Perfect Gym

Final Report

Master in Informatics and Computing Engineering

Formal Methods in Software Engineering

Turma 3 – Grupo 4:

Rui Emanuel Cabral de Almeida Quaresma, up201503005@fe.up.pt

Tiago Duarte Carvalho, up201504461@fe.up.pt

Faculty of Engineering of the University of Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

January 7, 2019

Conteúdo

1. Informal system description and list of requirements

1.1 Informal system description

Our system models a club management tool, the PerfectGym.

It has club owners registered as well as their clubs.

Inside a club we can manage its users (clients and employees (trainers and sales representatives)), groups, user access, the crm, gym classes and training sessions, personal training relations, invoices.

The actions that may be performed in a club include: add clients, trainers and sales representatives, add client groups (and add/ remove members to/from them) , add/remove a personal training relation between a client and a trainer, add a gym class, add a client to a gym class, add a training session, add tasks to an employee, set user access, add a newsletter, send messages to one of its users or to multiple users, send messages or offers to a group, create invoices of one or multiple payments active of a client, add leads to the CRM (with or without sales representative assigned), transform a lead into a client, get Reports on club statistics, client statistics and employee statistics, add products to the club, and get all of this information from the club.

All of these can also be performed in the correspondent instance. The actions in the club require a certain user access (either owner or at least employee)

Beside these, a user can send messages to other users. A client can send messages to one of its groups and perform purchases/payments of gym fees, personal training fees and products of the gym.

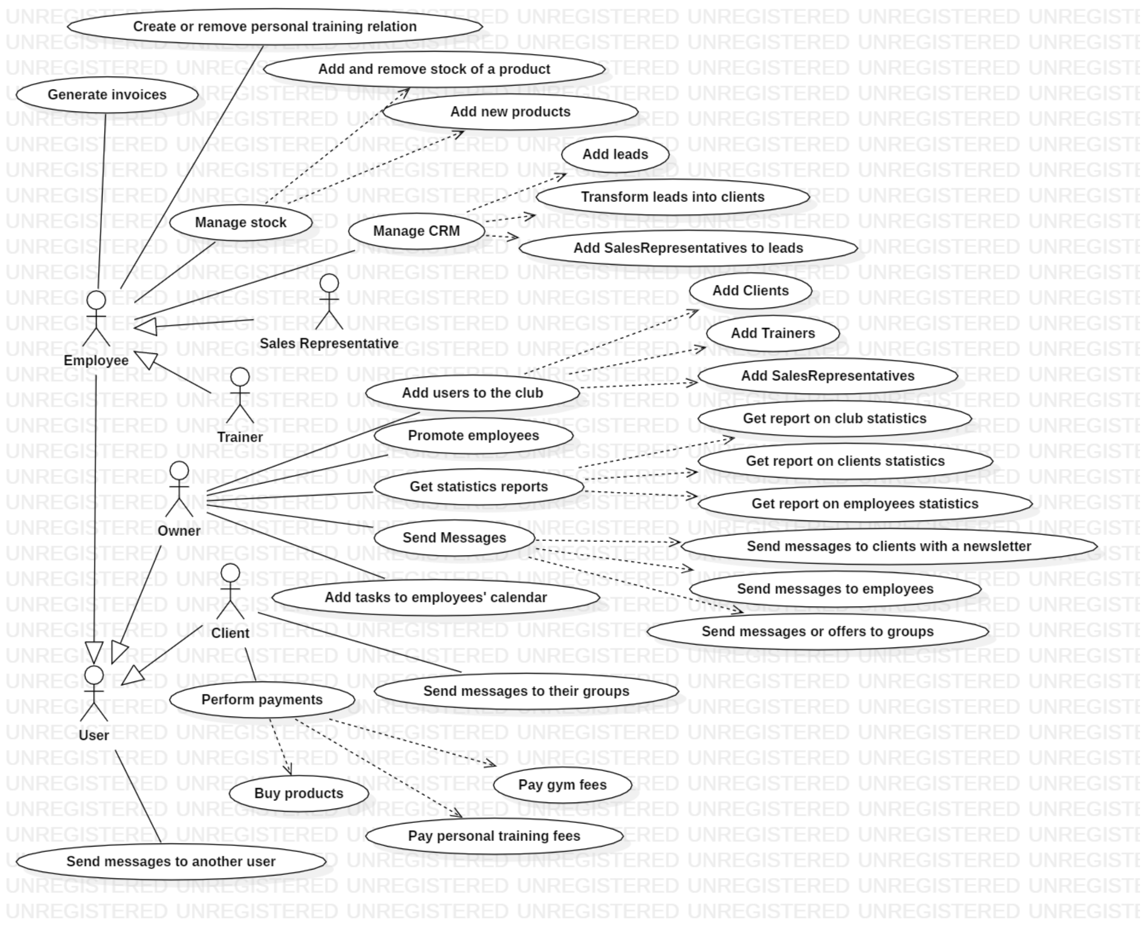
1.2 List of Requirements

Id	Priority	Description
R1	Mandatory	Create owners in the PerfectGym and create clubs assigning them to their owner.
R2	Mandatory	The club Owner must be able to add new clients, trainers and sales representatives to the club.
R3	Mandatory	The club Owner must be able to give employees Owner access.
R4	Mandatory	The club Owner must be able to get reports on club, clients and employees statistics.
R5	Mandatory	One of club employees must be able to manage the CRM by adding leads, transforming leads into new clients and assigning sales representatives to leads.
R6	Mandatory	The club Owner must be able to send messages to the clients (with a newsletter attached) and employees (individually or to all of them). Must also be able to send messages and offers to a group.
R7	Mandatory	One of club employees must be able to generate invoices with one or multiple active payments (of a given type) of one of its clients
R8	Mandatory	The club Owner must be able to add tasks (gym classes, training sessions, or others) to one of its employees calendar.
R9	Optional	One of club employees may create/remove a personal training relation between a trainer and a client.
R10	Optional	One of club employees may be able to add products to the club, as well as add stock and remove a product.
R11	Optional	A user may be able to send messages to other users and delete messages received.
R12	Optional	A client may be able to send messages to one of its groups.
R13	Optional	A client may be able to perform purchases/payments of gym fees, personal training fees and products of the gym.

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

2. Visual UML model

2.1 Use case model



Create or remove personal training relation
Add and remove stock of a product
Generate invoices
Add new products
Add leads
Transform leads into clients
Manage stock
Manage CRM
Add SalesRepresentatives to leads
Add Clients
Add Trainers
Employee
Sales Representative
Add SalesRepresentatives
Add users to the club
Trainer
Get report on club statistics
Promote employees
Get report on clients statistics
Get statistics reports
Owner
Get report on employees statistics
Send Messages
Send messages to clients with a newsletter
Send messages to employees
Add tasks to employees' calendar
Client
Send messages or offers to groups
Send messages to their groups
Perform payments
User
Pay gym fees
Buy products
Pay personal training fees
Send messages to another user

The major use case scenarios are described next.

Scenario	**Setup PerfectGym**
Description	Scenario for configuring PerfectGym.
Pre-conditions	None
Post-conditions	1. PerfectGym has clubs and owners. 2. Clubs cannot have the same name.
Steps	1. Add owners 2. Add clubs
Exceptions	none

Scenario	**Setup Club**
Description	Scenario for configuring a club.
Pre-conditions	1. Fee must be positive 2. Club name must be unique
Post-conditions	1. Club has clients, trainers and sales representatives. 2. Club has a newsletter 3. Club has products 4. Club has a fee associated 5. Club has an owner 6. Club has a CRM
Steps	1. Add clients 2. Add trainers 3. Add sales representatives 4. Add a newsletter
Exceptions	none

Scenario	**Add personal training sessions**
Description	Scenario for adding a new training session.
Pre-conditions	1. User responsible for this action must have employee access
Post-conditions	1. Club has a new training session. 2. Trainer has a new training session. 3. Client has a new training session. 4. Trainer has a new trainee, 5. Client has a trainer. 6 Trainer has a new task on its calendar
Steps	1. Add a new personal training relation if it does not exists 2. Add a training session
Exceptions	none

Scenario	**Add gym classes**
Description	Scenario for adding a new gym class.
Pre-conditions	1. User responsible for this action must have employee access
Post-conditions	1. Club has a new gym class session. 2 Trainer has a new task on its calendar
Steps	1. Add a gym class

Exceptions	none

Scenario	**Add group**
Description	Scenario for adding a new group
Pre-conditions	1. User responsible for this action must have employee access 2. Number of clients set in the creation of the group must be >= 1
Post-conditions	1. Club has a new group 2. Groups cannot have the same name
Steps	1. Select the clients 2. Create the group
Exceptions	None

Scenario	**Set user access**
Description	Scenario for giving an employee Owner access
Pre-conditions	1. User responsible for this action must have owner access 2. Target user must be an employee
Post-conditions	1. Target user has Owner access
Steps	1. Set user access
Exceptions	None

Scenario	**Get Reports**
Description	Scenario for getting reports
Pre-conditions	1. User responsible for this action must have owner access
Post-conditions	None
Steps	1. Get Report
Exceptions	None

Scenario	**Manage CRM**
Description	Scenario for adding leads to CRM
Pre-conditions	1. User responsible for this action must be have employee access
Post-conditions	1. CRM has leads
Steps	1. Add lead to CRM 2. Transform lead into client
Exceptions	None

Scenario	**Add invoice**
Description	Scenario for adding an invoice
Pre-conditions	1. User responsible for this action must be have employee access 2. Payments selected must belong to the given client and must be of the given type 3. Number of payments set in the creation of the invoice must be >= 1
Post-conditions	1. Invoice added to the club 2. Payments moved to clients payments history

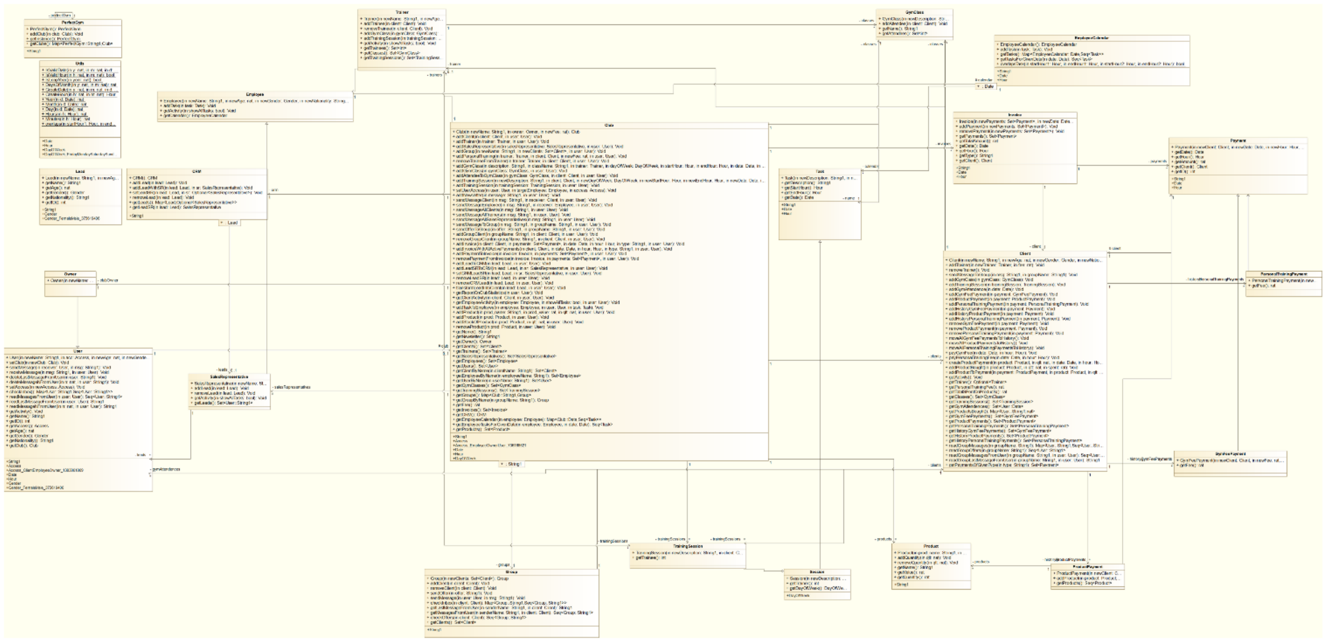
Steps	1. Choose client 2. Choose type 3. Select Payments 4. Set date
Exceptions	None

Scenario	**Add products**
Description	Scenario for adding a product to the club
Pre-conditions	1. User responsible for this action must be have employee access 2. Product may not already exist in the club
Post-conditions	1. Product added to the club
Steps	1. Create product2. Choose type
Exceptions	None

Scenario	**Send message**
Description	Scenario for sending a message
Pre-conditions	none
Post-conditions	None
Steps	1. Choose the destination of the message 2. Send message
Exceptions	None

Scenario	**Make payment**
Description	Scenario for making a payment
Pre-conditions	none
Post-conditions	1. Payment added to the client active payments
Steps	1. Choose the type of payment 2. Make the payment
Exceptions	None

2.2 Class Model



Class	Description
`Client`	Defines a client (subclass of User).
`Club`	Defines a club.
`CRM`	Defines a CRM.
`Employee`	Defines an employee (subclass of User, Superclass).
`EmployeeCalendar`	Defines an employeeCalendar.
`Group`	Defines a group.
`GymClass`	Defines a gymClass (subclass of Session).
`GymFeePayment`	Defines a gymFeePayment (subclass of Payment).
`Invoice`	Defines an invoice.
`Lead`	Defines a lead.
`Owner`	Defines an owner (subclass of User).
`Payment`	Defines a payment.
`PerfectGym`	Defines a perfectGym.
`PersonalTrainingPayment`	Defines a personalTrainingPayment (subclass of Payment).
`Product`	Defines a product.
`ProductPayment`	Defines a productPayment (subclass of Payment).
`SalesRepresentative`	Defines a salesRepresentative (subclass of Employee).
`Session`	Defines a session (subclass of Task, Superclass).
`Task`	Defines a task (Superclass).
`Trainer`	Defines a trainer (subclass of Employee).
`TrainingSession`	Defines a trainingSession (subclass of Session).
`User`	Defines a user (Superclass).
`Utils`	Defines multiple utils (functions and types).

Class	Description
MyTestCase	Superclass for the test classes; defines assertEquals, assertTrue and assertFalse
MyTestRunner	Calls all the tests.
ClientTest	Defines tests to the Client class.
ClubTest	Defines tests to the Club class.
EmployeeTest	Defines tests to the Employee class.
GroupTest	Defines tests to the Group class.
GymClassTest	Defines tests to the GymClass class.
InvoiceTest	Defines tests to the Invoice class.
LeadTest	Defines tests to the Lead class.
PaymentTest	Defines tests to the Payment class.
PerfectGymTest	Defines tests to the PerfectGym class.
ProductTest	Defines tests to the Product class.
SalesRepresentativeTest	Defines tests to the SalesRepresentative class.
SessionTest	Defines tests to the Session class.
TaskTest	Defines tests to the Task class.
UseCasesTest	Defines tests for the scenarios described above.

3. Formal VDM++ model

3.1 CRM

```
class CRM
types
  public String1 = seq of char; instance
variables
  private leads: map Lead to [SalesRepresentative] := {|->};
operations
  /**
    * CRM constructor
    */

  public CRM: () ==> CRM CRM()
    == return self post leads = {|-
    >};

    /**
      * Add lead
      */

    public addLead: Lead ==> () addLead(lead) ==
  (
    leads := leads munion {lead |-> nil}
  )
    pre lead not in set dom leads;

  /**
    * Add lead with sales representative
    */

    public addLeadWithSR: Lead * SalesRepresentative ==> () addLeadWithSR(lead, sr) ==
    (
    leads := leads munion {lead |-> sr}; sr.addLead(lead);
    )
    pre lead not in set dom leads;

  /**
    * Set lead with sales representative
    */

    public setLeadSR: Lead * [SalesRepresentative] ==> () setLeadSR(lead, sr) ==
    (
      if leads(lead) <> nil then
    (
      leads(lead).removeLead(lead);

    );

    if(sr <> nil) then
      sr.addLead(lead);

      leads(lead) := sr;

    )
    pre lead in set dom leads;

  /**
    * Remove lead
    */

  public removeLead: Lead ==> ()
  removeLead(lead) ==
  (
    dcl newLeads: map Lead to [SalesRepresentative] := {|->};
    for all l in set dom leads do if(lead <> l) then
        newLeads := newLeads munion {l|->leads(l)};

    if leads(lead) <> nil then
      leads(lead).removeLead(lead);

    leads := newLeads;
  )
  pre leads <> {|->} and lead in set dom leads;

  -- GETTERS

  /**
    *  Gets the client leads
    *
    *  @return map Lead to [SalesRepresentative]
    */
```

```
public pure getLeads: () ==> map Lead to [SalesRepresentative] getLeads() == return
leads
post RESULT = leads;

/**
  *  Gets one of the client leads sales representative
  *
  *  @return SalesRepresentative
  */

public pure getLeadSR: Lead ==> SalesRepresentative getLeadSR(lead) == return
leads(lead)
pre lead in set dom leads and
leads(lead) <> nil post RESULT =
leads(lead);


 end CRM
```

3.2 Client

```
class Client is subclass of User

 instance variables

   private trainer: [Trainer];
   private personalTrainingFee: rat; private classes: set
   of GymClass := {};
   private trainingSessions: set of TrainingSession := {};
   private gymAttendences: set of Date := {};
   private productsBought: map String1 to nat := {|->};
   private totalSpentOnProducts: rat;

   private gymFeePayments: set of GymFeePayment := {};
   private productPayments: set of ProductPayment := {};
   private personalTrainingPayments: set of PersonalTrainingPayment := {};

   private historyGymFeePayments: set of GymFeePayment := {};
   private historyProductPayments: set of ProductPayment := {};
   private historyPersonalTrainingPayments: set of PersonalTrainingPayment := {};

   inv not exists c1, c2 in set classes &
     c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1.getDate() = c2.getDate() and
     Utils‘overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());
   inv not exists t1, t2 in set trainingSessions &
     t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
     t1.getDate() = t2.getDate() and Utils‘overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());

 operations

   /**
     * Client constructor
     */

   public Client: String1 * nat * Gender * String1 ==> Client Client(newName, newAge,
     newGender, newNationality) ==
     (
       trainer := nil; personalTrainingFee := 0;
       totalSpentOnProducts := 0;
       User(newName, <Client>, newAge, newGender, newNationality);
     )
     post trainer = nil and personalTrainingFee = 0 and totalSpentOnProducts = 0 and classes = {}
            and trainingSessions = {} and gymAttendences = {} and productsBought = {|->} and
       gymFeePayments = {} and productPayments = {} and personalTrainingPayments = {} and
              historyGymFeePayments = {} and
     historyProductPayments = {} and historyPersonalTrainingPayments = {};

     /**
       * Adds a trainer to the client
       */

   public addTrainer: Trainer * rat==> ()
   addTrainer(newTrainer, fee) ==
   (
     if trainer <> nil then
       trainer.removeTrainee(self);

     trainer := newTrainer; personalTrainingFee :=
     fee;
   )
   pre newTrainer in set club.getTrainers() and fee > 0
   post trainer = newTrainer;

     /**
       * Remove a trainer from the client
       */

   public removeTrainer: () ==> ()
   removeTrainer() ==
   (
     trainer := nil; personalTrainingFee := 0;
   )
   pre trainer <> nil post
   trainer = nil;


   -- MESSAGES TO/FROM GROUP

   /**
     * Send message to one of its groups
     */

   public sendMessageToGroup: String1 * String1 ==> ()
   sendMessageToGroup(msg, groupName) == club.getGroupByName(groupName).sendMessage(self, msg)
   pre msg <> "" and groupName <> "" and groupName in set dom club.getGroups() and self in set club
           .getClients();
```

```
-- GYM CLASSES

/**
  * Adds a gym class to this client
  */

public addGymClass: GymClass ==> ()
addGymClass(gymClass) ==
(
  dcl classDate: Date := gymClass.getDate(); classes := classes
  union {gymClass}; addGymAttendence(classDate);
)
pre gymClass not in set classes
post classes = classes~ union {gymClass};

-- TRAINING SESSIONS

/**
  * Adds a training session to this client
  */

public addTrainingSession: TrainingSession ==> () addTrainingSession(trainingSession) ==
(
  dcl trainingSessionDate: Date := trainingSession.getDate(); trainingSessions :=
  trainingSessions union {trainingSession}; addGymAttendence(trainingSessionDate);
)
pre trainingSession not in set trainingSessions
post trainingSessions = trainingSessions~ union {trainingSession};

-- GYM ATTENDENCES

/**
  * Adds a gym attendence (date) to this client
  */

public addGymAttendence: Date ==> ()
addGymAttendence(date) == gymAttendences := gymAttendences union {date}
pre date not in set gymAttendences
post gymAttendences = gymAttendences~ union {date};

-- PAYMENTS

/**
  * Adds a gym fee payment to this client active payments
  */

public addGymFeePayment: GymFeePayment ==> ()
addGymFeePayment(payment) == gymFeePayments := gymFeePayments union {payment}
pre payment not in set gymFeePayments
post gymFeePayments = gymFeePayments~ union {payment};

/**
  * Adds a product payment to this client active payments
  */

public addProductPayment: ProductPayment ==> ()
addProductPayment(payment) == productPayments := productPayments union {payment}
pre payment not in set productPayments
post productPayments = productPayments~ union {payment};

/**
```

```
  * Adds a personal training payment to this client active payments
  */

public addPersonalTrainingPayment: PersonalTrainingPayment ==> () addPersonalTrainingPayment(payment) ==
personalTrainingPayments := personalTrainingPayments
        union {payment}
pre payment not in set personalTrainingPayments
post personalTrainingPayments = personalTrainingPayments~ union {payment};

/**
  * Adds a gym fee payment to this client history
  */

public addHistoryGymFeePayment: Payment ==> ()
addHistoryGymFeePayment(payment) == historyGymFeePayments := historyGymFeePayments union { payment}
pre payment not in set historyGymFeePayments
post historyGymFeePayments = historyGymFeePayments~ union {payment};

/**
  * Adds a product payment to this client history
  */

public addHistoryProductPayment: Payment ==> ()
addHistoryProductPayment(payment) == historyProductPayments := historyProductPayments union { payment}
pre payment not in set historyProductPayments
post historyProductPayments = historyProductPayments~ union {payment};

/**
  * Adds a personal training payment to this client history
  */

public addHistoryPersonalTrainingPayment: Payment ==> () addHistoryPersonalTrainingPayment(payment) ==
historyPersonalTrainingPayments :=
                  historyPersonalTrainingPayments union {payment}
pre payment not in set historyPersonalTrainingPayments
post historyPersonalTrainingPayments = historyPersonalTrainingPayments~ union {payment};

/**
  * Removes a gym fee payment from this client active payments and adds it to its history
  */

public removeGymFeePayment: Payment ==> () removeGymFeePayment(payment) ==
(
  addHistoryGymFeePayment(payment); gymFeePayments :=
  gymFeePayments \ {payment};
)
pre gymFeePayments <> {} and payment in set gymFeePayments
post gymFeePayments = gymFeePayments~ \ {payment};

/**
  * Removes a product payment from this client active payments and adds it to its history
  */

public removeProductPayment: Payment ==> () removeProductPayment(payment) ==
(
  addHistoryProductPayment(payment); productPayments :=
  productPayments \ {payment};
)
pre productPayments <> {} and payment in set productPayments
post productPayments = productPayments~ \ {payment};
```

```
/**
  * Removes a personal training payment from this client active payments and adds it to its history
  */

public removePersonalTrainingPayment: Payment ==> ()
removePersonalTrainingPayment(payment) ==
(
  addHistoryPersonalTrainingPayment(payment); personalTrainingPayments :=
  personalTrainingPayments \ {payment};
)
pre personalTrainingPayments <> {} and payment in set personalTrainingPayments
post personalTrainingPayments = personalTrainingPayments~ \ {payment};

/**
  * Moves all gym fee payments from this client active payments and adds it to its history
  */

public moveAllGymFeePaymentsToHistory: () ==> () moveAllGymFeePaymentsToHistory()
==
(
  for all payment in set gymFeePayments do
    removeGymFeePayment(payment);
)
post gymFeePayments = {};

/**
  * Moves all product payments from this client active payments and adds it to its history
  */

public moveAllProductPaymentsToHistory: () ==> () moveAllProductPaymentsToHistory()
==
(
  for all payment in set productPayments do
  (
    removeProductPayment(payment);
  )
)
post productPayments = {};

/**
  * Moves all personal training payments from this client active payments and adds it to its history
  */

public moveAllPersonalTrainingPaymentsToHistory: () ==> ()
moveAllPersonalTrainingPaymentsToHistory() ==
(
  for all payment in set personalTrainingPayments do
  (
    removePersonalTrainingPayment(payment);
  );
)
post personalTrainingPayments = {};

/**
  * Create gym fee payment and add it to the active gym fee payments
  */

public payGymFee: Date * Hour ==> ()
payGymFee(date, hour) ==
(
  dcl payment: GymFeePayment := new GymFeePayment(self, club.getFee(), date, hour); addGymFeePayment(payment);
);
```

```
/**
  * Create personal training payment and add it to the active personal training payments
  */

public payPersonalTrainingFee: Date * Hour ==> () payPersonalTrainingFee(date, hour) ==
(
  dcl payment: PersonalTrainingPayment := new PersonalTrainingPayment(self, personalTrainingFee, date, hour);
  addPersonalTrainingPayment(payment);
);


public createProductPayment: Product * nat * Date * Hour ==> ()
createProductPayment(product, qtt, date, hour) ==
(
  dcl payment: ProductPayment := new ProductPayment(self, product, qtt, date, hour); addProductPayment(payment);
);
-- PRODUCTS

/**
  * Add a product bought to this client
  */

public addProductBought: Product * nat * rat ==> () addProductBought(product, qtt,
spent) ==
(
  totalSpentOnProducts:= totalSpentOnProducts + spent;
  if product.getName() in set dom productsBought then
    productsBought(product.getName()) := productsBought(product.getName()) + qtt
  else
    productsBought := productsBought munion {product.getName() |-> qtt};
)
pre spent > 0
post totalSpentOnProducts = totalSpentOnProducts~ + spent;

/**
  * Add a product to a payment made by this client
  */

public addProductToPayment: ProductPayment * Product * nat ==> () addProductToPayment(payment,
product, qtt) == payment.addProduct(product, qtt) pre payment.getClient() = self;

-- GETTERS

/**
  * Get client activity
  */

  public getActivity: () ==> () getActivity() ==
  (
    dcl numClasses: nat := card classes;
    dcl numTrainingSessiosn: nat := card trainingSessions;
    dcl numAttendences: nat := card gymAttendences;
    dcl personalTrainer: String1 := "None";
    dcl numProductsBought : nat := card dom productsBought;

    if trainer <> nil then
      personalTrainer:= trainer.getName();
    IO‘println("********* CLIENT STATISTICS *********"); IO‘print("Personal
    trainer: "); IO‘println(personalTrainer);
    IO‘print("Number of gym classes: " );
    IO‘println(numClasses);
    IO‘print("Number of training sessions: " );
    IO‘println(numTrainingSessiosn); IO‘print("Number of gym
    attendences: " ); IO‘println(numAttendences);
    IO‘print("Number of different products bought: "); IO‘println(numProductsBought);
    IO‘println(""); IO‘println("************************************");
  );

/**
  *  Gets the client trainer
  *
  *  @return trainer
  */

public pure getTrainer: () ==> [Trainer] getTrainer() == return
trainer
post RESULT = trainer;

/**
  *  Gets the client personal training fee
  *
  *  @return personalTrainingFee
  */

public pure getPersonalTrainingFee: () ==> rat getPersonalTrainingFee() ==
return personalTrainingFee post RESULT = personalTrainingFee;

/**
  *  Gets the client total spent on products
  *
  *  @return totalSpentOnProducts
  */

public pure getTotalSPentOnProducts: () ==> rat getTotalSPentOnProducts() ==
return totalSpentOnProducts post RESULT = totalSpentOnProducts;

/**
  *  Gets the client classes
  *
  *  @return set of GymClass
  */

public pure getClasses: () ==> set of GymClass getClasses() ==
return classes
post RESULT = classes;

/**
  *  Gets the client training sessions
  *
  *  @return set of TrainingSession
  */

public pure getTrainingSessions: () ==> set of TrainingSession getTrainingSessions() ==
return trainingSessions
post RESULT = trainingSessions;
```

```
/**
 * Gets the client gym attendences
 *
 * @return set of Date
 */

public pure getGymAttendences: () ==> set of Date getGymAttendences() == return
gymAttendences
post RESULT = gymAttendences;

/**
 * Gets the client products bought
 *
 * @return map String1 to nat
 */

public pure getProductsBought: () ==> map String1 to nat
getProductsBought() == return productsBought
post RESULT = productsBought;

/**
 * Gets the client gym fee paymenrs
 *
 * @return set of GymFeePayment
 */

public pure getGymFeePayments: () ==> set of GymFeePayment getGymFeePayments() ==
return gymFeePayments
post RESULT = gymFeePayments;

/**
 * Gets the client product payments
 *
 * @return set of ProductPayment
 */

public pure getProductPayments: () ==> set of ProductPayment getProductPayments() ==
return productPayments
post RESULT = productPayments;

/**
 * Gets the client personal training payments
 *
 * @return set of PersonalTrainingPayment
 */

public pure getPersonalTrainingPayments: () ==> set of PersonalTrainingPayment getPersonalTrainingPayments() == return
personalTrainingPayments
post RESULT = personalTrainingPayments;


/**
 * Gets the client history of gym fee payments
 *
 * @return set of GymFeePayment
 */

public pure getHistoryGymFeePayments: () ==> set of GymFeePayment
getHistoryGymFeePayments() == return historyGymFeePayments
post RESULT = historyGymFeePayments;

/**
 * Gets the client history of product payments
 *
 * @return set of ProductPayment
```

```
  */

public pure getHistoryProductPayments: () ==> set of ProductPayment
getHistoryProductPayments() == return historyProductPayments
post RESULT = historyProductPayments;

/**
  * Gets the client history of personal training payments
  *
  * @return set of PersonalTrainingPayment
  */

public pure getHistoryPersonalTrainingPayments: () ==> set of PersonalTrainingPayment getHistoryPersonalTrainingPayments() ==
return historyPersonalTrainingPayments
post RESULT = historyPersonalTrainingPayments;

/**
  * Get the messages from one of its groups
  *
  * @return map String1 to seq of String1
  */

  public pure readGroupMessages: String1 ==> map String1 to seq of String1 readGroupMessages(groupName) ==
return club.getGroupByName(groupName).checkInbox(self) pre self in set club.getClients() and
  groupName in set dom club.getGroups()
post RESULT = club.getGroupByName(groupName).checkInbox(self);

/**
  * Get the offers from one of its groups
  *
  * @return seq of String1
  */

  public pure readGroupOffers: String1 ==> seq of String1
readGroupOffers(groupName) == return club.getGroupByName(groupName).checkOffers(self)
pre self in set club.getClients() and
  groupName in set dom club.getGroups()
post RESULT = club.getGroupByName(groupName).checkOffers(self);

/**
  * Get the messages, from one of its groups, sent by a given user
  *
  * @return seq of String1
  */

  public pure readGroupMessagesFromUser: String1 * User ==> seq of String1 readGroupMessagesFromUser(groupName, user) == return
club.getGroupByName(groupName).
        getMessagesFromUser(user.getName(), self)
pre
  self in set club.getClients() and
  groupName in set dom club.getGroups() and
  club.getGroupByName(groupName).getMessagesFromUser(user.getName(), self) <> [] and user in set club.getUsers()
post RESULT = club.getGroupByName(groupName).getMessagesFromUser(user.getName(), self);

/**
  * Get the last message, from one of its groups, sent by a given user
  *
  * @return String1
  */

  public pure readGroupLastMessageFromUser: String1 * User ==> String1 readGroupLastMessageFromUser(groupName, user) == return
club.getGroupByName(groupName).
        getLastMessageFromUser(user.getName(), self)
```

```
 pre
  self in set club.getClients() and
  groupName in set dom club.getGroups() and
  club.getGroupByName(groupName).getMessagesFromUser(user.getName(), self) <> [] and user in set club.getUsers()
 post RESULT = club.getGroupByName(groupName).getLastMessageFromUser(user.getName(), self);

 /**
  *  Gets the client active payments of a given type
  *
  *  @return set of Payment
  */

 public pure getPaymentsOfGivenType: String1 ==> set of Payment
 getPaymentsOfGivenType(type) ==
 (
  cases type:
    "product" -> return productPayments, "gymFee" ->
    return gymFeePayments,
    "personalTraining" -> return personalTrainingPayments
  end; return
  {};
 )
 pre type in set {"product", "gymFee", "personalTraining"};

  end Client
```

3.3 Club

```
class Club
types

 public String1 = seq of char;
 public Access = <Owner> | <Employee> | <User>;
 public Date = Utils‘Date;
 public Hour= Utils‘Hour;
 public DayOfWeek = Utils‘DayOfWeek;

instance variables

 private name: String1;
 private newsletter: [String1];

 private clients: set of Client := {};
 private salesRepresentatives: set of SalesRepresentative := {};
 private trainers: set of Trainer := {}; private groups: map
 String1 to Group := {|->}; private classes: set of GymClass := {};
 private trainingSessions: set of TrainingSession := {};

 private invoices: set of Invoice := {};
 private products: set of Product := {};

 private fee: rat;

 private crm: CRM;
 private clubOwner: Owner;

 inv not exists c1, c2 in set classes &
  c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1.getDate() = c2.getDate() and
  Utils‘overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());
 inv not exists t1, t2 in set trainingSessions &
  t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
  t1.getDate() = t2.getDate() and Utils‘overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());

 inv not exists p1, p2 in set products & p1 <> p2 and
  p1.getName() = p2.getName();

operations

 /**
  * Club constructor
  */

  public Club: String1 * Owner * rat ==> Club Club(newName,
  owner, newFee) ==
 (
  newsletter := nil; name :=
    newName; clubOwner :=
    owner;
    clubOwner.setClub(self); crm :=
  new CRM();
  fee := newFee;
    return self
 )
 pre newFee > 0 and newName <> ""
  post
    name = newName and
    newsletter = nil and clients
    = {} and
    salesRepresentatives = {} and
    trainers = {} and groups = {|-
    >} and clubOwner = owner
    and
  classes = {} and fee =
  newFee and invoices = {}
  and products = {} and
  trainingSessions = {};

    /**
      * Add a new client to the club
      */

  public addClient: Client * User ==> () addClient(client,
  user) ==
  (
    clients := clients union {client}; client.setClub(self);
  )
  pre client not in set clients and (isAtLeastEmployee(user)) and user in set getUsers()
  post clients = clients~ union {client};

  /**
    * Add a new trainer to the club
    */

  public addTrainer: Trainer * User ==> () addTrainer(trainer,
  user) ==
  (
    trainers := trainers union {trainer};
    trainer.setClub(self);
  )
  pre trainer not in set trainers and isOwner(user) and user in set getUsers()
  post trainers = trainers~ union {trainer};
```

```
/**
  * Add a new sales representative to the club
  */

public addSalesRepresentative: SalesRepresentative * User ==> () addSalesRepresentative(salesRepresentative, user) ==
(
  salesRepresentatives := salesRepresentatives union {salesRepresentative}; salesRepresentative.setClub(self);
)
pre salesRepresentative not in set salesRepresentatives and isOwner(user) and user in set
        getUsers()
post salesRepresentatives = salesRepresentatives~ union {salesRepresentative};


/**
  * Add a new group, by giving its name and its first clients, to the club
  */

public addGroup: String1 * set of Client * User ==> () addGroup(newName, newClients,
user) ==
    groups := groups munion {newName |-> new Group(newClients)}
pre newName <> "" and newName not in set dom groups and user in set getUsers() and forall c in set newClients & c
  in set clients and
  (not exists c1, c2 in set newClients & c1<>c2 and c1.getID() = c2.getID()) and ( isAtLeastEmployee(user))
post groups = groups~ munion {newName |-> groups(newName)};

/**
  * Add a client of personal training to a trainer
  */

public addPersonalTraining: Trainer * Client * rat * User ==> ()
addPersonalTraining(trainer, client, newFee, user) ==
(
  trainer.addTrainee(client); client.addTrainer(trainer,
  newFee);
)
pre
  trainer in set trainers and
  client in set clients and
  client.getID() not in set trainer.getTrainees() and user in set getUsers()
  and (isAtLeastEmployee(user));

/**
  * Remove trainee from trainer
  */

public removeTraineeFromTrainer: Trainer * Client * User==> ()
removeTraineeFromTrainer(trainer, client, user) ==
(
  trainer.removeTrainee(client);
  client.removeTrainer();
)
pre trainer in set trainers and client in set clients and
  (isAtLeastEmployee(user));

/**
  * Add a new gym class to the club
  */

public addGymClass: String1 * String1 * Trainer * DayOfWeek * Hour * Hour * Date * User ==> () addGymClass(description, className,
trainer, dayOfWeek, startHour, endHour, date, user) ==
(
  dcl gymClass: GymClass := new GymClass(description, className, trainer, dayOfWeek,startHour, endHour, date);
  classes := classes union {gymClass};
  trainer.addGymClass(gymClass);
  trainer.addTask(gymClass);
)
pre (isAtLeastEmployee(user)) and user in set getUsers() and description <> "" and className <> ""
post not exists c1, c2 in set classes & c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1. getDate() = c2.getDate() and
  Utils‘overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());

/**
  * Add an existing gym class to the club
  */
public addGymClass: GymClass * User ==> ()
addGymClass(gymClass, user) == classes := classes union {gymClass}
pre (isAtLeastEmployee(user)) and user in set getUsers() and gymClass not in set classes
post not exists c1, c2 in set classes & c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1. getDate() = c2.getDate() and
  Utils‘overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());

/**
  * Add a trainee to a class
  */

public addAttendeeToGymClass: GymClass * Client * User ==> () addAttendeeToGymClass(gymClass, client,
user) == gymClass.addAttendee(client)
pre client in set getClients() and (isAtLeastEmployee(user)) and user in set getUsers();

/**
  * Add a new training session to the club
  */

public addTrainingSession: String1 * Client * DayOfWeek * Hour * Hour * Date * User ==> () addTrainingSession(newDescription, client,
newDayOfWeek, newStartHour, newEndHour, newDate,
        user) ==
(
  dcl trainingSession: TrainingSession := new TrainingSession(newDescription, client, newDayOfWeek, newStartHour, newEndHour,
          newDate);
  trainingSessions := trainingSessions union {trainingSession};
  client.addTrainingSession(trainingSession);
  client.getTrainer().addTask(trainingSession);
  client.getTrainer().addTrainingSession(trainingSession);
)
pre (isAtLeastEmployee(user)) and user in set getUsers() and newDescription <> ""
post not exists t1, t2 in set trainingSessions &
  t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
  t1.getDate() = t2.getDate() and Utils‘overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());

/**
  * Add an existing training session to the club
  */
public addTrainingSession: TrainingSession * User ==> () addTrainingSession(trainingSession, user) == trainingSessions :=
trainingSessions union {
        trainingSession}
pre (isAtLeastEmployee(user)) and user in set getUsers() and trainingSession not in set
        trainingSessions
post not exists t1, t2 in set trainingSessions &
  t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
  t1.getDate() = t2.getDate() and Utils‘overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());
```

```
/**
  * Set the access of a user
  */

public setUserAccess: User * Employee * Access ==> ()
setUserAccess(user, targetEmployee, access) == targetEmployee.setAccess(access)
pre
  isOwner(user) and targetEmployee.getAccess() <>
  <Owner> and user in set getUsers() and
  access = <Owner> and
  targetEmployee in set getEmployees();

/**
  * Add newsletter to the club
  */

public addNewsletter: String1 * User==> () addNewsletter(message, user) ==
newsletter := message
pre isOwner(user) and user in set getUsers() and message <> "";

/**
  * Send message to a client as club owner
  */

public sendMessageClient: String1 * Client * User ==> () sendMessageClient(msg, receiver,
user) ==
(
  receiver.receiveMessage(msg, user);
  if newsletter <> nil then
    receiver.receiveMessage(newsletter, user);
)
pre msg <> "" and receiver in set clients and isOwner(user) and user in set getUsers();

/**
  * Send message to an employee as club owner
  */

public sendMessageEmployee: String1 * Employee * User ==> () sendMessageEmployee(msg, receiver, user)
== receiver.receiveMessage(msg, user)
pre msg <> "" and receiver in set getEmployees() and isOwner(user) and user in set getUsers();

/**
  * Send message to all clients as club owner
  */

public sendMessageAllClients: String1 * User==> () sendMessageAllClients(msg, user) ==
(
  for all client in set clients do
    sendMessageClient(msg, client, user);
)
pre msg <> "" and card clients > 0 and isOwner(user) and user in set getUsers();

/**
  * Send message to all trainers as club owner
  */

public sendMessageAllTrainers: String1 * User ==> () sendMessageAllTrainers(msg, user)
==
(
  for all trainer in set trainers do
    trainer.receiveMessage(msg, user)
)
```

```
pre msg <> "" and card trainers > 0 and isOwner(user) and user in set getUsers();

/**
  * Send message to all sales representatives
  */

public sendMessageAllSalesRepresentatives: String1 * User ==> () sendMessageAllSalesRepresentatives(msg, user) ==
(
  for all salesRepresentative in set salesRepresentatives do
    salesRepresentative.receiveMessage(msg, user)
)
pre msg <> "" and card salesRepresentatives > 0 and isOwner(user) and user in set getUsers();

-- GROUPS

/**
  * Send message to a group
  */

public sendMessageToGroup: String1 * String1 * User ==> ()
sendMessageToGroup(msg, groupName, user) == groups(groupName).sendMessage(user, msg)
pre msg <> "" and groupName <> "" and
  groupName in set dom groups and (user in set clients or isOwner(user)) and user in set
          getUsers();

/**
  * Send offer to a group
  */

public sendOfferToGroup : String1 * String1 * User ==> () sendOfferToGroup(offer, groupName, user) ==
groups(groupName).sendOffer(offer) pre offer <> "" and groupName <> "" and
  groupName in set dom groups and (isAtLeastEmployee(user)) and user in set getUsers();

/**
  * Add client to a group
  */

  public addGroupClient: String1 * Client * User ==> () addGroupClient(groupName, client, user) ==
  groups(groupName).addClient(client) pre groupName <> "" and
    client in set clients and groupName in set dom groups and isOwner(user) and user in set
              getUsers();

    /**
  * Remove client from a group
  */

  public removeGroupClient: String1 * Client * User ==> () removeGroupClient(groupName, client, user) ==
  groups(groupName).removeClient(client) pre groupName <> "" and
    client in set clients and groupName in set dom groups and isOwner(user) and user in set
            getUsers();

-- INVOICE

/**
  * Add an invoice of a type to the club
  */

public addInvoice: Client * set of Payment * Date * Hour * String1 * User ==> () addInvoice(client, payments, date, hour, type, user)==
(
  invoices := invoices union {new Invoice(payments, date, hour, type, false, client)};
)
pre type <> "" and isAtLeastEmployee(user) and card payments >= 1 and
  not exists p1, p2 in set payments & p1.getClient() <> p2.getClient() and user in set getUsers ();

/**
  * Add an invoice with all active payments of a client of a type to the club
  */

public addInvoiceWithAllActivePayments: Client * Date * Hour * String1 * User ==> () addInvoiceWithAllActivePayments(client, date, hour,
type, user)==
(
  invoices := invoices union {new Invoice( client.getPaymentsOfGivenType(type), date, hour, type
          , true, client)};
)
pre type <> "" and isAtLeastEmployee(user) and user in set getUsers();

/**
  * Add multiple payments to an invoice
  */

public addPaymentToInvoice: Invoice * set of Payment * User==> () addPaymentToInvoice(invoice,
payments, user) == invoice.addPayment(payments) pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Remove multiple payments from an invoice
  */

public removePaymentFromInvoice: Invoice * set of Payment * User==> () removePaymentFromInvoice(invoice,
payments, user) == invoice.removePayment(payments) pre isAtLeastEmployee(user) and user in set getUsers();

  -- CRM

    /**
  * Add a lead to the crm without attributing sales representative
  */

public addLeadToCRM: Lead * User==> () addLeadToCRM(lead,
user) == crm.addLead(lead)
pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Add a lead to the crm attributing sales representative
  */

public addLeadSRToCRM: Lead * SalesRepresentative * User ==> ()
addLeadSRToCRM(lead, sr, user) == crm.addLeadWithSR(lead, sr)
pre sr in set salesRepresentatives and isAtLeastEmployee(user) and user in set getUsers();

/**
  * Attribute a sales representative to a lead
  */

public setCRMLeadSR: Lead * SalesRepresentative * User ==> () setCRMLeadSR(lead, sr,
user) == crm.setLeadSR(lead, sr)
pre sr in set salesRepresentatives and isAtLeastEmployee(user) and user in set getUsers();

/**
  * Remove a lead from a sales representative
  */

public removeLeadSR: Lead * User ==> () removeLeadSR(lead,
user) ==
(
  crm.setLeadSR(lead, nil);
)
pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Remove a lead from crm
  */

public removeCRMLead: Lead * User==> () removeCRMLead(lead, user) ==
crm.removeLead(lead)
pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Transforms a lead into a client
  */

public transformLeadIntoClient: Lead * User==> () transformLeadIntoClient(lead, user)
==
(
  dcl client: Client := new Client(lead.getName(), lead.getAge(), lead.getGender(), lead. getNationality());
  crm.removeLead(lead); addClient(client,
  user);
)
pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Get report on club statistics
  */

public getReportOnClubStatistics: User ==> () getReportOnClubStatistics(user) ==
(
  dcl numClients: nat := card clients;
  dcl numTrainers: nat := card trainers;
  dcl numSalesRepresentatives: nat := card salesRepresentatives;
  dcl numClasses: nat := card classes;
  dcl numTrainingSessiosn: nat := card trainingSessions; IO‘println("********* CLUB
  STATISTICS *********"); IO‘print("Number of clients: ");
  IO‘println(numClients); IO‘print("Number of
  trainers: "); IO‘println(numTrainers);
  IO‘print("Number of sales representatives: ");
  IO‘println(numSalesRepresentatives); IO‘print("Number of gym
  classes: " ); IO‘println(numClasses);
  IO‘print("Number of training sessions: " );
  IO‘println(numTrainingSessiosn); IO‘println("");
  IO‘println("**********************************");
)
pre isAtLeastEmployee(user) and user in set getUsers();

/**
  * Get report on client activity
  */


public getClientActivity: Client * User ==> () getClientActivity(client, user) ==
client.getActivity()
pre isAtLeastEmployee(user) and client in set clients and user in set getUsers();

-- EMPLOYEES
```

```
/**
  * Get employee activity
  */

public getEmployeeActivity: Employee * bool * User ==> ()
getEmployeeActivity(employee, showAllTasks, user) == employee.getActivity(showAllTasks)
pre isAtLeastEmployee(user) and employee in set getEmployees() and user in set getUsers();

/**
  * Add a task to an employee
  */

public addTaskToEmployee: Employee * User * Task ==> () addTaskToEmployee(employee,
user, task) == employee.addTask(task)
pre employee in set getEmployees() and isOwner(user) and user in set getUsers();

-- PRODUCTS

/**
  * Create product for the club
  */

public addProduct: String1 * rat * nat * User ==> () addProduct(prod_name, prod_value,
qtt, user) ==
(
  dcl prod : Product := new Product(prod_name, prod_value, qtt); products := products
  union {prod};
)
pre prod_name <> "" and prod_value > 0 and qtt > 0 and isAtLeastEmployee(user) and user in set
        getUsers()
post not exists p1, p2 in set products & p1 <> p2 and
  p1.getName() =p2.getName();

/**
  * Add an existing product to the club
  */
public addProduct: Product * User ==> () addProduct(prod, user)
==
(
  products := products union {prod};
)
pre isAtLeastEmployee(user) and user in set getUsers()
post not exists p1, p2 in set products & p1 <> p2 and
  p1.getName() =p2.getName();

/**
  * Add stock of a product for the club
  */

public addStockOfProduct: Product * nat * User ==> () addStockOfProduct(prod, qtt,
user)== prod.addQuantity(qtt)
pre qtt > 0 and isAtLeastEmployee(user) and user in set getUsers();

/**
  * Remove a product
  */

public removeProduct: Product * User==> ()
removeProduct(prod, user) ==
(
  prod.removeQuantity(prod.getQuantity()); products := products \
  {prod};
)
```

```
pre products <> {} and prod in set products and isAtLeastEmployee(user) and user in set
        getUsers();

-- GETTERS

/**
 * Gets the club name
 *
 * @return name
 */

public pure getName : () ==> String1 getName() ==
return name
post RESULT = name;

/**
 * Gets the club newsletter
 *
 * @return newsletter
 */

public pure getNewsletter : () ==> String1 getNewsletter() ==
return newsletter
post RESULT = newsletter;

/**
 * Gets the club owner
 *
 * @return clubOwner
 */

public pure getOwner : () ==> Owner
getOwner() == return clubOwner post RESULT
= clubOwner;

/**
 * Gets the club clients
 *
 * @return set of Client
 */

public pure getClients : () ==> set of Client getClients() ==
return clients
post RESULT = clients;

/**
 * Gets the club trainers
 *
 * @return set of Trainer
 */

public pure getTrainers : () ==> set of Trainer getTrainers() ==
return trainers
post RESULT = trainers;

/**
 * Gets the club sales representatives
 *
 * @return set of SalesRepresentative
 */

public pure getSalesRepresentatives : () ==> set of SalesRepresentative getSalesRepresentatives() == return salesRepresentatives
post RESULT = salesRepresentatives;
```

```
/**
 * Gets the club employees (trainers + salesRepresentatives)
 *
 * @return set of User
 */

public pure getEmployees : () ==> set of Employee getEmployees() == return
trainers union salesRepresentatives post RESULT = trainers union
salesRepresentatives;

/**
 * Gets the club users (owner + clients + trainers + salesRepresentatives)
 *
 * @return set of User
 */

public pure getUsers : () ==> set of User
getUsers() == return {clubOwner} union clients union trainers union salesRepresentatives
post RESULT = {clubOwner} union clients union trainers union salesRepresentatives;

/**
 * Gets clients by name
 *
 * @return set of Client
 */

public pure getClientByName : String1 ==> set of Client getClientByName(clientName) ==
(
  dcl retClients: set of Client := {};
  for all c in set clients do if(c.getName() =
      clientName) then
        retClients := retClients union {c};
    return retClients;
)
pre clients <> {};

/**
 * Gets employees by name
 *
 * @return Employee
 */

public pure getEmployeeByName : String1 ==> set of Employee
getEmployeeByName(employeeName) ==
  (
  dcl retEmployees: set of Employee := {};
  for all e in set getEmployees() do if(e.getName() =
      employeeName) then
        retEmployees := retEmployees union {e};
    return retEmployees;
)
pre getEmployees() <> {};

/**
 * Gets users by name
 *
 * @return User
 */

public pure getUserByName : String1 ==> set of User getUserByName(userName) ==
(
  dcl retUsers: set of User := {};
  for all u in set getUsers() do
```

```
      if(u.getName() = userName) then
        retUsers := retUsers union {u};
    return retUsers;
)
pre getUsers() <> {};

/**
  *  Gets the club classes
  *
  *  @return set of GymClass
  */

public pure getGymClasses : () ==> set of GymClass getGymClasses() == return classes
post RESULT = classes;

/**
  *  Gets the club training sessions
  *
  *  @return set of TrainingSession
  */

public pure getTrainingSessions : () ==> set of TrainingSession getTrainingSessions() ==
return trainingSessions
post RESULT = trainingSessions;

/**
  *  Gets the club groups
  *
  *  @return map String1 to Group
  */

public pure getGroups : () ==> map String1 to Group getGroups() == return
groups
post RESULT = groups;

/**
  *  Gets a club group by name
  *
  *  @return Group
  */

public pure getGroupByName : String1 ==> Group
getGroupByName(groupName) == return groups(groupName) post RESULT =
groups(groupName);

/**
  *  Gets the club fee
  *
  *  @return fee
  */

public pure getFee : () ==> rat
getFee() == return fee
post RESULT = fee;

/**
  *  Gets the club invoices
  *
  *  @return set of Invoice
  */

public pure getInvoices : () ==> set of Invoice getInvoices() ==
return invoices
post RESULT = invoices;
```

```
/**
 *  Gets the club crm
 *
 *  @return CRM
 */

public pure getCRM : () ==> CRM getCRM() ==
return crm
post RESULT = crm;

/**
 *  Gets one of clubs employees calendar
 *
 *  @return map Date to seq of Task
 */

public pure getEmployeeCalendar: Employee ==> map Date to seq of Task getEmployeeCalendar(employee) ==
  return employee.getCalendar().getTasks()
pre employee in set getEmployees()
post RESULT = employee.getCalendar().getTasks();


/**
 *  Gets one of clubs employees tasks for a given day
 *
 *  @return seq of Task
 */

public getEmployeeTasksForGivenDate: Employee * Date ==> seq of Task getEmployeeTasksForGivenDate(employee, date) ==
  return employee.getCalendar().getTasksForGivenDate(date)
  pre employee in set getEmployees()
post RESULT = employee.getCalendar().getTasksForGivenDate(date);

/**
 *  Gets the products
 *
 *  @return products
 */

public getProducts: () ==> set of Product getProducts() ==
return products
post RESULT = products;

functions


public static isAtLeastEmployee(user: User) res:bool == user.getAccess() = <Owner> or
user.getAccess() = <Employee>;


public static isOwner(user: User) res:bool == user.getAccess() =
<Owner>;
end Club
```

3.4 Employee

```
class Employee is subclass of User
types

instance variables

  protected calendar: EmployeeCalendar;


operations
  /**
    * Employee constructor
    */

  public Employee: String1 * nat * Gender * String1 ==> Employee Employee(newName,
    newAge, newGender, newNationality) ==
    (
      calendar:= new EmployeeCalendar();
      User(newName, <Employee>, newAge, newGender, newNationality);
  );

    /**
    * Add task to employee
    */

    public addTask: Task ==> () addTask(task) ==
    calendar.addTask(task);

  -- GETTERS

  /**
    * Get employee activity
    */

    public getActivity: bool ==> ()
    getActivity(showAllTasks) == skip;

    /**
    *  Get employee calendar
    *
    *  @return calendar
    */

    public pure getCalendar: () ==> EmployeeCalendar getCalendar()
    == return calendar
```

```
post RESULT = calendar;



 end Employee
```

3.5 EmployeeCalendar

```
class EmployeeCalendar
types
  public String1 = seq of char; public Date
  = Utils‘Date; public Hour = Utils‘Hour;

values
instance variables
  private calendar: map Date to seq of Task := {|->};

  inv forall d in set dom calendar & not overlapsTasks(calendar(d));

operations

  /**
    * EmployeeCalendar constructor
    */

  public EmployeeCalendar: () ==> EmployeeCalendar EmployeeCalendar() == return self
    post calendar = {|->};

  /**
    * Add a task to the calendar
    */

    public addTask: Task ==> () addTask(task) ==
    (
      dcl date: Date := task.getDate();
      if date not in set dom calendar then
        calendar:= calendar munion {date |-> [task]}
      else
        calendar(date) := calendar(date) ^ [task];
    )
    post not overlapsTasks(calendar(task.getDate()));

  /**
    *  Gets the calendar tasks
    *
    *  @return map Date to seq of Task
    */

    public pure getTasks: () ==> map Date to seq of Task getTasks() == return
    calendar
    post RESULT = calendar;

  /**
    *  Gets the calendar tasks for a given date
    *
    *  @return seq of Task
    */

  public pure getTasksForGivenDate: Date ==> seq of Task
    getTasksForGivenDate(date) == return calendar(date) post RESULT =
    calendar(date);

functions

  /**
    *  Checks if there are any tasks that overlap
    */

  public static overlapsTasks(tasks: seq of Task) res:bool ==
    exists i, j in set inds tasks & i <> j and overlapsTask(tasks(i).getStartHour(), tasks(i). getEndHour(), tasks(j).getStartHour(),
            tasks(j).getEndHour());


  public overlapsTask: Hour * Hour * Hour * Hour-> bool
  overlapsTask(startHour1, endHour1, startHour2, endHour2) ==
    if ((startHour1 >= startHour2 and startHour1 < endHour2) or (endHour1 > startHour2
            and endHour1 <= endHour2) or (startHour1 <= startHour2 and endHour1 >=
            endHour2)) then true
      else false;

traces
    end EmployeeCalendar
```

3.6 Group

```
class Group
types
  public String1 = seq of char; instance
variables
  private clients: set of Client:= {};
```

```
    private groupInbox: map String1 to seq of String1 := {|->};
    private offers: seq of String1 := [];

  inv card clients >= 1;

operations

  /**
    * Group constructor
    */

    public Group: set of Client ==> Group
    Group(newClients) ==
  (
      clients := newClients;
      return self
    )
    post clients = newClients and groupInbox = {|->} and offers = [];

    /**
      * Add a new client to the group
      */

    public addClient: Client ==> ()
    addClient(client) == clients := clients union {client}
    pre client not in set clients
    post clients = clients~ union {client};

  /**
    * Remove a client from the group
    */

    public removeClient: Client ==> () removeClient(client) == clients := clients
    \ {client} pre card clients > 1 and client in set clients
    post clients = clients~ \ {client};

  /**
    * Add a new offer for the group
    */

    public sendOffer: String1 ==> () sendOffer(offer) == offers :=
    [offer] ^ offers pre offer <> ""
    post offers = [offer] ^ offers~;

  /**
    * Add a new message, from a given user, to the group inbox
    */

    public sendMessage: User * String1 ==> () sendMessage(user,
    msg) ==
    (
      if user.getName() not in set dom groupInbox then
        groupInbox := {user.getName() |-> [msg]} munion groupInbox
      else
        groupInbox(user.getName()) := groupInbox(user.getName()) ^ [msg];
    )
  pre msg <> "" and ( user in set clients or user.getAccess() = <Owner> );

    -- GETTERS

  /**
    *  Gets the inbox (the client must belong to the group)
    *
    * @return groupInbox
    */

    public pure checkInbox: Client ==> map String1 to seq of String1 checkInbox(client) ==
    return groupInbox
  pre client in set clients
  post RESULT = groupInbox;

    /**
      * Get group last message sent by a given user
      *
      * @return String1
      */

    public pure getLastMessageFromUser: String1 * Client ==> String1 getLastMessageFromUser(senderName,
    client) == return hd groupInbox(senderName) pre senderName in set dom groupInbox and client in set
    clients
    post RESULT = hd groupInbox(senderName);

    /**
      * Get group messages sent by a given user
      *
      * @return seq of String1
      */

    public pure getMessagesFromUser: String1 * Client ==> seq of String1
    getMessagesFromUser(senderName, client) == return groupInbox(senderName) pre senderName in
    set dom groupInbox and client in set clients
    post RESULT = groupInbox(senderName);

    /**
    * Gets the group offers
    *
    * @return offers
    */

    public pure checkOffers: Client ==> seq of String1 checkOffers(client) == return offers
  pre client in set clients
  post RESULT = offers;

  /**
    * Gets the group clients
    *
    * @return clients
    */

    public pure getClients: () ==> set of Client getClients() == return
    clients
  post RESULT = clients;

    end Group
```

3.7 GymClass

```
class GymClass is subclass of Session
types

values

instance variables private
  name: String1;
  private attendees: set of int := {};

operations
  /**
    * GymClass constructor
    */

  public GymClass: String1 * String1 * Trainer * DayOfWeek * Hour * Hour * Date==> GymClass GymClass(newDescription, className,
    newTrainer, newDayOfWeek, newStartHour, newEndHour,newDate
            ) ==
    (
    name := className;
    Session(newDescription, newTrainer, newDayOfWeek, newStartHour, newEndHour, newDate);
    )
  pre className <> ""
  post name = className;

  /**
    * Add an attendee to this class
    */

  public addAttendee: Client ==> ()
  addAttendee(client) ==
  (
    attendees:= attendees union {client.getID()}; client.addGymClass(self);
  )
  post attendees = attendees~ union {client.getID()};


    -- GETTERS

  /**
    *  Gets the gym class name
    *
    *  @return String1
    */

    public pure getName: () ==> String1 getName() ==
    return name
    post RESULT = name;

  /**
    *  Gets the gym class attendees
    *
    *  @return set of int
    */

  public pure getAttendees: () ==> set of int
    getAttendees() == return attendees
    post RESULT = attendees;


    end GymClass
```

3.8 GymFeePayment

```
class GymFeePayment is subclass of Payment
instance variables private fee:
        rat;
operations
         /**
   * GymFeePayment constructor
   */

  public GymFeePayment: Client * rat * Date * Hour==> GymFeePayment
   GymFeePayment(newClient, newFee, newDate, newHour) ==
   (
   fee:= newFee;
   newClient.addGymFeePayment(self);
              Payment(newClient, newDate, newHour, newFee);
        )
        pre newFee >= 0
        post fee = newFee ;

        -- GETTERS

  /**
   *  Gets the GymFeePayment fee
   *
   *  @return fee
   */

  public pure getFee : () ==> rat
  getFee() == return fee
  post RESULT = fee;

  end GymFeePayment
```

3.9 Invoice

,

```
class Invoice
types
  public String1 = seq of char; public Date =
  Utils‘Date;
        public Hour = Utils‘Hour;
values

instance variables

  private payments: set of Payment := {};
  private totalAmount: rat := 0;
  private date: Date; private hour:
  Hour; private type: String1;
  private client: Client;

  inv card payments >= 1;


operations

        /**
    * Invoice constructor
    */

  public Invoice: set of Payment * Date * Hour * String1 * bool * Client ==>Invoice Invoice(newPayments, newDate,
    newHour, newType, allActivePayments, newClient) == (

    payments:= newPayments; date
    := newDate;
    hour :=newHour; type
    :=newType;

    client := newClient;

    if allActivePayments = true then
    (
      cases type:
        "product" -> client.moveAllGymFeePaymentsToHistory(), "gymFee" ->
        client.moveAllProductPaymentsToHistory(),
        "personalTraining" -> client.moveAllPersonalTrainingPaymentsToHistory()
      end;

      for all p in set payments do
        totalAmount:= totalAmount + p.getAmount();

    )else (
      for all p in set payments do
      (
        totalAmount:= totalAmount + p.getAmount();
        cases type:
          "product" -> client.removeProductPayment(p), "gymFee" ->
          client.removeGymFeePayment(p),
          "personalTraining" -> client.removePersonalTrainingPayment(p)
        end;
      )
    );

      return self;
        )
pre card newPayments >= 1 and newType <> ""
post payments = newPayments and date = newDate and hour = newHour and
  type = newType and client = newClient;


        /**
  * Add payments to invoice
  */

public addPayment : set of Payment ==> ()
addPayment(newPayments) ==
(
  payments:= payments union newPayments;
  for all p in set newPayments do
  (
        totalAmount:= totalAmount + p.getAmount();
        cases type:
        "product" -> client.removeProductPayment(p), "gymFee" ->
        client.removeGymFeePayment(p),
        "personalTraining" -> client.removePersonalTrainingPayment(p)
      end;
  )

)
pre card newPayments >= 1 and newPayments inter payments = {};


      /**
  * Remove payments from invoice
  */

public removePayment: set of Payment ==> () removePayment(newPayments) ==
(
  payments := payments \ newPayments;
  for all p in set newPayments do
                    totalAmount:= totalAmount - p.getAmount();
)
pre payments <> {} and totalAmount > 0 and card newPayments >= 1;

-- GETTERS

/**
  *  Gets the invoice payments
  *
  *  @return set of Payment
  */

public pure getPayments: () ==> set of Payment getPayments() ==
  return payments
post RESULT = payments;

/**
  *  Gets the invoice total amount
  *
```

```
 *  @return totalAmount
 */

public pure getTotalAmount: () ==> rat
  getTotalAmount() == return totalAmount
post RESULT = totalAmount;

/**
 *  Gets the invoice date
 *
 *  @return date
 */

public pure getDate: () ==> Date getDate() ==
  return date
post RESULT = date;

/**
 *  Gets the invoice hour
 *
 *  @return hour
 */

public pure getHour: () ==> Hour getHour() ==
  return hour
post RESULT = hour;

/**
 *  Gets the invoice type
 *
 *  @return type
 */

public pure getType: () ==> String1 getType() ==
  return type
post RESULT = type;

/**
 *  Gets the invoice client
 *
 *  @return client
 */

public pure getClient: () ==> Client getClient() ==
  return client
post RESULT = client;

  end Invoice
```

3.10 Invoice

```
class Lead

 types

   public String1 = seq of char; public Gender =
   <Male> | <Female>;
 values

 instance variables private name:
   String1; private age: nat;
   private gender: Gender;
   private nationality: String1; public static
   curLeadID : int := 0; public id : int := curLeadID;

 operations
       /**
     * Lead constructor
     */

   public Lead: String1 * nat * Gender * String1 ==> Lead Lead(newName, newAge,
   newGender, newNationality) ==
   (
     name := newName; age :=
     newAge;
       gender := newGender; nationality :=
       newNationality;
     curLeadID := curLeadID +1;
     return self
   )
   pre newName <> "" and newNationality <> ""
   post name = newName and age =
       newAge and gender =
       newGender and
       nationality = newNationality and
       id = curLeadID~ and
       curLeadID = curLeadID~ + 1;


   -- GETTERS

   /**
     *  Gets the lead name
     *
     *  @return name
     */

   public pure getName: () ==> String1 getName() ==
     return name
   post RESULT = name;

   /**
     *  Gets the lead age
     *
```

```
  *  @return age
  */

 public pure getAge : () ==> nat
 getAge() == return age
 post RESULT = age;

 /**
  *  Gets the lead gender
  *
  *  @return gender
  */

 public pure getGender : () ==> Gender getGender() ==
 return gender
 post RESULT = gender;

 /**
  *  Gets the lead nationality
  *
  *  @return nationality
  */

 public pure getNationality : () ==>String1 getNationality()
 == return nationality post RESULT = nationality;

 /**
  *  Gets the lead id
  *
  *  @return id
  */

 public pure getID: () ==> int
   getID() == return id
 post RESULT = id;


functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
   end Lead
```

3.11 Owner

```
class Owner is subclass of User
types

values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
operations
  /**
   * Owner constructor
   */

  public Owner: String1 * nat * Gender * String1 ==> Owner
   Owner(newName, newAge, newGender, newNationality) == User(newName, <Owner>, newAge, newGender,
             newNationality);

  end Owner
```

3.12 Payment

```
class Payment
types
  public String1 = seq of char; public Date =
  Utils‘Date;
        public Hour = Utils‘Hour;
values
-- TODO Define values here
instance variables

  protected date: Date; protected hour:
  Hour; protected amount: rat := 0;
  public static curPaymentID : int := 0; protected id : int
  := curPaymentID; protected client: Client;
operations
  /**
   * Payment constructor
   */

  public Payment: Client * Date * Hour * rat ==> Payment Payment(newClient, newDate,
   newHour, newAmount) ==
   (
   client := newClient; date:=
   newDate; hour:= newHour;
   amount := newAmount;
   curPaymentID := curPaymentID +1;
     return self
  )
  pre newAmount > 0
        post client = newClient and date = newDate and hour = newHour and amount = newAmount and
              curPaymentID = curPaymentID~ + 1;
```

```
-- GETTERS

/**
  * Gets the payment date
  *
  * @return date
  */

public pure getDate: () ==> Date getDate() ==
  return date
post RESULT = date;

/**
  * Gets the payment hour
  *
  * @return hour
  */

public pure getHour: () ==> Hour getHour() ==
  return hour
post RESULT = hour;

/**
  * Gets the payment amount
  *
  * @return amount
  */

public    getAmount:    ()    ==>    rat
getAmount()  ==  return  amount  post
RESULT = amount;

/**
  * Gets the payment client
  *
  * @return client
  */

public pure getClient: () ==> Client getClient() ==
return client
post RESULT = client;

/**
  * Gets the payment id
  *
  * @return id
  */

public pure getID: () ==> int
getID() == return id
post RESULT = id;

functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
  end Payment
```

3.13 PerfectGYm

```
class PerfectGym
types
  public String1= seq1 of char; instance
variables

  private clubs: map String1 to Club := {|->};

  private static perfectGym: PerfectGym := new PerfectGym();
operations
  /**
    * PerfectGym constructor
    */

  public PerfectGym: () ==> PerfectGym PerfectGym() ==
  (return self)
  post clubs = {|->};

  /**
    * Add a new club to the PerfectGym
    */

  public addClub: Club ==> ()
  addClub(club) == clubs := clubs munion {club.getName() |-> club}
  pre club.getName() not in set dom clubs
  post clubs = clubs~ munion {club.getName() |-> club};

  -- GETTERS

  /**
    *  Gets the perfectGym instance (Singleton)
    *
    *  @return perfectGym
    */

  public pure static getInstance: () ==> PerfectGym getInstance() ==
  return perfectGym
  post RESULT = perfectGym;

  /**
    *  Gets the perfectGym clubs
    *
    *  @return clubs
    */

  public pure getClubs: () ==> map String1 to Club getClubs() ==
  return clubs
  post RESULT = clubs;
```

```
end PerfectGym
```

3.14 PersonalTrainingPayment

```
class PersonalTrainingPayment is subclass of Payment
instance variables private fee:
          rat;

operations
         /**
    * PersonalTrainingPayment constructor
    */

  public PersonalTrainingPayment: Client * rat * Date * Hour==> PersonalTrainingPayment PersonalTrainingPayment(newClient, newFee,
    newDate, newHour) ==
    (
    fee:= newFee; newClient.addPersonalTrainingPayment(self);
                Payment(newClient, newDate, newHour, newFee);
        )
        pre newFee >= 0
        post fee = newFee ;

        -- GETTERS

        /**
    *  Gets the PersonalTrainingPayment fee
    *
    *  @return fee
    */

  public pure getFee: () ==> rat
  getFee() == return fee
  post RESULT = fee;

functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
    end PersonalTrainingPayment
```

3.15 Product

```
class Product
types
  public String1= seq1 of char; values
-- TODO Define values here
instance variables private name
  : String1; private value : rat;
  private quantity: nat;

  inv quantity >= 0;
operations
  /**
    * Product constructor
    */

  public Product: String1 * rat * nat ==> Product Product(prod_name,
    prod_value, prod_quantity) == (
      name := prod_name; value :=
    prod_value;
    quantity := prod_quantity;
    return self;
    )
  pre prod_value >= 0 and prod_quantity >0 and prod_name <> ""
  post name = prod_name and value = prod_value and quantity = prod_quantity;

  /**
    * Add quantiy of this product
    */

  public addQuantity: nat ==> () addQuantity(qtt) == quantity:=
  quantity + qtt pre qtt > 0;

  /**
    * Remove quantity of this product
    */

  public removeQuantity: nat ==> () removeQuantity(qtt) ==
  quantity:= quantity - qtt pre quantity > 0 and qtt <= quantity;

  -- GETTERS

  /**
    *  Gets the product name
    *
    *  @return name
    */

  public pure getName: () ==> String1 getName() ==
  return name
  post RESULT = name;

  /**
    *  Gets the product value
    *
    *  @return value
    */

  public pure getValue: () ==> rat
  getValue() == return value
  post RESULT = value;

  /**
    *  Gets the product quantity
    *
    *  @return quantity
    */

  public pure getQuantity: () ==> rat getQuantity()
  == return quantity post RESULT = quantity;

    end Product
```

3.16 ProductPayment

```
class ProductPayment is subclass of Payment
types

values
-- TODO Define values here
instance variables

  private products: seq of Product := [];

              inv len products >= 1;
operations
  /**
   * ProductPayment constructor
   */

  public ProductPayment: Client * Product * nat * Date * Hour==> ProductPayment ProductPayment(newClient, newProduct, qtt , newDate,
   newHour) ==
   (
              dcl moneySpent: rat := newProduct.getValue() * qtt; products:=
              [newProduct]; newProduct.removeQuantity(qtt);

              newClient.addProductPayment(self); newClient.addProductBought(newProduct,
              qtt, moneySpent);
```

```
                Payment(newClient, newDate, newHour, moneySpent);
 )
 pre qtt >= 1
     post products = products ;

       /**
   * Add product to this payment
   */

       public addProduct: Product * nat==> () addProduct(product,
       qtt) ==
       (
               dcl moneySpent: rat := product.getValue() * qtt; products := products ^
               [product];
               amount:= amount + moneySpent; product.removeQuantity(qtt);
               client.addProductBought(product, qtt, moneySpent);
       );

       -- GETTERS

       /**
   *  Gets the productPayment products
   *
   *  @return seq of Product
   */

 public pure getProducts: () ==> seq of Product getProducts() ==
   return products
   post RESULT = products;


  end ProductPayment
```

3.17 SalesRepresentative

```
class SalesRepresentative is subclass of Employee
types

values

instance variables
  private leads: set of String1 := {};
operations
  /**
   * SalesRepresentative constructor
   */

  public SalesRepresentative: String1 * nat * Gender * String1 ==> SalesRepresentative
   SalesRepresentative(newName, newAge, newGender, newNationality) == Employee(newName, newAge, newGender, newNationality)
   post leads = {};

  /**
   * Add a lead to this sales representative
   */

   public addLead: Lead ==> ()
   addLead(lead) == leads := leads union {lead.getName()}
   pre lead.getName() not in set leads
   post leads= leads~ union {lead.getName()};

  /**
   * Remove a lead from this sales representative
   */

   public removeLead: Lead ==> ()
   removeLead(lead) == leads := leads \ {lead.getName()}
   pre lead.getName() in set leads
   post leads= leads~ \ {lead.getName()};

  -- GETTERS

/**
   * Get sales representative activity
   */


   public getActivity: bool ==> ()
   getActivity(showAllTasks) ==
   (
    dcl numLeads: nat := card leads;
    dcl tasks: map Date to seq of Task := calendar.getTasks();
    dcl i: nat := 0;
    dcl t: Task;

    IO‘println("********* SALES REPRESENTATIVE STATISTICS *********"); IO‘print("Number of leads: ");
    IO‘println(numLeads);

    if showAllTasks then
      (
       for all d in set dom tasks do
       (
        i := 0;
        IO‘print("Date: " ); IO‘println(d);

        while i < len tasks(d) do
        (
          t := tasks(d)(i);
          IO‘print(" Task: ");
          IO‘print(t.getDescription()); IO‘print("
          started at "); IO‘print(t.getStartHour());
          IO‘print(" and ended at ");
          IO‘println(t.getEndHour()); i:= i + 1;
        );
        IO‘println("");
       )
      );

    IO‘println("");
```

```
    IO`println("**************************************************");
  );

      /**
  *  Gets the SalesRepresentative leads
  *
  *  @return set of String1
  */

public pure getLeads: () ==> set of String1 getLeads() == return
  leads
  post RESULT = leads;

 end SalesRepresentative
```

3.18 Session

```
class Session is subclass of Task
types

  public DayOfWeek = Utils`DayOfWeek;
values

instance variables

  protected trainer: int;

  protected dayOfWeek: DayOfWeek;

operations
  /**
   * Session constructor
   */

  public Session: String1 * Trainer * DayOfWeek * Hour * Hour * Date ==> Session Session(newDescription, newTrainer,
   newDayOfWeek, newStartHour, newEndHour, newDate) == (
   trainer := newTrainer.getID(); dayOfWeek :=
   newDayOfWeek;
   Task(newDescription, newStartHour, newEndHour, newDate);
   )
  post trainer = newTrainer.getID() and dayOfWeek = newDayOfWeek and
     startHour = newStartHour and endHour = newEndHour and date = newDate;

   -- GETTERS

  /**
   *  Gets the session trainer
   *
   *  @return trainer
   */

   public pure getTrainer: () ==> int
   getTrainer() == return trainer
  post RESULT = trainer;

  /**
   *  Gets the session dayOfWeek
   *
   *  @return dayOfWeek
   */

   public pure getDayOfWeek: () ==> DayOfWeek getDayOfWeek()
   == return dayOfWeek
   post RESULT = dayOfWeek;
   end Session
```

3.19 Task

```
class Task
types

 public String1 = seq of char; public Date
 = Utils‘Date; public Hour = Utils‘Hour;
values
-- TODO Define values here
instance variables

 protected description: String1; protected
 startHour: Hour; -- HHMM protected endHour:
 Hour; -- HHMM protected date: Date; --
 YYYYMMDD

 inv endHour > startHour;

operations
 /**
  * Task constructor
  */

 public Task: String1 * Hour * Hour * Date ==> Task Task(newDescription,
 newStartHour, newEndHour, newDate) == (
  description:= newDescription;
  atomic(
  startHour := newStartHour;
```

```
   endHour := newEndHour;
   );
   date := newDate;
   return self;
  )
  pre newEndHour > newStartHour and newDescription <> ""
  post description = newDescription and startHour = newStartHour and endHour = newEndHour and date
           = newDate;

  -- GETTERS

  /**
   *  Gets the task description
   *
   *  @return description
   */

  public pure getDescription: () ==>String1
  getDescription() == return description post RESULT =
  description;

  /**
   *  Gets the task startHour
   *
   *  @return startHour
   */

  public pure getStartHour: () ==> Hour
  getStartHour() == return startHour post RESULT =
  startHour;

  /**
   *  Gets the task endHour
   *
   *  @return endHour
   */

  public pure getEndHour: () ==> Hour
  getEndHour() == return endHour post RESULT
  = endHour;

  /**
   *  Gets the task date
   *
   *  @return date
   */

  public pure getDate: () ==> Date getDate() ==
   return date
   post RESULT = date;


functions



traces
-- TODO Define Combinatorial Test Traces here
   end Task
```

3.20 Trainer

```
class Trainer is subclass of Employee
types

values
-- TODO Define values here
instance variables
  private trainees: set of int := {};
  private classes: set of GymClass := {};
  private trainingSessions: set of TrainingSession := {};

  inv not exists c1, c2 in set classes & c1 <> c2 and
    c1.getTrainer() = c2.getTrainer() and
    c1.getDate() = c2.getDate() and
    Utils‘overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());

  inv not exists t1, t2 in set trainingSessions &
    t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
    t1.getDate() = t2.getDate() and Utils‘overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());

operations
  /**
    * Trainer constructor
    */

  public Trainer: String1 * nat * Gender * String1 ==> Trainer
    Trainer(newName, newAge, newGender, newNationality) == Employee(newName, newAge, newGender,
             newNationality)
    post trainees = {} and classes = {} and trainingSessions = {};

  /**
    * Add trainee to this trainer
    */

  public addTrainee: Client ==> ()
  addTrainee(client) == trainees := trainees union {client.getID()}
  pre client.getID() not in set trainees
  post trainees = trainees~ union {client.getID()};

  /**
    * Remove trainee from this trainer
    */

  public removeTrainee: Client ==> ()
  removeTrainee(client) == trainees := trainees \ {client.getID()}
  pre trainees <> {} and client.getID() in set trainees
  post trainees = trainees~ \ {client.getID()};

    -- GYM CLASSES
```

```
/**
  * Add gym class
  */

public addGymClass: GymClass ==> ()
addGymClass(gymClass) == classes := classes union {gymClass}
pre gymClass not in set classes
post classes = classes~ union {gymClass} and not exists c1, c2
  in set classes &
  c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1.getDate() = c2.getDate() and
  Utils`overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());

-- TRAINING SESSIONS

/**
  * Add training session
  */

public addTrainingSession: TrainingSession ==> () addTrainingSession(trainingSession) == trainingSessions :=
trainingSessions union {
        trainingSession}
pre trainingSession not in set trainingSessions
post trainingSessions = trainingSessions~ union {trainingSession} and not exists t1, t2 in set
  trainingSessions &
  t1 <> t2 and (t1.getTrainer() = t2.getTrainer() or t1.getTrainee() = t2.getTrainee()) and
  t1.getDate() = t2.getDate() and Utils`overlaps(t1.getStartHour(), t1.getEndHour(), t2. getStartHour(), t2.getEndHour());

-- GETTERS

/**
  * Get trainer activity
  */


  public getActivity: bool ==> ()
  getActivity(showAllTasks) ==
  (
    dcl numClasses: nat := card classes;
    dcl numTrainingSessiosn: nat := card trainingSessions;
    dcl numTrainees: nat := card trainees;
    dcl tasks: map Date to seq of Task := calendar.getTasks();
    dcl i: nat := 0;
    dcl t: Task;

    IO`println("********* TRAINER STATISTICS *********"); IO`print("Number of
    gym classes: "); IO`println(numClasses);
    IO`print("Number of training sessions: ");
    IO`println(numTrainingSessiosn); IO`print("Number of trainees:
    " ); IO`println(numTrainees);

    if showAllTasks then
      (
        for all d in set dom tasks do
        (
          i := 1;
          IO`print("Date: " ); IO`println(d);


          while i <= len tasks(d) do
          (
              t := tasks(d)(i);
              IO‘print(" Task: ");
              IO‘print(t.getDescription()); IO‘print("
              started at "); IO‘print(t.getStartHour());
              IO‘print(" and ended at ");
              IO‘println(t.getEndHour()); i:= i + 1;
            );
            IO‘println("");
          )
        );

      IO‘println(""); IO‘println("*************************************");
    );

  /**
    * Gets the trainer trainees
    *
    * @return set of int
    */

  public pure getTrainees : () ==> set of int
  getTrainees() == return trainees
  post RESULT = trainees;

  /**
    * Gets the trainer classes
    *
    * @return set of GymClass
    */

  public pure getClasses: () ==> set of GymClass getClasses() ==
  return classes
  post RESULT = classes;

  /**
    * Gets the trainer training sessions
    *
    * @return set of TrainingSession
    */

  public pure getTrainingSessions: () ==> set of TrainingSession getTrainingSessions() ==
  return trainingSessions
  post RESULT = trainingSessions;


functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
    end Trainer
```

3.21 TrainingSession

```
class TrainingSession is subclass of Session
types

values

instance variables

        private trainee: int;

operations
  /**
    * TrainingSession constructor
    */

      public TrainingSession: String1 * Client * DayOfWeek * Hour * Hour * Date ==> TrainingSession TrainingSession(newDescription, client,
    newDayOfWeek, newStartHour, newEndHour, newDate) ==
    (
    dcl newTrainer: Trainer := client.getTrainer(); trainee :=
    client.getID();
      Session(newDescription, newTrainer, newDayOfWeek, newStartHour, newEndHour, newDate);
    )
    pre client.getTrainer() <> nil post trainee =
  client.getID();

  -- GETTERS
    /**
      *  Get trainee
      *
      *  @return trainee
      */

    public pure getTrainee: () ==> int getTrainee()
    == return trainee post RESULT = trainee;

   end TrainingSession
```

3.22 User

```
class User
types

  public String1 = seq of char;
  public Access = <Owner> | <Employee> | <Client>;
  public Date = Utils‘Date;
  public Hour = Utils‘Hour;
  public Gender = <Male> | <Female>;

instance variables protected
  name: String1;
  public static curUserID : int := 0;
  protected id : int := curUserID;
  protected inbox: map String1 to seq of String1 := {|->};
  protected club: [Club]; protected
  access: Access; protected age: nat;
  protected gender: Gender;
  protected nationality: String1;

operations
  /**
    * User constructor
    */

  public User: String1 * Access * nat * Gender * String1 ==> User User(newName, acc,
    newAge, newGender, newNationality) == (
      club := nil; name :=
      newName; age :=
      newAge;
      gender := newGender; nationality :=
      newNationality; curUserID := curUserID
      +1; access := acc;
      return self
    )
  pre newAge >= 12 and newName <> "" and newNationality <> ""
    post name = newName and age
      = newAge and gender =
      newGender and
      nationality = newNationality and
        id = curUserID~ and access
        = acc and club = nil and
      curUserID = curUserID~ + 1;

  /**
    * Sets this user club
    */

    public setClub: Club ==> () setClub(newClub) ==
    club:= newClub post club = newClub;

    -- MESSAGES

  /**
    * Sends a message from this user to the given user
    */

  public sendMessage: User * String1 ==> ()
```

```
sendMessage(receiver, msg) == receiver.receiveMessage(msg, self) pre receiver in set
club.getUsers() and msg <> "";

  /**
    * Puts the new message on the top of the inbox
    */

  public receiveMessage: String1 * User ==> ()
receiveMessage(msg, user) ==
(

    if user.getName() not in set dom inbox then
      inbox := {user.getName() |-> [msg]} munion inbox
    else
      inbox(user.getName()) := [msg] ^ inbox(user.getName()) ;
  )
  pre user in set club.getUsers() and msg <> "";

/**
  * Delete last message from a given user
  */

public deleteLastMessageFromUser: String1 ==> () deleteLastMessageFromUser(user) ==
inbox(user):= tl inbox(user) pre user in set dom inbox and inbox(user) <> [] ;

/**
  * Delete an n message from a given user
  */

public deleteMessageNFromUser: nat * String1 ==> ()
deleteMessageNFromUser(n, user) == inbox(user) := inbox(user)(1,..., n - 1) ^ inbox(user)((n +
        1) ,..., (len inbox(user)))
pre inbox(user) <> [] and n in set inds inbox(user)
post inbox(user) = inbox(user)(1,..., n - 1) ^ inbox(user)((n + 1),..., (len inbox(user)));  -- CHECK THIS POST

/**
  * Set this user access
  */

public setAccess: Access ==> () setAccess(newAccess) == access :=
newAccess;

-- GETTERS

/**
  *  Gets the user inbox
  *
  *  @return map String1 to seq of String1
  */

  public pure checkInbox: () ==> map String1 to seq of String1 checkInbox() == return
inbox
post RESULT = inbox;

  /**
  *  Gets the last messages from a given user
  *
  *  @return seq of String1
  */

public pure readMessagesFromUser: User ==> seq of String1
readMessagesFromUser(user) == return inbox(user.getName())
pre user.getName() in set dom inbox and user in set club.getUsers()
```

```
post RESULT = inbox(user.getName());

/**
 * Gets the last message from a given user
 *
 * @return String1
 */

public pure readLastMessageFromUser: User ==> String1 readLastMessageFromUser(user)
== return hd inbox(user.getName()) pre user.getName() in set dom inbox and user in set
club.getUsers() post RESULT = hd inbox(user.getName());

/**
 * Get an n message from a given user
 *
 * @return Strign1
 */

public readMessageNFromUser: nat * User ==> String1 readMessageNFromUser(n, user)
== return inbox(user.getName())(n)
pre inbox(user.getName()) <> [] and n in set inds inbox(user.getName()) and user in set club. getUsers()
post RESULT = inbox(user.getName())(n);

/**
 * Gets the user activity
 */

public getActivity: () ==> () getActivity() ==
skip;

/**
 * Gets the user name
 *
 * @return String1
 */

public pure getName : () ==> String1 getName() ==
return name
post RESULT = name;

/**
 * Gets the user id
 *
 * @return int
 */

public pure getID : () ==> int
getID() == return id
post RESULT = id;

  /**
 * Gets the user acces to the club
 *
 * @return Access
 */

public pure getAccess : () ==> Access getAccess() ==
return access
post RESULT = access;

/**
 * Gets the user age
 *
 *  @return age
 */

public pure getAge : () ==> nat
getAge() == return age
post RESULT = age;

/**
 *  Gets the user nationality
 *
 *  @return gender
 */

public pure getGender : () ==> Gender getGender() ==
return gender
post RESULT = gender;

/**
 *  Gets the user nationality
 *
 *  @return nationality
 */

public pure getNationality : () ==>String1 getNationality()
== return nationality post RESULT = nationality;

/**
 *  Gets the user club
 *
 *  @return club
 */

public pure getClub: () ==> Club getClub() ==
 return club
 post RESULT = club;

 end User
```

3.23 Utils

```
class Utils
types
 public DayOfWeek = <Monday> | <Tuesday> | <Wednesday> | <Thursday> | <Friday> | <Saturday> | < Sunday>;
 public Date = nat
            inv d == IsValidDate(d div 10000, (d div 100) mod 100, d mod 100);

   public Hour = nat
            inv h == IsValidHour(h div 100, h mod 100);


functions

 /**
 *  Checks if a date is valid
 *
 *  @return bool
 */

 public static IsValidDate: nat * nat * nat -> bool
   IsValidDate(y, m, d) ==
     y >= 1 and m >= 1 and m <= 12 and d >= 1 and d <= DaysOfMonth(y, m);

 /**
 *  Checks if an hour is valid
 *
 *  @return bool
 */

 public static IsValidHour: nat * nat -> bool
   IsValidHour(h,m) ==
     h >= 1 and h <= 24 and m >= 0 and m <= 60;

 /**
 *  Checks if an year is leap
 *
 *  @return bool
 */

 public static IsLeapYear: nat -> bool
   IsLeapYear(year) ==
     year mod 4 = 0 and year mod 100 <> 0 or year mod 400 = 0;

 /**
 *  Gets the number of days in a given month
 *
 *  @return nat
 */

       public static DaysOfMonth: nat * nat -> nat
       DaysOfMonth(y, m) == (
         cases m :
               1, 3, 5, 7, 8, 10, 12 -> 31,
               4, 6, 9, 11 -> 30,
               2 -> if IsLeapYear(y) then 29 else 28
             end
         )
       pre m >= 1 and m <= 12;

 /**
 *  Creates a new instance of a Date
 *
 *  @return Date
 */

 public static CreateDate: nat * nat * nat -> Date CreateDate(y, m, d)
 ==
   y * 10000 + m * 100 + d
 pre IsValidDate(y, m, d);

 /**
 *  Creates a new instance of an Hour
 *
 *  @return Hour
 */

 public static CreateHour: nat * nat -> Hour CreateHour(h, m) ==
        h * 100 + m
 pre IsValidHour(h, m);

 /**
 *  Gets the year of a given date
 *
 *  @return nat
 */

 public static Year: Date -> nat
       Year(d) ==
     d div 10000;

 /**
 *  Gets the month of a given date
 *
 *  @return nat
 */

 public static Month: Date -> nat
     Month(d) ==
       (d div 100) mod 100;

 /**
 *  Gets the day of a given date
 *
 *  @return nat
 */

 public static Day: Date -> nat
   Day(d) ==
       d mod 100;

 /**
 *  Gets the hours of a given hour
 *
 *  @return nat
 */

   public static Hours: Hour -> nat
```

```
      Hours(h) == h
        div 100;

  /**
  *  Gets the minutes of a given hour
  *
  *  @return nat
  */

    public static Minutes: Hour -> nat
      Minutes(h) == h
        mod 100;

  /**
  *  Verifies if two hours overlap
  *
  *  @return bool
  */

    public static overlaps: Hour * Hour * Hour * Hour-> bool overlaps(startHour1,
endHour1, startHour2, endHour2) == ((startHour1 >= startHour2 and startHour1 <
                                                                  endHour2) or
                                    (endHour1 > startHour2 and endHour1 <= endHour2) or
                                      (startHour1 <= startHour2 and endHour1 >= endHour2));


    end Utils
```

4. Model validation

4.1 ClientTest

```
class ClientTest is subclass of MyTestCase

 instance variables

   owner1: Owner := new Owner("Rui", 21, <Male>, "portuguese");

   club1: Club := new Club("Bombados", owner1, 17);
   client1: Client := new Client("Vasco", 25, <Male>,"brazilian");

   trainer1: Trainer := new Trainer("Alex", 33, <Male>,"english");

   product1: Product := new Product("Prota", 29.99, 40);

   gymClass1: GymClass := new GymClass("Aula de baixa intensidade", "Pilates", trainer1,<Tuesday>, Utils'CreateHour(09, 00),
     Utils'CreateHour(10, 00),
     Utils'CreateDate(2019, 01, 11));

   gymFeePayment1: GymFeePayment := new GymFeePayment(client1, 80,
     Utils'CreateDate(2019, 01, 22),
         Utils'CreateHour(08, 00));

   productPayment1: ProductPayment := new ProductPayment(client1, product1, 2, Utils'CreateDate(2019, 01, 30),
         Utils'CreateHour(18, 29));

   personalTrainingPayment1: PersonalTrainingPayment := new PersonalTrainingPayment(client1, 30, Utils'CreateDate(2019, 01, 23),
         Utils'CreateHour(08, 00));

 operations

   public Run: () ==> () Run() == (
     IO'println("\nClient Tests");

     club1.addTrainer(trainer1, owner1);
     club1.addClient(client1, owner1);

     client1.addTrainer(trainer1, 30);
     client1.addGymClass(gymClass1);
     client1.addGymAttendence(20191111);

     assertEqual(trainer1, client1.getTrainer()); assertEqual(30,
     client1.getPersonalTrainingFee()); assertEqual({gymClass1},
     client1.getClasses());
     assertEqual({20190111, 20191111}, client1.getGymAttendences());

     IO'println("\nInitiate GetPayments"); assertEqual({gymFeePayment1},
     client1.getGymFeePayments()); assertEqual({productPayment1},
     client1.getProductPayments());
     assertEqual({personalTrainingPayment1}, client1.getPersonalTrainingPayments());

     IO'println("\nInitiate RemovePayments");
     client1.removeGymFeePayment(gymFeePayment1);
     client1.removeProductPayment(productPayment1);
     client1.removePersonalTrainingPayment(personalTrainingPayment1);

     IO'println("\nInitiate GetHistoryPayments"); assertEqual({gymFeePayment1},
     client1.getHistoryGymFeePayments()); assertEqual({productPayment1},
     client1.getHistoryProductPayments());
     assertEqual({personalTrainingPayment1}, client1.getHistoryPersonalTrainingPayments());

     IO'println("\nInitiate Products"); client1.addProductBought(product1, 1, 29.99);
     assertEqual(89.97, client1.getTotalSPentOnProducts()); assertEqual({"Prota" |-> 3},
     client1.getProductsBought());

     IO'println("Finalizing Client Tests");

   );
```

end ClientTest

Function or operation	Line	Coverage	Calls
Client	34	100.0%	9
addGymAttendence	118	100.0%	6
addGymClass	88	100.0%	4
addGymFeePayment	128	100.0%	8
addHistoryGymFeePayment	152	100.0%	6
addHistoryPersonalTrainingPayment	168	100.0%	3
addHistoryProductPayment	160	100.0%	1
addPersonalTrainingPayment	144	100.0%	5
addProductBought	277	100.0%	3
addProductPayment	136	100.0%	2
addProductToPayment	292	0.0%	0
addTrainer	49	90.9%	0
addTrainingSession	103	100.0%	1
createProductPayment	266	0.0%	0
getActivity	301	93.1%	1
getClasses	361	100.0%	1
getGymAttendences	379	100.0%	1
getGymFeePayments	397	100.0%	1
getHistoryGymFeePayments	425	100.0%	1
getHistoryPersonalTrainingPayments	443	100.0%	1
getHistoryProductPayments	434	100.0%	1
getPaymentsOfGivenType	502	65.0%	0
getPersonalTrainingFee	343	100.0%	1
getPersonalTrainingPayments	415	100.0%	1
getProductPayments	406	100.0%	1

getProductsBought	388	100.0%	1
getTotalSPentOnProducts	352	100.0%	1
getTrainer	334	100.0%	5
getTrainingSessions	370	0.0%	0
moveAllGymFeePaymentsToHistory	212	0.0%	0
moveAllPersonalTrainingPaymentsToHistory	236	0.0%	0
moveAllProductPaymentsToHistory	223	66.6%	0
payGymFee	249	0.0%	0
payPersonalTrainingFee	259	0.0%	0
readGroupLastMessageFromUser	488	0.0%	0
readGroupMessages	452	0.0%	0
readGroupMessagesFromUser	474	0.0%	0
readGroupOffers	463	0.0%	0
removeGymFeePayment	176	100.0%	6
removePersonalTrainingPayment	200	100.0%	3
removeProductPayment	188	100.0%	1
removeTrainer	64	100.0%	1
sendMessageToGroup	79	0.0%	0
Client.vdmpp		65.7%	76

4.2 ClubTest

```
class ClubTest is subclass of MyTestCase

 instance variables

      owner1: Owner := new Owner("Rui", 21, <Male>, "portuguese");

   club1: Club := new Club("Bombados", owner1, 17);

   client1: Client := new Client("Maria", 23, <Female>, "portuguese"); client2: Client := new
   Client("Jorge", 25, <Male>, "spanish");

   trainer1: Trainer := new Trainer("Vasco", 25, <Male>, "brazilian"); trainer2: Trainer := new
   Trainer("Alex", 33, <Male>, "english");

   salesRepresentative1: SalesRepresentative := new SalesRepresentative("Joana", 21, <Female>, " portuguese");
   salesRepresentative2: SalesRepresentative := new SalesRepresentative("Manuel", 33, <Male>, " french");

   gymClass1: GymClass := new GymClass("Aula de baixa intensidade", "Pilates", trainer1,<Tuesday>, Utils‘CreateHour(09, 00),
     Utils‘CreateHour(10, 00),
     Utils‘CreateDate(2019, 01, 11));

   gymFeePayment1: GymFeePayment := new GymFeePayment(client1, 80,
     Utils‘CreateDate(2019, 01, 22),
         Utils‘CreateHour(08, 00));

   gymFeePayment2: GymFeePayment := new GymFeePayment(client1, 80,
     Utils‘CreateDate(2019, 04, 22),
         Utils‘CreateHour(08, 00));


   gymFeePayment3: GymFeePayment := new GymFeePayment(client1, 80,
     Utils‘CreateDate(2019, 03, 22),
         Utils‘CreateHour(18, 00));

   invoice1: Invoice := new Invoice({gymFeePayment3}, Utils‘CreateDate(2019, 04, 23),
         Utils‘CreateHour(08, 00), "gymFee", false,
     client1);

   personalTrainingPayment1: PersonalTrainingPayment := new PersonalTrainingPayment(client1, 50, Utils‘CreateDate(2019, 03, 13),
         Utils‘CreateHour(08, 00));

   lead1: Lead := new Lead("Maria", 23, <Female>, "portuguese"); lead2: Lead := new
   Lead("Jorge", 25, <Male>, "spanish");
```

```
  task1: Task := new Task("R e u n i o  com o  patr o",
      Utils‘CreateHour(09, 00),
      Utils‘CreateHour(10, 00),
      Utils‘CreateDate(2019, 01, 12));

  product1: Product := new Product("Prota", 29.99, 40);


operations

  public Run: () ==> () Run() == (
    IO‘println("\nClub Tests");

    IO‘println("Initiate                              addClient");
    club1.addClient(client1,                         owner1);
    club1.addClient(client2, owner1);

    IO‘println("Initiate addTrainer");
    club1.addTrainer(trainer1, owner1);
    club1.addTrainer(trainer2, owner1);

    IO‘println("Initiate addSalesRepresentative");
    club1.addSalesRepresentative(salesRepresentative1, owner1);
    club1.addSalesRepresentative(salesRepresentative2, owner1);


    trainer1.addGymClass(gymClass1);
    trainer1.addTask(gymClass1);
    assertEqual({20190111 |-> [gymClass1]}, club1.getEmployeeCalendar(trainer1)); assertEqual([gymClass1],
    club1.getEmployeeTasksForGivenDate(trainer1, 20190111));

    IO‘println("Initiate addGroup"); club1.addGroup("PuxadoresDeFerro", {client1}, owner1);

    IO‘println("Initiate addPersonalTraining"); club1.addPersonalTraining(trainer1, client1,
    40, owner1);

    IO‘println("Initiate addGymClass"); club1.addGymClass(gymClass1,
    owner1); assertEqual({gymClass1}, club1.getGymClasses());

    IO‘println("Initiate GymClass");
    club1.addGymClass("Aula de alta intensidade", "Zumba", trainer1, <Monday>, Utils‘CreateHour(16, 00),
        Utils‘CreateHour(17, 00),
        Utils‘CreateDate(2019, 01, 14), owner1); IO‘println("Finalizing
        GymClass1");
    club1.addGymClass("Aula de baixa intensidade", "Pilates", trainer2, <Tuesday>, Utils‘CreateHour(09, 00),
        Utils‘CreateHour(10, 00),
        Utils‘CreateDate(2019, 01, 15), owner1); IO‘println("Finalizing
        GymClass2");

    club1.addAttendeeToGymClass(gymClass1, client1, owner1);
    club1.addAttendeeToGymClass(gymClass1, client2, owner1);

    club1.addTrainingSession("Aula iniciante", client1, <Monday>, Utils‘CreateHour(14, 00),
        Utils‘CreateHour(15, 00),
        Utils‘CreateDate(2019, 01, 14), owner1); IO‘println("Finalizing
        TrainingSession1");

    assertEqual(club1.getTrainingSessions(), trainer1.getTrainingSessions());
```

```
club1.removeTraineeFromTrainer(trainer1, client1, owner1);

club1.setUserAccess(owner1, trainer1, <Owner>);

IO`println("Initiate addNewsletter"); club1.addNewsletter("Sales", owner1);

IO`println("Initiate sendMessageClient"); club1.sendMessageClient("Bom Ano", client1,
owner1);

IO`println("Initiate sendMessageEmployee"); club1.sendMessageEmployee("Tas despedido
bro!", trainer1, owner1);

IO`println("Initiate sendMessageAllClients"); club1.sendMessageAllClients("Feliz Natal",
owner1); IO`println("Initiate sendMessageAllTrainers");
club1.sendMessageAllTrainers("Feliz Natal", owner1); IO`println("Initiate
sendMessageAllSalesRepresentatives"); club1.sendMessageAllSalesRepresentatives("Feliz
Natal", owner1);

IO`println("Initiate sendMessageToGroup"); club1.sendMessageToGroup("Feliz Natal",
"PuxadoresDeFerro", owner1);
club1.sendOfferToGroup("Prota com 10% de desconto esta semana!! :O", "PuxadoresDeFerro", owner1
        );

IO`println("Initiate AddGroupClient"); club1.addGroupClient("PuxadoresDeFerro", client2,
owner1); club1.removeGroupClient("PuxadoresDeFerro", client2, owner1);

IO`println("Initiate Invoice");

club1.addInvoice(client1,{gymFeePayment1},
Utils`CreateDate(2019, 04, 23),
    Utils`CreateHour(08, 00), "gymFee",
owner1);

assertEqual(1, card club1.getInvoices());

club1.addInvoiceWithAllActivePayments(client1, Utils`CreateDate(2019, 04, 23),
    Utils`CreateHour(08, 00), "gymFee",
owner1);

club1.addPaymentToInvoice(invoice1, {new GymFeePayment(client1, 80, Utils`CreateDate(2019, 02, 22),
    Utils`CreateHour(18, 00))}, owner1);

club1.removePaymentFromInvoice(invoice1, {personalTrainingPayment1}, owner1);

IO`println("Initiate Clients"); assertEqual({client1, client2},
club1.getClients());
assertEqual({owner1, client1, client2, trainer1, trainer2, salesRepresentative1, salesRepresentative2}, club1.getUsers());
assertEqual({client1}, club1.getClientByName("Maria"));

IO`println("Initiate CRM");


assertEqual({|->}, club1.getCRM().getLeads());

club1.addLeadToCRM(lead1, owner1); club1.setCRMLeadSR(lead1,
salesRepresentative1, owner1);

club1.addLeadSRToCRM(lead2, salesRepresentative2, owner1);
```

```
  assertEqual({lead2.getName()}, salesRepresentative2.getLeads()); assertEqual(salesRepresentative1,
      club1.getCRM().getLeadSR(lead1));

  IO`println("Initiate removeLeadSR");
  club1.removeLeadSR(lead1, owner1);
  IO`println("Initiate removeCRMLead");
  club1.removeCRMLead(lead1, owner1);

  club1.transformLeadIntoClient(lead2, owner1);

  IO`println("Initiate addTaskToEmployee"); club1.addTaskToEmployee(trainer1, owner1,
  task1);

  IO`println("Initiate Products"); club1.addProduct(product1,
  owner1); club1.addStockOfProduct(product1, 7, owner1);
  assertEqual({product1}, club1.getProducts());
  club1.removeProduct(product1, owner1); assertEqual({},
  club1.getProducts()); club1.addProduct("Shaker", 3.99, 20,
  owner1);

  IO`println("Initiate Getters");
  assertEqual({trainer1, trainer2}, club1.getTrainers());
  assertEqual({salesRepresentative1, salesRepresentative2}, club1.getSalesRepresentatives()); assertEqual({trainer1, trainer2,
  salesRepresentative1, salesRepresentative2}, club1.
          getEmployees());

  assertEqual({trainer1}, club1.getEmployeeByName("Vasco")); assertEqual({trainer1},
  club1.getUserByName("Vasco"));

  assertEqual(owner1, club1.getOwner());
  assertEqual("Bombados", club1.getName()); assertEqual("Sales",
  club1.getNewsletter());

  assertEqual(1, card dom club1.getGroups());

  assertEqual(17, club1.getFee());

  assertEqual(club1, owner1.getClub());

  IO`println("Initiate Stats"); club1.getReportOnClubStatistics(owner1);
  club1.getClientActivity(client1, owner1); club1.getEmployeeActivity(trainer1, true,
  owner1);

  IO`println("Finalizing Club Tests");
 );

  end ClubTest
```

Function or operation	Line	Coverage	Calls
CRM	10	100.0%	4
addLead	17	100.0%	1
addLeadWithSR	27	100.0%	1
getLeadSR	89	100.0%	1
getLeads	80	100.0%	1
removeLead	58	100.0%	2
setLeadSR	38	100.0%	2
CRM.vdmpp		100.0%	12

Function or operation	Line	Coverage	Calls
Trainer	25	100.0%	5
addGymClass	50	79.0%	3
addTrainee	32	100.0%	1
addTrainingSession	63	44.8%	1
getActivity	77	100.0%	1
getClasses	137	0.0%	0
getTrainees	128	100.0%	1
getTrainingSessions	146	100.0%	1
removeTrainee	40	100.0%	1
Trainer.vdmpp		78.9%	14

Function or operation	Line	Coverage	Calls
TrainingSession	14	100.0%	1
getTrainee	30	0.0%	0
TrainingSession.vdmpp		80.7%	1

Function or operation	Line	Coverage	Calls
Club	46	100.0%	4
addAttendeeToGymClass	175	100.0%	2
addClient	75	100.0%	4
addGroup	112	90.0%	1
addGroupClient	299	100.0%	1
addGymClass	151	51.1%	0
addInvoice	317	91.8%	1
addInvoiceWithAllActivePayments	328	100.0%	1
addLeadSRToCRM	361	100.0%	1
addLeadToCRM	354	100.0%	2
addNewsletter	222	100.0%	1
addPaymentToInvoice	338	100.0%	1
addPersonalTraining	123	100.0%	1
addProduct	457	79.1%	1
addSalesRepresentative	99	100.0%	2
addStockOfProduct	482	100.0%	2
addTaskToEmployee	448	100.0%	2
addTrainer	87	100.0%	3
addTrainingSession	182	0.0%	0
getCRM	678	100.0%	2
getClientActivity	432	100.0%	2
getClientByName	576	100.0%	1
getClients	531	100.0%	3
getEmployeeActivity	441	100.0%	2
getEmployeeByName	592	100.0%	1
getEmployeeCalendar	687	100.0%	5
getEmployeeTasksForGivenDate	699	100.0%	5
getEmployees	558	100.0%	9
getFee	660	100.0%	1
getGroupByName	651	0.0%	0
getGroups	642	100.0%	1

getGymClasses	624	100.0%	1
getInvoices	669	100.0%	1
getName	504	100.0%	7
getNewsletter	513	100.0%	1
getOwner	522	100.0%	1
getProducts	710	100.0%	2
getReportOnClubStatistics	404	100.0%	1
getSalesRepresentatives	549	100.0%	1
getTrainers	540	100.0%	3
getTrainingSessions	633	100.0%	1
getUserByName	608	100.0%	1
getUsers	567	100.0%	61
isAtLeastEmployee	716	55.5%	31
isOwner	719	100.0%	18
removeCRMLead	385	100.0%	2
removeGroupClient	307	100.0%	1
removeLeadSR	375	100.0%	1
removePaymentFromInvoice	345	100.0%	2
removeProduct	489	100.0%	2
removeTraineeFromTrainer	139	100.0%	1
sendMessageAllClients	248	100.0%	1
sendMessageAllSalesRepresentatives	270	100.0%	1
sendMessageAllTrainers	259	100.0%	1
sendMessageClient	229	100.0%	3
sendMessageEmployee	241	100.0%	1
sendMessageToGroup	283	100.0%	1
sendOfferToGroup	291	100.0%	1
setCRMLeadSR	368	100.0%	2
setUserAccess	210	100.0%	1
transformLeadIntoClient	392	100.0%	1
Club.vdmpp		87.7%	214

4.3 EmployeeTest

```
class EmployeeTest is subclass of MyTestCase

 instance variables

   employee1: Employee := new Employee("Vasco", 25, <Male>, "brazilian");

   task1: Task := new Task("Reuni o com o patr o",
       Utils‘CreateHour(09, 00),
       Utils‘CreateHour(10, 00),
       Utils‘CreateDate(2019, 01, 12));


 operations

   public Run: () ==> () Run() == (
     IO‘println("\nEmployee Tests");

     employee1.addTask(task1);
```

```
    assertEqual({20190112 |-> [task1]}, employee1.getCalendar().getTasks());

    assertEqual([task1], employee1.getCalendar().getTasksForGivenDate(20190112));

    employee1.getActivity(true);

    IO'println("Finalizing Employee Tests");

  );

  end EmployeeTest
```

Function or operation	Line	Coverage	Calls
EmployeeCalendar	18	100.0%	9
addTask	25	100.0%	6
getTasks	41	100.0%	5
getTasksForGivenDate	50	100.0%	3
overlapsTask	62	96.1%	6
overlapsTasks	59	100.0%	6
EmployeeCalendar.vdmpp		98.9%	35

Function or operation	Line	Coverage	Calls
Employee	13	100.0%	9
addTask	23	100.0%	6
getActivity	31	100.0%	1
getCalendar	39	100.0%	6
Employee.vdmpp		100.0%	22

4.4 GroupTest

```
class GroupTest is subclass of MyTestCase

 instance variables

   client1: Client := new Client("Maria", 23, <Female>, "portuguese"); client2: Client := new
   Client("Jorge", 25, <Male>, "spanish");

   group1: Group := new Group({client1});

 operations

   public Run: () ==> () Run() == (
     IO'println("\nGroup Tests");

     group1.addClient(client2);
     assertEqual({client1, client2}, group1.getClients()); group1.removeClient(client2);
     assertEqual({client1}, group1.getClients());

     group1.sendOffer("Descontos muito bonitos"); group1.sendMessage(client1, "Feliz Natal
     Amigos");

     assertEqual({"Maria" |-> ["Feliz Natal Amigos"]}, group1.checkInbox(client1));

     assertEqual("Feliz Natal Amigos", group1.getLastMessageFromUser("Maria", client1));
```

```
    assertEqual(["Feliz Natal Amigos"], group1.getMessagesFromUser("Maria", client1));

    assertEqual(["Descontos muito bonitos"], group1.checkOffers(client1));

    IO‘println("Finalizing Group Tests");

  );

  end GroupTest
```

Function or operation	Line	Coverage	Calls
Group	16	100.0%	2
addClient	27	100.0%	2
checkInbox	68	100.0%	1
checkOffers	98	100.0%	1
getClients	108	100.0%	2
getLastMessageFromUser	78	100.0%	1
getMessagesFromUser	88	100.0%	1
removeClient	35	100.0%	2
sendMessage	51	75.0%	0
sendOffer	43	100.0%	4
Group.vdmpp		94.3%	16

4.5 GymClassTest

```
class GymClassTest is subclass of MyTestCase

  instance variables

    trainer1: Trainer := new Trainer("Vasco", 25, <Male>,"brazilian");

    client1: Client := new Client("Maria", 23, <Female>,"portuguese");

        gymClass1: GymClass := new GymClass("Aula de baixa intensidade", "Pilates", trainer1, <Tuesday
                >,
    Utils‘CreateHour(09, 00),
    Utils‘CreateHour(10, 00),
    Utils‘CreateDate(2019, 01, 15));


  operations

    public Run: () ==> () Run() == (
      IO‘println("\nGymClass Tests");

      gymClass1.addAttendee(client1);

      assertEqual("Pilates", gymClass1.getName()); assertEqual({client1.getID()},
      gymClass1.getAttendees());

      IO‘println("Finalizing GymClass Tests");

    );
```

```
end GymClassTest
```

Function or operation	Line	Coverage	Calls
GymClass	14	100.0%	5
addAttendee	26	100.0%	3
getAttendees	51	100.0%	1
getName	42	100.0%	1
GymClass.vdmpp		100.0%	10

4.6 InvoiceTest

```
class InvoiceTest is subclass of MyTestCase

 instance variables

  client1: Client := new Client("Maria", 23, <Female>, "portuguese");

  product1: Product := new Product("Prota", 29.99, 40); product2: Product := new
  Product("Shaker", 3.99, 60);

  gymFeePayment1: GymFeePayment := new GymFeePayment(client1, 80,
    Utils`CreateDate(2019, 01, 22),
        Utils`CreateHour(08, 00));

          gymFeePayment2: GymFeePayment := new GymFeePayment(client1, 80,
    Utils`CreateDate(2019, 02, 22),
        Utils`CreateHour(08, 00));

  invoice1: Invoice := new Invoice({gymFeePayment1}, Utils`CreateDate(2019, 04, 23),
        Utils`CreateHour(08, 00), "gymFee", false,
    client1);

  personalTrainingPayment1: PersonalTrainingPayment := new PersonalTrainingPayment(client1, 50, Utils`CreateDate(2019, 03, 13),
        Utils`CreateHour(08, 00));

          personalTrainingPayment2: PersonalTrainingPayment := new PersonalTrainingPayment(client1, 50,
             Utils`CreateDate(2019, 04, 13),
        Utils`CreateHour(08, 00));

  invoice2: Invoice := new Invoice({personalTrainingPayment1}, Utils`CreateDate(2019, 04,
    23),
        Utils`CreateHour(08, 00), "personalTraining", false,
    client1);


 operations

  public Run: () ==> () Run() == (
    IO`println("\nInvoice Tests");

    invoice2.addPayment({personalTrainingPayment2});

    assertEqual({personalTrainingPayment1, personalTrainingPayment2},invoice2.getPayments()); assertEqual(100,
    invoice2.getTotalAmount()); invoice2.removePayment({personalTrainingPayment1});
    assertEqual({personalTrainingPayment2}, invoice2.getPayments());
    assertEqual(50, invoice2.getTotalAmount());

      invoice1.addPayment({gymFeePayment2});
    assertEqual({gymFeePayment1, gymFeePayment2}, invoice1.getPayments()); assertEqual(160,
    invoice1.getTotalAmount());
    invoice1.removePayment({gymFeePayment1});
    assertEqual({gymFeePayment2},invoice1.getPayments()); assertEqual(80,
    invoice1.getTotalAmount());

    assertEqual(20190423, invoice1.getDate()); assertEqual(0800,
    invoice1.getHour()); assertEqual("gymFee", invoice1.getType());
    assertEqual(client1,invoice1.getClient());

        assertEqual(20190423, invoice2.getDate()); assertEqual(0800,
    invoice2.getHour()); assertEqual("personalTraining",
    invoice2.getType()); assertEqual(client1, invoice2.getClient());


    IO‘println("Finalizing Invoice Tests");

  );

  end InvoiceTest
```

Function or operation	Line	Coverage	Calls
GymFeePayment	8	100.0%	8
getFee	25	100.0%	1
GymFeePayment.vdmpp		100.0%	9

Function or operation	Line	Coverage	Calls
Invoice	25	93.7%	4
addPayment	70	94.1%	9
getClient	152	100.0%	2
getDate	125	100.0%	2
getHour	134	100.0%	2
getPayments	107	100.0%	4
getTotalAmount	116	100.0%	4
getType	143	100.0%	2
removePayment	91	100.0%	3
Invoice.vdmpp		95.9%	32

4.7 LeadTest

```
class LeadTest is subclass of MyTestCase

instance variables

  lead1: Lead := new Lead("Maria", 23, <Female>, "portuguese"); lead2: Lead := new
  Lead("Jorge", 25, <Male>, "spanish");

operations

  public Run: () ==> () Run() == (
    IO‘println("\nLead Tests");

    assertEqual("Maria", lead1.getName());

    assertEqual(23, lead1.getAge());

    assertEqual(<Female>, lead1.getGender());

    assertEqual("portuguese", lead1.getNationality());
    assertEqual(0, lead1.getID()); assertEqual(1,
    lead2.getID());

    IO‘println("Finalizing Lead Tests");

  );

  end LeadTest
```

Function or operation	Line	Coverage	Calls
Lead	21	100.0%	6
getAge	56	100.0%	2
getGender	65	100.0%	2
getID	83	100.0%	2
getName	47	100.0%	27
getNationality	74	100.0%	2
Lead.vdmpp		100.0%	41

4.8 PaymentTest

```
class PaymentTest is subclass of MyTestCase

 instance variables

   client1: Client := new Client("Maria", 23, <Female>, "portuguese");

   product1: Product := new Product("Prota", 29.99, 40); product2: Product := new
   Product("Shaker", 3.99, 60);

   gymFeePayment1: GymFeePayment := new GymFeePayment(client1, 80,
     Utils‘CreateDate(2019, 01, 22),
         Utils‘CreateHour(08, 00));

   personalTrainingPayment1: PersonalTrainingPayment := new PersonalTrainingPayment(client1, 50, Utils‘CreateDate(2019, 01, 23),
         Utils‘CreateHour(08, 00));

   productPayment1: ProductPayment := new ProductPayment(client1, product1, 2, Utils‘CreateDate(2019, 01, 30),
         Utils‘CreateHour(18, 29));

 operations

   public Run: () ==> () Run() == (
     IO‘println("\nPayment Tests");

     assertEqual(80, gymFeePayment1.getFee());

     assertEqual(50, personalTrainingPayment1.getFee());

     assertEqual([product1], productPayment1.getProducts());
     productPayment1.addProduct(product2, 1);
```

```
  assertEqual([product1, product2], productPayment1.getProducts());

  assertEqual(20190122, gymFeePayment1.getDate());
  assertEqual(0800, gymFeePayment1.getHour()); assertEqual(80,
  gymFeePayment1.getAmount()); assertEqual(client1,
  gymFeePayment1.getClient()); assertEqual(0,
  gymFeePayment1.getID());

  IO‘println("Finalizing Payment Tests");

 );

 end PaymentTest
```

Function or operation	Line	Coverage	Calls
PersonalTrainingPayment	9	100.0%	5
getFee	26	100.0%	1
PersonalTrainingPayment.vdmpp		100.0%	6

Function or operation	Line	Coverage	Calls
ProductPayment	15	100.0%	2
addProduct	33	100.0%	1
getProducts	50	100.0%	2
ProductPayment.vdmpp		100.0%	5

Function or operation	Line	Coverage	Calls
Owner	12	100.0%	4
Owner.vdmpp		100.0%	4

Function or operation	Line	Coverage	Calls
Payment	20	100.0%	15
getAmount	58	100.0%	12
getClient	67	100.0%	3
getDate	40	100.0%	1
getHour	49	100.0%	1
getID	76	100.0%	1
Payment.vdmpp		100.0%	33

4.9 PerfectGymTest

```
class PerfectGymTest is subclass of MyTestCase

 instance variables

     owner1: Owner := new Owner("Rui", 21, <Male>, "portuguese"); owner2: Owner := new
   Owner("Tiago", 21, <Male>, "portuguese");

   club1: Club := new Club("Bombados", owner1, 17); club2: Club :=
   new Club("PuxaFerro", owner2, 35);

   perfectGym1: PerfectGym := PerfectGym‘getInstance();
```

```
operations

  public Run: () ==> () Run() == (
    IO‘println("\nPerfectGym Tests");

    perfectGym1.addClub(club1);
    perfectGym1.addClub(club2);

    assertEqual({"Bombados"|->club1, "PuxaFerro"|->club2}, perfectGym1.getClubs());

    IO‘println("Finalizing PerfectGym Tests");
  );

   end PerfectGymTest
```

Function or operation	Line	Coverage	Calls
PerfectGym	13	100.0%	1
addClub	20	100.0%	6
getClubs	41	100.0%	1
getInstance	32	100.0%	1
PerfectGym.vdmpp		100.0%	9

4.10 ProductTest

```
class ProductTest is subclass of MyTestCase

 instance variables

      product1: Product := new Product("Prota", 29.99, 40);


 operations

  public Run: () ==> () Run() == (
    IO‘println("\nProduct Tests");

    assertEqual("Prota",product1.getName());
    assertEqual(29.99, product1.getValue()); assertEqual(40,
    product1.getQuantity());

    product1.addQuantity(10); assertEqual(50,
    product1.getQuantity());

    product1.removeQuantity(30); assertEqual(20,
    product1.getQuantity());

    IO‘println("Finalizing Product Tests");
  );

   end ProductTest
```

Function or operation	Line	Coverage	Calls
Product	16	100.0%	8
addQuantity	30	100.0%	2
getName	48	100.0%	10

getQuantity	66	100.0%	4
getValue	57	100.0%	4
removeQuantity	37	100.0%	5
Product.vdmpp		100.0%	33

4.11 SalesRepresentative

```
class SalesRepresentativeTest is subclass of MyTestCase

 instance variables

   salesRepresentative1: SalesRepresentative := new SalesRepresentative("Vasco", 25, <Male>, " brazilian");

   lead1: Lead := new Lead("Maria", 23, <Female>, "portuguese"); lead2: Lead := new
   Lead("Jorge", 25, <Male>, "spanish");


 operations

   public Run: () ==> () Run() == (
     IO‘println("\nSalesRepresentative Tests");

     salesRepresentative1.addLead(lead1); salesRepresentative1.addLead(lead2);
```

```
    assertEqual({lead1.getName(), lead2.getName()}, salesRepresentative1.getLeads()); salesRepresentative1.removeLead(lead2);
    assertEqual({lead1.getName()}, salesRepresentative1.getLeads());

    salesRepresentative1.getActivity(true);

    IO‘println("Finalizing SalesRepresentative Tests");

  );

  end SalesRepresentativeTest
```

Function or operation	Line	Coverage	Calls
SalesRepresentative	12	100.0%	3
addLead	19	100.0%	4
getActivity	38	36.0%	0
getLeads	82	100.0%	3
removeLead	27	100.0%	3
SalesRepresentative.vdmpp		64.2%	13

4.12 SessionTest

```
class SessionTest is subclass of MyTestCase

 instance variables

  trainer1: Trainer := new Trainer("Vasco", 25, <Male>, "brazilian");

      session1: Session := new Session("Treino", trainer1,<Monday>, Utils‘CreateHour(09,
      00),
      Utils‘CreateHour(10, 00),
      Utils‘CreateDate(2019, 01, 12));


 operations

  public Run: () ==> () Run() == (
    IO‘println("\nSession Tests");

    assertEqual(trainer1.getID(), session1.getTrainer()); assertEqual(<Monday>,
    session1.getDayOfWeek());

    IO‘println("Finalizing Session Tests");

  );

  end SessionTest
```

Function or operation	Line	Coverage	Calls
Session	17	100.0%	7
getDayOfWeek	43	100.0%	1
getTrainer	34	100.0%	149
Session.vdmpp		100.0%	157

4.13 TaskTest

```
class TaskTest is subclass of MyTestCase

instance variables

     task1: Task := new Task("Reuni o com o patr o",
     Utils‘CreateHour(09, 00),
     Utils‘CreateHour(10, 00),
     Utils‘CreateDate(2019, 01, 12));


operations

  public Run: () ==> () Run() == (
   IO‘println("\nTask Tests");

   assertEqual("Reuni  o com o patr o", task1.getDescription()); assertEqual(0900,
   task1.getStartHour());
   assertEqual(1000, task1.getEndHour()); assertEqual(20190112,
   task1.getDate());

   IO‘println("Finalizing Task Tests");

  );

   end TaskTest
```

Function or operation	Line	Coverage	Calls
Task	22	100.0%	10
getDate	70	100.0%	86
getDescription	43	100.0%	5
getEndHour	61	100.0%	17
getStartHour	52	100.0%	17
Task.vdmpp		100.0%	135

4.14 UserTest

```
class UserTest is subclass of MyTestCase

instance variables

 user1: Client := new Client("Rui", 21, <Male>, "portuguese");

 client1: Client := new Client("Rui", 21, <Male>, "portuguese");
 client2: Client := new Client("Tiago", 22, <Male>, "portuguese");

 owner1: Owner := new Owner("Maria", 21, <Female>, "portuguese");

 club1: Club := new Club("Bombados", owner1, 17);

operations

 public Run: () ==> ()
 Run() == (
  IO`println("\nUser Tests");

  club1.addClient(client1, owner1);
  club1.addClient(client2, owner1);

  assertEqual(21, client1.getAge());
  assertEqual(<Male>, client1.getGender());
  assertEqual("portuguese", client1.getNationality());

  client2.sendMessage(client1, "ola");
  assertEqual({"Tiago" |-> ["ola"]}, client1.checkInbox());
  assertEqual(["ola"], client1.readMessagesFromUser(client2));
  assertEqual("ola", client1.readLastMessageFromUser(client2));
  assertEqual("ola", client1.readMessageNFromUser(1, client2));

  client1.deleteLastMessageFromUser("Tiago");

  client2.sendMessage(client1, "ola");
  client1.deleteMessageNFromUser(1, "Tiago");

  user1.getActivity();

  IO`println("Finalizing User Tests");

 );

end UserTest
```

Function or operation	Line	Coverage	Calls
User	25	100.0%	48
checkInbox	104	100.0%	1
deleteLastMessageFromUser	79	100.0%	1
deleteMessageNFromUser	86	100.0%	1
getAccess	167	100.0%	104
getActivity	141	0.0%	0
getAge	176	100.0%	1
getClub	203	100.0%	2
getGender	185	100.0%	1
getID	158	100.0%	62
getName	149	100.0%	105
getNationality	194	100.0%	1
readLastMessageFromUser	123	100.0%	1
readMessageNFromUser	133	100.0%	1
readMessagesFromUser	113	100.0%	3
receiveMessage	65	100.0%	13

sendMessage	58	100.0%	2
setAccess	94	100.0%	2
setClub	49	100.0%	29
User.vdmpp		99.6%	378

4.15 UseCasesTest

```
class UseCaseTest is subclass of MyTestCase

instance variables

 owner3: Owner := new Owner("Bernardo", 21, <Male>, "portuguese");

 club3: Club := new Club("Protas", owner3, 47);

 perfectGym1: PerfectGym := PerfectGym`getInstance();

 client1: Client := new Client("Maria", 23, <Female>, "portuguese");
 client2: Client := new Client("Jorge", 25, <Male>, "spanish");

 trainer1: Trainer := new Trainer("Vasco", 25, <Male>, "brazilian");
 trainer2: Trainer := new Trainer("Alex", 33, <Male>, "english");

 salesRepresentative1: SalesRepresentative := new SalesRepresentative("Joana", 21, <Female>, "
     portuguese");
 salesRepresentative2: SalesRepresentative := new SalesRepresentative("Manuel", 33, <Male>, "
     french");

 employee1: Employee := new Employee("Zeca", 25, <Male>, "brazilian");

 lead1: Lead := new Lead("Maria", 23, <Female>, "portuguese");
 lead2: Lead := new Lead("Jorge", 25, <Male>, "spanish");

 gymFeePayment1: GymFeePayment := new GymFeePayment(client1, 80,
  Utils`CreateDate(2019, 01, 22),
    Utils`CreateHour(08, 00));

 task1: Task := new Task("Reuni o com o patr o",
   Utils`CreateHour(09, 00),
   Utils`CreateHour(10, 00),
   Utils`CreateDate(2019, 01, 12));

 product1: Product := new Product("Prota", 29.99, 40);

 productPayment1: ProductPayment := new ProductPayment(client1, product1, 2,
  Utils`CreateDate(2019, 01, 30),
    Utils`CreateHour(18, 29));


operations

 public Run: () ==> ()
 Run() == (
  IO`println("\nUseCase Tests");

  IO`println("Use Case R1");
  perfectGym1.addClub(club3);

  IO`println("Use Case R2");
  club3.addClient(client1, owner3);
  club3.addClient(client2, owner3);
  club3.addTrainer(trainer1, owner3);
  club3.addTrainer(trainer2, owner3);
  club3.addSalesRepresentative(salesRepresentative1, owner3);
  club3.addSalesRepresentative(salesRepresentative2, owner3);

  IO`println("Use Case R3");
  club3.setUserAccess(owner3, trainer1, <Owner>);

  IO`println("Use Case R4");
  club3.getReportOnClubStatistics(owner3);
  club3.getClientActivity(client1, owner3);
  club3.getEmployeeActivity(trainer1, true, owner3);

  IO`println("Use Case R5");
  assertEqual({|->}, club3.getCRM().getLeads());
  club3.addLeadToCRM(lead1, owner3);
  club3.setCRMLeadSR(lead1, salesRepresentative1, owner3);
  club3.addLeadSRToCRM(lead2, salesRepresentative2, owner3);
  assertEqual({lead2.getName()}, salesRepresentative2.getLeads());
    assertEqual(salesRepresentative1, club3.getCRM().getLeadSR(lead1));
  club3.removeLeadSR(lead1, owner3);
  club3.removeCRMLead(lead1, owner3);
  club3.transformLeadIntoClient(lead2, owner3);

  IO`println("Use Case R6");
  club3.addNewsletter("Sales", owner3);
  club3.sendMessageClient("Bom Ano", client1, owner3);
  club3.sendMessageEmployee("Tas despedido bro!", trainer1, owner3);
  club3.sendMessageAllClients("Feliz Natal", owner3);
  club3.sendMessageAllTrainers("Feliz Natal", owner3);

  IO`println("Use Case R7");
  club3.addInvoice(client1, {gymFeePayment1},
  Utils`CreateDate(2019, 04, 23),
    Utils`CreateHour(08, 00),
  "gymFee", owner3);
  assertEqual(1, card club3.getInvoices());

  IO`println("Use Case R8");
  employee1.addTask(task1);
  assertEqual({20190112 |-> [task1]}, employee1.getCalendar().getTasks());

  IO`println("Use Case R9");
  client1.addTrainer(trainer1, 30);
  assertEqual(trainer1, client1.getTrainer());

  IO`println("Use Case R10");
  assertEqual("Prota", product1.getName());
  assertEqual(29.99, product1.getValue());
  assertEqual(38, product1.getQuantity());
  product1.addQuantity(12);
  assertEqual(50, product1.getQuantity());
  product1.removeQuantity(30);
  assertEqual(20, product1.getQuantity());

  IO`println("Use Case R11");
  client2.sendMessage(client1, "ola");
  assertEqual({"Jorge" |-> ["ola"], "Bernardo" |-> ["Sales", "Feliz Natal", "Sales", "Bom Ano"]},
       client1.checkInbox());
  assertEqual(["ola"], client1.readMessagesFromUser(client2));
  assertEqual("ola", client1.readLastMessageFromUser(client2));
  assertEqual("ola", client1.readMessageNFromUser(1, client2));
  client1.deleteLastMessageFromUser("Jorge");
  client2.sendMessage(client1, "ola");
  client1.deleteMessageNFromUser(1, "Jorge");

  IO`println("Use Case R12");
  club3.addGroup("PuxadoresDeFerro", {client1}, owner3);
  club3.sendMessageToGroup("Feliz Natal", "PuxadoresDeFerro", owner3);

  IO`println("Use Case R13");
  assertEqual({productPayment1}, client1.getProductPayments());
  client1.removeProductPayment(productPayment1);
  assertEqual({productPayment1}, client1.getHistoryProductPayments());
  client1.addProductBought(product1, 1, 29.99);
  assertEqual(89.97, client1.getTotalSPentOnProducts());
  assertEqual({"Prota" |-> 3}, client1.getProductsBought());

  IO`println("Finalizing UseCase Tests");
 );

end UseCaseTest
```

5. Model Verification

5.1 Domain Verification

PO Name	Type
Club` sendMessageToGroup (msg, groupName, user)	legal map application

The code under analysis is:

```
public sendMessageToGroup: String1 * String1 * User ==> ()
        sendMessageToGroup(msg, groupName, user) ==

                groups(groupName).sendMessage(user, msg)

        pre msg <> "" and groupName <> "" and groupName in set dom groups and (user in set clients or
isOwner(user)) and user in set getUsers();
```

In this case the proof is easy because the verification groupName in set dom groups ensures that the groupName exists in groups, making sure that the group desired exists.

5.2 Invariants Verification

PO Name	Type
Club` addGymClass (gymClass, user)	legal map application

The code under analysis is:

```
public addGymClass: GymClass * User ==> ()

addGymClass(gymClass, user) == classes := classes union {gymClass}

pre (isAtLeastEmployee(user)) and user in set getUsers() and gymClass not in set classes

post not exists c1, c2 in set classes & c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1.getDate() =
c2.getDate() and        Utils`overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(),
c2.getEndHour());
```

The relevant invariant under analysis is:

```
inv not exists c1, c2 in set classes & c1 <> c2 and c1.getTrainer() = c2.getTrainer() and c1.getDate() =
c2.getDate() and Utils`overlaps(c1.getStartHour(), c1.getEndHour(), c2.getStartHour(), c2.getEndHour());
```

The post condition, assures that after the gymClass is added, there are no 2 gymClasses that have the same trainer and date and that overlap their hours. Hours overlap is verified in the following function:

```
public static overlaps: Hour * Hour * Hour * Hour-> bool
```

```
overlaps(startHour1, endHour1, startHour2, endHour2) ==
((startHour1 >= startHour2 and startHour1 < endHour2) or
        (endHour1 > startHour2 and endHour1 <= endHour2) or
        (startHour1 <= startHour2 and endHour1 >= endHour2));
```

This makes sure that the two hour intervals do not intersect.

6. Code Generation

After the Java code generation, the group faced a few problems easy to fix.

The first had to do with the created types that are quotes in the java code, It did not include this quotes, so they had to be added manually in each file that required them.

The second had to do with printing classes. Because of a loop that was created thanks to one class having an association to another, and this having an association to the first. The solution has been to change one of the classes association to a String (for example, leads names in the SalesRepresentative class instead of having the actual leads).

After correcting this issues we created a GUI to make it easier to test the model. In this GUI we implemented almost all methods and all of them worked, although no pre and post conditions as well as invariants were generated.

7. Conclusions

The model that was developed covers all the requirements as well as several others.

We also implemented a GUI to help the user test the application.

If space permitted more features would have been implemented and the GUI would have a better design.

This project took approximately 120 hours to develop.

Contribution:

- Rui Quaresma – 50 %
- Tiago Carvalho – 50 %

8. References

1. VDM slides, https://moodle.up.pt/pluginfile.php/183550/mod_resource/content/0/VDM%2B%2B1.pdf

2. VDM slides 2, https://moodle.up.pt/pluginfile.php/185263/mod_resource/content/0/VDM%2B%2B2.pdf

3. Overture Quick Start,
https://moodle.up.pt/pluginfile.php/25618/mod_resource/content/0/OverturQuickStartExercise.pdf