Software Design Model for the SoLongSucker Simulation System

Submitted by

Steven Braeger
Sarah Loewy
Andrew Marsh
COP 4331&EEL4884

Spring 2009

Table of Contents

List of Figures

1.0 Introduction

1.1 Document Purpose

The following document provides the details for the purpose and functionality of a program designed to simulate an interactive game known as So Long Sucker. This document is intended for anyone who wishes to understand the design and implementation of the So Long Sucker Simulation system. This document will cover how the system is designed, how it is implemented, and how it is useful..

1.2 Document Scope

This document outlines the production of the So Long Sucker simulation. This software will allow any user to simulate a real world game of So Long Sucker in a virtual environment using a computer. This simulation allows for interaction between simulated players using the computer as a virtual environment. The idea of the simulation is that computerized intelligences will be able to perform the same actions they would in a real game as in the simulation. The benefits of this system are that it allows clients to create and simulate artificial games of So Long Sucker, with different combinations of bargaining strategies. By examining the result of many games with many combinations of strategy, the user may determine the optimum bargaining strategy.

Despite all the benefits, there are limitations to a simulation. Not everything that can be performed in the real world can be performed in the virtual world. For example, the simulation created here does not allow for complex bargaining, such as multiple-player bargains. In the real world game, the possibilities for bargaining are infinite. Unfortunately, this will not be modeled in the virtual world.

Another limitation of playing the game in a virtual environment is the ability to read other players' facial expressions and body language. This can be important when making a bargain with another player since any player has the option to go back on any bargain without penalty. So, the practice of bluffing for a computer player may not be nearly as effective when dealing with a simulation.

1.3 Definitions, Acronyms, and Abbreviations

Definitions of the game-

Bargaining: The process of making a deal with another player perform a certain move in exchange for the transfer of one or more chips.

Board: A collection of Piles
Capture: Occurs when two consecutive chips of the same color are played on the same pile. After every capture, one chip from the capture must be killed.

Chip: A game piece that is represented solely by its color. Each player is assigned a different color.

Dead Box: All chips that are killed are placed in the dead box. Chips in the dead box cannot be brought back into the game.

Defeat: A player is defeated when it is his/her turn and he/she has no chips left to play.

Hand: The set of chips in a players posession.

Kill: The process of removing a chip from the game. Can only be performed on prisoners or after a capture.

Lay on: Playing a chip on top of a pile.

Lay off: Playing a chip on the table, effectively starting a new pile.

Move: The process of playing one chip on the board (either by laying on or laying off).

Prisoner: A chip of a certain color currently in the hand of a player of a different color.

Pile: A stack of chips on the playing board. Chips can be added onto a pile, never taken off.

Transfer: A method by which players exchange chips.

1.4 References

[1] Rapport, Anatol. (2001). *N-person Game Theory.* Mineola, NY: Dover Publications.

[2] *So Long Sucker.* (2009, February 24). In Wikipedia, The Free Encyclopedia. Retrieved 03:19, April 9, 2009, from http://en.wikipedia.org/w/index.php?title=So_Long_Sucker&oldid=272943666

[3] Workman, David. *Discrete Event Simulation,* Lec14-DESimulation-Spr09.ppt

[4] Workman, David. *Software Design Specification Template,* SDS-Template.doc

1.5 Document Overview

The rest of the documentation will be provided as follows:

System Overview

This section details the basics of the simulation. It goes into detail about some of the background involved and the objectives for implementing the simulator. It summarizes the rules of the game. Then, a comprehensive overview of the entire virtual world model will be provided. By way of a use case diagram and activity diagram, it will explain the interactions between actors and possible scenarios. A detailed model of the simulator will be provided in this section, along with the assumptions and dependencies relating to objects outside of the virtual world.

External Interface

This section gives a detailed description of the input and output file specifications. These descriptions will provide enough guidelines so that one will be able to construct a proper input file and interpret the data and messages in the output file. An interactive user interface specification that provides details and a screen-shot for all the prompts in the interface will also be included in this section.

Internal Software

This section provides a breakdown of all the software specifications. It will deal with an overview of the architecture including static and dynamic models. Two system class diagrams will be included, representing the static models, as well as a system activity diagram and an agent communication diagram which explains each message type, representing the dynamic models. Finally, a detailed description of each class, including its package, type, data members, methods, protocols, and its subclasses will be provided.

Implementation

This section highlights a summary of the progress of the overall simulation so far. It will provide a list of things learned through the practice of implementation. It will also detail the restrictions for executing the simulation as well as providing a sample input and output file that could be utilized in the simulation.

2.0 Overview and Summary of the Simulation System

2.1 Project Motivation and Objectives

Nash Simulations, Inc. , (hereafter referred to as “Nash Sims”), has observed a number of its most valuable employees become fed up with company practices regarding software development, and many of them have left for greener pastures at many of their competitors. Additionally, many of those whom Human Resources (HR) has convinced to stay, (even those not as valuable to the team) enjoy massively overweighted salaries and bonuses. In the face of decreased product quality and increasing labor costs, upper management tasked us to investigate this problem. After performing a detailed internal investigation into the details of HR, we discovered that 45%-65% of the employee defection cases could have been resolved with more effective and more competitive employment offers. In addition, 70% of the employees whom Upper Management identified as likely overcompensated were, in fact overcompensated. Of these, 10% were overcompensated due to payroll errors (with a fix to be suggested in document A4352), and 90% were overcompensated due to middle management fear of social rejection and lack of confidence to reject employee demands.

As a means of solving this problem, we decided to come up with a way to better train HR staff to bargain with potential and current employees. By training an artificial intelligence system to effectively bargain for resources, we aim to discover an optimum bargaining strategy that will be an effective algorithm to teach to HR executives. Furthermore, this system must be extensible to bargain with potential applicants automatically. Therefore, we decided that a simulation of a simple game based on economics professor John Nash's thesis work with collective bargaining would be the best choice as a general framework to teach the tools required. The specific game chosen, called So Long Sucker, was chosen, because it is well known to us, was created by John Nash himself, and is well-known for the bargaining behavior it teaches, as well as the ability to break and create bargains.

By implementing several simple strategies for bargaining in a market of players, such as “offer kind deals” or “offer cutthroat deals” or other strategies, and collecting relevant statistical data on which strategies functioned the best in particular games, we aim to discover which strategies are best in a dynamic labour market.

The remainder of this section and the next present a functional description of the virtual world model and architectural design of the So Long Sucker Simulation, the proposed discrete event simulator designed to achieve the business goals we outlined here.

2.2 The Virtual World Model

Rules of the game:

Flow of the game-

1. Assign each player a designated color.
2. Give each player 7 chips of their given color.

3. Randomly decide which player will make the first move. Play begins.
4. Play continues until only one player remains.

Winning/Losing-

1. If it becomes a player's turn, and that player has no chips to play, that player is removed from the game.

2. The last player who has not been removed from the game is declared the winner.

Order of play(within a move)-

1. A move is performed by placing a chip onto the table. This involves either placing a chip on a pile (laying on) or creating a new pile (laying off).

2. The move is then given to another player based on a number of options.

    1. If a player is defeated, the move goes to the player who just gave the defeated player the move and play continues.
    2. If a capture has just occurred, the player whose color made the capture must make the next move and play continues.
    3. If all players are represented in the stack, the move goes to the player whose most recently played chip is farthest down in that pile and play continues.

    4. Otherwise, the player with the current move can select any player whose chip is not in that pile. If the current player lays off, any other player can be selected for the next move and play continues.

Captures-

1. A capture occurs when two consecutive chips of the same color are played on the same pile.

2. The player making the capture takes all the chips in the pile.

3. One of the chips in the pile must be killed.
4. Play continues with the player whose color made the capture as listed in the rules for Order of Play.

Bargaining-

1. A bargain can occur at any time during the game.
2. A bargain must be discussed openly at the playing table.
3. A bargain can be accepted or denied at any time during the game.
4. Any player can decide to not uphold the bargain without a penalty.

Use Case Diagram:

The following diagram illustrates the virtual world for this game. The use cases are all inside the virtual world, and the actors exist outside of the virtual world.

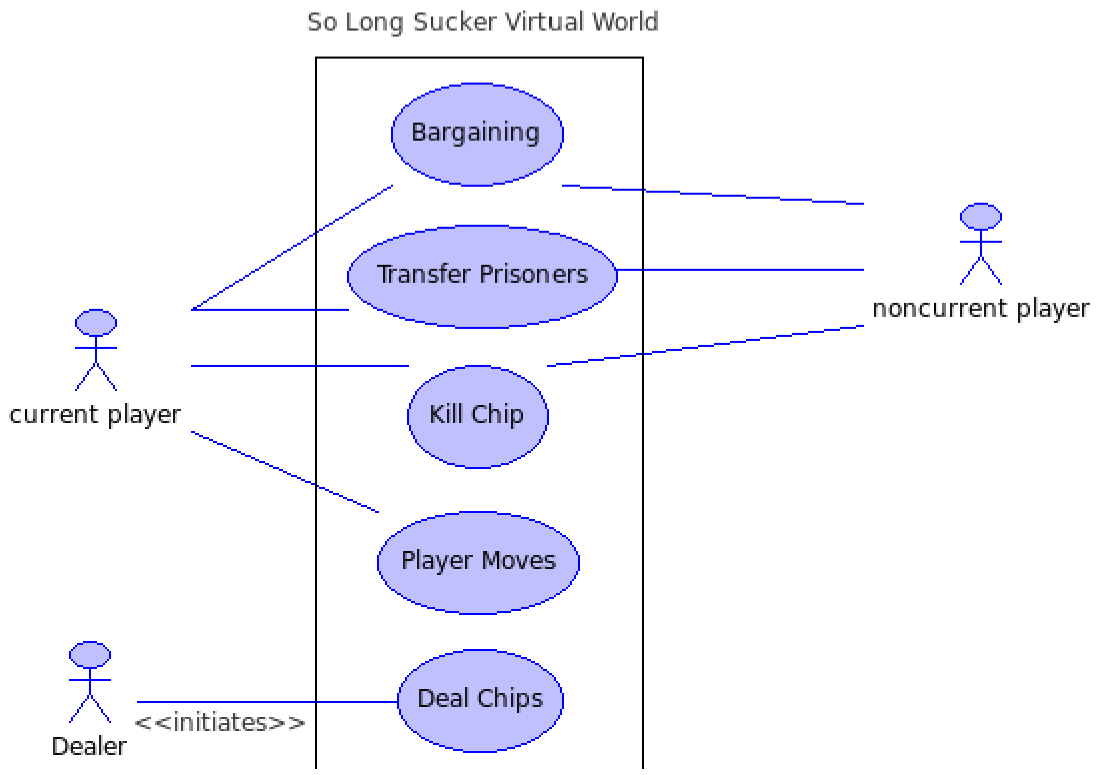


Figure 1: Use case diagram illustrating the virtual world

Use Cases:

Deal Chips

The process of dealing chips is done first before the game even begins. The dealer performs the action of dealing 7 chips to each player. After this, the player who is given the first move is selected at random. This process of dealing chips to enable gameplay can be seen as initializing the game. After initializing, no more chips are dealt and the dealer is no longer needed.

Bargaining

The most important thing to remember in bargaining is that it can happen at any time during the game, and it can be between any two players. The bargaining process leads to a player accepting or rejecting the offer made. Then, if accepted, often leads to a transfer of chips between the two players.

Transfer

A transfer of chips can occur at any time during the game between any two players, and it usually occurs after a bargain is made.

Kill Chip

Either the current player and any other player can kill a chip at any time. However, a player can only kill one of its own chips after a capture occurs. In all other cases, each player is only allowed to kill a prisoner.

After the game begins, there are four different cases that can occur: make a bargain, transfer a prisoner, kill a chip, or make a move on the table. Each player at the table can perform any of the first three cases at any time during the game. This is illustrated by the association between the non-current player and bargaining, transfer prisoners, and kill chip. Only one player at a time has the ability to move. That is shown with the current player and its association with the case player moves, meaning that that player places a chip on the board.

Activity Diagram

The activity diagram below provides an illustration of the general flow of the simulation from start to finish. It provides more detail and breaks down each situation case by case.

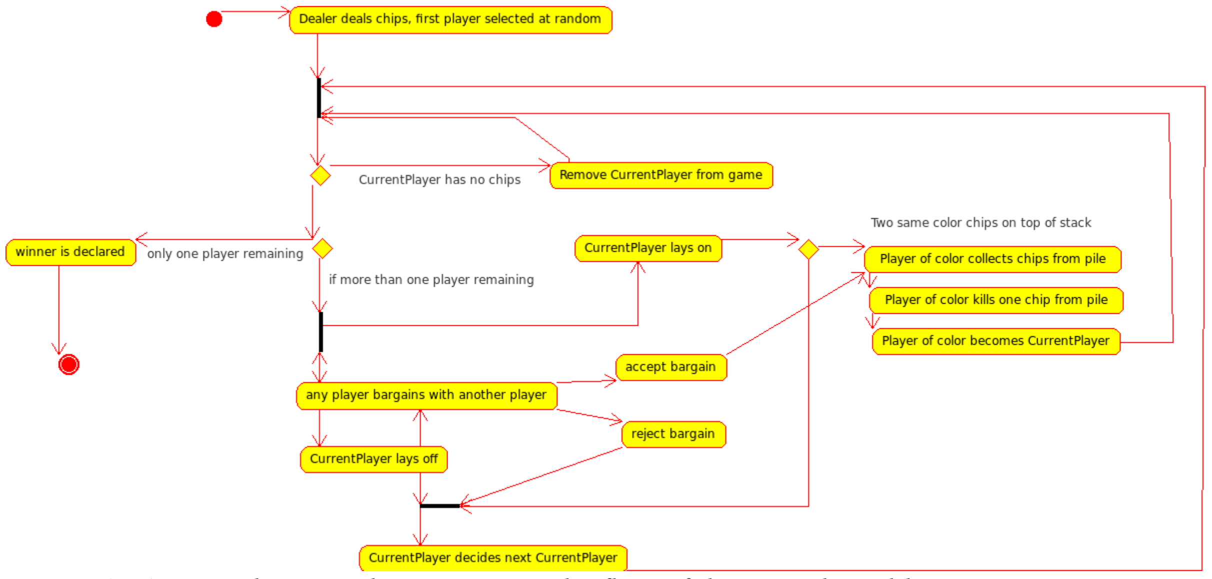

Dealer deals chips, first player selected at random
CurrentPlayer has no chips
Remove CurrentPlayer from game
winner is declared
only one player remaining
if more than one player remaining
CurrentPlayer lays on
Two same color chips on top of stack
Player of color collects chips from pile
Player of color kills one chip from pile
Player of color becomes CurrentPlayer
accept bargain
any player bargains with another player
reject bargain
CurrentPlayer lays off
CurrentPlayer decides next CurrentPlayer

Figure 2: Activity diagram demonstrating the flow of the virtual world

To show how the activity diagram illustrates the flow of the game, let's walk through a step-by-step scenario of how the game would be played. First, the dealer deals the chips, then play begins. Every time a player is selected, the first thing to check for is if that player has any chips remaining. If not, that player is removed from the game. Otherwise, play continues as normal.

2.3 Model of the Simulator

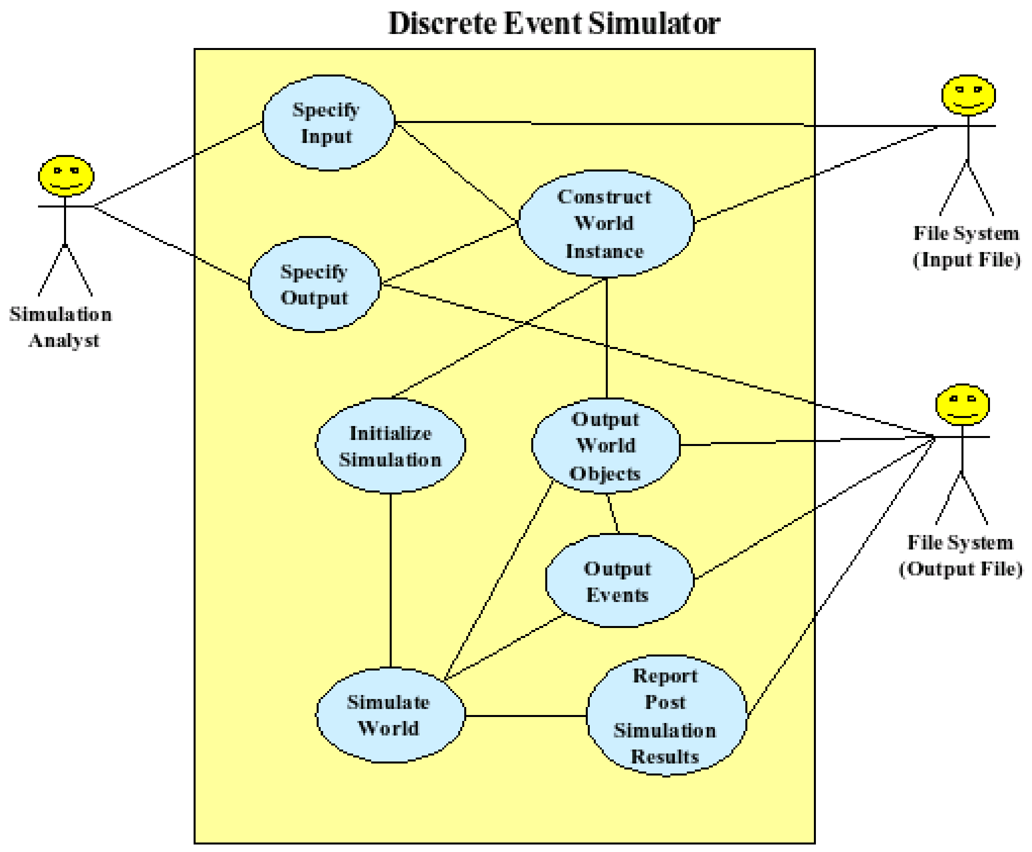


Figure 3: Model of the simulator

2.4 Assumptions and Dependencies

The simulator has been thoroughly tested on several different platforms. It was written specifically to run on the Olympus UCF server. This server runs the Ubuntu Linux operating system on an x86 processor. The software was written using ISO Standard C++ coding conventions. It was compiled using the GNU C++ Compiler, version 4.2.4, and requires at least that version to run. It has no other dependencies. To our knowledge, there are no relevant government regulations regarding the sale,distribution or use of this product. Furthermore, we deny any warranty, written or implied, including fitness for a particular purpose.

3.0 External Interface Specifications

3.1 Input (File) Specifications

This section describes the input file specifications relevant to successfully create a test run of the software. This software follows the Object Oriented Design Data specification. Specifically, the input file format serializes each set that will be needed to create and simulate any number of games.

Relevant tags:

SoLongSucker{ }SoLongSucker
: This tag initializes and runs exactly one execution of the simulated environment. It has only one property: numChips, which specifies the number of chips each player in the game has.

NancyNice{ }NancyNice
: This tag initializes and runs exactly one Player with the AI strategy of NancyNice.

JacobJerk{ }JacobJerk
: This tag initializes and runs exactly one Player with the AI strategy of JacobJerk.

RandyRandom{ }RandyRandom
: This tag initializes and runs exactly one Player with the AI strategy of RandyRandom.

The following is an example of two valid input files, representing two games, one with 50% NancyNice and JacobJerk, to simulate a half and half game, and the second, with 3 RandyRandom players.

```
SoLongSucker{
        numChips: 5
        NancyNice{}NancyNice
        JacobJerk{}JacobJerk
        JacobJerk{}JacobJerk
        NancyNice{}NancyNice
}SoLongSucker


SoLongSucker{
        numChips: 3
        RandyRandom{}RandyRandom
        RandyRandom{}RandyRandom
        RandyRandom{}RandyRandom
}SoLongSucker
```

Figure 4: Example Input File

3.2 Output (File) Specifications

The output for this program consists of two parts: 1) A message log detailing the messages the system sent and the time the system sent them: 2) the statistical data collected with regard to each game.

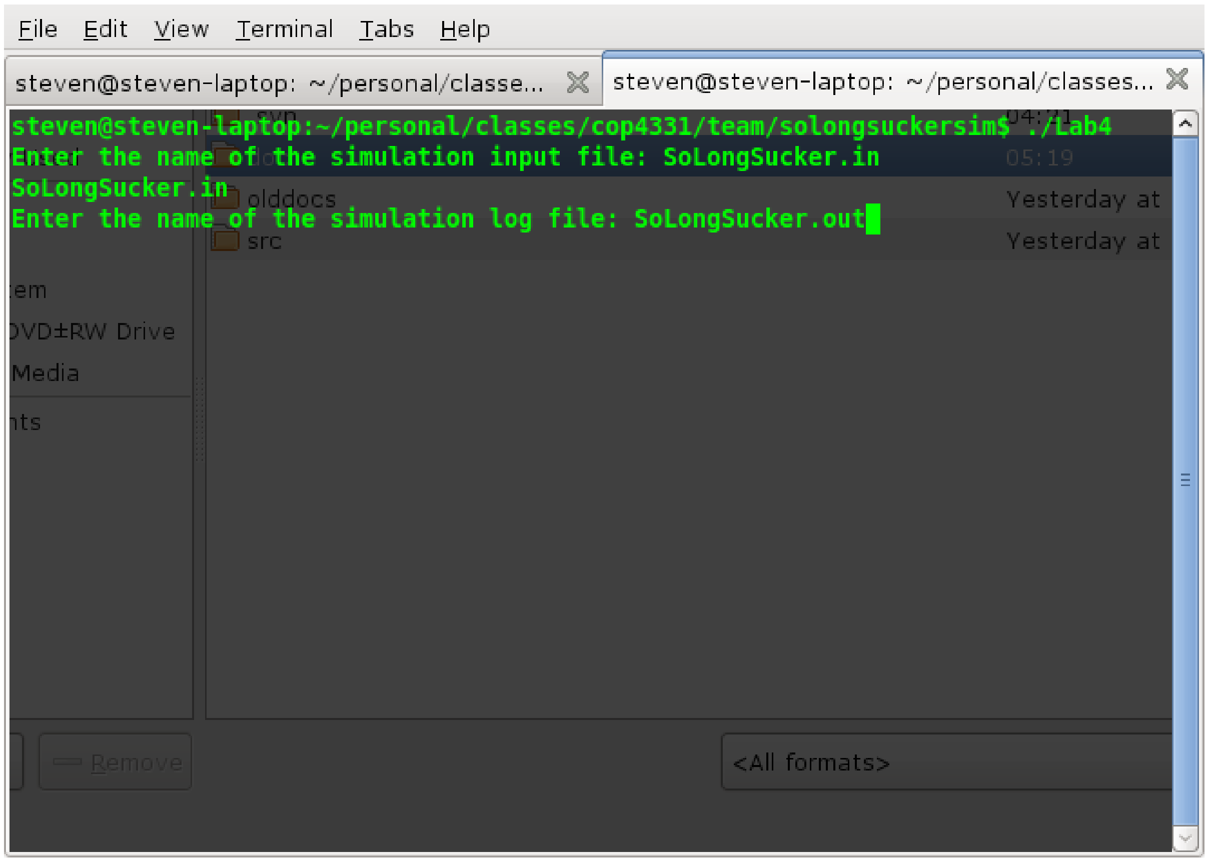
The total output will be sent to a file as specified by the user. An example of this file is given below.

```
PlayersMessage{  agent[ 0 ] = JacobJerk
        agent[ 1 ] = NancyNice
        agent[ 2 ] = NancyNice

}PlayersMessage

TIME: 1 SENDER: NancyNice RECVR: NancyNice  ChipMessage{  Handler: 3

                                    Description: Give the turn to NancyNice
                             }Message



TIME: 2 SENDER: NancyNice RECVR: NancyNice  BargainMessage{  Handler: 4

                                  Description: Nancy Nice offers a bargain to NancyNice
                                  Bargain{
                                             move: MoveProposal{
                                                      chip: 1
                                                      pile: 2
                                             }MoveProposal
                                             chips: 1 2 3 2
                                             nextTurn: 1
                                  }Bargain
                            }Message
```

Figure 5: Example Output file.

3.3 Interactive User Interface Specification

The interactive user interface, considering that this is a discrete event simulator, is very little. To begin, the user will be prompted to enter the input file controlling the simulation. It will then be prompted to enter the desired file containing the output from the simulation. A screen-shot of this process is here:


File Edit View Terminal Tabs Help
steven@steven-laptop: ~/personal/classe...
steven@steven-laptop: ~/personal/classes...

```
steven@steven-laptop:~/personal/classes/cop4331/team/solongsuckersim$ ./Lab4
Enter the name of the simulation input file: SoLongSucker.in
SoLongSucker.in
Enter the name of the simulation log file: SoLongSucker.out
```

Figure 6: Screenshot Demonstrating Input Specification

4.0 Internal Software Specification

4.1 Architectural Overview

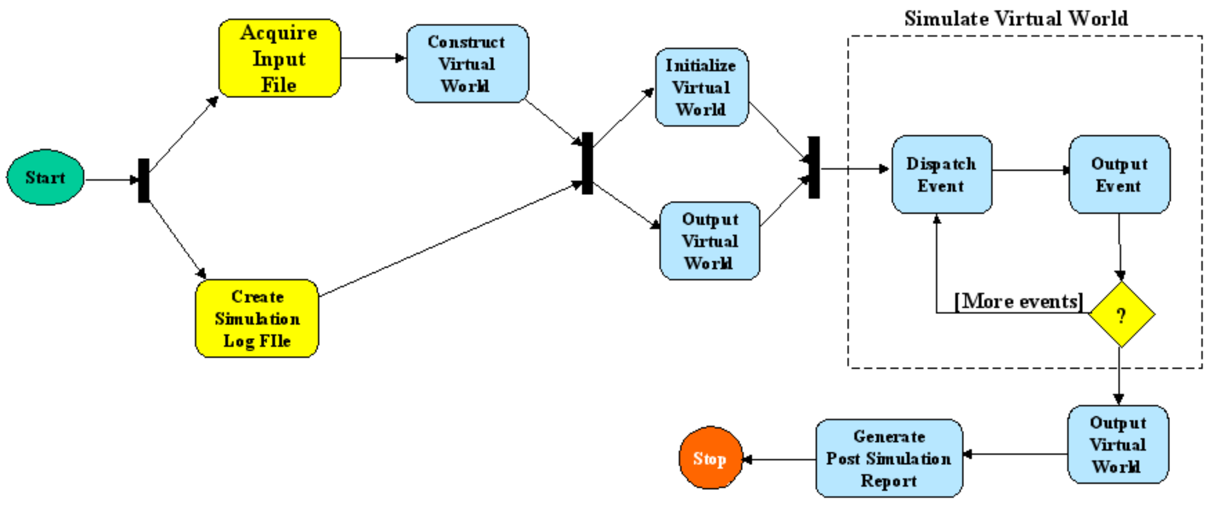


Figure 7: The dynamic model of the So Long Sucker System.

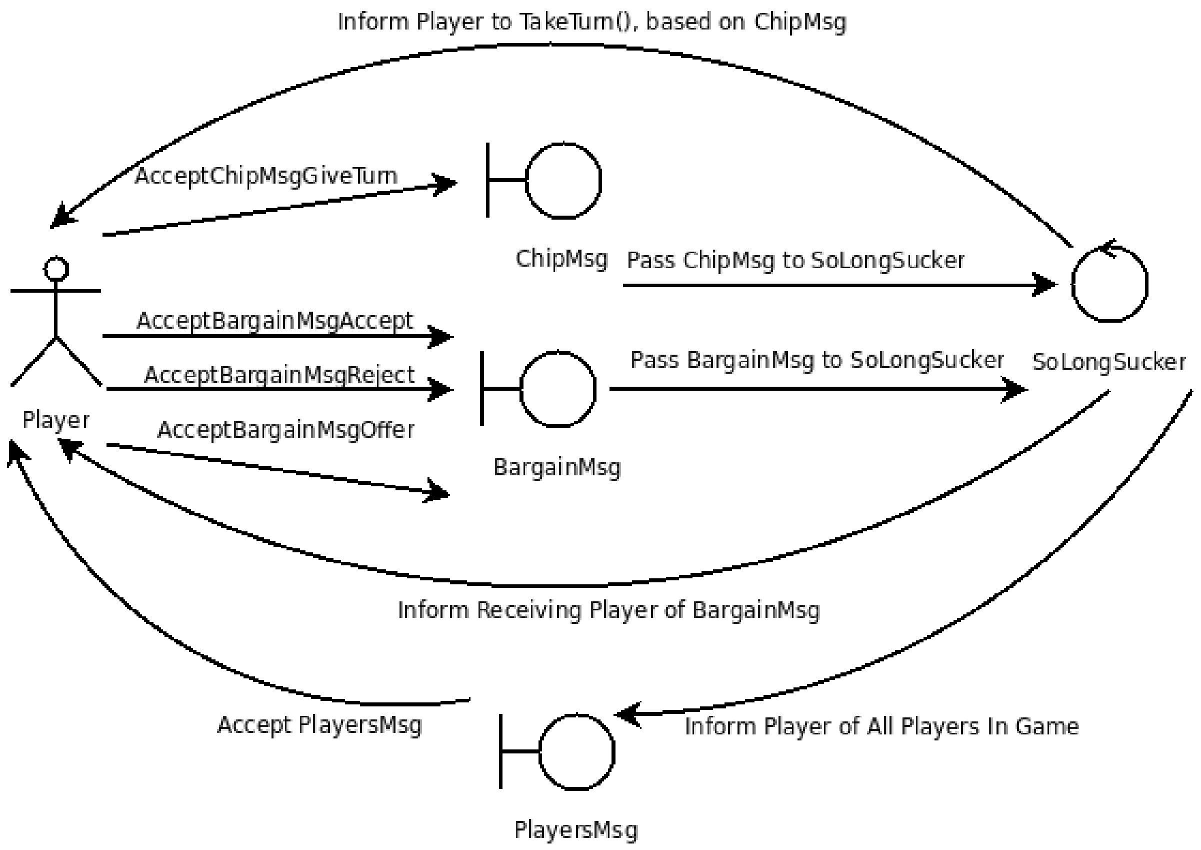

Inform Player to TakeTurn(), based on ChipMsg
AcceptChipMsgGiveTurn
ChipMsg
Pass ChipMsg to SoLongSucker
AcceptBargainMsgAccept
AcceptBargainMsgReject
Player
AcceptBargainMsgOffer
Pass BargainMsg to SoLongSucker
SoLongSucker
BargainMsg
Inform Receiving Player of BargainMsg
Accept PlayersMsg
Inform Player of All Players In Game
PlayersMsg

Figure 8: shows the interactions between Player and other Players via SoLongSucker transmitting the three core Message types:

Message Specifications:
BargainMsg exists to provide a method of the discrete simulator of So Long Sucker to inform players of Bargains to be passed between them.

Data Members: int handler, string description, Bargain bargain

ChipMsg exists to provide a method of the discrete simulator of So Long Sucker to inform Players when it is their next turn or to inform the Player in turn of the available Players to select for the next turn, according to rules.

Data Members: int handler, string description, Chip chip

PlayersMsg exists to inform a given Player of all other Players in the simulation.

Data Members: int handler, string description, Player* players

4.2 Package Specifications

In this section we describe the purpose of each package in the architecture and identify the classes that participate in the design. Figure 9 presents a view inside each package exposing the classes contained within and some of the key associations and relationships among classes belonging to different packages.

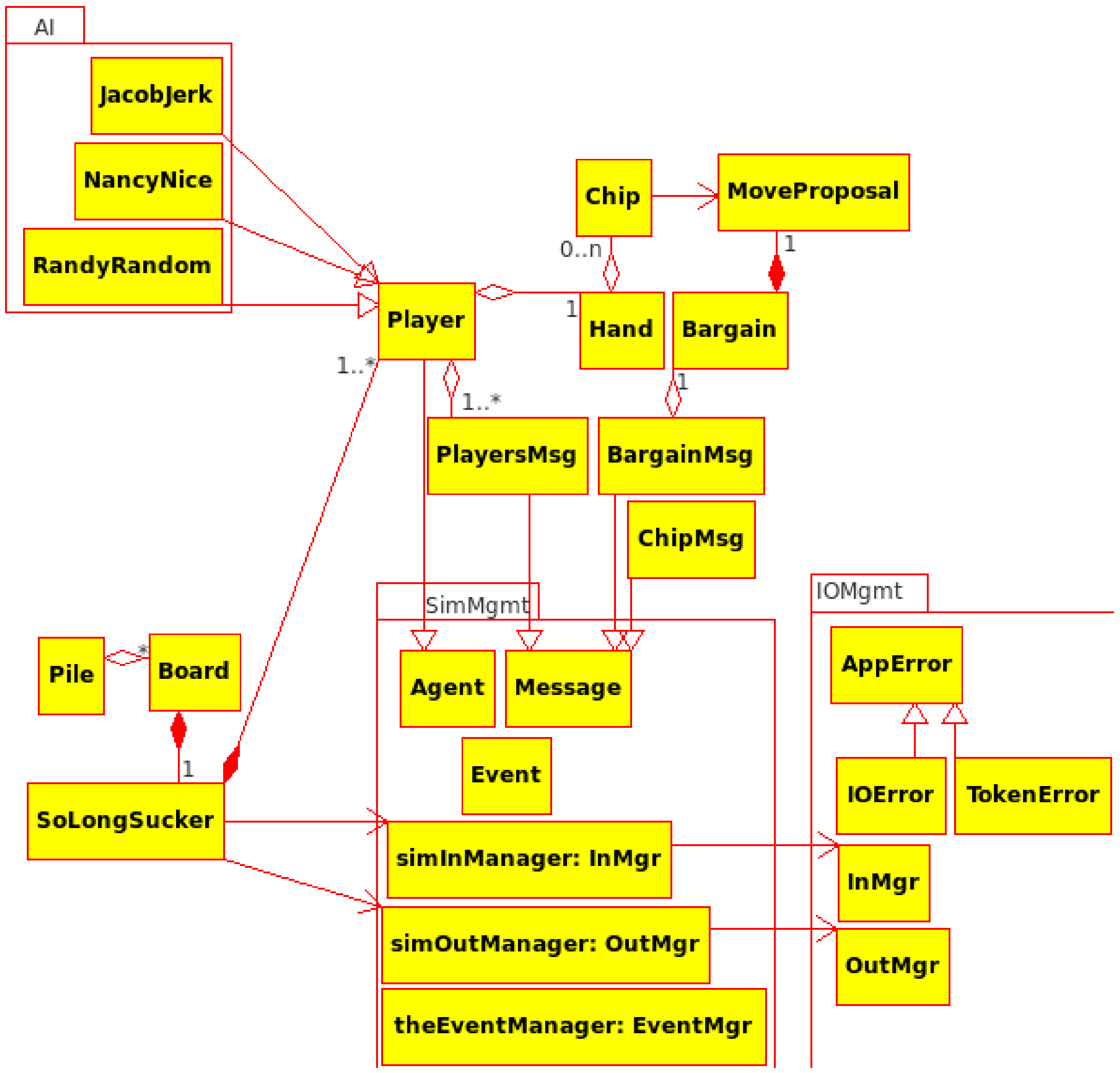


Figure 9: Class Diagram

Package IOMgmt.

This package provides several classes for managing file streams. The two primary classes used by the simulator are InMgr and OutMgr. In addition, two exception classes are provided. Class IOError is used exclusively by classes in IOMgmt for throwing exceptions when errors occur while attempting to perform stream operations (e.g. open(), append(), etc.). Class TokenError is provided to application clients of InMgr when trying to parse data from some input stream. TokenError is thrown by all Extract() and Get() methods of classes in SimModels that must read instances from the input file stream when the virtual world is being constructed.

Package SimMgmt.

This package encapsulates simulation infrastructure classes that do not depend on the particular virtual world being modeled. It contains four reusable classes and three reusable simulation objects. Class Agent is an abstract class that defines a set of methods that must be redefined in all active simulation objects. As shown in Figure 9, all active simulation objects (e.g., Player) must be derived subclasses of Agent. Each of these classes must redefine methods defined in class Agent.

Class EventMgr encapsulates and manages the event queue. A pre-defined instance of this class is called theEventMgr and is a member of this package. EventMgr provides services such as PostEvent() and getNextEvent() for adding and removing instances, respectively, of class Event.

Other services include MoreEvents() that determines whether or not the event queue is empty. Finally, EventMgr provides services GetRecvr(), GetSendr(), GetMsg() and Clock() which provide the identity of the receiver agent, the sender agent, the message sent, and the simulation time of the last event removed from the queue.

SimMgmt also holds two pre-defined objects for managing the simulation input and output streams. SimInMgr manages the input file stream and simOutMgr manages the simulation log. These objects are instance of classes IOMgmt::InMgr and IOMgmt::OutMgr, respectively.

Class Layer Descriptions

Figure 10 gives a detailed view of the custom members of the system. These classes would have to be changed or replaced to simulate a new virtual world or to enhance an existing simulator.

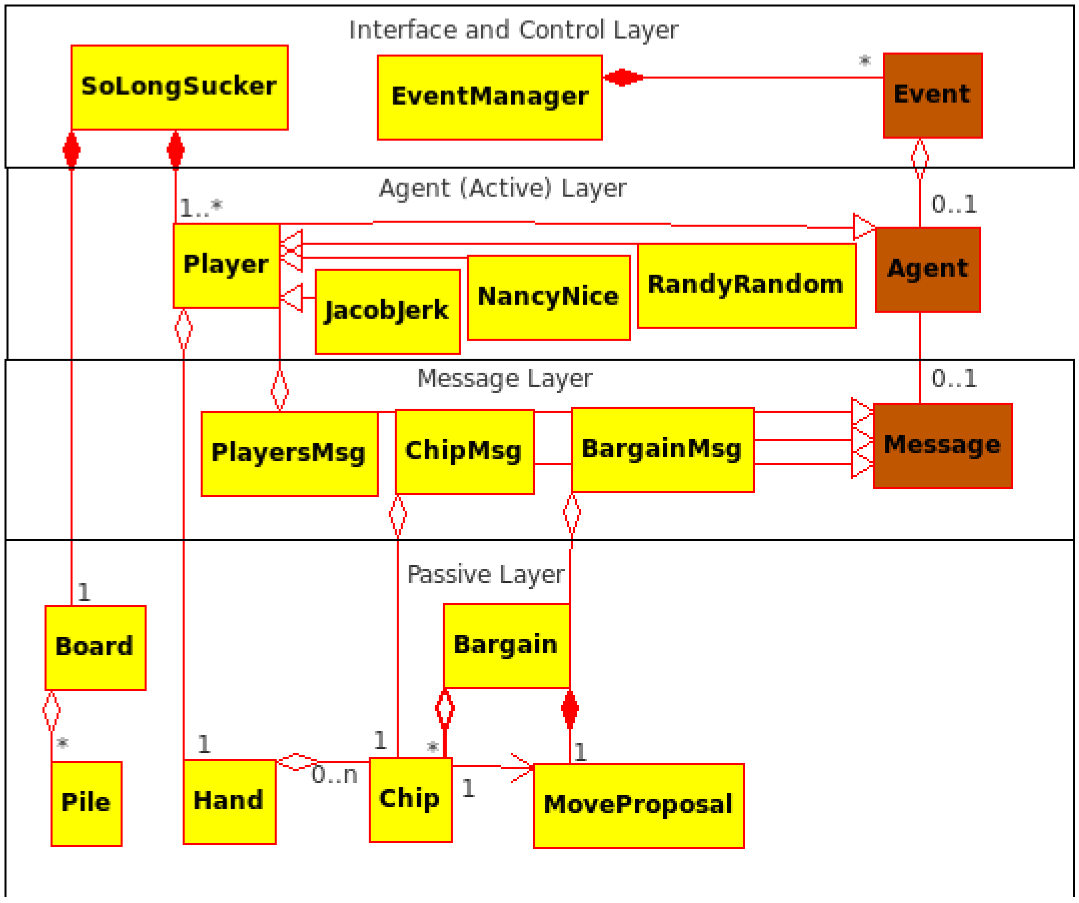


Figure 10: Class Layer Diagram

Interface and Control Layer

Class SoLongSucker defines and represents the "Simulated", or "Virtual World". An instance of this class is the object to be simulated. Note the composition relationship between World and the agent classes. These are objects that SoLongSucker constructs from the input file stream. Instances of other classes are created dynamically when Agents send messages to one another.

One of its methods of SoLongSucker, Simulate(), controls the simulation by removing Events from theEventMgr and dispatching the contained Message instance to the designated receiver Agent. The receiving Agent decodes and acts on the message it receives – this may spawn the creation of new Events that are posted to theEventMgr. This scenario repeats until all Events have been dispatched and the Event queue becomes empty.

Agent Layer

This layer consists of all active object classes or agents. Active objects must be instances of some subclass of abstract class Agent. The simulation progresses as a result of events created and serviced by agents. All simulation time must be accounted for in the time delays associated with sending messages from one agent to another. In other words, agents control the progress of the simulation and can only communicate with each other by sending messages through events. There is one exception to this statement – a direct call must be made from a sending agent to a public message constructor provided by the receiving agent. These calls are necessary as part of message passing mechanism and have no meaning in the virtual world.

An agent must call methods of passive classes directly but if such calls have meaning in the virtual world, then the associated simulation time must be accounted for in the time delay defined for the next message sent to another agent.

An Event has four components: a sender agent, a receiver agent, an instance of some message class, and an event time. When one Agent wishes to interact with another, it must do so by creating an instance of Event that defines a "future" time at which the interaction will occur. The message component defines an action to the receiver agent and possibly carries data necessary to complete the action. The simulation time associated with the new event is computed by the sender and must include virtual world delays associated with any actions the sender may perform to cause the interaction with the receiver – this includes interactions between the sender and any passive objects.

Message Classes

Messages contain two important data members, a message handler id or code, and, optionally, a passive data object that is needed by the message handler to execute the requested action. Message instances are created by the receiver agent. For each potential incoming message, the receiver agent class assigns an integer code that identifies the private message handler that will be called to service or act on the message when it is received during simulation. Message handlers are called by the receiver's Dispatch() method – the Dispatch() method uses the handler code to identify an internal private method that will act on the message and its data (if any). Furthermore, as mentioned above, the receiver agent must provide a public message constructor method, for each message it can expect to receive from some other agent. These message constructors may take passive data as parameters and create an appropriate instance of some message class. The purpose of these message constructors is to allow the receiver to encode the handler id as part of the message so that it will know how to dispatch it when the corresponding interaction event occurs (in the future).

Passive Layer

The passive layer contains all classes that model problem data and inanimate objects of the simulated world. Agents make direct calls on passive objects, but must account for the simulation time consumed when doing so. Passive objects may make direct calls to each other, if necessary.   Passive objects may be passed from one agent to another as part of an instance of some message.

4.3 Class Specifications (SimModels only)

4.3.1 Agent Subclasses

Player
RandyRandom
JacobJerk
NancyNice

4.3.2 Message Classes

Message
PlayersMsg
ChipMsg
BargainMsg

4.3.3 Passive Classes

Bargain
Chip
Board
Hand
Pile
MoveProposal

4.3.4 Table of Contents

4.3.5 Detailed Class specifications.

Name: Bargain
Type: Entity
Purpose: Bargain exists to provide a basis of communication between existing Players in the simulation. A Player can offer a set of Chips in hopes of getting the next turn and requesting a move or the Player in turn could offer to perform a move and give the turn to a player, provided that they give them a certain number of Chips.
Data Members: MoveProposal move, multiset<Chip> chips, Chip nextTurn
Methods: public Bargain()
public Bargain(MoveProposal move, multiset<Chip> chips, Chip nextTurn)
public ~Bargain()

Name: BargainMsg
Type: Boundary
Purpose: BargainMsg exists to provide a method of the discrete simulator of So Long Sucker to inform players of Bargains to be passed between them.
Data Members: int handler, string description, Bargain bargain
Methods: public BargainMsg(int handler, string description)
public BargainMsg(int handler, string description, Bargain& bargain)
public ~BargainMsg()

Name: Board
Type: Entity
Purpose: Board exists as a container for Piles.
Data Members: list<Pile> piles
Methods: public Board()
public ~Board()

Name: Chip
Type: Entity
Purpose: Chip exists as both a Player identification color and as tokens for play in Piles.

Name: ChipMsg
Type: Boundary
Purpose: ChipMsg exists to provide a method of the discrete simulator of So Long Sucker to inform Players when it is their next turn or to inform the Player in turn of the available Players to select for the next turn, according to rules.
Data Members: int handler, string description, Chip chip
Methods: public ChipMsg(int handler, string description)
public ChipMsg(int handler, string description, Chip& chip)
public ~ChipMsg()

Name: Hand
Type: Entity
Purpose: Hand is a container for Chips held by a Player.
Methods: public Hand()
public Hand(unsigned size, Chip chipId)
public ~Hand()

Name: JacobJerk
Type: Control
Purpose: JacobJerk is a subclass of Player and provides a strategy by which to bargain during the simulation.
Data Members: Hand hand, Chip chipId
Methods: public JacobJerk()
public JacobJerk(unsigned sizeOfHand, Chip chipId);
public ~JacobJerk()

public Initialize(Message* players)
public Dispatch(Message* msg)

public Bargain* AcceptBargainOffer();
public Bargain* AcceptBargainAccept();
public Bargain* AcceptBargainReject();

public void TakeTurn();

public ChipMsg* AcceptChipMsgGiveTurn(list<Player> validPlayers);

Name: MoveProposal
Type: Entity
Purpose: MoveProposal acts a way to indicate what move the Player intends to take.
Data Members: Chip chip, list<Pile>::iterator pile
Methods: public MoveProposal()
public MoveProposal(Chip chip, list<Pile>::iterator pile)
public ~MoveProposal()

Name: NancyNice
Type: Control
Purpose: NancyNice is a subclass of Player and provides a strategy by which to bargain during the simulation.
Data Members: Hand hand, Chip chipId
Methods: public NancyNice()
public NancyNice(unsigned sizeOfHand, Chip chipId);
public ~NancyNice()

public Initialize(Message* players)
public Dispatch(Message* msg)

public Bargain* AcceptBargainOffer();
public Bargain* AcceptBargainAccept();
public Bargain* AcceptBargainReject();

public void TakeTurn();
public ChipMsg* AcceptChipMsgGiveTurn(list<Player> validPlayers);

Name: Pile
Type: Entity
Purpose: Acts as an ordered container for Chips.
Methods: public Pile()
public ~Pile()

Name: Player
Type: Control
Purpose: Player is a subclass of Agent and provides a main actor for play during the simulation.
Data Members: Hand hand, Chip chipId
Methods: public Player()
public Player(unsigned sizeOfHand, Chip chipId);
public ~Player()

public Initialize(Message* players)
public Dispatch(Message* msg)

public Bargain* AcceptBargainOffer();
public Bargain* AcceptBargainAccept();
public Bargain* AcceptBargainReject();

public void TakeTurn();
public ChipMsg* AcceptChipMsgGiveTurn(list<Player> validPlayers);

Name: PlayersMsg
Type: Boundary
Purpose: To inform a given Player of all other Players in the simulation.
Data Members: int handler, string description, Player* players
Methods: public PlayersMsg(int handler, string description)
public PlayersMsg(int handler, string description, Player* players)
public ~PlayersMsg()

Name: RandyRandom
Type: Control
Purpose: RandyRandom is a subclass of Player and provides a strategy by which to bargain during the simulation.
Data Members: Hand hand, Chip chipId
Methods: public RandyRandom()
public RandyRandom(unsigned sizeOfHand, Chip chipId);
public ~RandyRandom()

public Initialize(Message* players)
public Dispatch(Message* msg)

public Bargain* AcceptBargainOffer();
public Bargain* AcceptBargainAccept();
public Bargain* AcceptBargainReject();

public void TakeTurn();
public ChipMsg* AcceptChipMsgGiveTurn(list<Player> validPlayers);

Name: SoLongSucker
Type: Control
Purpose: SoLongSucker serves as the ground truth object for the simulation, starting and maintaining system and system log data.
Data Members: Player* players, Board board
Methods: public SoLongSucker()
public ~SoLongSucker()

public void Initialize();
public void Simulate();
public void WrapUp();

public Message* AcceptChipsDiscard(multiset<Chip> chips);

Client Table

Class::Client Method	**Methods Called in This Class**

Server Table

Method of This Class	**Class::Method(s) called from This Method (exclude library methods)**

5.0 Implementation Summary