CS4500 Fall 2009 Sub Project 1
Karl Lieberherr and Ahmed Abdelmeged and Bryan Chadwick
September 6, 2009
Posted at:
http://www.ccs.neu.edu/home/lieber/courses/cs4500/f09/project/assignment1/Project1_CS4500_f09.pdf

Notes:
======
NOTE: This description is intentionally a bit underspecified.
Ask questions in class to clarify the requirements so that
you have a more complete picture.
In any software development project you have to dig for the requirements.
There is usually room for interpreting the requirements
in different ways.

Part 1
==========

We are going to create an artificial world that we populate with little creatures that buy and sell computational challenges. Each creature gets the same initial life energy and they are responsible for the buy and sell decisions they make. The computational challenges are opportunities to win or lose energy. Each computational challenge that is bought consumes energy but there is the potential to get that energy back through clever use of a second resource: computational energy.

You have to take good care of those little creatures because you will be graded by what you "teach" them. A good dose of your computer science and mathematics knowledge will flow into those little creatures and if it does not, your grade will suffer. The little creatures will trade with each other using the knowledge you have given them. The life energy your creature will have left at the end of the game will influence your grade in a significant way. The life energy we also call money.

Here are some trading examples: A summary of robot competitions: Spring 2009 Competition 11 . The agents Celtics and Bender were the winners. Remarkable is also the bottom agent, called GenericPlayer,
which is a very simple agent basically generated automatically from a game description. It has only minimal intelligence and it will be given to you to add intelligence to it.
Detailed History Example shows you how an agent battle might unfold. Many more competitions are described here: Spring 2009 competitions .

The software development project of Fall 2009 is about designing and implementing a new software development process for problem solving software for domains that are computationally hard (like solving NP-hard problems).
A key feature of the software development process is that we want egoistic software that can fight for itself and prove that it is better than other software that solves
the same problems. Every week, the competitions tell us which agent is the best. We call those competitions Specker Challenge Game (SCG) competitions. The SCG competitions take place in an artificial world that we have carefully designed to help us in the real world of software development. What are the main challenges of software development?
First we need to write robust software and second we need to write software that scales.
Therefore in the SCG world, we punish agents that are not robust and that don't scale.

Examples of non-robust agents are: agents that contain bugs and agents that solve the problems sub-optimally. Non-scalable agents are agents that use too many resources such as time and energy.

SCG is a zero-sum game which means that if an agent A is punished, another agent makes a profit for finding a weakness in agent A. Although the agents are egoistic, they help the software developers behind an agent, to find bugs or inefficiencies.
The SCG, which is a completely automatic agent game, is embedded in a human game which runs for a several weeks where the software developers improve their agents based on feedback from earlier competitions. The software developers want there agent to win and therefore they will be strongly encouraged to write robust and scalable software. We assume that the salary or the reputation of the software developers depends directly on the performance of their agent in the competitions.
Indeed, your grade will depend on the performance of your agent.

What is egoistic software? It is software that cannot only solve problems but also pose hard problems
to other agents. It is software that claims beliefs about properties of the problem solving software when applied to a niche (set of sub problems). Those beliefs can be either constructively supported or constructively discounted.
Beliefs are turned into computational challenges by adding a price to them. A computational challenge is a pair: (belief, price). When an agent accepts a challenge,
it thinks the belief behind the challenge is wrong. Upon acceptance the price has to be paid and then the offering agent (called Alice) delivers a problem to the accepting agent (called Bob) for which Alice thinks that her belief cannot be discounted.
If Bob nevertheless succeeds in discounting Alice' belief, Bob wins, otherwise he loses. If Bob loses, he loses the price paid for the challenge. If Bob wins, he makes a handsome profit, like winning 3 * p, where p is the price
for the challenge.

We think of the beliefs being constructed by experimenting with the problem solving software and observing its behavior. If the problem solving software is buggy or slow, the resulting beliefs might be wrong and easily discounted by other agents.

Egoistic software is problem solving software that is self-interested. Egoistic software communicates with other egoistic software to achieve a common good: reliable, efficient problem solving software. The self-interest is carefully designed so that a
community of egoistic software (also called a community of egoistic agents) achieve the desired common good.

What does it mean to say that agents are self-interested? It does not necessarily
mean that they want to cause harm to each other, or even that they care only about themselves. Instead, it means that each agent has his own description of which
states of the world he likes - which can include good things happening to other agents.

What are the BENEFITS of egoistic software for you? Your peers help you test your software and you feel good because you help your peers test their software.
Development of egoistic software is a community experience because your agent will get constructive feedback from the other agents that you are encouraged to
incorporate in the next version of your agent. You must digest the constructive feedback from the class community. This is an effective way to learn from your peers.

The agent who makes the most money wins the game. The game is not only about eliminating bugs and inefficiencies but also about expressing "correct" beliefs about the software and the problem domain to which
it applies. There is a niche language to formulate niches of sub problems. The game will punish posting beliefs that can be constructively discounted and it will punish accepting beliefs that cannot be discounted.
After the game has been played for a while by strong agents, only correct beliefs will survive and the game accumulates a body of knowledge about the behavior of the problem solving software on niches. This knowledge is very useful in applying the software effectively.

A simple instance of SCG:
The objects traded are computational challenges. Each challenge has a creator and can be bought by a player at the price offered by the creator. (We use agent and player as synonyms.) We say that a buyer owns a challenge. When you own a challenge with predicate T (defining a niche), you have the right to request a problem (raw material) satisfying T and you have the right to sell back to the creator the finished product you produce from the raw material at a price that is equal to the quality of your product. The better quality you produce, the more you will be paid. On the other hand, the creator of the challenge
has only the obligation to give you a problem (raw material) satisfying predicate T, but he/she may make it hard for you to produce a high quality product.
An SCG competition (or contest) is controlled by an administrator that controls a list of players. A player consists of a name, a list of offers (challenges offered for sale), a list of sold challenges bought by other players), a list of bought challenges (challenges bought from other players), and a money field for the money owned by the player. Initially, each player gets 5 million dollars.

In this simple instance of SCG, a challenge consists of a predicate T and a price p. The belief expressed by the challenge is that there exists a problem satisfying predicate T so that no solution to the problem is better than p.
If Alice poses challenge (T,p) and Bob accepts it, Alice supports her belief by giving a problem to Bob for which he cannot find a solution of a quality better than p. If Bob succeeds in finding a solution of a quality
higher than p, Alice' belief is discounted and Bob wins and makes a profit.

Alternatively, we say that a challenge consists of a belief b and a price p. A belief b consists of a predicate T and a real number p1 in the closed interval [0,1]; b = (T,p1). We consider challenges = ((T,p),p) in this simple instance of SCG.

A challenge consists of a required name, a required predicate, a required creator (a player), a required price, and a few fields representing the state of the challenge as follows. If the optional boughtBy field is present, the challenge has been bought by another player. If the optional rawMaterial field is present, the buyer has requested raw material (currently unspecified). If the optional finished field is present, the buyer has delivered the finished product which is currently unspecified and it has a quality field which says how much the buyer has to be paid by the creator of the challenge.

Create an object-oriented design for The Specker Challenge Game. Include also the history, the list of moves that happened during a game (the administrator keeps track of this information). The moves that must be recorded include CreateChallenge, BuyChallenge, DeliverRawMaterial, DeliverFinishedProduct. CreateChallenge creates a challenge (unsold). BuyChallenge records who bought the challenge. DeliverRawMaterial records the raw material delivered by the creator of the challenge. DeliverFinishedProduct records the

finished product delivered by the buyer and the quality (a real number in [0,1]) of the product. All prices are real numbers in [0,1] (fraction of a million $).

Here is an example of the kind of information you need to represent. This representation contains redundancy and you should attempt to reduce some or all of the redundancy in your design.

```
(player
    name Peter
    offers
      (challenge
         name d1
         belief unspecified
         creator Peter
         price 0.75
       challenge
         name d2
         belief unspecified
         creator Peter
         price 0.618
       )
    soldOffers
      (challenge
         name d3
         belief unspecified
         creator Peter
         price 0.85
         boughtBy Paul
       challenge
         name d4
         belief unspecified
        creator Peter
        price 0.618
         boughtBy Paul
       )
    boughtOffers
      (challenge
      name d5
      belief unspecified
       creator Paul
         price 0.85
        boughtBy Peter
       challenge
         name d6
         belief unspecified
         creator Paul
         price 0.6
         boughtBy Peter
       )
   money 0
)
history
  (create
     challenge
```

```
      name d1
      belief unspecified
     creator Peter
    price 0.75
  create
    challenge
      name d4
      belief unspecified
       creator Peter
       price 0.618
  buy
    challenge
      name d3
       belief unspecified
      creator Peter
       price 0.85
       boughtBy Paul

  delivered
    challenge
      name d3
       belief unspecified
       creator Peter
      price 0.85
       boughtBy Paul
       rawMaterial
         unspecified raw material
  finished
    challenge
       name d3
       belief unspecified
       creator Peter
       price 0.85
       boughtBy Paul
       rawMaterial
         unspecified raw material
       finished
         unspecified finished product
         quality 0.25
)
```

The above example is intentionally incomplete and not from a real game. The purpose is only to show the kind of information that needs to be represented.

Turn in your object-oriented design, together with the corresponding Java or C# classes. The classes must be capable to parse a game history and print it back out. But you should not have to write any Java or C# code to define the details of the parsing and printing functionality. A tool should generate this information from a schema that you write for your object-oriented design.

I suggest that you use the DemeterF tool, a brand new tool developed in our college by Bryan Chadwick. See file: how-to-use-DemeterF in the course directory.

You may use any other Java or C# tool that has similar data-binding capabilities.

The details of the game will be introduced later. Basically, the players get a turn during which they create new challenges or buy one from one of the players. It is explicitly allowed to create a challenge with predicate T and price p even if there is already a challenge of the same predicate T but with a higher price offered by some player.

If the challenges are too expensive, nobody will buy them. For example, if all cost one (million). The rules of the game will make the players buy or lower the prices: Here is an important rule:

When it is a player's turn: The player must buy at least one challenge from another player or re-offer a fraction of all challenges that are for sale with a lower price. This will create new challenges of an existing predicate, one for each predicate. In addition, a player may create new challenges using new predicates. (The idea is that if a player does not want to buy, s/he must demonstrate that the prices are too high by lowering a fraction of them. Several challenges of the same predicate will be usually for sale and the clever buyer will normally choose the cheapest one.)

The game terminates after a fixed number of rounds (say 20). The winning player is the one who has the most energy (money) at the end of the game.

1. by careful thinking. Buy challenges where you are guaranteed to make money.

2. by exploiting mistakes of others: (a) Sell challenges where the buyer uses suboptimal techniques to create the finished product. This will lower the amount of money you have to pay for the finished product and will increase your profit. (b) Buy challenges where you think the seller will make a mistake by giving you raw material from which you can produce a high quality product.

So in order to play the game well you must be good at: 1. spotting the best buys (which includes spotting the bad buys) 2. Solving the problems well (creating a high quality finished product out of the raw material).

In addition it helps, if you know the shortcomings of the other players but this is not needed to win in this game.

Of course, this all depends on the beliefs and predicates used, and what it means to finish a raw product. This will be specified in project 2.

What to turn in for project 1/part 1: Turn in your object-oriented design, a DemeterF class dictionary or equivalent schema that can represent the required information. Turn in your parser and printer that has been generated from the schema.

The projects are done using pair programming. Find a partner and let me know if you cannot find one. Make sure you are a balanced team by comparing your answers to the questionnaire. We will be practicing pair programming:
http://www.ccs.neu.edu/home/lieber/courses/cs4500/f09/project/pair-prog/pair-prog.html

You must turn in a development diary for each project: http://www.ccs.neu.edu/home/lieber/courses/cs4500/f09/project/pair-prog/development-diary.html

Part 2
==========

Form a team of two students (the same pair as in part 1)
and play a few rounds of
SCG/classic/MAX-SAT (the raw materials are conjunctive normal forms) as described in Requirements Document .

This is an important document describing the requirements for the SCG.

Turn in the game history over 3 rounds.
In each round a player has one turn.

Follow the language definition for game histories in:

/h
http://www.ccs.neu.edu/home/lieber/courses/cs4500/f09/DemeterF/for-project1/abstract-histories/

to turn in your history.

Turn in your answers electronically on Blackboard. Header to use:

Software Development CS4500
Fall 2009
Project Number 1