Fantasy League Object Oriented Development (FLOOD)

Language Tutorial & Reference Manual

COMS W4115: Programming Language & Translators

Stephanie Aligbe
sna2111@columbia.edu

Elliot Katz
epk2102@columbia.edu

Tam Le
tvl2102@columbia.edu

Dillen Roggensinger
der2127@columbia.edu

Anuj Sampathkumaran
as4046@columbia.edu

March 23, 2011

Contents

1

Introduction

As defined on Wikipedia: *fantasy sport (also known as* ***rotisserie****,* ***roto****, or* ***owner simulation****) is a game where participants act as owners to build a team that competes against other fantasy owners based on the statistics generated by the real individual players or teams of a professional sport.*

The popularity of fantasy sports has exploded in recent years. A 2007 study by the *Fantasy Sports Trade Association (FSTA)* estimated nearly 30 million people in the U.S. and Canada, ranging in age from 12 and above, participated in organized fantasy sports leagues. In comparison, an estimated 3 million people played in the early 1990s, swelling to 15 million by the early 2000s. This impressive growth looks to continue since the study revealed teenagers in both the U.S. and Canada play fantasy sports at a higher rate than the national average, with 13 percent of teens playing in the U.S. and 14 percent playing in Canada.

The FSTA study also estimated the spending habits and overall economic impact of fantasy sports players. Consumers engaging in this ever growing hobby spent $800 million directly on fantasy sports products and an additional $3 billion worth of related media products (such as DirecTV's NFL Sunday Ticket and satellite radio's coverage of MLB). Moreover, the growing popularity of fantasy sports is not restricted to North America alone. A recent 2008 study by a European-based market research company estimated the number of

fantasy sports players in Britain range between 5.5 and 7.5 million and vary in age between 16-64, of which 80 percent participated in fantasy soccer.

Thus, the **FLOOD** programming language is targeted to address a specific problem domain: the creation of fantasy gaming league applications. From the ground up, the language is designed to make it as straitforward as possible for programmers to create fantasy gaming applications. This task will entail defining a league and its type (sports, financial markets, election polling etc.), establish the rules of governance, enumerate the users/teams and individual league players, and set various other control parameters.

In this sense, **FLOOD** is very much a "high-level" and "domain-specific" language. Such as **R** and **S** are languages designed specifically to perform statistics calculations, **SQL** for relational database queries, and **Mathematica** and **Maxima** for symbolic mathematics, **FLOOD** is dedicated to solving the particular problem of fantasy gaming. Given the ubiquity of fantasy sports as a recreational hobby for millions of people, the creation of a domain-specific language to attack this problem in a clear and concise native programming etymology, as opposed to the application of a general purpose language such as **C** and **Java**, is a challenging and worthwhile undertaking.

2

A Quick Tutorial

The goal of this tutorial is to jump into the **FLOOD** language by creating a simple fantasy league. We won't concern ourselves with implementation details just yet. The language itself is meant to be simple yet sufficiently extensible, however the tutorial will not delve into the various features that make this possible.

Experienced Java programmers will be familiar with most of the syntax since it is a subset of the popular C-like syntax. In certain instances we have opted to use Python syntax for improved readability. Where possible we compare or contrast a specific feature with Java, C++, or Python which comprise the main influences of **FLOOD**.

Some examples that aren't strictly necessary for the sample program are included to clarify points of possible confusion. These are noted when used. Additionally, we repeat portions of the **FLOOD** standard library for ease of reading. The full library is included in the *Reference Manual*. Please be aware of the distinction made between programmer, the person actually writing a **FLOOD** program (most likely yourself if you are reading this), and *User*, the end-user of the application. Additionally, *Player* refers to a player of the fantasy league, whether this is an actual person as in the case of a quarterback in football or a non-human thing as is the case with $AAPL$ (Apple) in the stock market.

2.1 Getting Started - The "Hello World" Program

FLOOD has a very specific application domain. It's goal is to create fantasy leagues therefore the ability to print to console is fruitless. However, the GUI will have a message box where messages can be displayed to the User. **FLOOD** hides the details of the user-interface so that the programmer can concentrate on the rules of their specific fantasy league. While the first simple program may seem more complicated than a Java "hello world" example, most programs generally will not be any more elaborate than this case.

Since **FLOOD** is built on top of Java byte-code, the programmer will not need to worry about system compatibility issues. Heeding the eternal wisdom of Kernighan and Ritchie, we also must point out that compilation will only succeed if the programmer hasn't botched anything such as missing characters, misspelling keywords or some other mistake that can cause errors in compilation.

Let's start by showing what the "main" class would look like. In keeping with the theme of fantasy leagues the function *main* has been replaced by the equivalent *play*.

Listing 2.1: BasketballPlay.fld

```
1  class BasketballPlay:
2      public play() {
3          League myfbl = new League (  Happy League  );
4          mybl.setMaxUsers(10);
5          mybl.setMinUsers(4);
6
7          mybl.addUser(new User(  Eli  ));
8          mybl.addUser(new User(  Dillen  ));
9          mybl.addUser(new User(  Anuj  ));
10         mybl.addUser(new User(  Tam  ));
11         mybl.addUser(new User(  Steph  ));
12
13         mybl.addAction(new Action(  Field Goal Attempt  , -0.45);
14         mybl.addAction(new Action(  Field Goal Made  , 1.0);
15         mybl.addAction(new Action(  Free Throw Attempt  , -0.75);
16         mybl.addAction(new Action(  Free Throw Made  , 1.0);
17         mybl.addAction(new Action(  3-Point Shot Made  , 3.0);
18         mybl.addAction(new Action(  Point Scored  , 0.5);
19         mybl.addAction(new Action(  Rebound  , 1.5);
20         mybl.addAction(new Action(  Assist  , 2.0);
21         mybl.addAction(new Action(  Steal  , 3.0);
```

```
22          mybl.addAction(new Action(  Turnover  , -2.0);
23          mybl.addAction(new Action(  Blocked Shot  , 3.0);
24
25          SnakeDraft blDraft = new SnakeDraft(mybl);
26
27          Flood.launchGui(blDraft, mybl);
28      }
```

At first glance this appears to be very close to a Java program. Let's point out a couple of differences. First, note the convention for **FLOOD** files is to name them with the *.fld* extension. Secondly, each file must contain one and only one class—as oppose to Java, there is no ability to nest classes.

Classes are defined using the semicolon. Since there is no question of scope within a file, code below the class definition is considered part of the class. Code above the class definition is an error. Import statement must immediately follow the class definition. Instance variables can be placed anywhere below the class definition outside of method blocks.

Java uses the *main* method as the beginning of execution. In keeping with the theme of fantasy leagues, the *play* method is used as the equivalent of *main*. Every **FLOOD** program must have a *play* method. **FLOOD** convention dictates that the class containing *play* be named after the type of league being designed. For instance, in this example the type of league can clearly be inferred to be related to basketball.

Listing 2.2: A basketball league

```
1 League myfbl = new League (  Happy League  )
2 mybl.setMaxUsers(10);
3 mybl.setMinUsers(4);
```

Every **FLOOD** program must instantiate a League object. The League object contains most of the other objects needed for the **FLOOD** program including User, Player and Action lists. Lists will be discussed in later sections but for now they can be thought of as Java LinkedLists (although there are some subtle differences). For the minimum program we could have left the max and min user setting at their respective default value but as an example we set them above. The League object will later be passed to the GUI however it

provides the first example of extension within **FLOOD**. We could have written a new class that extends League to add functionality such as divisions within the league. For now the basic League object will suffice to create our first program.

A method in **FLOOD** is similar to a method in Java or function in C++. We call a method on an object by using the object's name followed by a period followed by the method of that object with an optional argument list in between mandatory parenthesis.

Listing 2.3: Setting maximum users

```
1 mybl.setMaxUsers(10);
```

In this example we are calling *setMaxUsers* on our *mybl* object. The *mybl* object is an instantiation of the League class. The method *setMaxUsers* has been defined to take an argument of type int. We pass the value to 10 to the method which performs some action internally. In this case, it sets the maximum number of user in the league to 10.

Listing 2.4: Adding a User

```
1 mybl.addUser(new User(  Eli  ));
```

We then begin adding users to the league. Once again we use this as an illustration of what can be added. We add five Users and give each names. It would have been possible to leave this portion of later in the program.

Listing 2.5: Adding an Action

```
1 mybl.addAction(new Action(  Field Goal Attempt  , -0.45);
```

Actions are added to the league using the *addAction* method. Actions consist of any event that is associated with a point value, positive or negative, which are part of the evaluation of the strength of a User (or team). The actions will be associated with a file uploaded each time-period which will contain player-action statistics. The statistics will be translated via this list of actions to points added to each User. The league will then know while Users are "winning" and rank them.

Listing 2.6: Launching the GUI

```
1 Flood.launchGui(blDraft, mybl);
```

The final part of the code launches the GUI and adds both the League and the Draft. Hiding the GUI is one of the main features of **FLOOD**. There is no need to program any part of the user-interface. The GUI will know how to hook into the **FLOOD** classes and connect the buttons on the interface to the actions defined by the programmer. The output will be a GUI window which will control the flow of the program. The flow of the program will depend upon the user. For instance, the only way to update the scores of the Users is by adding player-action files to the program. Player-Action files would contain new statistics such as:

Listing 2.7: Sample Player statistics

```
1 LeBron James        7 Rebound
2 Lebron James        4 Assist
3 Carmelo Anthony     5 Steal
4 Carmelo Anthony     25 Point Scored
```

This is everything needed to run a simple **FLOOD** program. This basketball league would mirror equivalent leagues in Yahoo Sports or ESPN Fantasy sans network connectivity.

2.2 Variables & Arithmetic Expressions

Just as any robust programming language requires a comprehensive computational model, **FLOOD** provides the user with a set of arithmetic expressions and variable types to work on. To use a variable and work on it, it requires to be declared at or before first usage. A declaration defines the properties of the variables. A declaration is of the form, type name followed by the list of variables to be declared as that type. This list needs to be comma separated such as:

Listing 2.8: User.fld

```
Class User:
    get int points;
    get str name, handle;
    setget int maxSize;
    List <Players> teamAthletes;

    addPlayer(Player athlete);
    removePlayer(Player athlete);
```

Here, the variables "name" and "handle" are both declared as type "str" (string). Notice that here, the variables are declared at the very beginning. Alternatively, like mentioned earlier, variables can be declared as and when needed. For example:

Listing 2.9: SnakeDraft.fld

```
Class SnakeDraft is Draft:
    private League game;

    public draftFunction(int turn) {
        return turn \% game.teams.length();
    }

    /* Limit of 8 players per team, and there must be at least 1 center, 2
       guards and 2 forwards per team. */
    public bool pickPlayer(User team, Player athlete) {
        int centers = 0, guards = 0, forwards = 0, total = team.
           teamAthletes.size();
        if (total < 8) {
            if (total < 4) {
                team.addPlayer(athlete);
                return true;
            }
    }
```

Here, the integer variables centers, guards, fowards and total are declared right when they are needed. This obviates the need to keep going back to the beginning of the code to declare variables, making declarations much more convenient.

FLOOD offers the set of arithmetic expressions required to create a comprehensive fantasy league of the users choice. This set comprises of the standard addition, subtraction, multiplication and division operators.

Listing 2.10: SnakeDraft.fld

```
/* Limit of 8 players per team, and there must be at least 1 center, 2
   guards and 2 forwards per team. */
```

```
2  private bool evaluate(User u, Player p) {
3      int centers = 0, guards = 0, forwards = 0, total = u.teamAthletes.size
           ();
4      if (total < 8) {
5          if (total < 4) {
6              return true;
7          }
8
9          for person in u.teamAthletes {
10             if (person.position ==   center  )
11                 centers = centers + 1;
12             if (person.position ==   guard  )
13                 guards = guards - centers;
14             if (person.position ==   forward  )
15                 forwards = forwards / 2;
16             if (person.position ==   mid  )
17                 mid++;
18             if (centers == 3 and p.position ==   center  )
19                 return false;
20             if (forwards == 5 and p.position ==   forward  )
21                 return false;
22             if (guards == 5 and p.position ==   guard  )
23                 return false;
24
25             return true;
26         }
27
28         return false;
29     }
30 }
```

This code snippet provides a glimpse of the arithmetic capabilities of **FLOOD**. The '+', '-', '/', and '*' operators are binary operators and can be used to add, subtract, divide and multiply floats and integers.

2.3 Loops: For and While

Creating a fantasy league in **FLOOD** can range from simple computations to complex algorithms involved in drafts. To facilitate the latter, the programmer has the choice of using loops to make life easier.

The syntax for the while loop follows the standard convention:

Listing 2.11: While loop

```
while (total < 4) {
    centers = centers + 1;
    guards = guards - centers;
    forwards = forwards / 2;
    total++;
}
```

The while loop operates as follows: The condition in parentheses is tested. If it is true (total is less than 4), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is checked again, and if true, the body is executed again. When the test becomes false (total equals or exceeds 4) the loop ends, and execution continues at the statement that follows the loop. The body of a while can be one or more statements enclosed in braces, as above, or a single statement without braces, as in

Listing 2.12: While loop

```
while (total < 4)
    centers = centers + 1;
```

The for statement is a loop, a generalization of the while. **FLOOD** provides a easy to use for loop syntax that allows the user to iterate through any list, for example:

Listing 2.13: While loop

```
/* Limit of 8 players per team, and there must be at least 1 center, 2
   guards and 2 forwards per team. */
private bool evaluate(User u, Player p) {
    int centers = 0, guards = 0,forwards = 0, total = u.teamAthletes.size()
       ;
    if (total < 8) {
        if (total < 4) {
            return true;
        }

        for person in u.teamAthletes {
            if (person.position ==   center  )
                centers++;
            if (person.position ==   guard  )
                guards++;
            if (person.position ==   forward  )
                forwards++;
            if (centers == 3 and p.position ==   center  )
                return false;
            if (forwards==5 and p.position ==   forward  )
```

```
19                  return false;
20              if (guards == 5 and p.position ==   guard  )
21                  return false;
22
23              return true;
24          }
25
26          return false;
27      }
28  }
```

Here, *u.teamAthletes* is a list of *person* and typically, the user may need to iterate through this list. This *for* loop allows the user to iterate through this list conveniently without having to manually find out the length of the list or worry about the starting position of the list.

2.4 Conditionals

FLOOD provides the programmer with a conditional in the form of the 'if' expression that is defined as follows:

Listing 2.14: While loop

```
1  if (person.position ==   center  )
2      centers++;
3  else
4      centers --;
```

Here, the program checks the condition enclosed in the bracket. If this condition is met, in this case, if the person's position is "center," then the program executes the next statement. In order to include multiple statements to be executed in case the condition is met, the '{..}' parenthesis pair can be used as follows:

Listing 2.15: While loop

```
1  if (person.position ==   center  ) {
2      centers++;
3      forwards ++;
4  }
5  else {
6      centers --;
```

```
7       forwards --;
8   }
```

If this condition is not met, then the statement or statements enclosed in body of the else condition will be executed i.e. decrement the value of the variables, 'centers' and 'forwards.'

2.5 Essential FLOOD Classes

The **FLOOD** language has 5 base classes necessary for the creation of a fantasy league. They are described individually below in detail.

Action

Action is a class that encapsulates an event in a league along with an associated point value. These actions are then put into the league in order to distribute points among the users depending on how their players perform. For example, for a basketball fantasy league will have an actions as follows:

Listing 2.16: Action

```
1    Field Goal Attempt     <=>      -0.45
2    Field Goal Made        <=>      1.0
```

which state that any player that performs that action will have the corresponding value added to the player's team. The base class has two instance variables as seen above with respective getters and setters but can be expanded as necessary.

Player

A Player is a unit tradable entity in the league. For example, Michael Jordan is a Player in the Basketball league. This base class provides a set of data structures that entirely define a unit player. Effectively, this base class provides a group of setters that allows the player name and position attributes set for each Player object.

User

A User is a person who play the fantasy league. Each user has a name, points and a list of Players associated with it. The list of Players is considered to be the User's team. There are a set of functions that can be used to add and remove players from a particular users team:

Listing 2.17: Action

```
addPlayer(Player athlete);
removePlayer(Player athlete);
```

which can be used in the following manner:

Listing 2.18: Action

```
/* Add a player to a team if possible* /
public bool pickPlayer(User u, Player p) {
    bool addition;
    addition = evalutate(u,p);

    if (addition) {
        u.addPlayer(p);
        return true;
    }

    return false;
}
```

In the above code snippet, an object of class Player, p, is added to an object of class User, u, using the function addPlayer provided by the User class. Similarly, the removePlayer function can be used to remove Player objects from a User object.

Draft

A Draft is a class that specifies different rules and regulations regarding the addition, trade or drop of a Player from a User's team. The base class provides a very broad implementation that applies minimal restrictions but is fully expandable. Its methods include the following:

Listing 2.19: Action

```
public int draftFunction(int turn)
public bool pickPlayer(User u, Player p)
public bool trade(User u1, Player p1, User u2, Player p2)
public dropPlayer(User, Player)
public playersLeft()
```

For example, when creating a draft for a basketball league, it is necessary that every team has at least 1 center, 2 forwards and 2 guards. Below is a possible function to enforce this, setting the max number of players per team to be 8. The evaluate function is a private method used to say if a team can afford to take the player trying to be drafted without violating the constraints concerning the number of each type of specific player.

Listing 2.20: Action

```
/* Add a player to a team if possible */
public bool pickPlayer(User u, Player p) {
    bool addition;
    addition = evaluate(u,p);
    if (addition) {
        u.addPlayer(p);
        return true;
    }

    return false;
}

/* Limit of 8 players per team, and there must be at least 1 center, 2
   guards and 2 forwards per team. */
private bool evalutate(User u, Player p) {
    int centers = 0, guards = 0,forwards = 0, total = u.teamAthletes.size()
       ;

    if (total < 8) {
        if (total < 4) {
            return true;
        }

        for person in u.teamAthletes {
            if (person.position ==   center  )
                centers++;
            if(person.position ==   guard  )
                guards++;
            if(person.position ==   forward  )
                forwards++;
            if(centers==3 and p.position ==   center  )
                return false;
            if(forwards==5 and p.position ==   forward  )
                return false;
            if(guards == 5 and p.position ==   guard  )
                return false;

            return true;
        }

        return false;
    }
```

```
41 }
```

League

The League class is the pivot of all the base classes. It provides a set of variables and functions that are needed to comprehensively define any fantasy league in **FLOOD**. The variables and functions in this class are as follows:

2.6 Scope

FLOOD scope modifiers provide the ability to limit access to methods and fields as is warranted by the program. It enables encapsulation of data within classes and the ability to hide the data from outside access. **FLOOD** also provide novel approaches using *set*, *get*, and *setget*. These modifiers default to private but generate setters and getters respective to their names. Public scope exposes a method or field to all outside access, and private scope completely hides a method or field from all outside access.

3

Reference Manual

3.1 Introduction

3.2 Lexical Conventions

3.2.1 Tokens

3.2.2 Comments

3.2.3 Identifiers

3.2.4 Keywords

3.2.5 Constants

3.3 Syntax Notation

3.3.1 Expressions

3.3.2 Declarations

3.3.3 Statements

3.3.4 Scope and Linkage

3.4 Object Orientated Programming

FLOOD is an objected oriented programming (OOP) language by design so in order to become an adept coder, a fundamental understanding of OOP concepts is necessary. To get

started, in this section we review the four basic tenets of OOP:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3.4.1 Abstraction

In general, an abstraction is a model or an ideal—although not all of the details are present, the general parameters are, which then can be filled in with details. Furthermore, an abstraction is clear enough to tell one abstraction from another.

As a concrete example, say a software company has two job openings to fill, the first for a web designer and the second for a web programmer. In advertising for the position, the company would not describe a specific person to fill each position but instead in more general terms consisting of the skills and experience needed of each candidate to fill the positions. Hence, the listings for the jobs will be two *abstractions* representing the two separate positions:

- Web Programmer
  - Experience programming in a team-based enviornment.
  - Experience with middleware and database programming.
  - OOP and Design Pattern programming skills.
- Web Designer
  - Experienced with creating web graphics.

– Familiarity with animation graphics.

– Experience with vector graphics.

Discerning the differences between the two positions and their broad requirements are straightforward enough, but the particular details are left relatively open-ended. For example, a programmer is unlikely to apply for the Designer position and a designer is just as unlikely to apply for the Programmer position. However, a pool of applicants could have a wide range of skills which would serve as the concrete details for each position—one applicant for the Programmer position may have Java middleware programming experience in addition to Oracle database skills while another may have experience in .NET and MS SQL Server. In this particular case, the *abstraction* is the Programmer job description and the details are filled in by each applicant's unique skill set and experience.

In *Object-Oriented Design with Applications* (Benjamin/Cummings), Grady Booch, a design pattern pioneer, provides the following definition of *abstraction* that is both clear and succinct:

> *An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.*

Booch's definition describes in a well-defined manner the two job descriptions. Each description provides the essential and necessary characteristics of the position while distinguishing from one another.

Listing 3.1: FLOOD Abstraction example

```
1 Sample abstraction code...
2 Sample abstraction code...
3 Sample abstraction code...
```

3.4.2 Encapsulation

In the context of OOP, encapsulation is often presented in terms of the *black box*, the notion in which the internal representation and inner workings of an object, for all intents and purposes, are hidden from outside view and inspection.

We encounter examples of *black boxes* in every day life. For example, the automobile. A car owner get into the driver's seat, inserts and turns his or her key, starts the engine, presses down on the gas pedal to accelerate and alternately the break pedal to decelerate and stop. These interactions between car and driver are enough to operate the vehicle but the majority of car owners do not know, nor usually care, about the exact science and engineering principles behind the multitude of technologies which actually make a car work. Automobiles are not transparent. They're black boxes.

Hence, an overriding benefit derived from the concept of a black box is that users need not be knowledgeable of nor worry about the myriad complexities involved in the inner workings of the box. End users just have to know how to interact with it, secure in the notion that whatever makes the black box works does so according to design.

Listing 3.2: FLOOD Encapsulation example

```
1 Sample encapsulation code...
2 Sample encapsulation code...
3 Sample encapsulation code...
```

3.4.3 Inheritance

The third key concept of OOP is inheritance. In the simplest terms, inheritance refers to how one class inherits the properties and methods of another class. If **Class A** has methods X(), Y(), and Z(), and **Class B** is a subclass of **Class A** (extends), it too will have methods X(), Y() and Z(). **Class B** is known as the *subclass* (or *derived* class) and **Class A** is the *superclass* (or *ancestor* class).

Listing 3.3: FLOOD Inheritance example

```
1 Sample inheritance code...
2 Sample inheritance code...
3 Sample inheritance code...
```

3.4.4 Plymorphism

Listing 3.4: FLOOD Plymorphism example

```
1 Sample plymorphism code...
2 Sample plymorphism code...
3 Sample plymorphism code...
```

3.5 Grammar

4

Standard Library

4.1 Football

4.2 Basketball

4.3 Rugby

4.4 Financial Markets