UNA PICCOLA INTRODUZIONE A SWARM: Listings in ObjectiveC

Preparata da Marie-Edith Bissey
Dipartimento di Politiche Pubbliche e Scelte Collettive POLIS
Università del Piemonte Orientale
email: bissey@sp.unipmn.it

March 7, 2001

Contents

1 Un mercato semplice

Listing 1: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
#import <objectbase/SwarmObject.h>
#import <simtools.h>
#import <collections.h>
#import <random.h>

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
    int myName;
    int moneySpent;
    }

// define methods for the consumer
```

```
// this method passes values for the consumer variables
-setConsumerName:(int)name Budget:(int)budget;

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName;
-(int)getBudget;
@end
```

Listing 2: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"

@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-setConsumerName:(int)name Budget:(int)budget
    {
    myName=name;
    myBudget=budget;
    return self;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    return k;
    }

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
```

```
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
    }

@end
```

Listing 3: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import "Consumer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    Consumer * aConsumer;
    int modelTime;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
// this method creates the consumers
-buildObjects;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
```

```
-buildActions;
-activateIn: (id) swarmContext;

@end
```

Listing 4: Il modelSwarm: implementazione

```
// ModelSwarm.m

#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;

// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
// initialise time variable
    obj -> modelTime=0;
    return obj;
    }

-createEnd
  {
  return [super createEnd];
  }

-buildObjects
    {
    int budget=10;
    int consumerName=1;
    [super buildObjects];
    // create the consumer
    aConsumer=[Consumer create: [self getZone]];
    [aConsumer setConsumerName:consumerName Budget:budget];
    return self;
    }

-marketDay
    {
    int go;
    int spending;
    go=[aConsumer goToTheMarket];
    if (go)
        {
        spending=[aConsumer spend];

        // now, print a report of the consumer's actions
        printf("This is time %d\n",modelTime);
        printf("I am consumer %d, I went to the market and spent %d\n"
             ,[aConsumer getConsumerName],spending);
        printf("I have %d of currency left.\n",[aConsumer
```

```
            calculateRemainingBudget]);
        }
    else
        {
        // print consumer's state
        printf("This is time %d\n",modelTime);
        printf("I am consumer %d, I did not go to the market.\n",[
            aConsumer getConsumerName]);
        printf("I have %d of currency left.\n",[aConsumer getBudget]);
        }
    return self;
    }

-buildActions
    {
    // create an action group for the actions the model need to
    // perform at each time period (this is not necessary here
    // are there is only one action).
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];

    // now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

@end
```

Listing 5: Il file main

```
// main.m

#import "ModelSwarm.h"

int main(int argc, const char ** argv)
    {
    ModelSwarm * modelSwarm;

    initSwarm(argc, argv);

// create the modelSwarm
    modelSwarm =  [ModelSwarm createBegin: globalZone];
    modelSwarm=[modelSwarm createEnd];
```

```
// set the seed of the random generator so results can be reproduced
// (this is facultative)
    [randomGenerator setStateFromSeed:934850934];
// build objects and actions of modelSwarm
    [modelSwarm buildObjects];
    [modelSwarm buildActions];
    [modelSwarm activateIn: nil];
    [[modelSwarm getActivity] run];
    return 0;
    }
```

Listing 6: Il Makefile

```
APPLICATION=market
OBJECTS=main.o ModelSwarm.o Consumer.o
include $(SWARMHOME)/etc/swarm/Makefile.appl
main.o: main.m ModelSwarm.h
ModelSwarm.o: ModelSwarm.h ModelSwarm.m Consumer.h
Consumer.o: Consumer.h Consumer.m
```

2 Il mercato con collezioni di oggetti

Listing 7: La classe per il wrapper Integer: interfaccia

```
// Integer.h

#import <objectbase/SwarmObject.h>
#import <collections.h>

@interface Integer: SwarmObject
    {
    int value;
    }
-setValue: (int)val;
-(int)getMyValue;
-(int)compare:otherId;
@end
```

Listing 8: La classe per il wrapper Integer: implementazione

```
// Integer.m

#import "Integer.h"

@implementation Integer

// to set the value of the object
-setValue:(int) val
    {
    value=val;
    return self;
    }
```

```
// to retrieve the value of the object
-(int) getMyValue
    {
    return value;
    }

// to compare objects
- (int) compare: otherId
{
        if ([self getMyValue]== [otherId getMyValue])
            {
            return 0;
            }
        else if ([self getMyValue] >[otherId getMyValue])
            {
            return 1;
            }
        else
            {
            return -1;
            }
}
@end
```

Listing 9: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <collections.h>
#import <random.h>

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
    int myMaxBudget;
    int myName;
    int moneySpent;
    id <Map> mapOfSpending;
    id <Array> arrayOfVisits;
    }

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
    int)startBudget;

// methods to create the map of spending and the array of visits
```

```
-createMapOfSpending:aZone;
-createArrayOfVisits:aZone:(int)size;

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget;

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value;
-updateVisits:(int)offset:(int)value;

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName;
-(int)getBudget;
-(int)getVisitValue:(int)offset;
-(int)getSpendingValue:(int)key;
@end
```

Listing 10: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"
#import "Integer.h"

@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
    int)startBudget;
   {
   myName=name;
   myBudget=startBudget;
   myMaxBudget=maxBudget;
   }
```

```
// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone
    {
    mapOfSpending=[Map create:aZone];
    return self;
    }

-createArrayOfVisits:aZone:(int)size
    {
    // note, arrays are of fixed size, so they need the setCount method
    // when created
    arrayOfVisits=[Array create:aZone setCount:size];
    return self;
    }

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget
    {
    // take myBudget and add to it a random variable between
    // 0 and myMaxBudget (need to add as myBudget may not
    // be 0 (if not all was spent))
    myBudget+=[uniformIntRand getIntegerWithMin:0 withMax:myMaxBudget];
    return myBudget;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    return k;
    }

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }
```

```
// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value
    {
    Integer * keyObject;
    Integer * valueObject;

    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [mapOfSpending at:keyObject insert:valueObject];
    return self;
    }

-updateVisits:(int)offset:(int)value
    {
    Integer * valueObject;

    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [arrayOfVisits atOffset:offset put:valueObject];
    return self;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
    }

// get value at offset, in arrayOfVisits
-(int)getVisitValue:(int)offset
    {
    Integer * element;
    element=[arrayOfVisits atOffset:offset];
    return [element getMyValue];
    }

// get value of element at key
-(int)getSpendingValue:(int)key
    {
    Integer * keyObject;
    Integer * element;
```

```
    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    element=[mapOfSpending at:keyObject];
    return [element getMyValue];
    }

@end
```

Listing 11: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import "Consumer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    int modelTime;
    int maxTime;
    int startBudget;
    int maxBudget;
    int notFinished;
    int numberOfConsumers;
    id <List> listOfConsumers;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
// this method creates the consumers
-buildObjects;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
-increaseTime;
-checkToStop;
-buildActions;
-activateIn: (id) swarmContext;

@end
```

Listing 12: Il modelSwarm: implementazione

```
// ModelSwarm.m
```

```
#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;

// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
// initialise time variable
    obj -> modelTime=0;
    obj -> maxTime=5;
    obj -> numberOfConsumers=3;
    obj -> startBudget=0; // no endowment
    obj -> maxBudget=10;
    obj -> notFinished=1;
    return obj;
    }

-createEnd
  {
  return [super createEnd];
  }

-buildObjects
    {
    int i;
    int name;

// create the list of consumers
    listOfConsumers=[List create:[self getZone]];

// iterate over all possible consumers (from 1 to numberOfConsumers)
    for (i=1;i<=numberOfConsumers;++i)
        {
        Consumer * aConsumer;
// name of consumer=index i
        name=i;
// create the consumers
        aConsumer=[Consumer create: [self getZone]];
        [aConsumer setConsumerName:name MaxBudget:maxBudget StartBudget
             :startBudget];
// create the map and arrays (the array is created with a size of
// maxTime+1, as time starts at 0
        [aConsumer createMapOfSpending:[self getZone]];
        [aConsumer createArrayOfVisits:[self getZone]:maxTime+1];
// add consumer to the list
        [listOfConsumers addFirst:aConsumer];
        }
    return self;
    }

-marketDay
```

```
    {
    int go;
    int spending;
    int budget;
    id <Index> i=nil; // to iterate over the list of consumers
    Consumer * listElement;

// iterate over the list of consumers
// first, create the index i
    i=[listOfConsumers begin:[self getZone]];
    while ((listElement=[i next])!=nil)
        {
// update the budget of the consumer
        budget=[listElement findBudget];

// is he going to the market?
        go=[listElement goToTheMarket];
        if (go)
            {
            spending=[listElement spend];
// add 1 to arrayOfVisits, at the position corresponding to
// current modelTime
            [listElement updateVisits:modelTime:1];
// add spending at the key modelTime in the mapOfSpending
            [listElement updateSpending:modelTime:spending];
// now, print a report of the consumer's actions
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I spent %d (from map)\n",[listElement
                getSpendingValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                calculateRemainingBudget]);
            }
        else
            {
// add 0 to arrayOfVisits at modelTime
            [listElement updateVisits:modelTime:0];
// add 0 to mapOfSpending at modelTime
            [listElement updateSpending:modelTime:0];
// print consumer's state
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                getBudget]);
            }
        }// end of iteration on listOfConsumers
```

```
// it is good practice to drop unused objects like indexes when they
// are no longer needed
        [i drop];
    return self;
    }

-increaseTime
    {
// at the end of a period, modelTime need to be increased by 1
    ++modelTime;
    return self;
    }
-checkToStop
    {
// if modelTime<maxTime, then notFinished is 1, otherwhise it
// is 0
    if (modelTime<=maxTime)
        {
        notFinished=1;
        }
    else
        {
        notFinished=0;
// make sure the programs terminates and we are back to prompt
        [getTopLevelActivity() terminate];

        }
    return self;
    }


-buildActions
    {
// create an action group for the actions the model need to
// perform at each time period
// first, we tell people to go to the market (marketDay)
// then we increase the time of the model (increaseTime)
// finally we check whether this time is still valid (checkToStop)
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];
    [modelActions createActionTo:self message:M(increaseTime)];
    [modelActions createActionTo:self message:M(checkToStop)];

// now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    [modelSchedule setRepeatInterval:1];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
```

```
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

@end
```

Listing 13: Il file main

```
// main.m

#import "ModelSwarm.h"

int main(int argc, const char ** argv)
    {
    ModelSwarm * modelSwarm;

    initSwarm(argc, argv);

// create the modelSwarm
    modelSwarm =  [ModelSwarm createBegin: globalZone];
    modelSwarm=[modelSwarm createEnd];
// set the seed of the random generator so results can be reproduced
// (this is facultative)
    [randomGenerator setStateFromSeed:100000];
    [modelSwarm buildObjects];
    [modelSwarm buildActions];
    [modelSwarm activateIn: nil];
    [[modelSwarm getActivity] run];
    return 0;
    }
```

Listing 14: Il Makefile

```
APPLICATION=market
OBJECTS=main.o ModelSwarm.o Consumer.o Integer.o
include $(SWARMHOME)/etc/swarm/Makefile.appl
main.o: main.m ModelSwarm.h
ModelSwarm.o: ModelSwarm.h ModelSwarm.m Consumer.h
Consumer.o: Consumer.h Consumer.m Integer.h
Integer.o: Integer.h Integer.m
```

3 Importare parametri da files

Listing 15: La classe per il wrapper Integer: interfaccia

```
// Integer.h

#import <objectbase/SwarmObject.h>
#import <collections.h>
```

```
@interface Integer: SwarmObject
    {
    int value;
    }
-setValue: (int)val;
-(int)getMyValue;
-(int)compare:otherId;
@end
```

Listing 16: La classe per il wrapper Integer: implementazione

```
// Integer.m

#import "Integer.h"

@implementation Integer

// to set the value of the object
-setValue:(int) val
    {
    value=val;
    return self;
    }

// to retrieve the value of the object
-(int) getMyValue
    {
    return value;
    }

// to compare objects
- (int) compare: otherId
{
        if ([self getMyValue]== [otherId getMyValue])
            {
            return 0;
            }
        else if ([self getMyValue] >[otherId getMyValue])
            {
            return 1;
            }
        else
            {
            return -1;
            }
}
@end
```

Listing 17: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
```

```
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <collections.h>
#import <random.h>

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
    int myMaxBudget;
    int myName;
    int moneySpent;
    id <Map> mapOfSpending;
    id <Array> arrayOfVisits;
    }

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
     int)startBudget;

// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone;
-createArrayOfVisits:aZone:(int)size;

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget;

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value;
-updateVisits:(int)offset:(int)value;

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName;
```

```
-(int)getBudget;
-(int)getVisitValue:(int)offset;
-(int)getSpendingValue:(int)key;
@end
```

Listing 18: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"
#import "Integer.h"

@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
    int)startBudget;
    {
    myName=name;
    myBudget=startBudget;
    myMaxBudget=maxBudget;
    }

// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone
    {
    mapOfSpending=[Map create:aZone];
    return self;
    }

-createArrayOfVisits:aZone:(int)size
    {
    // note, arrays are of fixed size, so they need the setCount method
    // when created
    arrayOfVisits=[Array create:aZone setCount:size];
    return self;
    }

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget
    {
    // take myBudget and add to it a random variable between
    // 0 and myMaxBudget (need to add as myBudget may not
    // be 0 (if not all was spent))
    myBudget+=[uniformIntRand getIntegerWithMin:0 withMax:myMaxBudget];
    return myBudget;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
```

```
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    return k;
    }

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value
    {
    Integer * keyObject;
    Integer * valueObject;

    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [mapOfSpending at:keyObject insert:valueObject];
    return self;
    }

-updateVisits:(int)offset:(int)value
    {
    Integer * valueObject;

    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [arrayOfVisits atOffset:offset put:valueObject];
    return self;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
```

```
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
    }

// get value at offset, in arrayOfVisits
-(int)getVisitValue:(int)offset
    {
    Integer * element;
    element=[arrayOfVisits atOffset:offset];
    return [element getMyValue];
    }

// get value of element at key
-(int)getSpendingValue:(int)key
    {
    Integer * keyObject;
    Integer * element;
    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    element=[mapOfSpending at:keyObject];
    return [element getMyValue];
    }

@end
```

Listing 19: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import "Consumer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    int modelTime;
    int maxTime;
    int startBudget;
    int maxBudget;
    int notFinished;
```

```
    int numberOfConsumers;
    id <List> listOfConsumers;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
// this method creates the consumers
-buildObjects;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
-increaseTime;
-checkToStop;
-buildActions;
-activateIn: (id) swarmContext;

@end
```

Listing 20: Il modelSwarm: implementazione

```
// ModelSwarm.m

#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;

// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
    return obj;
    }

-createEnd
  {
  return [super createEnd];
  }

-buildObjects
    {
    int i;
    int name;

// create the list of consumers
    listOfConsumers=[List create:[self getZone]];

// iterate over all possible consumers (from 1 to numberOfConsumers)
    for (i=1;i<=numberOfConsumers;++i)
        {
        Consumer * aConsumer;
// name of consumer=index i
```

```
        name=i;
// create the consumers
        aConsumer=[Consumer create:[self getZone]];
        [aConsumer setConsumerName:name MaxBudget:maxBudget StartBudget
             :startBudget];
// create the map and arrays (the array is created with a size of
// maxTime+1, as time starts at 0
        [aConsumer createMapOfSpending:[self getZone]];
        [aConsumer createArrayOfVisits:[self getZone]:maxTime+1];
// add consumer to the list
        [listOfConsumers addFirst:aConsumer];
        }
    return self;
    }

-marketDay
    {
    int go;
    int spending;
    int budget;
    id <Index> i=nil; // to iterate over the list of consumers
    Consumer * listElement;

// iterate over the list of consumers
// first, create the index i
    i=[listOfConsumers begin:[self getZone]];
    while ((listElement=[i next])!=nil)
        {
// update the budget of the consumer
        budget=[listElement findBudget];

// is he going to the market?
        go=[listElement goToTheMarket];
        if (go)
            {
            spending=[listElement spend];
// add 1 to arrayOfVisits, at the position corresponding to
// current modelTime
            [listElement updateVisits:modelTime:1];
// add spending at the key modelTime in the mapOfSpending
            [listElement updateSpending:modelTime:spending];
// now, print a report of the consumer's actions
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I spent %d (from map)\n",[listElement
                getSpendingValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                calculateRemainingBudget]);
            }
        else
```

```
            {
// add 0 to arrayOfVisits at modelTime
            [listElement updateVisits:modelTime:0];
// add 0 to mapOfSpending at modelTime
            [listElement updateSpending:modelTime:0];
// print consumer's state
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                getBudget]);
            }
        }// end of iteration on listOfConsumers
// it is good practice to drop unused objects like indexes when they
// are no longer needed
        [i drop];
    return self;
    }

-increaseTime
    {
// at the end of a period, modelTime need to be increased by 1
    ++modelTime;
    return self;
    }
-checkToStop
    {
// if modelTime<maxTime, then notFinished is 1, otherwhise it
// is 0
    if (modelTime<=maxTime)
        {
        notFinished=1;
        }
    else
        {
        notFinished=0;
// make sure the programs terminates and we are back to prompt
        [getTopLevelActivity() terminate];

        }
    return self;
    }


-buildActions
    {
// create an action group for the actions the model need to
// perform at each time period
// first, we tell people to go to the market (marketDay)
// then we increase the time of the model (increaseTime)
// finally we check whether this time is still valid (checkToStop)
```

```
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];
    [modelActions createActionTo:self message:M(increaseTime)];
    [modelActions createActionTo:self message:M(checkToStop)];

// now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    [modelSchedule setRepeatInterval:1];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

@end
```

Listing 21: Il file main

```
// main.m

#import "ModelSwarm.h"

int main(int argc, const char ** argv)
    {
    ModelSwarm * modelSwarm;
    id <LispArchiver> archiver;

    initSwarm(argc, argv);

// create the modelSwarm
// load the parameters from file: first create an archiver instance,
// then load the parameters, with a check that the file exists, or that
// the key (modelSwarm) is in the file. The LispArchiver also uses
// the createBegin and createEnd methods of the modelSwarm
    archiver=[LispArchiver create:globalZone setPath:"parameters.scm"];
    if((modelSwarm=[archiver getWithZone:globalZone key:"modelSwarm"])
       ==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

// set the seed of the random generator so results can be reproduced
// (this is facultative)
    [randomGenerator setStateFromSeed:100000];
    [modelSwarm buildObjects];
```

```
    [modelSwarm buildActions];
    [modelSwarm activateIn: nil];
    [[modelSwarm getActivity] run];
    return 0;
    }
```

Listing 22: Il Makefile

```
APPLICATION=market
OBJECTS=main.o ModelSwarm.o Consumer.o Integer.o
include $(SWARMHOME)/etc/swarm/Makefile.appl
main.o: main.m ModelSwarm.h
ModelSwarm.o: ModelSwarm.h ModelSwarm.m Consumer.h
Consumer.o: Consumer.h Consumer.m Integer.h
Integer.o: Integer.h Integer.m
```

Listing 23: Il file parameters.scm

```
(list
    (cons 'modelSwarm
        (make-instance 'ModelSwarm
                    #:modelTime 0
                    #:maxTime 5
                    #:numberOfConsumers 3
                    #:startBudget 0 ;no endowment
                    #:maxBudget 10
                    #:notFinished 1 ))
)
```

4 L'interfaccia grafica

Non facciamo vedere i files della classe Integer, che non sono stati cambiati.

Listing 24: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <collections.h>
#import <random.h>

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
    int myMaxBudget;
    int myName;
    int moneySpent;
    id <Map> mapOfSpending;
```

```
    id <Array> arrayOfVisits;
    int currentTime;
    }

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
     int)startBudget;

// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone;
-createArrayOfVisits:aZone:(int)size;

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget;

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value;
-updateVisits:(int)offset:(int)value;

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName;
-(int)getBudget;
-(int)getVisitValue:(int)offset;
-(int)getSpendingValue:(int)key;
-(int)getVisit;
-(int)getSpending;
@end
```

Listing 25: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"
#import "Integer.h"
```

```
@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-(void)setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(
     int)startBudget;
    {
    myName=name;
    myBudget=startBudget;
    myMaxBudget=maxBudget;
    }

// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone
    {
    mapOfSpending=[Map create:aZone];
    return self;
    }

-createArrayOfVisits:aZone:(int)size
    {
    // note, arrays are of fixed size, so they need the setCount method
    // when created
    arrayOfVisits=[Array create:aZone setCount:size];
    return self;
    }

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget
    {
    // take myBudget and add to it a random variable between
    // 0 and myMaxBudget (need to add as myBudget may not
    // be 0 (if not all was spent))
    myBudget+=[uniformIntRand getIntegerWithMin:0 withMax:myMaxBudget];
    return myBudget;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    return k;
    }

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
```

```
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value
    {
    Integer * keyObject;
    Integer * valueObject;

    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [mapOfSpending at:keyObject insert:valueObject];
    return self;
    }

-updateVisits:(int)offset:(int)value
    {
    Integer * valueObject;
// set the currentTime (equivalent to modelTime)
    currentTime=offset;
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [arrayOfVisits atOffset:offset put:valueObject];
    return self;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
    }
```

```
// get value at offset, in arrayOfVisits
-(int)getVisitValue:(int)offset
    {
    Integer * element;
    element=[arrayOfVisits atOffset:offset];
    return [element getMyValue];
    }

// get value of element at key
-(int)getSpendingValue:(int)key
    {
    Integer * keyObject;
    Integer * element;
    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    element=[mapOfSpending at:keyObject];
    return [element getMyValue];
    }

-(int)getVisit
    {
    return [self getVisitValue:currentTime];
    }

-(int)getSpending
    {
    return [self getSpendingValue:currentTime];
    }
@end
```

Listing 26: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import "Consumer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    int modelTime;
    int maxTime;
    int startBudget;
    int maxBudget;
    int notFinished;
    int numberOfConsumers;
```

```
    id <List> listOfConsumers;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
// this method creates the consumers
-buildObjects;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
-increaseTime;
-(int)checkToStop;
-buildActions;
-activateIn: (id) swarmContext;
// these methods are used to get informations for characteristics
// of the model
-getListOfConsumers;
-getConsumer:(int)name;
-(int)getCurrentTime;
@end
```

Listing 27: Il modelSwarm: implementazione

```
// ModelSwarm.m

#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;
    id <ProbeMap> modelProbeMap;
// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
// create the probemap for the model
    modelProbeMap=[EmptyProbeMap createBegin:aZone];
    [modelProbeMap setProbedClass:[self class]];
    modelProbeMap=[modelProbeMap createEnd];

    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"maxTime"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        numberOfConsumers"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        startBudget"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        maxBudget"
                inClass:[self class]]];
    [probeLibrary setProbeMap:modelProbeMap For:[self class]];
    return obj;
```

```
    }

-createEnd
  {
  return [super createEnd];
  }

-buildObjects
    {
    int i;
    int name;
// create the list of consumers
    listOfConsumers=[List create:[self getZone]];

// iterate over all possible consumers (from 1 to numberOfConsumers)
    for (i=1;i<=numberOfConsumers;++i)
        {
        Consumer * aConsumer;
// name of consumer=index i
        name=i;
// create the consumers
        aConsumer=[Consumer create: [self getZone]];
        [aConsumer setConsumerName:name MaxBudget:maxBudget StartBudget
             :startBudget];
// create the map and arrays (the array is created with a size of
// maxTime+1, as time starts at 0
        [aConsumer createMapOfSpending:[self getZone]];
        [aConsumer createArrayOfVisits:[self getZone]:maxTime+1];
// add consumer to the list
xprintid(aConsumer);
        [listOfConsumers addFirst:aConsumer];
        }
    return self;
    }

-marketDay
    {
    int go;
    int spending;
    int budget;
    id <Index> i=nil; // to iterate over the list of consumers
    Consumer * listElement;

// iterate over the list of consumers
// first, create the index i
    i=[listOfConsumers begin:[self getZone]];
    while ((listElement=[i next])!=nil)
        {
// update the budget of the consumer
        budget=[listElement findBudget];

// is he going to the market?
        go=[listElement goToTheMarket];
        if (go)
```

```
            {
            spending=[listElement spend];
// add 1 to arrayOfVisits, at the position corresponding to
// current modelTime
            [listElement updateVisits:modelTime:1];
// add spending at the key modelTime in the mapOfSpending
            [listElement updateSpending:modelTime:spending];
// now, print a report of the consumer's actions
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I spent %d (from map)\n",[listElement
                getSpendingValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                calculateRemainingBudget]);
            }
        else
            {
// add 0 to arrayOfVisits at modelTime
            [listElement updateVisits:modelTime:0];
// add 0 to mapOfSpending at modelTime
            [listElement updateSpending:modelTime:0];
// print consumer's state
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                ;
            printf("Did I go to the market? %d (from array)\n",[
                listElement getVisitValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                getBudget]);
            }
        }// end of iteration on listOfConsumers
// it is good practice to drop unused objects like indexes when they
// are no longer needed
        [i drop];
    return self;
    }

-increaseTime
    {
// at the end of a period, modelTime need to be increased by 1
    ++modelTime;
    return self;
    }
-(int)checkToStop
    {
// if modelTime<maxTime, then notFinished is 1, otherwhise it
// is 0 (return respectively 0 or 1 for the observer)
    if (modelTime<=maxTime)
        {
```

```
        notFinished=1;
        return 0;
        }
    else
        {
        notFinished=0;
        return 1;
        }
    }


-buildActions
    {
// create an action group for the actions the model need to
// perform at each time period
// first, we tell people to go to the market (marketDay)
// then we increase the time of the model (increaseTime)
// finally we check whether this time is still valid (checkToStop)
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];
    [modelActions createActionTo:self message:M(increaseTime)];
    [modelActions createActionTo:self message:M(checkToStop)];

// now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    [modelSchedule setRepeatInterval:2];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

// methods to pass parameters to other classes
-getListOfConsumers
    {
    return listOfConsumers;
    }

-getConsumer:(int)name
    {
// note that elements are entered in the list using the addFirst
// method, so the first element is the latest agent created.
    return [listOfConsumers atOffset:[listOfConsumers getCount]-name];
    }

-(int)getCurrentTime
```

```
    {
    return modelTime;
    }
@end
```

Listing 28: L'ObserverSwarm: interfaccia

```
// ObserverSwarm.h

#import <objectbase.h>
#import <activity.h>
#import <collections.h>
#import <simtools.h>
#import <simtoolsgui.h>
#import <simtoolsgui/GUISwarm.h>
#import <analysis.h>
#import "ModelSwarm.h"

@interface ObserverSwarm : GUISwarm
    {
    int displayFrequency;
    int displayConsumerName;

    id displayActions;
    id displaySchedule;

    ModelSwarm * modelSwarm;

    id <EZGraph> spendingGraph;
    id <EZGraph> consumerGraph;
    }

+createBegin: (id) aZone;
-createEnd;
-buildObjects;
-buildActions;
-activateIn: (id) swarmContext;
-observerCheckToStop;
@end
```

Listing 29: L'ObserverSwarm: implementazione

```
// ObserverSwarm.m

#import "ObserverSwarm.h"

@implementation ObserverSwarm

+createBegin: (id) aZone
    {
    ObserverSwarm * obj;
    id <ProbeMap> probeMap;

    obj = [super createBegin: aZone];
```

```
// probe map for observer
    probeMap=[EmptyProbeMap createBegin:aZone];
    [probeMap setProbedClass:[self class]];
    probeMap=[probeMap createEnd];

    [probeMap addProbe:[probeLibrary getProbeForVariable:"
         displayFrequency"
                                    inClass:[self class]]];
    [probeMap addProbe:[probeLibrary getProbeForVariable:"
         displayConsumerName"
                                    inClass:[self class]]];

    [probeLibrary setProbeMap: probeMap For:[self class]];

    return obj;
    }

-createEnd
    {
    return [super createEnd];
    }

-buildObjects
    {
    id <LispArchiver> archiver;

    [super buildObjects];

// create the modelSwarm, reading parameters from a file
    archiver=[LispArchiver create:[self getZone] setPath:"parameters.
         scm"];
    if((modelSwarm=[archiver getWithZone:[self getZone] key:"modelSwarm
         "])==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

// create probes for modelSwarm and observerSwarm
    CREATE_ARCHIVED_PROBE_DISPLAY (modelSwarm);
    CREATE_ARCHIVED_PROBE_DISPLAY (self);

// set control panel to state stopped
    [controlPanel setStateStopped];

// Then we ask the model to build itself.
    [modelSwarm buildObjects];

// create graphics
// the spendingGraph shows the averag, total, min, max spending of
// consumers during the game
    spendingGraph = [EZGraph createBegin: self];
    SET_WINDOW_GEOMETRY_RECORD_NAME(spendingGraph);
```

```
    [spendingGraph setTitle: "Agents' spending"];
    [spendingGraph setAxisLabelsX: "Time" Y: "Spending"];
    spendingGraph = [spendingGraph createEnd];
    [spendingGraph createAverageSequence: "Average Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];
    [spendingGraph createTotalSequence: "Total Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];
    [spendingGraph createMinSequence: "Minimum Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];

// the consumerGraph shows the spending and the number of times
// consumer 1 went to the market
    consumerGraph=[EZGraph createBegin:self];
    SET_WINDOW_GEOMETRY_RECORD_NAME(consumerGraph);
    [consumerGraph setTitle:"A consumer"];
    [consumerGraph setAxisLabelsX: "Time" Y: "Visits/Spending"];
    consumerGraph=[consumerGraph createEnd];
    [consumerGraph createSequence: "Went to the market"
        withFeedFrom:[modelSwarm getConsumer:displayConsumerName]
            andSelector:M(getVisit)];
    [consumerGraph createSequence:"Spent"
        withFeedFrom:[modelSwarm getConsumer:displayConsumerName]
            andSelector:M(getSpending)];
    return self;
    }

-buildActions
    {
    [super buildActions];
    [modelSwarm buildActions];

// define displayActions
    displayActions = [ActionGroup create: self];
// update the graphs
    [displayActions createActionTo: spendingGraph message:M(step)];
    [displayActions createActionTo: consumerGraph message:M(step)];
    [displayActions createActionTo: self message:M(observerCheckToStop)
        ];
    [displayActions createActionTo: actionCache message: M(doTkEvents)
        ];
// define displaySchedule
    displaySchedule = [Schedule createBegin: self];
    [displaySchedule setRepeatInterval: 2];
    displaySchedule = [displaySchedule createEnd];
    [displaySchedule at: 1 createAction: displayActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
```

```
    [displaySchedule activateIn: self];
    [modelSwarm activateIn: self];
    return [self getSwarmActivity];
    }

-observerCheckToStop
    {
    if ([modelSwarm checkToStop]==1)
        {
        [controlPanel setStateStopped];
        }
    return self;
    }
@end
```

Listing 30: Il file main

```
// main.m

#import "ObserverSwarm.h"

int main(int argc, const char ** argv)
    {
    ObserverSwarm * observerSwarm;
    id <LispArchiver> archiver;

    initSwarm(argc, argv);

// create the observerSwarm, reading parameters from a file
    archiver=[LispArchiver create:globalZone setPath:"parameters.scm"];
    if((observerSwarm=[archiver getWithZone:globalZone key:"
        observerSwarm"])==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

    SET_WINDOW_GEOMETRY_RECORD_NAME(observerSwarm);
    [observerSwarm buildObjects];
    [observerSwarm buildActions];
    [observerSwarm activateIn: nil];
    [observerSwarm go];

    return 0;
    }
```

Listing 31: Il Makefile

```
APPLICATION=market
OBJECTS=main.o ModelSwarm.o Consumer.o Integer.o
include $(SWARMHOME)/etc/swarm/Makefile.appl
main.o: main.m ModelSwarm.h
ModelSwarm.o: ModelSwarm.h ModelSwarm.m Consumer.h
Consumer.o: Consumer.h Consumer.m Integer.h
```

```
Integer.o: Integer.h Integer.m
```

5 Rappresentare agenti nello spazio

Listing 32: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
//#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <collections.h>
#import <random.h>
#import "Integer.h"

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
    int myMaxBudget;
    int myName;
    int moneySpent;
    int marketGoer;
    id <Map> mapOfSpending;
    id <Array> arrayOfVisits;
    int currentTime;
    int positionX,positionY;
    }

// define methods for the consumer
// this method passes values for the consumer variables
-setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(int)
     startBudget Goer:(int)goer;

// these methods deal with the positioning of the consumer
// on the space
-setPositionX:(int)x Y:(int)y;
// draw itself on the raster
-drawSelfOn:(id <ZoomRaster>)raster;
// make the color of the agent depends on whether he is in the market
     or not
-(int)getStrategyColor;

// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone;
-createArrayOfVisits:aZone:(int)size;

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget;
```

```
// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value;
-updateVisits:(int)offset:(int)value;

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName;
-(int)getBudget;
-(int)getVisitValue:(int)offset;
-(int)getSpendingValue:(int)key;
-(int)getVisit;
-(int)getSpending;
-(int) getPositionX;
-(int) getPositionY;
@end
```

Listing 33: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"

@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(int)
      startBudget Goer:(int)goer;
    {
    myName=name;
    myBudget=startBudget;
    myMaxBudget=maxBudget;
    marketGoer=goer;
    return self;
    }

// these methods deal with the positioning of the consumer
// on the space
```

```
-setPositionX:(int)x Y:(int)y
    {
    positionX=x;
    positionY=y;
    return self;
    }

// draw itself on the raster
-drawSelfOn:(id <ZoomRaster>)raster
    {
    [raster drawPointX:positionX Y:positionY Color:[self
         getStrategyColor]];
    return self;
    }

// make the color of the agent depends on whether he is in the market
// or not. getStrategyColor links a color to the type of player and
// the strategy played. The output is an integer corresponding to the
// index of the color in the color map
-(int) getStrategyColor
    {
    if (marketGoer==0)
        {
        return 3;
        }
    else if (marketGoer==1)
        {
        return 1;
        }
    else
        {
        printf("wrong marketGoer value\n");
        exit(0);
        }
    }


// methods to create the map of spending and the array of visits
-createMapOfSpending:aZone
    {
    mapOfSpending=[Map create:aZone];
    return self;
    }

-createArrayOfVisits:aZone:(int)size
    {
    // note, arrays are of fixed size, so they need the setCount method
    // when created
    arrayOfVisits=[Array create:aZone setCount:size];
    return self;
    }

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
```

```
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget
    {
    // take myBudget and add to it a random variable between
    // 0 and myMaxBudget (need to add as myBudget may not
    // be 0 (if not all was spent))
    myBudget+=[uniformIntRand getIntegerWithMin:0 withMax:myMaxBudget];
    return myBudget;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    marketGoer=k;
    return k;
    }

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }

// these methods are used to add elements to the map of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)key:(int)value
    {
    Integer * keyObject;
    Integer * valueObject;

    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [mapOfSpending at:keyObject insert:valueObject];
    return self;
    }
```

```
-updateVisits:(int)offset:(int)value
    {
    Integer * valueObject;
// set the currentTime (equivalent to modelTime)
    currentTime=offset;
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [arrayOfVisits atOffset:offset put:valueObject];
    return self;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
    }

// get value at offset, in arrayOfVisits
-(int)getVisitValue:(int)offset
    {
    Integer * element;
    element=[arrayOfVisits atOffset:offset];
    return [element getMyValue];
    }

// get value of element at key
-(int)getSpendingValue:(int)key
    {
    Integer * keyObject;
    Integer * element;
    keyObject=[Integer create:[self getZone]];
    [keyObject setValue:key];
    element=[mapOfSpending at:keyObject];
    return [element getMyValue];
    }

-(int)getVisit
    {
    return [self getVisitValue:currentTime];
    }

-(int)getSpending
    {
    return [self getSpendingValue:currentTime];
    }
-(int) getPositionX
```

```
    {
    return positionX;
    }

-(int) getPositionY
    {
    return positionY;
    }
@end
```

Listing 34: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import <space.h>
#import <space/Discrete2d.h>
#import <space/Grid2d.h>
#import "Consumer.h"
#import "Integer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    int modelTime;
    int maxTime;
    int startBudget;
    int maxBudget;
    int notFinished;
    int numberOfConsumers;
    id <List> listOfConsumers;
    int worldWidth,worldHeight;
    int sizeOfMarket;
    int xMin,yMin,xMax,yMax;
    id <Grid2d> world;
    id <Discrete2d> market;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
// this method creates the consumers
-buildObjects;
// this method positions the consumers in the world, excluding the
     market,
// or only in the market according to the value of the last argument
```

```
-findPositionInWorld:(id <Grid2d>)aWorld For:(Consumer *)aGuy
      ExcludeMarket:(int)exclude;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
-increaseTime;
-(int)checkToStop;
-buildActions;
-activateIn: (id) swarmContext;
// these methods are used to get informations for characteristics
// of the model
-getListOfConsumers;
-getConsumer:(int)name;
-(int)getCurrentTime;
-(int)getWorldWidth;
-(int)getWorldHeight;
-getWorld;
-getMarket;
@end
```

Listing 35: Il modelSwarm: implementazione

```
// ModelSwarm.m

#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;
    id <ProbeMap> modelProbeMap;
// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
// create the probemap for the model
    modelProbeMap=[EmptyProbeMap createBegin:aZone];
    [modelProbeMap setProbedClass:[self class]];
    modelProbeMap=[modelProbeMap createEnd];

    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"maxTime"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
         numberOfConsumers"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
         startBudget"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
         maxBudget"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
         worldWidth"
                inClass:[self class]]];
```

```
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
         worldHeight"
               inClass:[self class]]];
    [probeLibrary setProbeMap:modelProbeMap For:[self class]];
    return obj;
    }

-createEnd
  {
  return [super createEnd];
  }

-buildObjects
    {
    int i,x,y;
    int name;
    id <ProbeMap> consumerProbe;

    [super buildObjects];

// set the position of the market in the center of the world
    xMin=(worldWidth-sizeOfMarket)/2;
    xMax=xMin+sizeOfMarket;
    yMin=(worldHeight-sizeOfMarket)/2;
    yMax=yMin+sizeOfMarket;

// initialise the world as a grid2d and fill it with "nil" objects
    world=[Grid2d create:[self getZone] setSizeX:worldWidth Y:
         worldHeight];
    [world fillWithObject:nil];
// initialise market as a Discrete2d and fill it with a value object
// to get the yellow square representing the market
    market=[Discrete2d create:[self getZone] setSizeX:worldWidth Y:
         worldHeight];
    for (y=yMin; y<yMax; y++)
        {
        for (x=xMin; x<xMax; x++)
            {
            [market putValue:2 atX: x Y: y];
            }
        }

// create the list of consumers
    listOfConsumers=[List create:[self getZone]];

// iterate over all possible consumers (from 1 to numberOfConsumers)
    for (i=1;i<=numberOfConsumers;++i)
        {
        Consumer * aConsumer;
// name of consumer=index i
        name=i;
// create the consumers
        aConsumer=[Consumer create: [self getZone]];
```

```
        [aConsumer setConsumerName:name MaxBudget:maxBudget StartBudget
             :startBudget Goer:0];
// set their positions to -999
        [aConsumer setPositionX:-999 Y:-999];
// create the map and arrays (the array is created with a size of
// maxTime+1, as time starts at 0
        [aConsumer createMapOfSpending:[self getZone]];
        [aConsumer createArrayOfVisits:[self getZone]:maxTime+1];
// position the consumers in the world, excluding market
        [self findPositionInWorld:world For:aConsumer ExcludeMarket:1];
// probe for consumer
        consumerProbe=[EmptyProbeMap createBegin:[self getZone]];
        [consumerProbe setProbedClass:[Consumer class]];
        consumerProbe=[consumerProbe createEnd];
        [consumerProbe addProbe:[probeLibrary getProbeForVariable:"
             myName"
                inClass:[Consumer class]]];
        [probeLibrary setProbeMap:consumerProbe For:[Consumer class]];
// add consumer to the list
        [listOfConsumers addFirst:aConsumer];
        }
    return self;
    }

// this method positions the consumers in the world, excluding the
     market,
// or only in the market according to the value of the last argument
-findPositionInWorld:(id <Grid2d>)aWorld For:(Consumer *)aGuy
     ExcludeMarket:(int)exclude
    {
    int trialX,trialY;
// set trialX and trialY to negative values (not in the world) to start
      the
// while loop
    trialX=-999;
    trialY=-999;
// if the consumers have already been put on a space (positionX and
     positionY are not
// -999, then put a nil object at their current position
    if (([aGuy getPositionX]>=0)&&([aGuy getPositionY]>=0))
        {
        [aWorld putObject:nil atX:[aGuy getPositionX] Y:[aGuy
             getPositionY]];
        }
// put consumers in the world randomly. if exclude=1, then the part of
// the world corresponding to the market is excluded from the
// possibilities, if exclude=0, the consumer is put in the market
// when choosing a value for the position of the consumer, we need
// to check that: (1) both coordinates are positive, (2) they are
// in the part of the world where they should be, (3) there is not
// already a player in the spot. This explain why the while condition
// is quite complex
    if (exclude==1)
        {
```

```
        while( ((trialX<0)&&(trialY<0)) ||
            ((trialX>=xMin)&&(trialX<=xMax)&&(trialY>=yMin)&&(trialY<=
                yMax)) ||
            ([world getObjectAtX:trialX Y:trialY]!=nil) )
            {
            trialX=[uniformIntRand getIntegerWithMin:0 withMax:
                worldWidth-1];
            trialY=[uniformIntRand getIntegerWithMin:0 withMax:
                worldHeight-1];
            }
        }
    else if (exclude==0)
        {
        while ( ((trialX<0)&&(trialY<0)) ||
            ([world getObjectAtX:trialX Y:trialY]!=nil) )
            {
            trialX=[uniformIntRand getIntegerWithMin:xMin withMax:xMax
                ];
            trialY=[uniformIntRand getIntegerWithMin:yMin withMax:yMax
                ];
            }
        }
    else
        {
        printf("wrong value for exclude!\n");
        exit(0);
        }
    [aGuy setPositionX:trialX Y:trialY];
    [aWorld putObject:aGuy atX:trialX Y:trialY];
    return self;
    }


-marketDay
    {
    int go;
    int spending;
    int budget;
    id <Index> i=nil; // to iterate over the list of consumers
    Consumer * listElement;

// iterate over the list of consumers
// first, create the index i
    i=[listOfConsumers begin:[self getZone]];
    while ((listElement=[i next])!=nil)
        {
// update the budget of the consumer
        budget=[listElement findBudget];

// is he going to the market?
        go=[listElement goToTheMarket];
        if (go)
            {
            spending=[listElement spend];
```

```
// add 1 to arrayOfVisits, at the position corresponding to
// current modelTime
            [listElement updateVisits:modelTime:1];
// add spending at the key modelTime in the mapOfSpending
            [listElement updateSpending:modelTime:spending];
// now, print a report of the consumer's actions
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                 ;
            printf("Did I go to the market? %d (from array)\n",[
                 listElement getVisitValue:modelTime]);
            printf("I spent %d (from map)\n",[listElement
                 getSpendingValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                 calculateRemainingBudget]);
// put in position in world on market space
            [self findPositionInWorld:world For:listElement
                 ExcludeMarket:0];
            }
        else
            {
// add 0 to arrayOfVisits at modelTime
            [listElement updateVisits:modelTime:0];
// add 0 to mapOfSpending at modelTime
            [listElement updateSpending:modelTime:0];
// print consumer's state
            printf("This is time %d\n",modelTime);
            printf("I am consumer %d\n",[listElement getConsumerName]);
            printf("My current budget is %d\n",[listElement getBudget])
                 ;
            printf("Did I go to the market? %d (from array)\n",[
                 listElement getVisitValue:modelTime]);
            printf("I have %d of currency left.\n",[listElement
                 getBudget]);
// put in position in world outside market
            [self findPositionInWorld:world For:listElement
                 ExcludeMarket:1];
            }
        }// end of iteration on listOfConsumers
// it is good practice to drop unused objects like indexes when they
// are no longer needed
        [i drop];
    return self;
    }

-increaseTime
    {
// at the end of a period, modelTime need to be increased by 1
    ++modelTime;
    return self;
    }
-(int)checkToStop
    {
```

```
// if modelTime<maxTime, then notFinished is 1, otherwhise it
// is 0 (return respectively 0 or 1 for the observer)
    if (modelTime<=maxTime)
        {
        notFinished=1;
        return 0;
        }
    else
        {
        notFinished=0;
        return 1;
        }
    }


-buildActions
    {
// create an action group for the actions the model need to
// perform at each time period
// first, we tell people to go to the market (marketDay)
// then we increase the time of the model (increaseTime)
// finally we check whether this time is still valid (checkToStop)
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];
    [modelActions createActionTo:self message:M(increaseTime)];
    [modelActions createActionTo:self message:M(checkToStop)];

// now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    [modelSchedule setRepeatInterval:2];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

// methods to pass parameters to other classes
-getListOfConsumers
    {
    return listOfConsumers;
    }

-getConsumer:(int)name
    {
// note that elements are entered in the list using the addFirst method
    ,
```

```
// so the first element is the latest agent created.
    return [listOfConsumers atOffset:[listOfConsumers getCount]-name];
    }

-(int)getCurrentTime
    {
    return modelTime;
    }
-(int)getWorldWidth
    {
    return worldWidth;
    }
-(int)getWorldHeight
    {
    return worldHeight;
    }
-getWorld
    {
    return world;
    }
-getMarket
    {
    return market;
    }
@end
```

Listing 36: L'ObserverSwarm: interfaccia

```
// ObserverSwarm.h

#import <objectbase.h>
#import <activity.h>
#import <collections.h>
#import <simtools.h>
#import <simtoolsgui.h>
#import <simtoolsgui/GUISwarm.h>
#import <analysis.h>
#import <space.h>
#import <space/Object2dDisplay.h>
#import "ModelSwarm.h"
#import "Consumer.h"

@interface ObserverSwarm : GUISwarm
    {
    int displayFrequency;
    int displayConsumerName;

    id displayActions;
    id displaySchedule;

    ModelSwarm * modelSwarm;

    id <EZGraph> spendingGraph;
    id <EZGraph> consumerGraph;
```

```
    id <Colormap> colorMap;
    id <ZoomRaster> worldRaster;
    id <Object2dDisplay> worldDisplay;
    id <Value2dDisplay> marketDisplay;
    int zoomFactor;
    }

+createBegin: (id) aZone;
-createEnd;
-buildObjects;
-buildActions;
-activateIn: (id) swarmContext;
-observerCheckToStop;
@end
```

Listing 37: L'ObserverSwarm: implementazione

```
// ObserverSwarm.m

#import "ObserverSwarm.h"

@implementation ObserverSwarm

+createBegin: (id) aZone
    {
    ObserverSwarm * obj;
    id <ProbeMap> probeMap;

    obj = [super createBegin: aZone];

// probe map for observer
    probeMap=[EmptyProbeMap createBegin:aZone];
    [probeMap setProbedClass:[self class]];
    probeMap=[probeMap createEnd];

    [probeMap addProbe:[probeLibrary getProbeForVariable:"
        displayFrequency"
                                    inClass:[self class]]];
    [probeMap addProbe:[probeLibrary getProbeForVariable:"
        displayConsumerName"
                                    inClass:[self class]]];

    [probeLibrary setProbeMap: probeMap For:[self class]];

    return obj;
    }

-createEnd
    {
    return [super createEnd];
    }

-buildObjects
```

```
    {
    id <LispArchiver> archiver;

    [super buildObjects];

// create the modelSwarm, reading parameters from a file
    archiver=[LispArchiver create:self setPath:"parameters.scm"];
    if((modelSwarm=[archiver getWithZone:self key:"modelSwarm"])==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

// create probes for modelSwarm and observerSwarm
    CREATE_ARCHIVED_PROBE_DISPLAY (modelSwarm);
    CREATE_ARCHIVED_PROBE_DISPLAY (self);

// set control panel to state stopped
    [controlPanel setStateStopped];

// Then we ask the model to build itself.
    [modelSwarm buildObjects];

// create graphics
// the spendingGraph shows the averag, total, min, max spending of
// consumers during the game
    spendingGraph = [EZGraph createBegin: self];
    SET_WINDOW_GEOMETRY_RECORD_NAME(spendingGraph);
    [spendingGraph setTitle: "Agents' spending"];
    [spendingGraph setAxisLabelsX: "Timex2" Y: "Spending"];
    spendingGraph = [spendingGraph createEnd];
    [spendingGraph createAverageSequence: "Average Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];
    [spendingGraph createTotalSequence: "Total Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];
    [spendingGraph createMinSequence: "Minimum Spending"
        withFeedFrom: [modelSwarm getListOfConsumers] andSelector:M(
            getSpending)];

// the consumerGraph shows the spending and the number of times
// consumer 1 went to the market
    consumerGraph=[EZGraph createBegin:self];
    SET_WINDOW_GEOMETRY_RECORD_NAME(consumerGraph);
    [consumerGraph setTitle:"A consumer"];
    [consumerGraph setAxisLabelsX: "Timex2" Y: "Visits/Spending"];
    consumerGraph=[consumerGraph createEnd];
    [consumerGraph createSequence: "Went to the market"
        withFeedFrom:[modelSwarm getConsumer:displayConsumerName]
            andSelector:M(getVisit)];
    [consumerGraph createSequence:"Spent"
        withFeedFrom:[modelSwarm getConsumer:displayConsumerName]
            andSelector:M(getSpending)];
```

```
// create the raster
    zoomFactor=10;
// colorMap
    colorMap=[Colormap create:self];
    [colorMap setColor:0 ToName:"black"];
    [colorMap setColor:1 ToName:"blue"];
    [colorMap setColor:2 ToName:"yellow"];
    [colorMap setColor:3 ToName:"red"];
// worldRaster
    worldRaster=[ZoomRaster create:self];
    SET_WINDOW_GEOMETRY_RECORD_NAME(worldRaster);
    [worldRaster setColormap:colorMap];
    [worldRaster setZoomFactor:zoomFactor];
    [worldRaster setWidth:[modelSwarm getWorldWidth] Height:[modelSwarm
          getWorldHeight]];
    [worldRaster setWindowTitle:"A little town"];
    [worldRaster pack];//to initialise and display
// marketDisplay
    marketDisplay=[Value2dDisplay createBegin:self];
    [marketDisplay setDisplayWidget:worldRaster colormap:colorMap];
    [marketDisplay setDiscrete2dToDisplay:[modelSwarm getMarket]];
    marketDisplay=[marketDisplay createEnd];
// worldDisplay
    worldDisplay=[Object2dDisplay createBegin:self];
    [worldDisplay setDisplayWidget:worldRaster];
    [worldDisplay setDiscrete2dToDisplay:[modelSwarm getWorld]];
    [worldDisplay setObjectCollection:[modelSwarm getListOfConsumers]];
    [worldDisplay setDisplayMessage:M(drawSelfOn:)];
    worldDisplay=[worldDisplay createEnd];
// set up buttons for probes
    [worldRaster setButton:ButtonRight Client:worldDisplay
            Message:M(makeProbeAtX:Y:)];
// display the world raster at the beginning
    [worldRaster erase];
    [marketDisplay display];
    [worldDisplay display];
    [worldRaster drawSelf];


    return self;
    }

-buildActions
    {
    [super buildActions];
    [modelSwarm buildActions];

// define displayActions
    displayActions = [ActionGroup create: self];
// update the graphs
    [displayActions createActionTo:spendingGraph message:M(step)];
    [displayActions createActionTo:consumerGraph message:M(step)];
    [displayActions createActionTo:worldRaster message:M(erase)];
```

```
    [displayActions createActionTo:marketDisplay message:M(display)];
    [displayActions createActionTo:worldDisplay message:M(display)];
    [displayActions createActionTo:worldRaster message:M(drawSelf)];
    [displayActions createActionTo:self message:M(observerCheckToStop)
        ];
    [displayActions createActionTo:probeDisplayManager message: M(
        update)];
    [displayActions createActionTo:actionCache message: M(doTkEvents)];
// define displaySchedule
    displaySchedule = [Schedule createBegin: self];
    [displaySchedule setRepeatInterval: 2];
    displaySchedule = [displaySchedule createEnd];
    [displaySchedule at: 1 createAction: displayActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [displaySchedule activateIn: self];
    [modelSwarm activateIn: self];
    return [self getSwarmActivity];
    }

-observerCheckToStop
    {
    if ([modelSwarm checkToStop]==1)
        {
        printf("THE MODEL HAS FINISHED TO RUN!\n");
        [controlPanel setStateStopped];
        }
    return self;
    }
@end
```

6 Girare più volte la simulazione

Listing 38: La classe per il Consumer: interfaccia

```
// consumer.h
// load program libraries
#import <objectbase.h>
//#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <collections.h>
#import <random.h>
#import "Integer.h"

@interface Consumer:SwarmObject
    {
// define variables for the consumer
    int myBudget;
```

```
    int myMaxBudget;
    int myName;
    int moneySpent;
    int marketGoer;
    id <List> listOfSpending;
    id <Array> arrayOfVisits;
    int currentTime;
    int positionX,positionY;
    }

// define methods for the consumer
// this method passes values for the consumer variables
-setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(int)
     startBudget Goer:(int)goer;

// these methods deal with the positioning of the consumer
// on the space
-setPositionX:(int)x Y:(int)y;
// draw itself on the raster
-drawSelfOn:(id <ZoomRaster>)raster;
// make the color of the agent depends on whether he is in the market
     or not
-(int)getStrategyColor;

// methods to create the list of spending and the array of visits
-createListOfSpending:aZone;
-createArrayOfVisits:aZone:(int)size;

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget;

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket;

// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend;

// this method resets the budget of the consumer once the goods have
// been bought
-(int)calculateRemainingBudget;

// these methods are used to add elements to the list of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)value;
-updateVisits:(int)offset:(int)value;

// these methods are needed to pass values of the consumer to other
```

```
// parts of the program
-(int)getConsumerName;
-(int)getBudget;
-(int)getVisitValue:(int)offset;
-(int)getSpendingValue;
-(int)getVisit;
-(int)getSpending;
-(int)getPositionX;
-(int)getPositionY;
-getAllSpending;
@end
```

Listing 39: La classe per il Consumer: implementazione

```
// consumer.m
// load the header files
#import "Consumer.h"

@implementation Consumer

// define methods for the consumer
// this method passes values for the consumer variables
-setConsumerName:(int)name MaxBudget:(int)maxBudget StartBudget:(int)
      startBudget Goer:(int)goer;
    {
    myName=name;
    myBudget=startBudget;
    myMaxBudget=maxBudget;
    marketGoer=goer;
    return self;
    }

// these methods deal with the positioning of the consumer
// on the space
-setPositionX:(int)x Y:(int)y
    {
    positionX=x;
    positionY=y;
    return self;
    }

// draw itself on the raster
-drawSelfOn:(id <ZoomRaster>)raster
    {
    [raster drawPointX:positionX Y:positionY Color:[self
          getStrategyColor]];
    return self;
    }

// make the color of the agent depends on whether he is in the market
// or not. getStrategyColor links a color to the type of player and
// the strategy played. The output is an integer corresponding to the
// index of the color in the color map
-(int) getStrategyColor
```

```
    {
    if (marketGoer==0)
        {
        return 3;
        }
    else if (marketGoer==1)
        {
        return 1;
        }
    else
        {
        printf("wrong marketGoer value\n");
        exit(0);
        }
    }


// methods to create the list of spending and the array of visits
-createListOfSpending:aZone
    {
    listOfSpending=[List create:aZone];
    return self;
    }

-createArrayOfVisits:aZone:(int)size
    {
    // note, arrays are of fixed size, so they need the setCount method
    // when created
    arrayOfVisits=[Array create:aZone setCount:size];
    return self;
    }

// this method draws a random number between 0 and
// maxBudget to determine the budget of the consumer
// (no need to pass arguments as myMaxBudget is a
// global variable for the consumer)
-(int)findBudget
    {
    // take myBudget and add to it a random variable between
    // 0 and myMaxBudget (need to add as myBudget may not
    // be 0 (if not all was spent))
    myBudget+=[uniformIntRand getIntegerWithMin:0 withMax:myMaxBudget];
    return myBudget;
    }

// this method draws a random number: returns 0 if the consumer
// stays home or 1 if he goes to the market
-(int) goToTheMarket
    {
    int k;
    k=[uniformIntRand getIntegerWithMin:0 withMax:1];
    marketGoer=k;
    return k;
    }
```

```
// this method will only be used if the consumer goes to the
// market. It draws a random number from 0 to the value in myBudget
// and returns it: it is the amount spent.
-(int) spend
    {
    moneySpent=[uniformIntRand getIntegerWithMin:0 withMax:myBudget];
    return moneySpent;
    }

// this method resets the budget of the consumer once the goods have
// been bought
-(int) calculateRemainingBudget
    {
    myBudget-=moneySpent;
    return myBudget;
    }

// these methods are used to add elements to the list of spendings,
// and the array of visits. They will also take care of casting the
// (int) values into Integer objects
-updateSpending:(int)value
    {
    Integer * valueObject;

    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [listOfSpending addFirst:valueObject];
    return self;
    }

-updateVisits:(int)offset:(int)value
    {
    Integer * valueObject;
// set the currentTime (equivalent to modelTime)
    currentTime=offset;
    valueObject=[Integer create:[self getZone]];
    [valueObject setValue:value];

    [arrayOfVisits atOffset:offset put:valueObject];
    return self;
    }

// these methods are needed to pass values of the consumer to other
// parts of the program
-(int)getConsumerName
    {
    return myName;
    }

-(int)getBudget
    {
    return myBudget;
```

```
    }

// get value at offset, in arrayOfVisits
-(int)getVisitValue:(int)offset
    {
    Integer * element;
    element=[arrayOfVisits atOffset:offset];
    return [element getMyValue];
    }

// get value of element at key
-(int)getSpendingValue
    {
    Integer * element;
    element=[listOfSpending getFirst];
    return [element getMyValue];
    }

-(int)getVisit
    {
    return [self getVisitValue:currentTime];
    }

-(int)getSpending
    {
    return [self getSpendingValue];
    }
-(int)getPositionX
    {
    return positionX;
    }

-(int)getPositionY
    {
    return positionY;
    }
-getAllSpending
    {
    return listOfSpending;
    }
@end
```

Listing 40: Il modelSwarm: interfaccia

```
// modelSwarm.h

// load program libraries
#import <objectbase.h>
#import <objectbase/Swarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <simtools.h>
#import <random.h>
#import <space.h>
```

```
#import <space/Discrete2d.h>
#import <space/Grid2d.h>
#import "Consumer.h"
#import "Integer.h"

@interface ModelSwarm:Swarm
    {
// here are declared variables which are global to the ModelSwarm class
    id <Schedule> modelSchedule;
    id <ActionGroup> modelActions;
    int modelTime;
    int maxTime;
    int startBudget;
    int maxBudget;
    int notFinished;
    int numberOfConsumers;
    id <List> listOfConsumers;
    int worldWidth,worldHeight;
    int sizeOfMarket;
    int xMin,yMin,xMax,yMax;
    id <Grid2d> world;
    id <Discrete2d> market;
    }

// creation methods which allow us to initialise parameters
+createBegin:(id) aZone;
-createEnd;
-setSimulationParameters:(int)simStartBudget;
// this method creates the consumers
-buildObjects;
// this method positions the consumers in the world, excluding the
     market,
// or only in the market according to the value of the last argument
-findPositionInWorld:(id <Grid2d>)aWorld For:(Consumer *)aGuy
     ExcludeMarket:(int)exclude;
// this method deals with what happens on a market
-marketDay;
// these methods deal with the running of the model
-increaseTime;
-(int)checkToStop;
-buildActions;
-activateIn: (id) swarmContext;
// these methods are used to get informations for characteristics
// of the model
-getListOfConsumers;
-getConsumer:(int)name;
-(int)getCurrentTime;
-(int)getWorldWidth;
-(int)getWorldHeight;
-getWorld;
-getMarket;
-getAllSpending;
@end
```

Listing 41: Il modelSwarm: implementazione

```
// ModelSwarm.m

#import "ModelSwarm.h"

@implementation ModelSwarm

+createBegin: (id) aZone
    {
    ModelSwarm * obj;
    id <ProbeMap> modelProbeMap;
// call the createBegin method of the superClass
    obj = [super createBegin: aZone];
// create the probemap for the model
    modelProbeMap=[EmptyProbeMap createBegin:aZone];
    [modelProbeMap setProbedClass:[self class]];
    modelProbeMap=[modelProbeMap createEnd];

    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"maxTime"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        numberOfConsumers"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        startBudget"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        maxBudget"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        worldWidth"
                inClass:[self class]]];
    [modelProbeMap addProbe:[probeLibrary getProbeForVariable:"
        worldHeight"
                inClass:[self class]]];
    [probeLibrary setProbeMap:modelProbeMap For:[self class]];
    return obj;
    }

-createEnd
  {
  return [super createEnd];
  }

-setSimulationParameters:(int)simStartBudget
    {
    startBudget=simStartBudget;
    return self;
    }

-buildObjects
    {
    int i,x,y;
    int name;
```

```
    id <ProbeMap> consumerProbe;

    [super buildObjects];

// set the position of the market in the center of the world
    xMin=(worldWidth-sizeOfMarket)/2;
    xMax=xMin+sizeOfMarket;
    yMin=(worldHeight-sizeOfMarket)/2;
    yMax=yMin+sizeOfMarket;

// initialise the world as a grid2d and fill it with "nil" objects
    world=[Grid2d create:[self getZone] setSizeX:worldWidth Y:
          worldHeight];
    [world fillWithObject:nil];
// initialise market as a Discrete2d and fill it with a value object
// to get the yellow square representing the market
    market=[Discrete2d create:[self getZone] setSizeX:worldWidth Y:
          worldHeight];
    for (y=yMin; y<yMax; y++)
        {
        for (x=xMin; x<xMax; x++)
            {
            [market putValue:2 atX: x Y: y];
            }
        }

// create the list of consumers
    listOfConsumers=[List create:[self getZone]];

// iterate over all possible consumers (from 1 to numberOfConsumers)
    for (i=1;i<=numberOfConsumers;++i)
        {
        Consumer * aConsumer;
// name of consumer=index i
        name=i;
// create the consumers
        aConsumer=[Consumer create: [self getZone]];
        [aConsumer setConsumerName:name MaxBudget:maxBudget StartBudget
              :startBudget Goer:0];
// set their positions to -999
        [aConsumer setPositionX:-999 Y:-999];
// create the list and arrays (the array is created with a size of
// maxTime+1, as time starts at 0
        [aConsumer createListOfSpending:[self getZone]];
        [aConsumer createArrayOfVisits:[self getZone]:maxTime+1];
// position the consumers in the world, excluding market
        [self findPositionInWorld:world For:aConsumer ExcludeMarket:1];
// probe for consumer
        consumerProbe=[EmptyProbeMap createBegin:[self getZone]];
        [consumerProbe setProbedClass:[Consumer class]];
        consumerProbe=[consumerProbe createEnd];
        [consumerProbe addProbe:[probeLibrary getProbeForVariable:"
              myName"
                 inClass:[Consumer class]]];
```

```
        [probeLibrary setProbeMap:consumerProbe For:[Consumer class]];
// add consumer to the list
        [listOfConsumers addFirst:aConsumer];
        }
    return self;
    }

// this method positions the consumers in the world, excluding the
     market,
// or only in the market according to the value of the last argument
-findPositionInWorld:(id <Grid2d>)aWorld For:(Consumer *)aGuy
     ExcludeMarket:(int)exclude
    {
    int trialX,trialY;
// set trialX and trialY to negative values (not in the world) to start
     the
// while loop
    trialX=-999;
    trialY=-999;
// if the consumers have already been put on a space (positionX and
     positionY are not
// -999, then put a nil object at their current position
    if (([aGuy getPositionX]>=0)&&([aGuy getPositionY]>=0))
        {
        [aWorld putObject:nil atX:[aGuy getPositionX] Y:[aGuy
             getPositionY]];
        }
// put consumers in the world randomly. if exclude=1, then the part of
// the world corresponding to the market is excluded from the
// possibilities, if exclude=0, the consumer is put in the market
// when choosing a value for the position of the consumer, we need
// to check that: (1) both coordinates are positive, (2) they are
// in the part of the world where they should be, (3) there is not
// already a player in the spot. This explain why the while condition
// is quite complex
    if (exclude==1)
        {
        while( ((trialX<0)&&(trialY<0)) ||
            ((trialX>=xMin)&&(trialX<=xMax)&&(trialY>=yMin)&&(trialY<=
                 yMax)) ||
            ([world getObjectAtX:trialX Y:trialY]!=nil) )
            {
            trialX=[uniformIntRand getIntegerWithMin:0 withMax:
                 worldWidth-1];
            trialY=[uniformIntRand getIntegerWithMin:0 withMax:
                 worldHeight-1];
            }
        }
    else if (exclude==0)
        {
        while ( ((trialX<0)&&(trialY<0)) ||
            ([world getObjectAtX:trialX Y:trialY]!=nil) )
            {
```

```
            trialX=[uniformIntRand getIntegerWithMin:xMin withMax:xMax
                ];
            trialY=[uniformIntRand getIntegerWithMin:yMin withMax:yMax
                ];
            }
        }
    else
        {
        printf("wrong value for exclude!\n");
        exit(0);
        }
    [aGuy setPositionX:trialX Y:trialY];
    [aWorld putObject:aGuy atX:trialX Y:trialY];
    return self;
    }


-marketDay
    {
    int go;
    int spending;
    int budget;
    id <Index> i=nil; // to iterate over the list of consumers
    Consumer * listElement;

// iterate over the list of consumers
// first, create the index i
    i=[listOfConsumers begin:[self getZone]];
    while ((listElement=[i next])!=nil)
        {
// update the budget of the consumer
        budget=[listElement findBudget];

// is he going to the market?
        go=[listElement goToTheMarket];
        if (go)
            {
            spending=[listElement spend];
// add 1 to arrayOfVisits, at the position corresponding to
// current modelTime
            [listElement updateVisits:modelTime:1];
// add spending at the key modelTime in the mapOfSpending
            [listElement updateSpending:spending];
// now, print a report of the consumer's actions
//          printf("This is time %d\n",modelTime);
//          printf("I am consumer %d\n",[listElement getConsumerName]);
//          printf("My current budget is %d\n",[listElement getBudget])
    ;
//          printf("Did I go to the market? %d (from array)\n",[
    listElement getVisitValue:modelTime]);
//          printf("I spent %d (from list)\n",[listElement
    getSpendingValue]);
//              printf("I have %d of currency left.\n",[listElement
    calculateRemainingBudget]);
```

```
// put in position in world on market space
            [self findPositionInWorld:world For:listElement
                 ExcludeMarket:0];
            }
        else
            {
// add 0 to arrayOfVisits at modelTime
            [listElement updateVisits:modelTime:0];
// add 0 to listOfSpending at modelTime
            [listElement updateSpending:0];
// print consumer's state
//          printf("This is time %d\n",modelTime);
//          printf("I am consumer %d\n",[listElement getConsumerName]);
//          printf("My current budget is %d\n",[listElement getBudget])
    ;
//          printf("Did I go to the market? %d (from array)\n",[
    listElement getVisitValue:modelTime]);
//          printf("I have %d of currency left.\n",[listElement
    getBudget]);
// put in position in world outside market
            [self findPositionInWorld:world For:listElement
                 ExcludeMarket:1];
            }
        }// end of iteration on listOfConsumers
// it is good practice to drop unused objects like indexes when they
// are no longer needed
        [i drop];
    return self;
    }

-increaseTime
    {
// at the end of a period, modelTime need to be increased by 1
    ++modelTime;
    return self;
    }
-(int)checkToStop
    {
// if modelTime<maxTime, then notFinished is 1, otherwhise it
// is 0 (return respectively 0 or 1 for the observer)
    if (modelTime<=maxTime)
        {
        notFinished=1;
        return 0;
        }
    else
        {
        notFinished=0;
        return 1;
        }
    }


-buildActions
```

```
    {
// create an action group for the actions the model need to
// perform at each time period
// first, we tell people to go to the market (marketDay)
// then we increase the time of the model (increaseTime)
// finally we check whether this time is still valid (checkToStop)
    modelActions=[ActionGroup createBegin:self];
    modelActions=[modelActions createEnd];
    [modelActions createActionTo:self message:M(marketDay)];
    [modelActions createActionTo:self message:M(increaseTime)];
    [modelActions createActionTo:self message:M(checkToStop)];

// now schedule the actions in time
    modelSchedule = [Schedule createBegin: self];
    [modelSchedule setRepeatInterval:2];
    modelSchedule = [modelSchedule createEnd];
    [modelSchedule at: 0 createAction: modelActions];
    return self;
    }

-activateIn: (id) swarmContext
    {
    [super activateIn: swarmContext];
    [modelSchedule activateIn: self];

    return [self getActivity];
    }

// methods to pass parameters to other classes
-getListOfConsumers
    {
    return listOfConsumers;
    }

-getConsumer:(int)name
    {
// note that elements are entered in the list using the addFirst method
    ,
// so the first element is the latest agent created.
    return [listOfConsumers atOffset:[listOfConsumers getCount]-name];
    }

-(int)getCurrentTime
    {
    return modelTime;
    }
-(int)getWorldWidth
    {
    return worldWidth;
    }
-(int)getWorldHeight
    {
    return worldHeight;
    }
```

```
-getWorld
    {
    return world;
    }
-getMarket
    {
    return market;
    }
-getAllSpending
    {
    Consumer * element;
    Integer * spending;
    id <List> elementSpending;
    id <List> listOfAllSpending;
    id <Index> i=nil;
    id <Index> j=nil;

    listOfAllSpending=[List create:[self getZone]];
    // iterate through the elements of listOfConsumers
    // to create elementSpending
    i=[listOfConsumers begin:[self getZone]];
    while ((element=[i next])!=nil)
        {
        elementSpending=[List create:[self getZone]];
        elementSpending=[element getAllSpending];
        // iterate through the elements of elementSpending to create
        // listOfAllSpending
        j=[elementSpending begin:[self getZone]];
        while ((spending=[j next])!=nil)
            {
            [listOfAllSpending addFirst:spending];
            }
        }
    return listOfAllSpending;
    }

@end
```

Listing 42: L'ExperimentSwarm: interfaccia

```
// ExperSwarm.h

#import <simtoolsgui/GUISwarm.h>
#import <objectbase/SwarmObject.h>
#import <activity.h>
#import <collections.h>
#import <objectbase.h>
#import <analysis.h>
#import <simtools.h>
#import <simtoolsgui.h>
#import <gui.h>
#import "ModelSwarm.h"
#import "SimulationData.h"
```

```
@interface ExperSwarm: GUISwarm
  {
  int minStartBudget;
  int maxStartBudget;
  int incStartBudget;
  int setStartBudget;
  id <List> listOfAllSpending;
  float averageSpending;

  ModelSwarm * modelSwarm;

  id <Graph> spendingGraph;
  id <GraphElement> spending;

  id <ProbeMap> modelProbeMap;
  }

+ createBegin: (id) aZone;
- createEnd;
- buildObjects;
- activateIn: (id) swarmContext;
// run the simulation
- run;

@end
```

Listing 43: L'ExperimentSwarm: implementazione

```
// ExperSwarm.m

#import "ExperSwarm.h"


// implementation of ExperSwarm object

@implementation ExperSwarm

+ createBegin: aZone
  {
  ExperSwarm *obj;
  id <ProbeMap> experProbeMap;

  obj = [super createBegin: aZone];

// probe map for ExperSwarm
  experProbeMap = [EmptyProbeMap createBegin: aZone];
  [experProbeMap setProbedClass: [self class]];
  experProbeMap = [experProbeMap createEnd];

  [experProbeMap addProbe: [probeLibrary getProbeForVariable: "
       minStartBudget"
                                    inClass: [self class]]];
  [experProbeMap addProbe: [probeLibrary getProbeForVariable: "
       maxStartBudget"
```

```
                                    inClass: [self class]]];
  [experProbeMap addProbe: [probeLibrary getProbeForVariable: "
       incStartBudget"
                                    inClass: [self class]]];

  [probeLibrary setProbeMap: experProbeMap For: [self class]];

  return obj;
  }

- createEnd
  {
  return [super createEnd];
  }

- buildObjects
  {
  [super buildObjects];

  CREATE_ARCHIVED_PROBE_DISPLAY (self);

// Allow the user to alter experiment parameters
  [controlPanel setStateStopped];

// build the Graph for model results
  spendingGraph = [Graph createBegin: self];
  SET_WINDOW_GEOMETRY_RECORD_NAME(spendingGraph);
  spendingGraph = [spendingGraph createEnd];
  [spendingGraph setTitle: "Average Spending"];
  [spendingGraph setWidth:500 Height:400];
  [spendingGraph setAxisLabelsX:"Start Budget" Y:"Average Spending"];

  spending=[spendingGraph createElement];
  [spending setLabel: "average spending on the market"];
  [spending setColor: "blue"];

  return self;
  }

- run
  {
  id <LispArchiver> outFile;
  SimulationData * simLoop;
  id <LispArchiver> archiver;
  id <Index> i=nil;
  int sum=0;
  Integer * spendingElement;

// start running simulation: for each possible starting budget
  for (setStartBudget=minStartBudget;
        setStartBudget<=maxStartBudget;setStartBudget+=incStartBudget)
    {
printf("this round, StartBudget is %d \n",setStartBudget);
```

```
//create setup file for Model Swarm

    simLoop=[SimulationData create:[self getZone]];
    [simLoop initPara:setStartBudget];
    outFile=[LispArchiver create:self setPath:"loop.scm"];
    [outFile putShallow:"modelSwarm" object:simLoop];
    [outFile sync];
    [outFile drop];

// create listOfAllSpending
    listOfAllSpending=[List create:[self getZone]];

//load the data for the modelSwarm
    archiver=[LispArchiver create:globalZone setPath:"parameters.scm"];
    if((modelSwarm=[archiver getWithZone:globalZone key:"modelSwarm"])
         ==nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [archiver drop];

//load the data modified for the experiment (must come after the
//data for the model as it modifies the values of some of them)
//archiver loads the data from the file loop.scm. They are a
     SimulationData
//object, so use the setPara:model method to initialise the modelSwarm
    archiver=[LispArchiver create:globalZone setPath:"loop.scm"];
    if((simLoop=[archiver getWithZone:globalZone key:"modelSwarm"])==
          nil)
        {
        raiseEvent(InvalidOperation,"can't find file or key\n");
        }
    [simLoop setPara:modelSwarm];
    [archiver drop];
    [simLoop drop];

    [modelSwarm buildObjects];
    [modelSwarm buildActions];

// run modelSwarm until it finishes
        while ([modelSwarm checkToStop]==0)
          {
          [modelSwarm marketDay];
          [modelSwarm increaseTime];
          [modelSwarm checkToStop];
          }

// put values in listOfAllSpending from modelSwarm
    listOfAllSpending=[modelSwarm getAllSpending];
// get average spending to use in graph
    i=[listOfAllSpending begin:[self getZone]];
    while ((spendingElement=[i next])!=nil)
        {
        sum+=[spendingElement getMyValue];
```

```
        }
    averageSpending=sum/[listOfAllSpending getCount];
printf("averageSpending %f\n",averageSpending);

// add data for graph
    [spending addX:setStartBudget Y:averageSpending];
    } // end of for loop on startBudget

// end of experiment, draw graph and stop control panel
  printf("End of the experiment\n");
  [spendingGraph pack];
  [controlPanel setStateStopped];

  return self;
  }

- activateIn: swarmContext
  {
  [super activateIn: swarmContext];

  return [self getActivity];
  }


@end
```