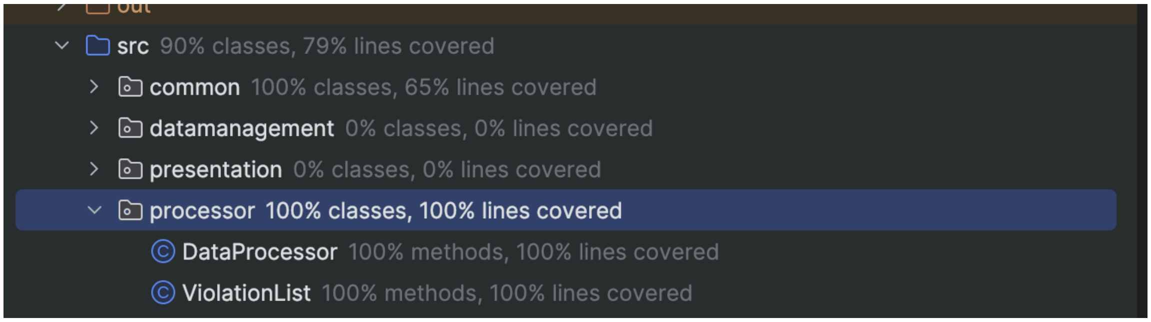
- Screenshots of your IDE that demonstrate that your test cases achieved 100% statement coverage of all methods in classes in the “processor” tier
- 

# Design Patterns, Java Features, and Memoization Implementation

## 1. Design Patterns

### Pattern 1: Singleton Pattern

**Class:** `processor.DataProcessor`

**Description:** The Singleton pattern ensures that only one instance of the `DataProcessor` class exists throughout the application lifecycle. This is appropriate for `DataProcessor` because it serves as a central data processing component that should maintain a single state with all the data (violations, properties, and population) loaded once.

**Implementation Details:**

- **Private static field:** `instance` - (line 10) holds the single instance
- **Private constructor:** `DataProcessor(List<ParkingViolation>, List<Property>, Map<String, Integer>)` - line 20, prevents external instantiation
- **Public static method:** `getInstance(List<ParkingViolation>, List<Property>, Map<String, Integer>)` - line 42, creates the instance on first call, returns existing instance on subsequent calls
- **Public static method:** `getInstance()` lines 51 - returns the singleton instance or null if not initialized
- **Public static method:** `resetInstance()` line 59 - resets the singleton for testing purposes

**Usage:** The singleton is initialized in `presentation.Main.main()` and accessed throughout the application to perform data processing operations.

---

### Pattern 2: Iterator Pattern

**Class:** `processor.ViolationList`

**Description:** The Iterator pattern provides a way to access elements of a collection sequentially without exposing its underlying representation. `ViolationList` wraps a `List<ParkingViolation>` and implements the `Iterator<ParkingViolation>` interface, allowing iteration over violations while maintaining internal state.

**Implementation Details:**

- **Implements:** `java.util.Iterator<ParkingViolation>` (line 7)
- **Private field:** `violations` (line 8) - the underlying list being iterated
- **Private field:** `currentIndex` (line 9) - tracks current position in iteration
- **Method:** `hasNext()` (lines 24-26) - returns true if more elements exist
- **Method:** `next()` (lines 29-34) - returns next element and advances index
- **Method:** `reset()` (lines 19-21) - resets iterator to beginning, allowing multiple iterations
- **Method:** `size()` (lines 36-38) - returns total number of violations

**Usage:** `ViolationList` is used in `DataProcessor.getFinesPerCapita()` (lines 99-106) to iterate over parking violations. The `reset()` method allows the iterator to be reused for multiple iterations over the same data.

---

## 2. Java Features

### Feature 1: Streams and Lambda Expressions

**Class:** `processor.DataProcessor` **Method:** `getAverageMarketValue(String zipCode)` **Lines:** 118 - 123

**Description:** Java Streams API with lambda expressions is used to filter and process the properties list in a functional programming style. This provides a more concise and readable way to perform operations on collections.

**Implementation:**

```
OptionalDouble average = properties.stream()
    .filter(property -> property != null
            && zipCode.equals(property.getZipCode())
            && property.hasValidMarketValue())
    .mapToDouble(Property::getMarketValue)
    .average();
```

**Details:**

- **Stream creation:** `properties.stream()` creates a stream from the properties list
- **Lambda expression in filter:** `property -> property != null && zipCode.equals(property.getZipCode()) && property.hasValidMarketValue()` filters properties matching the ZIP code with valid market values
- **Method reference:** `Property::getMarketValue` converts Property objects to their market value doubles
- **Terminal operation:** `average()` calculates the average of the filtered values, returning an `OptionalDouble`

**Benefits:**

- More declarative and readable code
- Automatic parallelization potential
- Functional programming style reduces side effects

---

### Feature 2: Varargs (Variable Arguments)

**Class:** `processor.DataProcessor` **Method:** `processZipCodes(Function<String, Integer> calculator, String... zipCodes)` **Line:** 195

**Description:** Varargs allows a method to accept a variable number of arguments of the same type. This feature is used in `processZipCodes()` to accept multiple ZIP codes as arguments, making the method more flexible and convenient to use.

**Implementation:**

```
public Map<String, Integer> processZipCodes(Function<String, Integer> calculator,
                                             String... zipCodes) {
    Map<String, Integer> results = new LinkedHashMap<>();
    for (String zipCode : zipCodes) {
        if (zipCode != null) {
            String normalizedZip = zipCode.length() >= 5 ? zipCode.substring(0, 5) : zipCode;
            int result = calculator.apply(normalizedZip);
            results.put(normalizedZip, result);
        }
    }
    return results;
}
```

**Details:**

- **Varargs parameter:** `String... zipCodes` - accepts zero or more String arguments
- **Usage:** The method can be called with any number of ZIP codes:
  - `processZipCodes(calculator, "19103")`
  - `processZipCodes(calculator, "19103", "19104", "19105")`
  - `processZipCodes(calculator)` (empty array)
- **Internal handling:** The varargs parameter is treated as an array, iterated using an enhanced for loop (line 233)

**Benefits:**

- Flexible method signature
- Cleaner API than passing arrays or lists
- Backward compatible (can be called with array argument)

**Note:** This method also uses **Generics** ( `Function<String, Integer>` ) to accept a function that maps ZIP codes to integers, demonstrating another Java feature.

---

## 3. Memoization Implementation

**Class:** `processor.DataProcessor`

**Description:** Memoization is implemented to cache the results of expensive calculations, avoiding redundant computations when the same ZIP code is queried multiple times. This significantly improves performance for repeated queries.

**Cache Fields:**

- `averageMarketValueCache` (line 15) - `Map<String, Integer>` storing average market values by ZIP code
- `averageTotalLivableAreaCache` (line 17) - `Map<String, Integer>` storing average total livable areas by ZIP code

**Initialization:** Both caches are initialized as empty `HashMap` instances in the constructor (lines 42-43).

**Methods Using Memoization:**

### Method 1: `getAverageMarketValue(String zipCode)`

**Implementation Steps:**

1. **Cache Check** line 112: Before calculating, checks if the result for the ZIP code already exists in `averageMarketValueCache`

   ```
   if (averageMarketValueCache.containsKey(zipCode)) {
       return averageMarketValueCache.get(zipCode);
   }
   ```

2. **Calculation**: If not cached, performs the calculation using Streams API to filter and average valid market values

3. **Cache Storage** line 128: Stores the computed result in the cache for future use

```
averageMarketValueCache.put(zipCode, result);
```

### Method 2: `getAverageTotalLivableArea(String zipCode)`

**Implementation Steps:**

1. **Cache Check** (lines 139): Checks `averageTotalLivableAreaCache` for existing result

    ```
    if (averageTotalLivableAreaCache.containsKey(zipCode)) {
        return averageTotalLivableAreaCache.get(zipCode);
    }
    ```

2. **Calculation**: If not cached, iterates through properties to calculate average total livable area
3. **Cache Storage** (line 155): Stores result in `averageTotalLivableAreaCache`

    ```
    averageTotalLivableAreaCache.put(zipCode, result);
    ```

**Cache Management:**

- **Clearing:** Both caches are cleared in `resetInstance()` (lines 79-80) when the singleton is reset for testing
- **Lifetime:** Caches persist for the lifetime of the `DataProcessor` instance, providing persistent caching across multiple method calls

**Benefits:**

- **Performance:** Eliminates redundant calculations for repeated ZIP code queries
- **Efficiency:** O(1) lookup time for cached results vs. O(n) calculation time
- **Memory Trade-off:** Uses additional memory to store cached results, but significantly improves response time for frequently queried ZIP codes

**Example Usage:**

```
DataProcessor processor = DataProcessor.getInstance(violations, properties, population);

// First call - calculates and caches
int avg1 = processor.getAverageMarketValue("19103"); // Performs calculation

// Second call - returns cached result
int avg2 = processor.getAverageMarketValue("19103"); // Returns from cache instantly
```

Catherine’s Contribution

I developed the ParkingViolation class (common/ParkingViolation.java) which represents parking violation records with validation methods. I also implemented ParkingViolationReader (datamanagement/ParkingViolationReader.java) to read parking violations from both CSV and JSON formats, including proper parsing of quoted fields and ZIP code normalization.

Additionally, I enhanced the codebase with modern Java features: I integrated lambda expressions and streams in DataProcessor.getAverageMarketValue() for functional-style data processing, implemented generics and varargs in the processZipCodes() method, and utilized Function interfaces in Main.java for flexible menu handling.

I authored the complete project documentation including the comprehensive README.md with setup instructions, usage examples, and UML class diagram. I also completed the analysis document as required by the project specifications.

Riz’s Contribution
I implemented the core data models and processing logic, including Property class with validation methods for market value and livable area, PropertyReader to parse CSV files with quoted field handling and ZIP code normalization, and PopulationReader to read population data from text files.I implemented the DataProcessor singleton class with memoization caching, including methods for calculating total population, fines per capita, average total livable area, market value per capita, and decimal formatting. I also created the ViolationList iterator wrapper class for violation iteration. I created the test suite with 11 test classes covering all major functionality, including edge cases and boundary conditions. I improved test coverage to 100% statement coverage by adding test cases for accumulation scenarios, invalid inputs, and edge cases.