- Michael Winslow
- http://michaelwinslow.net/
- Active: 1980-present

- http://doodle.com/vx6qrw662fhcp46i?
- Homework problems for Thu

- Lab 8: Random numbers

```
from random import *

randrange(0,100,10)
random()
```

Simulating Racquetball

- *Simulation* can solve real-world problems by modeling real-world processes
- Provide otherwise unobtainable information
- Computer simulation is used to predict the weather, design aircraft, create special effects for movies, etc.
- *http://www.youtube.com/watch?v=4jf8W2rwbXw*

The Challenge!

- *Denny Dibblebit* often plays racquetball with players who are slightly better than him
- Denny usually loses his matches!
- Shouldn't players who are *a little* better win *a little* more often?
- Write a simulation to see if slight differences in ability can cause such large differences in scores

The Software Development Process

- **Analyze the Problem**
- **Determine Specification**
- **Create a Design**
- **Implement Design**
- **Test/Debug the Program**
- **Maintain Program**

Analysis and Specification

- Racquetball is played between **two players** using a racquet to **hit a ball in a four-walled court**
- One **player starts** the game by putting the ball in motion – ***serving***
- Players try to **alternate hitting the ball** to keep it in play, referred to as a ***rally***. The rally **ends** when one player **fails to hit a legal shot**

Analysis and Specification

- The player who **misses the shot loses the rally**. If the loser is the player who served, service passes to the other player
- If the **server wins the rally, a point is awarded**. Players can only score points during their own service
- **First player to reach 15 points** wins the game

Analysis and Specification

- In our simulation, ability level of player will be represented by probability that player wins rally when serving
- **Example**: Players with 0.60 probability win a point on 60% of their serves
- Program will
  1. prompt the user to enter the service probability for both players
  2. simulate multiple games of racquetball
  3. print a summary of the results

Analysis and Specification

- **Assume:**
  - All inputs are legal numeric values: no error or validity checking required
  - In simulated game, player A serves first

Analysis and Specification

- **Input:** Prompt for
  - Service probabilities of players A and B
  - Number of games to be simulated

- **Output:** Prints out report showing
  - number of games simulated
  - number of wins and winning percentage for each player

```
Games simulated: 500
Wins for A: 268 (53.6%)
Wins for B: 232 (46.4%)
```

PseudoRandom Numbers

- Racquetball simulation can use of the `random` function to determine if a player has won a serve
- Suppose a player's service probability is 70%, or 0.70

```
if <player wins serve>:
   score = score + 1
```

- Need to insert probabilistic function that will succeed 70% of the time

PseudoRandom Numbers

- Generate a random number between 0 and 1
  - Exactly 70% of the interval 0..1 is to the left of 0.7
- 70% of the time the random number will be < 0.7,
  - Will be ≥ 0.7 the other 30% of the time
  - (The = goes on the upper end since the random number generator can produce a 0 but not a 1.)

PseudoRandom Numbers

- If `prob` represents the probability of winning the server, `random() < prob` will succeed with the correct probability

```
if random() < prob:
    score = score + 1
```

Analysis and Specification

- **Input:** Prompt for
  - Service probabilities of players A and B
  - Number of games to be simulated

- **Output:** Prints out report showing
  - number of games simulated
  - number of wins and winning percentage for each player

```
Games simulated: 500
Wins for A: 268 (53.6%)
Wins for B: 232 (46.4%)
```

Design?

Top-Down Design

- In top-down design, complex problem expressed as solution in terms of smaller, simpler problems
- Smaller problems are then solved by expressing them in terms of smaller, simpler problems
- Continue until problems are trivial to solve. Then, little pieces are put back together as a solution to the original problem

Summary of Top-Down Design

1. Express the algorithm as a series of smaller problems
2. Develop an interface for each of the small problems
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems
4. Repeat the process for each smaller problem

Top-Level Design

- Typically a program uses the input, process, output (IPO) pattern
- Proposed design/algorithm for racquetball simulation:

```
Print an introduction
Get the inputs: probA, probB, n
Simulate n games of racquetball using probA and probB
Print a report on the wins for playerA and playerB
```

Top-Level Design

- Too high level? Whatever we don't know how to do, we'll ignore for now
- Assume that all components needed to implement algorithm have been written already by your lab partner. Your task is to finish this top-level algorithm using those components

Top-Level Design

- First print an introduction
- This is easy; we don't want to bother with it

```
def main():
    printIntro()
```

- Assume there's a printIntro function

Top-Level Design

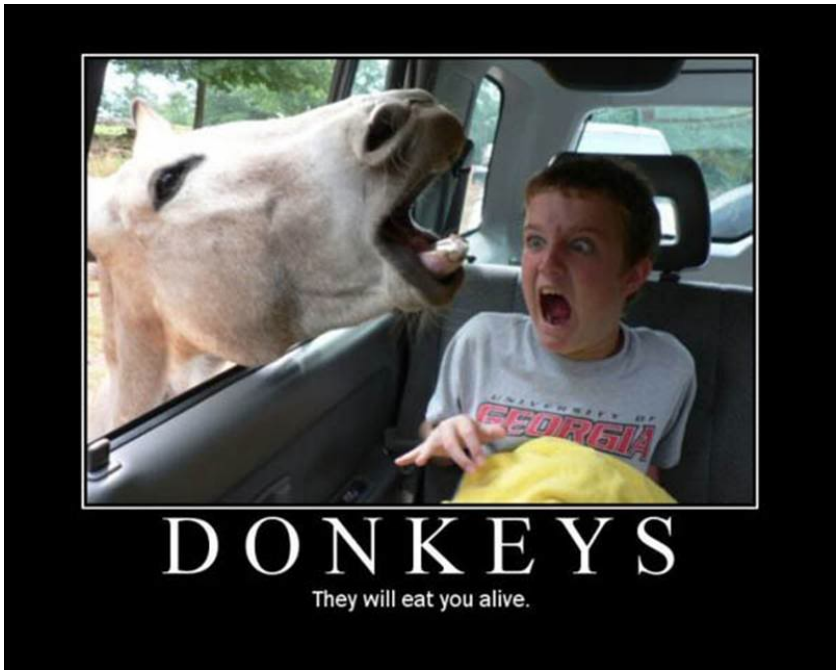
- The next step is to get inputs
- We know how to do that. Let's assume there's already a component that can do that called getInputs
- getInputs gets the values for `probA`, `probB`, and `n`

- ```
  def main():
      printIntro()
      probA, probB, n = getInputs()
  ```


  ```

Top-Level Design

- Now need to simulate *n* games of racquetball using values of `probA` and `probB`
- How would we do that? We can put off writing this code by putting it into a function, `simNGames`, and add a call to this function in `main`



Top-Level Design

- If you were going to simulate game by hand, what inputs would you need?
  - `probA`
  - `probB`
  - `n`
- What values would you need to get back?
  - number of games won by player A
  - number of games won by player B
- Defines inputs and outputs for `simNGames` function

Top-Level Design

- Now know that main program must look like this:

```
def main():
   printIntro()
   probA, probB, n = getInputs()
   winsA, winsB = simNGames(n, probA, probB)
```

- What information would you need to be able to produce the output from the program?
- You'd need to know how many wins there were for each player – these will be the inputs to the next function

Top-Level Design

- The complete main program:

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

Separation of Concerns

- Original problem has now been decomposed into four independent tasks:
  - `printIntro`
  - `getInputs`
  - `simNGames`
  - `printSummary`
- Names, parameters, and expected return values of these functions have been specified. This information is known as the interface or signature of the function

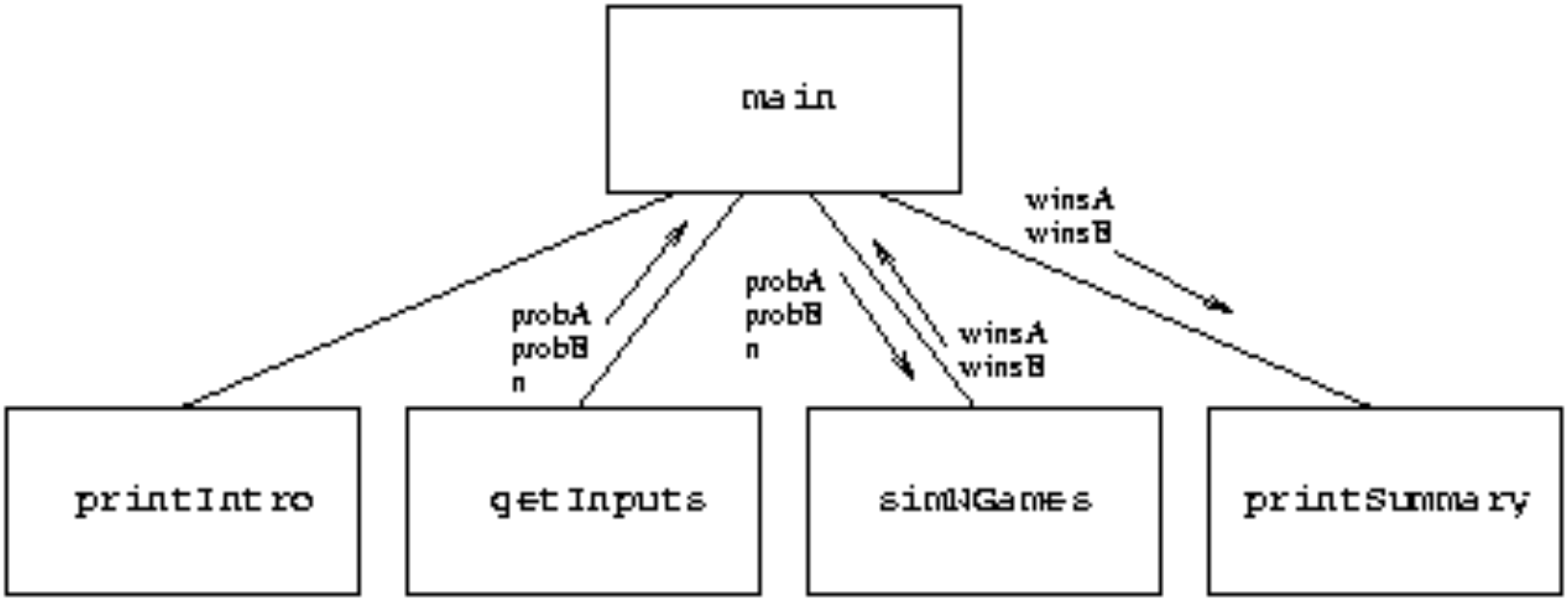
Separation of Concerns

- Having this information (the interfaces), allows us to work on each of these pieces independently.
- For example, as far as `main` is concerned, *how* `simNGames` works is not a concern as long as given inputs produces correct outputs

Separation of Concerns

- In a structure chart (or module hierarchy), each component in design is a rectangle
- A line connecting two rectangles indicates that the one above uses the one below
- Arrows and annotations show the interfaces between components

Separation of Concerns



Separation of Concerns

- At each level of design, interface tells us which details of lower level are important
- General process of determining important characteristics of something and ignoring other details is called abstraction
- Top-down design process is a systematic method for discovering useful abstractions

Second-Level Design

- Next step: repeat process for each of the modules defined in previous step
- The `printIntro` function should print an introduction to program. Code for this is straightforward

Second-Level Design

```
def printIntro():
    # Prints an introduction to the program
    print("This program simulates a game of racquetball between two")
    print('players called "A" and "B".  The abilities of each player is')
    print("indicated by a probability (a number between 0 and 1) that")
    print("the player wins the point when serving. Player A always")
    print("has the first serve.\n“)
```

- In second line, since we wanted double quotes around A and B, string is enclosed in apostrophes
- No new functions ➔ no changes to structure chart

Second-Level Design

- In `getInputs`, prompt for and get three values, which are returned to main program

```
def getInputs():
    # RETURNS probA, probB, number of games to simulate
    a = eval(input("What is the prob. player A wins a serve? "))
    b = eval(input("What is the prob. player B wins a serve? "))
    n = eval(input("How many games to simulate? "))
    return a, b, n
```

Designing simNGames

- Function simulates *n* games and keeps track of how many wins there are for each player
- "Simulate *n* games" sound like a counted loop, and tracking wins sounds like a good job for accumulator variables

Designing simNGames

```
• Initialize winsA and winsB to 0
  loop n times
      simulate a game
      if playerA wins
          Add one to winsA
      else
          Add one to winsB
```

Designing simNGames

- We already have the function signature:

```
def simNGames(n, probA, probB):
   # Simulates n games of racquetball between players A and B
   # RETURNS number of wins for A, number of wins for B
```

- With this information, it's easy to get started!

```
def simNGames(n, probA, probB):
    # Simulates n games of racquetball between players A and B
    # RETURNS number of wins for A, number of wins for B
    winsA = 0
    winsB = 0
    for i in range(n):
```

Designing simNGames

- The next thing is simulate a game of racquetball. Not sure how to do that? Put it off until later!
- Let's assume there's a function called `simOneGame` that can do it
- The inputs to `simOneGame` are easy – the probabilities for each player. What is the output?

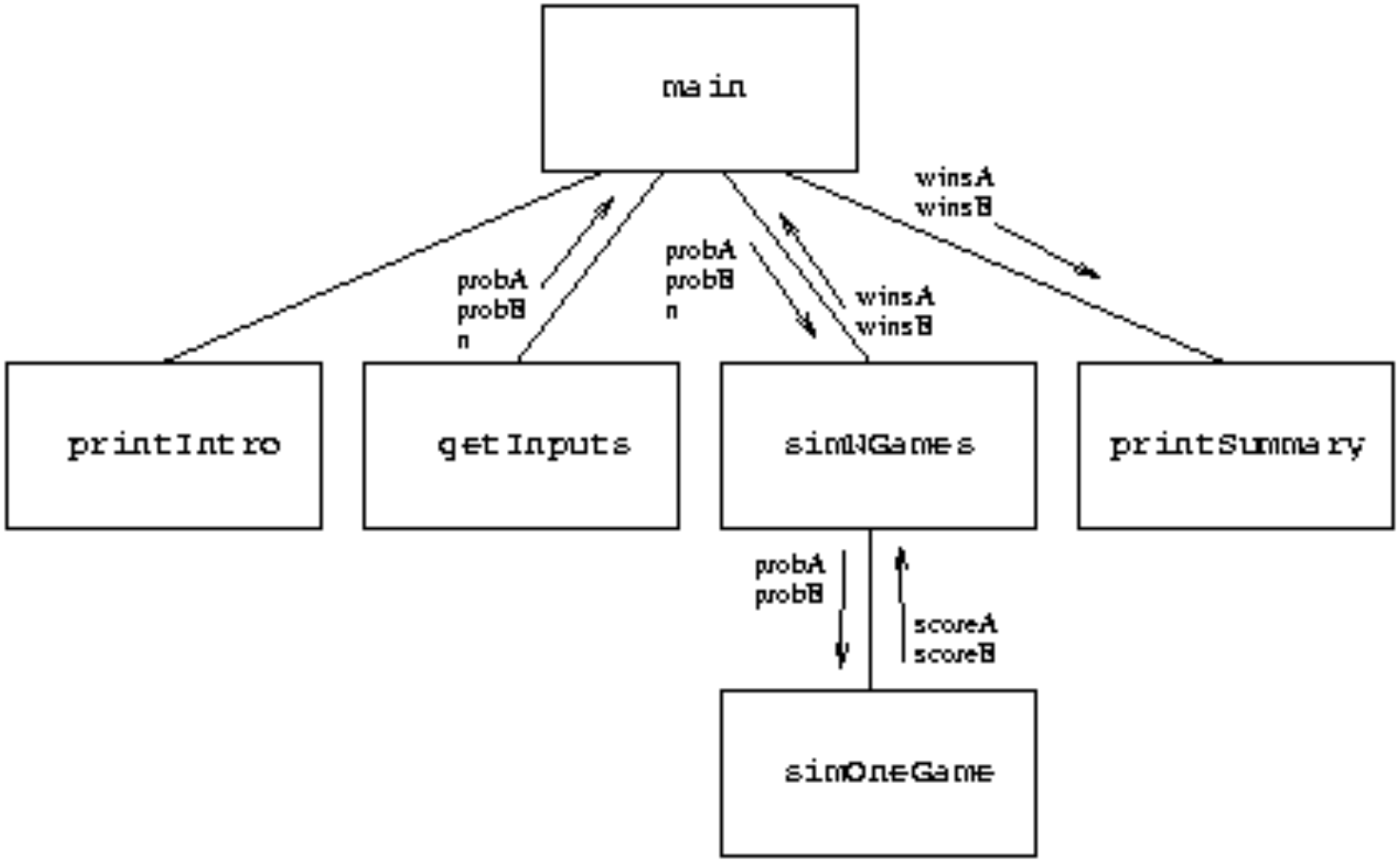
Designing simNGames

- We need to know who won the game. How can we get this information?
- Easiest way is to pass back the final score
- Player with the higher score wins and gets their accumulator incremented by one

Designing simNGames

```
def simNGames(n, probA, probB):
    # Simulates n games of racquetball between players A and B
    # RETURNS number of wins for A, number of wins for B
    winsA = winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB
```

Designing simNGames



Analysis and Specification

- Racquetball is played between **two players** using a racquet to **hit a ball in a four-walled court**
- One **player starts** the game by putting the ball in motion – ***serving***
- Players try to **alternate hitting the ball** to keep it in play, referred to as a ***rally***. The rally **ends** when one player **fails to hit a legal shot**

Analysis and Specification

- Player who **misses the shot loses the rally**. If the loser is the player who served, service passes to the other player
- If the **server wins the rally, a point is awarded**. Players can only score points during their own service
- **First player to reach 15 points** wins the game

Third-Level Design

- The next function we need to write is `simOneGame`, where the logic of the racquetball rules lies
- Players keep doing rallies until game is over, which implies the use of an indefinite loop, since we don't know ahead of time how many rallies there will be before the game is over

Third-Level Design

- We also need to keep track of the score and who's serving. The score will be two accumulators, so how do we keep track of who's serving?
- One approach is to use a string value that alternates between "A" or "B"

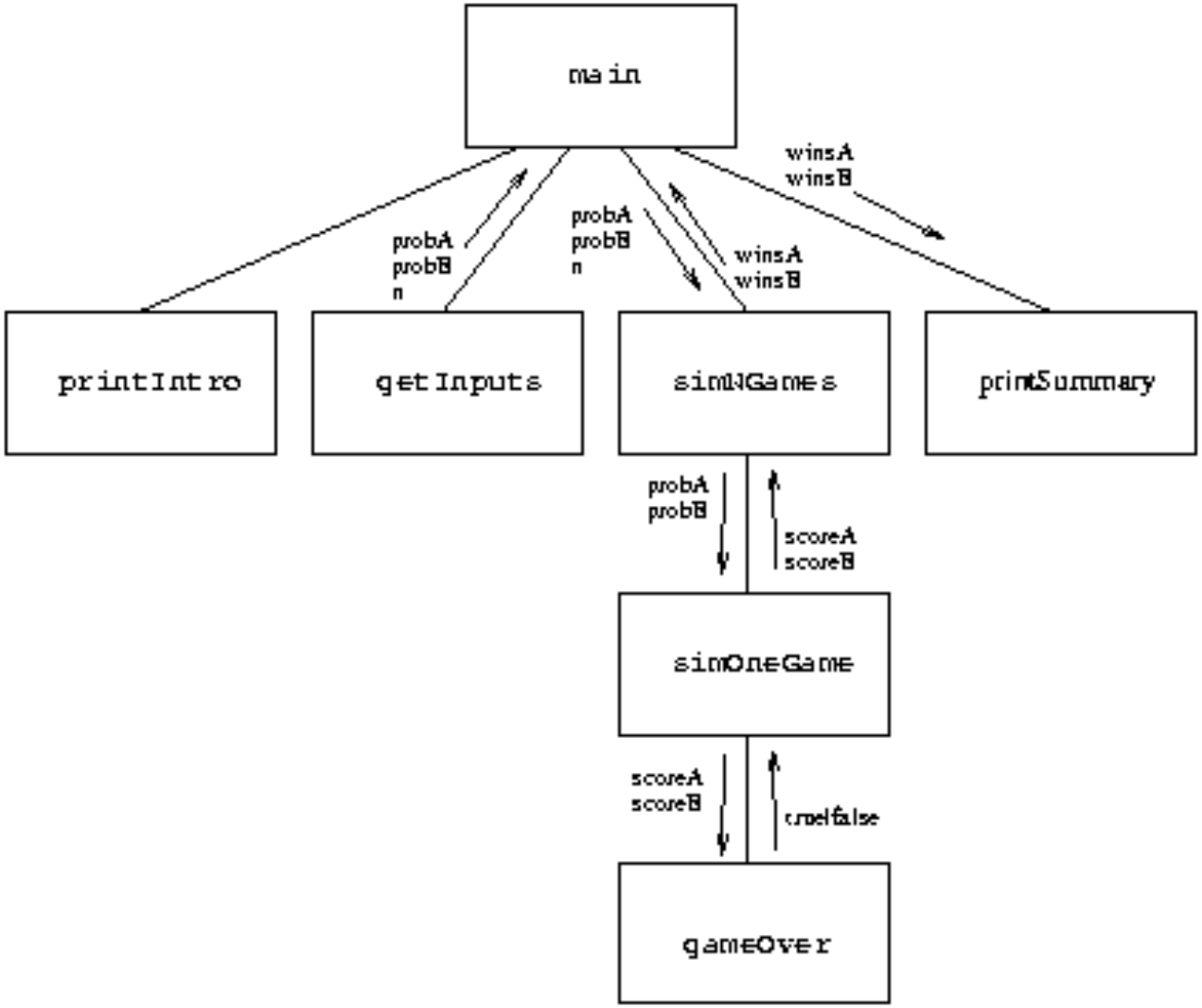
Third-Level Design

- ```
  Initialize scores to 0
  Set serving to "A"
  Loop while game is not over:
     Simulate one serve of whichever player is serving
     update the status of the game
  Return scores
  ```

- ```
  def simOneGame(probA, probB):
     scoreA = 0
     scoreB = 0
     serving = "A"
     while <condition>:
  ```

- What will the condition be?? Let's take the two scores and pass them to another function that returns `True` if the game is over, `False` if not

Third-Level Design



Third-Level Design

- At this point, `simOneGame` looks like this:

```
def simOneGame(probA, probB):
    # Simulates a single game or racquetball between players A and B
    # RETURNS A's final score, B's final score
    serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
```

Third-Level Design

- Inside the loop, we need to do a single serve. We'll compare a random number to the provided probability to determine if the server wins the point (`random() < prob`)
- The probability we use is determined by whom is serving, contained in the variable `serving`

Third-Level Design

- If A is serving, then we use A's probability, and based on the result of the serve, either update A's score or change the service to B

```
if serving == "A":
    if random() < probA:
       scoreA = scoreA + 1
    else:
       serving = "B"
```

Third-Level Design

- Likewise, if it's B's serve, we'll do the same thing with a mirror image of the code

```
if serving == "A":
    if random() < probA:
       scoreA = scoreA + 1
    else:
       serving = "B"
else:
    if random() < probB:
       scoreB = scoreB + 1
    else:
       serving = "A"
```

Putting the function together:

```
def simOneGame(probA, probB):
    # Simulates a single game or racquetball between
    # players A and B RETURNS A's final score, B's final score
    # serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

Finishing Up

- There's just one tricky function left, `gameOver`. Here's what we know:

```
def gameOver(a,b):
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
```

- According to the rules, the game is over when either player reaches 15 points. We can check for this with the boolean:
  `a==15 or b==15`

Finishing Up

- So, the complete code for `gameOver` looks like this:

```
def gameOver(a,b):
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
    return a == 15 or b == 15
```

- `printSummary` is equally simple!

```
def printSummary(winsA, winsB):
    # Prints a summary of wins for each player.
    n = winsA + winsB
    print "\nGames simulated:", n
    print "Wins for A: {0} ({1:0.1%})".format(winsA, winsA)/n)
    print "Wins for B: {0} ({1:0.1%})".format(winsB, winsB/n)
```

- Notice % formatting on the output

Summary of the Design Process

- We started at the highest level of our structure chart and worked our way down
- At each level, we began with a general algorithm and refined it into precise code
- This process is sometimes referred to as *step-wise refinement*

Summary of the Design Process

1. Express the algorithm as a series of smaller problems
2. Develop an interface for each of the small problems
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems
4. Repeat the process for each smaller problem