Chance-It:
An Object-Oriented Capstone Project For CS-1

Joel C. Adams
Department of Computer Science
Calvin College
Grand Rapids, MI 49546
adams@calvin.edu

Abstract

Most people enjoy playing games. Most CS-1 students will enjoy a final project that involves computational game-playing. **Chance-It** is a simple two-person dice game with many possible strategies at varying levels of sophistication and complexity. These features make the problem of formalizing and encoding a strategy to play Chance-It an interesting final project for CS-1.

This paper describes an object-oriented final project for CS-1 in which students build `Player1` and `Player2` classes to play Chance-It. A `ChanceItGame` class and driver are provided to coordinate the interactions of these classes. The project provides students with an enjoyable introduction to object-oriented design and the problem of formalizing and codifying human strategy in software. Examples are given in C++, but convert easily to Java.

Introduction

In the words of the irrepressible Mary Poppins,

> *Just a spoon full of sugar helps the medicine go down,*
> *... in the most delightful way.*

The point of this song is that by combining something fun with a "good-for-you-but-difficult" activity, that activity can be made to seem less difficult.

This idea has been frequently applied in Computing. Thanks to the computer's ability to behave as a universal machine, computer scientists have found it relatively easy to explore many computational problems using games. In programming computers to "play at" psychotherapy [9], and to actually play checkers [6], chess [7], and other games [8], some of the most notable names in Computing have explored diverse problems within gaming frameworks. Games thus provide an enjoyable context in which computer scientists can perform serious work.

The idea of trying to make learning fun underlies much of educational software. In Computer Science education, this idea has been taken a step further by creative projects like the Karel the Robot [4] and Get-a-life [5] simulations, the Animal game [1], and the Nim tournament [3], in which students study, extend or create games as programming projects.

In this paper, we describe the game of Chance-It — a simple dice game — and how it has provided the basis for a CS-1 final project several times in recent years. The project involves three software components:

- a class to represent a game of Chance-It;
- classes to represent the players of the game; and
- a driver program to start the game

In the remainder of this paper, we describe the details of the Chance-It game, followed by the design, implementation, and use of these classes in a tournament setting.

Rules Of Play

The six rules governing Chance-It are fairly simple:

1. A game of Chance-It involves two *players*. The player with the highest total score after 100 *rounds* is the winner.

2. During a round, each player takes a *turn*. At the beginning of the game, each player rolls a dice, with the higher roll going first each round. A player's total score is the sum of their turn scores (see below).

3. At the beginning of their turn, a player rolls two six-sided dice and remembers the sum, called their *first roll*.

4. After their first roll, a player must decide whether to stop or continue (i.e., chance-it).

5. If a player decides to stop, then their turn ends and their score for that turn is their maximum roll during that turn; if the player decides to continue, they roll the dice again.

6. If, in re-rolling the dice, a player re-rolls their first roll, then their turn ends and their score for that turn is zero; otherwise, they must decide whether to stop or continue (and return to rule 5).

Playing The Game

To introduce students to the game, we always play one demonstration game in class using student "volunteers." Such a game might proceed as follows:

Suppose that Ann and Bob volunteer to play the game. They begin by each rolling a die: Ann rolls a 5 and Bob rolls a 3, so Ann goes first during each round.

Ann's first roll is a six. She decides to roll again, rolls a four, decides to roll again, rolls a nine and decides to stop. Her turn ends and her score is nine.

During his turn, Bob's first roll is a five. He decides to roll again, rolls an eight and decides to stop. His turn ends, and at the end of the first round, Ann leads nine to eight.

Ann begins the second round by rolling a four. She decides to roll again and rolls a six, decides to roll again and rolls another six, decides to roll again and rolls a ten and decides to stop, bringing her total score to nineteen.

Bob begins his second turn by rolling a seven. He decides to roll again and rolls a three. He decides to roll again and re-rolls a seven, his first roll. His turn ends and his score for that turn is zero. After two rounds, Ann has nineteen and Bob has eight.

Play continues in this fashion; the player with the highest total score at the end of 100 rounds is the winner.

A player may thus roll the dice as many times as they wish during their turn and add the highest of those rolls to their total score, provided they do not re-roll their first roll.

Designing the Game Software

In our CS-1 course, we model software development throughout the course using object-centered design [1]. The design of the software for this game provides a useful final case study for the students. In conducting this study, we identify these trivial objects (all representable by integers):

- a round
- a player's total score
- a player's turn score
- a player's first roll on their turn
- a player's maximum roll on their turn

as well as the following non-trivial objects:

- the Chance-It game
- the player who goes first each turn
- the player who goes second each turn
- two dice

The dice are simple to model in C++ using the `RandomInt` class given in [1]. (In Java, these dice can be modeled using Java's predefined `Random` class.) Since the remaining non-trivial objects cannot be directly modeled using existing classes or types, we build classes to do so.

Since this is CS-1, we provide the students with a code framework that they extend. This reduces the amount of code they must write, allowing them to focus on and master the project concepts, without being overwhelmed by details.

The Class Interfaces

Once we have our basic design, we begin to design the class interfaces, through which we guide the students.

The `ChanceItGame` class interface is quite simple: since the only thing one does with a game is *play* it, we provide a method to play the game, plus a constructor by which game objects can be created. In C++, we write:

```
class ChanceItGame
{
 public:
   ChanceItGame(int numberOfRounds = 100);
   void Play();
};
```

The interface for each player class is a bit more complicated, but fairly intuitive. Each player has public methods to construct itself, return its name, retrieve its score, and take a turn. In addition to these public methods, we have found it useful to add three private utility methods: one to update its local store using the opponent's last roll, one to raise its total score by a given amount, and one to decide whether or not to stop rolling.

By using object-centered design from the outset in CS-1, our students make a gradual transition from traditional topics to inheritance and object-oriented design (OOD) by the end of CS-1. The Chance-It player classes help students learn to apply OOD by consolidating the attributes each player has in common in a `Player` parent class:

```
class Player
{
 public:
  Player();
  int TakeTurn(int opponentsLastRoll);
  int Score() const;
  virtual string Name() const = 0;
protected:
  void RaiseScore(int howMuch);
  virtual void UpdateData(int oppLastRoll) = 0;
  virtual bool StopRolling() const = 0;
};
```

By prototyping those methods we expect the derived class to provide as *pure virtual functions* in the `Player` class, we ensure that derived classes maintain our `Player` interface.

The `Player1` and `Player2` classes are then derived from `Player` and inherit the common code; for example:

```
class Player1 : public Player
{
 public:
   Player1();
   string Name() const;
 protected:
   void UpdateData(int oppLastRoll);
   bool StopRolling() const;
};
```

Universities that do not teach inheritance in CS-1 can instead build a `Player1` class, copy it, and replace each occurrence of `Player1` in the copy with `Player2`.

The ChanceItGame Class

The code framework that we provide for our students includes the complete `ChanceItGame` class. Since a Chance-It game *has* two players, its data members include:

```
#include "Player1.h"
#include "Player2.h"

class ChanceItGame
{
 public:
   ChanceItGame(int numRounds = 100);
   void Play();
 private:
   Player1 player1;
   Player2 player2
   int roundsPerGame;
};
```

In addition to the two data members representing players, we include a `roundsPerGame` data member to store the number of rounds in a game. By having our class constructor method initialize this data member with a value it receives via a parameter (default value 100), users can play shorter or longer games if they wish:

```
inline ChanceItGame::ChanceItGame(int numRounds)
{
   roundsPerGame = numRounds;
}
```

We also provide the students with a simple driver program that reads the number of rounds the user wishes to play, instantiates a `ChanceItGame` object to play that many rounds, and then sends that object the `Play()` message:

```
#include <iostream.h>
#include "ChanceItGame.h"

int main()
{
   cout << "\nLet's play some ChanceIt!\n"
        << "\nHow many rounds (e.g., 100) ";
   int rounds;
   cin >> rounds;

   ChanceItGame aGame(rounds);
   aGame.Play();
}
```

The `Play()` method encodes the "rules of play" of a game of Chance-It as described earlier. Most of its time is spent in a for loop that counts for the requisite number of rounds. Each round, the body of the loop sends each of the game's players the `TakeTurn()` message, which returns the score that player received for that turn. Each player's name and score is displayed both as they roll and at the end of their turn, so that a player's "strategy" can be observed by watching their rolling behavior.

Once control leaves the loop, the game is over, so an if statement is used to identify the winner (if any) and display an appropriate message.

The C++ code for the `Play()` method is as follows:

```
int ChanceItGame::Play()
{
  int score1,
      score2,
      lastRoll1,
      lastRoll2 = 0;

  for (int rnd = 1; rnd <= roundsPerGame; rnd++)
  {
    // the first player's turn
    cout << "\n*** " << player1.Name()
         << ", round " << round << endl;
    lastRoll1 = player1.TakeTurn(lastRoll2);
    score1 += lastRoll1;
    // the second player's turn
    cout << "\n*** " << player2.Name()
         << ", round " << round << endl;
    lastRoll2 = player2.TakeTurn(lastRoll1);
    score2 += lastRoll2;
    // summarize round
    cout << "\n*****************************\n"
         << " Round " << rnd << " - "
         << player1.Name() << ": "
         << score1 << "; "
         << player2.Name() << ": " << score2
         << "\n*****************************\n"
         << endl;
  }
  // announce winner of game
  if (score1 > score2)
  {
    cout << "\n\n" << player1.Name()
         << " wins the game!" << endl;
    return 1;
  }
  else if (score2 > score1)
  {
    cout << "\n\n" << player2.Name()
         << " wins the game!" << endl;
    return 2;
  }
  else
  {
    cout << "\n\n" << "We have a tie!" << endl;
    return 0;
  }
}
```

We have written the `Play()` method to return an integer indicating which player won the game, or zero in case of a tie. We have used this feature in a more sophisticated driver program that plays a best-of-three match.

Whose Dice?

The question of who should "own" the dice is an interesting one. Since many board games come with their own dice, it might seem natural for the dice to be members of our `ChanceItGame` class. However, it is a player that *rolls* the dice. Since a player uses the dice, we define the dice as members of our `Player` class:

```
#include "RandomInt.h"

class Player
{
 public:
  Player();
  int TakeTurn(int opponentsLastRoll);
  int Score() const;
  virtual string Name() const = 0;
protected:
  void RaiseScore(int howMuch);
  virtual void UpdateData(int oppLastRoll) = 0;
  virtual bool StopRolling() const = 0;
 private:
  RandomInt die1, die2;
};
```

To initialize these data members, we supply a minimal constructor method for class `Player`:

```
inline Player::Player()
 : die1(1, 6), die2(1, 6)
{}
```

The students are free to extend this constructor as they add additional data members to class `Player`. To aid them, we spend class time helping the students identify some of a `Player`'s *has-a relationships*. Using our behavioral description of the game, students typically identify these:

- each player *has a* total score
- each player *has a* first roll in their turn
- each player *has a* maximum roll in their turn

We illustrate how to declare, initialize, and access a "total score" data member. Completing the class by doing so for the remaining objects is left for the students.

Strategies

To help students begin to formulate a strategy, we recommend that they play the game a few times, until they can identify what factors contribute to a successful strategy.

Depending on the relative intelligence of their strategy, students may elect to add additional data members to keep track of their opponents score, what round it is, and so on. For example, a simple strategy is to not stop until a 7 or higher has been rolled. A more complex strategy might use the probability of re-rolling the first roll, adjusted by factors such as what round it is, who is ahead, and so on.

We encourage students to develop *adaptive strategies* that change as the game progresses. For example, a player who is behind may decide to take greater risks as the game progresses, while a player who is ahead may decide to "play it safe" by being conservative. To do so, the student must minimally add data members that store their opponent's score, and what round it is.

If a game consists of too few rounds, the variance intrinsic to dice-rolling can allow poor strategies to defeat intelligent strategies. To minimize the likelihood of this, we recommend that games consist of 100 rounds.

The Player::TakeTurn() Member

In addition to the preceding code, the final thing we provide for our students is the `Player::TakeTurn()` method.

This method encodes the basic behavior of a player's turn. To permit a player to use an adaptive strategy, this method receives the opposing player's last roll via a parameter. It proceeds to perform the first roll using the player's dice, updates any local state information needed for the player's strategy, and then enters a "rolling loop" that continues until the player's `StopRolling()` method returns true, or the player re-rolls their first roll:

```
int Player::TakeTurn(int opponentsLastRoll)
{
  myFirstRoll = die1.Next() + die2.Next();
  myCurrentRoll = myFirstRoll;
  int turnScore = myFirstRoll;

  cout << "\n First Roll: " << die1
       << " + " << die2
       << " = " << myFirstRoll;

  UpdateData(opponentsLastRoll);

  for (;;)
  {
    if (StopRolling())
    {
      cout << " -  Stopping.\n" << endl;
      break;
    }
    else
      cout << " -  Continuing ... ";

    myCurrentRoll = die1.Next() + die2.Next();

    cout << "\n Next Roll: " << die1
         << " + " << die2
         << " = " << myCurrentRoll;

    if (myCurrentRoll != myFirstRoll)
      turnScore = Max(turnScore, myCurrentRoll);
    else
    {
      cout << " -  OOOOPS!\n" << endl;
      turnScore = 0;
      break;
    }
  }

  RaiseScore(turnScore);

  return turnScore;
}
```

Since the `StopRolling()` method makes a `Player`'s decision, all of a student's strategy must be encoded within that method. As a method, any information required to make the decision must be stored in the player's data members. In order for `StopRolling()` to base its decision on the most current information, students must define the `UpdateData()` method in such a way as to ensure that this information is correctly updated each turn.

Running A Clean Game

The `TakeTurn()` method returns the player's score for that turn. By doing so, `ChanceItGame::Play()` can keep its own record of a player's score, rather than relying upon a `Player`'s `Score()` method, which a clever "cheater" could modify to return an inflated score.

In order to prevent creative cheating (e.g., always returning a 12 from `TakeTurn()`, or substituting "loaded" dice), students are not permitted to make any modifications to the `TakeTurn()` method or the `RandomInt` class.

As it is, `Player2` has a slight advantage: where the player going first can know the total scores as of the end of the preceding round, the player going second has this same information *plus* `Player1`'s score in the current round. Our better students take advantage of this and define their `Player2::StopRolling()` method differently from their `Player1::StopRolling()` method.

The Tournament

The climax of our CS-1 course is our Chance-It double-elimination tournament, which we hold at the end of the semester. At the start of the tournament, students are placed in the first round of the winner's bracket in random order, with the first and second students chosen for a match supplying `Player1` and `Player2`, respectively.

The tournament follows the usual double-elimination format: the loser of a match moves from the winner's bracket to a "not-winners" bracket, with the winners of the two brackets facing off in the final match. (The winner of the "not-winner's" bracket must defeat the winner of the winner's bracket *twice* to win the tournament.)

Managing The Tournament

One problem in staging the tournament is getting the C++ compiler to find the students' `Player1` and `Player2` classes. To achieve this, the compiler must be instructed to search the student directories (e.g., using the -I switch for UNIX C++ compilers) in which those classes are stored.

Another problem is that if each student has their `Player1` and `Player2` classes in the same directory, then the compiler will find both the `Player1` and `Player2` classes in whichever student's directory it searches first. We solve this by requiring that each student store their `Player1` and `Player2` classes in distinct directories. We then direct the compiler to search one student's `Player1` directory, and the other student's `Player2` directory.

To simplify the compilation, we use a `Makefile` that defines macros for the paths to these directories. This allows the binary for a particular tournament to be created just by changing these macros to the paths appropriate for that match, and then using `make` to perform the translation.

Since translation takes about as long as a match, a teaching assistant does the translation for the next match while the current match is being played. (If students store their classes in uniform locations, this is easily automated.)

Closing Remarks

We believe that a carefully devised game can serve as "sugar" that helps students "swallow" difficult concepts, without losing the "nutritional content" of those concepts. To that end, we have described a gaming tournament that, used as a CS-1 final project, can help students understand object-oriented concepts. In our experience, students are motivated by the project because (i) the problem of endowing software objects with "intelligence" is an interesting one, (ii) the provided code framework gives students a starting point on the project, and (iii) the tournament puts each student's work on public display.

Since most students have used dice, the game itself presents few conceptual obstacles for students. In addition, the plethora of possible strategies allows students of different abilities to complete the project and participate in the tournament. Such participation seems to bring a positive sense of *closure* to our students' CS-1 experience.

The user-interface for our game/tournament is at present text-based. We plan to add a graphical user interface (GUI) showing the player's dice rolls and running totals.

We appreciate the sponsorship of Metrowerks [2], which donates prizes for our tournament.

References

1. Adams, J., Leestma, S., Nyhoff, L., *C++ An Introduction to Computing* (2cd Ed.), Prentice-Hall, Englewood Cliffs, New Jersey, 1997.

2. Metrowerks Corporation. *http://www.metrowerks.com*.

3. Pargas, R., Underwood, J. and Lundy J., Tournament Play in CS1, *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education* (March, 1997), 214-218.

4. Pattis, R., *Karel The Robot: A Gentle Introduction To The Art Of Programming*, Wiley, New York, 1981.

5. Pattis, R., Teaching OOP in C++ Using an Artificial Life Framework, *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education* (March, 1997), 39-43.

6. Samuel, A. L., Some Studies In Machine Learning Using The Game Of Checkers. *IBM Journal of Research and Development (3)*, 1959, 210-229.

7. Simon, H.A. and Munakata, T. AI Lessons, *Communications of the ACM* (August, 1997), 23-25.

8. Turing, A.M., Strachey, C., Bates, M.A., and Bowden, B.V., Digital Computers Applied To Games. In B.V. Bowden, Ed. *Faster Than Thought*, Pitman, London, 1953.

9. Weizenbaum, J., *Computer Power and Human Reason*, W.H. Freeman and Company, 1976.