Universität Bremen

Fachbereich 3: Mathematik/Informatik

Online Sales Management System

Design of Information System

Nina Döge

Sevra Umesh

Haleh Vakili-Tahamy

Sommersemester 2017

by

Prof. Dr. Martin Gogolla

Frank Hilken

Table of Contents

1. Introduction

This homework has been developed during the course of the "Design of Information Systems" class at the University of Bremen, which took place in the summer semester of 2017. The goal of this paper is to document the development of a database system with UML (Unified Modeling Language). In this particular case the system to be developed was an Online Sales Management System. The modeling was done in USE (UML based Specification Environment) , a tool for UML, and OCL (Object Constraint Language) which was created and is maintained by the Database Systems group at the University of Bremen.

Online shopping is the act of purchasing products or services over the Internet. Online shopping has grown in popularity over the years, mainly because people find it more convenient to shop from the comfort of their home or office instead of having to travel in order to buy goods. One of the most enticing factors about online shopping, particularly during the holiday seasons, is that it eliminates the need to wait in long lines or spend a lot of time searching in different stores for a particular item. Online shops have many advantages over conventional establishments. Any time of the day you can look at and buy the goods you desire, even on holidays or in the late hours of midnight. Additionally it is not even necessary to leave the house and deal with heavy traffic and/or air pollution. Online shopping saves the client time and trouble, while also making the task of shopping itself easier through the availability of search functions and recommending potentially interesting products based on previous shopping behavior.

An online shop website is used to sell goods or services. And in order to buy these the users of the system need to be able to store the products they are interested in, place orders and be able to review all the products the already bought in the past. The system also needs to account for the employees, which are needed in order to manage the stores roster of available products and categories.

This document describes the structure of the developed system through UML and OCL within the context of USE version 4.2. After a basic description of the systems bare bones functionality, the classes and their relationships within the system are described via Class and Object Diagrams. This is followed by a section describing the systems invariants and operations, paying special attention to the correctness of pre- and postconditions. Then the systems inner consistency is validated through several test scenarios, which test for positive as well as for negative outcomes. In the end the system as a whole is shown in action through several queries and associated Object Diagrams.

2. System Description

The system developed during this course is an online shop, which can be used to sell basically anything. In order to enable this functionality, the system needs to be able to fulfill some basic functions. For example, users need to be able to browse, search, buy and rate products. In order to make buying as easy as possible, it is customary to give the user a virtual shopping cart, in which products can be stored for a while, before being bought as a whole. Each such sale should result in an order being placed at the store. This order will remain active until the delivery of the ordered products to the user is completed.

In addition to this, users should be able to rate and review products that they have already bought in the past. These ratings and reviews can then serve as a means of additional information for other users, which may be interested in purchasing the same product. Another useful feature to ease browsing and aid users in finding the products they desire, would be a category system, where each category can contain multiple products as well as additional categories.

Last but not least all of these products and categories need to be managed, prices of existing products may change, old ones may need to be removed and new ones added instead. For this purpose the online shops employees will need functions that allow for this to happen.

Drawing from this general description, we can extrapolate that the system will most likely need classes for:

- Products
- Categories
- Users
- Shopping Carts
- Employees
- Orders
- Ratings

It is also clear that users will need strong links to their shopping carts and ratings. Products on the other hand will also have ratings, but belong to categories and both products and categories will be managed by employees. Since orders result from the contents of a user’s shopping cart at the moment of buying, it may be reasonable to assume that the two will share some similarities. The same is true for users and employees, as both are persons, but have access to different functionalities within the store.

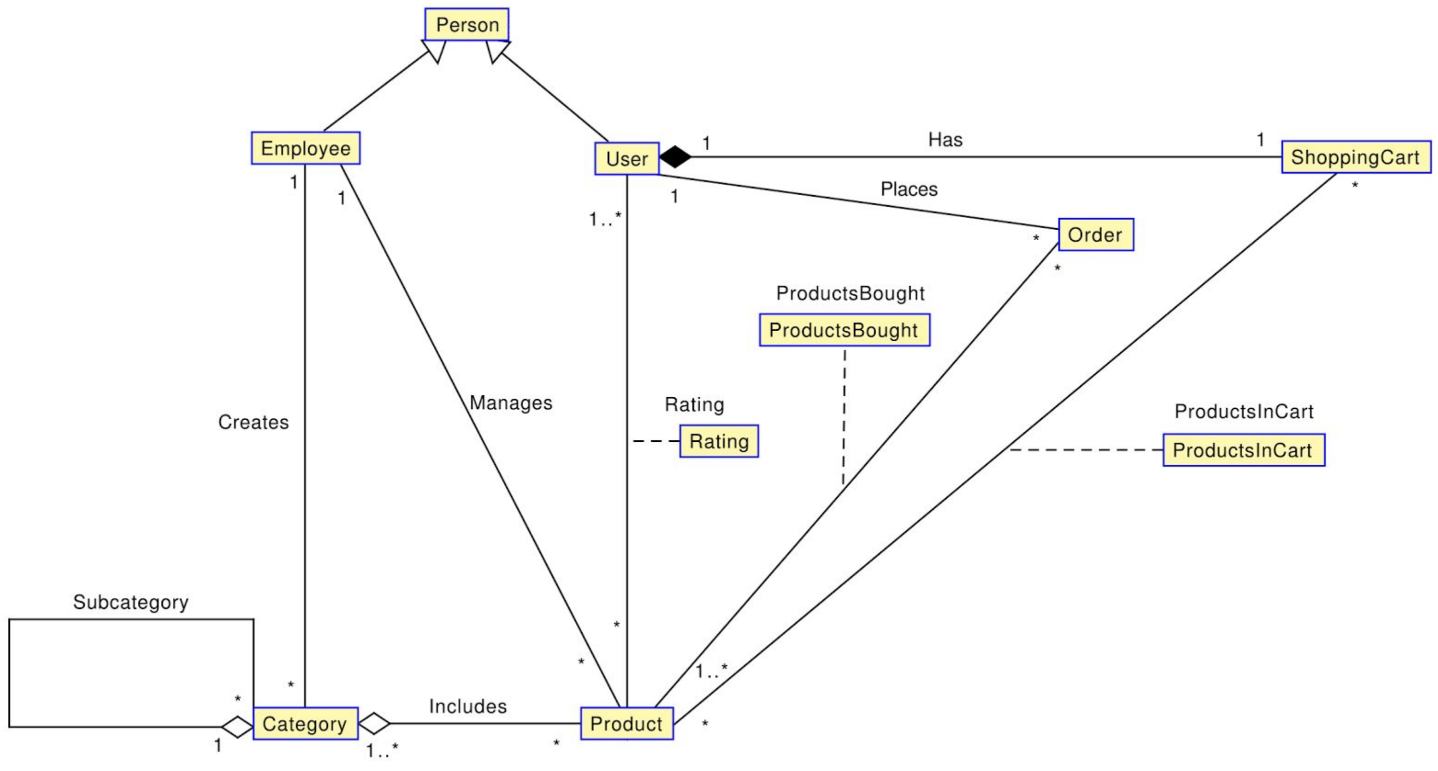
3. Class Structure

This section describes the systems structure on the class level. In various class diagrams, we will show which class realises what functionality and how the relationships between the different classes function. While operations will be named and their functions outlined in this section, for detailed information about their pre- and postconditions as well as their actual implementations you should look at the chapters Invariants and Operations respectively.

3.1 Class Diagrams

As discussed previously in the System Description, the system has the seven classes Employee, User, Product, Category, ShoppingCart, Rating and Order. But a look at the diagram below will reveal that we also introduced three additional classes: Person, ProductsInCart and ProductsBought. The class Person acts as a superclass for Employee and User, all attributes and operations that an instance of Person has, will also be present in any instance of Employee or User.

The two other classes are, like Rating, association classes. ProductsInCart keeps track of all products that are currently within a given ShoppingCart and ProductsBought does the same thing for Order.



The diagram also includes six Associations

1. Category to Category
2. Category to Product
3. Employee to Category
4. Employee to Product
5. User to ShoppingCart
6. User to Order

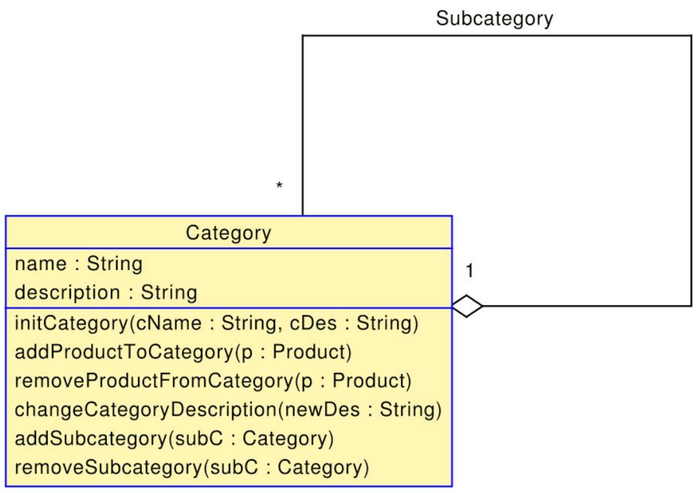
Each User has exactly one ShoppingCart, which can be used to store Products via ProductsInCart, those can be bought, by creating an Order and ProductsBought. Once this has happened a User can create one Rating for each Product bought, but a Product may have many different Ratings from various different Users. Each Product belongs to at least one Category and both Categories and Products are managed (Created,Changed,Destroyed) by Employees, who like Users are Persons and thus inherit attributes from the Person class.

3.2 Class Descriptions

3.2.1 Category

This class models the categories within the online shop, they serve to organize and search products according to purpose. Each category can contain various products as well as additional categories. For example there may be a category “Books”, with the various available genres as subcategories.

Each such subcategory can only belong to one supercategory though.

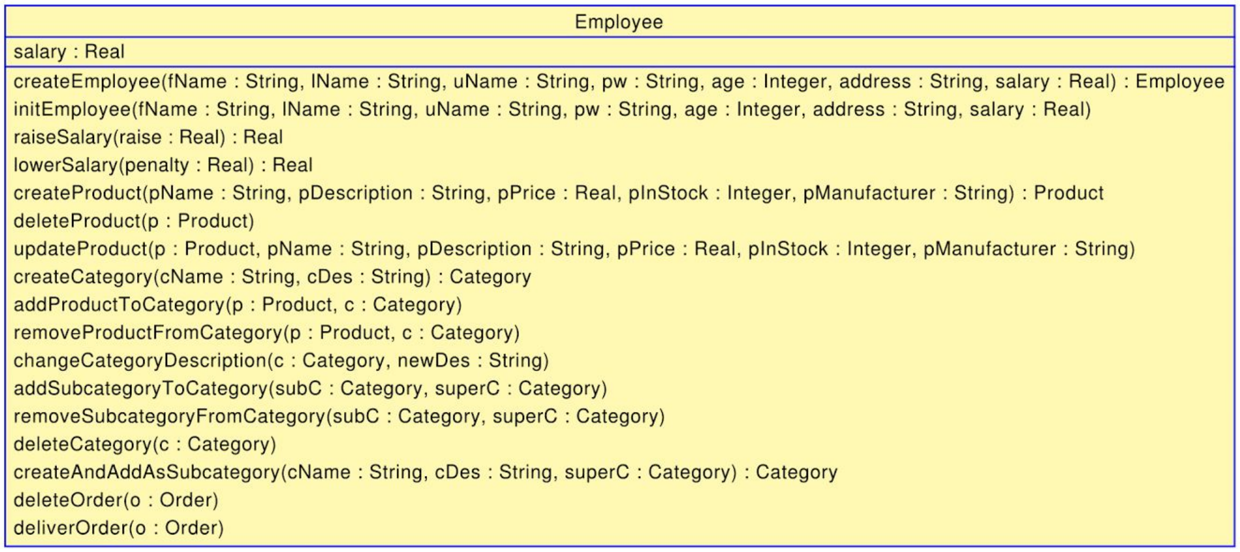


Subcategory
*
1
Category
name : String
description : String
initCategory(cName : String, cDes : String)
addProductToCategory(p : Product)
removeProductFromCategory(p : Product)
changeCategoryDescription(newDes : String)
addSubcategory(subC : Category)
removeSubcategory(subC : Category)

Attributes	
name:String	This is the name of the category.
description:String	The description of the category, ideally this should also give a broad overview about its subcategories.
Operations	
initCategory(cName : String, cDes : String)	This operation initializes a new category object with a name and a description.
addProductToCategory(p:Product)	This operation adds a new product to the category, that was not previously a part of it.
removeProductFromCategory(p:Product)	This operation removes an old product from the category. But it does not delete the product itself from the store.
changeCategoryDescription(newDes: String)	This operation allows the category’s description to be changed.
addSubcategory(subC:Category)	This operation adds a new subcategory to the category.
removeSubcategory(subC:Category)	This operation removes a subcategory from the the category.

3.2.2 Employee

This class represents the people working for the online shop. Like User, this class inherits the attributes and operations from Person, but it also contains additional information relevant to the employee's status as a worker for the store. Since the Employee is responsible for managing the contents of the entire store, the class has many functions that are relevant to this tasks.



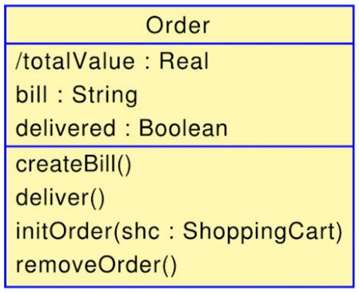
```
Employee
salary : Real
createEmployee(fName : String, lName : String, uName : String, pw : String, age : Integer, address : String, salary : Real) : Employee
initEmployee(fName : String, lName : String, uName : String, pw : String, age : Integer, address : String, salary : Real)
raiseSalary(raise : Real) : Real
lowerSalary(penalty : Real) : Real
createProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String) : Product
deleteProduct(p : Product)
updateProduct(p : Product, pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)
createCategory(cName : String, cDes : String) : Category
addProductToCategory(p : Product, c : Category)
removeProductFromCategory(p : Product, c : Category)
changeCategoryDescription(c : Category, newDes : String)
addSubcategoryToCategory(subC : Category, superC : Category)
removeSubcategoryFromCategory(subC : Category, superC : Category)
deleteCategory(c : Category)
createAndAddAsSubcategory(cName : String, cDes : String, superC : Category) : Category
deleteOrder(o : Order)
deliverOrder(o : Order)
```

Attributes	
Salary:Real	The employee's salary.
Operations	
createEmployee(fName:String,lName:String,uName:String,pw:String,age:Integer,address:String,salary:Real) : Employee	This operation creates and initializes a new Employee instance.
initEmployee(fName:String,lName:String,uName:String,pw:String,age:Integer,address:String,salary:Real)	This operation initializes an empty Employee object.
raiseSalary(raise:Real) :Real	This operation raises the employee's salary.
lowerSalary(penalty:Real) :Real	This operation lowers the employee's salary.

createProduct(pName:String,pDescription:String,pPrice:Real,pInStock:Integer,pManufacturer:String) : Product	This operation creates and initializes a new Product to the store.
deleteProduct(p:Product)	This operation deletes a Product from the store, including all its associations etc.
updateProduct(p:Product,pName:String,pDescription:String,pPrice:Real,pInStock:Integer,pManufacturer:String)	This operation updates a given Product's attributes with new values.
createCategory(cName:String,cDes:String) : Category	This operation creates and initializes a new Category.
addProductToCategory(p:Product,c:Category)	Adds a Product to a Category.
removeProductFromCategory(p:Product,c:Category)	Removes a Product from a Category.
changeCategoryDescription(c:Category,newDes:String)	This operation changes the description of an existing Category.
addSubcategoryToCategory(subC:Category,superC:Category)	This operation adds a subcategory to another Category.
removeSubcategoryFromCategory(subC:Category,superC:Category)	This operation removes a subcategory from another Category.
deleteCategory(c:Category)	This operation deletes a Category from the system.
createAndAddAsSubcategory(cName:String,cDes:String,superC:Category) : Category	This operation creates a new Category and immediately adds it to an existing one as a subcategory.
deleteOrder(o:Order)	This operation deletes an undelivered Order and returns the Products contained within it to the store.
deliverOrder(o:Order)	This operation delivers an undelivered Order.

3.2.3 Order

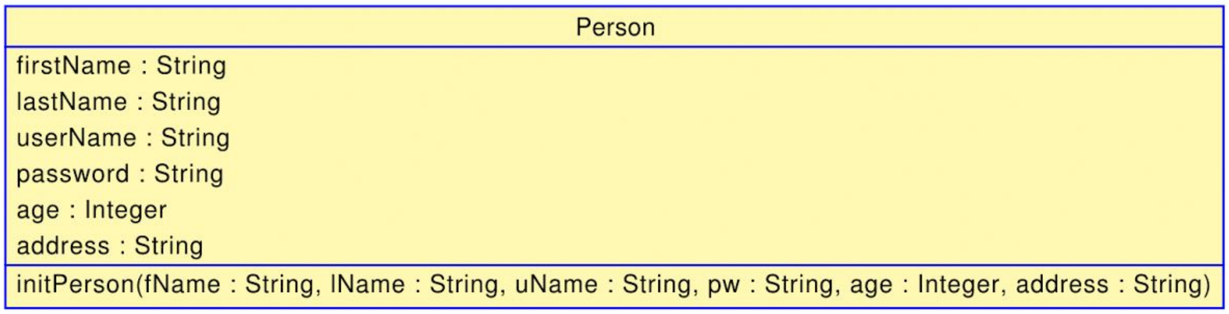
An Order represents Products, which were bought by a given User. It retains the total value as well as the contents of the bill.



Attributes	
totalValue: Real	This derived attribute contains the value of all prices of the Products contained within the Order.
bill: String	This attribute contains all Products and their respective attribute values at the time when the Order was delivered. Even if a Product contained in the bill is removed later on, its original form is preserved in this string of information.
delivered: Boolean	This attribute denotes whether the Order has been successfully delivered or not.
Operations	
createBill()	This operation creates a bill for the Order when it is delivered.
deliver() : Boolean	This operation determines whether the order has been delivered and modifies the delivered attribute accordingly.
initOrder(shc:ShoppingCart)	This operation initializes an empty Order object by giving it the contents of the ShoppingCart that will be bought.
removeOrder()	This operation takes the contents of an undelivered Order and restores the amount of ordered Products back to the Product's respective inStock attribute while removing all it ProductsBought instances.

3.2.4 Person

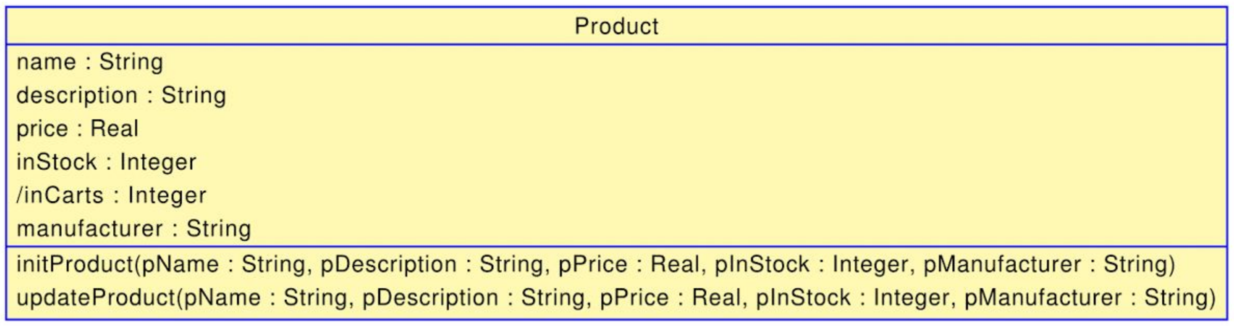
This class holds the common attributes needed to model a Person, it acts as a superclass for Employee and User, which each inherit its attributes and operations.


Person
firstName : String
lastName : String
userName : String
password : String
age : Integer
address : String
initPerson(fName : String, lName : String, uName : String, pw : String, age : Integer, address : String)

Attributes	
firstName:String	This attribute holds the first name of the person.
lastName:String	This attribute describes the last name of the person.
userName:String	This attribute represents the username of the person.
password:String	This attributes holds the password of the person.
Age:Integer	This attribute contains the age of the person.
Address:String	This describes the person's address.
Operations	
initPerson(fName:String,lName:String, uName:String,pw:String,age:Integer,a ddress:String)	This operation initializes an empty Person object. It is used by both the Employee and the User class in their respective init operations to initialize the attributes they both inherit form attributes.

3.2.5 Product

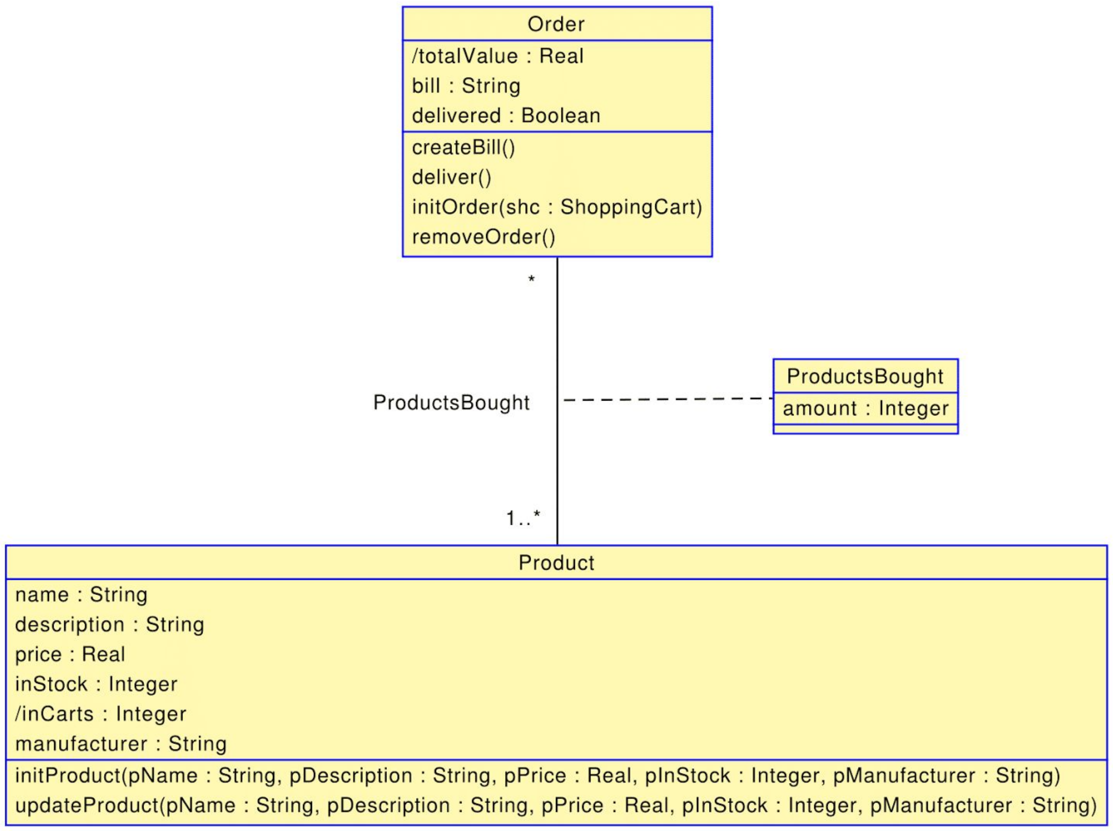
This class models all products within the system. Each product belongs to one or more categories and can be placed in ShoppingCarts and subsequently bought by Users.


Product
name : String
description : String
price : Real
inStock : Integer
/inCarts : Integer
manufacturer : String
initProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)
updateProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)

Attributes	
name : String	It is the name of the product.
description : String	It is the description of the product.
price : Real	It is the price of the product.
inStock: Integer	This shows how many of the product are currently in stock and can be sold.
inCarts: Integer	This derived attribute shows how many of the product are currently placed in ShoppingCarts.
manufacturer: String	This lists the manufacturer of the product.
Operations	
initProduct(pName:String,pDescription:String,pPrice:Real,pInStock:Integer,pManufacturer:String)	This operation initializes an empty Product object.
updateProduct(pName:String,pDescription:String,pPrice:Real,pInStock:Integer,pManufacturer:String)	This operation changes the attribute values of the product.

3.2.6 ProductsBought

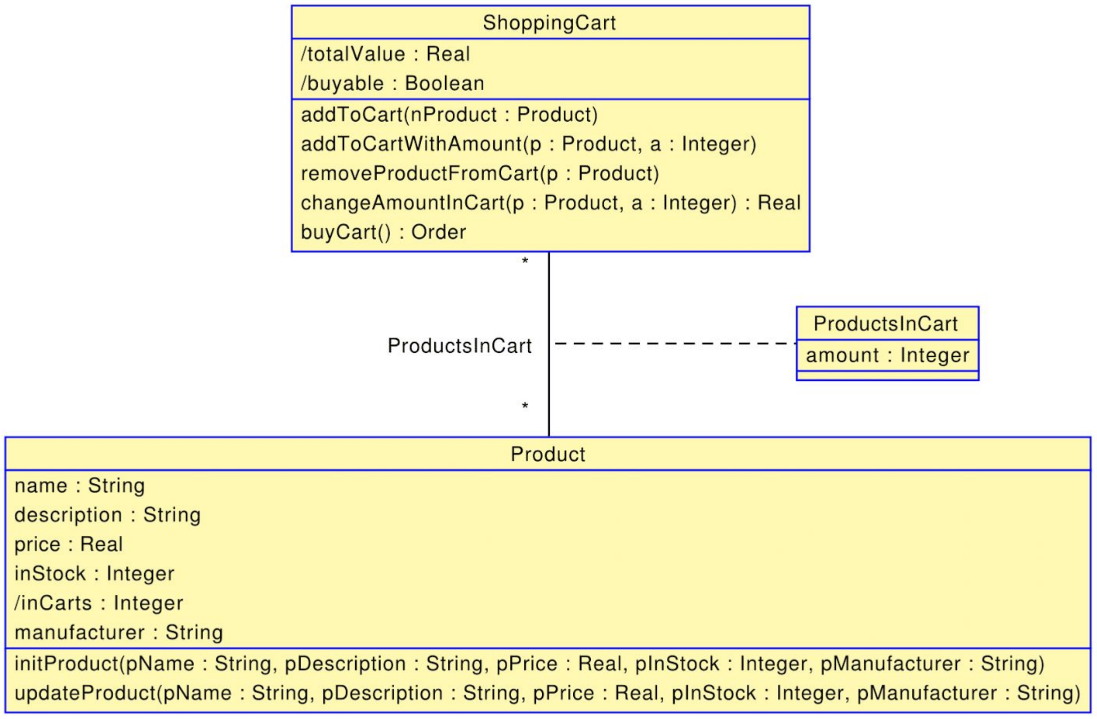
This class manages the Products that a User has already bought and for which an Order has been placed. It exists as an associationclass between Order and Product, wherein each Order contains many Products, which amount is tracked by ProductsBought.


Order
/totalValue : Real
bill : String
delivered : Boolean
createBill()
deliver()
initOrder(shc : ShoppingCart)
removeOrder()
*
ProductsBought
ProductsBought
amount : Integer
1..*
Product
name : String
description : String
price : Real
inStock : Integer
/inCarts : Integer
manufacturer : String
initProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)
updateProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)

Attributes	
amount : Integer	This attribute represents the specific number in which a given product was ordered. For example a user could order the same “Bar of Chocolate” five times, in this case the amount:5 would be stored in ProductsBought for the Product “Bar of Chocolate” for the user who bought it.

3.2.7 ProductsInCart

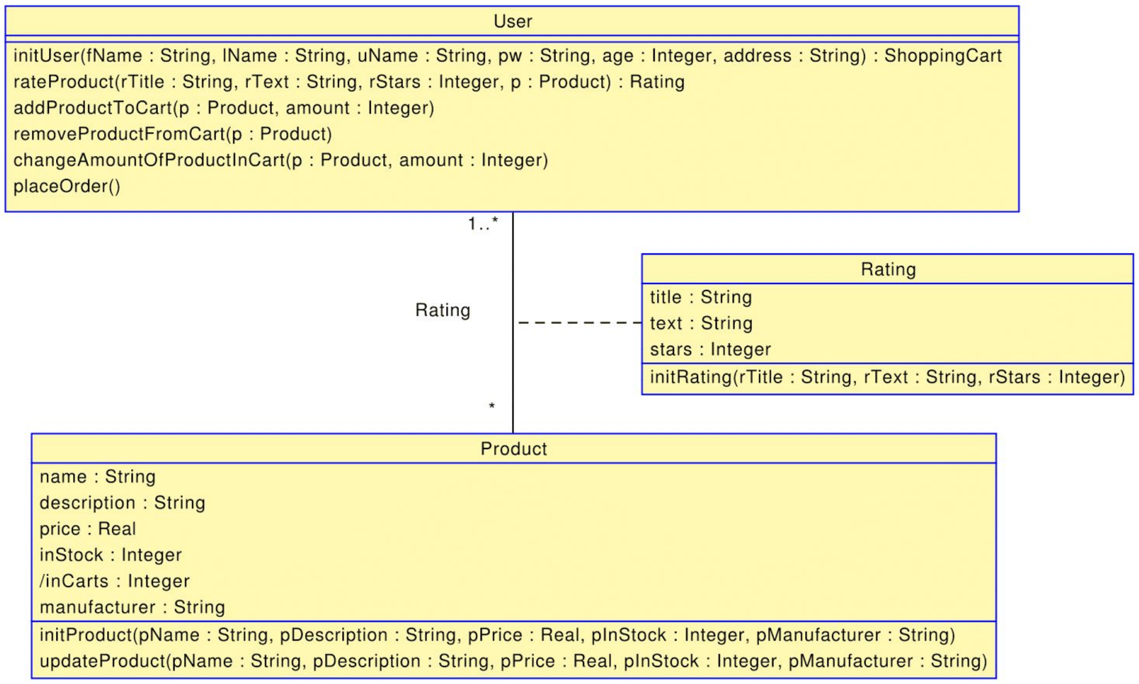
Similar to ProductsBought, this class describes the Products that are currently within a given ShoppingCart. It exists as an associationclass between ShoppingCart and Product and like in ProductsBought, one ShoppingCart can contain many Products, which amount is tracked by ProductsInCart.


ShoppingCart
/totalValue : Real
/buyable : Boolean
addToCart(nProduct : Product)
addToCartWithAmount(p : Product, a : Integer)
removeProductFromCart(p : Product)
changeAmountInCart(p : Product, a : Integer) : Real
buyCart() : Order
*
ProductsInCart
*
ProductsInCart
amount : Integer
Product
name : String
description : String
price : Real
inStock : Integer
/inCarts : Integer
manufacturer : String
initProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)
updateProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)

Attributes	
amount : Integer	This attribute shows the amount in which a given Product is present within the ShoppingCart.

3.2.8 Rating

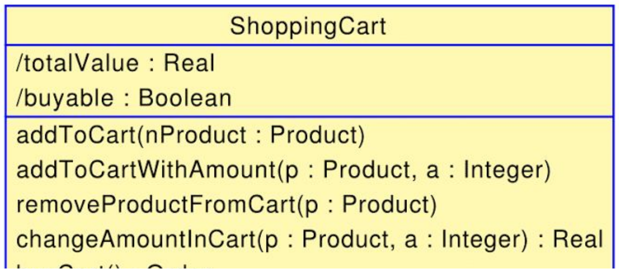
This class represents the review of a given Product by a given User. Ratings serve as an orientation help for Users, as they provide additional information about the Product's quality and usefulness as experienced by Users who already bought the Product in question.


User
initUser(fName : String, lName : String, uName : String, pw : String, age : Integer, address : String) : ShoppingCart
rateProduct(rTitle : String, rText : String, rStars : Integer, p : Product) : Rating
addProductToCart(p : Product, amount : Integer)
removeProductFromCart(p : Product)
changeAmountOfProductInCart(p : Product, amount : Integer)
placeOrder()
1..*
Rating
*
Rating
title : String
text : String
stars : Integer
initRating(rTitle : String, rText : String, rStars : Integer)
Product
name : String
description : String
price : Real
inStock : Integer
/inCarts : Integer
manufacturer : String
initProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)
updateProduct(pName : String, pDescription : String, pPrice : Real, pInStock : Integer, pManufacturer : String)

Attributes	
title : String	This is the title of the review.
text: String	This attribute contains the actual written review of the rating.
stars : Integer	This represents the ratings value between 0 and 5, which will be displayed in the form of stars in the final system.
Operations	
initRating(rTitle:String,rText:String,rStars:Integer)	This operation initializes an empty Rating object.

3.2.9 ShoppingCart

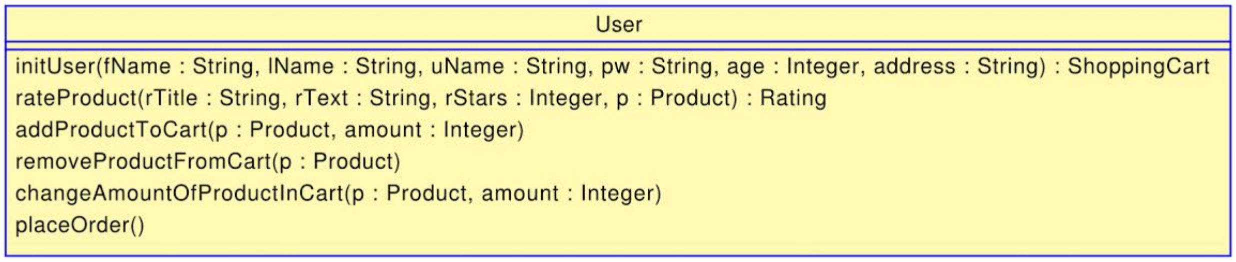
This class represents the shopping cart. User's can add, remove, edit the amount of or purchase Products within the cart. The ShoppingCart shows the Products which were chosen by a given User. Every ShoppingCart belongs to exactly one User. The ShoppingCart also shows the total value of all Products currently in it.


ShoppingCart
/totalValue : Real
/buyable : Boolean
addToCart(nProduct : Product)
addToCartWithAmount(p : Product, a : Integer)
removeProductFromCart(p : Product)
changeAmountInCart(p : Product, a : Integer) : Real

Attributes	
totalValue: Real	This derived attribute contains the total value of the combined prices for all Products that are currently in the cart.
buyable: Boolean	This derived attribute shows whether the cart can be bought in its current form. If it for example contains more of a Product than are actually in stock, this value will be false.
Operations	
addToCart(nProduct:Product)	This operation adds exactly one Product to the cart.
addToCartWithAmount(p:Product,a:Integer)	This operation adds a specific amount of a Product to the cart.
removeProductFromCart(p:Product)	This operation removes a Product from the cart.
changeAmountInCart(p:Product, a:Integer)	This operation changes the amount in which a given Product is present within the cart.
buyCart() : Order	This operation places an Order and empties the ShoppingCart.

3.2.10 User

This class models a User within the online shop, otherwise known as a client or customer. All of the User’s attributes are inherited from Person, but the class defines some unique operations.


User
initUser(fName : String, lName : String, uName : String, pw : String, age : Integer, address : String) : ShoppingCart
rateProduct(rTitle : String, rText : String, rStars : Integer, p : Product) : Rating
addProductToCart(p : Product, amount : Integer)
removeProductFromCart(p : Product)
changeAmountOfProductInCart(p : Product, amount : Integer)
placeOrder()

Operations	
initUser(fName:String,lName:String,uName:String,pw:String,age:Integer,address: String) : ShoppingCart	This operation initializes the User and all its attributes to the system, it also creates the User’s specific ShoppingCart.
rateProduct(rTitle:String,rText:String,rStars:Integer,p:Product) : Rating	This operation creates a Rating for a Product the User has already bought.
addProductToCart(p:Product,amount:Integer)	Adds a specified amount of a given Product to the User’s ShoppingCart.
removeProductFromCart(p:Product)	Removes a Product from the User’s ShoppingCart.
changeAmountOfProductInCart(p:Product,amount:Integer)	Changes the amount of a specific Product within the User’s ShoppingCart.
placeOrder()	Buys the current contents of the User’s ShoppingCart and creates an Order.

3.3 Associations, Roles and Multiplicities

The associations between classes define the relation between objects of these classes in the final system. Each association exists between two classes. Each class in an association has a role and a multiplicity. The role defines the purpose each class serves in the association and the multiplicity describes how many instances of each class will be able to connect in this manner in the final system. There are three types of multiplicities:

- **1** - There exists exactly one instance of this class in this association
- **1..*** - There exist one or more instance(s) of this class in the association
- ***** - There exist zero or more instances of this class in the association

Since an association of any type always exists between two different classes, they also always include two multiplicities, one multiplicity for class A and one multiplicity for class B. For example if A has the multiplicity 1 and B has the multiplicity *, the association is called a one-to-many association, where one instance of A has relations to many instances of B, but each instance of B only has one relation to a single instance of A.

There exist different types of associations within our model: Associations, aggregations and compositions. Associations are the the most plain variant, they merely represent a connection of some kind. An aggregation on the other hand implies hierarchy, a Category can contain several other Categories, but each of these subcategories could again theoretically(not in our model) belong to several different supercategories. Even stricter is the composition, which represents a part-whole relationship. One User has exactly one ShoppingCart and no ShoppingCart can exist without belonging to exactly one User.

In addition to this, association classes also share all the characteristics of plain associations, thus the three associations, resulting in the association classes described in 3.2.6, 3.2.7 and 3.2.8 will also be described further in the following section.

3.3.1 Creates

This association exists between the classes Employee and Category. 1 instance of Employee can create * instances of Category. Each Category is a creation that has one Employee, who acts as its creators.

Multiplicities			
Employee	1	Category	*
Roles			
Employee	creator	Category	category

3.3.2 Has

This composition is between User and ShoppingCart. Each User has exactly 1 ShoppingCart, and every ShoppingCart belongs to exactly 1 User. Since a ShoppingCart exists entirely within the context of the User who owns it, the ShoppingCart is to be viewed as a necessary part of the User.

Multiplicities			
User	1	ShoppingCart	1
Roles			
User	owner	ShoppingCart	cart

3.3.3 Includes

This aggregation connects the classes Category and Product. Each Category can contain any amount of Products, whereas each Product belongs to 1 or more Categories.

Multiplicities			
Category	1..*	Product	*
Roles			
Category	category	Product	product

3.3.4 Manages

This association shows the relation between Employee and Product. Each Employee can manage any number of Products. And every Product will be managed by one Employee during its lifetime.

Multiplicities			
Employee	1	Product	*
Roles			
Employee	manager	Product	product

3.3.5 Places

This association models the behavior between User and Order. Each User can place 0 or more Orders within the system, but each Order belongs to exactly 1 User.

Multiplicities			
User	1	Order	*
Roles			
User	buyer	Order	order

3.3.6 ProductsBought

This associationclass exists between Order and Product. Any Order will include 1 or more Products. But a given Product may show up in any number of Orders, this could be none at all or 3475.

Multiplicities			
Order	*	Product	1..*
Roles			
Order	order	Product	product

3.3.7 ProductsInCart

This association class connects the classes Product and ShoppingCart. A ShoppingCart can contain 0 or more Products and any Product can likewise be present in 0 or more ShoppingCarts.

Multiplicities			
Product	*	ShoppingCart	*
Roles			
Product	product	ShoppingCart	cart

3.3.8 Rating

This associationclass is the result of the dynamic between User and Product. Users can rate any Product that they have already bought, which can be 0 or more. Likewise Products can have any number of Users writing reviews about them

Multiplicities			
User	1..*	Product	*
Roles			
User	Author	Product	ReviewedItem

3.3.9 Subcategory

This aggregation connects Category to itself. Each Category can contain 0 or more other Categories. But every Category can only be included in 1 Supercategory.

Multiplicities			
Category	1	Category	*
Roles			
Category	supercategory	Category	subcategory

3.4 Alternative Modelling Approaches

No model is ever perfect. Some properties of the system could have been modelled differently and still produced similar results. This section details some of these possible alternatives. Each of these alternatives is described on its own, detailing how it would affect the class diagram and possible effects on overall system behaviour. Most alternative choices are based around using different or additional classes or changing the associations and their multiplicities between already existing classes.

3.4.1 User and Employee as separate entities

Currently User and Employee inherit most of their attributes from the Person class. But it would also be possible to model both as totally separate entities. Though their similarities would of course still exist. This might have the advantage that the distinction between Users, who use the system and Employees who work within the system and create its contents is more strictly enforced and thus more obvious.

3.4.2 Multiple Creators

Currently a Product can have only 1 Employee who created it. This is accurate to a degree, but over its lifetime a Product is likely to be edited by several different Employees and a 1..* multiplicity would better reflect this fact. A Product may spend some time being edited on behind the scenes before actually going online in the store. Such editing is usually done by several people and thus it makes sense to credit them.

On the other hand modelling the association as Employees can create * Products, but each Product only has only 1 Employee who created it, creates more orderly responsibilities. This could be more advantageous when working with a large amount of products later on, because if every product has one creator or in-store “owner”, this person is of course the expert on it and should also be in charge of managing the product throughout its lifetime. But if for example this one person who is responsible would stop working for the store, the Product would be orphaned. Thus an association with multiple creators for a Product might be more advantageous in the long run.

3.4.3 Super/Subcategory changes

In the model, a subcategory can only belong to 1 supercategory. But the aggregation would in fact be more flexible, if a subcategory could have 1..* supercategories. This could allow for things like the Category “Books” and the Category “Movies” to share the subcategory “Comedy” for example. It was decided against this, because sharing Categories like this could create mixed search results when only filtering with the subcategory, a User searching books may also be recommended movies, when the search function is not used properly. To prevent such issues, the Categories for different types of Products are kept separated, which is enabled by only allowing 1 supercategory per subcategory.

3.4.4 Super/Subcategory as Composition

Alternatively super and subcategories are currently connected via an aggregation. So why not make the exclusivity stronger by using a composition instead? While this is certainly possible, it was decided against this, because subcategories are whole Categories on their own and a supercategory does not necessarily needs subcategories either. They are not in a true part-whole relationship, so while many subcategories may make up a supercategory, both can exist independently from each other, which makes the aggregation a better choice for the association than a composition would.

3.4.5 The Address class

Currently Address is a simple String attribute of the Person class. But actually an address is a complex entity of its own, containing Country[String], State[String], PostalCode[Integer], City[String], StreetName[String] and HouseNumber[String]. HouseNumber has to be a String, because “numbers like” 34a, 34b and 34c can exist. Handling the Address as a singular entity would allow Users to have multiple Addresses, for example a personal and a work address. With such an Address class, the model would also need to be adjusted in other places. For example each Order would need an attribute DeliveryAddress[Address], which would need to be set by the User before actually placing the Order as a pending delivery. The Address class would be connected via an Aggregation to User, each Address can belong to 1..* Users, because several people can live in the same place and vice versa each User can have 1..* Addresses, like in the example above a personal and a work address.

4. Invariants

In the modeling process, invariants are a very helpful concept to express global constraints that need to be satisfied in the system at all times and may never be violated. There also exist pre- and postconditions, which instead have to be fulfilled immediately prior and directly after the execution of a specific operation. A pre condition is something that must be true before the use case can be invoked. e.g. a precondition for a use case "Buy Book on WebSite" might be "The user has accessed the website and wants to buy a book". A postcondition is something that must be true after the use case is finished. e.g. The user has successfully purchased a book.

An invariant is something that must always be true within every possible state the system can take. e.g. The user's ID matches an ID from the Users table in the database and each User always has a unique username.

For this chapter our invariants will be modelled in OCL. This allows for the precise formulation of invariants while staying fully independent from a definitive implementation.

4.1 Class Category

This chapter describes the invariants for the Category class.

4.1.1 Name of a Category

Each Category must have a name.

```
context Category inv categoryHaveName:
      self.name <> ''
```

4.1.2 Products within a Category are unique

Each Product may only be included within a Category once.

```
context Category inv uniqueProductInCategory:
      Category.allInstances->forAll(p| p.product->isUnique(name))
```

4.2 Class Employee

This chapter describes the invariants for the Employee class.

4.2.1 Employee Must have a salary

Each employee should have a salary more than zero

```
      context Employee inv mustHaveSalar
            Employee.allInstances->forAll(e| e.salary >0)
```

4.3 Class Order

This chapter describes the invariants for the Order class.

4.3.1 Amount of each Product in an Order greater than zero

If there is a Product in Order, its amount has to be greater than zero.

```
      context Order inv productinOrderNotZero:
            self.productsBought->forAll(o| o.amount >0)
```

4.3.2 Enough Product in Stock

If there is a Product in an Order, there needs to be a greater or equal amount of said Product in stock.

```
      context Order inv enoughInStock:
            self.productsBought->forAll( o| if o.order.delivered = false then
            o.product.inStock >= o.amount else 1=1 endif )
```

4.3.3 No duplicate Products in an Order

Each Product in an Order has to be unique, if multiple instances of the same Product are to be purchased, the amount has to be increased instead.

```
context Order inv notDuplicateOrderProduct:
            Order.allInstances->forAll(p| p.product->isUnique(name))
```

4.4 Class Person

This chapter describes the invariants for the Person class, which thus also apply to Employee and User.

4.4.1 Unique Username

All Persons must have unique username because we have to identify each individual user. Also having non-unique usernames would complicate the login procedure, especially if Employees and Users could have the same username.

```
context Person inv uniqueUserName:
      Person.allInstances->isUnique(userName)
```

4.4.2 Name, Family and Username of User

Each Person must have a first name, a last name and a username and password and address. The first and last name are relevant for creating a valid address for delivery and the username of course is the unique handle by which the system identifies its users.

```
context Person inv haveNameAndfamily:
      self.firstName <> '' and
      self.lastName <> '' and
      self.userName <> '' and
      self.password <> '' and
      self.address <> ''
```

4.5 Class Product

This chapter describes the invariants for the Product class.

4.5.1 Price of Product

Each Product needs to have a price that is greater than zero.

```
context Product inv priceNotZero:
      self.price> 0
```

4.5.2 Each Product must have a name

Each product must have a name. Without a name, the Product can not be directly searched in the search-engine and also is less identifiable for the User, which lowers the usability of the store.

```
context Product inv productHaveNamedescInstock:
      name <> '' and
      description <> '' and
      manufacturer <> '' and
      inStock >=0
```

4.6 Class ShoppingCart

This chapter describes the invariants for the ShoppingCart class.

4.6.1 No duplicate Products in the Shopping Cart

In each ShoppingCart, each Product has to be unique, if multiple instances of the same Product are added to the same ShoppingCart, the amount has to be increased instead of adding a new instance of the already contained Product.

```
context ShoppingCart inv notDuplicateCartProduct:
      ShoppingCart.allInstances->forAll(p| p.product->isUnique(name))
```

4.6.2. Buyable ShoppingCart

A ShoppingCart may only be buyable if all the Products contained within it are actually in stock.

```
context ShoppingCart inv buyableShoppingcart:
      self.productsInCart->forAll(pin |  pin.product.inStock < pin.amount
      implies self.buyable = false)
```

4.6.3 Amount of each Product in a ShoppingCart

If there is a Product in a ShoppingCart, its amount in the ShoppingCart has to be greater than zero.

```
context ShoppingCart inv productinSchCNotZero:
      self.productsInCart->forAll(o| o.amount >0)
```

4.7 Class User

This chapter describes the invariants for the User class.

4.7.1 Users can only rate products that they already bought

All Users can create ratings for Products. But they can only rate Products that they already bought and delivered in the past and which are thus present in the ProductsBought instances associated with the User.

```
context User inv rateForBought:
      self.reviewedItem-> forAll(P |
      self.order->select(o|o.delivered).product->includes(P))
```

4.7.2 Each User has exactly one ShoppingCart

For every User object there has to exist exactly one ShoppingCart object.

```
context User inv uniqueShoppingcart:
      User.allInstances->forAll(u| u.cart->size=1)
```

5. Operations

Operations create the actual main functionalities of a system. They are used to create, change and destroy objects, thus manipulating the data within the system. This creates the dynamic behaviors necessary for any kind of software application.

In the following section the operations of each class of the model will be described in detail, including their interactions with each other. This is important because in this model an operation from class Employee might instruct the class Product to execute one of its operations etc. Thus there exists significant interplay between the operations of various different classes.

The operations are sorted depending on which class they belong to. That means one subsection covers all operations for a specific class. Each operation will be described with is name, parameters, return values and pre- and postconditions. The pre- and postconditions will also be presented in their actual implementation and the exact working of the each operation will also be described.

All classes have an init operation, which will be said to initialize the attributes of a given object. This is technically incorrect, as all attributes of every class do have an automatic init or derived attached to them which means that they are never left undefined, even in newly created objects. We still chose the prefix init for describing the operations which are used to give sensible values to an object's attributes for the first time and thus describe this process as 'initializing' in the following chapter.

5.1 Class Category

The Category class contains operation for managing Category data as well as operations for the creating, changing and deleting of subcategories. It also provides operations for adding or removing Products to and from Categories.

The Category class belongs to the basic classes within the system, as its operation relate entirely to its own instances. Category operations pertain always to a specific Category object, they never instruct operations of another class to do anything.

5.1.1 initCategory

This operation initializes the attributes for a new Category object. It is intended to be called after a new Category object was created. A new Category will execute initCatgeory and thus set its name and description attributes.

```
!new Category

!Category1.initCategory('Books','Find something to read!')
```

Parameters	
cName : String	This string contains the name of the Category.
cDes : String	This string contains the description of the Category.
Preconditions	
NoDuplicateCategory	`not Category.allInstances->exists(c | c.name = cName and c.description = cDes)`
	This precondition checks whether the initialization would create a duplicate of an existing Category, if it would, the operation cannot be executed.
NoEmptyParameters	`cName <> '' and cDes <> ''`
	There may be no empty arguments given to the operation.
CategoryIsEmpty	`self.name = '' and self.description = ''`
	The Category has to be in its newly created empty state.
Postconditions	
UniqueCategoryName	`Category.allInstances->isUnique(name)`
	This postcondition ensures that even after the initialization all Category.
CategoryAttributesSet	`self.name = cName and self.description = cDes`
	The attributes of the Category object now have the values that were passed into the operation.

5.1.2 addProductToCategory

This operation adds a Product to the Category, by establishing a new association between the Category object and a given Product object that is not already included in the Category. This is handled as a simple insert operation for the Includes association (3.3.3) between the Category and the Product.

Parameters	
p : Product	The Product that will be added to the Category.
Preconditions	
ProductExists	`Product.allInstances->includes(p)`
	This precondition ensures that the Product to be added exists.
ProductNotInCategory	`self.product->excludes(p)`
	This precondition checks, whether the Product is already contained within the Category.
Postconditions	
ProductInCategory	`self.product->includes(p)`
	This postcondition ensures that the Product was successfully added to the Category.
AmountProductInCategory	`self.subcategory->forAll(c1 | (self.subcategory->closure(subcategory).product->size + self.product->size) >= (c1.subcategory->closure(subcategory).product->size + c1.product->size) )`
	This postcondition ensures that the Category contains more or the same amount of Products as all of its subcategories.

5.1.3 removeProductFromCategory

This operation removes a specific Product from the Category by destroying the existing association between the Category object and the Product object. It executes a delete on the Includes association between the two objects.

Parameters	
p : Product	The Product that will be removed from the Category.
Preconditions	
ProductExists	`Product.allInstances->includes(p)`
	This precondition ensures that the Product to be added exists.
ProductInCategory	`self.product->includes(p)`
	This precondition ensures that the Product that will be removed is currently contained within the Category.
Postconditions	
ProductNotInCategory	`Category.allInstances->forAll( c | c.product ->excludes( p))`
	This postcondition ensures that the Product is no longer included in the Category after the operation has been executed.
AmountProductInCategory	`self.subcategory->forAll(c1 | (self.subcategory->closure(subcategory).product->size + self.product->size) >= (c1.subcategory->closure(subcategory).product->size + c1.product->size)`
	This postcondition ensures that the Category contains more or the same amount of Products as all of its subcategories.

5.1.4 changeCategoryDescription

This operation updates the description of the Category by setting the description attribute.

Parameters	
newDes : String	This string contains the Category's new description.
Preconditions	
NotMyDes	`self.description <> newDes`
	The new description has to be different from the old description or the operation will not be executed.
NoEmptyNewDes	`newDes <> ''`
	The new description may not be empty.
Postconditions	
ChangedDesc	`self.description = newDes`
	Assures that the operation was successful and the Category's description has changed as intended.

5.1.5 addSubcategory

This operation adds a given Category as a subcategory to the Category, by creating a new Subcategory aggregation (3.3.9) between the two Categories. The Category executing the operation will be the supercategory, while the one given as an argument becomes the subcategory.

Parameters	
subC : Category	The Category that will be added as a subcategory.
Preconditions	
SubcategoryExists	`Category.allInstances->includes(subC)`
	This precondition checks whether the intended subcategory actually exists.

SubcategoryIsNotSubCategoryOfThis Category	`self.subcategory->excludes(subC)`	
	The Category that is to be added as a subcategory may not already be a subcategory to this Category.	
Postconditions		
SubcategoryIsSubcategoryOfThisCat egory	`self.subcategory->includes(subC)`	
	This postcondition checks that the added Category is now a subcategory of the Category that executed the operation.	

5.1.6 removeSubcategory

This operation removes a subcategory from the Category that executes the operation. This is achieved by deleting the Subcategory aggregation between the two Category objects.

Parameters	
subC : Category	The Category that is to be removed from the Subcategory aggregation with this Category.
Preconditions	
SubcategoryExists	`Category.allInstances->includes(subC)`
	The subcategory that is to be removed has to exists in order for the operation to work.
SubcategoryIsSubcategoryOfThisCat egory	`self.subcategory->includes(subC)`
	In order to be removed the Category that shall be removed has to be a subcategory of this Category.
Postconditions	
SubcategoryIsNotSubcategoryOfThis Category	`self.subcategory->excludes(subC)`
	After the operation has concluded, the Category that was supposed to be removed, may no longer be a subcategory of this Category.

5.2 Class Employee

The Employee class is a very important class for the system, because Employees can create, change and destroy most of the other objects in it. The Employee class has operations for creating and changing Employees, but also manages Categories, Products and Orders.

Most operations of the Employee class are delegations. For example all operations that work with Categories, tell a specific Category object how it should change its data, thus the Employee class can be understood as issuing commands to many of the other classes in the system.

Very important is the fact, that Employee's have operations for delivering and deleting Orders. Orders can be delivered, when they are undelivered but meet all other requirements to be delivered, like containing more than zero Products for example. They can be deleted by an Employee if they are undelivered, but are not supposed to or cannot be delivered for some reason, for example an Order that a User no longer wants, or an Order which contained dangerous or faulty Products that had to be removed from the system and turned the Order into an empty Order.

5.2.1 createEmployee

With this function an Employee can created a new additional Employee. The operation creates a new Employee object and then calls the initEmployee operation (5.2.2), which in turn relies on the initPerson operation of the Person class (5.4.1) to initialize all the attributes of the newly created, empty Employee object.

Parameters	
fName : String	The firstname of the Employee.
lName : String	The lastname of the Employee.
uName : String	The username of the Employee.
pw : String	The password of the Employee.
age : Integer	The age of the Employee.
address : String	The address of the Employee
salary : Real	The salary of the Employee.
Return Value	
Employee	The operations returns the newly created Employee object, so it can be evaluated in the postconditions.

Preconditions	
PositiveSalary	`salary > 0`
	The proposed salary for the Employee has to be greater than zero.
NoEmptyParameters	`fName <> '' and lName <> '' and uName <> '' and pw <> '' and age > 0 and address <> ''`
	There may be no empty arguments given to the operation.
Postconditions	
EmployeeExists	`Employee.allInstances->includes(result)`
	After the operation has been carried out, there needs to exist an Employee object that matches to the one created by this operation.

5.2.2 initEmployee

This operation initializes the empty Employee object. Especially it handles the setting of the Employee's salary, while the rest of the initialization task is related to the Person class's initPerson operation (5.4.1) through executing the following SOIL statement:

```
self.initPerson(fName,lName,uName,pw,age,address)
```

Parameters	
fName : String	The firstname of the Employee.
lName : String	The lastname of the Employee.
uName : String	The username of the Employee.
pw : String	The password of the Employee.
age : Integer	The age of the Employee.
address : String	The address of the Employee
salary : Real	The salary of the Employee.

Preconditions	
PositiveSalary	`salary > 0`
	The proposed salary for the Employee needs to be greater than zero.
Postconditions	
SalarySetCorrectly	`self.salary = salary`
	After the operation has concluded, the Employee's salary has to be a positive number.

5.2.3 raiseSalary

This operation raises the salary of the Employee by setting the salary attribute.

Parameters	
raise : Real	The amount by which the Employee's salary shall be raised.
Return Value	
Real	This value represents the former salary of the Employee.
Preconditions	
RaiseGreaterThanZero	`raise > 0`
	The argument given for the raise parameter has to be positive.
Postconditions	
SalaryMoreThanZero	`self.salary > 0`
	After the operation has concluded, the salary of the Employee has to be greater than zero.

SalaryIncreased	`self.salary > result`
	When compared to the Employee's previous salary, the current salary has to be higher.

5.2.4 lowerSalary

This operation decreases the salary of the Employee who executes it, by setting the salary attribute.

Parameters	
penalty : Real	The amount by which the Employee's salary should be decreased.
Return Value	
Real	This value preserves the salary the Employee had, before the operation was executed.
Preconditions	
PenaltyGreaterThanZero:	`penalty > 0`
	The penalty has to be given as a positive amount, as the operation uses subtraction to decrease the salary.
Postconditions	
SalaryMoreThanZero	`self.salary > 0`
	After the operation has been carried out, the Employee's salary still has to be greater than zero.
SalaryDecreased	`self.salary < result`
	When compared to the Employee's previous salary, the current salary has to be lower.

5.2.5 createProduct

With this operation an Employee can create a new Product. Once such a Product is created, a new Manages association (3.3.4) is established between it and the Employee who executed the operation. Afterwards, the Product object is instructed to execute its initProduct operation with the values given to it by the Employee (5.5.1).

Parameters	
pName : String	The name for the Product.
pDescription : String	The description for the Product.
pPrice : Real	The price for the Product.
pInStock : Integer	The amount of the Product that is in stock.
pManufacturer : String	The manufacturer of the Product.
Return Value	
Product	The Product that was created and initialized.
Preconditions	
NoDuplicateProduct	`not Product.allInstances->exists(p | p.name = pName and p.description = pDescription and p.price = pPrice and p.manufacturer = pManufacturer)`
	The arguments given to the operations may not correspond to an already existing Product.
Postconditions	
ProductExists	`Product.allInstances->includes(result)`
	The created Product exists.
CreatedByEmployee	`self.product->includes(result)`
	The created Product has an association to the Employee object that executed the operation.

5.2.6 deleteProduct

This operation deletes a Product object from the system by destroying it. This also removes all associations the Product was a part of.

Parameters	
p : Product	The Product that shall be deleted from the system.
Preconditions	
TheProductExists	`Product.allInstances->includes(p)`
	The Product has to exist.
Postconditions	
ProductNoLongerExists	`Product.allInstances->excludes(p)`
	The Product that is given as an argument no longer exists within the system.

5.2.7 updateProduct

This operation updates all attributes of a given Product by prompting the Product to execute its updateProduct operation (5.5.2), which sets the Product's attributes to the new values that are given to the operation as arguments.

Parameters	
p : Product	The Product that will be updated.
pName : String	The name for the Product.
pDescription : String	The description for the Product.
pPrice : Real	The price for the Product.
pInStock : Integer	The amount of the Product that is in stock.
pManufacturer : String	The manufacturer of the Product.

Preconditions	
TheProductExists	`Product.allInstances->includes(p)`
	The Product has to exist.
ValuesChange	`pName <> p.name or pDescription <> p.description or pPrice <> p.price or pInStock <> p.inStock or pManufacturer <> p.manufacturer`
	At least one of the argument values has to be different from the current value of the attribute.
NoDuplicateProduct	`not Product.allInstances->exists(p | p.name = pName and p.description = pDescription and p.price = pPrice and p.manufacturer = pManufacturer)`
	The argument values create a unique Product.

5.2.8 createCategory

This operation allows the Employee to create a new Category. Once the Category is created, it is inserted into a new Creates association (3.3.1) with the Employee object. Afterwards it is instructed to execute its initCategory operation (5.1.1) and is saved as the result value for the operation.

Parameters	
cName : String	The name of the Category.
cDes : String	The description of the Category.
Return Value	
Category	The newly created Category.
Preconditions	
NoEmptyAttributes	`cName <> '' and cDes <> ''`
	The arguments given to the operation may not be empty.

Postconditions	
EmployeeCreatedCategory	`self.category->includes(result)`
	There exists an association between the Employee and the created Category.
CategoryIsCreated	`result.oclIsNew()`
	The created Category is a newly created object.
CategoryType	`result.oclIsTypeOf(Category)`
	The created Category is a Category.

5.2.9 addProductToCategory

This operation adds a Product to a Category, by instructing the Category to execute its addProductToCategory operation (5.1.2) with the chosen Product as the argument.

Parameters	
p : Product	The Product that should be added to a Category.
c : Category	The Category that a Product should be added to.
Preconditions	
CategoryExists	`Category.allInstances->includes(c)`
	The Category has to exist.
ProductExists	`Product.allInstances->includes(p)`
	The Product has to exist.
ProductNotInCategory	`c.product->excludes(p)`
	The Product is not yet included in the Category.
Postconditions	
ProductIsInCategory	`c.product->includes(p)`
	The Category includes the Product.

5.2.10 removeProductFromCategory

This operation removes a Product from a Category, by instructing the Category to execute its removeProductFromCategory operation (5.1.3) with the chosen Product as the argument.

Parameters		
p : Product	The Product to be removed from a Category.	
c : Category	The Category that a Product should be removed from.	
Preconditions		
CategoryExists	`Category.allInstances->includes(c)`	The Category has to exist.
ProductExists	`Product.allInstances->includes(p)`	The Product has to exist.
ProductIsInCategory	`c.product->includes(p)`	The Category includes the Product.
Postconditions		
ProductNotInCategory	`c.product->excludes(p)`	The Product is no longer included in the Category.

5.2.11 changeCategoryDescription

This operation allows an Employee to change a Category's description. It instructs a Category to execute its changeCategoryDescription operation (5.1.4). It has no postconditions, as the changes to the Category are already controlled by the postconditions of the corresponding operation in the Category class.

Parameters	
c : Category	The Category which description should change.
newDes : String	The new description for the Category.
Preconditions	
CategoryExists	`Category.allInstances->includes(c)`
NewDescriptionNotEmpty	`newDes <> ''`
	The new Description may not be empty.

5.2.12 addSubcategoryToCategory

This operation adds a subcategory to another Category. The future supercategory is instructed to execute its addSubcategory operation (5.1.5) with the subcategory as its argument.

Parameters	
subC : Category	The subcategory.
superC : Category	The supercategory that will contain the subcategory.
Preconditions	
SupercategoryDoesExist	`Category.allInstances->includes(superC)`
	The subcategory Category has to exist.
SubcategoryDoesExist	`Category.allInstances->includes(subC)`
	The supercategory Category has to exist.

SubcategoryIsNotSubcategoryOfSupercategory	`superC.subcategory->excludes(subC)`
	The subcategory is not already a subCategory of the supercategory.
Postconditions	
SubcategoryIsSubcategoryOfSupercategory	`superC.subcategory->includes(subC)`
	Afterwards the subcategory is in a Subcategory aggregation to the supercategory.

5.2.13 removeSubcategoryFromCategory

This operation removes an existing subcategory from another Category. This is achieved by instructing the supercategory to execute its removeSubcategory operation (5.1.6).

Parameters	
subC : Category	The subcategory that will be removed.
superC : Category	The supercategory that contains the subcategory.
Preconditions	
SupercategoryDoesExist	`Category.allInstances->includes(superC)`
	The subcategory Category has to exist.
SubcategoryDoesExist	`Category.allInstances->includes(subC)`
	The supercategory Category has to exist.
SubcategoryIsSubcategoryOfSupercategory	`superC.subcategory->includes(subC)`
	The Category given as subC has to be a subcategory to the Category given as superC.
Postconditions	
SubcategoryIsNotSubcategoryOfSupercategory	`superC.subcategory->excludes(subC)`
	The Category given as subC no longer is a subcategory of the category given as superC.

5.2.14 deleteCategory

This operation allows an Employee to delete a Category by calling destroy() on it. This also removes all of the associations the Category was a part of, and Products that only belonged to this Category may end up without a Category afterwards.

Parameters	
c : Category	The Category that will be deleted.
Preconditions	
CategoryExists	`Category.allInstances->includes(c)`
	The Category that is to be deleted has to exists.
Postconditions	
CategoryDoesNotExist	`Category.allInstances->excludes(c)`
	The Category no longer exists.

5.2.15 createAndAddAsSubcategory

This operation allows an Employe to create a new Category and immediately add it as a subcategory to another already existing Category. In the first step, this operation calls the Employee's createCategory operation (5.2.8) and afterwards it executes the addSubcategory operation (5.1.5) of the Category given as superC. Its postconditions are technically redundant.

Parameters	
cName: String	The name of the Category that will be created.
cDes : String	The description of the Category that will be created.
superC : Category	The Category that the newly created category will be added as a subcategory to.
Return Value	
Category	The newly created subcategory Category.

Preconditions	
SupercategoryDoesExist	`Category.allInstances->includes(superC)`
	The Category given as superC has to exist.
Postconditions	
SubCategoryDoesExist	`Category.allInstances->includes(result)`
	The newly created Category has to exist.
SubcategoryIsSubcategoryOfSupercategory	`superC.subcategory->includes(result)`
	The newly created Category is now a subcategory to the Category given as superC.

5.2.16 deleteOrder

The Employee can delete undelivered Orders. This operation also allows to remove Orders, that ended up empty due to Product deletion, from the system. When executed the operation calls the removeOrder operation (5.3.4) on the Order that will be deleted, before destroying it via destroy().

Parameters	
o : Order	The Order that shall be deleted.
Preconditions	
OrderExists	`Order.allInstances->includes(o)`
	The Order given in the argument has to exist.
OrderNotDelivered	`o.delivered = false`
	The Order has to be undelivered.
Postconditions	
OrderIsDeleted	`Order.allInstances->excludes(o)`
	The Order no longer exists.

5.2.17 deliverOrder

This operation allows an Employee to deliver an Order by executing a given Order's deliver operation (5.3.2).

Parameters	
o : Order	The Order that shall be delivered.
Preconditions	
OrderExists	`Order.allInstances->includes(o)`
	The Order given in the argument has to exist.
OrderNotDelivered	`o.delivered = false`
	The Order has to be undelivered.
Postconditions	
OrderDelivered	`o.delivered = true`
	The Order has to be delivered.

5.3. Class Order

Orders are created whenever a User buys the contents of their ShoppingCart. Orders serve as means for keeping track of purchases, determining which Products a User may rate and possibly for predicting future shopping behaviour.

All operations of the Order class pertain to itself.

5.3.1 createBill

This operation will produce a bill for the Order. The bill lists all Products contained in the Order and all their attributes in a single String that is saved as the Order's bill attribute. This ensures that Orders keep information, even if the Product they ordered no longer exists in the shop. Because if a Product is deleted from the system, it is fully destroyed, including all associations that the Product was a part of, this means such Products are removed from ProductsBought and are no longer included in the totalValue calculation for any Order, essentially falsifying these values.

Preconditions	
HasNoBill	`self.bill = ''`
	The Orderdont have any bill.
Postconditions	
HasBill	`self.bill <> ''`
	Bill is not empty.The bill lists all Products contained in the Order

5.3.2 deliver

This operation sets the delivered attribute on the Order to true after creating a bill for the Order by executing its createBill operation (5.3.1).

Preconditions	
HasProducts	`self.productsBought->size > 0 and self. productsBought->forAll(p | p.amount > 0)`
	In order for an Order to be deliverable, it has to contain one or more Products.
OrderIsNotDelivered	`self.delivered = false`
	The Order has to be undelivered.
Postconditions	
IsDelivered	`self.delivered = true`
	Afterwards the Order has to be delivered.

5.3.3 initOrder

This operation initializes a new, empty Order with the contents of a ShoppingCart.

First it creates a new Places association (3.3.5) between the User who owns the ShoppingCart and itself. Afterwards it iterates over all Products in the ShoppingCart.

For each Product in the ShoppingCart, the Product's inStock amount is reduced by the amount in which it is ordered and it is inserted into a ProductsBought associationclass (3.3.6) between the Order and itself. Once this is done the operation sets the amount attribute for the ProductsBought association class for each Product based on the amount attribute of the ProductsInCart associationclass (3.3.7) for the corresponding Product.

Parameters	
shc : ShoppingCart	The ShopppingCart used to create the Order.
Preconditions	
ExistsShoppingcart	`ShoppingCart.allInstances -> includes(shc)`
	The ShoppingCart given as an argument has to exist.
OrderIsEmpty	`self.productsBought->isEmpty()`
	The Order contains no Products.
CartIsBuyable	`shc.buyable = true`
	The ShoppingCart is buyable.
Postconditions	
OrderNotEmpty	`self.productsBought->size() > 0`
	The Order contains more than zero Products.
OrderHasShoppingCartContents	`self.productsBought.product = shc.productsInCart.product`
	The Products contained in the Order are the same as in the ShoppingCart.

5.3.4 removeOrder

If an order is not delivered , it can be deleted. This operation removes the Order's effect on the system by restoring Products inStock that were affected by it and removing all ProductsBought instances associated with it.

Even empty Orders can be deleted, empty Orders can happen, if a Product that is ordered in an undelivered Order is deleted from the system.

Preconditions	
AmountOfProductsNotNegative	`self.productsBought->size >= 0`
	There may be no negative amount of Products in the Order.
OrderUndelivered	`self.delivered = false`
	Only undelivered Orders may be removed from the system.
NoMoreProducts	`self.productsBought->size = 0`
	The Order contains no more Products.

5.4 Class Person

The Person class serves as the basic template for both the Employee and User classes. The only operation in it is for the initialization of all its attributes , but in a setting where each attribute would have its own independent update setter, these would also be implemented in this class. Since Employee and User inherit from Person, both classes can call upon initPerson as if it were one of their own operations

5.4.1 initPerson

This operation will initialize all attributes of the Person object by setting them to the values passed in as arguments.

Parameters	
fName : String	The firstname of the Person.
lName : String	The lastname of the Person.
uName : String	The username of the Person.
pw : String	The password for the Person’s account.
age : Integer	The age of the Person.
Address : String	The address of the Person.
Preconditions	
PersonNotInitialized	`self.firstName = '' and self.lastName = '' and self.userName = '' and self.password = '' and self.age = 0 and self.address = ''`
	All attributes of the Person have to be in their default state, String are = ‘’ and Integers are = 0.
NoEmptyParameters	`fName <> '' and lName <> '' and uName <> '' and pw <> '' and age <> 0 and address <> ''`
	All arguments have to have a non-default vaue.
AgeNotZeroOrNegative	`age > 0`
	The argument given for age has to be greater than zero.

Postconditions	
UniqueUsername	`Person.allInstances->isUnique(userName)`
	Each Person has to have a unique username.
PersonInitialized	`self.firstName = fName and self.lastName = lName and self.userName = uName and self.password = pw and self.age = age and self.address = address`
	All attributes of the Person have been set to the correct values.

5.5 Class Product

The Product class is very vital for the system, but has only few operations. Namely an init and an update operation, which are only responsible for changing the attribute values of a given Product instance. Each Product can always only change itself.

5.5.1 initProduct

This operation initializes a new, empty Product object by setting all attributes to the new values given to the operation when it was called. It can only be called once.

Parameters	
pName : String	The name of the Product.
pDescription : String	The description of the Product.
pPrice : Real	The price of the Product.
pInStock : Integer	The amount of the Product that is in stock.
pManufacturer : String	The name of the manufacturer of the Product.
Preconditions	
NoDuplicateProduct	`Product.allInstances->forAll(p | p.name<>pName)`
	The new name of the Product may not correspond to the name of any other existing Product.

NoEmptyFields	`pName <> '' and pDescription <> '' and pManufacturer <> ''`	There may be no empty arguments when the operation is called.
StockNotSmallerThanZero	`pInStock >=0`	The new amount in stock has to be greater than zero.
PriceMoreThanZero	`pPrice > 0`	The new price has to be greater than zero.
ProductIsEmpty	`self.name = '' and self.description = '' and self.price = 0 and self.manufacturer = ''`	All values for the Product's attributes have got to be in their initial default states.
Postconditions		
ProductNameIsUnique	`Product.allInstances->isUnique(name)`	The name has still to be unique.
ProductChanged	`self.name = pName and self.description = pDescription and self.price = pPrice and self.manufacturer = pManufacturer`	The Product's attributes have to have been set to the values given in the operations arguments.

5.5.2 updateProduct

This operation updates all attributes of the Product that executes it by setting them.

Parameters	
pName : String	The name of the Product.
pDescription : String	The description of the Product.
pPrice : Real	The price of the Product.
pInStock : Integer	The amount of the Product that is in stock.
pManufacturer : String	The name of the manufacturer of the Product.

Preconditions	
ProductRemainsInStock	`pInStock >=0`
	The amount of the product in stock has to be greater or equal to zero.
NoEmptyFields	`pName <> '' and pDescription <> '' and pPrice <> 0 and pInStock <> 0 and pManufacturer <> ''`
	There may be no empty arguments when the operation is called.
ValuesChange	`pName <> self.name or pDescription <> self.description or pPrice <> self.price or pInStock <> self.inStock or pManufacturer <> self.manufacturer`
	At least one of the arguments given has to be different from the current value of the Product's attribute.
NoDuplicateProduct	`not Product.allInstances->exists(p | p.name = pName and p.description = pDescription and p.price = pPrice and p.manufacturer = pManufacturer)`
	The execution of the operation may not result in a duplicate Product.
Postconditions	
PriceMoreThanZero	`pPrice > 0`
	The price has to be greater than zero.
ProductChanged	`self.name = pName and self.description = pDescription and self.price = pPrice and self.inStock = pInStock and self.manufacturer = pManufacturer`
	The attributes of the Product have to be equal to the arguments given to the operation.

5.6 Class Rating

Rating is an associationclass that exists between User and Product. Users can create one Rating for every they Product bought, independent of the amount they bought a specific type of product in. Ratings can only be created once and are not editable.

5.6.1 initRating

This operation initializes the attributes of a new, empty Rating object with the appropriate values.

Parameters	
rTitle: String	The title of the Rating.
rText : String	The text for the Rating.
rStars : Integer	The number of stars for the Rating.
Preconditions	
RatingIsEmpty	`self.title = '' and self.text = '' and self.stars = Undefined`
	Only empty Rating objects can be initialized.
Postconditions	
RatingIsNotEmpty	`self.title <> '' and self.text <> '' and self.stars <> Undefined`
	After the operation has been executed, the Rating may no longer have empty attributes.
RatingStarsInBounds	`self.stars >= 0 and self.stars <= 5`
	The Rating’s stars have to be a number between 0 and 5. 0 being the lowest possible value and 5 being the highest.

5.7 Class ShoppingCart

The ShoppingCart class keeps track of the Products contained within it. It also determines whether its contents can be bought and how expensive it would be to do so via derived attributes that calculate their values on their own.

Each ShoppingCart belongs to one User and is always either empty and not buyable or filled with various different products that may be bought if all of them are in stock.

5.7.1 addToCart

This function will add a specific product with the default amount of 1 to the ShoppingCart, by creating a new ProductsInCart association between the Product in the operations argument and the ShoppingCart that is executing the operation.

Parameters	
nProduct : Product	The Product that shall be added to the ShoppingCart.
Preconditions	
NoDuplicateproductinShc	`self.product->excludes(nProduct)`
	The Product to be added may not already be contained in the ShoppingCart.
Postconditions	
NewRelationExists	`self.productsInCart->exists(pin	pin.product->includes(nProduct))`
	Checks whether the ProductsInCart association class was created successfully.

5.7.2 addToCartWithAmount

This function will add a given Product with a specified amount to the current ShoppingCart by creating a new ProductsInCart associationclass between the two, which will keep track of the specified amount.

Parameters	
p : Product	The Product that is to be added to the ShoppingCart.
a : Integer	The amount with which the Product will be present in the ShoppingCart.
Preconditions	
AmountGreaterThanZero	`a > 0`
	The amount of the Product has to be greater than zero.
AmountDoesNotExceedInStock	`a <= p.inStock`
	But the amount of the Product may not exceed the number of that Product that are actually in stock.
NoDuplicateproductinShc	`self.product->excludes(p)`
	The Product to be added may not already be contained in the ShoppingCart.
Postconditions	
NewRelationExists	`self.product->includes(p)`
	Checks whether the ProductsInCart association class was created successfully.
CorrectAmount	`self.productsInCart->exists(pic | pic.amount = a and pic.product = p)`
	Checks whether the amount in the ProductsInCart associationclass wa set correctly.

5.7.3 removeProductFromCart

This operation removes a Product from the ShoppingCart, by deleting the corresponding ProductsInCart associationclass via destroy().

Parameters	
p : Product	The Product to be removed from the ShoppingCart.
Preconditions	
ProductIsInCart	`self.product->includes(p)`
	The Product has to be included in the ShoppingCart.
Postconditions	
ProductNotInCart	`self.productsInCart->select(pin | pin.product->includes(p))->isEmpty()`
	The Product is no longer in the ShoppingCart.

5.7.4 changeAmountInCart

This operation updates the amount of a specific Product in the current ShoppingCart. It iterates over the ShoppingCart's associated ProductsInCart until it finds the one that corresponds to the one given as an argument and then proceeds to safe its former amount before changing it.

Parameters	
p : Product	The Product which amount will be changed.
a : Integer	The new amount for the Product.
Return Value	
Real	The former amount of the Product.

Preconditions	
ProductInCart	`self.product->includes(p)`
	The Product has to be already in the ShoppingCart.
AmountGreaterThaneZero	`a > 0`
	The amount given in a has to be greater than zero.
AmountDoesNotExceedInStock	`a <= p.inStock`
	a has to be smaller than the amount in stock.
Postconditions	
AmountHasChanged	`result <> a`
	The amount of the Product has to have changed.

5.7.5 buyCart

This operation will create a new Order with the contents of the ShoppingCart. First it creates a new Order and initializes it with its initOrder operation (5.3.3), afterwards the ShoppingCart removes its ProductsInCart and returns to its empty state.

Return Value	
Order	The newly created Order.
Preconditions	
CartIsBuyable	`self.buyable = true`
	The ShoppingCart has to be buyable.
Postconditions	
ThisOrderDoesExist	`self.owner.order->includes(result)`
	The Order wars created.

ThisCartIsEmpty	`self.productsInCart->size = 0`
	The ShoppingCart was emptied.
ThisCartIsNotBuyable	`self.buyable = false`
	The ShoppingCart is no longer buyable.

5.8 Class User

The User represents clients of the store and thus has operations for interacting with Products, like placing them in and removing them from the User's ShoppingCart, buying the contents of said ShoppingCart as well as creating Ratings for Products that they bought in the past.

5.8.1 initUser

This operation initializes all attributes of a new User object. It creates a new ShoppingCart for the User, before calling the initPerson operation (5.4.1) for the actual attribute initialization.

Parameters	
fName : String	The firstname of the User.
lName : String	The lastname of the User.
uName : String	The username of the User.
pw : String	The password of the User.
age : Integer	The age of the User.
Address : String	The address of the User.
Return Value	
ShoppingCart	The ShoppingCart of the User.
Preconditions	
NoCart	`self.cart = Undefined`
	The User may not have a ShoppingCart before executing this operation.

Postconditions	
HasOneCart	`self.cart->size = 1`
	The User has exactly one ShoppingCart.
CartIsTheCreatedOne	`self.cart = result`
	The ShoppingCart is identical to the one created by the operation.
CartEmpty	`self.cart.product->size = 0`
	The ShoppingCart is empty.
CartValueZero	`self.cart.totalValue = 0`
	The ShoppingCart's total value is zero.
CartNotBuyable	`self.cart.buyable <> true`
	The ShoppingCart is not buyable.

5.8.2 rateProduct

Through this operation the User can create a Rating for an already bought Product. First a new Rating object is created and afterwards it is instructed to execute its initRating operation (5.6.1).

Parameters	
rTitle : String	The title for the Rating.
rText : String	The actual text for the Rating.
rStars : Integer	The Rating's star value.
p : Product	The product to be rated.
Return Value	
Rating	The newly created Rating.

Preconditions		
ProductWasBought	`self.order->select(o | o.delivered).product -> includes(p)`	
	The Product that is to be rated was already bought by the User.	
Postconditions		
RatingBelongsToUser	`self.rating->includes(result)`	
	The new Rating belongs to the User.	
RatingExistsForProduct	`p.rating->includes(result)`	
	The new Rating was created for the product given in p.	
RatingIsCreated	`result.oclIsNew()`	
	The new Rating was newly created.	
RatingType	`result.oclIsTypeOf(Rating)`	
	The new Rating is an instance of the Rating class.	

5.8.3 addProductToCart

This operation allows the User to add a specific amount of a Product to their ShoppingCart. This is achieved by instructing the User's ShoppingCart to carry out its addToCartWithAmount operation (5.7.2).

Parameters	
p : Product	The Product to be added to the User's ShoppingCart.
a : Integer	The amount in which the Product shall be added to the User's ShoppingCart.

Preconditions	
ProductNotInCart	`self.cart.product->excludes(p)`
	The Product is not in the User's ShoppingCart.
AmountIsPositive	`amount > 0`
	The amount for the Product is greater than zero.
AmountNotBiggerThanProductInStock	`amount <= p.inStock`
	The amount of the Product is smaller or equal to the amount of the Product in stock.
Postconditions	
ProductInCart	`self.cart.product->includes(p)`
	The Product was added to the User's ShoppingCart.

5.8.4 removeProductFromCart

With this operation the User can remove a Product from their ShoppingCart. It tells the User's ShoppingCart to execute its own removeProductFromCart operation (5.7.3).

Parameters	
P : product	The Product that is to be removed.
Preconditions	
ProductInCart	`self.cart.product->includes(p)`
	The Product is in the User's ShoppingCart.
Postconditions	
ProductNotInCart	`self.cart.product->excludes(p)`
	The Product is no longer in the User's ShoppingCart.

5.8.5 changeAmountOfProductInCart

This operation calls the User’s ShoppingCart’s changeAmountInCart operation (5.7.4).

Parameters	
p : Product	The Product which amount in the User’s ShoppingCart is to be changed.
amount : Integer	The number to which the Product’s amount in the User’s ShoppingCart should be changed.
Preconditions	
AmountWillChanged	`self.cart.product->select(pr | pr = p).productsInCart->select(pic | pic.cart = self.cart).amount->asOrderedSet()->first() <> amount`
	The amount of the Product in the User’s ShoppingCart will changed.
Postconditions	
AmountWasSetCorrectly	`self.cart.product->select(pr | pr = p).productsInCart->select(pic | pic.cart = self.cart).amount->asOrderedSet()->first() = amount`
	The amount of the Product in the User’s ShoppingCart was set to the value given in amount.

5.8.6 placeOrder

With this operation the User can place an Order for the current contents of the User's ShoppingCart. This is achieved by calling the User's ShoppingCart's buyCart operation (5.7.5).

Preconditions		
ProductsInCart	`self.cart.product->size > 0`	There have to be more than zero Products in the ShoppingCart.
CartIsBuyable	`self.cart.buyable`	The ShoppingCart has to be buyable.
Postconditions		
ShoppingCartIsEmpty	`self.cart.product->size = 0`	The ShoppingCart is empty.
CartNotBuyable	`self.cart.buyable = false`	The ShoppingCart is no longer buyable.

6. Scenarios (Test Cases)

This chapter describes the various test cases that were used to develop the system. There are separate sections for test cases regarding invariants and test cases regarding operations.

6.1 Test Cases for Invariants

Each invariant is tested with one positive and one negative test case and the specific purpose of each test case is described. The positive test case performs the desired behavior, while the negative test case shows how the tested invariant fails..

6.1.1 Class Category

This section describes test cases for the specific constraints of the Category class.

6.1.1.1 Name of Categories

Each category has to have a name.

Positive Case	Every Category has to have a name, when a new Category object is created, it name has to be set as well.

System Setup	`!new Category`
	`!Category1.name := 'Cloths'`

All invariants are satisfied.	

Negative Case	But if a Category is created without a name
System Setup	`!new Category`
The categoryHaveName invariant (4.1.1) fails.	

6.1.1.2 No Duplicate Products in the same Category

Each product has just one category.

Positive Case	There may be no Products with the same name in the same Category.

System Setup	`!new Category`
	`!Category1.name := 'Clothes'`
	`!new Product`
	`!Product1.name := 'Trousers'`
	`!Product1.description := 'Nice Trousers.'`
	`!Product1.manufacturer := 'Lewis'`
	`!Product1.inStock := 10`
	`!insert (Category1,Product1) into Includes`
	`!new Product`
	`!Product2.name := 'T-shirt'`
	`!Product2.description := 'A shirt.'`
	`!Product2.manufacturer := 'Puma'`
	`!Product2.inStock := 10`
	`!new Category`
	`!Category2.name := 'Shirts'`
	`!insert (Category1,Product2) into Includes`
	`!Product1.price := 12.99`
	`!Product2.price := 5.99`

All invariants are satisfied.	

Negative Case	Adding two different Products with the same name to the same Category is not a valid system state.

System Setup	`!new Category`
	`!Category1.name := 'Clothes'`
	`!new Product`
	`!Product1.name := 'Trousers'`
	`!Product1.description := 'Nice Trousers.'`
	`!Product1.manufacturer := 'Lewis'`
	`!Product1.inStock := 10`
	`!insert (Category1,Product1) into Includes`
	`!new Product`
	`!Product2.name := 'Trousers'`
	`!Product2.description := 'Cool trousers.'`
	`!Product2.manufacturer := 'Denim'`
	`!Product2.inStock := 10`
	`!insert (Category1,Product2) into Includes`
	`!Product1.price := 12.99`
	`!Product2.price := 5.99`

The uniqueProductInCategory (4.1.2) invariant fails.	

6.1.2 Class Employee

This section describes test cases for the specific constraints of the Employee class.

6.1.2.1 Employee has salary

Each Employee must have Salary more than zero.

Positive Case	This case creates a valid new Employee with a salary more than zero.

System Setup	`!new Employee`
	`!Employee1.firstName := 'A'`
	`!Employee1.lastName := 'B'`
	`!Employee1.userName := 'AB'`
	`!Employee1.password := '123'`
	`!Employee1.age := 20`
	`!Employee1.address := 'Bremen'`
	`!Employee1.salary := 4000`

All invariants are satisfied.	

Negative Case	We create a Employee without any salary and we receive an error because every Employee has to have a salary bigger than zero

System Setup	`!new Employee`
	`!Employee1.firstName := 'A'`
	`!Employee1.lastName := 'B'`
	`!Employee1.userName := 'AB'`
	`!Employee1.password := '123'`
	`!Employee1.age := 20`
	`!Employee1.address := 'Bremen'`

The mustHaveSalary invariant (4.2.1) fails.	

6.1.3 Class Order

This subchapter details testcases for invariants regarding the Order class.

6.1.3.1 Amount of a Product in an Order must be greater than zero

No Order may contain a Product that has no amount. Products, which amount is set to zero need to be removed from the Order.

Positive Case	When each Product in an Order has an amount greater than zero, the Order is valid.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm'`
	`!Product1.price := 10`
	`!Product1.inStock := 20`
	`!Product1.manufacturer := 'Puma'`
	`!new Order`
	`!insert (Order1,Product1) into ProductsBought`
	`!ProductsBought1.amount := 10`

All invariants are satisfied.	

Negative Case	Orders, where at least one Product has an amount of zero cause an error.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm'`
	`!Product1.price := 10`
	`!Product1.inStock := 20`
	`!Product1.manufacturer := 'Puma'`
	`!new Order`
	`!insert (Order1,Product1) into ProductsBought`
	`!ProductsBought1.amount := 0`

The productinOrderNotZero (4.3.1) invariant fails.	

6.1.3.2 Products in freshly placed Orders may not exceed Products in Stock

An Order may not be placed, if any of the Products contained in it are not in stock or only in stock in to small quantities.

Positive Case	When the amount of each Product in an Order is less or equal than the amount of each Product in stock, everything is well.

System Setup	`!new User`
	`!new Order`
	`!new Product`
	`!insert (User1,Order1) into Places`
	`!insert (Order1,Product1) into ProductsBought`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!Order1.delivered := false`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm'`
	`!Product1.price := 10`
	`!Product1.inStock := 5`
	`!Product1.manufacturer := 'Puma'`
	`!ProductsBought1.amount := 4`

All invariants are satisfied.	

Negative Case	If less Products are in stock than are within the Order, the invariant fails.

System Setup	`!new User`
	`!new Order`
	`!new Product`
	`!insert (User1,Order1) into Places`
	`!insert (Order1,Product1) into ProductsBought`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!Order1.delivered := false`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm'`
	`!Product1.price := 10`
	`!Product1.inStock := 5`
	`!Product1.manufacturer := 'Puma'`
	`!ProductsBought1.amount := 20`

The enoughInStock invariant (4.3.2) fails.	

6.1.3.4 Every Order must contain at least one Product

In every Order there must exist at least one Product. Empty Orders should not occur.

Positive Case	An Order with at least one Product in it, is a valid Order.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm'`
	`!Product1.price := 10`
	`!Product1.inStock := 5`
	`!Product1.manufacturer := 'Puma'`
	`!new Order`
	`!insert (Order1,Product1) into ProductsBought`
	`!ProductsBought1.amount := 5`

All invariants are satisfied.	

Positive Case	An empty Order can exist in the system without issue.
System Setup	`!new Order`
All invariants are satisfied.	

6.1.4 Class Person

The classes User and Employee both inherit attributes and operations from Person. All invariants that apply to the Person class thus include the classes User and Employee. So even if the User class is mostly used in this test cases, they can also be executed with the Employee or Person class or a mix of all three and still work.

6.1.4.1 Unique Username for Users

Each User must have a unique username.

Positive Case	An Order with at least one Product in it, is a valid Order.

System Setup	`!new User`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!new User`
	`!User2.firstName := 'Wolfgang'`
	`!User2.lastName := 'Schmidt'`
	`!User2.userName := 'wSch'`

All invariants are satisfied.	

Negative Case	We create two Users with the same username and receive an error from the system.

System Setup	`!new User`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!new User`
	`!User2.firstName := 'Wolfgang'`
	`!User2.lastName := 'Schmidt'`
	`!User2.userName := 'RiSch'`

The uniqueUserName invariant (4.4.1) fails.	

6.1.4.2 Users must have names

Each User must have a first, a last and a username.

Positive Case	This case creates a valid new User.

System Setup	`!new User`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`

All invariants are satisfied.	

Negative Case	Creating an empty User creates an error.
System Setup	`!new User`
The haveNameAndfamily invariant (4.4.2) fails.	

6.1.4.3 Unique Username for Employees

Each Employee must have a unique username.

Positive Case	Employees with different usernames can coexist and the system does not produce an error.

System Setup	`!new Employee`
	`!Employee1.firstName := 'Bob'`
	`!Employee1.lastName := 'Felix'`
	`!Employee1.userName := 'BFe'`
	`!Employee1.salary := 400`
	`!new Employee`
	`!Employee2.firstName := 'Claas'`
	`!Employee2.lastName := 'Stern'`
	`!Employee2.userName := 'CSt'`
	`!Employee2.salary := 400`

All invariants are satisfied.	

Negative Case	We create two Employees with the same username and receive an error.

System Setup	`!new Employee`
	`!Employee1.firstName := 'Bob'`
	`!Employee1.lastName := 'Felix'`
	`!Employee1.userName := 'BFe'`
	`!Employee1.salary := 400`
	`!new Employee`
	`!Employee2.firstName := 'Claas'`
	`!Employee2.lastName := 'Stern'`
	`!Employee2.userName := 'BFe'`
	`!Employee2.salary := 400`

Again the uniqueUserName invariant (4.4.1) fails	

6.1.4.4 Employees must have names

Each Employee must have a first, a last and a username.

Positive Case	This case creates a valid new Employee.

System Setup	`!new Employee`
	`!Employee1.firstName := 'Bob'`
	`!Employee1.lastName := 'Felix'`
	`!Employee1.userName := 'BFe'`
	`!Employee1.salary := 400`

All invariants are satisfied.	

Negative Case	We create an Employee without any initialisation und we receive a constraint error because first, last and username are Null.
System Setup	`!new Employee`
In this case both the mustHaveSalary (4.2.1) and the uniqueUserName invariant (4.4.1) fail.	

6.1.5 Class Product

This section details test cases specific to the Product class.

6.1.5.1 Price of Product

Each Product must have a price that is greater than zero.

Positive Case	A Product with a price greater than zero is correct.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm T-Shirt'`
	`!Product1.price := 10.50`
	`!Product1.inStock := 10`
	`!Product1.manufacturer := 'Puma'`

All invariants are satisfied.	

Negative Case	If we create a Product with price of zero we will receive a constraint error.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm T-Shirt'`
	`!Product1.price := 0`
	`!Product1.inStock := 10`
	`!Product1.manufacturer := 'Puma'`

The priceNotZero invariant (4.5.1) fails.	

6.1.5.2 Product Name and Price

Each Product must have a name and a price.

Positive Case	A Product created with a name and a price is accepted by the system.

System Setup	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm T-Shirt'`
	`!Product1.price := 10.50`
	`!Product1.inStock := 10`
	`!Product1.manufacturer := 'Puma'`

All invariants are satisfied.	

Negative Case	An empty Product causes an error.
System Setup	`!new Product`
Both the priceNotZero invariant (4.5.1) and the productHaveNamedescInstock (4.5.2) invariants fail.	

6.1.6 Class ShoppingCart

This section details the ShoppingCart class and its constraints.d.

6.1.6.1 Amount of a Product in a Shoppingcart must be greater than zero

Analogous to the behavior of Orders (6.1.3.1), a Shoppingcart may only contain Products with an amount that is greater than zero

Positive Case	The amount of the Product in the ShoppingCart is more than zero.

System Setup	`!new User`
	`!User1.firstName := 'Rira'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!User1.age := 50`
	`!new ShoppingCart`
	`!insert (User1,ShoppingCart1) into Has`
	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm T-Shirt'`
	`!Product1.price := 10.50`
	`!Product1.inStock := 10`
	`!Product1.manufacturer := 'Puma'`
	`!insert (Product1,ShoppingCart1) into ProductsInCart`
	`!ProductsInCart1.amount := 1`

All invariants are satisfied.	

Negative Case	The amount of the Product in the ShoppingCart is zero.
System Setup	`!new User`
	`!User1.firstName := 'Rira'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!User1.age := 50`
	`!new ShoppingCart`
	`!insert (User1,ShoppingCart1) into Has`
	`!new Product`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'long arm T-Shirt'`
	`!Product1.price := 10.50`
	`!Product1.inStock := 10`
	`!Product1.manufacturer := 'Puma'`
	`!insert (Product1,ShoppingCart1) into ProductsInCart`
	`!ProductsInCart1.amount := 0`
The productsinSchCNotZero (4.6.3) invariant fails.	

6.1.7 Class User

The User class also has a few unique constraints. They relate to user specific actions, like when which products can or cannot be rated.

6.1.7.1 Users can only rate Products that they bought

All Users can write ratings, but they should only be able to do so for Products that they already bought in the past. There may never exist a Rating in the system that violates this principle.

Positive Case	Users can rate Products that they bought in delivered Orders.

System Setup	`!new User`
	`!new Product`
	`!new Order`
	`!insert (User1,Order1) into Places`
	`!insert (Order1,Product1) into ProductsBought`
	`!Order1.delivered := true`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!User1.age := 65`
	`!User1.address := 'Breitenweg 1'`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'Ladies T-Shirt'`
	`!Product1.price := 20.95`
	`!Product1.inStock := 20`
	`!Product1.manufacturer := 'Sprit'`
	`!ProductsBought1.amount := 5`
	`!insert (User1,Product1) into Rating`
	`!Rating1.title := 'Not bad'`
	`!Rating1.text := 'the material was not so good'`
	`!Rating1.stars := 3`

All invariants are satisfied.	

Negative Case	Creating a Rating between a User and a Product that was not bought by the User is not correct.

System Setup	`!new User`
	`!new Product`
	`!new Order`
	`!insert (User1,Order1) into Places`
	`!insert (Order1,Product1) into ProductsBought`
	`!Order1.delivered := true`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!User1.age := 65`
	`!User1.address := 'Breitenweg 1'`
	`!Product1.name := 'T-Shirt'`
	`!Product1.description := 'Ladies T-Shirt'`
	`!Product1.price := 20.95`
	`!Product1.inStock := 20`
	`!Product1.manufacturer := 'Sprit'`
	`!new Product`
	`!Product2.name := 'Trousers'`
	`!Product2.description := 'Ladies Pants'`
	`!Product2.price := 14.95`
	`!Product2.inStock := 10`
	`!Product2.manufacturer := 'Sprit'`
	`!ProductsBought1.amount := 5`
	`!insert (User1,Product2) into Rating`
	`!Rating1.title := 'Not bad'`
	`!Rating1.text := 'the material was not so good'`
	`!Rating1.stars := 3`

The rateForBought invariant (4.7.1) fails.	

6.1.7.2 Unique Shopping Cart

Whenever a User is created, the User must also have a ShoppingCart, which is unique and belongs only to this specific User.

Positive Case	A User with exactly one ShoppingCart is correct.

System Setup	`!new User`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!new ShoppingCart`
	`!insert (User1,ShoppingCart1) into Has`

All invariants are satisfied.	

Negative Case	If a User with two ShoppingsCarts is added, we receive an error.

System Setup	`!new User`
	`!User1.firstName := 'Rita'`
	`!User1.lastName := 'Schmidt'`
	`!User1.userName := 'RiSch'`
	`!new ShoppingCart`
	`!insert (User1,ShoppingCart1) into Has`
	`!new ShoppingCart`
	`!insert (User1,ShoppingCart2) into Has`

The unqiueShoppingcart invariant (4.7.2) fails.	

6.2 Test Cases for Operations

Each operation is tested with one positive and one or more negative test cases. The positive test case performs the desired behavior, while the negative test case produces an error in either a pre- or a postcondition of the tested operation.

6.2.1 Class Category

This section describes the tests for the operations of the category class.

6.2.1.1 initCategory()

This operation is used to initialize a new Category object (5.1.1).

Positive Case	

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`

All pre- and postconditions are satisfied.	

Negative Case	Initializing a Category with an empty argument is not correct.

System Setup	`!new Category`
	`!Category1.initCategory('','Women Fashion')`

The NoDuplicateParameters precondition fails, as a newly created category is already initialized with the empty string `''` for all its attributes.	

Negative Case	Initializing two Categories with the the same values is not correct.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!new Category`
	`!Category2.initCategory('Women','Women Fashion')`

The NoDuplicateCategory precondition fails.	

Negative Case	Trying to initialize a Category that is already initialized is not correct.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Category1.initCategory('Women','Women’s Fashion')`

The CategoryIsEmpty precondition fails.	

6.2.1.2 addProductToCategory()

This operation is used for to add Products into Categories (5.1.2).

Positive Case	This successfully adds a Product to a Category.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!Employee1.createCategory('Men','Men Fashion')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt',`
	`20,20, 'Sprit')`
	`!Category2.addProductToCategory(Product1)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to add a Product to a Category that was already added to it results in an error.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!Employee1.createCategory('Men','Men Fashion')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt',`
	`20,20, 'Sprit')`
	`!Category2.addProductToCategory(Product1)`
	`!Category2.addProductToCategory(Product1)`

The ProductNotInCategory precondition fails.	

6.2.1.3 removeProductFromCategory()

This operation is used to remove the Products from Categories (5.1.3).

Positive Case	This test case shows how to remove a Product from a Category.
System Setup	!new Employee !Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen ',20000) !Employee1.createCategory('Men','Men`s Fashion') !Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit') !Category1.addProductToCategory(Product1) !Category1.removeProductFromCategory(Product1)
All pre- and postconditions are satisfied.	

Negative Case	Trying to remove a product that is not included in a Category from it creates an error.
System Setup	!new Employee !Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen ',20000) !Employee1.createCategory('Men','Men`s Fashion') !Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit') !Category1.removeProductFromCategory(Product1)
The ProductInCategory precondition fails.	

6.2.1.4 changeCategoryDescription()

This operation is used to change Category descriptions (5.1.4).

Positive Case	This successfully changes a Category’s description.
System Setup	!new Category !Category1.initCategory('Women','Women Fashion') !Category1.changeCategoryDescription('Fashion for new era')
All pre- and postconditions are satisfied.	

Negative Case	A Category’s description cannot be changed to an empty string.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Category1.changeCategoryDescription('')`

The NoEmptyNewDes precondition fails.	

Negative Case	A Category’s description can only be changed to a value that is not equal to the one it already has.

System Setup	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Category1.changeCategoryDescription('Women Fashion')`

The NotMyDes precondition fails.	

6.2.1.5 addSubcategory()

This operation adds a subcategory to a Category (5.1.5).

Positive Case	This successfully adds a subcategory to a Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Employee1.createCategory('Shirt','Shirts')`
	`!Category1.addSubcategory(Category2)`

All pre- and postconditions are satisfied.	

Negative Case	Only Categories that are not already a Subcategory of a specific Category can be added to it.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Employee1.createCategory('Shirt','Shirts')`
	`!Category1.addSubcategory(Category2)`
	`!Category1.addSubcategory(Category2)`

The SubcategoryIsNotSubcategoryOfThisCategory precondition fails.	

6.2.1.6 removeSubcategory()

This operation removes subcategories form Categories (5.1.6).

Positive Case	This removes a subcategory from a Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Employee1.createCategory('Shirt','Shirts')`
	`!Category1.addSubcategory(Category2)`
	`!Category1.removeSubcategory(Category2)`

All pre- and postconditions are satisfied.	

Negative Case	Only Categories that are subcategories of a Category can be removed from it.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new Category`
	`!Category1.initCategory('Women','Women Fashion')`
	`!Employee1.createCategory('Shirt','Shirt')`
	`!Category2.removeSubcategory(Category1)`

The SubcategoryIsSubcategoryOfThisCategory precondition fails.	

6.2.2 Class Employee

This section tests operations of the Employee class.

6.2.2.1 createEmployee()

This operation (5.2.1) is used to create and simultaneously initialize a new Employee.

Positive Case	This case creates and initializes a new Employee object.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`

All pre- and postconditions are satisfied.	

Negative Case	Doing so with an empty argument causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','','CSe','123',30,'Bremen',30000)`

The NoEmptyParameters precondition fails.	

Negative Case	Giving a newly creates Employee a salary of zero also creates an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',0)`

The PositiveSalary precondition does not hold.	

6.2.2.2 initEmployee()

This operation is used to initialize an Employee. For a detailed description of the operation, take look at subchapter 5.2.2 initEmployee.

Positive Case	This case initializes a new Employee object.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to initialize an Employee with an empty argument, causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('','Felix','BFe','123',29,'Bremen',20000)`

The precondition NoEmptyParameters, in the subsequently called operation initPerson (5.4.1) does not hold in this test case.	

Negative Case	Trying to initialize an Employee with no salary, causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',0)`

The precondition PositiveSalary does not hold in this test case.	

6.2.2.3 raiseSalery()

This operation is used for to raise the salary of an Employee (5.2.3).

Positive Case	This case increase the Salary of an Employee.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee2.raiseSalary(1000)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to raise the salary of an Employee with a negative amount is invalid.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee2.raiseSalary(-1000)`

The RaiseGreaterThanZero precondition fails.	

6.2.2.4 lowerSalery()

This operation (5.2.4) is used to lower the salary of an Employee.

Positive Case	This case decrease the Salary of Employee.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.lowerSalary(1000)`

All pre- and postconditions are satisfied.	

Negative Case	This case decrease the Salary of Employee.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.lowerSalary(-1000)`

The PenaltyGreaterThanZero precondition does not hold.	

6.2.2.5 createProduct()

This operation is used to create and initialize a new Product (5.2.5).

Positive Case	This successfully creates a new Product.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`

All pre- and postconditions are satisfied.	

Negative Case	Trying to create a Product with empty attributes causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('','New arrival T-shirt', 20,20, 'Sprit')`

This causes two preconditions in the called initProduct operation (5.5.1) to fail. Namely NoDuplicateProduct and NoEmptyFields.	

Negative Case	Trying to create a duplicate of an already existing Product also causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`

The NoDuplicateProduct precondition of createProduct (5.2.5) fails.	

6.2.2.6 deleteProduct()

This operation is used by Employees to delete Products (5.2.6). Since you cannot pass not exiting objects into an operation in the USE shell, this operation has no negative test cases.

Positive Case	This successfully deletes an existing Product.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!Employee1.deleteProduct(Product1)`

All pre- and postconditions are satisfied.	

6.2.2.7 updateProduct()

This operation can update the attributes of Products if used correctly (5.2.7).

Positive Case	This successfully updates an existing Product.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 24.99,20, 'Sprit')`
	`!Employee1.updateProduct(Product1,'T-shirt','Long T-shirt',9.99,25,'Didana')`

All pre- and postconditions are satisfied.	

Negative Case	Trying to update a Product with empty values is not correct.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 24.99,20, 'Sprit')`
	`!Employee1.updateProduct(Product1,'T-shirt','',9.99,25,'Didana')`

The NoEmptyFields precondition in the Product's updateProduct operation (5.5.2) fails.	

Negative Case	Trying to update a Product with empty values is not correct.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 24.99,20, 'Sprit')`
	`!Employee1.updateProduct(Product1,'T-shirt','New arrival T-shirt', 24.99,20, 'Sprit')`

Both of the NoDuplicateproduct preconditions in the Employee and the Product class fail.	

6.2.2.8 createCategory()

This operation is used for to create a new Category and initialize it (5.2.8).

Positive Case	This case creates a new Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createCategory('Books','Good for reading.')`

All pre- and postconditions are satisfied.	

Negative Case	In order to create a new category, both the name and the description may not be empty.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createCategory('Books','')`

The NoEmptyAttributes precondition fails.	

6.2.2.9 addProductToCategory()

This operation is used for to add Products into to various Categories (5.2.9).

Positive Case	This shows how an Employee adds a Product to a Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createCategory('Men','Men’s Fashion')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 14.99, 20, 'Sprit')`
	`!Employee1.addProductToCategory(Product1,Category1)`

All pre- and postconditions are satisfied.	

Negative Case	Each Product can only be added to a category once, this also applies when Products are added by Employees.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createCategory('Men','Men’s Fashion')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 14.99, 20, 'Sprit')`
	`!Employee1.addProductToCategory(Product1,Category1)`
	`!Employee1.addProductToCategory(Product1,Category1)`

The ProductNotInCategory precondition fails.	

6.2.2.10 removeProductFromCategory()

This operation is for removing Products from Categories (5.2.10).

Positive Case	This test case removes a Product from a Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)`
	`!Employee1.createCategory('Men','Men’s Fashion')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 14.99, 20, 'Sprit')`
	`!Employee1.addProductToCategory(Product1,Category1)`
	`!Employee1.removeProductFromCategory(Product1,Category1)`

All pre- and postconditions are satisfied.	

Negative Case	Only Products that are actually in a Category can be removed from one. This test case tries to remove a product from a Category that does not contain it.
System Setup	`!new Employee` `!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',30000)` `!Employee1.createCategory('Men','Men’s Fashion')` `!Employee1.createProduct('T-shirt','New arrival T-shirt', 14.99, 20, 'Sprit')` `!Employee1.removeProductFromCategory(Product1,Category1)`
The ProductIsInCategory precondition fails.	

6.2.2.11 changeCategoryDescription()

This operation allows Employees to change Category descriptions (5.2.11).

Positive Case	This case changes the description of a Category.
System Setup	`!new Employee` `!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',330000)` `!Employee1.createCategory('Men','Fashion')` ``!Employee1.changeCategoryDescription(Category1, 'Men`s Fashion')``
All pre- and postconditions are satisfied.	

Negative Case	Trying to change a Category description to an empty string causes an error.
System Setup	`!new Employee` `!Employee1.initEmployee('Class','Stern','CSe','123',30,'Bremen',330000)` ``!Employee1.createCategory('Men','Men`s Fashion')`` `!Employee1.changeCategoryDescription(Category1, '')`
The NewDescriptionNotEmpty precondition fails.	

6.2.2.12 addSubcategoryToCategory()

This operation allows Employees to add subcategories to Categories (5.2.12).

Positive Case	This way an Employee can add a subcategory to a Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women’s Fashion')`
	`!Employee1.createCategory('Shoes','All the shoes one could ever want')`
	`!Employee1.addSubcategoryToCategory(Category2,Category1)`

All pre- and postconditions are satisfied.	

Negative Case	Only a Category that is not already a subcategory of the desired Category can be added to it.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women’s Fashion')`
	`!Employee1.createCategory('Shoes','All the shoes one could ever want')`
	`!Employee1.addSubcategoryToCategory(Category2,Category1)`
	`!Employee1.addSubcategoryToCategory(Category2,Category1)`

The SubcategoryIsNotSubcategoryOfSupercategory precondition fails.	

6.2.2.13 removeSubcategoryFromCategory()

This operation allows Employees to remove subcategories form Categories(5.2.13).

Positive Case	This test case demonstrates the removal of a subcategory from another Category.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women’s Fashion')`
	`!Employee1.createCategory('Shoes','All the shoes one could ever want')`
	`!Employee1.addSubcategoryToCategory(Category2,Category1)`
	`!Employee1.removeSubcategoryFromCategory(Category2,Category1)`

All pre- and postconditions are satisfied.	

Negative Case	Only categories that are subcategories of a Category can be removed from one.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women’s Fashion')`
	`!Employee1.createCategory('Shoes','All the shoes one could ever want')`
	`!Employee1.addSubcategoryToCategory(Category2,Category1)`
	`!Employee1.removeSubcategoryFromCategory(Category2,Category1)`
	`!Employee1.removeSubcategoryFromCategory(Category2,Category1)`

The SubcategoryIsSubcategoryOfSupercategory precondition fails.	

6.2.2.14 deleteCategory()

This operation allows Employees to delete Categories (5.2.14). Again a negative test case is missing as only existing Categories can be passed into the operation in the USE shell.

Positive Case	

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women’s Fashion')`
	`!Employee1.deleteCategory(Category1)`

All pre- and postconditions are satisfied.	

6.2.2.15 createAndAddAsSubcategory()

This operation lets an Employee create a new subcategory as well as adding it to an already existing Category (5.2.15). Cases with empty parameters or duplicate Categories are handled in the operations called by this one.

Positive Case	This test case creates a new Category and adds it as a subcategory.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createCategory('Women','Women Fashion')`
	`!Employee1.createAndAddAsSubcategory('Shirts','Lots of Shirts', Category1)`

All pre- and postconditions are satisfied.	

6.2.2.16 deliverOrder()

This operation is used by Employees to change the status of Orders from undelivered to delivered (5.2.17).

Positive Case	In this case we entered all operations and attributes correctly and then deliver the Order successfully.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`

All pre- and postconditions are satisfied.	

Negative Case	Only undelivered Orders can be delivered

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!Employee1.deliverOrder(Order1)`

The OrderNotDelivered precondition fails.	

Negative Case	Only Orders that contain more than zero Products can be delivered.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!ProductsBought1.amount := 0`
	`!Employee1.deliverOrder(Order1)`

The HasProducts precondition fails.	

6.2.2.17 deleteOrder()

This operation is used by Employees to delete undelivered Orders (5.2.16).

Positive Case	This case demonstrates how to successfully delete an undelivered Order.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deleteOrder(Order1)`

All pre- and postconditions are satisfied.	

Negative Case	Orders that have already been delivered cannot be deleted.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!Employee1.deleteOrder(Order1)`

The OrderNotDelivered precondition fails.	

6.2.3 Class Order

This chapter describes test cases for the operations of the Order class.

6.2.3.1 deliver()

This operation is used to deliver an Order (5.3.2).

Positive Case	An undelivered Order with Products can be delivered.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 9.99, 20, 'Sprit')`
	`!new ShoppingCart`
	`!User1.addProductToCart(Product1,3)`
	`!User1.placeOrder()`
	`!Order1.deliver()`

All pre- and postconditions are satisfied.	

Negative Case	An already delivered Order cannot be delivered.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 9.99, 20, 'Sprit')`
	`!new ShoppingCart`
	`!User1.addProductToCart(Product1,3)`
	`!User1.placeOrder()`
	`!Order1.deliver()`
	`!Order1.deliver()`

The OrderIsNotDelivered precondition fails.	

Negative Case	An Order without Products cannot be delivered.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 9.99, 20, 'Sprit')`
	`!new ShoppingCart`
	`!User1.addProductToCart(Product1,3)`
	`!User1.placeOrder()`
	`!destroy(ProductsBought1)`
	`!Order1.deliver()`

The HasProducts precondition fails.	

6.2.3.2 initOrder()

This operation initializes a new Order (5.3.3).

Positive Case	This test case successfully creates and initializes a new Order.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!new Order`
	`!Order1.initOrder(ShoppingCart1)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to init an Order with an empty ShoppingCart produces an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!new Order`
	`!Order1.initOrder(ShoppingCart1)`

The CartIsbuyable precondition fails.	

6.2.3.3 removeOrder()

This operation restores the Products contained in an undelivered Order back to the system (5.3.4).

Positive Case	Removing an Order that contains Products is no problem.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!new Order`
	`!Order1.initOrder(ShoppingCart1)`
	`!Order1.removeOrder()`

All pre- and postconditions are satisfied.	

Negative Case	Trying to remove an already delivered Order causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!Order1.removeOrder()`

The OrderUndelivered precondition fails.	

6.2.4 Class Person

This chapter details the test cases for the operations of the Person class.

6.2.4.1 initPerson

This operation initializes the attributes of a Person object (5.4.1).

Positive Case	This test case creates and initializes a new Person.

System Setup	`!new Person`
	`!Person1.initPerson('Stefen','Donner','SDu','123',25,'Bremen')`

All pre- and postconditions are satisfied.	

Negative Case	The initialization of a Person may contain no empty arguments.

System Setup	`!new Person`
	`!Person1.initPerson('','Donner','SDu','123',25,'Bremen')`

The NoEmptyParameters precondition fails.	

Negative Case	In this test case, the same Person object is initialized twice, which is not correct.

System Setup	`!new Person`
	`!Person1.initPerson('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Person1.initPerson('Stefen','Donner','SDu','123',25,'Bremen')`

The PersonNotInitialized precondition fails.	

Negative Case	In this case a Person is initialized with a negative age, which causes an error.

System Setup	`!new Person`
	`!Person1.initPerson('Stefen','Donner','SDu','123',-5,'Bremen')`

The AgeNotZeroOrNegative precondition fails.	

6.2.5 Class Product

This section details the test cases used for the operations of the Product class.

6.2.5.1 initProduct()

This operation is used to initialize Products (5.5.1).

Positive Case	This test case successfully initializes a new Product.

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`

All pre- and postconditions are satisfied.	

Negative Case	This test case demonstrates that each Product can only be initialized once.

System Setup	`!new Product`
	`!Product1.initProduct('Skirt','Traditional skirt',30,20,'ZARA')`
	`!Product1.initProduct('Skirt','Traditional skirt',30,20,'ZARA')`

The ProductIsEmpty and NoDuplicateProduct preconditions fail.	

Negative Case	A Product's initialization may contain no empty arguments.

System Setup	`!new Product`
	`!Product1.initProduct('','Traditional skirt',30,20,'ZARA')`

The NoEmptyFields and NoDuplicateProduct preconditions fail.	

Negative Case	A Product's price has to be positive.

System Setup	`!new Product`
	`!Product1.initProduct('Skirt','Traditional skirt',0.00,20,'ZARA')`

The PriceMoreThanZero precondition fails.	

Negative Case	A Product that is to be initialized has to be in stock.

System Setup	`!new Product`
	`!Product1.initProduct('Skirt','Traditional skirt',9.99,-1,'ZARA')`

The StockNotSmallerThanZero precondition fails.	

6.2.5.2 updateProduct()

This operation is used to update the attributes of Products (5.5.2).

Positive Case	Update the product

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`
	`!Product1.updateProduct('Shoes','Sport Shoes',50.30,10,'Puma')`

All pre- and postconditions are satisfied.	

Negative Case	If you try to update the product with null attribute then it throws the error.

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`
	`!Product1.updateProduct('Shoes','',50.30,10,'Puma')`

The NoEmptyFields precondition fails.	

Negative Case	Whenever the updateProduct operation is used at least one attribute value has to change.

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`
	`!Product1.updateProduct('Shoes','Sport Shoes',50.30,10,'Nike')`

The ValuesChange and NoDuplicateProduct preconditions fail.	

Negative Case	The inStock attribute values may never be smaller than zero.

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`
	`!Product1.updateProduct('Shoes','Sport Shoes',50.30,-1,'Nike')`

The ProductRemainsInStock precondition fails.	

Negative Case	The price of a Product may never be set to value smaller than or equal to zero.

System Setup	`!new Product`
	`!Product1.initProduct('Shoes','Sport Shoes',50.30,10,'Nike')`
	`!Product1.updateProduct('Shoes','Sport Shoes',-5.99,10,'Nike')`

The PriceMoreThanZero postcondition fails.	

6.2.6 Class Rating

This chapter details the test cases for the Rating class's operations.

6.2.6.1 initRating()

Thi operation initializes new Ratings for Products (5.6.1).

Positive Case	Test case for a properly executed initRating operation.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!new Rating between(User1,Product1)`
	`!Rating1.initRating('Very Good','Nice delivery',5)`

All pre- and postconditions are satisfied.	

Negative Case	A Rating can only be initialized once.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt',`
	`20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!new Rating between(User1,Product1)`
	`!Rating1.initRating('Very Good','Nice delivery',5)`
	`!Rating1.initRating('Very Good','Nice delivery',5)`

The RatingIsEmpty precondition fails.	

Negative Case	The stars for a Rating have to be 0 at a minimum and 5 at a maximum.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen`
	`',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt',`
	`20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!new Rating between(User1,Product1)`
	`!Rating1.initRating('Very Good','Nice delivery',15)`

The RatingStarsInBounds postcondition fails.	

6.2.7 Class ShoppingCart

In this section we describe the test cases used for the operations of the ShoppingCart class.

6.2.7.1 addToCart()

This operation is used to add Products to ShoppingCarts (5.7.1).

Positive Case	Add product into the cart

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`

All pre- and postconditions are satisfied.	

Negative Case	If you try to add a duplicate entry to a ShoppingCart, it causes an error..

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.addToCart(Product1)`

The NoDuplicateproductinShc precondition fails.	

6.2.7.2 addToCartwithAmount()

This operation is used to add Products to ShoppingCarts in specific amounts.

Positive Case	To add product to the cart with amount.
System Setup	`!new Employee` `!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)` `!new User` `!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')` `!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')` `!ShoppingCart1.addToCartWithAmount(Product1,2)`
All pre- and postconditions are satisfied.	

Negative Case	Trying to add a Product with an amount of zero or less produces an error.
System Setup	`!new Employee` `!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)` `!new User` `!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')` `!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')` `!ShoppingCart1.addToCartWithAmount(Product1,0)`
The AmountGreaterThanZero precondition fails.	

Negative Case	Trying to add a Product with an amount that is greater than the amount of the Product that is in stock creates an error.
System Setup	`!new Employee` `!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)` `!new User` `!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')` `!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')` `!ShoppingCart1.addToCartWithAmount(Product1,30)`
The AmountDoesNotExceedInStock precondition fails.	

Negative Case	Trying to add the same Product to the same ShoppingCart twice creates an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCartWithAmount(Product1,1)`
	`!ShoppingCart1.addToCartWithAmount(Product1,1)`

The NoDuplicateproductinShc precondition fails.	

6.2.7.3 removefromCart()

This operation is used to remove Products from ShoppingCarts (5.7.3).

Positive Case	This test case demonstrates how to remove a Product from a ShoppingCart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.removeProductFromCart(Product1)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to remove a Product from a ShoppingCart in which it is not included, causes an error.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.removeProductFromCart(Product1)`

The ProductIsInCart precondition fails.	

6.2.7.4 changeAmountInCart()

This operation can change the amount of Products in ShoppingCarts (5.7.4).

Positive Case	To change amount of product in shopping cart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.changeAmountInCart(Product1,3)`

All pre- and postconditions are satisfied.	

Negative Case	If the amount in the ShoppingCart is changed to a value greater than the amount in which the Product is in stock, an error is produced.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.changeAmountInCart(Product1,33)`

The AmountDoesNotExceedInStock precondition fails.	

Negative Case	If the amount in the ShoppingCart is changed to a number equal to or smaller than zero, an error is produced.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.changeAmountInCart(Product1,0)`

The AmountGreaterThanZero precondition fails.	

Negative Case	Only the amount of a Product contained in a Shoppingcart can be changed by it.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.changeAmountInCart(Product1,1)`

The ProductInCart precondition fails.	

6.2.7.5 buyCart()

This operation buys the contents of a ShoppingCart (5.7.5).

Positive Case	This test case successfully buys the contents of a ShoppingCart and thus creates an Order.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.addToCart(Product1)`
	`!ShoppingCart1.buyCart()`

All pre- and postconditions are satisfied.	

Negative Case	Only buyable ShoppingCarts can be bought.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!ShoppingCart1.buyCart()`

The CartIsBuyable precondition fails.	

6.2.8 Class User

And at last this chapter describes the test cases for the operations of the User class.

6.2.8.1 initUser()

This operation used to initialize Users (5.8.1). Negative test cases would be analogous to the ones shown for the Person class (6.2.4.1).

Positive Case	This text case contains valid values for a User initialization.

System Setup	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`

All pre- and postconditions are satisfied.	

6.2.8.2 addProductToCart()

This operation allows Users to add Products to their ShoppingCarts (5.8.3).

Positive Case	User add product to the shopping cart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`

All pre- and postconditions are satisfied.	

Negative Case	A Product that is already in the ShoppingCart of a User cannot be added to it again.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.addProductToCart(Product1,1)`

The ProductNotInCart precondition fails.	

Negative Case	A Product has to be added with an amount that is greater than zero.
System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,0)`
The AmountIsPositive precondition fails.	

Negative Case	A Product has to be added with an amount smaller or equal to the amount in stock.
System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,26)`
The AmountNotBiggerThanProductInStock precondition fails.	

6.2.8.3 removeProductFromCart()

This operation is used by Users to remove Products from their ShoppingCarts (5.8.4).

Positive Case	User remove product from the shopping cart.
System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.removeProductFromCart(Product1)`
All pre- and postconditions are satisfied.	

Negative Case	Users can only remove Products from their ShoppingCarts that are contained within them.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.removeProductFromCart(Product1)`

The ProductInCart precondition fails.	

6.2.8.4 changeAmountOfProductInCart()

This operation is used by Users to change the amount of Products in their ShoppingCarts (5.8.5).

Positive Case	Change the amount of product in shopping cart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.changeAmountOfProductInCart(Product1,2)`

All pre- and postconditions are satisfied.	

Negative Case	The operation may only be used with an amount value that is different from the current amount in which a Product is in the ShoppingCart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt',20, 5, 'Sprit')`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.changeAmountOfProductInCart(Product1,1)`

The AmountWillChang precondition fails.	

6.2.8.5 placeOrder()

This operation allows Users to place Orders for their ShoppingCart contents (5.8.6).

Positive Case	User place order which products are in the shopping cart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`

All pre- and postconditions are satisfied.	

Negative Case	Only buyable ShoppingCarts can be bought by their Users.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.placeOrder()`

The ProductsInCart and CartIsBuyable preconditions fails.	

6.2.8.6 rateProduct()

This operation is used by Users to rate Products that they bought (5.8.2).

Positive Case	User place order which products are in the shopping cart.

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!Employee1.deliverOrder(Order1)`
	`!User1.rateProduct('Very Good','Nice delivery',5,Product1)`

All pre- and postconditions are satisfied.	

Negative Case	Trying to rate a product that is ordered but not bought results in an error

System Setup	`!new Employee`
	`!Employee1.initEmployee('Bob','Felix','BFe','123',29,'Bremen',20000)`
	`!new User`
	`!User1.initUser('Stefen','Donner','SDu','123',25,'Bremen')`
	`!Employee1.createProduct('T-shirt','New arrival T-shirt', 20,20, 'Sprit')`
	`!User1.addProductToCart(Product1,1)`
	`!User1.placeOrder()`
	`!User1.rateProduct('Very Good','Nice delivery',5,Product1)`

The ProductWasBought precondition fails.	

7. Queries

Use the following code to create a system state, in which all queries from this chapter will deliver (more or less interesting) results. The initial chapters contain a lot of very simply queries. For more advanced examples feel free to skip ahead to subchapter 7.8 More Complex Queries. Also note that the <X> present in some queries denotes queries in which a specific USE object is necessary as input so User<X> can stand for User1, User2 or User3 in the model below.

```
!new User
!User1.initUser('Wolfgang','Schmidt','WoSchmi','secret',43,'34 Omimi,
Weirdstreet 42')
!new User
!User2.initUser('Johanna','Sun','JoSun','secret',52,'261 Island, Streetlane 1')
!new User
!User3.initUser('Sarah','Moon','SaMoo','secret',22,'21 Longland, AppleArch 20')

!new Employee
!Employee1.initEmployee('Rainer','Lala','RaLa','secret',25,'34 Omimi, Plainroad
4',400)
!new Employee
!Employee2.initEmployee('E','T','ET','secret',33,'???',500)

!Employee1.createProduct('dvd','a dvd', 9.99, 15, 'DVDMaker')
!Employee1.createProduct('book','a book', 4.95, 50, 'Scribbler')
!Employee1.createProduct('Rare Fruit','damn rare', 999.95, 0, 'FruitFinders')
!Employee1.createProduct('Orange','An orange.',2.99,30,'FruitFinders')
!Employee1.createProduct('Banana','Beloved by minions around the
globe.',0.99,30,'FruitFinders')
!Employee1.createProduct('Apple','Eat one a day to keep the doctor
away.',2.99,42,'FruitFinders')
!Employee1.createProduct('Cucumber','This is green.',1.29,5,'VegWorld')

!Employee1.createCategory('Movies','moving pictures')
!Employee1.createCategory('Horror','scary')
!Employee1.createCategory('Drama','ahhhh')
!Employee2.createCategory('Food','Tasty and keeps you alive.')
!Employee2.createCategory('Fruits','Very healthy.')
!Employee2.createCategory('Vegetables','Also quite healthy.')

!Employee1.addSubcategoryToCategory(Category2,Category1)
!Employee1.addSubcategoryToCategory(Category3,Category1)
!Employee2.addSubcategoryToCategory(Category5,Category4)
```

```
!Employee1.addProductToCategory(Product1, Category1)
!Employee1.addProductToCategory(Product3, Category5)
!Employee1.addProductToCategory(Product4, Category5)
!Employee1.addProductToCategory(Product5, Category5)
!Employee1.addProductToCategory(Product6, Category5)
!Employee1.addProductToCategory(Product7, Category6)

!User1.addProductToCart(Product1,1)
!User1.addProductToCart(Product5,3)
!User2.addProductToCart(Product4,3)
!User2.addProductToCart(Product5,12)
!User2.addProductToCart(Product6,2)
!User2.addProductToCart(Product7,3)

!User1.placeOrder()
!User2.placeOrder()

!Employee1.deliverOrder(Order1)
!Employee1.deliverOrder(Order2)

!User2.addProductToCart(Product4,2)
!User2.placeOrder()
!Employee1.deliverOrder(Order3)

!User1.rateProduct('Fascinating','These bananas are so cool.',4,Product5)
!User1.rateProduct('Generic','This DVD is empty and boring.',0,Product1)
!User2.rateProduct('Nice','These oranges are in good condition.',4,Product4)
!User2.rateProduct('Awesome','The bananas are outstandingly tasty!',5,Product5)
!User2.rateProduct('Okay...I guess','These apples have a very bland taste to
them.',3,Product6)

!Employee2.updateProduct(Product3,'Rare Fruit','Damn rare.', 999.95, 1,
'FruitFinders')

!User1.addProductToCart(Product3,1)
!User2.addProductToCart(Product3,1)
```

7.1 Queries regarding Products

All of these queries deliver Products, which meet different requirements.

7.1.1 Get all Products

```
Product.allInstances()
```

This provides all Products currently available in the system. It serves as a base for following queries.

7.1.2 Get all Products of Category <X>

```
Category<X>.product
Product.allInstances()->select(p | p.category->includes(Category<X>))
```

There are at least two ways to do this: Either a specific Category<X> can be prompted to evaluate its product role, which belongs to the Includes association (?). Or out of all Products, we select only those, which category role matches to the specific category which Products we want to look at.

7.1.3 Get all Products with at least one Rating

```
Product.allInstances()->select(p | p.rating->isEmpty() = false)
```

From all Products we select those, that have a non empty rating role attached to them. This provides us a list with all Products that were rated by any User at least once. Conversely to get all Products without a Rating you would need to remove the ‘`= false`’ part of the selection.

7.1.4 Get all Products with a Price below <X>

```
Product.allInstances()->select(p | p.price <= <X>)
```

Select from all Products each Product with a price that is lower or equal to a given number. The next two queries are similar to this one, but check for equal and higher or equal price instead.

7.1.5 Get all Products with a Price of exactly <X>

```
Product.allInstances()->select(p | p.price = <X>)
```

7.1.6 Get all Products with a price above <X>

```
Product.allInstances()->select(p | p.price >= <X>)
```

7.1.7 Get all Products that were sold at least once

```
Product.allInstances()->select(p | p.productsBought->size() > 0)
```

This query selects all Products that have at least one productsBought associationclass (?) and thus were bought at least once.

7.1.8 Get all Products currently in ShoppingCarts

```
Product.allInstances()->select(p | p.productsInCart->size() > 0)
```

Similar to the last one, this query selects all Products that have at least one productsInCart associationclass (?) and are thus present in at least one ShoppingCart.

7.1.9 Get all Products that bought by a specific User

```
Product.allInstances()->select(p | p.order.buyer->includes(User<X>))
```

This query delivers all Products, which belong to an Order bought by a specific User. This is achieved by checking each Products order (?) and selecting every entry that includes the desired User as a buyer.

7.1.10 Get all Products that are in stock

```
Product.allInstances()->select(p | p.inStock > 0)
```

This query again selects Products based on a simple attribute value check, every Product with inStock greater than zero is in stock.

7.1.11 Get all Products that are out of stock

```
Product.allInstances()->select(p | p.inStock = 0)
```

Conversely if you check for Products with an inStock attribute value of exactly zero, you get all Products that are currently out of stock.

7.1.12 Get all Products of which there are more in Carts than in Stock

```
Product.allInstances()->select(p | p.inStock < p.productsInCart.amount->sum())
```

This query is slightly more interesting. It takes the inStock attribute as described in last two queries. But then compares it to the summed amount of the same Product currently in ShoppingCarts. This summed amount represents the total amount of a specific Product that is currently in all ShoppingCarts combined. If this amount is greater than the amount in stock, it means that not all of the potential Orders can be served by the shop. Thus it could be sensible to display a warning message to all Users, which have such a Product in their carts, so that they can purchase quickly and still get the product before it goes out of stock.

7.2 Queries regarding Users

These queries deliver user objects with specific qualities.

7.2.1 Get all Users

```
User.allInstances()
```

This is the basic query to get all Users currently in the system.

7.2.2 Get all Users who placed at least one Order

```
User.allInstances()->select(u | u.order->isEmpty() = false)
```

This query selects all Users, whose order role in the Places association (3.3.5) is not empty, which means that the User placed at least one Order with the shop. If you want to get all Users that never purchased anything remove the '`= false`' part at the end of the query (7.2.7).

7.2.3 Get all Users who published a Rating

```
User.allInstances()->select(u | u.rating->isEmpty() = false)
```

Similar to the previous query, but with the rating role belonging to the Rating associationclass (3.3.8) instead.

7.2.4 Get all Users with buyable ShoppingCarts

```
User.allInstances()->select(u | u.cart.buyable = true)
```

This query selects all Users, whose ShoppingCart is currently buyable. This is determined by traversing each User's cart role in the Has association(3.3.2) to get to the ShoppingCart and then evaluating its buyable attribute.

7.2.5 Get all Users with not buyable ShoppingCarts

```
User.allInstances()->select(u | u.cart.buyable = false)
```

Opposite case to the previous query.

7.2.6 Get all Users with empty ShoppingCarts

```
User.allInstances()->select(u | u.cart.product->size = 0)
```

In a similar manner the cart role can be traversed to get to the ShoppingCart's product role in the ProductsInCart associationclass (3.2.7) and select all Users, for whom at least one such object exists.

7.2.7 Get all Users that never bought anything

```
User.allInstances()->select(u | u.order->isEmpty())
```

This query delivers all Users that never placed an Order and thus never bought anything from the shop.

7.3 Queries regarding Employees

This queries deliver Employee objects with specific properties.

7.3.1 Get all Employees

```
Employee.allInstances()
```

The base query that delivers all Employees in the system.

7.3.2 Get all Employees with a salary above <X>

```
Employee.allInstances()->select(e | e.salary >= <X>)
```

This query works by selecting all Employees with a salary attribute value above or equal to a given number. The next query is similar but instead selects based either on value lower or equal to the given number.

7.3.3 Get all Employees with a salary below <X>

```
Employee.allInstances()->select(e | e.salary <= <X>)
```

7.3.4 Get all Employees that created at least one Category

```
Employee.allInstances()->select(e | e.category->size() > 0)
```

This query selects every Employee who has at least one category role in a Creates association (3.3.1).

7.3.5 Get all Employees that created no Category

```
Employee.allInstances()->select(e | e.category->size() = 0)
```

Opposite case to the previous query.

7.3.6 Get all Employees that created at least one Product

```
Employee.allInstances()->select(e | e.product->size() > 0)
```

Similar to finding Employees who already created Categories, Employees who already created Products can be found by traversing their product role in the Manages association (3.3.4) and checking whether is greater than zero or is not empty.

7.3.7 Get all Employees that created no Product

```
Employee.allInstances()->select(e | e.product->size() = 0)
```

Set the comparison operator to zero and you get all Employees that never created a Product.

7.4 Queries regarding Ratings

This queries deliver Ratings with specific qualities.

7.4.1 Get all Ratings

```
Rating.allInstances()
```
The base query for viewing all Ratings in the system.

7.4.2 Get all Ratings of Product <X>

```
Product<X>.rating
Rating.allInstances()->select(r | r.reviewedItem = Product<X>)
```
Again there exist two obvious approaches. Option one is to traverse the rating role of the desired Product. Option two is to select all Ratings which have the desired Product as their reviewedItem. In both cases the query relies on the roles in the Rating associationclass (3.3.8).

7.4.4 Get all Ratings published by User <X>

```
User<X>.rating
Rating.allInstances()->select(r | r.author = User<X>)
```
In a similar manner to the previous query. In order to get all Ratings authored by a specific User, we either ask for the User's rating roles directly, or select from all Ratings those, that have the User in question as their author role.

7.4.5 Get all Ratings with less than <X> stars

```
Rating.allInstances()->select(r | r.stars <=  <X>)
```
By selecting all Ratings with a stars attribute value of lower or equal to the specified amount, this query filters Ratings according to their stars. This is also true for the next two queries, but they ask for an equal or higher or equal amount of stars respectively.

7.4.6 Get all Ratings with exactly <X> stars

```
Rating.allInstances()->select(r | r.stars =  <X>)
```

7.4.7 Get all Ratings with more than <X> stars

```
Rating.allInstances()->select(r | r.stars >=  <X>)
```

7.5 Queries regarding Orders

In this section we cover queries that deliver specific Orders.

7.5.1 Get all Orders

```
Order.allInstances()
```

Execeute this query to get all Orders in the system.

7.5.2 Get all Orders with a total value above <X>

```
Order.allInstances()->select(o | o.totalValue >= <X>)
```

This query selects all Orders with a total value that is greater or equal to a given number.

7.5.3 Get all Orders with a total value below <X>

```
Order.allInstances()->select(o | o.totalValue <= <X>)
```

This query selects all Orders with a total value that is smaller or equal to a given number.

7.5.4 Get all Orders of User <X>

```
User<X>.order
Order.allInstances()->select(o | o.buyer = User<X>)
```

To get all Orders of a specific User, we can either start the query from the User and ask for all its Order roles directly, or select all Orders where the buyer role corresponds to the desired User.

7.5.5 Get all Orders containing a specific Product

```
Order.allInstances()->select(o | o.product->includes(Product<X>))
```

This query delivers all Orders that contain a given Product. This is achieved by traversing the product role of the Order in the Places association (3.3.5) and checking whether it contains the desired Product.

7.6 Queries regarding Categories

These queries can be used to find Category objects with specific properties.

7.6.1 Get all Categories

```
Category.allInstances()
```

This is the basic query to get all Categories in the system.

7.6.2 Get all Categories with at least one Subcategory

```
Category.allInstances->select(c | c.subcategory->size() >= 1)
```

This query selects all Categories, which have at least one subcategory role.

7.6.3 Get all Categories with no Subcategory

```
Category.allInstances->select(c | c.subcategory->size() = 0)
```

This query selects all Categories that have no subcatgory roles.

7.6.4 Get all Categories that are exclusive Supercategories

```
Category.allInstances()->select(c | c.supercategory->isEmpty())
```

By selecting all Categories without a supercategory roles, this query delivers all true Supercategories in the system.

7.6.5 Get all Categories containing at least one Product

```
Category.allInstances()->select(c | c.product->size > 0)
```

This query selects all categories with at least one product role in the Includes association (3.3.3). The next query is the same except that it asks for exactly zero product roles and thus delivers Categories that contain no Products at all.

7.6.6 Get all Categories that contain exactly zero Products

```
Category.allInstances()->select(c | c.product->size = 0)
```

7.6.7 Get all categories created by Employee <X>

```
Employee<X>.category
```

By traversing a given Employees category role in the Creates association, this query delivers all categories created by a specific Employee.

7.7 Queries regarding Subcategories

This queries specifically deliver Subcategories as their results.

7.7.1 Get all Subcategories

```
Category.allInstances()->select(c | c.supercategory->isEmpty() = false)
```

This query gets all Subcategories in the system, by selecting all Categories that have a non-empty supercategory role in the Subcategory association (3.3.9).

7.7.2 Get all Subcategories of Category <X>

```
Category<X>.subcategory
```

This query delivers all Subcategories of a given category by traversing its subcategory role.

7.8 More Complex Queries

This last chapter of the document covers some more complex queries that either require more operations, traverse greater distances in the model or both.

7.8.1 The most bought Product in the system

```
Product.allInstances()->sortedBy(p | p.productsBought.amount->sum())->last()
```

This query takes all Products in the system and sorts them according to the total amount bought. Since sorting this way list Products that were bought in low amounts first and Products with high amounts last, choosing the last entry in the resulting Sequence produces the most bought Product of the system.

7.8.2 The most bought Product of Category <X>

```
Category<X>.product->sortedBy(p | p.productsBought.amount->sum())->last
```

Similar to the previous query. After getting all Products within the category that we want to look at, the amount bought for each Product is summed and used as an indicator by which to sort the Products. Again the sorting goes lowest to highest number and thus we chose the last entry to determine the most bought Product of the chosen Category.

7.8.3 The most bought from Category

```
Category.allInstances()->sortedBy(c |
c.product.productsBought.amount->sum())->last()
```

In Order to determine the most bought from Category the first step is to collect all Categories in the system. Afterwards the Products in each Category are used to determine their respective amount in ProductsBought and all of this is summed. At last we pick the last element in the Sequence to get the Category with the highest amount of sold Products.

7.8.4 The most highly rated Product

```
Product.allInstances->select(p | not p.rating->isEmpty())->sortedBy(p |
p.rating.stars->sum()/p.rating->size())->last()
```

The Product with the best average Rating can be found, by first selecting all Products with at least one Rating. Afterwards the Products with Ratings are sorted by the value of all their stars divided by the amount of Ratings they received, effectively sorting them by average stars value. In the end we just pick the last element to get the Product with the highest average stars.

7.8.5 The most lowly rated Product

```
Product.allInstances->select(p | not p.rating->isEmpty())->sortedBy(p |
p.rating.stars->sum()/p.rating->size())->first()
```

Opposite case to the previous query.

7.8.6 The Product with the most Ratings

```
Product.allInstances->sortedBy(p | p.rating->size())->last()
```

By sorting the Ratings of each Product after size, we can find the Product with the most Ratings in the system. Again we have to pick the last element in the Sequence because sortedBy sorts lowest to highest value.

Appendix A: USE Specification

```
model OSMS
-------***************************************************************************-
------
class Person
       attributes
       firstName : String init: ''
       lastName : String init: ''
       userName : String init: ''
       password : String init: ''
       age : Integer init: 0
       address : String init: ''
       operations

-------------------------------------------------------------------------------------
----
       --initializes the empty person object attributes with values
       initPerson(fName:String,  lName:String,  uName:String,  pw:String,  age:Integer,
address:String)
             begin
             self.firstName := fName;
             self.lastName := lName;
             self.userName := uName;
             self.password := pw;
             self.age := age;
             self.address := address;
             end
       pre  PersonNotInitialized:  self.firstName  =  ''  and  self.lastName  =  ''  and
self.userName = '' and self.password = '' and self.age = 0 and self.address = ''
       pre NoEmptyParameters: fName <> '' and lName <> '' and uName <> '' and pw <> ''
and age <> 0 and address <> ''
       pre AgeNotZeroOrNegative: age > 0
       post UniqueUsername: Person.allInstances->isUnique(userName)
       post  PersonInitialized:  self.firstName  =  fName  and  self.lastName  =  lName  and
self.userName  =  uName  and  self.password  =  pw  and  self.age  =  age  and  self.address  =
address
end
```

```
-------**************************************************************************-
------
class User < Person
       attributes

       operations

------------------------------------------------------------------------------------
----
       --initializes the empty user object with attribute values and a ShoppingCart
       initUser(fName:String,  lName:String,  uName:String,  pw:String,  age:Integer,
address:String) : ShoppingCart
              begin
              declare shc : ShoppingCart;
              shc := new ShoppingCart();
              insert(self,shc) into Has;
              self.initPerson(fName,lName,uName,pw,age,address);
              result := shc;
              end
       pre NoCart: self.cart = Undefined
       post HasOneCart: self.cart->size = 1
       post CartIsTheCreatedOne: self.cart = result
       post CartEmpty: self.cart.product->size = 0
       post CartValueZero: self.cart.totalValue = 0
       post CartNotBuyable: self.cart.buyable <> true

------------------------------------------------------------------------------------
---
       --creates a new rating for a product that the user has bought
       rateProduct(rTitle:String, rText:String, rStars:Integer, p:Product) : Rating
              begin
              declare r:Rating;
              r := new Rating between (self,p);
              r.initRating(rTitle,rText,rStars);
              result:=r;
              end
       pre   ProductWasBought:   self.order->select(o   |   o.delivered).product   ->
includes(p)
```

```
        post RatingBelongsToUser: self.rating->includes(result)
        post RatingExistsForProduct: p.rating->includes(result)
        post RatingIsCreated: result.oclIsNew()
        post RatingType: result.oclIsTypeOf(Rating)

-------------------------------------------------------------------------------------
---
        --adds a specific amount of a product to the users shopping cart
        addProductToCart(p:Product, amount:Integer)
              begin
              self.cart.addToCartWithAmount(p, amount);
              end
        pre ProductNotInCart: self.cart.product->excludes(p)
        pre AmountIsPositive: amount > 0
        pre AmountNotBiggerThanProductInStock: amount <= p.inStock
        post ProductInCart: self.cart.product->includes(p)

-------------------------------------------------------------------------------------
---
        --removes all instances of a specific product from the users shopping cart
        removeProductFromCart(p:Product)
              begin
              self.cart.removeProductFromCart(p);
              end
        pre ProductInCart: self.cart.product->includes(p)
        post ProductNotInCart: self.cart.product->excludes(p)

-------------------------------------------------------------------------------------
---
        --changes the amount of a specific product that is already in the shopping cart
        changeAmountOfProductInCart(p:Product, amount:Integer)
              begin
              self.cart.changeAmountInCart(p, amount)
              end
        pre     AmountWillChange:     self.cart.product->select(pr     |     pr     =
p).productsInCart->select(pic | pic.cart = self.cart).amount->asOrderedSet()->first()
<> amount
```

```
	post	AmountWasSetCorrectly:	self.cart.product->select(pr	|	pr	=
p).productsInCart->select(pic | pic.cart = self.cart).amount->asOrderedSet()->first()
= amount

-------------------------------------------------------------------------------------
---
	--buys the contents of the shopping cart
	placeOrder()
		begin
		self.cart.buyCart();
		end
	pre ProductsInCart: self.cart.product->size > 0
	pre CartIsBuyable: self.cart.buyable
	post ShoppingCartIsEmpty: self.cart.product->size = 0
	post CartNotBuyable: self.cart.buyable = false
end

-------*****************************************************************************-
------
class Product
	attributes
	name : String  init: ''
	description : String  init: ''
	price : Real init: 0
	inStock: Integer init: 0
	inCarts: Integer derived: productsInCart.amount->sum()
	manufacturer: String  init: ''

	operations

-------------------------------------------------------------------------------------
---
	--initializes all attributes of the empty product
	initProduct(pName:String,  pDescription:String,  pPrice:Real,  pInStock:Integer,
pManufacturer:String)
	begin
		self.name:=pName;
		self.description := pDescription;
		self.price := pPrice;
```

```
             self.inStock := pInStock;
             self.manufacturer := pManufacturer;
      end
      pre NoDuplicateProduct:  Product.allInstances->forAll(p |  p.name<>pName)
      pre NoEmptyFields: pName <> '' and pDescription <> '' and pManufacturer <> ''
      pre StockNotSmallerThanZero: pInStock >=0
      pre PriceMoreThanZero: pPrice > 0
      pre ProductIsEmpty: self.name = '' and self.description = '' and self.price = 0
and self.manufacturer = ''
      post ProductNameIsUnique: Product.allInstances->isUnique(name)
      post ProductChanged: self.name = pName and self.description = pDescription and
self.price = pPrice and self.manufacturer = pManufacturer

-------------------------------------------------------------------------------------
---
      --updates the attribute values of the product
      updateProduct(pName:String, pDescription:String, pPrice:Real, pInStock:Integer,
pManufacturer:String)
      begin
             self.name := pName;
             self.description := pDescription;
             self.price := pPrice;
             self.inStock := pInStock;
             self.manufacturer := pManufacturer;
      end
      pre ProductRemainsInStock: pInStock >=0
      pre NoEmptyFields: pName <> '' and pDescription <> '' and pPrice <> 0 and
pInStock <> 0 and pManufacturer <> ''
      pre ValuesChange: pName <> self.name or pDescription <> self.description or
pPrice <> self.price or pInStock <> self.inStock or pManufacturer <> self.manufacturer
      pre NoDuplicateProduct: not Product.allInstances->exists(p | p.name = pName and
p.description = pDescription and p.price = pPrice and p.inStock = pInStock and
p.manufacturer = pManufacturer)
      post PriceMoreThanZero: pPrice > 0
      post ProductChanged: self.name = pName and self.description = pDescription and
self.price = pPrice and self.inStock = pInStock and self.manufacturer = pManufacturer
end
```

```
-------************************************************************************-
------
      class Category
      attributes
            name : String  init: ''
            description : String  init: ''

      operations

--------------------------------------------------------------------------------
---
      --initializes a category or changes its name/description
      initCategory(cName:String, cDes:String)
            begin
            self.name := cName;
            self.description := cDes;
            end
      pre NoDuplicateCategory: not Category.allInstances->exists(c | c.name = cName
and c.description = cDes)
      pre NoDuplicateParameters: cName <> '' and cDes <> ''
      pre CategoryIsEmpty: self.name = '' and self.description = ''
      post UniqueCategoryName: Category.allInstances->isUnique(name)
      post CategoryAttributesSet: self.name = cName and self.description = cDes

--------------------------------------------------------------------------------
----
      --adds a product into a category
      addProductToCategory(p:Product)
            begin
            insert(self,p) into Includes;
            end
      pre ProductExists: Product.allInstances->includes(p)
      pre ProductNotInCategory: self.product->excludes(p)
      post ProductInCategory: self.product->includes(p)
      post        AmountProductInCategory:        self.subcategory->forAll(c1        |
(self.subcategory->closure(subcategory).product->size    +    self.product->size)    >=
(c1.subcategory->closure(subcategory).product->size + c1.product->size)  )
```

```
-------------------------------------------------------------------------------------
---
      --removes a product from a category
      removeProductFromCategory(p:Product)
            begin
            delete(self,p) from Includes;
            end
      pre ProductExists: Product.allInstances->includes(p)
      pre ProductInCategory: self.product->includes(p)
      post   ProductNotInCategory:   Category.allInstances->forAll(   c   |   c.product
->excludes( p))
      post        AmountProductInCategory:        self.subcategory->forAll(c1        |
(self.subcategory->closure(subcategory).product->size    +    self.product->size)    >=
(c1.subcategory->closure(subcategory).product->size + c1.product->size)  )

-------------------------------------------------------------------------------------
---
      --changes the description of a category
      changeCategoryDescription(newDes:String)
            begin
            self.description := newDes;
            end
      pre NotMyDes: self.description <> newDes
      pre NoEmptyNewDes: newDes <> ''
      post ChangedDesc: self.description = newDes

-------------------------------------------------------------------------------------
---
      --adds a Subcategory to a category
      addSubcategory(subC:Category)
            begin
            insert(self,subC) into Subcategory;
            end
      pre SubcategoryExists: Category.allInstances->includes(subC)
      pre SubcategoryIsNotSubcategoryOfThisCategory: self.subcategory->excludes(subC)
      post SubcategoryIsSubcategoryOfThisCategory: self.subcategory->includes(subC)
```

```
-------------------------------------------------------------------------------------
---
      --removes a Subcategory from a category
      removeSubcategory(subC:Category)
            begin
            delete(self,subC) from Subcategory;
            end
      pre SubcategoryExists: Category.allInstances->includes(subC)
      pre SubcategoryIsSubcategoryOfThisCategory: self.subcategory->includes(subC)
      post                                SubcategoryIsNotSubcategoryOfThisCategory:
self.subcategory->excludes(subC)

end

-------**********************************************************************-
------
class Employee < Person
      attributes
      salary: Real init: 0
      operations

-------------------------------------------------------------------------------------
---
      --creates a new employee and initializes all of its attributes
      createEmployee(fName:String,     lName:String,     uName:String,     pw:String,
age:Integer, address:String, salary:Real) : Employee
      begin
            declare e : Employee;
            e := new Employee();
            e.initEmployee(fName,lName,uName,pw,age,address, salary);
            result:=e;
      end
      pre PositiveSalary: salary > 0
      pre NoEmptyParameters: fName <> '' and lName <> '' and uName <> '' and pw <> ''
and age > 0 and address <> ''
      post EmployeeExists: Employee.allInstances->includes(result)
```

```
-------------------------------------------------------------------------------------
---
      --initializes all attributes of the empty employee
      initEmployee(fName:String, lName:String, uName:String, pw:String, age:Integer,
address: String, salary : Real)
      begin
            self.initPerson(fName,lName,uName,pw,age,address);
            self.salary := salary;
      end
      pre PositiveSalary: salary > 0
      post SalarySetCorrectly: self.salary = salary

-------------------------------------------------------------------------------------
---
      --raises the salary
      raiseSalary(raise:Real) : Real
      begin
            result := self.salary;
            self.salary := self.salary + raise;
      end
      pre RaiseGreaterThanZero: raise > 0
      post SalaryMoreThanZero: self.salary > 0
      post SalaryIncreased: self.salary > result

-------------------------------------------------------------------------------------
---
      --lowers the salary
      lowerSalary(penalty:Real) : Real
      begin
            result := self.salary;
            self.salary := self.salary - penalty;
      end
      pre PenaltyGreaterThanZero: penalty > 0
      post SalaryMoreThanZero: self.salary > 0
      post SalaryDecreased: self.salary < result

-------------------------------------------------------------------------------------
---
```

```
	--creates a new product and tells it to initialize its attributes
	createProduct(pName:String, pDescription:String, pPrice:Real, pInStock:Integer, 
pManufacturer:String) : Product
	begin
		declare p : Product;
		p := new Product();
		insert(self,p) into Manages;
		p.initProduct(pName, pDescription, pPrice , pInStock, pManufacturer);
		result := p;
	end
	pre NoDuplicateProduct: not Product.allInstances->exists(p | p.name = pName and 
p.description = pDescription and p.price = pPrice and p.manufacturer = pManufacturer)
	post ProductExists: Product.allInstances->includes(result)
	post CreatedByEmployee: self.product->includes(result)

-------------------------------------------------------------------------------------
---
	--deletes a product
	deleteProduct(p:Product)
	begin
		destroy(p);
	end
	pre TheProductExists: Product.allInstances->includes(p)
	post ProductNoLongerExists: Product.allInstances->excludes(p)

-------------------------------------------------------------------------------------
----
	--tells product to update its attribute values
	updateProduct(p:Product,pName:String,     pDescription:String,     pPrice:Real, 
pInStock:Integer, pManufacturer:String)
	begin
		p.updateProduct(pName, pDescription, pPrice , pInStock, pManufacturer);
	end
	pre ProductExists: Product.allInstances->includes(p)
	pre ValuesChange: pName <> p.name or pDescription <> p.description or pPrice <> 
p.price or pInStock <> p.inStock or pManufacturer <> p.manufacturer
	pre NoDuplicateProduct: not Product.allInstances->exists(p | p.name = pName and 
p.description = pDescription and p.price = pPrice and p.manufacturer = pManufacturer)
```

```
-------------------------------------------------------------------------------------
----
      --creates a new category and tells it to initialize its attributes
      createCategory(cName:String, cDes:String) : Category
      begin
            declare c:Category;
            c:=new Category();
            insert(self,c) into Creates;
            c.initCategory(cName,cDes);
            result := c;
      end
      pre  NoEmptyAttributes: cName <> '' and cDes <> ''
      post EmployeeCreatedCategory: self.category->includes(result)
      post CategoryIsCreated: result.oclIsNew()
      post CategoryType: result.oclIsTypeOf(Category)

-------------------------------------------------------------------------------------
----
      --tells a category to add a product
      addProductToCategory(p:Product, c:Category)
      begin
            c.addProductToCategory(p);
      end
      pre CategoryExists: Category.allInstances->includes(c)
      pre ProductExists: Product.allInstances->includes(p)
      pre ProductNotInCategory: c.product->excludes(p)
      post ProductIsInCategory: c.product->includes(p)

-------------------------------------------------------------------------------------
---
      --tells a category to remove a product
      removeProductFromCategory(p:Product, c:Category)
      begin
            c.removeProductFromCategory(p);
      end
      pre CategoryExists: Category.allInstances->includes(c)
      pre ProductExists: Product.allInstances->includes(p)
      pre ProductIsInCategory: c.product->includes(p)
      post ProductNotInCategory: c.product->excludes(p)

-------------------------------------------------------------------------------------
---
      --tells a category to change its description
      changeCategoryDescription(c:Category, newDes:String)
      begin
            c.changeCategoryDescription(newDes);
      end
      pre CategoryExists: Category.allInstances->includes(c)
      pre NewDescriptionNotEmpty: newDes <> ''

-------------------------------------------------------------------------------------
---
      --tells a category to add a Subcategory
      addSubcategoryToCategory(subC:Category, superC:Category)
      begin
            superC.addSubcategory(subC);
      end
      pre SupercategoryDoesExist: Category.allInstances->includes(superC)
      pre SubcategoryDoesExist: Category.allInstances->includes(subC)
      pre                                 SubcategoryIsNotSubcategoryOfSupercategory:
superC.subcategory->excludes(subC)
      post                                   SubcategoryIsSubcategoryOfSupercategory:
superC.subcategory->includes(subC)

-------------------------------------------------------------------------------------
---
      --tells a category to remove a Subcategory
      removeSubcategoryFromCategory(subC:Category, superC:Category)
      begin
            superC.removeSubcategory(subC);
      end
      pre SupercategoryDoesExist: Category.allInstances->includes(superC)
      pre SubcategoryDoesExist: Category.allInstances->includes(subC)
      pre SubcategoryIsSubcategoryOfSupercategory: superC.subcategory->includes(subC)
      post                                SubcategoryIsNotSubcategoryOfSupercategory:
superC.subcategory->excludes(subC)
```

```
------------------------------------------------------------------------------------
---
      --deletes a category
      deleteCategory(c:Category)
      begin
            destroy(c)
      end
      pre CategoryExists: Category.allInstances->includes(c)
      post CategoryDoesNotExist: Category.allInstances->excludes(c)

------------------------------------------------------------------------------------
---
      --tells a category to create a new category and add it as a Subcategory to an
existing category
      createAndAddAsSubcategory(cName:String,    cDes:String,    superC:Category)    :
Category
      begin
            result := self.createCategory(cName,cDes);
            superC.addSubcategory(result);
      end
      pre SupercategoryDoesExist: Category.allInstances->includes(superC)
      post SubcategoryDoesExist: Category.allInstances->includes(result)
      post                                    SubcategoryIsSubcategoryOfSupercategory:
superC.subcategory->includes(result)

------------------------------------------------------------------------------------
---
      --deletes an undelivered Order
      deleteOrder(o:Order)
      begin
            o.removeOrder();
            destroy(o);
      end
      pre OrderExists: Order.allInstances->includes(o)
      pre OrderNotDelivered: o.delivered = false
      post OrderIsDeleted: Order.allInstances->excludes(o)
```

```
---------------------------------------------------------------------------------------
---
      --tells an order to execute its delivery
      deliverOrder(o:Order)
      begin
             o.deliver();
      end
      pre OrderExists: Order.allInstances->includes(o)
      pre OrderNotDelivered: o.delivered = false
      post OrderDelivered: o.delivered = true
end

-------*************************************************************************-
------
class ShoppingCart
      attributes
      totalValue:    Real    derived:    productsInCart->iterate(pic;    r:Real=0    |
r+(pic.product.price*pic.amount))
      buyable:       Boolean       derived:       productsInCart->forAll(pic       |
(pic.product.inStock>=pic.amount)) and productsInCart->size>0
      operations

---------------------------------------------------------------------------------------
---
      --adds a product to the cart with an automatic amount of 1
      addToCart(nProduct:Product)
      begin
             insert(nProduct,self) into ProductsInCart;

             --if there is not any amount , the default is 1
             for pin in self.productsInCart do
             if pin.product.name = nProduct.name  then
                          pin.amount := 1;
             end
             end;
      end
      pre NoDuplicateproductinShc: self.product->excludes(nProduct)
      post        NewRelationExists:        self.productsInCart->exists(pin        |
pin.product->includes(nProduct))

-------------------------------------------------------------------------------------
---
      --adds a specific amount of a product to the cart
      addToCartWithAmount(p:Product, a:Integer)
      begin
             insert(p,self) into ProductsInCart;

             for pic in self.productsInCart do
             if pic.product.name = p.name  then
                          pic.amount := a;
             end
             end;
      end
      pre AmountGreaterThanZero: a > 0
      pre AmountDoesNotExceedInStock: a <= p.inStock
      pre NoDuplicateproductinShc: self.product->excludes(p)
      post NewRelationExists: self.product->includes(p)
      post  CorrectAmount:  self.productsInCart->exists(pic  |  pic.amount  =  a  and
pic.product = p)

-------------------------------------------------------------------------------------
---
      --removes a product from the cart
      removeProductFromCart(p:Product)
      begin
             delete(p,self) from ProductsInCart;
      end
      pre ProductIsInCart: self.product->includes(p)
      post         ProductNotInCart:        self.productsInCart->select(pin        |
pin.product->includes(p))->isEmpty()

-------------------------------------------------------------------------------------
---
      --changes the amount of a product in the cart
      changeAmountInCart(p:Product, a:Integer) : Real
      begin
```

```
            for pic in self.productsInCart do
            if pic.product = p  then
                  result := pic.amount; --safe former value
                  pic.amount := a;
            end
            end;
      end
      pre ProductInCart: self.product->includes(p)
      pre AmountGreaterThanZero: a > 0
      pre AmountDoesNotExceedInStock: a <= p.inStock
      post AmountHasChanged: result <> a

--------------------------------------------------------------------------------------
---
      --tells order to create a new order and empties the cart afterwards
      buyCart() : Order
      begin
            declare o:Order;
            o := new Order();
            o.initOrder(self);
            result := o;

            for pic in self.productsInCart do
            delete(pic.product,self) from ProductsInCart;
            end;
      end
      pre CartIsBuyable: self.buyable = true
      post ThisOrderDoesExist: self.owner.order->includes(result)
      post ThisCartIsEmpty: self.productsInCart->size = 0
      post ThisCartIsNotBuyable: self.buyable = false
end

-------***********************************************************************-
------
class Order
      attributes
      totalValue:   Real   derived:   productsBought->iterate(pb;   r:Real   =   0   |
pb.product.price*pb.amount)
      bill: String init: ''
```

```
	delivered: Boolean init: false
	operations

-------------------------------------------------------------------------------------
---
	--creates a bill that lists all products contained in an order
	--this ensures that orders keep information, even if the product they ordered
no longer exists in the shop
	createBill()
	begin
		self.bill := 'Bill for '
			+self.buyer.userName
			+ ' with total value '
			+ self.totalValue.toString()
			+' ---'
			+' containing '
			+  self.productsBought->iterate(pb;  r:String  =  ''  |  r  +
pb.amount.toString()+'x         '+pb.product.name+'         '+pb.product.description+'
'+pb.product.price.toString()+' '+pb.product.manufacturer+' --- ');
	end
    pre HasNoBill: self.bill = ''
    post HasBill: self.bill <> ''
-------------------------------------------------------------------------------------
---
	--delivers an order and creates a bill
	deliver()
	begin
		self.delivered := true;
		self.createBill();
	end
	pre      HasProducts:      self.productsBought->size      >      0      and
self.productsBought->forAll(p | p.amount > 0)
	pre OrderIsNotDelivered: self.delivered = false
	post IsDelivered: self.delivered = true

-------------------------------------------------------------------------------------
---
	--initializes a new order from the contents of a shopping cart
	initOrder(shc:ShoppingCart)
```

```
        begin
              declare ProductSum:Integer;

              self.delivered := false;

              insert(shc.owner,self) into Places;

              for pic in shc.productsInCart do
              insert(self,pic.product) into ProductsBought;
              ProductSum := pic.amount;
              pic.product.inStock := pic.product.inStock - pic.amount;

              for pb in pic.product.productsBought do
                    if pb.product = pic.product and pb.order = self then
                          pb.amount := ProductSum;
                          ProductSum := 0;
                    end;
              end;
              end;
        end
        pre ExistShoppingcart: ShoppingCart.allInstances -> includes(shc)
        pre OrderIsEmpty: self.productsBought->isEmpty()
        pre CartIsbuyable: shc.buyable = true
        post OrderNotEmpty: self.productsBought->size() > 0
        post      OrderHasShoppingCartContents:      self.productsBought.product      =
shc.productsInCart.product

---------------------------------------------------------------------------------------
---
        --removes an orders effect on the system by restoring products instock that
were affected by it
        --even empty orders can be deleted, empty orders can happen, if a product that
is ordered in an undelivered order is deleted from the system
        --@TODO alternative, do not allow deletion of products currently contained in
undelivered orders
        removeOrder()
        begin
              for p in self.productsBought do
                    p.product.inStock := p.product.inStock + p.amount;
                    delete(self,p.product) from ProductsBought;
              end;
       end
       pre AmountOfProductsNotNegative: self.productsBought->size >= 0
       pre OrderUndelivered: self.delivered = false
       post NoMoreProducts: self.productsBought->size = 0
end

-------*****************************************************************-
------
-------associations

associationclass ProductsInCart
between
       Product[*] role product
       ShoppingCart[*] role cart
attributes
       amount : Integer init: 0
end

associationclass ProductsBought
between
       Order[*] role order
       Product[1..*] role product
attributes
       amount : Integer init: 0
end

associationclass Rating
between
       User[1..*] role author
       Product[*] role reviewedItem
attributes
       title : String  init: ''
       text: String  init: ''
       stars : Integer
operations
       initRating(rTitle:String, rText:String, rStars:Integer)
       begin
```

```
        self.title := rTitle;
        self.text :=  rText;
        self.stars := rStars;
    end
    pre  RatingIsEmpty:  self.title  =  ''  and  self.text  =  ''  and  self.stars  =
Undefined
    post RatingIsNotEmpty: self.title <> '' and self.text <> '' and self.stars <>
Undefined
    post RatingStarsInBounds: self.stars >= 0 and self.stars <= 5
end

composition Has between
    User[1] role owner
    ShoppingCart[1] role cart
end

aggregation Subcategory between
    Category[1] role supercategory
    Category[*] role subcategory
end

association Places between
    User[1] role buyer
    Order[*] role order
end

aggregation Includes between
    Category[1..*] role category
    Product[*] role product
end

association Creates between
    Employee[1] role creator
    Category[*] role category
end

association Manages between
    Employee[1] role manager
    Product[*] role product
```

```
end

-------***************************************************************************-
------
-------Invariants

constraints

context Person inv uniqueUserName:
  Person.allInstances->isUnique(userName)

context Person inv haveNameAndfamily:
  self.firstName <> '' and
  self.lastName <> '' and
  self.userName <> '' and
  self.password <> '' and
  self.address <> ''

context User inv rateForBought:
       self.reviewedItem->                          forAll(P                          |
self.order->select(o|o.delivered).product->includes(P))



context User inv uniqueShoppingcart:
    User.allInstances->forAll(u| u.cart->size=1)

context Product inv priceNotZero:
              self.price> 0

--We don not need to implement it
--context Product inv productHasCategory:
--            self.category->size >= 1

context Product inv productHaveNamedescInstock:
       name <> '' and
       description <> '' and
       manufacturer <> '' and
       inStock >=0
```

```
--We don not need to implement it
--context Order inv orderHasProduct:
--           self.product->size >= 1


context Order inv enoughInStock:
       self.productsBought->forAll(   o|       o.order.delivered   =   false   implies
o.product.inStock >= o.amount  )


context Order inv notDuplicateProduct:
       Order.allInstances->forAll(p| p.product->isUnique(name))

context ShoppingCart inv notDuplicateCartProduct:
       ShoppingCart.allInstances->forAll(p| p.product->isUnique(name))

context Order inv productinOrderNotZero:
       self.productsBought->forAll(o| o.amount >0)

context Category inv categoryHaveName:
  self.name <> ''

context Category inv uniqueProductInCategory:
   Category.allInstances->forAll(p| p.product->isUnique(name))

context ShoppingCart inv buyableShoppingcart:
       self.productsInCart->forAll(pin  |   pin.product.inStock  <  pin.amount  implies
self.buyable = false)

context ShoppingCart inv productinSchCNotZero:
       self.productsInCart->forAll(o| o.amount >0)

--we dont have to check it , because it is always true
--context Category inv CategoryGreaterThanSubcategory:
--     Category.allInstances->forAll(c        |        c.subcategory->forAll(c1       |
--(c.subcategory->closure(subcategory).product->size     +     c.product->size)     >=
--(c1.subcategory->closure(subcategory).product->size + c1.product->size)  ))

--we dont need to implement it,use can check this case
```

```
--context Category inv nocyclicSubcategory:
--     Category.allInstances->forAll(c                                                |
c.subcategory->closure(subcategory)->excludes(c))

context Employee inv mustHaveSalary:
       Employee.allInstances->forAll(e| e.salary >0)
```