available at the store
	public pure getProductsAvailable: () ==> set of Product
	getProductsAvailable() ==
		return productsAvailable
		pre true
		post RESULT = productsAvailable;

	-- Get cashiers that work in the store
	public pure getCashiers: () ==> set of StoreCashier
	getCashiers() ==
		return cashiers
		pre true
```

```
		post RESULT = cashiers;

	-- Get cashier using his name
	public pure getCashier: NotEmptyString ==> StoreCashier
	getCashier(name) ==
		(
		dcl cashier: StoreCashier;
		for all c in set cashiers do
			if (c.getName() = name) then cashier := c;
		return cashier
		)
		pre true
		post RESULT in set cashiers and RESULT.getName() = name;

	-- Get the names of all cashiers that work in the store
	public pure getCashierNames: () ==> set of NotEmptyString
	getCashierNames() ==
		(
		dcl names: set of NotEmptyString := {};
		for all cashier in set cashiers do names := names union
{cashier.getName()};
		return names;
		)
		pre true
		post true;

	-- Calculate inventory value
	public pure getInventoryValue: () ==> real
	getInventoryValue() ==
		(
		dcl sum: real := 0;
		for all product in set productsAvailable do
			sum := sum + product.getBuyPrice();
		return sum;
		)
		pre true
		post true;

	-- Calculate revenue using products sold
	public pure getRevenue: () ==> real
	getRevenue() ==
		(
		dcl sum: real := 0;
		for all product in set productsSold do
			sum := sum + (product.getSellPrice() - product.getBuyPrice());
		return sum;
		)
		pre true
		post true;

	-- Update the location
	public setLocation: NotEmptyString ==> ()
	setLocation(l) ==
		location := l
		pre true
		post location = l;

	-- Remove a cashier from the store
	public removeCashier: StoreCashier ==> ()
	removeCashier(c) ==
		cashiers := cashiers \ {c}
		pre c in set cashiers
```

```
            post c not in set cashiers;

      -- Add a credit note to a client
      public addCreditNote: real * Product ==> ()
      addCreditNote(clientId, p) ==
                  (
                  (if clientId not in set dom clientsCreditNotes
                        then clientsCreditNotes := clientsCreditNotes ++ {clientId |-
> 0}
                  );
                  clientsCreditNotes := clientsCreditNotes ++
                        {clientId |-> clientsCreditNotes(clientId) +
p.getCreditNotesValue()}
                  )
                  pre true
                  post checkCreditNotes(clientId, p, clientsCreditNotes,
clientsCreditNotes~);

      /**
      Spend a credit note
      The value of the credit note to send should be less or equal than the creditNotes
amount
      */
      public spendCreditNote: nat * real ==> ()
      spendCreditNote(clientId, value) ==
                  clientsCreditNotes := clientsCreditNotes ++ {clientId |->
clientsCreditNotes(clientId) - value}
                  pre clientId in set dom clientsCreditNotes and value <=
clientsCreditNotes(clientId)
                  post clientsCreditNotes = clientsCreditNotes~ ++ {clientId |->
clientsCreditNotes~(clientId) - value};

      -- Check if the store has a given product in stock
      public pure hasProduct: Product ==> bool
      hasProduct(p) ==
            return p in set productsAvailable
            post RESULT = p in set productsAvailable;

functions
      -- Check that the credit notes amount of a client has increased by the value of
the product
      public checkCreditNotes: nat * Product * ClientsToCredits * ClientsToCredits ->
bool
      checkCreditNotes(clientId, p, newCreditNotes, oldCreditNotes) ==
      (if clientId not in set dom oldCreditNotes then
                  newCreditNotes(clientId) = p.getCreditNotesValue()
      else
                  newCreditNotes = oldCreditNotes ++ {clientId |->
oldCreditNotes(clientId) + p.getCreditNotesValue()}
      )
end Store
```

```
-- A cashier that works in a store
-- A cashier needs to authorize most most of the operations
class StoreCashier
types
	public NotEmptyString = seq1 of char;

instance variables
	name: NotEmptyString;
	store: Store; -- cashier workplace

operations
	public StoreCashier: NotEmptyString * Store ==> StoreCashier
	StoreCashier(n, w) ==
		(name := n; store := w; return self)
		pre len n > 0
		post name = n and store = w and RESULT = self;

	-- Get the name of the cashier
	public pure getName: () ==> NotEmptyString
	getName() ==
		return name
		pre true
		post RESULT = name;

	-- Update the name of the cashier
	public setName: NotEmptyString ==> ()
	setName(n) ==
		name := n
		pre true
		post name = n;

	-- Get the store in which the cashier works
	public pure getStore: () ==> Store
	getStore() ==
		return store
		pre true
		post RESULT = store;

	-- Register that a product was bought.
	public buyProductInCash: Product ==> ()
	buyProductInCash(p) ==
		(
		store.buyProduct(p);
		)
		pre true
		post true;

	-- Register that a product was bought and credit notes were created for the
client.
	public buyProductInCreditNotes: Product * Client ==> ()
	buyProductInCreditNotes(p, client) ==
		(
		store.buyProductInCreditNotes(p, client.getId());
		)
		pre true
		post true;


	-- Sell to client. Receive in cash or gift cards
	public sellProduct: Product ==> ()
	sellProduct(p) ==
		(
```

```
            store.sellProduct(p);
            )
            pre true
            post true;

        -- Sell to client. Receive in credit notes
        public sellProductInCreditNotes: Product * Client ==> ()
        sellProductInCreditNotes(p, client) ==
            (
                store.sellProductInCreditNotes(p, client.getId());
            )
            pre true
            post true;

        -- Sell a giftcard to a client
        public sellGiftCard: (nat1) ==> GiftCard
        sellGiftCard(value) ==
            return store.sellGiftCard(value)
            pre true
            post true

end StoreCashier
```

```
-- A toy that is sold in the Kid2Kid stores.
class Toy is subclass of Product
types
	public ToyCategory = <Puzzles> | <Legos> | <Cars>;
	public NotEmptyString = seq1 of char;
instance variables
	private minAge: nat; -- Min age is mandatory
	private maxAge: [nat]:=nil; -- optional, nil as default
	private subCategory: ToyCategory;
	inv minAge < 13 and minAge > 0
operations
	public Toy: ProductState * [CustomString] * nat * [nat] * ToyCategory ==> Toy
	Toy(productState, desc, minAgeInput, maxAgeInput, subCategoryInput) ==
		(

		idCounter := idCounter + 1;
		id := idCounter;
		state := productState;
		description := desc;
		minAge := minAgeInput;
		maxAge := maxAgeInput;
		subCategory := subCategoryInput;
		setPrices();
		return self;
		)
		pre minAgeInput > 0 and minAgeInput < 13
		post minAge = minAgeInput and
		subCategory = subCategoryInput and
		state = productState and
		RESULT = self;

end Toy
```

```
/*
A transfer of value between a client and a store.
Every transaction needs to be authorized by a store cashier
*/
class Transaction
instance variables
	protected value: real;
	protected date: Date;
	protected client: Client;
	protected productsTransacted: set of Product := {};
	protected storeAuthorizer: StoreCashier;

operations
	-- Get the products transacted
	public pure getProducts: () ==> set of Product
	getProducts() ==
		(return productsTransacted)
		pre true
		post RESULT = productsTransacted;

	-- Get the cashier that authorized the transaction
	public pure getCashier: () ==> StoreCashier
	getCashier() ==
		return storeAuthorizer
		pre true
		post RESULT = storeAuthorizer;

	-- Get the value of the the transaction
	public pure getValue: () ==> nat
	getValue() ==
		return value
	pre true
	post RESULT = value;

	-- Get the total value of the products transacted
	public pure getSumProductValues: () ==> nat
	getSumProductValues() ==
		return sumProductValues(productsTransacted)
		pre true
		post RESULT = sumProductValues(productsTransacted);

	-- Sum product values
	protected pure sumProductValues: set of Product ==> nat
	sumProductValues(pSet) ==
		is subclass responsibility;

end Transaction
```

```
-- Test the Kid2Kid system
class Kid2KidTest
types
	public NotEmptyString = seq1 of char;
instance variables
	kid2kid : Kid2Kid := new Kid2Kid();
	client: Client := new Client("abc");
	store: Store := new Store("Porto");
	cashier: StoreCashier;
	today: Date := new Date(4,1,2019);
	products: seq of Product := [];
operations

	private assertTrue: bool ==> ()
	 assertTrue(cond) == return
	 pre cond;

	-- Login as admin
	private loginAdmin: () ==> ()
	loginAdmin() ==
	(
		assertTrue(kid2kid.login("Bad admin name") = <LoggedOut>);
		assertTrue(kid2kid.login("Admin") = <Admin>);
	);

	-- Login as cashier
	private loginCashier: NotEmptyString * NotEmptyString ==> ()
	loginCashier(storeName, cashierName) ==
	(
		assertTrue(kid2kid.login("Bad store name", "Bad cashier name") =
<LoggedOut>);
		assertTrue(kid2kid.login(storeName, cashierName) = <Cashier>);
		assertTrue(kid2kid.getLoggedInCashier().getName() = cashierName);
	);

	-- Add a client to the system
	private testAddClient: () ==> ()
		testAddClient() ==
		(
			assertTrue(card kid2kid.getClients() = 0);
			kid2kid.addClient(client);
			assertTrue(card kid2kid.getClients() = 1);
			kid2kid.addClient(new Client("abcd"));
			assertTrue(card kid2kid.getClients() = 2);
			assertTrue(not exists i, j in set kid2kid.getClients() & i <> j and
i.getId() = j.getId());
		);

	-- Add a store to the system
	private testAddStore: () ==> ()
		testAddStore() ==
		(
			assertTrue(card kid2kid.getStores() = 0);
			kid2kid.addStore(store);
			assertTrue(card kid2kid.getStores() = 1);
		);

	-- Add a cashier to an existing store
	private testAddCashierToStore: (NotEmptyString) ==> ()
		testAddCashierToStore(name) ==
		(
			dcl prevNumCashiers: nat := card store.getCashiers();
```

```
                cashier := new StoreCashier(name, store);
                store.addCashier(cashier);
                assertTrue(card store.getCashiers() = prevNumCashiers + 1);
        );

    -- Buy products as Admin and pay in cash. Test that transactions of cashier have
increased
    private testStoreBuyProductsInCashAsAdmin: () ==> ()
    testStoreBuyProductsInCashAsAdmin() ==
        (
                dcl toy: Product := new Toy(<New>, nil, 5, nil, <Cars>);
                dcl numTransactions: nat := card
kid2kid.getTransactionsOfCashier(cashier);
                kid2kid.buyProductInCash(toy,client,cashier,today);
                assertTrue(card kid2kid.getTransactionsOfCashier(cashier) =
numTransactions + 1);
        );

    -- Buy products as cashier. Test that transactions of cashier have increased
    private testStoreBuyProductsInCashAsCashier: () ==> ()
    testStoreBuyProductsInCashAsCashier() ==
        (
                dcl toy: Product := new Toy(<New>, nil, 5, nil, <Cars>);
                dcl numTransactions: nat := card
kid2kid.getTransactionsOfLoggedInCashier();
                kid2kid.buyProductInCash(toy, client, kid2kid.getLoggedInCashier(),
today);
                assertTrue(card kid2kid.getTransactionsOfLoggedInCashier() =
numTransactions + 1);
        );

    -- Products sold are saved in the store and in the client
    private testStoreBuyProductsInCash: () ==> (nat)
        testStoreBuyProductsInCash() ==
        (
                dcl toy: Product := new Toy(<New>, nil, 5, nil, <Cars>);
                dcl gear: Product := new Gear(<High_Use>, nil, 1, 20, <Bathtubs>);
                dcl furniture: Product := new Furniture(<New>, nil, <Beds>);
                products := [toy, gear, furniture];
                assertTrue(elems products inter store.getProductsAvailable() = {});
                assertTrue(elems products  inter client.getProductsSold() = {});
                kid2kid.buyProductInCash(toy,client,cashier,today);
                kid2kid.buyProductInCash(gear,client,cashier,today);
                kid2kid.buyProductInCash(furniture,client,cashier,today);
                assertTrue(elems products subset store.getProductsAvailable());
                assertTrue(elems products subset client.getProductsSold());
                return len products;
        );

    -- Credit notes are added to a client
    private testStoreBuyProductsInCreditNotes: () ==> (nat)
        testStoreBuyProductsInCreditNotes() ==
        (
                dcl boots: Product := new Footwear(<Low_Use>, "Boots", 20,
<Boots>);
                dcl jeans: Product := new Clothing(<Low_Use>, "Blue Jeans", 1, 2,
<Jeans>);
                dcl productsSet: set of Product := {boots, jeans};
                dcl clientPrevCredit: real :=
store.getCreditNotesOfClient(client.getId());
                assertTrue(productsSet inter store.getProductsAvailable() = {});
                assertTrue(productsSet inter client.getProductsSold() = {});
```

```
                kid2kid.buyProductInCreditNotes(boots, client, cashier, today);
                kid2kid.buyProductInCreditNotes(jeans, client, cashier, today);
                assertTrue(productsSet subset store.getProductsAvailable());
                assertTrue(productsSet subset client.getProductsSold());
                assertTrue(store.getCreditNotesOfClient(client.getId()) =
clientPrevCredit + boots.getCreditNotesValue() + jeans.getCreditNotesValue());
                return card productsSet;
        );

    -- Get cashiers returns expected number of cashiers
    private testGetCashiers: nat ==> ()
        testGetCashiers(expectedNumber) ==
        (
                assertTrue(card kid2kid.getCashiers() = expectedNumber);
        );

    -- Cashier names are as expected
    private testGetCashierNames: set of NotEmptyString ==> ()
        testGetCashierNames(namesSet) ==
        (
                assertTrue( (namesSet \ kid2kid.getCashierNames()) = {});
        );

    -- Remove a cashier and re-add him.
    private testRemoveCashier: () ==> ()
        testRemoveCashier() ==
        (
        dcl c: StoreCashier := cashier;
        assertTrue(c in set kid2kid.getCashiers());
        kid2kid.removeCashier(c);
        assertTrue(c not in set kid2kid.getCashiers());
        store.addCashier(c);
        assertTrue(c in set kid2kid.getCashiers());
        );

    -- Remove a client and re-add him
    private testRemoveClient: () ==> ()
        testRemoveClient() ==
        (
        dcl c: Client := client;
        assertTrue(c in set kid2kid.getClients());
        kid2kid.removeClient(c);
        assertTrue(c not in set kid2kid.getClients());
        kid2kid.addClient(c);
        assertTrue(c in set kid2kid.getClients());
        );

    -- Number of transactions is as expected
    private testGetTransactions: nat ==> ()
        testGetTransactions(expectedNumber) ==
        (
        assertTrue(card kid2kid.getTransactions() = expectedNumber);
        );

    -- Number of products in the store is as expected
    private testGetProductsAtStore: (nat) ==> ()
        testGetProductsAtStore(expectedNum) ==
        (
                assertTrue(card kid2kid.getProductsAtStore(store) = expectedNum);
        );

    -- GiftCards work as expected
```

```
	private testStoreGiftCards: (bool) ==> ()
		testStoreGiftCards(isAdmin) ==
		(
		dcl gcValue:nat := 40;
		dcl gc: GiftCard := kid2kid.sellGiftCard(client, cashier, today, gcValue);
		dcl saleProducts: Sale, saleGC: Sale;
		assertTrue(card kid2kid.getActiveGiftCards() = 1); -- GiftCards are
activated when sold.
		assertTrue(gc in set client.getGiftCardsBought()); -- Client saves
GiftCards bought.
		kid2kid.sellProductInGiftCards(products(1), client, cashier, {gc}, today);
-- A client buys a product with a giftcard
		assertTrue(card kid2kid.getActiveGiftCards() = 0); -- GiftCards gets
deactivated.
		assertTrue(products(1) in set client.getProductsBought());
		if isAdmin then (
			-- Test sale transactions with GiftCards
			saleProducts := iota x in set kid2kid.getTransactions() &
isofclass(Sale, x) and x.getValue() = gcValue;
			assertTrue(card saleProducts.getGiftCardsUsed() = 0);
			assertTrue(card saleProducts.getGiftCardsSold() = 1);
			saleGC := iota x in set kid2kid.getTransactions() & isofclass(Sale,
x) and x.getValue() = products(1).getSellPrice();
			assertTrue(card saleGC.getGiftCardsUsed() = 1);
			assertTrue(card saleGC.getGiftCardsSold() = 0);
			)
		);

	-- Check store products available and client products bought.
	private testStoreSellProductInCash: () ==> ()
		testStoreSellProductInCash() ==
		(
		assertTrue(products(2) in set store.getProductsAvailable());
		kid2kid.sellProductInCash(products(2), client, cashier, today);
		assertTrue(products(2) not in set store.getProductsAvailable());
		assertTrue(products(2) in set client.getProductsBought());
		);

	-- Check that credit notes of client decrease when store sells a product and
client pays with credit notes
	private testStoreSellProductInCreditNotes: () ==> ()
		testStoreSellProductInCreditNotes() ==
		(
		dcl clientPrevCredit: real :=
store.getCreditNotesOfClient(client.getId());
		assertTrue(clientPrevCredit > 0);
		assertTrue(products(3) in set store.getProductsAvailable());
		kid2kid.sellProductInCreditNotes(products(3), client, cashier, today);
		assertTrue(products(3) not in set store.getProductsAvailable());
		assertTrue(products(3) in set client.getProductsBought());
		assertTrue(store.getCreditNotesOfClient(client.getId()) = clientPrevCredit
- products(3).getSellPrice()); -- credit notes decreased
		);

	-- Edit client name
	private testEditClient: () ==> ()
		testEditClient() ==
		(
		assertTrue(client in set kid2kid.getClients());
		client.setName("123");
		assertTrue("123" = client.getName());
		client.setName("abc");
```

```
        assertTrue("abc" = client.getName());
        );

    -- Get static value of giftcards possible values
    private testGiftCardPossibleValues: () ==> ()
        testGiftCardPossibleValues() ==
        (
        assertTrue(GiftCard`getPossibleValues() = {10, 20, 40});
        );

    -- Test set description of product
    private testProductEdit: () ==> ()
        testProductEdit() ==
        (
        dcl p: Clothing := new Clothing(<New>, "J", 10, 12, <Jeans>);
        assertTrue(p.getDescription() = "J");
        p.setDescription("abc");
        assertTrue(p.getDescription() = "abc");
        );

    -- Test udpate sell price of product
    private testProductPricing: () ==> ()
        testProductPricing() ==
        (
        dcl p: Clothing := new Clothing(<New>, "J", 10, 12, <Jeans>);
        assertTrue(p.getBuyPrice() = 10);
        assertTrue(p.getSellPrice() = 13);
        p.setSellPrice(20);
        assertTrue(p.getSellPrice() = 20);
        );

    -- Test all admin operations
    public testAdminOperations: () ==> ()
        testAdminOperations() ==
            (
            testAddClient();
            testAddStore();
            testAddCashierToStore("Joao");
            testAddCashierToStore("Renato");
            testGetCashiers(2);
            testGetCashierNames({"Joao", "Renato"});
            testGetTransactions(0);
            testRemoveCashier();
            testRemoveClient();
            testStoreBuyProductsInCashAsAdmin();
            testCashierOperations(true);
            );

    -- Test all cashier operations
    public testCashierOperations: (bool) ==> ()
        testCashierOperations(isAdmin) ==
            (
            dcl numProducts: nat := card kid2kid.getProductsAtStore(store);
            numProducts := numProducts + testStoreBuyProductsInCash();
            numProducts := numProducts + testStoreBuyProductsInCreditNotes();
            testGetProductsAtStore(numProducts);
            testStoreGiftCards(isAdmin);
            testStoreSellProductInCash();
            testStoreSellProductInCreditNotes();
            testEditClient();
            );
```

```
	-- Test other miscelanious operations
	public testMiscOperations: () ==> ()
		testMiscOperations() ==
		(
		testGiftCardPossibleValues();
		testProductEdit();
		testProductPricing();
		);

	public static main: () ==> ()
		main() ==
		(
			dcl kid2KidTest: Kid2KidTest := new Kid2KidTest();
			kid2KidTest.loginAdmin();
			kid2KidTest.testAdminOperations();
			kid2KidTest.loginCashier("Porto", "Joao");
			kid2KidTest.testCashierOperations(false);
			kid2KidTest.testStoreBuyProductsInCashAsCashier();
			kid2KidTest.testMiscOperations();
		);
end Kid2KidTest
```