Legacy Code Refactoring and Design Patterns

Legacy Code Refactoring and Design Patterns

By: Robert Bossert, Cameron Connelly, Jayden Dalglish, David Ortega, and Jay Patel

Table of Contents:

Speculative Generality:

Temporary Field:

Validation:

Refactoring Tools:

IntelliJ Built-in Refactor Tool:

SonarLint C++ Refactoring Tool:

Java:

Sprint 1:

CodeSnippetThree.java – Logged Changes

CodeSnippetFive.java – Logged Changes

CodeSnippetNine.java – Logged Changes + Test Building:

Song Refactoring Katas - Logged Changes

Sprint 2:

Banking Application – Logged Changes

Sprint 3:

Yatzy – Logged Changes

Sprint 4:

DrinkMachine - Logged Changes

Python:

Sprint 1:

Tennis1.py - Logged Changes

Tennis6.py – Logged Changes

Sprint 2:

TicTacToe.py – Logged Changes + Test Building:

Tennis4.py – Logged Changes

Tennis3.py – Logged Changes

C++:

Sprint 3:

Liar's Dice Game - Logged Changes + Test Building:

Sprint 4:

Sources:

Introduction:

This document possesses all research and work done by the Legacy Code Refactoring and Design Pattern team at Rowan University. Throughout this document are various code samples, including ‘Legacy Code’ and what the team turns the legacy code into by increasing the effectiveness and readability of the code. Each code smell example will be illustrated with the original code, the definition of the problem, the process of fixing it, and why it’s better our way.

Why is this project being done?

This project is being run to demonstrate the importance of refactoring code. Throughout the code segments and examples, it must be noted and highlighted that zero functionality of the code will be changed. The code we have taken and refactored functionally provides the same results. However, what will be changed is the code's innards, how it looks, how effective it is, and how it’s easier to add more to it.

We believe as a team that having the ability to refactor is one of the best practices any computer programmer can have. While that opinion isn’t universally accepted, it's what our team and our sponsor have decided; this project will illustrate that opinion and demonstrate how highly important it truly is.

What is a ‘successful’ refactor?

Obviously, coding, like all forms of writing and communication, has styles to it. It’s universally impossible to say that one specific style is ‘best’, the same goes for

programming. It's possible to consider coding as an 'art form' in a way. Due to this subjective way of thinking, as a team, we decided it was important to discuss what our preferred style is and what better will be in our eyes. As a reader of this, you may hold various opinions and possibly contradicting opinions to ours.

- Readability: Increasing the readability of our refactored code is one of our highest priorities. It should only take a programmer a brief amount of time to look at any block of code and understand what is going on without having to read several paragraphs of comments. We designed our code to be self-telling; the code itself should be able to tell a reader what it's doing.
    - Comments: Comments are good and useful tools, but only in brevity. Throughout our code, you will only see comments that are concise and to the point.
    - Structure: The code will be broken up into smaller parts and possess the same indentations/style throughout all lines of the segmented code.
- Effectiveness: Often the code is not computing most efficiently. In our time as students, this usually doesn't matter. But in the real world, where runtime and speed of code can make drastic changes to programs, it becomes a top priority. During these refactored projects we will speak on the newly improved efficiency of the code. While it functionally provides the same outputs, decreasing runtimes is a huge priority for us.
- Testing: A large aspect of the refactoring process is automated testing. Having code that can check itself in turns to make sure functionality is intact and runtimes are consistent can be highly beneficial in detecting coding bugs. We will

be using automated testing provided in our legacy code or building our own in order to refactor.

Definitions:

The following definitions are labeled to make sure readers and developers of this project are all familiar with the same terminology. These definitions are labeled and created by the developers of this project, as a reader of the document you may find that these definitions do not line up with your own; remember that these are the definitions outlined for the usage in this specific project.

Legacy Code:

Code that's inherited from someplace else that the editor hasn't worked on prior. Inheriting this code can cause it to be refactored in need for it to be utilized more productively in other areas of code. The code is oftentimes functional, simply working from older styles or using older aspects of the systems, and could be turned into a more efficient aspect of code.

These codes may be lacking in updated terminology, style of coding, and testing. The functionality of the code most likely is still intact, it's simply difficult to add further functionality to it because it's deemed as 'ugly' code. By refactoring, it becomes more flexible code and able to incorporate the ability of added functionality to its processes.

Code Smells:

A component in the code that indicates a much larger problem throughout the code. Doesn't necessarily cause an issue in functionality, but makes code hard to read or lowers the efficiency of the program.

They are often divided into their own large-scale classifications and separated into levels of importance to the code. Oftentimes certain code smells exist but may lack a damaging quality to the code, stay wary of such things as they may prove damaging in the future.

Refactoring:

The process of removing code smells in order to keep code well maintained. Refactoring is often a process that works side-by-side with normal coding in adding functionality. This process is used in order to remove code smells and possible future damages to the code that could cause problems further down the line. An editor/creator of code should remember to keep code cleaner than when they first got it.

Test Code:

A section of code that searches for functionality issues in the code. This is useful for when one is refactoring to make sure the functionality of the initial code is working. Tests are usually broken up into several methods that test each aspect of the code, including initialization, inner processes, and ending processes. Test Code is usually built prior to the refactoring process to maintain proper functionality.

Refactoring Tool:

A tool that is used in detecting code smells and design flaws. Automatically can add features to increase readability and will highlight potential code smells discovered in the code. Each one is primarily an extension of the software used to code or already built in prior. It alerts based on performance and possible security issues that may linger.

Code Smells:

Alternative Classes with Different Interfaces:

It's possible that when you need classes to work with one another their interfaces don't properly line up. It's most likely that you need to change the functions declaration to match one another, and then possibly move certain functionality together to keep more organized.

Comments:

Comments can be used to signify a code smell. If there is a large block of comments describing what the code does it often means the code isn't self-explanatory. Comments can be very useful but should be concise and brief, never long-winded sections as it would simply be covering up badly structured or labeled code.

Conditional Complexity:

Oftentimes with switch statements, if statements, ternary operators, loops, etc. code can experience lengthy issues of deeply nested code. The more straightforward

your code is the better, reduce this code and untangle the nest that it creates by removing unnecessary levels of complexity.

Data Class:

A Data Class is simply a class that has fields, getting and setting methods, and then nothing else to it. These classes' only purpose is to hold data and is only manipulated by other classes. Look for where these getting and setting methods are used by other classes and merge them into these Data Classes to provide a purpose. Some of these fields may hold data that is public, and if that occurs, encapsulate it to make sure it’s off public. Data Classes are almost always a surefire sign that behavior is located in the wrong place.

Data Clumps:

Data clumps are bunches of data that tend to be around each other frequently. Lots of the time it can be seen as fields in several different classes, or possibly parameters. Move them together into their own class to keep track of them, calling the data instead of continuously repeating it in several different functions or classes.

Divergent Change:

Software is structured ‘softly’ making it malleable to provide for different needs as seen fit. Often when a change is required in the system we should jump to a specific spot and change the code we need to. If that isn’t possible it becomes difficult to determine what that single spot is because many of the functions overlap.

Divergent change can also occur when a singular function or module is changed in various ways for various reasons. It creates confusion in your code by creating overlapping tendencies. It's best to split these up and create a divide in your code.

Duplicated Code:

Code that provides the same thing in two separate locations in your code. This can cause problems if there is a slight difference in the code, it's better to extract the function and place the
code under one umbrella that can be called rather than having several different locations filled with the same code.

Feature Envy:

Feature Envy happens when one class or function spends more of its time interacting with another part of the code outside its section rather than what's inside its own section. When this occurs it's best to move that code together, extracting the parts that are communicating heavily outside and merging it with the code it spends its time with.

Insider Trading:

Occurs when two classes or modules keep sending information/data to one another. It can be best to create a third module with all the data that they take from, or perhaps merge their functions to make them more separated in terms of data sending and receiving. Inheritance could also be a beneficial solution.

Lack of Validation:

While writing your code it can often be found that certain things in your program can occur when a user inputs something that possibly may break or damage code. Adding validations around user-inputted information can save a lot of headaches in possible code breakage.

Large Class:

Sometimes classes are trying to do too much at once by having too many fields. Most of the time it’s best to extract from these large classes separating them into multiple smaller classes; when doing this make sure you're extracting the functionality that works best together, rather than creating an ‘Insider Trading’ bug by having the classes communicate with each other too frequently.

Lazy Element

When elements are added to improve upon the structure of code it can be extremely helpful. Sometimes however they are not, instead providing clutter or don’t seem like they have a real reason to be individually defined. It’s better to downsize and merge with others.

Long Function:

The longer a function is the more confusing the function can get. Having concise functions that are more broken down increases readability greatly in code. Oftentimes this can cause functions to have a high amount of parameters as well which can make things hard to move around and fix, it's better to keep code as short as possible.

Long Parameter List:

Having long sets of parameter lists can be confusing and harmful to code. Usually, a long parameter list illustrates that the class is trying to perform too much in your code, creating issues later down the line. It's best to encapsulate data and separate classes that may experience these long parameter lists.

Loops:

Loops are considered far too irrelevant to keep up with modern programming and are recommended to be replaced with pipelines instead.

Removal of infinite loops that could break the code. Oftentimes while using loops it's possible for the loop to continuously stay true, causing the code to break by being unable to access the other functionalities of it.

Global Data:

Global Data is data that can be modified from anywhere in the code base and there's no way to know where it was modified from. This can lead to a great amount of bugs with very little trackability of the bugs, which can cause hours of unnecessary searching. It's possible to get away with small amounts of global data but the more your

code has the harder it is to handle. It is best to encapsulate the data, bringing its accessibility to a smaller scope.

Message Chains:

Message chains occur when clients continuously ask for objects creating a long line of getThis methods. When this occurs it's best to encapsulate the data to separate creating a 'middle man' between your code that will get the objects for you.

Middle Man:

Sometimes in code, there is too much delegation occurring. This code smell is the opposite of the Message Chain, where too much delegation to other classes it may be best to remove these delegations and push your classes together.

Mutable Data:

Data that can be changed throughout the code can cause many problems throughout your code base. It's often considered best practice to have immutable data and oftentimes considered better for a new data structure to be made to preserve old data. It may be required to encapsulate the data, or perhaps split into smaller pieces to reduce/remove changeability in your variables. This often grows to be a large problem once your variable scope grows, the more scope your variable possesses the less likely it should be to change.

Mysterious Name:

Code needs to be mundane and clear, having improperly or unnamed variables, classes, modules, functions, etc will only lead to confusion. Having everything in code properly labeled can save hours of confusion.

Performance:

Oftentimes performance can be a huge problem in code. Having things done inefficiently can lower performance times which can lead to problems in the functionality of the code. If it's noticed that runtimes for your program are slower than it's supposed to be, take note of it and see if there is a recurring problem that's causing such high runtimes.

Primitive Obsession:

Programming is based on your basic data types, strings, integers, etc. It may be better to create a type for yourself to see things more properly labeled. This can be useful when working with money, measurements, time, etc.

Refused Bequest:

Refused Bequest occurs when subclasses inherit methods and data from their parent classes but don't require the methods or data from the parent. Usually, this means your class hierarchy is improperly formed, it could be best to create another

sibling class and push all unused aspects into that; that way the parent class is only filled with commonality between its children classes.

All superclasses do not need to be abstract.

Repeated Switches:

Switch from repeated switches in various classes to polymorphism. It cleans up code by removing heavy repetition which can clutter and confuse.

Shotgun Surgery:

Considered the opposite of Divergent Change; when you make one change you have to go through your code and make small little edits throughout your code. It would be best to move all those changes underneath a singular module to keep that data together. Shot Surgery occurs because logic in your code is spread out making it poorly separated.

Speculative Generality:

Happens when part of your code or machinery has stopped being used. Merge classes together, remove unnecessary parameters, or take out dead code. Functionalities and purposes of code can change, make sure you clean it up.

Temporary Field:

Sometimes when coding some fields, variables, or aspects of data are only used very conditionally. It can be confusing if this data is left in larger aspects of code that are frequently used and it's best to separate these into their own classes or sections of code to illustrate their separation of usage.

Validation:

Oftentimes code is required to be checked for things to continue in the code, validation to make sure proper inputs are occurring with user interaction. When the code is missing this, it can get

Refactoring Tools:

IntelliJ Built-in Refactor Tool:

IntelliJ is an Integrated Development Environment (IDE) developed by JetBrains. It contains a built-in refactoring tool that analyzes code and offers ways to refactor it if any smells are detected.

If a bug or code smell is detected, it will be underlined depending on the severity of the smell. Code that is underlined in yellow is considered a warning, cautioning the user that the code may or may not work depending on the issue. Code that is underlined in red is detected as an error and is a serious problem that must be fixed for the code to run. When code is underlined, IntelliJ can suggest ways to resolve the issue based on patterns it recognizes in the code. Some solutions it suggests can tweak the functionality of the code, which is something that we tried to prevent during our project. Another feature included in this refactoring tool is when you rename a function, class, or method, it will also change all the other references to it throughout the code.

Overall, this refactoring tool has been a tremendous help in identifying code smells and giving tips on how to resolve them. It acts as a helping hand and guides you through the refactoring process. Code refactoring is an important part of the software development process, but it is also one of the most annoying. Using IntelliJ's built-in refactoring tool, it reduced the hassle of detecting code smells and refactoring them, which ultimately made the whole process faster.

SonarLint C++ Refactoring Tool:

SonarLint is a free IDE extension developed by Sonar. It contains a code smell detection system along with its own error base that shows potential fixes and labels code smells by proper names.

Using SonarLint provides easy access to possible code issues and improves the stylization of code while writing. The refactoring tool provides levels of warning along with the errors, including possible performance issues, security issues, and conditional complexity. Certain codesmells provide higher degrees of worry which is denoted by the color coding system provided in SonarLint, highlighting where the smell is underneath the code.

Often the warnings provide possible solutions as well. The solutions aren't formed from the specific code you are working with but are given as if it were a textbook showing an example before and after showing the code smell being fixed. Not only does it show examples, but it goes into great detail about what the smell is, why it is a problem, and why it should be fixed. Those three definitions are important aspects because sometimes a certain smell simply isn't worth a programmer's time and energy to fix, knowing the level of severity of the code smell helps determine if it's worth one's time.

Overall, we found SonarLint to be extremely useful and a great tool especially for it being free for the community to use. It provided helpful insight and information to code smells that weren't brought to our attention by other research we did at the beginning of this project. I'd recommend SonarLint to anyone doing projects at home/personally who is looking for an affordable way to develop software while making sure the style and codebase is concise, efficient, and professional.

Java:

Sprint 1:

CodeSnippetThree.java – Logged Changes

Lazy Elements & Speculative Generality

- Removed the variable ‘i’
  - never used/referenced
  - had no impact on the intended result
- Removed the function ‘foobar’ and its call in the initialization of variable ‘i’
  - never used/referenced
  - had no impact on the intended result

Mysterious Name

- Changed name of ‘getFormattedItemNames’ function to ‘formatItemNames’
  - ‘get’ is misleading as its intended purpose is to format the items in the list in the specified way (“Item name: item”)
- Changed the names of ‘output_result’ and ‘item_name’ variables to follow Java variable naming conventions
  - output_result → formattedOutput
  - Item_name → itemName

Performance

- Replaced the String variable ‘output_result’ in the ‘formatItemNames’ method with a StringBuilder variable named ‘formattedOutput’
  - StringBuilder allows for the creation of mutable String objects
- Removed string concatenation inside the for loop
  - Previous code creates multiple immutable string objects
  - New code clearly states what will happen when the for loop runs
    - Adds “Item name: ” as prefix
    - Then adds the item in the list after the prefix
    - Lastly adds a new line at the end of the formatted string
    - Repeats until all items in the List are processed
  - Before: output_result = output_result + "Item name: " + item_name + "\n";
  - After: formattedOutput.append("Item name: ").append(itemName).append("\n");

CodeSnippetFive.java – Logged Changes

Primitive Obsession

- Created a new class 'Product' to encapsulate the qualities/behaviors of products
    - Holds the quantity and price of a particular product
    - Calculates the base price of an item by multiplying its price by the quantity

Mysterious Name

- Changed the names of some variables for better readability and follow Java naming conventions
    - _quantity → quantity
    - _itemPrice → itemPrice

Long Function

- Broke down the 'getPrice()' function into multiple smaller functions to improve readability
    - calculateBasePrice()
        - calculates the base price before a discount is added by multiplying price by quantity
    - calculateDiscountFactor()
        - determines the discount factor based on whether the base price is greater than or less than 1000
    - calculateFinalPrice()
        - Given the instances of products, this method gets the basePrice of product objects using the calculateBasePrice() function found in the Product class and assigns it to the basePrice variable
        - The discount amount is then found by calling the calculateDiscountFactor() function on the basePrice variable that was just created
        - Lastly, the function calculates the final price after the discount is applied by multiplying the base price by the discount amount and returning that value

CodeSnippetNine.java – Logged Changes + Test Building:

- Test Methods
    - Product()
        - Creates an object of Product.
    - makeProducts()
        - Creates multiple Product objects with varying data within the bounds of Integer. An instance testing discountLevel 1, discountLevel2, negative quantity, negative itemPrice, both negative quantity and itemPrice.
    - testResults()
        - Returns a boolean value which verifies the result of the test. The results are verified from hardcoded values which are expected from the test.

Primitive Obsession

- Created a new class 'Product' to encapsulate the qualities/behaviors of products
    - Holds the quantity and price of a particular product
    - Calculates the base price of an item by multiplying its price by the quantity
    - Calculates the discounted price of an object depending on the item's quantity

Mysterious Name

- Changed the names of some methods and variables for better readability and follow Java naming conventions
- Methods:
    - getPrice() → getDiscountedPrice()
- Variables:
    - __quantity → quantity

- _itemPrice → itemPrice

Mutable Data

- Changed variables to include the final keyword, making the variable immutable and not allowing them to be modified from other parts of the project.
- Variables:
  - private int quantity → private final int quantity
  - private int itemPrice → private final int itemPrice

Lazy Element

- Removed the finalPrice variable and instead returned the function call which saved memory space and improved code readability.

Song Refactoring Katas - Logged Changes

Duplicated Code:

- Throughout the variable song continuous usage of information was being repeated.
  - Each repeated information was extracted and placed in an array labeled animals[] keeping consistency throughout the song with each piece.

Lazy Element & Speculative Generality:

- The song overall was a large lump of code provided lack of structure.
  - Adding elements in from array animal[] made the data easier to change, made it more concise, and aided in structure.

Primitive Obsession:

- Separating the animals in the song makes it clearer to know which will be altered in the future, and easier to alter.

- Extracting each animal provided a separation keeping what's important properly labeled.

Testing:

- A test file called 'SongTest.java' was created in order to create a runnable test throughout the alteration of the main file 'Song.java'
  - The test file grabbed the variable *Song* from the song class and printed it out.

Sprint 2:

Banking Application – Logged Changes

Primitive Obsession & **Data Class**

- Created a new class 'BankAccount' to hold information about the account details such as name and balance.
  - This class also deals with actions such as transferring money and displaying a current balance
  - Necessary changes were made in the main method/class to access its functionality
    - Ex. 'updateRecipientBalance' (formerly 'addRecipientMoney') now calls the 'transfer' function to handle the addition of money to the selected account
    - This was done for all account functionality that was previously handled with functions in the main class
- Before the creation of the 'BankAccount' class, all actions and information were stored in the main 'ATMGame' file
  - Bank accounts were stored as arrays within an array, but now are stored as a 'BankAccount' object to store all necessary information and directly handle actions
- Created a new class 'Validation' to do all necessary validation checks for depositing/transferring money
  - Handles all user input validation
  - Necessary changes were made in the main method/class to access its functionality

        - Ex. All validations were done within the main class, but now are used via a call to the Validation class and any function such as ‘isValidNumber’
        - This was done for all validation functionality that was previously handled with functions in the main class

Mysterious Name

- Changed the name of multiple variables and functions
    - All scanner input variables were changed to ‘userInput’ for better clarity of what the variable holds
        - ‘userAnswerCollection’ and ‘userAnswerValidation’ functions were also changed to match the variable name
    - Changed ‘addRecipientMoney’ and ‘removeSenderMoney’ function names to ‘updateRecipientBalance’ and ‘updateSenderBalance’
    - Changed ‘moneyTransferStatus’ to ‘transferStatus’
    - Changed ‘checkValidNumber’ to ‘isValidNumber’
- Removed the use of the ternary operator in the ‘moneyTransferStatus’ method
    - The ternary operator is difficult to read in comparison to a simple if/else statement

Lazy Elements & Speculative Generality

- Unnecessary functions such as ‘bankTransferStatus’ and ‘storeUserBank’ were removed
    - Their functionality is still used directly with the interaction of the ‘Validation’ and ‘BankAccount’ classes respectively

Long Parameter List

- Removed unnecessary parameters within the functions in the main ‘ATMGame’ class
    - functions updated include: ‘updateRecipientBalance’, ‘updateSenderBalance’, ‘userAccountCollection’, and ‘availableAccounts’
    - The creation of the ‘BankAccount’ class allowed for the removal of many unnecessary/redundant parameters
        - Ex. in the original code, ‘transferToRecipient’ uses type conversions for adding money to an account while also accessing the account in the array each time
        - In the refactored function we just call the deposit function from the ‘BankAccount’ class on the user-inputted amount which does not require any type conversions and makes the code much more readable

Temporary Field

- Removed temporary and redundant boolean fields/variables from functions 'userInputValidation' and 'main'
    - In 'userInputValidation', removed the 'status' and 'formStatus' fields
        - A boolean value is now returned on valid input, and an error message is returned on invalid input
    - In 'main', removed the 'systemStatus' field
        - This field was defaulted to true so it wasn't necessary
        - The program runs until the user reaches the break statement

Duplicate Code & Loops

- Fixed an error causing an infinite loop when validating a user input (y/n)
    - When an invalid input was entered by the user when validating if they want to proceed with the transfer an infinite loop would run regardless if valid input was entered after the invalid one
- Fixed an output error when selecting an account to transfer to
    - Originally the output was asking the user to enter a number from 0-2 for a bank account to transfer to when there are 4 accounts
    - Changed the print statement to dynamically change the maximum number to be the length of the BankAccount Array - 1
        - This will account for any changes made in the future if necessary

Performance

- Replaced the String variable 'myText' in the 'availableAccounts' function with a StringBuilder variable called 'accountList'
    - StringBuilder allows for the creation of mutable String objects
- Removed string concatenation inside the for loop
- Lists all available accounts to transfer money to

Sprint 3:

Yatzy – Logged Changes

Duplicate Code

- Removed multiple functions whose purpose was to calculate a specific score when a particular value on a die was rolled

    - Created a single method called ‘sumDiceValue’ to handle the logic for all numbers on a die

Mysterious Name

- Changed the name of the ‘chance’ function to ‘totalScore’
    - The purpose of this function is to add up the total score of all dice
- Changed the name of the ‘yatzy’ function to ‘checkIfYatzy’
    - The purpose of this function is to check if all rolled dice have the same value, and if so it is a ‘yatzy’, and a player is given 50 points
    - Otherwise, continue with other logic
- Changed the name of ‘counts’ array declaration in multiple functions to ‘dieFrequency’
    - This tracks the frequency of the values rolled for all dice
- Changed the name of the ‘score_pair’ function to ‘checkForSinglePair’
    - This tracks if a single pair has been found and rewards the appropriate points based on the pair value
- Changed the name of the ‘two_pair’ method to ‘checkForTwoPair’
    - This tracks if two pairs have been found and rewards the appropriate points based on the two pairs’ values
- Changed the name of ‘pair’ and ‘threeOfKind’ variables to ‘isAPair’ and ‘isThreeOfAKind’
    - Name changes allow for an easier understanding of their purposes
- Changed the names of ‘three_of_a_kind’ and ‘four_of_a_kind’ to ‘checkThreeOfAKind’ and ‘checkFourOfAKind’
    - Follows standard Java naming conventions

Long Function

- Simplified the logic in the ‘totalScore’ function to return the sum of all dice

Other

- Replaced multiple ternary operators with if/else statements for better clarity
- Included missing checks for certain things such as missing small and large straight sequences

Test Building

- ‘testOnes’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 1
    - When a 1 is rolled, the sum is incremented by 1

    - The sum refers to the number of times the value of 1 was rolled in a given turn of 5 dice (frequency)
- ‘testTwos’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 2
    - When a 2 is rolled, the sum is incremented by 1
- ‘testThrees’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 3
    - When a 3 is rolled, the sum is incremented by 1
- ‘testFours’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 4
    - When a 4 is rolled, the sum is incremented by 1
- ‘testFives’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 5
    - When a 5 is rolled, the sum is incremented by 1
- ‘testSixes’
    - Tests the ‘sumDiceValue’ function for the value of rolling a 6
    - When a 6 is rolled, the sum is incremented by 1
- ‘testTotalScore’
    - Tests the ‘checkTotalScore’ function
    - Verifies that the dice values being added match the expected value
    - The expected value is the sum of the values of the dice from the roll which is returned as the score
- ‘testSinglePair’
    - Tests the ‘checkForSinglePair’ function
    - Counts the frequency each particular value appears
    - If the frequency is more than one, a pair is present
    - Returns the value of the first pair found as a score
- ‘testTwoPair’
    - Tests the ‘checkForTwoPair’ function
    - Verifies there are exactly two pairs
    - If the frequency is more than one for two values, a “two pair” is present
    - Returns the value of the sum of the two pairs as a score
- ‘testThreeOfAKind’
    - Tests the ‘checkThreeOfAKind’ function
    - Verifies that 3 of the dice rolled have the same value
    - The value is then multiplied by 3
        - Ex. if three of five dice are rolled with a value of 4, the output score will be 12 (4 * 3)
- ‘testFourOfAKind’
    - Tests the ‘checkFourOfAKind’ function

    - Verifies that 4 of the dice rolled have the same value
    - The value is then multiplied by 4
        - Ex. if four of five dice are rolled with a value of 4, the output score will be 16
          (4 * 4)
- ‘testSmallStraight’
    - Tests the ‘checkSmallStraight’ function
    - Counts the frequency each particular value appears
    - Verifies if the sequence of 4 increasing/decreasing values are present
        - Ex. 1, 2, 3, 4 or 2, 3, 4, 5
    - If it does, a score of 15 is awarded
- ‘testLargeStraight’
    - Tests the ‘checkLargeStraight’ function
    - Counts the frequency each particular value appears
    - Verifies if the sequence of 5 increasing/decreasing values are present
        - Ex. 1, 2, 3, 4, 5 or 2, 3, 4, 5, 6
    - If it does, a score of 20 is awarded
- ‘testFullHouse’
    - Tests the ‘checkFullHouse’ function
    - Verifies that a pair and three of a kind are present
        - Ex. 3, 3, 3, 4, 4
    - If they are present, a score of 18 is awarded
- ‘testForYatzy’
    - Tests the ‘checkForYatzy’ function
    - Verifies that all dice rolled are the same value
        - Ex. 2, 2, 2, 2, 2
    - If this condition is met, a score of 50 is awarded

Sprint 4:

DrinkMachine - Logged Changes

Long Function

- The original code had functions that would each do multiple tasks, making them long. As multiple tasks per function is not a good practice many of the functions were broken down into multiple smaller functions.

- The constructor in the DrinkMachine class originally had the initialization of the ingredients and recipe. As well as the calculations of what the drink would cost to make and sell.
  - ingredientInitalization()
    - This method is initializing the ingredients, what they cost and then adding it to the ingredientList.
  - recipeInitalization()
    - This method is initializing the recipe of a drink, and then adding it to the drinkList.
  - calculateCost()
    - Gets the ingredients cost and calculates the cost for each of the recipes.
  - calculateDrinkCost()
    - This is just taking the cost calculated for a recipe and then setting it to a specific drink.
- The makeDrink method was originally checking to see if a drink was able to be made and doing the calculations for deducting the ingredients from the stock instead this was split into two methods.
  - deductIngredients()
    - This method is just seeing what a drink costs to make and deducting it from the cost.

Large Class:

- Some of the classes were doing things that would be better off in their own classes to help improve readability, maintainability, and scalability.
    - Split DrinkMachine into the following classes:
        - MainDrinkMachine Class
            - Moved the main method from the DrinkMachine Class to its own class.
        - DisplayMenus Class
            - Took all the displaying that was done in the program and had it done in its own class. Then called it wherever it was needed.
        - InputHanding Class
            - Took the ‘startIO’ method and made a new class which moved all of the handling of user input to its own class.
        - InventoryManager Class
            - Added the ‘restockIngredients’ and ‘deductIngredient’ from the DrinkMachine Class to have all of calculations of the ingredients be done in a separate class
        - DrinkCalculator Class
            - Instead of having these calculations mixed in with the DrinkMachine class. The ‘calculateCost’ and ‘calculateDrinkCost’ was moved to another class

which performed the calculations dealing with the cost of the drink.

Other:

- Added the following methods which were needed to make the other methods work.
    - getDrinkListSize()
        - This getter was used for the input handling which required the size of the drinkList list.
    - getDrinkIndex()
        - This method was used to get the index of the drink from the drinkList list which was required in the input handling.
    - updateDrinkCost()
        - This method was used to update the prices which were calculated from the DrinkCalculator class.

Python:

Sprint 1:

Tennis1.py - Logged Changes

Long Function & Data Clumps:

- The code itself was very stuffed in the function score, so everything was broken down into separate methods instead
    - Tie_Score:
        - This handled situations for when the score equaled one another [fifteen-all, thirty-all, love-all]
    - Is_Game_Over:
        - Determines what occurs once one player has a score greater than or equal to four.
    - Decide_Winner:
        - Determines the winner based on the score differences between the two players, keeping in mind the advantages and disadvantages of each while being tied and coming out of it.
    - Regular_Score:
        - Returns the score of the game as it progresses; displaying it within the context of each player
    - Get_Score_Description:
        - Calls other methods based on what is occurring in the code

Lazy Element:

- Merging of players points 1 & 2 into one single variable to keep track of.
    - It allowed us to create a useful element that was continuously called making our data simpler and more versatile.

Tennis6.py – Logged Changes

Lazy Element & Long Function

- Combined the player score variables, self.player1_score and self.player1_score, into a dictionary (self.scores)
    - Allows for better organization/easier tracking of scores
- Broke down the ‘score’ function into smaller functions that are easier to maintain, read, and understand specific functionality better
    - ‘possible_scores’
        - Defines all possible scores a player can have based on their score from 0-3
    - ‘assign_point’
        - Increments a player's score by 1
    - ‘is_tied’
        - Check both players' scores to see if they are the same
            - Returns ‘True’ if the condition is met
    - ‘display_tied_score’
        - Takes in the score of the first player since the score is tied
        - If the score is less than 3 it concatenates the score of the tied game and ‘-All’ to show it is tied
        - Otherwise, the score is greater than or equal to 3 changing the ‘-All’ to ‘Deuce’
    - ‘display_non_tied_score’
        - Gets the scores for both players

        - Formats a string to properly output in terms of tennis scoring
            - ex. Fifteen-Thirty
    - ‘is_game_finished’
        - Gets the highest score among the players and checks if it is greater than or equal to 4
            - Returns ‘True’ if the condition is met
    - ‘display_final_score’
        - The score difference between the players is calculated by subtracting the second player’s score from the first player’s score
            - If the difference is 1, player2 is leading by 1 score
            - If the difference is -1, player 1 is leading by 1 score
            - If the difference in the score is greater than or equal to 2, player1 has won the match
            - Otherwise, if all previous conditions fail, player2 has won the match
    - ‘score’
        - First, the function checks if the score is tied by running the ‘is_tied’ function
            - If the score is tied it will run the ‘display_tied_score’ function and display its output
        - If this condition is not met, the function ‘is_game_finished’ will run to check if the game is over

        - If the game is over the function ‘display_final_score’ will run and display the output
- Finally, if none of the above conditions were met, the game is still in progress, the ‘display_non_tied_score’ function is ran, and its output is displayed

Sprint 2:

TicTacToe.py – Logged Changes + Test Building:

- Test Methods:
    - Def setUp(self):
        - Calls the game for it to begin
    - Def test_initial_turn(self):
        - Makes sure that the game starts with player ‘X’ going first
    - Def test_valid_moves(self):
        - Makes sure two things occur:
            - If it’s player X’s turn make sure player X plays, then make sure the next turn goes to player 0
            - If it’s player 0’s turn make sure player 0 plays, then make sure the next goes to player X
    - Def test_occupied_square(self):
        - Makes sure if a player tries to play an already occupied square that they don’t lose their turn and doesn’t fill out the already occupied square
    - Def test_horizontal_win(self):
        - Tests horizontal winning condition
            - Makes sure the game is finished
            - Make sure the correct player is declared the winner
    - Def test_vertical_win(self):
        - Tests vertical winning condition

                - Makes sure the game is finished
                - Make sure the correct player is declared the winner
        - Def test_diagonal_win(self):
            - Tests diagonal winning condition
                - Makes sure the game is finished
                - Make sure the correct player is declared the winner
        - Def test_draw(self)
            - Tests if the game ends in a draw
                - Tests if the entire board is filled out with no remaining spaces
                    - Tests if the game properly finishes because there are no remaining spaces
                - Tests to see if the winner is blank and verifies
        - Def test_invalid_input(self):
            - Checks if an error is raised when:
                - A number greater than 9 is entered
                - A number less than 0 is entered
                - A set of non-numeric characters is entered
- Logged Changes:
    - **Loops**
        - Removed for loops turning them in comprehension in the winner function
        - Removal of for loop in the finished method

- **Duplicated Code:**
  - In the winner function code was repeated individually for each type of winning (Row, Column, and Diagonal); instead, it became merged underneath one for loop
- **Comments**
  - Original had a complete lack of comments.
- **Long Function:**
  - The original move() function covered multiple things, including turn switching and move validation. I deemed that too long, making it confusing and hard to read about what the actual function was properly doing
    - validate_move(self,action) Took the validation of move, now the move function calls validate_move in order to confirm each move is a legal play
    - toggle_turn(self) Took the turn switching in the move method and placed it in its own method.
- **Mutable Data:**
  - Another reason why the move() function was broken apart was because of how many different variables were changed throughout the method. We broke it down in order to preserve it.

Tennis4.py – Logged Changes

Large Class:

- Broke down the TennisGame4 class into three smaller classes and combined unnecessary classes in order to abstract the code.
    - TennisGame4
        - score()
        - won_point()
        - evaluate_score()
        - get_score_name()
    - Scoring
        - Is_deuce()
        - has_won()
        - has_advantage()
    - TennisResult
        - format()

Duplicated Code:

- Many functions and classes contained repeated code so that it could account for both the server and the receiver. This was confusing and unnecessary.
    - has_advantage(self, player_score, opponent_score)
        - Combines the function of server_has_advantage() and receiver_has_advantage() and uses the two extra

parameters to distinguish between players. Returns the Boolean value of the test.

- has_won(self, player_score, opponent_score)
  - Combines the function of server_has_won() and receiver_has_won() and uses the two extra parameters to distinguish between players. Returns the Boolean value of the test.

Data Class:

- Unnecessary classes that just made the code less readable and more confusing. Was fixed by merging its functionality with other classes.
  - Deuce
  - GameServer
  - GameReceiver
  - AdvantageServer
  - AdvantageReceiver
  - DefaulResult

Tennis3.py – Logged Changes

Mysterious Name:

- Changed the names of some variables for better readability and to follow Python standard naming conventions.
  - p1 → player1_score
  - p2 → player2_score
  - p1N → player1_name
  - p2N → player2_name
  - n → name

- p → POINTS

Long Function:

- Split up the “score” function into 4 smaller functions, as a function should be doing on task per. To make code cleaner, more maintainable, and more reusable.
  - ‘score’
    - Checks the type of score to return by calling the specific method.
      - If the play is considered ‘standard’ it will return the ‘standard_score’ method.
      - Otherwise it will return the ‘advanced score’ method.
  - ‘standard_play’
    - Make sure that the score is considered standard play (Less than 4), and the total score of both players is less than 6.
  - ‘standard_score’
    - Check if scores are equivalent and then display the score of player 1, and “All”.
    - Otherwise it will display the score of each individual player from the ‘POINTS’ array.
  - ‘advanced_score’
    - This method is called when the score for each player is greater than 3, and their total score is greater than 5.
      - Checks to see if score is tied, and if it is returns “Deuce”
    - Defines the leader and the score difference of players.
    - Returns whether the player is the leader, and win conditions.

Other Changes:

- Moved “POINTS” array to class level, and defined as constant.
  - Since this array is going to not be changed, it was changed to be constant naming conventions.
  - Moved to Class level as it is going to be shared data amongst many methods.

C++:

Sprint 3:

Liar's Dice Game - Logged Changes + Test Building:

- Test Methods
    - testBidInitialization()
        - Tests for the Bid Class; checks if bid.getNumber() and bid.getAmount() for number of dice from what is initialized with the Bid bid(x,y) function
    - testBidSetters()
        - Tests for the Bid Class; checks if bid.setNumber and bid.setAmount for number of dice is working properly
    - testPlayerInitialization()
        - Test for player class; checks if p.getName(), p.getHand(), p.rollDice works properly
        - Checks to make sure hand is properly initialized and that all numbers rolled are in between the numbers 1-6 when the hand is initialized
    - testPlayerRolling()
        - Test for player class; check if p.rollDice is working properly after player initialization
    - testPlayerCounting()
        - Test for Player class; ensuring hand values are reflected to player they are assigned too

- testBiddingLogic()
    - Test for main class; ensures bids get updated properly by testing through a valid bid of the game, and an invalid bid for the game. This is done with two separate assert functions making sure that both getNumber() and getAmount() are checked between a previous bid and a current bid.
- testScoringLogic()
    - Test for main class; ensures scoring is updated when a player guesses either correct or incorrect. Initializes the test by setting score and index for current player, setting total bid for current player, and setting total counter for current player.
- Int main() [for test]
    - Runs through all the methods and outputs whether all of them passed or not.

- Main.cpp
    - Void cls()
        - Clears screen when called
    - Void introduction()
        - Start when program loads, calls five cout statements explaining the game
    - Int playerCount()

        - Gets user input on how many people are playing, making sure the entry is valid and returning the number
    - void initializePlayer(int numPlayers, vector<string>& names)
        - Gets user input on players names, updating them until player cout runs out. Fills a vector with the names
    - void playerScore(vector<int>& score, int numPlayers)
        - Sets up an array of that holds player scores
    - void playerDice(Player* players, int numPlayers, vector<string>& names);
        - Sets up player's dice rolls
    - void totalDieCount(Player* players, int* tot_counter, int numPlayers)
        - Creates an array of holding the number of total die of each number in ordered to be used for bidding later
    - void showDice(Player* players, int numPlayers)
        - Get user input to show the dice of the player entered. Ask for another key to be inputted, clear screen once inputted. Ask again until everyone has seen their dice– quit
        - If name is invalid alert then clear screen.
    - void bidding(Player* players, int numPlayers, int* tot_counter, vector<int>& score)

        - Initializing bidding, asks for user input to determine if a someone would like to challenge to see if they were lying or not
        - If the callout is invalid, alert the user and get a proper one.
        - If callout was yes, increase liar variable and break out of if statement; calling revealGame() to see whether the callout was correct or not
        - If callout was no, continue bidding keeping valid bid == 0
        - If validBid is == 0 continue until callout is yes; getting input for the next bet until a yes callout is made.
            - Validate if bid was legal
    - void revealGame(Player* players, int* tot_counter, int numPlayers, vector<int>& score, const Bid& currentBid)
        - Check against the total counter from the current bid whether or not it was a correct callout, if the challenger was right they gain a point and the liar loses a point; if the challenger was wrong, they lose a point and the ‘liar’ gains a point.
    - bool continueGame()
        - Prompt if game should continue or not
            - Validate if response is legal
        - If the answer was no, increase the round to 1 in main, ending the game.

        - If answer was yes, keep round set to 0 and refresh at the do in main
    - static void reportFinal(Player* players, vector<int>& score, int numPlayers)
        - Print out final recorded scores after all the games.
    - Int main()
        - Calls all methods
        - Set up player array, total count array, names vector, and score vector
        - Initializes round count, numPlayers,
- Logged Changes:
    - Temporary Fields:
        - Many aspects of the initial code were used simply in the beginning, separating themselves from the bulk of the code which is playing multiple rounds of Liar's dice as it is a quick game. All these things that began the game, but were never touched afterwards (Player initialization, introduction, etc.) were formed into their own aspects and tested in order to make sure they still work.
    - Long Function:
        - Everything was thrown into the int main (besides void cls()) making things extremely confusing and almost impossible to read and edit. The first thing that needed to be done was the

separation of all the data in order to understand what was truly happening

- This was solved by encapsulation, taking functions of the data that held together and forming their own classes with.

- Global Data:
  - With everything being underneath the int main, every aspect of the code had full access to one another, making it difficult to understand where data was being used.
- Mutable Data:
  - A lot of the global data that was used throughout the main was frequently being used all over the place keeping it hard where they were being changed from.
  - Improvements of type safety by turning parameter pointers into constant pointers
  - Improvements of type safety by turning pass-by-reference parameters into constant pass-by-reference parameters.
- Mysterious Name:
  - In the original code there were a few aspects of the code that were difficult to understand what they did because of how they were named. The main example of this was the Bid p; which was supposed to represent the player's current bid.

This variable was changed to currentBid in order to keep understanding the code easier.

        - Proper rule declaration of name (call_Out -> callOut)
    - Lazy Element:
        - Oftentimes in the original code, code was declared and then defined somewhere else later in the code for no reason, when it could be declared and assigned a value at the same time. Merging these together made code much more efficient and concise structurally.
    - Comments:
        - Due to the design of the original code most of the code required comments that were unnecessary now with everything encapsulated and names were more properly aligned.
        - Added nested comments inside clear functions in order to provide reasoning on why brackets were left blank
    - Performance:
        - Removal of ‘endl’ to increase Performance; replaced with /n
        - Turning methods and variables static to increase performance and security
    - Conditional Complexity
        - Removal of If statements intertwined creating three levels deep of if statements in the bidding class

- Lack of Validation:

Sprint 4:

Sudoku – Logged Changes

Primitive Obsession & Mysterious Name

- Gave the constant size of the grid a new name, 'GRID_SIZE', while also making it a type alias of 'Grid'
  - Replaced usages of [N][N] grid with the new 'Grid' type
- Renamed multiple variables
  - 'startRow' → 'rowStart'
  - 'startCol' → 'colStart'

Long Function

- Broke down the 'isSafe' and 'solveSudoku' functions into smaller parts focusing on more specific tasks
- 'inRow'
  - Check if a value or the number looking to be placed in a specific row already exists in the row being evaluated
  - Iterates through each column to check for a matching value in the given row
    - Returns true if the value exists, false if not
- 'inCol'
  - Check if a value in a specific column already exists in the column being evaluated
  - Iterates through each row to check for a matching value in the given column
    - Returns true if the value exists, false if not
- 'inSubgrid'
  - Check if a value in a specific 3x3 subgrid already exists
  - Iterates through the first 3 rows and columns of a specified subgrid based on the variables 'rowStart', 'row, 'colStart', and 'col' (9 total spaces to check)
    - Returns true if the value exists, false if not
- 'isSafe'
  - Verifies that a specific value does not already exist in a row, column, or subgrid it is looking to be placed in

        - Returns true if these conditions are met and places the value on the grid, false if not, and moves to the next value

Test Building

- Created multiple functions to test the functionality of the previously developed functionality in the Sudoku class
- ‘testInRow’
    - Takes in the grid, and two number values for row and value being verified for placement
    - Ex. check that in 0 (row 1), the number 7 exists in the row
    - Ex. check that in 0 (row 1), the number 6 does not exist in the row
- ‘testInCol’
    - Takes in the grid, and two number values for column and value being verified for placement
    - Ex. check that in 0 (row 1), the number 4 exists in the row
    - Ex. check that in 1 (row 2), the number 1 does not exist in the row\
- ‘testInSubgrid’
    - Takes in values for the start of a 3x3 subgrid and a value to test for placement in the gird
    - Ex. check that within the subgrid starting at row and column 0, the value 6 exists
    - Ex. check that within the subgrid starting at row and column 0, the value 2 does not exist
- ‘testIsSafe’
    - Takes in the values for a row, column, and value to test for placement in the grid
    - Ex. check that the value 1 can be placed in the first row and third column in an open space
    - Ex. check that the value 5 can not be placed in the first row and third column in an open space
- ‘testSolveSudoku’
    - Takes in the grid and its starting point to verify if it can be solved
    - Ex. check that the grid starting at row one and column one can be solved

Sources:

“ATMGame.java”: GitHub Repository, created by Charlie Roberts, version 1.0, GitHub, https://github.com/CharlieWoo01/Banking-Application/commit/9b36ca0506233230826d8f210fc587ea84cae6b3#diff-a9ae5517285c578bf7db4ba8203ffa8bc9354b7c3730a062f8837314678bf17e

“CodeSnippetThree.java”: GitHub Repository, created by Sumedha Verma, version 1.0, GitHub, https://github.com/sumedhav/clean-code-snippets/blob/master/Code_Snippet_3/CodeSnippetThree.java

“CodeSnippetFive.java”: GitHub Repository, created by Prachi Tyagi, version 1.0, GitHub, https://github.com/sumedhav/clean-code-snippets/blob/master/Code_Snippet_5/CodeSnippetFive.java

“CodeSnippetNine.java”: GitHub Repository, created by Prachi Tyagi, version 1.0, GitHub, https://github.com/sumedhav/clean-code-snippets/blob/master/Code_Snippet_9/CodeSnippetNine.java

“Algorithm to Solve Sudoku: Sudoku Solver.” *GeeksforGeeks*, 30 July 2024, https://www.geeksforgeeks.org/sudoku-backtracking-7/

“Drink.java”; “DrinkMachine.java”; “Ingredient.java”; “IngredientName.java”; “Recipe.java”; “RecipeFactory.java”: GitHub Repository, created by Ted Young, version 1.0, GitHub, https://github.com/tedyoung/mycmt-drinkmachine/tree/master/src

Fejér, Attila. “Code Smells.” *Baeldung on Computer Science*, 18 Mar. 2024, www.baeldung.com/cs/code-smells.

Fowler, Martin, and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.

“main_Zhou.cpp”; “Player.h”; “Player.cpp”; “Bid.h”; “Bid.cpp” : GitHub Repository, created by Shuyu Zhou, version 1.0, GitHub, https://github.com/szhou12/RefinedLiarsGame

“Song.java”: GitHub Repository, created by Egga Hartung, version 1.0, GitHub, https://github.com/sleepyfox/code-dojo-39/blob/master/java/Song.java

“Tennis1.py”: GitHub Repository, created by Emily Bache, version 1.0, GitHub, https://github.com/emilybache/Tennis-Refactoring-Kata/blob/main/python/tennis1.py

“Tennis3.py”: GitHub Repository, created by Emily Bache, version 1.0, GitHub, https://github.com/emilybache/Tennis-Refactoring-Kata/blob/main/python/tennis3.py

“Tennis4.py”: GitHub Repository, created by Emily Bache, version 1.0, GitHub, https://github.com/emilybache/Tennis-Refactoring-Kata/blob/main/python/tennis4.py

“Tennis6.py”: GitHub Repository, created by Emily Bache, version 1.0, GitHub, https://github.com/emilybache/Tennis-Refactoring-Kata/blob/main/python/tennis6.py

“VideoStoreTest.java”; “Customer.java”; “Movie.java”; “Rental.java”: GitHub Repository, created by Manuel Rivero, version 1.0, Github,

https://github.com/Codesai/code-smells-refactoring-training-java/tree/master/02-refactoring-video-store/src/main/java/video_store

“Yatzy.java”; “YatzyTest.java”: GitHub repository, created by Morgan Courbet, version 1.0, GitHub, https://github.com/AlexandruTanasescu/yatzy-java-refactoring-kata/tree/master/java/src