

MFES

January 6, 2019

Contents

1	Date	1
2	Event	2
3	EventManager	4
4	EventTest	6
5	MainTest	9
6	Ticket	9
7	TicketManager	10
8	TicketTest	11
9	User	12
10	UserManager	14
11	UserTest	16

1 Date

```
class Date
/*
  Contains Date type to be used in events
*/
types
public Date :: day : nat
           month : nat
           year : nat
  inv d == d.year > 2018 and
       d.month <= 12 and
       d.day <= DaysOfMonth(d.year, d.month);
values
instance variables

operations
functions
  -- Functions to check if a date is valid, used in Date's invariant
```

```

public DaysOfMonth : nat * nat -> nat
DaysOfMonth(y, m) ==
  if (m = 2) then (
    if isLeapYear(y) then 29
    else 28
  )
  else if (m = 4 or m = 6 or m = 9 or m = 11) then 30
  else 31;

-- Checks if a year is leap year

public isLeapYear : nat -> bool
isLeapYear (y) ==
  y mod 4 = 0 and y mod 100 <> 0 or y mod 400 = 0;
traces
end Data

```

2 Event

```

class Event is subclass of Data
/*
  Contains the core model of the event.
*/
types
  public String = seq of char;
values
instance variables
  -- Event's name
  private name : String;
  -- Event's date
  private date : Date;
  -- Event's max number of tickets
  private capacity : nat;
  -- Event's popularity, used to sort display of events
  private popularity: nat := 0;
  -- Set of ticket IDs for the event
  private tickets : set of nat := {};
  inv card tickets <= capacity;
  -- Ticket price to be subtracted from buyer
  private ticketPrice : rat;
operations
  -- Event constructor, receives event name, capacity, price and Date

  public Event : String * nat * rat * Date ==> Event
  Event(n, c, p, d) == (
    name := n;
    capacity := c;
    ticketPrice := p;
    date := d;
    return self
  )
  pre p > 0
  post name = n and capacity = c and tickets = {} and popularity = 0;

  -- Returns Event's name

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

```

```

-- Returns Event's max number of tickets

public pure getCapacity : () ==> nat
getCapacity()==(
  return capacity;
);

-- Returns ticket price for the event

public pure getTicketPrice : () ==> nat
getTicketPrice()==(
  return ticketPrice;
);

-- Returns fill percentage, dividing number of tickets by capacity * 100

public pure getFillPercent : () ==> real
getFillPercent()==(
  return (card (tickets) / capacity ) * 100;
);

-- Returns set of ticket IDS

public pure getTickets : () ==> set of nat
getTickets() == (
  return tickets;
);

/*
  Adds a ticket ID to set of ticket IDS
  Pre condition checks if ticket amount is smaller than the max and if ticket ID isnt in set of
  IDS
*/

public addTicket : nat ==> ()
addTicket(ticket) == (
  tickets := tickets union {ticket}
)
pre card tickets <= capacity and ticket not in set tickets
post tickets = tickets~ union {ticket};

/*
  Removes a ticket ID from set of ticket IDS
  Pre condition checks if ticket ID is in set of ticket IDS
*/

public removeTicket : nat ==> ()
removeTicket(i) == (
  tickets:= tickets \ {i};
)
pre i in set tickets
post tickets = tickets~ \ {i};

-- Increases event popularity by 10

public promote : () ==> ()
promote() == (
  popularity := popularity + 10;
)
pre popularity + 10 <= 100;

-- Returns event's popularity

public getPopularity : () ==> nat

```

```

getPopularity() == (
  return popularity;
);

-- Returns earnings made by an event, multiplying number of tickets by ticket price

public getEarnings : () ==> rat
getEarnings() == (
  return (card tickets) * ticketPrice;
);

-- Returns event's Date

public getDate : () ==> Date
getDate() == (
  return date;
);

functions
traces
end Event

```

3 EventManager

```

class EventManager
/*
  Contains every event.
  Defines the operations available related to events.
*/
types
public String = seq of char;
values
instance variables
  -- map of event name String to Event, representing every event created
  private events : map String to Event := {|->};
operations
  -- Event manager constructor, receives a map String to Event

  public EventManager : map String to Event ==> EventManager
  EventManager(evs) == (
    events := evs;
    return self;
  )
  post events = evs;

  -- Returns map String to Event representing every event

  public pure getEvents : () ==> map String to Event
  getEvents() == (
    return events;
  );

  -- Returns a specific Event, receives String for event's name

  public pure getEvent : String ==> Event
  getEvent (e) == (
    return events(e);
  )
  pre e in set dom events;

```

```

/*
  Adds an event to events map
  Pre condition checks if there isn't an event with same name
  Post condition checks if event was added to map
*/

public addEvent : Event ==> ()
addEvent(event) == (
  events := events ++ {event.getName() |-> event}
)
pre event.getName() not in set dom events
post events = events~ ++ {event.getName() |-> event};

/*
  Removes an event from events map
  Pre condition checks if event exists in map
  Post condition checks if event was removed
*/

public removeEvent : String ==> ()
removeEvent(e) == (
  events:= {e} <-: events;
)
pre e in set dom events
post events = {e} <-: events~;

-- Checks if an event exist in events map, returns a bool and receives event name

private pure eventExists : String ==> bool
eventExists(e) == (
  return e in set dom events
);

/*
  Returns set of nat for ticket ID's from an event, receives event name
  Pre condition checks if event exists
*/

public pure getEventTickets : String ==> set of nat
getEventTickets(e) == (
  return events(e).getTickets()
)
pre eventExists(e);

/*
  Returns set of nat for ticket ID's from an event owned by a certain user
  Receives event name and User name
  Pre condition checks if event exists
*/

public getEventTicketsUser : String*String*TicketManager ==> set of nat
getEventTicketsUser(e,u,tm) == (
  dcl tickets : set of nat := {};
  for all ticket in set (events(e).getTickets()) do
  (
    if tm.getTickets()(ticket).getOwner() = u
    then tickets := tickets union {ticket};
  );
  return tickets;
)
pre eventExists(e);

/*
  Returns rat for fill percentage in an event
  Receives event name

```

```

    Pre condition checks if event exists
    */

public getEventFillPercent : String ==> rat
getEventFillPercent(e) == (
    return events(e).getFillPercent();
)
pre eventExists(e);

/*
    Adds a ticket to a specific event
    Receives ticket ID and event name
    Pre condition checks if event exists
    */

public addTicket : nat*String ==> ()
addTicket(t,e) == (
    events(e).addTicket(t);
)
pre eventExists(e);

/*
    Returns a rat for a specific event's earnings
    Receives a string for event's name
    Pre condition checks if event exists
    */

public getEarnings : String ==> rat
getEarnings(e) == (
    return events(e).getEarnings();
)
pre eventExists(e);

functions
traces
end EventManager

```

4 EventTest

```

class EventTest is subclass of Data
types
-- TODO Define types here
values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
e : Event := new Event("evento1", 15, 10, mk_Date(10,1,2019));
e2 : Event := new Event("evento2", 15, 10, mk_Date(29,2,2020));
e3 : Event := new Event("evento3", 15, 10, mk_Date(29,4,2020));
e4 : Event := new Event("evento4", 15, 10, mk_Date(28,2,2021));
u : User := new User("dank", "memes");
em : EventManager := new EventManager({ "evento1" |-> e, "evento2" |-> e2, "evento3" |-> e3, "
    evento4" |-> e4});
um : UserManager := new UserManager({ "dank" |-> u});
tm : TicketManager := new TicketManager({|->});
operations
-- TODO Define operations here

private assertTrue: bool ==> ()

```

```

    assertTrue(cond) == return
pre cond;

private testGetDate: () ==> ()
testGetDate() == (
    assertTrue(e.getDate() = mk_Date(10,1,2019));
);

private testGetName: () ==> ()
testGetName() == (
    assertTrue(e.getName() = "event01");
);

private testCapacity: () ==> ()
testCapacity() == (
    assertTrue(e.getCapacity() = 15);
);

private testGetTicketPrice: () ==> ()
testGetTicketPrice() == (
    assertTrue(e.getTicketPrice() = 10);
);

private testGetTickets : () ==> ()
testGetTickets() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("event01",tm,em);
    um.buyTicket("event01",tm,em);
    assertTrue(e.getTickets() = {6,7});
);

private testRemoveTicket : () ==> ()
testRemoveTicket() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("event01",tm,em);
    e.removeTicket(8);
    assertTrue(e.getTickets() = {});
);

private testGetFillPercent : () ==> ()
testGetFillPercent() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("event01",tm,em);
    assertTrue(em.getEventFillPercent("event01") = (1 / 15) * 100);
);

private testGetEarnings : () ==> ()
testGetEarnings() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("event01",tm,em);
    assertTrue(em.getEarnings("event01") = 10);
);

```

```

private testPromotion : () ==> ()
testPromotion() == (
    e.promote();
    assertTrue(e.getPopularity() = 10);
);

private testManagerGetEvent : () ==> ()
testManagerGetEvent() == (
    assertTrue(em.getEvent("eventol") = e);
);

private testManagerAddEvent : () ==> ()
testManagerAddEvent() == (
    dcl eventTest : Event := new Event("eventTest",10,10,mk_Date(1,1,2021));
    em.addEvent(eventTest);
    assertTrue(em.getEvent("eventTest") = eventTest);
);

private testManagerRemoveEvent : () ==> ()
testManagerRemoveEvent() == (
    dcl eventTest : Event := new Event("eventTest",10,10,mk_Date(1,1,2021));
    em.addEvent(eventTest);
    em.removeEvent("eventTest");
    assertTrue(card dom em.getEvents() = 4);
);

private testManagerGetEventTickets : () ==> ()
testManagerGetEventTickets() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("eventol",tm,em);
    um.buyTicket("eventol",tm,em);
    assertTrue(card em.getEventTickets("eventol") = 2);
);

private testManagerGetEventTicketsUser : () ==> ()
testManagerGetEventTicketsUser() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.buyTicket("eventol",tm,em);
    um.buyTicket("eventol",tm,em);
    um.buyTicket("evento2",tm,em);
    assertTrue(card em.getEventTicketsUser("eventol","dank",tm) = 2);
);

public static main: () ==> ()
main() == (
    new EventTest().testGetDate();
    new EventTest().testGetName();
    new EventTest().testCapacity();
    new EventTest().testGetTicketPrice();
    new EventTest().testGetTickets();
    new EventTest().testRemoveTicket();
    new EventTest().testGetFillPercent();
    new EventTest().testGetEarnings();
    new EventTest().testPromotion();
    new EventTest().testManagerGetEvent();
    new EventTest().testManagerAddEvent();

```



```

    new EventTest().testManagerRemoveEvent();
    new EventTest().testManagerGetEventTickets();
    new EventTest().testManagerGetEventTicketsUser();
  );
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end EventTest

```

5 MainTest

```

class MainTest
types
-- TODO Define types here
values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
operations
-- TODO Define operations here

public static main: () ==> ()
main() == (
  new TicketTest().main();
  new EventTest().main();
  new UserTest().main();
);
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end MainTest

```

6 Ticket

```

class Ticket
/*
  Defines a ticket for an event and owned by a user.
*/
types
  public String = seq of char;
values
instance variables
  -- Ticket owner's name
  private owner : String;
  -- Ticket event's name
  private event : String;
  -- Ticket id
  private id: nat;
  -- Count used for ids
  static count : nat := 0;
operations
  -- Ticket constructor, receives owner's name and event's name

```

```

public Ticket: String * String ==> Ticket
Ticket(o,e) == (
  owner := o;
  event := e;
  id := count;
  count := count + 1;
  return self;
)
pre count >= 0
post owner = o and event = e and id = count - 1;

-- Returns ticket owner's name

public pure getOwner : () ==> String
getOwner() == (
  return owner;
);

-- Returns ticket event's name

public pure getEvent : () ==> String
getEvent() == (
  return event;
);

-- Returns ticket's ID

public pure getID : () ==> nat
getID() == (
  return id;
);

functions
traces
end Ticket

```

7 TicketManager

```

class TicketManager
/*
  Contains every ticket.
  Defines the operations available related to tickets.
*/
types
values
instance variables
  private tickets : map nat to Ticket := {|->};
operations
  -- TicketManager constructor, receives map nat (ticket ID) to Ticket

  public TicketManager : map nat to Ticket ==> TicketManager
  TicketManager(ts) == (
    tickets := ts;
    return self;
  )
  post tickets = ts;

  -- Returns map nat (ticketID) to Ticket

```

```

public pure getTickets : () ==> map nat to Ticket
getTickets() == (
  return tickets;
);

/*
  Adds a ticket to tickets map
  pre condition checks if ticket id isnt in tickets map
  post condition checks if ticket was added to map
*/

public addTicket : Ticket ==> ()
addTicket(ticket) == (
  tickets := tickets ++ {ticket.getID() |-> ticket}
)
pre ticket.getID() not in set dom tickets
post tickets = tickets~ ++ {ticket.getID() |-> ticket};

/*
  Removes a ticket from tickets map
  pre condition checks if ticket id is in map
*/

public removeTicket : nat ==> ()
removeTicket(i) == (
  tickets:= {i} <-: tickets;
)
pre i in set dom tickets

functions
traces
end TicketManager

```

8 TicketTest

```

class TicketTest is subclass of Data
types
-- TODO Define types here
values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
e : Event := new Event("eventol", 15, 10, mk_Date(21,2,2019));
u : User := new User("dank", "memes");
em : EventManager := new EventManager({ "cenas" |-> e });
um : UserManager := new UserManager({"dank" |-> u});
tm : TicketManager := new TicketManager({|->});
t : Ticket := new Ticket("dank", "eventol");
operations
-- TODO Define operations here

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

private testGetID : () ==> ()
testGetID() == (
  assertTrue(t.getID() = 1);
);

```

```

private testGetOwner : () ==> ()
testGetOwner() == (
  assertTrue(t.getOwner() = "dank");
);

private testGetEvent : () ==> ()
testGetEvent() == (
  assertTrue(t.getEvent() = "evento1");
);

private testRemoveTicket : () ==> ()
testRemoveTicket() == (
  dcl previousCard : nat := card dom tm.getTickets();
  dcl ticketTest : Ticket := new Ticket("dank", "evento3");
  tm.addTicket(ticketTest);
  tm.removeTicket(ticketTest.getID());
  assertTrue(card dom tm.getTickets() = previousCard);
);

public static main: () ==> ()
main() == (
  new TicketTest().testGetID();
  new TicketTest().testGetOwner();
  new TicketTest().testGetEvent();
  new TicketTest().testRemoveTicket();
);
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end TicketTest

```

9 User

```

class User
/*
  Represents a user.
  Defines the variables and operations available to users.
*/
types
  public String = seq of char;
values
instance variables
  private name : String;
  private funds : nat := 0;
  private password : String;
  private events : set of String := {};
  private tickets : set of nat := {};
operations
  -- User constructor, receives Strings for user name and password

  public User: String * String ==> User
  User(n,p) == (
    name := n;
    password := p;

```

```

    return self;
  )
  post name = n and password = p and tickets = {} and events = {};

  -- Returns set of Strings representing the names of user's events

  public pure getEvents : () ==> set of String
  getEvents() == (
    return events;
  );

  /*
   Adds an event to user's events set of String
   pre condition checks if event isn't already in set
  */

  public addEvent : String ==> ()
  addEvent(e) == (
    events := events union {e}
  )
  pre e not in set events;

  /*
   Removes an event from user's events
   pre condition checks if event is in set
  */

  public removeEvent : String ==> ()
  removeEvent(e) == (events := events \ {e})
  pre e in set events;

  -- Returns user's name

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

  -- Returns user's funds

  public pure getFunds : () ==> nat
  getFunds() == (
    return funds;
  );

  -- Returns user's password

  public pure getPassword : () ==> String
  getPassword() == (
    return password;
  );

  public addFunds : nat ==> ()
  addFunds(f) == (
    funds := funds + f;
  )
  pre funds + f <= 10000
  post funds = funds~ + f;

  public removeFunds : nat ==> ()
  removeFunds(f) == (
    funds := funds - f;
  )

```

```

pre funds - f >= 0
post funds = funds~ - f;

public pure getTickets : () ==> set of nat
  getTickets() == (
    return tickets;
  );

public buyTicket : nat * nat ==> ()
  buyTicket(ticketPrice,t) == (
    removeFunds(ticketPrice);
    addTicket(t);
  );

public addTicket : nat ==> ()
  addTicket(ticket) == (
    tickets := tickets union {ticket};
  )
pre ticket not in set tickets
post tickets = tickets~ union {ticket};

public removeTicket : nat ==> ()
  removeTicket(i) == (
    tickets:= tickets \ {i};
  )
pre i in set tickets
post tickets = tickets~ \ {i};

functions
traces
end User

```

10 UserManager

```

class UserManager
types
-- TODO Define types here
public String = seq of char;
values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
  private current_user : String := "";
  private users : map String to User; -- id to User
operations
-- TODO Define operations here

public UserManager : map String to User ==> UserManager
  UserManager(uss) == (
    users := uss;
    return self;
  );

public login : String * String ==> bool
  login(u,p) == (

```

```

    if users(u).getPassword() = p
    then (current_user := u; return true)
    else return false
  )
pre u in set dom users;

public logout : () ==> ()
logout() == (
  current_user := ""
)
pre isLoggedIn();

public register : String * String ==> ()
register(u,p) == (
  users := users ++ {u |-> new User(u,p)};
)
pre u not in set dom users;

public addUser : User ==> ()
addUser(user) == (
  users := users ++ {user.getName() |-> user}
)
pre user.getName() not in set dom users
post users = users~ ++ {user.getName() |-> user};

public pure getUser : String ==> User
getUser(u) == (return users(u));
pre u in set dom users;

public pure getCurrentUser : () ==> String
getCurrentUser() == (return current_user);

public pure getUsers : () ==> map String to User
getUsers() == (return users;);

private pure isLoggedIn : () ==> bool
isLoggedIn() == (return current_user <> "" ););

public buyTicket : String * TicketManager * EventManager ==> ()
buyTicket(e,tm,em) == (
  dcl ticket : Ticket := new Ticket(current_user, em.getEvents() (e).getName());
  tm.addTicket(ticket);
  em.addTicket(ticket.getID(), e);
  users(current_user).buyTicket(em.getEvents() (e).getTicketPrice(), ticket.getID())
pre isLoggedIn();

public pure getUserTickets : () ==> set of nat
getUserTickets() == (return users(current_user).getTickets())
pre isLoggedIn();

public getUserTicketsEvent : String * TicketManager ==> set of nat
getUserTicketsEvent(e,tm) == (
  dcl tickets : set of nat := {};
  for all ticket in set (users(current_user).getTickets()) do
    (

```

```

    if tm.getTickets()(ticket).getEvent() = e
    then tickets := tickets union {ticket};
  );
  return tickets;
);

public promoteEvent : String*EventManager ==> ()
promoteEvent(e,em) == (
  em.getEvents()(e).promote();
  users(current_user).removeFunds(10);
)
pre isLoggedIn();

public createEvent : Event*EventManager ==> ()
createEvent(e,em) == (
  em.addEvent(e);
  users(current_user).addEvent(e.getName());
)
pre isLoggedIn();

functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end UserManager

```

11 UserTest

```

class UserTest is subclass of Data
types
-- TODO Define types here
values
-- TODO Define values here
instance variables
-- TODO Define instance variables here
e : Event := new Event("evento1", 15, 10, mk_Date(10,1,2019));
e2 : Event := new Event("evento2", 15, 10, mk_Date(29,2,2020));
e3 : Event := new Event("evento3", 15, 10, mk_Date(29,4,2020));
e4 : Event := new Event("evento4", 15, 10, mk_Date(28,2,2021));
u : User := new User("dank", "memes");
em : EventManager := new EventManager({ "evento1" |-> e, "evento2" |-> e2, "evento3" |-> e3, "
evento4" |-> e4});
um : UserManager := new UserManager({"dank" |-> u});
tm : TicketManager := new TicketManager({|->});
t : Ticket := new Ticket("dank", "evento1");
operations
-- TODO Define operations here

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

private testGetName : () ==> ()
testGetName() == (
  assertTrue(u.getName() = "dank");

```



```

);

private testAddFunds : () ==> ()
testAddFunds() == (
  u.addFunds(100);
  assertTrue(u.getFunds() = 100);
);

private testRemoveFunds : () ==> ()
testRemoveFunds() == (
  u.addFunds(100);
  u.removeFunds(10);
  assertTrue(u.getFunds() = 90);
);

private testGetPassword : () ==> ()
testGetPassword() == (
  assertTrue(u.getPassword() = "memes");
);

private testAddEvent : () ==> ()
testAddEvent() == (
  u.addEvent("eventTest1");
  u.addEvent("eventTest2");
  assertTrue(u.getEvents() = {"eventTest1","eventTest2"});
);

private testRemoveEvent : () ==> ()
testRemoveEvent() == (
  u.addEvent("eventTest1");
  u.addEvent("eventTest2");
  u.removeEvent("eventTest1");
  assertTrue(u.getEvents() = {"eventTest2"});
);

private testAddTicket : () ==> ()
testAddTicket() == (
  u.addTicket(0);
  u.addTicket(1);
  assertTrue(u.getTickets() = {0,1});
);

private testRemoveTicket : () ==> ()
testRemoveTicket() == (
  u.addTicket(0);
  u.addTicket(1);
  u.removeTicket(0);
  assertTrue(u.getTickets() = {1});
);

private testBuyTicket : () ==> ()
testBuyTicket() == (
  u.addFunds(100);
  u.buyTicket(10,0);
  assertTrue(u.getTickets() = {0});
  assertTrue(u.getFunds() = 90);
);

```

```

private testLogout : () ==> ()
testLogout() == (
    assertTrue(um.login("dank", "memes"));
    assertTrue(um.getCurrentUser() = "dank");
    um.logout();
    assertTrue(um.getCurrentUser() = "");
);

private testWrongPass : () ==> ()
testWrongPass() == (
    um.register("testUser", "testPass");
    assertTrue(not um.login("testUser", "wrongpass"));
);

private testPromote : () ==> ()
testPromote() == (
    u.addFunds(100);
    assertTrue(um.login("dank", "memes"));
    um.promoteEvent("evento3", em);
    assertTrue(e3.getPopularity() = 10);
);

private testGetUser : () ==> ()
testGetUser() == (
    assertTrue(um.getUser("dank") = u);
);

private testGetUsers : () ==> ()
testGetUsers() == (
    assertTrue(um.getUsers() ("dank") = u);
);

private testGetUserTickets : () ==> ()
testGetUserTickets() == (
    assertTrue(um.login("dank", "memes"));
    u.addFunds(100);
    u.buyTicket(10,0);
    assertTrue(um.getUserTickets() = {});
    assertTrue(u.getFunds() = 90);
);

private testGetUserTicketsEvent : () ==> ()
testGetUserTicketsEvent() == (
    assertTrue(um.login("dank", "memes"));
    u.addFunds(100);
    um.buyTicket("evento4", tm, em);
    um.buyTicket("evento4", tm, em);
    um.buyTicket("evento4", tm, em);
    um.buyTicket("evento3", tm, em);
    assertTrue(um.getUserTicketsEvent("evento4", tm) = {33, 34, 35});
);

public static main: () ==> ()
main() == (
    new UserTest().testGetName();
    new UserTest().testAddFunds();
);

```

```

new UserTest().testRemoveFunds();
new UserTest().testGetPassword();
new UserTest().testAddEvent();
new UserTest().testRemoveEvent();
new UserTest().testAddTicket();
new UserTest().testRemoveTicket();
new UserTest().testBuyTicket();
new UserTest().testLogout();
new UserTest().testWrongPass();
new UserTest().testPromote();
new UserTest().testGetUser();
new UserTest().testGetUsers();
new UserTest().testGetUserTickets();
new UserTest().testGetUserTicketsEvent();
);
functions
-- TODO Define functiones here
traces
-- TODO Define Combinatorial Test Traces here
end UserTest

```