# Assignment 4: Pig Tournament – Player Building

This assignment focusses on object oriented programming as a design paradigm, in particular programming to an interface.

A classic dice game called Pig works as follows:

Two or more players take turns casting a die. A player turn proceeds as follows:

* Score for a player turn starts at zero
* If the player rolls a 1, their turn is over, and they score zero points
* If the player rolls anything else, the number (2-6) that they roll gets added to their score so far for the turn.
* At that point, the player must make a decision – whether to roll again, or whether to stop.
  + If the player chooses to stop, they get their score for the turn so far.
  + If the player chooses to roll again, they continue
    - For each future roll, a roll of 2-6 gets added to their score for the turn, and they have another choice whether to roll again or not
    - On a roll of 1, however, their turn is over, and they score 0 for the turn (e.g. all points accumulated so far during the turn are lost).

The game continues until the round where at least one player reaches a score of 100 – the player with the highest score at the end of that round wins the game.

The Assignment 4 Code Download has the machinery to run a Pig Tournament built in.

There are a number of classes pre-built:

* PigGame
  + A class that will run a game between a list of players
* PigGamePlayer
  + A class that contains the player, their score in a game, and their score history
* PigGameState
  + A class that contains the state of a Pig Game – basically a set of public properties
* IPigPlayer
  + An interface that defines the characteristics of any Pig Player
    - A Name, a type
    - A decision whether to roll or not
* BasePigPlayer
  + An abstract class that is \*one\* possible implementation of the getName and getType functionality of the interface, that can be a base class for other players
* PPAlwaysStick
  + The incredibly simple player that always takes their first roll.
* PPRandomStick
  + Pass in a random number generator and a % and it will roll with that probability
* PPRollNTimes
  + A Player that will try to roll N times before stopping.

As usual, there will be some guided Core Tasks which must be completed by everyone attempting the assignment, plus a number of Extension tasks, of which two are required to be completed.

## Core Task: Game Count UI (0.5 pts)

Right now, the tournament pairs up players, does \*not\* have players play themselves, but otherwise does a round-robin. So e.g. for three players A, B, and C, A plays B, A plays C, B plays A, B plays C, C plays A and C plays B. Right now roundRobinGames is set to 10, so there are 60 games in a 3-player tournament. Create an Interface for the user to pick this number (default the input to 10 games).

## Core Task: BasePlayer ToString (0.5 pts)

Right now, there are player types and player names, but the abstract class BasePlayer doesn’t have a ‘ToString’ to combine them. Add a ToString class to the base player.

## Core Task: PigTournament Class (1 pts)

There is machinery to run a tournament, but there is no PigTournament class. Instead of building a tournament in the StartTournament button handler, use proper object-oriented style and make this a class.

* Think about what a Tournament needs to know to start (Hint: playerList, roundRobinGames, Random Number Generator) for building the Constructor.
* You’ll want a method to run the tournament.
* You’ll also want a method to build out a tournament report string.

## Core Task: Tournament Reporting (2 pts)

Right now, I can see how many games were won by each player, but not who beat who. Expand the Tournament reporting so that you can report on individual player records. That is, I want to know that player A won 35 matches, with a won-loss record of 19-1 against player B, 8-12 against player C, and 8-12 against player D.

* Right now the tournament just stores the number of wins for each player in a Dictionary object
* You might want a Tournament Results class that you can pass the two players and a getWinner() of each game and have it build up the full records by opponent
* How you display this is totally up to you
* Please submit a screenshot of a couple different tournament reports as part of the submission

## Core Task: PigPlayers (4 pts)

There are a three example game players that have been built out. Your task here is to build out 5 more players. There can be two more very simple players (0.5 pts each) like random stick, and 3 more complicated players (1pt each). At least one of the complicated players should be a derived or composed class using another player, adding some additional functionality.

* Simple Player Ideas
  + We have a player that wants to roll N times – what about a player that wants to score N points each turn?
  + What about a player that stops if they just rolled a 5 or a 6?
  + What about a front-runner, that only stops rolling if they are ahead of all the other players?
  + What about a ‘keep-it-close’ player, that will roll until they are within N points of the leader?
* Specialized strategy Ideas
  + Any of the simple strategies are \*probably\* better if they add logic something like this:
    - If another player is already over 100, keep rolling until I’m ahead.
  + There are a few similar specialized strategies
    - If I’m within N points of 100, keep rolling until I’m over 100, because someone else will likely win this turn otherwise.
    - If I’m more than N points behind, I’d better try to get at least M points to catch up or I’ll eventually lose.
    - Do something different on the first turn to try to get a big lead.
  + These specialized strategies aren’t \*full\* strategies, though, because they only help in certain circumstances. So you still need a simple strategy to attach it to.
* Combining base strategies and specialized strategies
  + As code in a new class
    - Write the specialized logic and the general behaviour in the same class.
  + Inheritance
    - Write the specialized logic in a derived class, and if it doesn’t give you the answer, just use the base class
  + Composition
    - Think of IPigPlayer as a strategy more than a complete player
    - Your specialized strategy could be a class that takes another IPigPlayer as an argument in the constructor as the ‘base’ player
    - If your condition holds, use the specialized strategy
    - If it doesn’t use the base strategy – call the toRollOrNotToRoll of the ‘base’ player.
* Have fun with this!

## Extensions (do at least 2 pts)

* (1 pt) Right now, participation in the tournament is determined by which players are added to the playerList variable, and there is only one copy of each player playing Add a user interface for the user to pick how many of each player are in the tournament (e.g. so some available players might be participating twice, and others zero times).
* (1pt) There is no reason that a PigGame only has to involve 2 players. You could have 3, 4, or more playing at once. The PigGame class will handle this already. Build an alternate tournament setting that will have all the players play each other in every game instead of a bunch of 2-player games
* (1pt) Expand on the above by setting 2, 3, 4, 5 player and all-player variants for a tournament as a tournament setting.
* (1 pt) No coding required. What is the best player you’ve come up with? Do they consistently win over other players? Why is the winning player strategy better than others?
* (1 pt) Save a tournament result through serialization
* (2pts) Take the ‘specialized’ strategy player idea to the next level – have a meta-player than can take a \*list\* of specialized strategies, each of which have a ‘doesMyConditionApply’ function that returns a true or false about whether their special condition should be used. Also have a base strategy. On such a player’s turn, the logic would be:
  + Foreach specialized strategy in the list
    - If (doesMyConditionApply)
    - Call the toRollOrNotToRoll for that specialized strategy
  + If none of the conditions apply, then use the base strategy

Hint: You’ll want a ‘Specialized Strategy’ Interface for the ‘doesMyConditionApply’. Then your meta-player can take any number of ‘Specialized Strategy’ players and \*one\* base player.