Agent Simulation using Object-Oriented Methodology

Weidong Sun

A Major Report

In

The Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

November 2000

ABSTRACT

Agent Simulation using Object-Oriented Methodology

Weidong Sun

The intention of this project is to design and implement a simulation model by using Object-Oriented methodology. Simulation has been extensively applied to fields like ecosystem, stock market and aerospace. This project will concentrate on ecosystem.

Much effort has been devoted to analyzing the characteristics of ecosystem and interactions among the elements of the ecosystem. Based on our findings, a simulation model was developed to extract and reflect the most fundamental activities in an ecosystem, such as trading, combating, reproduction, just to name a few. By building a simulation model, the ecosystem becomes analyzable and predictable.

Important features of this project are the experimentation of the object-oriented paradigm and the utilization of UML and C++ language. This project is intended to serve as a base for those who aim to explore further in the field of simulation.

Acknowledgments

I wish to express my sincere gratitude to my supervisor, Dr. Peter Grogono, for all his enthusiastic support, careful supervision, and consistent guidance during the development of this major report. Also my thanks to Dr. Rajjan Shinghal for reviewing my major report.

Table of Contents

CHAPTER 1 INTRODUCTION 1
Aim of the Project 1
Motivation of the Project 1

CHAPTER 2 BACKGROUND 4
Agent 4
Agent Interaction 5
Combat 5
Trade 6
Reproduction 6
Environment 7
Simulation Cycle 7

CHAPTER 3 DESIGN 9
Class CSimApp 9
Class CEnvironment 9
Class CCell 9
Class CAgentList 10
Class CAgent 10
Class CGenome 10
Class CSimString 10
Class CSimList 11
Class CSimParam 11
Class CSummaryTable 12
User Interface 12

CHAPTER 4 IMPLEMENTATION 13
Class CSimApp 13
Class CEnvironment 14
Class CCell 15
Class CAgentList 16
Class CAgent 17
Class CGenome 20
Class CSimString 21
Class CSimList 23
Class CSimParam 24
Class CSummaryTable 25

CHAPTER 5 RESULT 32
Example 1 33
Example 2 40

CHAPTER 6 CONCLUSION 44
Experience on Object orient Programming 44
Future Improvement 45

CHAPTER 7 BIBLIOGRAPHY 46

APPENDIXES 47
CSimString.h 48
CGenome.h 57
CSimList.h 61
CAgent.h 54
CAgentList.h 53
Ccell.h 51
Cenvironment.h 49
CsimParam.h 62
CsummaryTable.h 63
CsimApp.h 48
Utility.h 65

List of Figures

Figure 1: Class Diagram for Agent Simulation Application 27
Figure 2: Sequence Diagram of CSimApp::Initialize() 28
Figure 3: Collaboration Diagram of CSimApp::Initialize() 29
Figure 4: Sequence Diagram of CSimApp::RunOneCycle() 30
Figure 5: Collaboration Diagram of CSimApp::RunOneCycle() 31
Figure 6: before simulation start / after simulation stop 34
Figure 7: Initial Statistics of the simulation 35
Figure 8: Simulation Statistics after 50 cycles for Example 1 36
Figure 9: Simulation statistics after 100 cycles in Example 1 37
Figure 10: Simulation statistics after 150 cycles in Example1 38
Figure 11: Simulation statistics after 200 cycles in Example 1 39
Figure 12: Simulation Statistics after cycle 50 and 100 in Example 2 41
Figure 13: Simulation Statistics after cycle 200 and 350 in Example 2 42
Figure 14: Simulation Statistics after cycle 1000 in Example 2 43

Chapter 1 Introduction

Aim of the Project

This project is about modeling an ecosystem. A system is a group of parts that are interacting according to some kind of process [1]. An ecosystem is a biotic and functional system or unit, which is able to sustain life and includes all biological and non-biological variables in that unit [4].

The intention of the project is to design and implement a simple simulation model that is as realistic as possible by analyzing the key elements and interactions among them and simulating their behavior, while ignoring less important details. And also by using object-oriented methodology and patterns, we plan to build some classes that might serve as basic modular classes for general simulation modeling. People who are interested in simulation may take advantage of these classes.

Our particular example will be the agent simulation in ecosystem by using object-oriented modeling technique with UML and C++ languages.

Motivation of the Project

Many interesting systems are difficult to analyze and control using traditional methods. These include natural ecosystems, immune systems, and other social organizations. The reason for the difficulty is nonlinear interactions among system components. Those components are extremely adaptive, have the ability of self-organization and have a large number of feedback mechanisms. For instance, the growth of living organisms is dependent on many interacting factors, which are functions of time. Feedback mechanisms will simultaneously regulate all the factors and rates and they also interact and are also functions

of time [7]. An ecosystem has so many interacting components that it is difficult ever to be able to examine all these relationships. Useful and predictive mathematical treatments are difficult, due primarily to non-linearity, discreteness, spatial inhomogeneities, and the changing behavior of the primitive elements of the system [3].

However, there are still several approaches suitable for the study of ecosystems:

(1) Empirical studies where bits of information are collected, and an attempt is made to integrate and assemble these into a complete picture.

(2) Comparative studies where a few structural and a few functional components are compared for a range of ecosystem types.

(3) Experimental studies where manipulation of a whole ecosystem is used to identify and elucidate mechanisms.

(4) Modeling or computer simulation studies.

Among these, the last approach is increasingly popular in recent years, thanks mainly to the rapid advance of computer technology. There are four modeling techniques that have been developed recently: object-oriented models, individual-based models, model constructed by using artificial intelligence and expert systems, and fuzzy knowledge-based models. They have been developed as new methods of model construction following recognition of the shortcoming in the data and in the rigidity of the traditional models.

As mentioned previously the aim of this project is to develop an object-oriented model (OOM). OOMs are based on the idea that programs should represent the interactions between abstract representation of real objects rather than a linear sequence of calculations commonly associated with programming, commonly referred to as procedural programming [5].

The central idea of object-oriented programming (OOP) is the concept of a *class* that describes both the structure of an object and a set of procedures for initializing and using it in the model. One obvious example of a class is the definition of a population, which is the basic building block for many ecological models. Populations are characterized by variables such as mean size, age, number and exhibit processes such as reproduction, growth and mortality. Each type of population is unique, although there are many similarities, such as the above mentioned processes. We can therefore treat different classes of populations accordingly and need only to add those particular features which need to be different in the model context.

The OOP defines different processes in different modules that can be used in the various classes. It is possible to have several different versions of the processes. OOP also offers a mechanism that lets us hide the more detailed information about the internal description of objects, so that we can use it without having to describe it explicitly in OOM.

OOP offers many advantages for developers of ecological models. First of all there is a close connection between object and natural grouping. The concept of inheritance is directly borrowed from biology. OOP makes it possible to develop models, that are simpler to interpret for the modeler and that easily can be modified and refined.

Chapter 2 Background

An ecosystem has the following properties:

- a collection of primitive components, called "agents";
- nonlinear interactions between agents and between agents and their environment;
- unanticipated global behaviors that result from the interactions;
- Agents that adapt their behavior to other agents and environmental constraints, causing system components and behaviors to evolve over time.

Based on these properties, we have developed the following concepts in our OOM.

Agent

Depending on specific situation, an agent could be applied to a cell in an organic system or an individual of a kind of species. Our project is more likely to be applied to the former case.

Each agent has a genome that is roughly analogous to a single chromosome in a haploid species. A genome has several genes that encode internal preferences and behavioral rules of the agent and so forth. One agent will interact with another on the basis of its own internal conditions (rules for interaction). This allows the possibility of sophisticated interactions between agents, including mimicry, bluffing, other forms of deception, and some intransitivities. For example, in mimicry, an agent can appear dangerous but actually be unwilling to fight. An example of intransitive combat relationships would be the following: An agent A might attack an agent B, and B attack C, but it does not follow that A will attack C. This has obvious parallels in real-world systems. The (internal) condition genes possessed

by every agent are, namely, *attack*, *defend*, *beauty*, *combat*, *trade*, *lust*, *eat* and *give*. These genes are used to determine what sort of interaction will take place between a pair of agents, and what the outcome will be.

The *attack*, *defend* and *combat* genes determine the mutual threats and the likelihood of combat between a pair of agents. Likewise, *beauty* and *lust* genes are for reproduction. *Trade* and *attack* genes determine the willingness to exchange resources. *Eat* and *give* genes determine the resource transmission between agents and the environment (paying taxes and eating foods).

Agents also have a reservoir, or stomach, in which to store resources. Resources from the reservoir are used to pay taxes, to produce offspring, and for trade.

Agent Interaction

There are three direct forms of agent-agent interaction: combat, trading, and reproduction. All of these interactions take place between agents that are located at the same site and all involve the transfer of resources between agents. Tests for interactions are always conducted in a random order. The interaction between any two agents could be none or one of the three forms. For example, if one of the three forms of interaction take place between agent A and B, the other two forms will not be tested; if not, one of the other two forms of interaction will be tested. When all the three forms of interactions are tested yet nothing happens, then there is no interaction between these two agents at this time.

Combat

Combat is an idealization of any antagonistic interaction between real-world entities. The likelihood of combat is decided by *attack*, *defend* and *combat* genes. If two agents in a

real-world system are behaving in a competitive (threaten) fashion, this would be modeled in the simulation by designing the agents that engage in combat. When combat occurs, one agent is killed (and its resources are placed in the reservoir of the survivor), unless it surrenders first. There are four possibilities, as shown following:

1) If A threatens B and B threatens A, then the combat occurs;
2) If A threatens B but B does not threatens A, then B either surrenders to avoid the combat or the combat occurs;
3) If both A and B don't threaten each other, then nothing happens.

Trade

Unlike combat, trading requires mutual agreement. *Trade* and *attack* genes determine the possibility of trading. If for both parties the willingness for trading is greater than the other party's attacking intention, trade will occur. When trade takes place, resources are transferred between the agents' *reserves*. The agents' *give* genes limit the size of transfer.

Reproduction

Like trading, reproduction takes place only if both agents are willing. The *lust* and *beauty* genes determine the likelihood of reproduction. If the mutual attractions are great enough, then the reproduction occurs between two agents, later called "parents". Reproduction consists of the following three steps, which are described in detail below:

1) If the parents have sufficient resources to construct the child, the next step follows. Otherwise no reproduction occurs.
2) A new child is created. Its genome is constructed from the parents' genomes.
3) Part of the parents' resources will be given to the child.

A crossover point is a position in the genome. A genome of length n has n+1 possible crossover points. When reproduction does occur, the crossover points in the genomes of parent agents A and B are chosen randomly. The genome of the child agent consists of the left part of the crossover point in genome of A and the right part of the crossover point in genome of B.

Environment

Agents exist in an environment that consists of a two-dimensional grid of "sites" (or "cells") and each agent is located at a site. There are usually many agents at one site. Each agent begins its life in one site and moves away from that site if there is no food available for it. Each site produces renewable resources. Resources are represented by different letters of the alphabet, and genomes are constructed from the same letters.

Simulation Cycle

There is a fixed sequence of events that takes place in a single simulation cycle, which is described below. This description mentions several system parameters, which are defined more carefully in the next section.

1) The site produces a fixed amount of resources, and these are distributed as equally as possible among the agents at the site that are genetically able to collect them.
2) Agents collect resources from the site if any are available. Agents that do not acquire any resources migrate to a neighboring site. The neighboring site is selected at random from among those permitted by the geography of the world.
3) Interactions between agents occur at each site. These include trade, mating, and combat. For each interaction, both agents are selected randomly from the same site.

4) Each agent at each site is taxed (probabilistically). Each site exacts a resource tax from each agent with a given (worldwide) probability. If an agent does not possess the resources to pay the tax, it is deleted (killed) and its resources are returned to the environment

5) Agents are killed if they are inactive for several continue cycles. When an agent at a site dies, its resources are returned to the environment, thereby becoming available to other agents at that site.

Chapter 3 Design

The key for designing an Agent Simulation System using object-oriented technique is to identify the objects and classes of objects in the system. As described in previous chapter, we find the nouns such as agent, site (or cell) and environment, just to name a few, that would represent the object in the Simulation System. Therefore, several types of classes are defined, including the following:

CSimString	CGenome	CSimList	CAgent	CAgenList
CCell	CEnviroment	CSimParam	CSummaryTable	
CSimApp				

Figure 1 on page 27 is the class diagram of this application, which demonstrates the static structures and the relations among those classes.

Class CSimApp

This is the overall simulation class to represent this simulation.

Class CEnvironment

This class represents the overall environment that all agents live in. There is a two-dimensional grid of "sites" in the environment. Each site called a "cell". There is only one CEnviroment object in this simulation.

Class CCell

This class represents the site in an *environment*. There is a two-dimensional grid of "sites". Each site called a "cell". Each cell provides a fixed amount of resources (food) to the

agents living in that cell. Each agent begins its life in a cell and gets (eats) the food from the cell it lives in and also returns some resources back to the cell as "tax".

Class CAgentList

This is a linked list class. Its node is an instance of CAgent class. There are two CAgentList instances in each cell: BeforeList and AfterList. The BeforeList holds the agents that are not involved in interaction during current cycle yet. The AfterList holds the agents that have been involved in interaction during the current cycle.

Class CAgent

This is the core element in our simulation that represents the agents in an ecosystem. Each *agent* object has only one genome and one resource pool. The genome represents the identity of this agent. The resource pool keeps the collection of different resources that this agent currently owns.

Class CGenome

This class represents the genome that a agent has. Each genome consists of 8 genes: *attack, defend, beauty, combat, trade, lust, eat* and *give*.

Class CSimString

This class represents the general type of different genes and resource pools. A gene (CSimString object) consists of a number of different resources (represented as letters).

Class CSimList

This class represents the genome pool. The Genome pool contains all the different genomes, each associated with one or more agent object(s).

In order to monitor the varieties of genomes existing in the environment during the simulation and more importantly, to save the storage cost, I adopt a design pattern called "flyweight". Flyweight is a shared object that can be used in multiple contexts simultaneously [6]. In this application, genome is designed as a "flyweight" based on our assumption that some agents have identical genomes. For example, if agent A's genome is the same as agent B's, instead of creating two identical genome objects, this application creates only one such genome in the genome pool, which is to be shared by agent A and B.

One thing nice about using flyweight to implement genome object is that the shared genome object acts as an independent object in each context – it is indistinguishable from a genome object that's not shared.

However, the flyweight may introduce run-time costs associated with transferring, finding, and/or some other related computations. Consequently, in this application, it requires not only a constant time to create a genome but also an additional $O(n)$ time to find the matching genome from the genome pool.

Class CSimParam

We create the CSimParam class to centralize the simulation parameters and hence to extend the extensibility and reusability of the design. A *simulation parameter* is a value that remains constant for a particular run of the simulation. The values given in this outline are

intended to be typical, but further experiments will be needed to find values that give interesting results.

- SIDE – a value to define the number of cells in a environment. The default value is 10, giving 10×10 = 100 cells.
- POP – The number of agents in one cell during the initialization (i.e., population). There should be, on average, more than one agent in a cell. The number of agents is 500 by default.
- VARIETY – Number of different resources type. The default value is 5.
- TAX - Some of these resources would be returned to the environment as taxes.
- MINMATCH – the minimum number of common resource in *beauty* gene that allows two engaged agents to reproduce. The value is 0.1 by default.
- FRAC – a fraction of the remaining resources from parent agents that will be transferred to their child during reproduction. Default value is 0.33.
- ACTIVE – The number of cycles after which the simulation will destroy an agent that is apparently inactive. The default value is 10.

Class CSummaryTable

This class is used to gather run-time simulation data for future analysis. The data include number of reproductions, agents, newborns, deaths, combats, trades and battles won/drawn.

User Interface

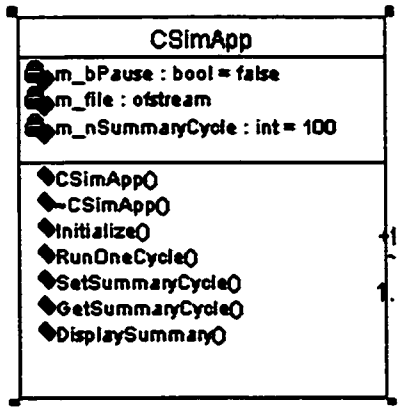
Since the classes for the User Interface are not essential to this report, I will not cover the topic here. However, detailed discussion can be found in Changjiang Zhang's major report [8].

Chapter 4 Implementation

Each class in the simulation has a set of functions (public and private) to carry out certain tasks. We summarize the main member functions along with a brief description in associated classes.

Class CSimApp

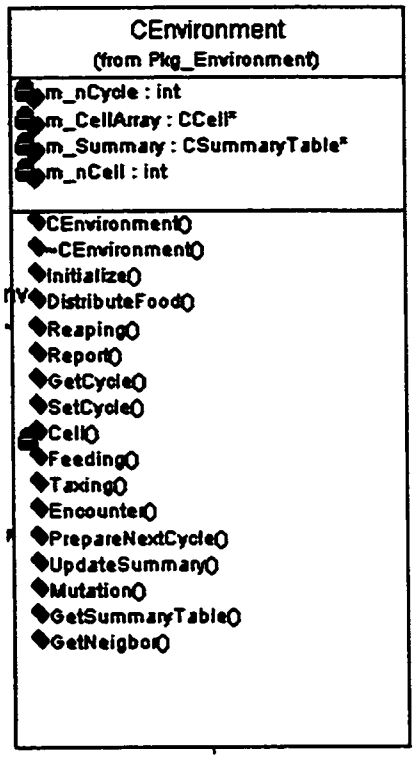
- *CSimApp():* Constructor
- *~CSimApp():* Destructor
- *GetCycleNumber():* return the number of simulation cycles that have occurred so far.
- *CSummaryTable* GetSummaryTable():* return the current summary simulation information at this moment.



- *void Initialize():* simulation initialization, which will create an environment object, and this environment object will create specified cell objects. Each cell object will create an agent element in turn.
- *void Initialize():* initialization which will create environment object and call CEnvironment::initialize(). See Figure 2 on page 28 and Figure 3 on page 29 for its sequence diagram and collaboration diagram, respectively.
- *void RunOneCycle():* enable the simulation to go through one cycle. See Figure 4 on page 30 and Figure 5 on page 31 for its sequence diagram and collaboration diagram, respectively.

Class CEnvironment

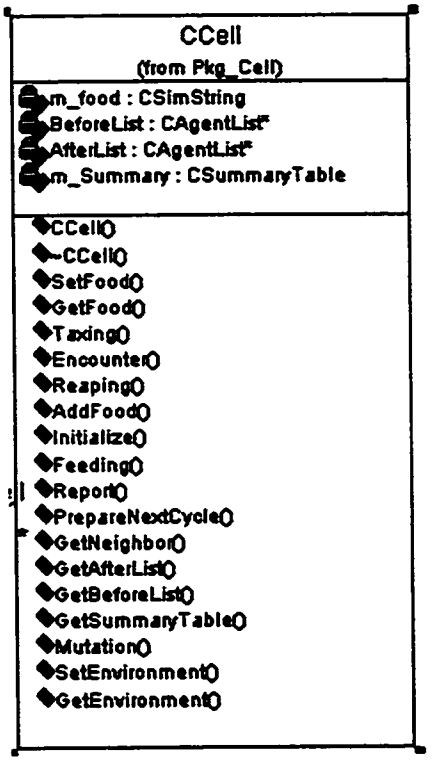
- *CEnvironment(int x = 1, int y = 1):* Constructor
- *~CEnvironment():* Destructor
- *CSummaryTable* GetSummaryTable():* accessor to return statistical information of this environment.
- *CCell* GetNeigbor(CCell*, DIRECTION):* return the neighbor cell of a specified cell at certain direction.
- *int GetCycle():* return the number of cycles this simulation has been running.
- *void SetCycle(int):* mutator to set the number of cycles this simulation has been running.
- *void Mutation():* enable all agents live in this environment to have a chance to do gene mutation.



- *void Initialize():* initialize this environment by creating cell objects.
- *void PrepareNextCycle():* enable each cell to finish the final step of current cycle.
- *void DistributeFood( ):* distribute food (resource) into this environment.
- *void Reaping():* remove agents that are inactive and return their resources to the cell they are currently living in.
- *void Report(ostream &):* write statistic information of this environment into specified output stream.
- *void Feeding():* feed all agents live in this environment.

- *void Taxing():* force all agents to pay tax to the environment where they live.
- *void Encounter():* enable all agents to encounter with each other.
- *void GenomeSetReport(ostream &):* write all genomes information inside this environment into a specified output stream.

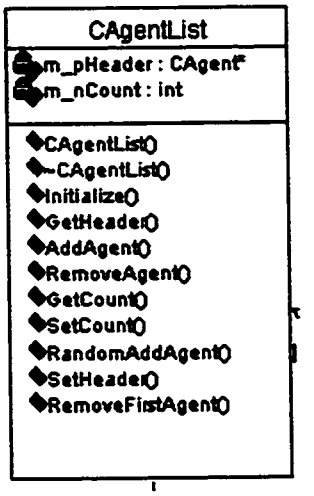
Class CCell



- *CCell(CEnvironment *pEnv = 0):* Constructor to create cell object inside specified environment.
- *~CCell():* Destructor to destroy this cell.
- *CEnvironment* GetEnvironment():* accessor to get the environment where the cell is located.
- *CSummaryTable* GetSummaryTable():* accessor to get the statistic information of this cell.
- *CAgentList* CgetAfterList():* accessor to get the AfterList of this cell.
- *CAgentList* GetBeforeList():* accessor to get the BeforeList of this cell.
- *void Initialize():* cell initialization by initialize its BeforeList and statistic information.
- *void PrepareNextCycle():* remove all agents from AfterList and add them randomly into BeforeList.
- *CCell* GetNeighbor(DIRECTION theDirection):* return the address of neighbor Cell at certain direction.

- *void Mutation():* enable all agents that live in this cell to do gene mutation.
- *void Taxing():* force all agents inside this cell to pay taxes to the cell.
- *Encounter():* enable all agents inside this cell to start interact each other.
- *void Reaping():* clean up the inactive agents by removing them from the environment, free the resources back to the environment.
- *Void AddFood(CsimString &):* distribute some food (resources) into this cell
- *void Feeding():* enable all agents inside this cell to eat some food (resources) from the cell.

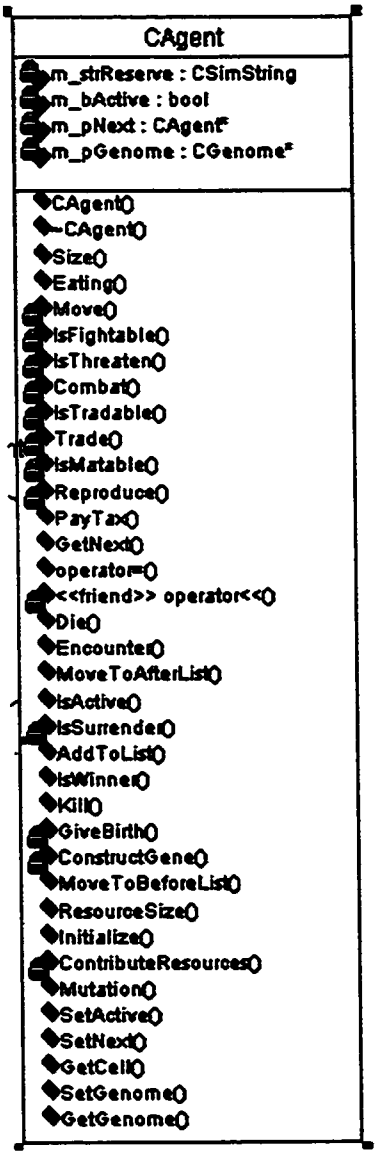
Class CAgentList



- *CAgentList():* constructor to create an empty list object.
- *~CAgentList():* Destructor to desotry the list object.
- *CAgent* GetHeader():* accessor to return the first agent element in this list.
- *int GetCount():* accessor to get the total number of agents this list contains.
- *void SetHeader(CAgent*):* mutator to set the first agent element in the list.
- *void SetCount(int):* mutator to set the number of agents in the list.
- *void Initialize( ):* this function create CSimParam.POP number of agent objects, initialize them, and put them into this list.
- *void AddAgent(CAgent*):* add an agent into this list.
- *CAgent* RemoveFirstAgent():* Remove the first agent node from this list
- *void RemoveAgent(CAgent*):* remove a specified agent from this list

- *void RandomAddAgent(CAgent* theAgent):* Add *theAgent* into random position of this list.

Class CAgent



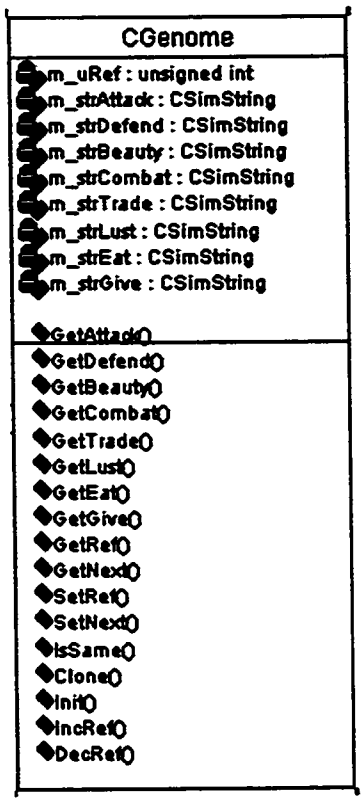
- *CAgent(CCell *aCell = 0):* Constructor to create agent object with empty genome and put it into specified cell location.
- *~CAgent():* destructor to destroy the agent object and remove it from memory.
- *CCell* GetCell():* return the cell where this agent lives.
- *CAgent* GetNext():* accessor to get the address of next agent it points to.
- *CGenome* GetGenome():* accessor to get the genome this agent contains.
- *bool IsActive():* decide whether this agent is active or not.
- *void SetNext(CAgent*):* mutator to enable this agent keep an address of another one.
- *void SetActive(bool type):* Set this agent to be active or inactive.
- *SetGenome(CGenome*):* mutator to set this agent's genome.
- *bool Initialize():* initialize the agent by create an genome randomly.

- *bool IsWinner(int part, int total):* return whether this agent is winner during the combat based on the value of *part* and *total.*
- *int ResourceSize(int type):* return the number of specific type of resources the agent has, including reserves.
- *int Size():* returns the total number of all type of resources this agent have.
- *void AddToList(CAgentList* theList):* add this agent into a list specified by *theList.*
- *bool Mutation():* simulating the mutation of this agent.
- *void Kill(CAgent* agentB):* simulating this agent kill another one (*agentB*) by getting all resources of it and deleting it from memory.
- *void Eating():* simulating the agent eat food by transferring resources from the cell to it; if no food is available, then it moves to neighbor cell.
- *void PayTax():* simulating the agent pay tax by releasing some resources back to the cell where it lives.
- *void Die():* simulating the agent dies naturally by releasing all its resources to the cell where it lives and then being deleted from memory.
- *void Encounter(CAgent* theAgent):* implement an encounter between this agent and another one (*theAgent).*
- *void ContributeResources(CAgent* AgentB, CAgent* Child):* the number of resources specified by Child is taken from Reserves of both this agent and AgentB randomly, and a fraction (specified as a simulation parameter) of remaining Reserves of parent agents is transferred to this child.

- *CAgent* GiveBirth(CAgent* agentB):* this function will create a new agent object based on this agent and *agentB*. If successful, return a pointer to the new agent, otherwise return 0.
- *CSimString ConstructGene(CSimString& LeftGene, CSimString& RightGene):* construct a new gene based on *LeftGene* and *RightGene*, and return this new gene.
- *bool IsSurrender(int part, int total):* return whether this agent has surrendered or not before combat based on the value of *part* and *total*. If not, then the combat is unavoidable.
- *IsFightable(Cagent*):* decide whether this agent is threatening the other.
- *IsTradable(Cagent*):* decide whether these two agents can trade.
- *IsMatable(Cagent*):* decide whether these two agents can mate to reproduce a child.
- *void Move(DIRECTION direction):* Move to neighbor cell under specific direction and put this agent into the BeforeList of the new cell.
- *void Combat(CAgent*):* simulating this agent fight (combat) with another for life by killing the defeated one, taking over its resources and removing it from the environment.
- *void Trade(CAgent*):* simulating trading between these two agents by exchanging their resources.
- *void Reproduce(CAgent*):* simulating reproduction by creating a child agent and transfer part of parent's resource to it.
- *Friend ostream& operator << (ostream&, CAgent&):* operator overloading which enable to use "<<" operator on agent object directly.

Class CGenome

- *CGenome(Cgenome* gn):* constructor to create a genome (CGenome) object based on another genome pointed by *gn*.
- *~Cgenome():* destructor to destroy this object and remove from memory.
- *CSimString* GetAttack():* accessor which returns the pointer of *attack* gene inside of this genome.
- *CSimString* GetDefend():* accessor which returns the pointer of *defend* gene inside of this genome.
- *CSimString* GetBeauty():* accessor which returns the pointer of *Beauty* gene inside of this genome.
- *CSimString* GetCombat():* accessor which returns the pointer of *combat* gene inside of this genome.



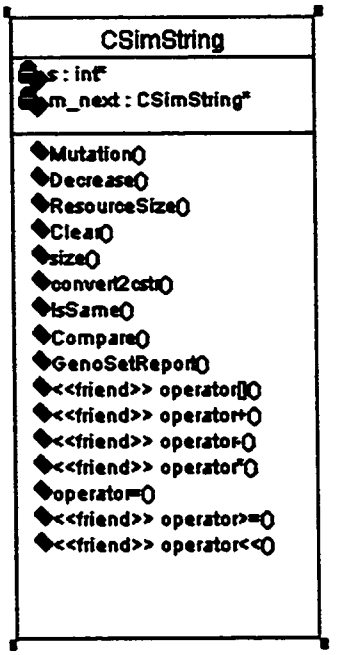
- *CSimString* GetTrade():* accessor which returns the pointer of *trade* gene inside of this genome.
- *CSimString* GetLust():* accessor which returns the pointer of *lust* gene inside of this genome.
- *CSimString* GetEat():* accessor which returns the pointer of *eat* gene inside of this genome.
- *CSimString* GetGive():* accessor which returns the pointer of *give* gene inside of this genome.
- *CSimString* GetRef():* accessor which returns the number of reference to this genome object.

- CGenome* GetNext(): accessor which returns the pointer of a genome to which this one point.
- *void SetRef(unsigned int ref):* mutator to change the number of reference to this genome object.
- *void SetNext(CGenome* pNext):* mutator to keep an address of another genome specified by pNext.
- *void Init():* genome initialization.
- *void IncRef();* increase the value of *m_uRef* data member by one.
- *void DecRef():* decrease the value of *m_uRef* data member by one.
- *int size():* return the total number of resources the genome have.
- *int compare(CGenome*):* compare the two genomes. If this genome is greater than the other, it returns a positive value; If it is less than the other, the function returns a negative value; If it is equal, the function returns zero.
- *Void GenoSetReport(ostream&):* this function writes genome sequence information into output stream.
- *CGenome & operator = (CGenome & aGenome):* operator overloading which enable to use assignment operator on CGenome object directly.

Class CSimString

- *CSimString(char * p):* Constructor to create a gene (CsimString object) based on the string *p*.
- *CSimString(int size):* Constructor to create a gene (CSimString object) based on the *size*.

- *CSimString(CSimString& theString):* Copy Constructor to create a gene based on another gene obejct.



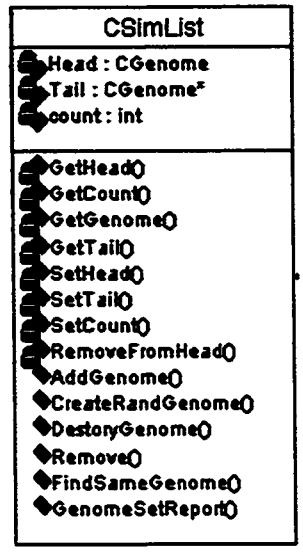
- *~CSimString():* Destructor to remove the object from memory.
- *void Mutation():* simulating the mutation inside this gene object.
- *void Decrease(int type, int number):* Decrease specified number of specific type of resource from this gene.
- *int ResourceSize(int type):* Return the number of specific type of resource inside this gene.
- *void Clear():* Set all resources number to zero.
- *int size():* Return the sum of the gene's resources.
- *void convert2cstr(char* buffer , int size):* Convert the gene sequence to a zero terminated string and store it at buffer.
- *friend bool transfer(CSimString &from, CSimString & to, CSimString & limit):* this function moves resources from one gene to another, subject to the given limit.
- *bool IsSame(CSimString& aString):* decide whether this gene have the same type and number of resources as the other has.
- *int Compare(CSimString& aString):* comparing the two genes, if this gene is greater than the other, return a positive value; if it is less than the other, return a negative value; if it is equal, return a zero.
- *void GenoSetReport(ostream&):* write the gene sequence into a output stream.

- *int & operator [] (int):* operator overloading which enable to use "[]" operator on CSimString object directly.
- *CSimString operator + (CSimString &):* operator overloading which enable to use "+" operator on CSimString object directly.
- *int operator – (CSimString &):* operator overloading which enable to use "-" operator on CSimString object directly.
- *int operator * (CSimString &):* operator overloading which enable to use "*" operator on CSimString object directly.
- *CSimString & operator = (CSimString &):* operator overloading which enable to use assignment operator on CSimString object directly.
- bool operator >= (CSimString &): operator overloading which enable to use ">=" operator on CSimString object directly.
- Friend ostream& operator << (ostream &, CSimString&): operator overloading which enable to use "<<" operator on CSimString object directly.

Class CSimList

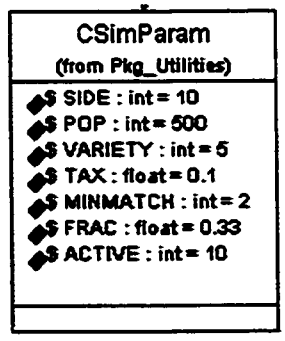
- *CGenome* GetHead():* accessor to get the first genome (CGenome) object in this genome pool.
- *int GetCount():* accessor to get the total number of genomes this pool contains.
- *CGenome* GetGenome(int i):* return the *i*th genome (CGenome object) in the pool.
- *CGenome* GetTail():* accessor to get the last genome object in the pool list.
- *void SetHead(CGenome*):* mutator to set the first genome in the pool.
- *void SetTail(CGenome*):* mutator to set the last genome in the pool.

- *Void SetCount(int):* mutator to modify the total number of genomes in the pool.
- *CGenome* RemoveFromHead():* remove the first genome from the pool.
- *CSimList():* constructor to create a genome pool.
- *Void AddGenome(CGenome*):* put the specified genome object into the pool.
- *CGenome* CreateRandGenome():* create a random genome. If there is a genome currently in genome pool that have the same gene sequence as the one just created, then return this existing one and destroy this created genome; otherwise, put this created genome into genome pool and return it.
- *void Remove(CGenome*):* remove a specified genome object from the genome pool.
- *CGenome* FindSameGenome(CGenome*):* find an genome object from the genome pool such that it has the same gene sequence as the specified one has.
- *void GenomeSetReport(ostream &):* write the genome sequence information of all genomes inside of this pool.

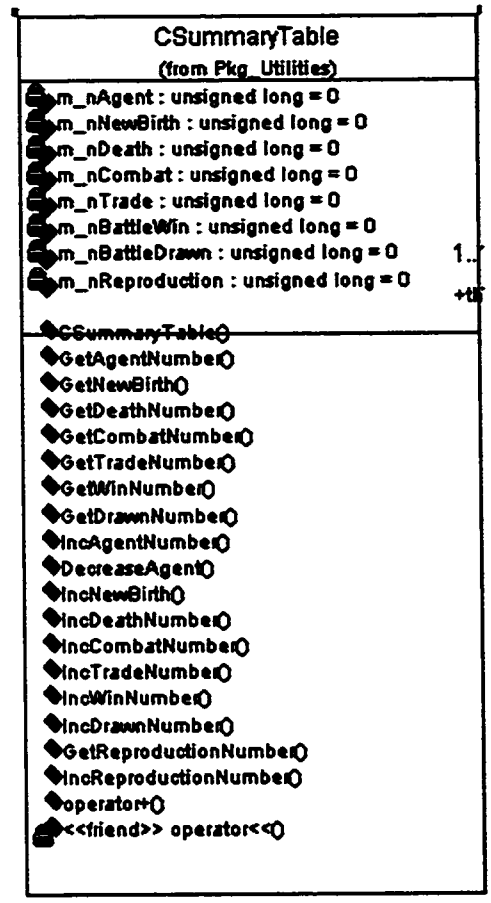


Class CSimParam

There are no member functions associated with this class. All data members are class data members. The sole purpose of creating this class is to centralize the simulation parameters.



Class CSummaryTable



- *CSummaryTable():* constructor
- *unsigned long GetReproductionNumber():* accessor to get the number of times the reproduction has occurred so far.
- *unsigned long GetAgentNumber():* accessor to get the total number of agents that are still alive.
- *unsigned long GetNewBirth():* accessor to get the number of new born agents so far.
- *unsigned long GetDeathNumber():* accessor to get the number of agents that have dead and destroyed so far.
- *unsigned long GetCombatNumber():* accessor to get the number of times the combat has happened so far.
- *unsigned long GetTradeNumber():* accessor to get the number of times the trading activities has happened so far.
- *unsigned long GetWinNumber():* accessor to get the number of winners for all the combats happened so far.
- *unsigned long GetDrawnNumber():* accessor to get the number of time the escape has happened so far.

- *void IncReproductionNumber(int count = 1):* increase the number of reproductions by count (default is 1).
- *void IncAgentNumber(int count = 1):* increase the number of agents by count (default is 1).
- *void DecreaseAgent(int count = 1):* decrease the number of agents by count (default is 1).
- *void IncNewBirth(int count = 1):* increase the number of births by count (default is 1).
- *void IncDeathNumber(int count = 1):* increase the number of dead agents by count (default is 1).
- *void IncCombatNumber(int count = 1):* increase the total number of combats by count (default is 1).
- *void IncTradeNumber(int count = 1):* increase the total number of trades taken place by count (default is 1).
- *void IncWinNumber(int count = 1):* increase the total number of winners for the combat by count (default is 1).
- *void IncDrawnNumber(int count = 1):* increase the total number of escapes by count (default is 1).
- *friend ostream& operator <<(ostream&, CSummaryTable&):* operator overloading which enables the "<<" operator to be used on *CSummaryTable* object directly.
- *CSummaryTable operator + (CSummaryTable&):* operator overloading which enables the "+" operator to be used on *CSummaryTable* object directly.

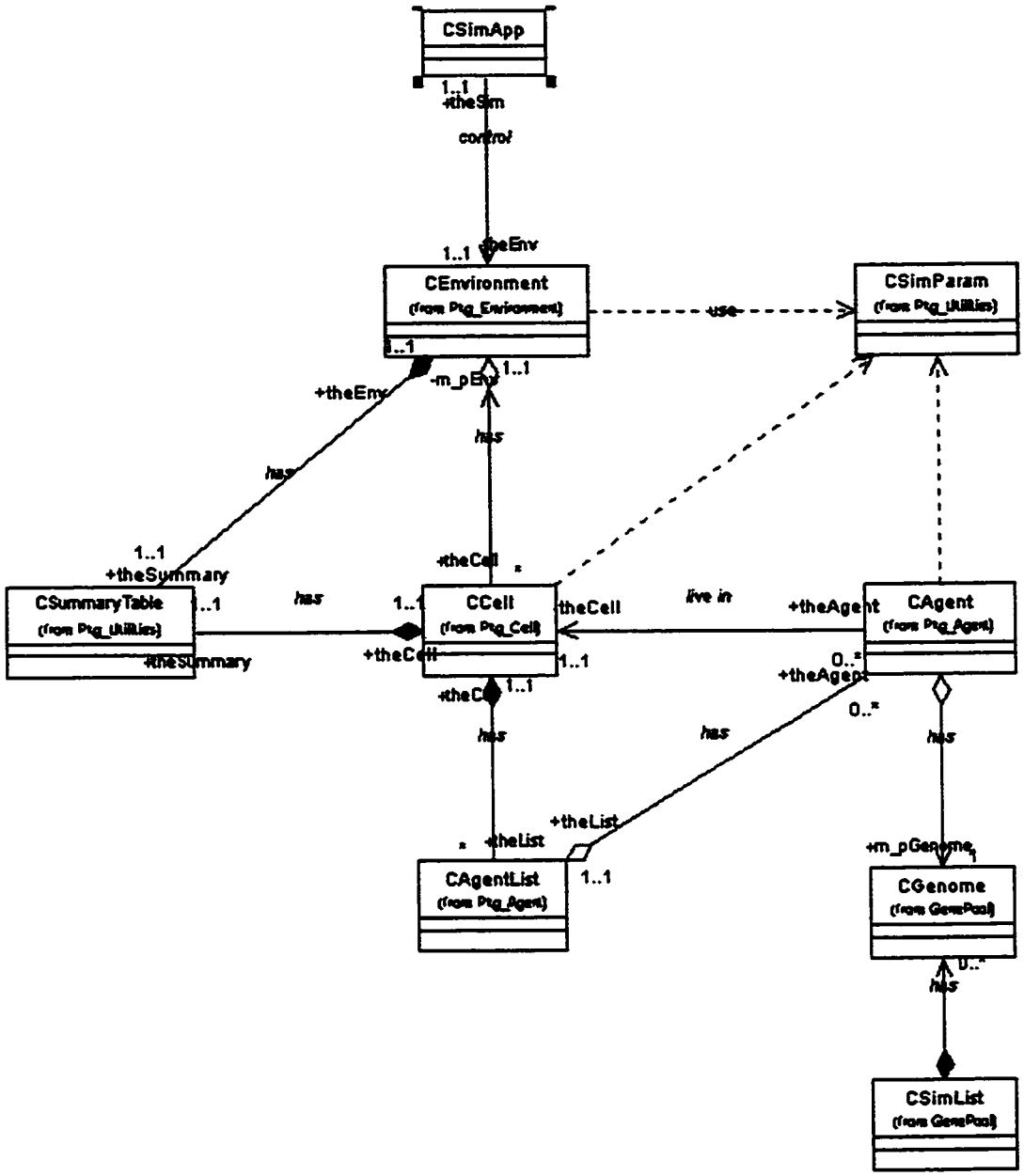


Figure 1: Class Diagram for Agent Simulation Application

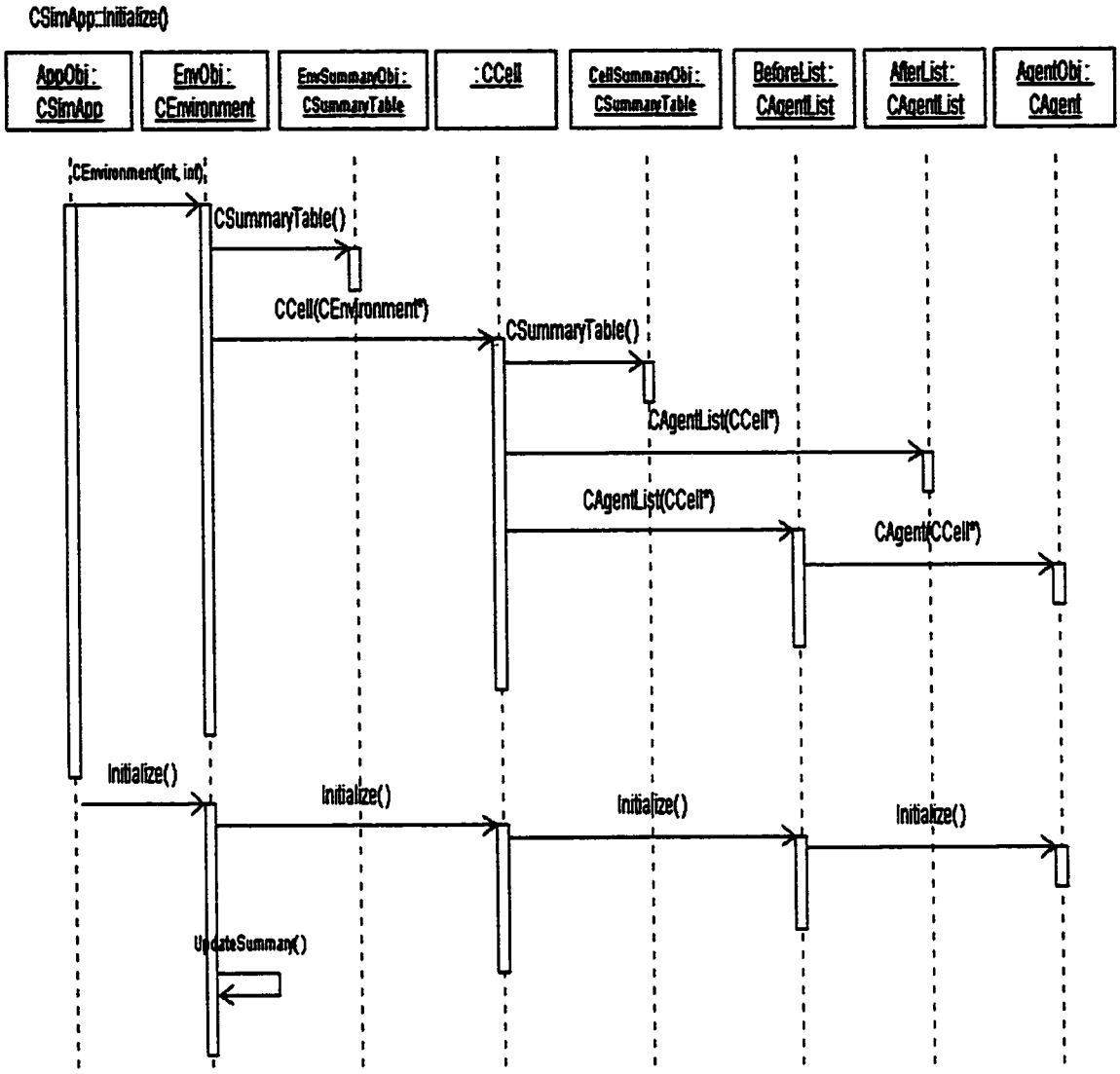


Figure 2: Sequence Diagram of CSimApp::Initialize()

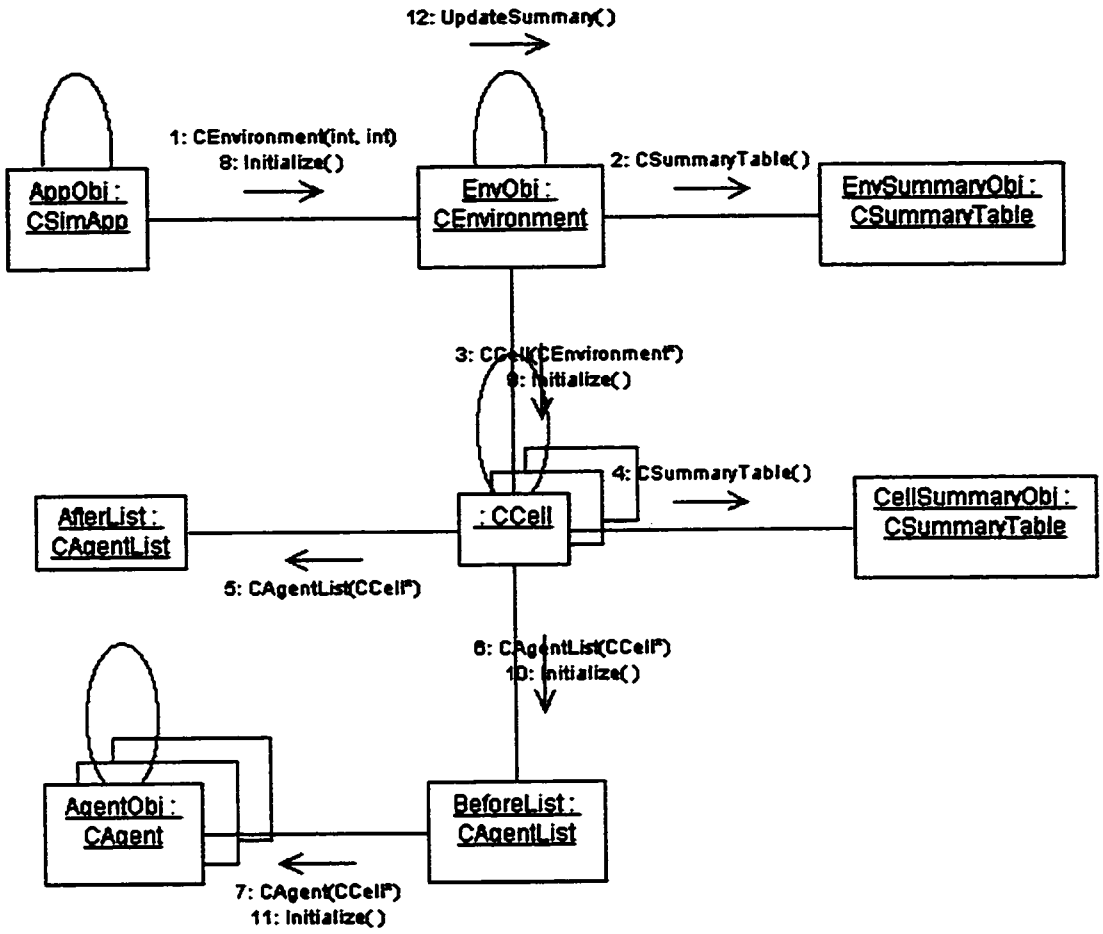


Figure 3: Collaboration Diagram of CSimApp::Initialize()

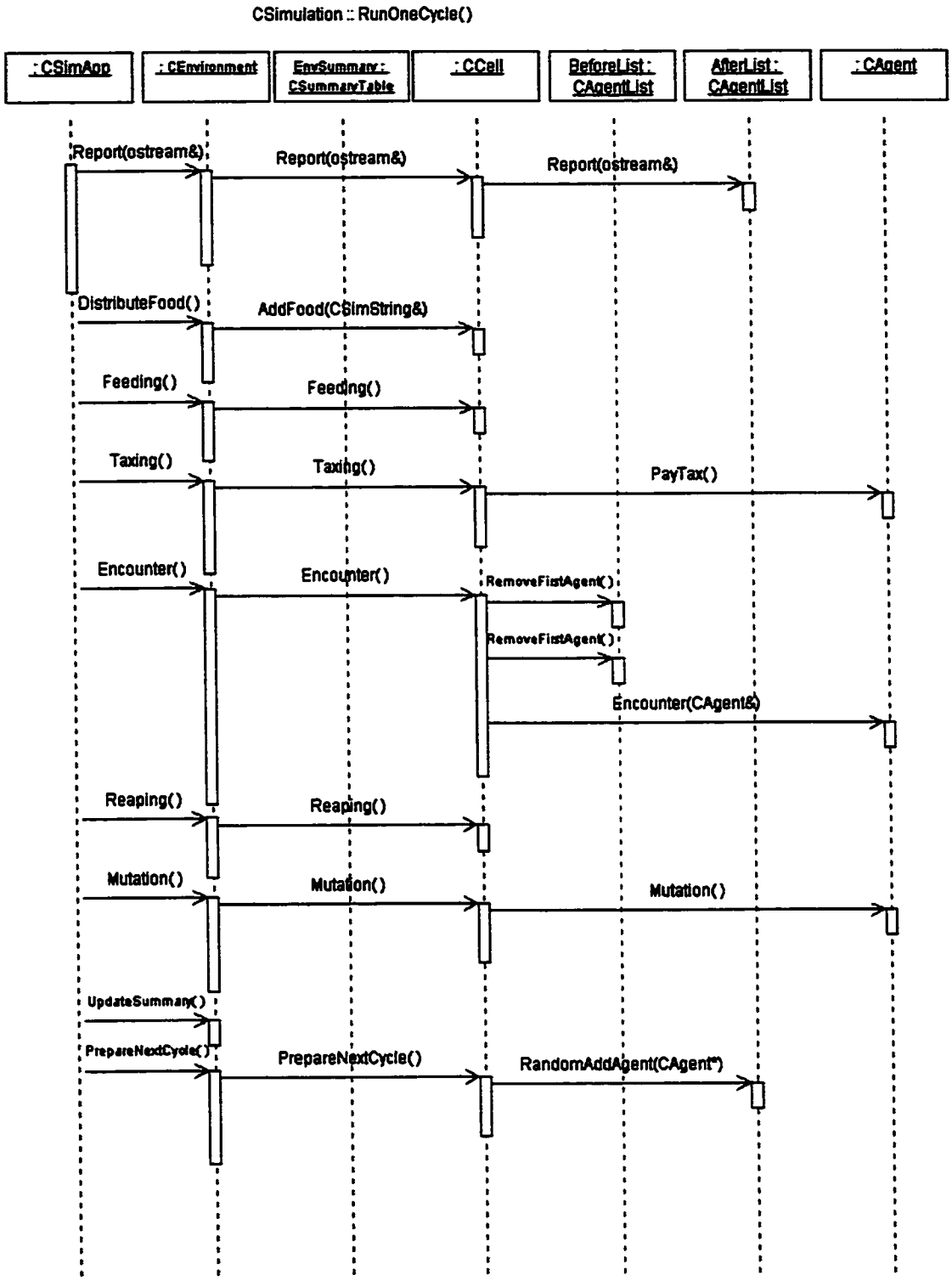


Figure 4: Sequence Diagram of CSimApp::RunOneCycle()

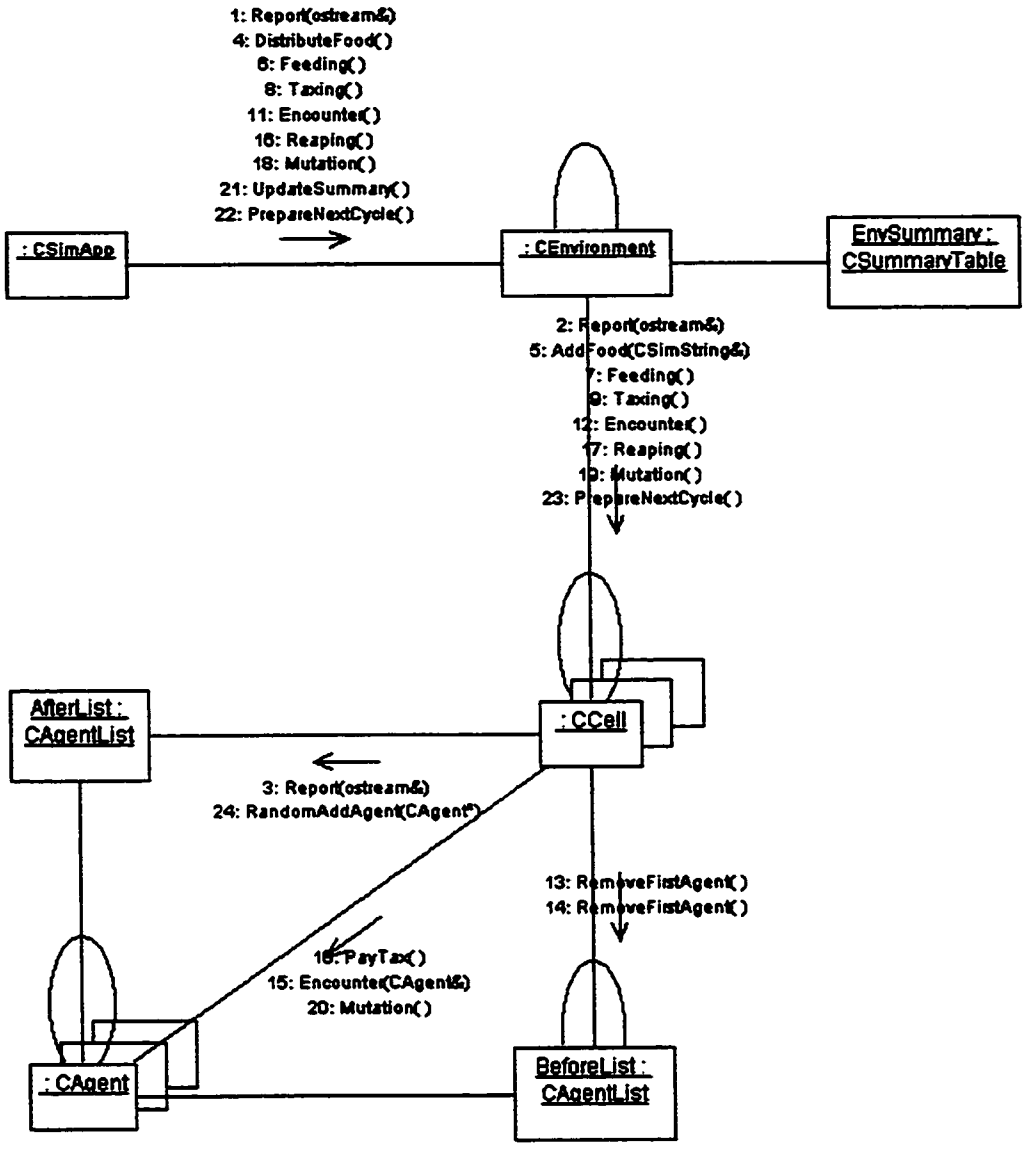


Figure 5: Collaboration Diagram of CSimApp::RunOneCycle()

Chapter 5 Result

During our development, we were challenged by several problems. One of them is that, after the application starts running all agents would die after a number of cycles no matter what simulation parameters we chose.

We investigated the problem and concluded that it was because we used a fixed sequence of agent interaction: fighting first, then trading, and reproducing at last. This sequence was based on Echo [2]. So the fighting had higher probability than trading and reproduction. Only if the combat was drawn, could trading and reproduction happen. But most agents were killed during combat and the agent population always decreased. Consequently, the agent population rapidly decreased to zero.

The solution we sought was to change the scenario of encounter. We gave a random sequence of interactions: when two agents encounter, the possibilities of fighting, trading and reproduction were made as even as possible. The application seems to have worked well since then.

Another interesting observation is that no agents actually share the same genome, contrary to what we assumed during the development. The best explanation could be drawn from real world experience. Identical twins do have identical genes; fraternal twins do not. Other than twins, the probability of identical human genomes is negligible.

In this chapter, we use two examples describing the results that the Agent Simulation is capable of producing. Note here that even if we use the same set of simulation parameter values to run the simulation, the results will be always different. The reason is that the interactions and the initialization of agents are implemented as randomly as possible.

Both examples use the same following parameters:

- SIDE = 5
- POP = 500
- VARIETY = 5
- TAX = 0.1
- MINMATCH = 2
- FRAC = 0.33
- ACTIVE = 10

Example 1

The first example will present a simulation process using all the default simulation parameters. The default simulation parameters and their values are shown in Section 0. The summary table displays every 50 simulation cycles.

Step 1. Start the program:

The user interface starts as shown in Figure 6. It displays "Simulation is stopped" in the middle of the window.

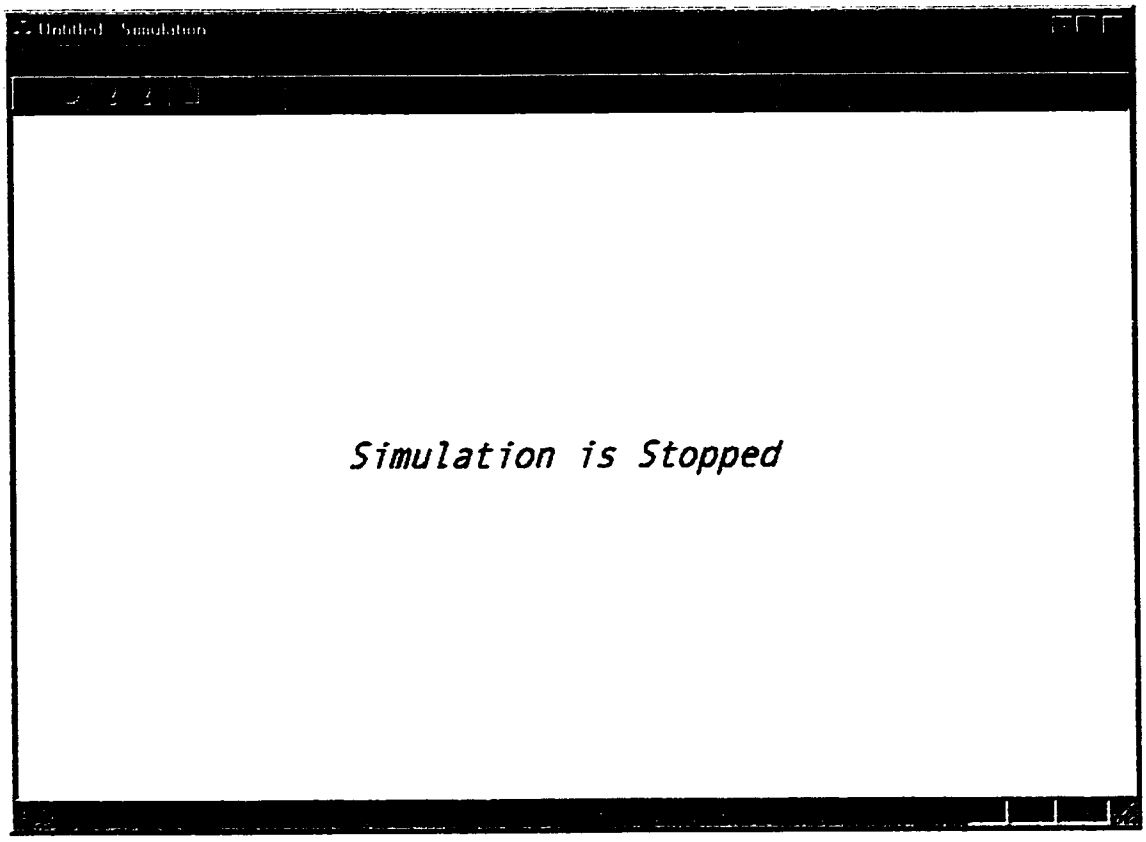


Figure 6: before simulation start / after simulation stop

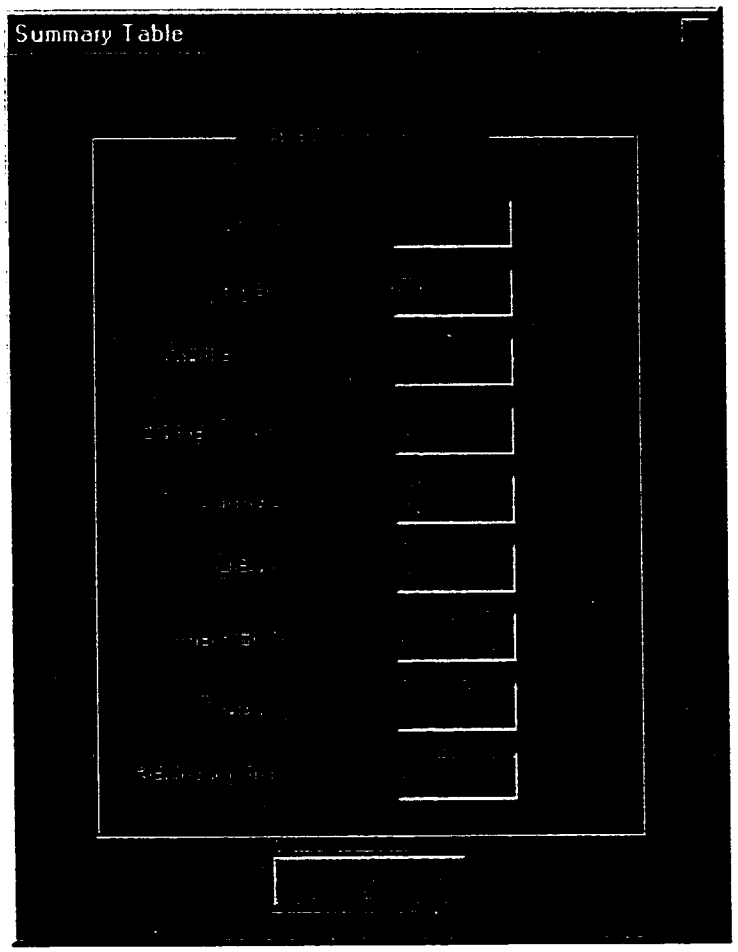


Figure 7: Initial Statistics of the simulation

Step 2. Checking default simulation parameters:

The user clicks on menu Setup -> Simulation Parameter, the Simulation Parameter dialog box is shown with all default parameters on it. Click OK. The dialog box disappears.

Step 3. Start the simulation:

The user clicks on the menu Simulation -> Start or the Toolbar button to start the simulation. The message on the windows changes to "Starting the simulation, Please wait..."

Step 4. The summary table dialog box is shown with the initial statistic information as shown in Figure 7.

Step 5. The user clicks on OK to hide the summary table dialog box. The message on the window shows "Simulation is running".

Step 6. After every 50 cycles, a new summary table dialog box appears. The user observes the statistics and click on OK button.

Step 7. After about 150 simulation cycles, the agent number start to increases and the trend grows dramatically along with the simulation continues. The user stops the program by clicking on the "stop" button on the toolbar.

The simulation result is captured in after 50 cycles is shown in Figure 8.

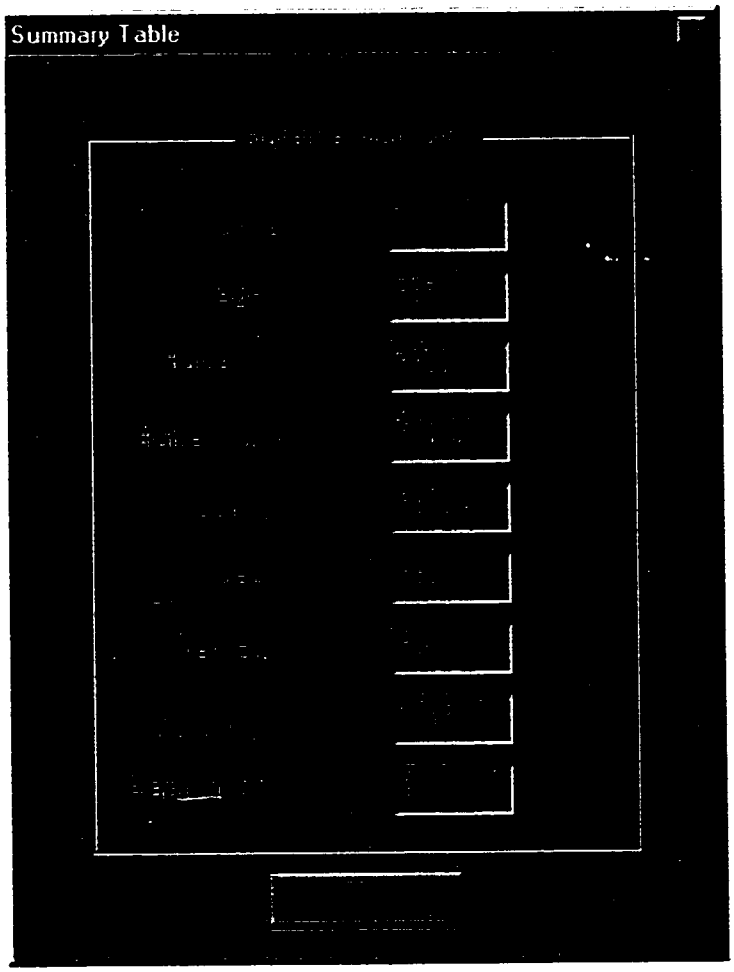


Figure 8: Simulation Statistics after 50 cycles for Example 1

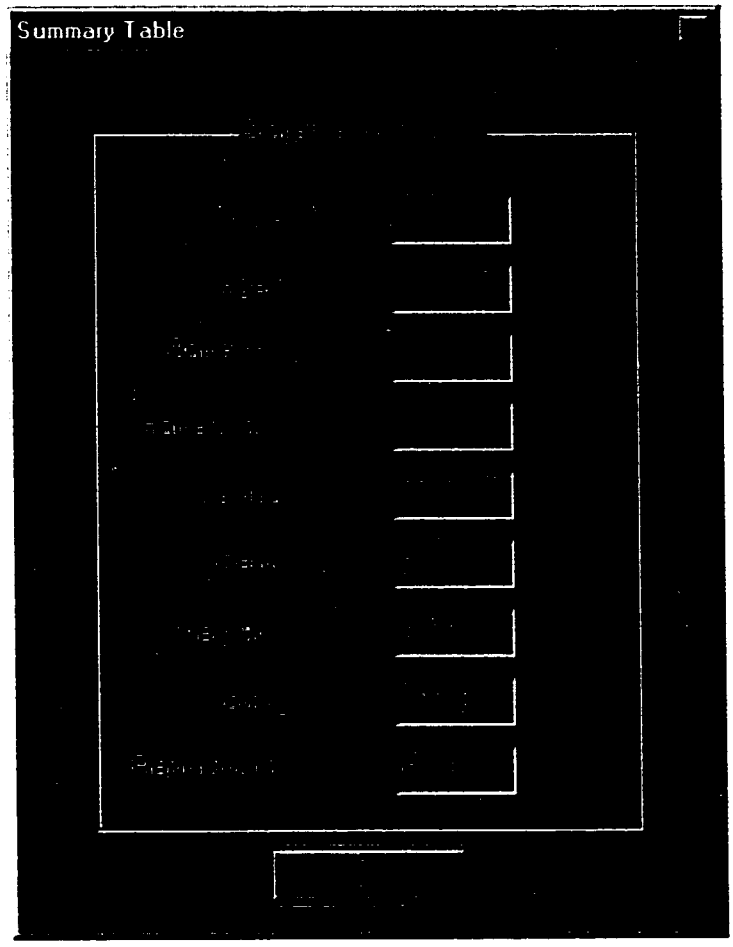


Figure 9: Simulation statistics after 100 cycles in Example 1

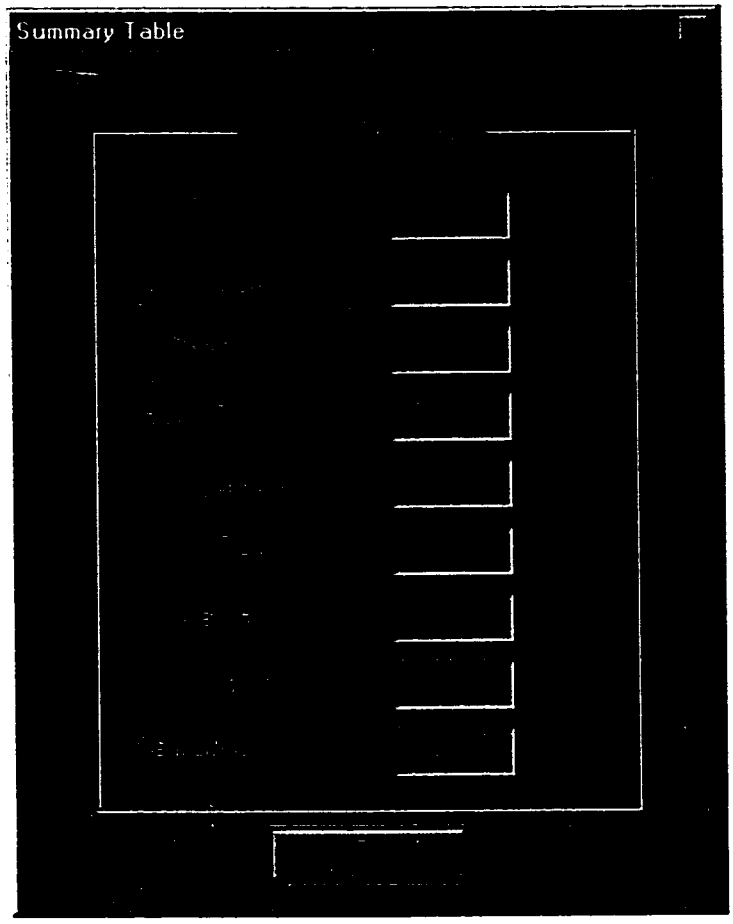


Figure 10: Simulation statistics after 150 cycles in Example1

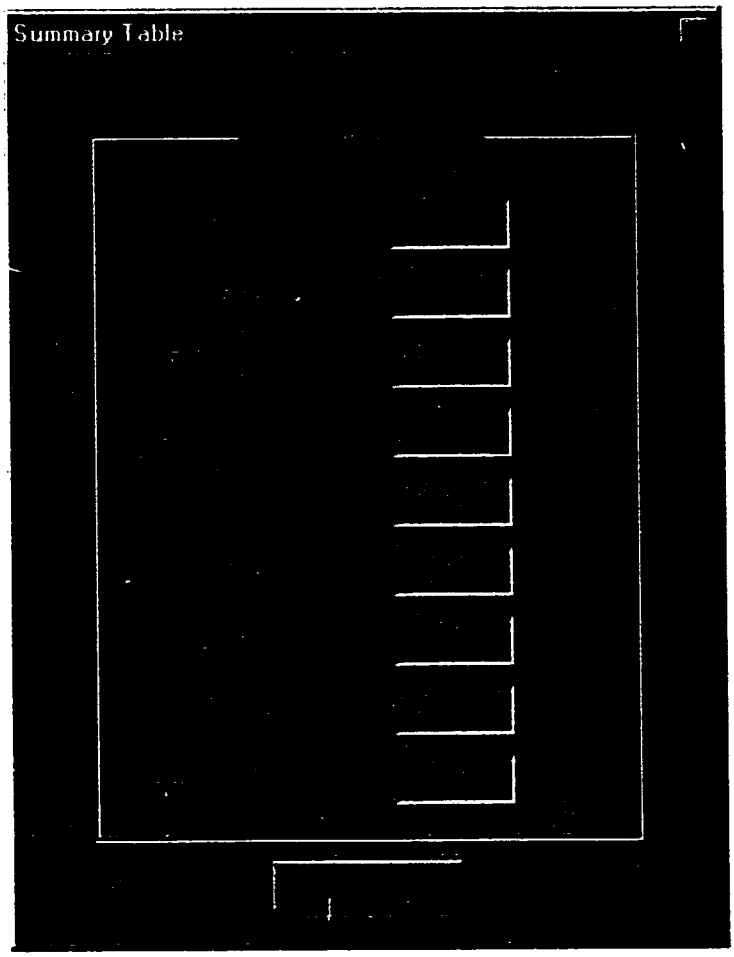


Figure 11: Simulation statistics after 200 cycles in Example 1

Initially, the total number of agents in the environment was 10*10*500=50000. After 50 simulation cycles, as shown in Figure 8, even there were 40252 times of reproduction happened, only 30614 new agents were produced. That means some parent agents did not have enough resource to produce a child. "73563 battle win" means 73563 agents were killed after fighting with other agents. "71699 battle drawn" means that there is 71699 life or death battles were avoid and they might proceed to trade or reproduce or nothing.

After another 50 cycles, the number of agents stabilized. Fighting did not happen a lot but trading and reproduction did. The situation starts to stabilize. The agents begin to tolerate each other and live in peace. This was determined by their genes.

Figure 10 shows the number of agents started to increase at the end of 150 simulation cycles. The living situation is improved and stabilized. The survival agents are used to living with others.

Figure 11 shows from 150 to 200 simulation cycles, the number of agents starts to grow.

Trading and reproduction happened frequently. To continue running the simulation will take a long time and the PC memory could hardly handle the large number of agents.

Example 2

With the second example will present a simulation process using some user entered simulation parameters:

- POP = 300: the initial population is decreased
- TAX = 0.2: the tax rate is increased

We follow the procedures as in example 1.

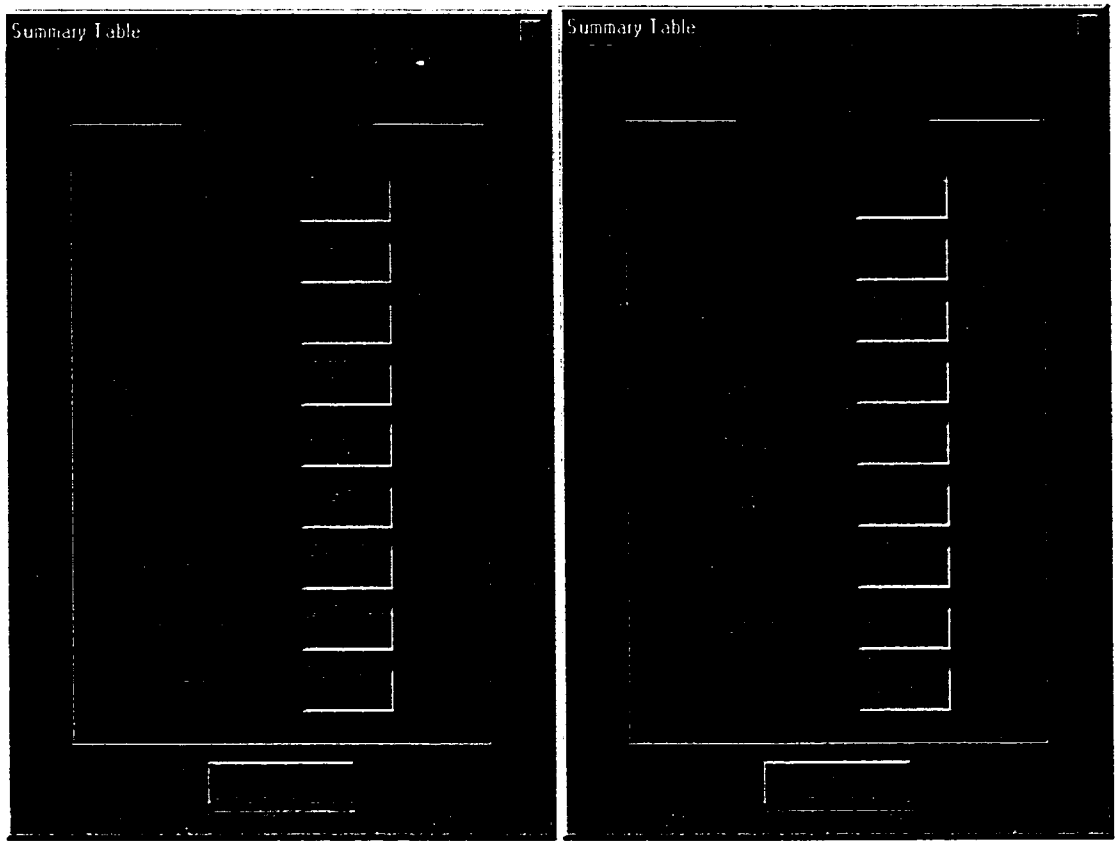


Figure 12: Simulation Statistics after cycle 50 and 100 in Example 2

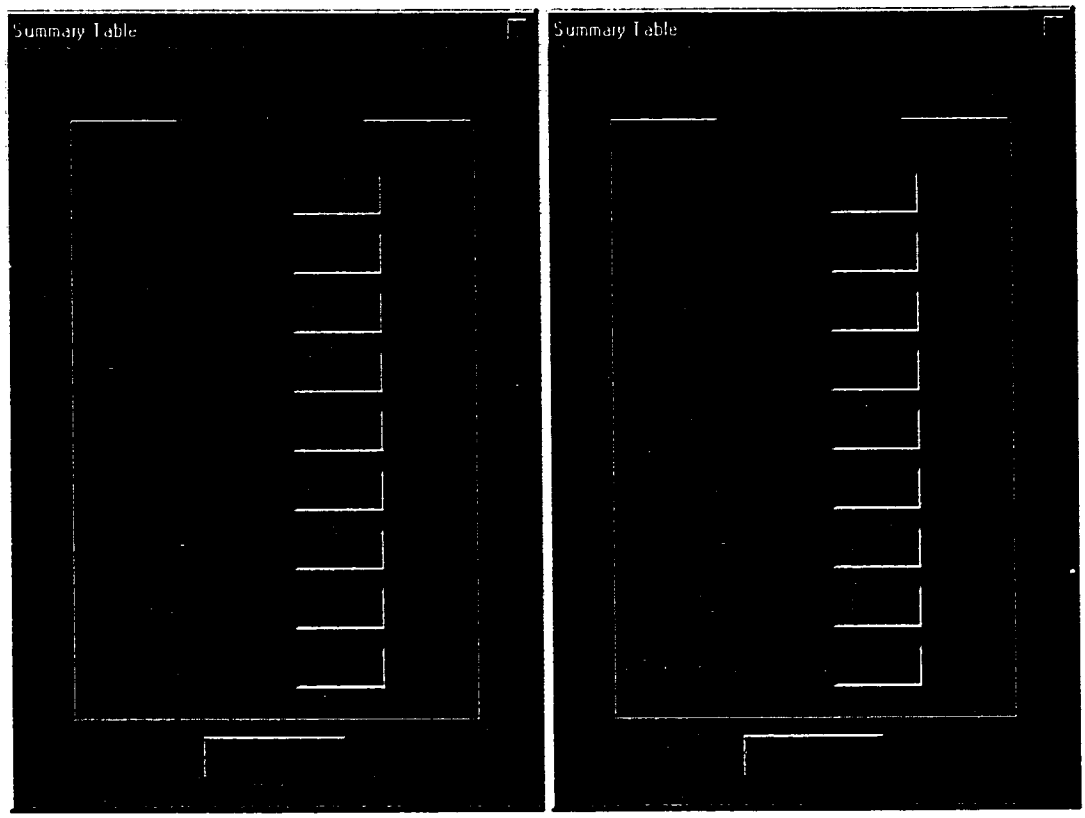


Figure 13: Simulation Statistics after cycle 200 and 350 in Example 2

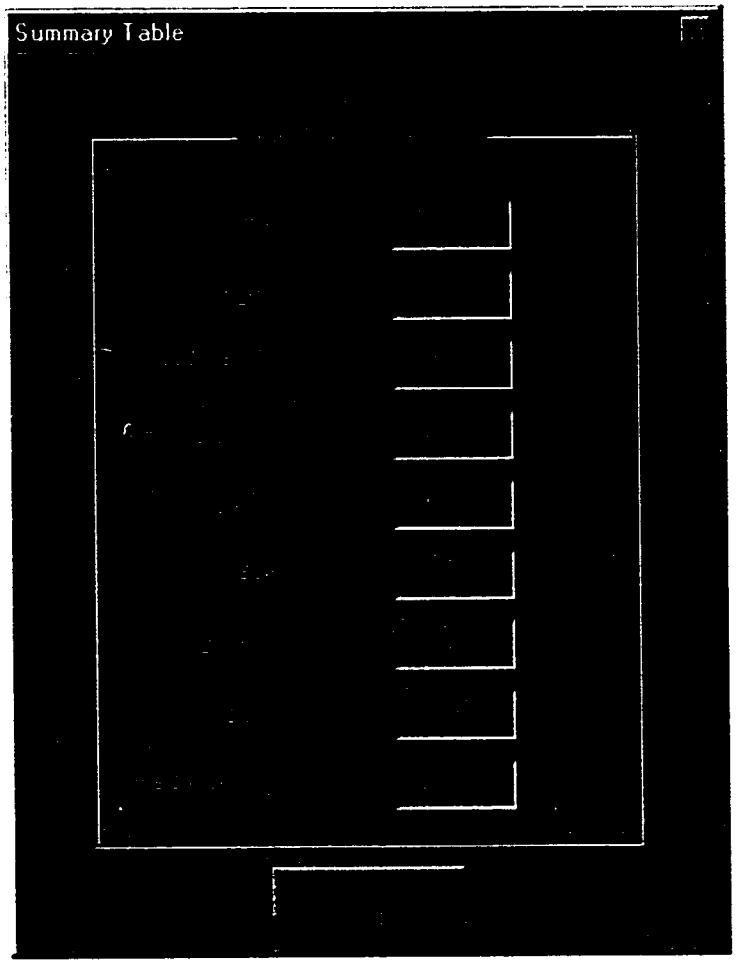


Figure 14: Simulation Statistics after cycle 1000 in Example 2

From above figures we see that since tax rate was increased, agents did not have that much extra resources to produce new agents. But because the survived agents were very strong (hard to die during fighting), the agent number decreased very slowly.

Chapter 6 Conclusion

Running the two examples above demonstrates that the implementation of Agent Simulation, the graphical user interface and the Help system perform well and meet all our requirements.

It offers users a convenient and efficient way to observe the interactions among agents.

Experience on Object orient Programming

Traditional programming languages separated data from functionality. Typically, data was aggregated into structures that were then passed among various functions that created, read, altered, and otherwise managed that data.

C++, as an Object-oriented language, is concerned with the creation, management, and manipulation of objects. An object encapsulates data and the methods used to manipulate that data.

Object-oriented programming offers a new and powerful model for writing computer software.

It speeds the development of new programs, improves the maintenance, reusability, and modifiability of software. Object-oriented programming focuses on the creation and manipulation of objects, such as agents, agent lists, cells, and environments. This type of programming gives us a greater level of abstraction; we, the programmers can concentrate on how the objects interact without having to focus on the details of the implementation of the object.

We used a genome pool in which identical genomes could be shared by different agents. It turned out that there were no identical genomes and sharing did not take place. This is

similar to the situation in a human population where no two people (apart from identical twins) have the same genome.

Future Improvement

In this project we presented the design and implementation of the Agent Simulation System with Object-oriented methodology and Microsoft Visual C++ toolkits. One of the limitations of the system is that it only provides the results (as numbers) of the whole environment.

If user wants to learn the statistics on a cell basis, he/she has to read the result file. Therefore we suggest the following extensions:

The details of simulation information can be set in different levels based on the user's interest in the future version

- The mutation probability is hard coded in the application. It may be treated as a simulation parameter in the future version.
- The user will be able to select simulation results of any individual cell from the interface.
- The simulation results can be shown as graphical charts or curves, which are more challenging but more user friendly.
- Tools will be provided to analyze the simulation results for user.

Chapter 7 Bibliography

1. Howard T., Odum *System Ecology*: In Introduction, P4. A Wiley_Interscience Publication 1983.
2. Peter T. Hraber, Terry Jones and Stephanie Forrest, *The Ecology of Echo,* Artificial Life Volume 3, Number 3, P165–190. Massachusetts Institute of Technology 1997.
3. S.E. Jorgensen, *Fundamentals of Ecological Modelling*, P106. Elsevier 1994.

4) S.E. Jorgensen, *Fundamentals of Ecological Modelling*, P14. Elsevier 1994.

5) Silvert, W., *Object-oriented ecosystem modelling*, Ecol. Modelling, submitted 1993.

6) Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, P196. Addison Wesley 1994.

7) Straskraba, M., *Cybernetic-categories of ecosystem dynamics*, P81. ISEM-Journal, Vol.2, 1980.

8) Changjiang Zhang, *Agent Simulation Using Object Oriented Methodology*, Concordia University, submitted 2000.

Appendixes

CSimApp.h

```
#ifndef _INC_CSIMAPP_3825DA4A0064_INCLUDED
#define _INC_CSIMAPP_3825DA4A0064_INCLUDED

#include <fstream.h>

class CSummaryTable;
class CEnvironment;

//the element which represents the simulation
class CSimApp
{
public:
      CSimApp();          //constructor
      ~CSimApp();         //destructor

      int GetCycleNumber();
      CSummaryTable* GetSummaryTable();
      void Report(ostream& fout);
      void Initialize();
      void RunOneCycle();

private:
      CEnvironment* theEnv;

      //the simulation is currently pause or not.
      bool m_bPause;

      //the reference of a file into which the report will be
      //written
      ofstream m_file;
};

#endif /* _INC_CSIMAPP_3825DA4A0064_INCLUDED */
```

CEnvironment.h

```
#ifndef _INC_CENVIRONMENT_380E51A60208_INCLUDED
#define _INC_CENVIRONMENT_380E51A60208_INCLUDED

#include "Utility.h"

class CSummaryTable;
class ostream;
class CSimParam;
class CCell;

//simulation element to represent the environment where the all
//agents live in the simulation.
class CEnvironment
{
public:
      CEnvironment(int x = 1, int y = 1); //Constructor
      ~CEnvironment();                    //Destructor

      CSummaryTable* GetSummaryTable();         //accessor
      CCell* GetNeigbor(CCell*, DIRECTION);     //accessor
      int GetCycle();                           //accessor

      void SetCycle(int cycle);           //mutator

      //enable all agents in this environment to start gene mutation.
      void Mutation();

      // update the statistic information.
      void UpdateSummary();

      void Initialize();
      void PrepareNextCycle();

      //simulation behavior
      //simulating some food (resource) being distributed in
      //this environment.
      void DistributeFood( );

      //Remove some inactive agents from this environment.
      void Reaping();

      //write statistic information into stream.
      void Report(ostream& fout);

      //simulation behavior
      //smulating agents eat some food (resource) from this
      //environment.
      void Feeding();

      //simulation behavior
      //simulating all agents in this environment to pay tax for
      //right to live
      void Taxing();
```

```
        //simulation behavior
        //simulating the encounter behavior for all agents in
        //this environment.
        void Encounter();

        //write all different genomes which agents contain into
        //stream.
        void GenomeSetReport(ostream &fout);

private:
        //number of cells the environment contains
        int m_nCell;

        //the array of cells in this environment
        CCell* m_CellArray;

        CSummaryTable * m_Summary;

        //Number of cycles have been run so far.
        int m_nCycle;

        //return the pointer to m_CellArray[x][y]
        CCell* Cell(int x, int y);
};

#endif /* _INC_CENVIRONMENT_380E51A60208_INCLUDED */
```

CCell.h

```
#ifndef _INC_CCELL_380527E40276_INCLUDED
#define _INC_CCELL_380527E40276_INCLUDED

#include "CSummaryTable.h"

class CSummaryTable;
class ostream;
class CSimString;
class CSimParam;
class CAgentList;
class CEnvironment;

//simulation element to represent the site in which some agents live.
class CCell
{
public:
      CCell(CEnvironment *pEnv = 0);       //constructor
      ~CCell();                            //destructor

      CEnvironment* GetEnvironment();      //accessor
      CSummaryTable* GetSummaryTable();    //accessor
      CAgentList* GetAfterList();          //accessor
      CAgentList* GetBeforeList();         //accessor
      CSimString & GetFood();              //accessor

      void SetEnvironment(CEnvironment* pEnv);  //mutator
      void SetFood(CSimString & fd);            //mutator

      void Initialize();

      //write statistic information of this cell into stream.
      bool Report(ostream& fout);

      //put all agents in this cell randomly back into beforelist
      void PrepareNextCycle();

      //return the neighbor cell
      CCell* GetNeighbor(DIRECTION theDirection);

      //simulation behavior
      //asking all agents in this cell to do gene mutation.
      void Mutation();

      //simulation behavior
      //forcing all agent in this cell to pay tax by transfer
      //some resource back to the cell.
      void Taxing();

      //simulation behavior
      //enable all agents in this cell to start encounter each other.
      void Encounter();
```

```
        //remove inactive agents in this cell by delete it from
        //memory and reset others active flag.
        void Reaping();

        //simualtin behavior
        //distribute some food (resource) into this cell
        void AddFood(CSimString & fd);

        //simulation behavior
        //enable all agents in this cell to eat some food
        void Feeding();

private:
        CEnvironment* m_pEnv;
        CSummaryTable m_Summary;
        CSimString m_food;

        //agents haven't done encounters are in this list in
        //current cycle
        CAgentList * BeforeList;

        //the agents have done encounters are in this list in
        //current cycle
        CAgentList * AfterList;

};

#endif /* _INC_CCELL_380527E40276_INCLUDED */
```

CAgentList.h

```
#ifndef _INC_CAGENTLIST_38190528029E_INCLUDED
#define _INC_CAGENTLIST_38190528029E_INCLUDED

class CAgent;
class CCell;

//the list which contains agents node
class CAgentList
{
public:
      CAgentList();                          //constructor
      ~CAgentList();                         //Destructor

      CAgent* GetHeader();                   //accessor
      int GetCount();                        //accessor
      void SetHeader(CAgent* theAgent);      //mutator
      void SetCount(int count);              //mutator

      //initilize the list
      void Initialize(CCell* );

      //add the agent into the head of this list
      void AddAgent(CAgent* theAgent);

      //Remove the first agent node from the list
      CAgent* RemoveFirstAgent();

      //remove the specified agent node from this list
      void RemoveAgent(CAgent* theAgent);

      //Add theAgent into random position of this list
      void RandomAddAgent(CAgent* theAgent);

private:
      CAgent* m_pHeader;
      int m_nCount;

};

#endif /* _INC_CAGENTLIST_38190528029E_INCLUDED */
```

CAgent.h

```
#ifndef _INC_CAGENT_37BABCFC02AF_INCLUDED
#define _INC_CAGENT_37BABCFC02AF_INCLUDED


#include "Utility.h"
#include "CSimString.h"

class CCell;
class ostream;

class CSimParam;
class CAgentList;

//The core element to reprent agent in the environment.
class CAgent
{
public:

      CAgent(CCell *aCell = 0);      //constructor
      ~CAgent();                     //destructor
      CCell* GetCell();              //accessor
      CAgent* GetNext();             //accessor
      CGenome* GetGenome(void);      //accessor
      bool IsActive();               //accessor


      void SetNext(CAgent* theAgent);      //mutator
      void SetActive(bool type);           //mutator
      void SetGenome(CGenome*);            //mutator


      //initilize this agent by creating a genome and reservoir.
      bool Initialize();

      //decide this agent wins or not in the combat based
      //on the possibility of part/total.
      bool IsWinner(int part, int total);

      //returns the total number of particular resource this
      //agent contains.
      int ResourceSize(int type);

      //returns the total number of all resources this agent contains.
      int Size();

      //put this agent into the front of thelist.
      void AddToList(CAgentList* theList);

      //simulation behavior
      //simulating agent's gene mutation by random change its gene
      //sequence and/or number.
      bool Mutation();
```

```
	//Simulation behavior
	//simulating one agent killing another by getting all its
	//resources and delete it from memory.
	void Kill(CAgent* agentB);

	//simulation behavior
	//simulating the agent eating food by transferring
	//resources from cell to the agent. If no food available,
	//then it moves to neighbor cell.
	void Eating();

	//simulation behavior
	//simulating the agent paying tax by releasing some
	//resource to the cell in which it stays.
	void PayTax();

	//Simulation behavior
	//simulating the agent dies naturally by releasing its all
	//resource to the cell in which it stays and then delete
	//from memory.
	void Die();

	//simulation behavior
	//simulating the encounter between the two agents.
	void Encounter(CAgent* theAgent);

	//operator overloading
	CAgent &  operator=(CAgent& theAgent);
	friend ostream& operator<<(ostream& out, CAgent& theAgent);

private:
	//the number of resources specified by Child is taken
	//from Reserves of both this agent and AgentB randomly.
	//and 1/3 remaining Reserves of both this and AgentB is
	//moved to Child.
	void ContributeResources(CAgent* AgentB, CAgent* Child);

	//create a child agent based on this agent and agentB.
	//If succeed, return a pointer of the new agent,
	//ohterwise, return 0.
	CAgent* GiveBirth(CAgent* agentB);

	//construct a new gene based on LeftGene and RightGene,
	//return CString object as the new gene.
	CSimString ConstructGene(CSimString& LeftGene, CSimString&
RightGene);

	//decide whether this agent is surrender during the combat
	//based on the part/total possibility.
	bool IsSurrender(int part, int total);

	//decide whether these two agents will fight (combat)
	//each other or not.
	bool IsFightable(CAgent* theAgent);

	//decide whether this agent is threaten the other (theAgent).
	bool IsThreaten(CAgent* theAgent);
```

```
    //decide whether these two agent can trade or not.
    bool IsTradable(CAgent* theAgent);

    //decide whether the two agents can mate to reproduce.
    bool IsMatable(CAgent* theAgent);

    //Simualtion behavior
    //simulating agent movement by moving to neighbor cell
    //under specific direction
    void Move(DIRECTION direction);

    //Simulation behavior
    //simulating one agent fight (combat) with another to live by
    //killing the defeated one, taking over its reources and
    //removing it from theenvironment.
    void Combat(CAgent* agentB);

    //simulation behavior
    //simulating trading between these two agets by exchanging
    //their resources in Reserve gene.
    void Trade(CAgent* agentB);

    //simulation behavior
    //simulating reproduction by creating a child agent and
    //transfer part of parent's resource to it.
    void Reproduce(CAgent* agentB);

    bool m_bActive;
    CGenome * m_pGenome;
    CSimString m_strReserve;
    CAgent* m_pNext;
    CCell* theCell;
};

#endif /* _INC_CAGENT_37BABCFC02AF_INCLUDED */
```

CGenome.h

```
#ifndef _CGENOME_H_
#define _CGENOME_H_

class CSimString;

//the element which represent the genome sequence each agent contains
//many different agents may have same genome sequence.
class CGenome
{
private:
      //number of agents which have this same genome sequence
      unsigned int m_uRef;

      CSimString m_strAttack;        //attach gene
      CSimString m_strDefend;        //defend gene
      CSimString m_strBeauty;        //beauty gene
      CSimString m_strCombat;        //combat gene
      CSimString m_strTrade;         //trade gene
      CSimString m_strLust;          //lust gene
      CSimString m_strEat;           //eat gene
      CSimString m_strGive;          //give gene

      CGenome * m_pNext;

public:
      CGenome(CGenome* gn = 0);            //constructor
      ~CGenome();                          //destructor

      friend class CSimString;

      CSimString* GetAttack();             //accessor
      CSimString* GetDefend();             //accessor
      CSimString* GetBeauty();             //accessor
      CSimString* GetCombat();             //accessor
      CSimString* GetTrade();              //accessor
      CSimString* GetLust();               //accessor
      CSimString* GetEat();                //accessor
      CSimString* GetGive();               //accessor
      unsigned int GetRef(void);           //accessor
      CGenome* GetNext(void);              //accessor

      void SetRef(unsigned int ref);       //mutator
      void SetNext(CGenome* pNext);        //mutator

      //return if these two genome have same sequence
      bool IsSame(CGenome *aGenome);

      //clone and return another genome based on this genome
      CGenome* Clone();

      void Init();
      void IncRef();
      void DecRef();
```

```
        int size();
        int Compare(CGenome*);


        //write gene sequence into stream
        void GenoSetReport(ostream &fout);

        //operator overloading
        CGenome & operator=(CGenome & aGenome);
};

#endif
```

CSimString.h

```
#ifndef _INC_CSTRING_37FD42B002F8_INCLUDED
#define _INC_CSTRING_37FD42B002F8_INCLUDED

#include <fstream.h>
class CSimParam;


//the element to represent gene.
class CSimString
{
public:

      CSimString(char * p);                //Constructor
      CSimString(int size = CSimParam::VARIETY);      //construtor
      CSimString(CSimString& theString);  //copy construtor
      ~CSimString();                       //destructor

      //simalulting the gene mutation
      void Mutation();

      //Decrease specified number of specified resource
      void Decrease(int type, int number);

      //return the number of specified type of resource
      int ResourceSize(int type);

      //empty its resources.
      void Clear();

      //return total number of all resources this gene string contains
      int size();

      //convert this gene string to char string and put it into
      //where the buffer point.
      void convert2cstr(char* buffer , int size  );

      //transfer resources from "from" gene string to "to" string
      //based on restriction of string "limit", return true if any
      //resource being transferred.
      friend bool transfer(CSimString & from, CSimString & to, CSimString
 & limit);

      //return true if both genes have same number of different
      //resources
      bool IsSame(CSimString& aString);

      int Compare(CSimString& aString);

      //write gene string into stream
      void GenoSetReport(ostream&);

      //operator overloading
      int & operator[](int r);
```

```
        CSimString operator+(CSimString & x);
        int operator-(CSimString & x);
        int operator*(CSimString & x);
        CSimString & operator=(CSimString & theString);
        bool operator>=(CSimString & x);
        friend ostream& operator<<(ostream& out, CSimString& theString);

private:
        int * s;
        CSimString *m_next;
};

#endif /* _INC_CSTRING_37FD42B002F8_INCLUDED */
```

CSimList.h

```
#ifndef _CSIMLIST_H_
#define _CSIMLIST_H_

//list data structure represents genome pool in this simulation.
class CSimList
{
private:
      CGenome *Head;
      CGenome *Tail;
      int Count;

      CGenome* GetHead(void);        //accessor
      int GetCount();                /accessor
      CGenome* GetGenome(int);       //accessor
      CGenome* GetTail(void);        //accessor

      void SetHead(CGenome*);        //mutator
      void SetTail(CGenome*);        //mutator
      void SetCount(int);            //mutator

      CGenome* RemoveFromHead();
public:
      CSimList();
      void AddGenome(CGenome*);
      CGenome* CreateRandGenome();
      void DestoryGenome(CGenome*);
      void Remove(CGenome*);
      CGenome* FindSameGenome(CGenome*);
      void GenomeSetReport(ostream &fout);
};

#endif
```

CSimParam.h

```
#ifndef _INC_CSIMPARAM_381B1A0102DA_INCLUDED
#define _INC_CSIMPARAM_381B1A0102DA_INCLUDED

class CEnvironment;

//all key simulation parameter are grouped in this class
class CSimParam
{
public:
      //The lasted number of cycles to measure an agent is
      //active or not.
      static int ACTIVE;

      //the Grid number SIDExSIDE
      static int SIDE;

      //the number of agents
      static int POP;

      //number of resource type
      static int VARIETY;

      //parameter used for taxing to agent by cell
      static float TAX;

      //the ninimum trading willingness.
      static int MINMATCH;

      //the fraction of the remaining reserves of parent after
      //reproduction.
      static float FRAC;

private:
};

#endif /* _INC_CSIMPARAM_381B1A0102DA_INCLUDED */
```

CSummaryTable.h

```
#ifndef _INC_CSUMMARYTABLE_38222EB00276_INCLUDED
#define _INC_CSUMMARYTABLE_38222EB00276_INCLUDED

class ostream;

//the summary statistic information about this simulation is
//stored here.
class CSummaryTable
{
public:
      CSummaryTable();  //construtor

      unsigned long GetReproductionNumber();     //accessor
      unsigned long GetAgentNumber();            //accessor
      unsigned long GetNewBirth();               //accessor
      unsigned long GetDeathNumber();            //accessor
      unsigned long GetCombatNumber();           //accessor
      unsigned long GetTradeNumber();            //accessor
      unsigned long GetWinNumber();              //accessor
      unsigned long GetDrawnNumber();            //accessor

      //increase reproduction number by count
      void IncReproductionNumber(int count = 1);

      //increase agent number by count
      void IncAgentNumber(int count = 1);

      //decrease agent number by count
      void DecreaseAgent(int count = 1);

      //increase number of new birth by count
      void IncNewBirth(int count = 1);

      //increase death number by count
      void IncDeathNumber(int count = 1);

      //increase combat number by count
      void IncCombatNumber(int count = 1);

      //increase trading number by count
      void IncTradeNumber(int count = 1);

      //increase winner number by count
      void IncWinNumber(int count = 1);

      //increase drawn number by count
      void IncDrawnNumber(int count = 1);

      //operator overloading
      friend ostream& operator<<(ostream&, CSummaryTable&);
      CSummaryTable operator+(CSummaryTable& x);
```

```
private:
	//total number of reproduction
	unsigned long m_nReproduction;

	//Total number of agents
	unsigned long m_nAgent;

	//total number of child agents being created
	unsigned long m_nNewBirth;

	//total number of agents dead
	unsigned long m_nDeath;

	//total number of combats so far
	unsigned long m_nCombat;

	//total number of tradings
	unsigned long m_nTrade;

	//total number of battle win/lose so far
	unsigned long m_nBattleWin;

	//total number of battle escaped so far
	unsigned long m_nBattleDrawn;

};

#endif /* _INC_CSUMMARYTABLE_38222EB00276_INCLUDED */
```

Utility.h

```
/* this file include global function's prototype */

#ifndef _INC_DIRECTION_380E774401F4_INCLUDED
#define _INC_DIRECTION_380E774401F4_INCLUDED

class CGenome;

enum DIRECTION {UP = 0, DOWN, LEFT, RIGHT, TOTALDIRECTIONS};

int randint(int);
DIRECTION randomDirection();
CGenome* CreateRandGenome();
void DistoryGenome(CGenome *);
void InitGenome(CGenome*);
void AddGenome(CGenome*);
CGenome* FindSameGenome(CGenome*);
void GenomeListReport(ostream &);

#endif /* _INC_DIRECTION_380E774401F4_INCLUDED */
```