Identifying and Designing Classes

- identify classes through textual analysis
  - what are the things (nouns) that feature in a description of the program's task?
    - e.g. tile, bag, rack, board, word, …
  - distinguish kinds of things (classes) from specific instances (objects)
    - e.g. "the player" and "their opponent" → class Player
  - recognize synonyms
  - eliminate things already well-served by existing types
    - e.g. score → int
    - e.g. word → String

Identifying and Designing Classes

- instance variables
  - what needs to be represented about this thing?
  - in textual analysis, look for possessives
    - e.g. player's score, player's played tiles
- methods
  - in textual analysis, look for verbs
    - e.g. "exchange one or more tiles for an equal number from the bag", "play at least one tile on the board", "draw tiles [from the bag]"
  - for actions that involve more than one kind of thing, break it down into how it affects each thing
    - e.g. "exchange tiles" requires removing specific tiles from the rack, adding tiles to the bag, removing (random) tiles from the bag, and adding tiles to the rack
  - put methods in the classes whose representation they manipulate
    - e.g. play a word on the board → GameBoard

- Scrabble
  - BoardSquare – tile contained, scoring info
    - create a particular kind (triple word score, etc)
    - place tile
  - GameBoard – tiles on the board and their arrangement
    - create empty board
    - play word (return score)
  - TileRack – contents
    - create empty rack
    - add tile
    - remove particular tile
  - TileBag – contents
    - create containing all tiles
    - draw tile (remove random tile)
    - exchange tile
  - Tile – letter, point value
    - create particular tile (with letter, point value)
  - Dictionary – the words that are legal to play
    - create containing all legal words
    - look up word (determine if a particular word is legal)
  - Player – score, tiles
    - create with no tiles, score 0
    - add to score

Completing the Design

- strive for a complete list of needed operations
  - review specifications, descriptions, etc to make sure no operations were missed (textual analysis)
  - write pseudocode to help identify program needs not present (or obvious) in the real-world version
    - e.g. printing the contents of the tile rack
  - think through the flow of information
    - e.g. the board manages the board squares, so we will need to ask the board to place a word rather than interacting directly with the squares
- complete the abstractions
  - e.g. getters to access stored information
    - but not necessarily every getter possible
  - e.g. ways to add to, remove from, and iterate through collections
  - some of these things may not be required by this particular program, but are helpful for future reusability

- Scrabble
  - BoardSquare – tile contained, scoring info
    - create a particular kind (triple word score, etc)
    - place tile
  - GameBoard – tiles on the board and their arrangement
    - create empty board
    - play word (return score)
    - **display board**
  - TileRack – contents
    - create empty rack
    - add tile
    - remove particular tile
    - **print tiles**
  - TileBag – contents
    - create containing all tiles
    - draw tile (remove random tile)
    - exchange tile
  - Tile – letter, point value
    - create particular tile (with letter, point value)
    - **get letter, get point value**
  - Dictionary – the words that are legal to play
    - create containing all legal words
    - look up word (determine if a particular word is legal)
  - Player – score, tiles
    - create with no tiles, score 0
    - manipulate set of tiles (add tile, remove particular tile)
    - add to score
    - **get score**

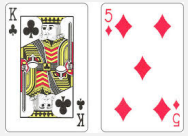
How do you know all the classes needed for a program and how do you know everything that is needed in each class? How do you tell if something is reusable or not?

- identifying classes and their elements
  - try to be thorough in each step
    - start from a complete description
    - think about what else would typically go with each concept
  - remember that it doesn't have to be perfect or complete on the first pass
    - most important from a practical standpoint is to identify the major things
      - especially classes, as those dictate the organization
    - missing elements may be identified as you progress to greater detail, such as pseudocode or even code
      - adding is less upheaval than changing
  - the goal is to plan enough to avoid having to redo a lot because something important wasn't anticipated

Designing Reusable Classes

Several factors influence reusability –

- how specific a thing is
  - e.g. Uno card vs standard playing card



Designing Reusable Classes

Several factors influence reusability –

- how you design the class
  - flexibility
    - e.g. `DiceCollection` with 5 dice vs *n* dice, or not restricting to 6-sided dice
  - include sufficient methods to support the abstraction
    - Yahtzee needs roll the dice, print the dice values, set aside dice
    - other applications might need to undo setting aside, or sum the values of the dice, or add and/or remove dice from the collection
  - avoid overly specialized methods
    - print prints with certain formatting
    - roll, print, sum all involve going through the collection – provide support for iteration rather than specific tasks

Choose *one* of the character recipes, listed below, and give every player that set of coins. Different characters make the whole game different, but *every player must start with the same set of coins.*

There is a "pot" in the middle of the table. Players will take turns playing one coin into the pot. After you play a coin, you may withdraw from the pot any set of coins that adds up to less than the value of the coin you put in. For example, if you put in a dime, you can take back up to 9 cents. *(You should always take back as much as you can.)*

The goal is to run your opponent out of coins. Your score is the number of cents that you have remaining when your opponent plays his last coin. Keep score over multiple games, alternating who goes first.

To play with more than two players, use the same basic rules, with the turn passing to the left. In this case, when one player is knocked out, the game is over, and the player with the most points wins.

- identify classes
- identify what needs to be represented → instance variables
- identify verbs/actions → methods

- Pot – contents
  - play coin into the pot
  - withdraw coins
- Player – set of coins, score
- ~~Coin – value~~ seems like just an integer value
- SetOfCoins – contents
- CharacterRecipe – particular collection of coins

Choose *one* of the character recipes, listed below, and give every player that set of coins. Different characters make the whole game different, but *every player must start with the same set of coins.*

There is a "pot" in the middle of the table. Players will take turns playing one coin into the pot. After you play a coin, you may withdraw from the pot any set of coins that adds up to less than the value of the coin you put in. For example, if you put in a dime, you can take back up to 9 cents. *(You should always take back as much as you can.)*

The goal is to run your opponent out of coins. Your score is the number of cents that you have remaining when your opponent plays his last coin. Keep score over multiple games, alternating who goes first.

To play with more than two players, use the same basic rules, with the turn passing to the left. In this case, when one player is knocked out, the game is over, and the player with the most points wins.

(in progress)