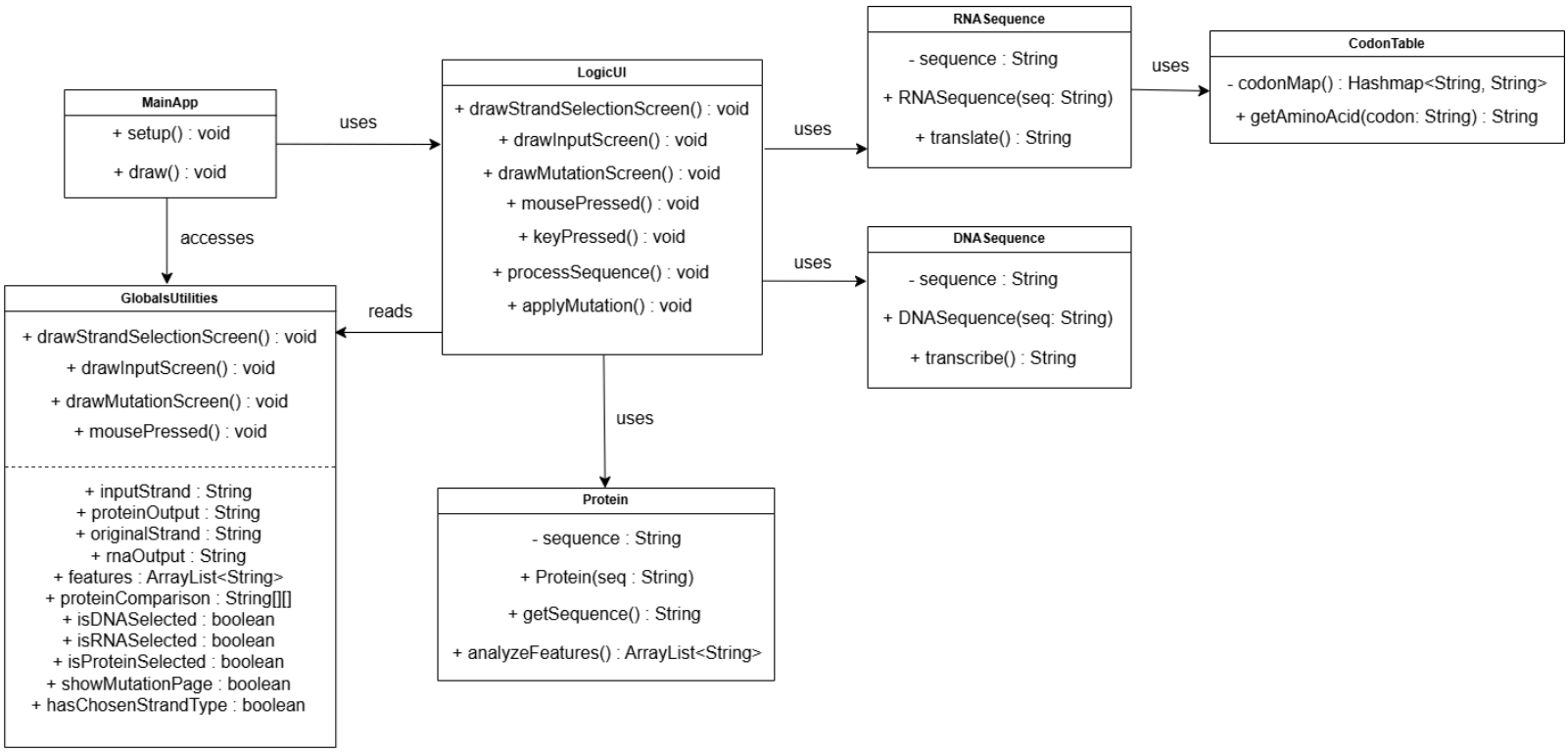
Criterion C: Development

Overview

The development of the Protein Sequence Analyzer required iterative testing, implementation of user feedback, and application of several advanced programming techniques to meet the functional and non-functional requirements, including OOP principles, lists, and 2D arrays. Ultimately, the program enables users to input DNA, RNA, or protein sequences, and simulate biological mutations. This section (Criterion C) outlines the key parts of this program and justifies the use of specific features and structures within the code.

UML Diagram



Complex Code

1. **Blinking Cursor & GUI Input Boxes**

```
// Input box
boolean hoveringBox = isMouseOver(20, 90, 700, 40);
inputBoxSelected = hoveringBox || inputBoxSelected;
fill(hoveringBox || inputBoxSelected ? color(200) : 240);
stroke(0);
rect(20, 90, 700, 40, 6);
fill(50);
textAlign(LEFT, CENTER);
text(inputStrand + (showCursor && inputBoxSelected ? "|" : ""), 25, 110);
```

Construct Used: GUI control + Boolean logic

Explanation: This feature uses the programming construct of conditional logic in combination with GUI rendering to create a blinking cursor effect. It toggles a boolean (showCursor) using a counter and displays the cursor only when the text box is selected. This enhances usability by mimicking text input behavior within a custom graphical interface.

2. **Mutation Application Logic (Point / Deletion / Insertion)**

```
if (mutationType.equals("Point")) {
  sb.setCharAt(pos, mutationValue.charAt(0));
} else if (mutationType.equals("Deletion")) {
  sb.deleteCharAt(pos);
} else if (mutationType.equals("Insertion")) {
  sb.insert(pos, mutationValue);
}
```

Construct Used: Conditional branching + string manipulation algorithm

Explanation: This mutation system employs the if-else branching construct to determine which mutation type to apply. It uses a StringBuilder object to efficiently manipulate the character-level sequence data, implementing mutation algorithms that reflect biological changes such as point mutations, deletions, and insertions.

3. Codon Mapping Using HashMap

```
CodonTable() {
  codonMap = new HashMap<String, String>();

  String[][] codons = {
    {"UUU", "Phe"}, {"UUC", "Phe"}, {"UUA", "Leu"}, {"UUG", "Leu"},
    {"CUU", "Leu"}, {"CUC", "Leu"}, {"CUA", "Leu"}, {"CUG", "Leu"},
    {"AUU", "Ile"}, {"AUC", "Ile"}, {"AUA", "Ile"}, {"AUG", "Met"},
    {"GUU", "Val"}, {"GUC", "Val"}, {"GUA", "Val"}, {"GUG", "Val"},
    {"UCU", "Ser"}, {"UCC", "Ser"}, {"UCA", "Ser"}, {"UCG", "Ser"},
    {"CCU", "Pro"}, {"CCC", "Pro"}, {"CCA", "Pro"}, {"CCG", "Pro"},
    {"ACU", "Thr"}, {"ACC", "Thr"}, {"ACA", "Thr"}, {"ACG", "Thr"},
    {"GCU", "Ala"}, {"GCC", "Ala"}, {"GCA", "Ala"}, {"GCG", "Ala"},
    {"UAU", "Tyr"}, {"UAC", "Tyr"}, {"UAA", "Stop"}, {"UAG", "Stop"},
    {"CAU", "His"}, {"CAC", "His"}, {"CAA", "Gln"}, {"CAG", "Gln"},
    {"AAU", "Asn"}, {"AAC", "Asn"}, {"AAA", "Lys"}, {"AAG", "Lys"},
    {"GAU", "Asp"}, {"GAC", "Asp"}, {"GAA", "Glu"}, {"GAG", "Glu"},
    {"UGU", "Cys"}, {"UGC", "Cys"}, {"UGA", "Stop"}, {"UGG", "Trp"},
    {"CGU", "Arg"}, {"CGC", "Arg"}, {"CGA", "Arg"}, {"CGG", "Arg"},
    {"AGU", "Ser"}, {"AGC", "Ser"}, {"AGA", "Arg"}, {"AGG", "Arg"},
    {"GGU", "Gly"}, {"GGC", "Gly"}, {"GGA", "Gly"}, {"GGG", "Gly"}
  };

  for (String[] pair : codons) {
    codonMap.put(pair[0], pair[1]);
  }
}
```

Data Structure Used: Abstract Data Type | HashMap

Explanation: The codon table is implemented using a HashMap<String, String>, an abstract data type that supports constant-time key-value retrieval. This structure maps RNA codons to their corresponding amino acid abbreviations, allowing for efficient, accurate translation of sequences without repetitive conditionals.

4. Text Wrapping for Output (Protein Features)

```
for (int i = 0; i < features.size(); i++) {
  text("- " + features.get(i), colX, y + row * lineH);
  row++;
  if ((y + (row + 1) * lineH) > height - 50) {
    row = 0;
    colX += colWidth;
  }
}
```

Construct Used: Looping construct + layout algorithm

Explanation: The program uses a for loop to display protein features vertically and wraps them into new columns based on the screen height. This layout algorithm checks the pixel boundaries and resets position dynamically to prevent overflow, enhancing readability and interface responsiveness.

5. **Exception Handling for Mutation Input**

```
try {
  int pos = Integer.parseInt(mutationPositionInput) - 1;
  if (pos < 0) return;
...
  } catch (NumberFormatException e) {
    println("Position input is not a valid number.");
  } catch (Exception e) {
    println("Unexpected mutation error: " + e.getMessage());
  }
}
```

Construct Used: try-catch block for input validation

Explanation: To ensure robust user input handling, a try-catch block is used to validate and parse the mutation position input. This prevents runtime errors from non-numeric or out-of-range values, and demonstrates proper use of exception handling to ensure program stability.

6. **Page Navigation with State Flags**

```
if (!hasChosenStrandType) {
  drawStrandSelectionScreen();
} else if (showMutationPage) {
  drawMutationScreen();
} else {
  drawInputScreen();
}
```

Construct Used: Boolean flags for state-driven control flow

Explanation: The application simulates a multi-screen environment using Boolean state variables such as hasChosenStrandType and showMutationPage. These flags control which screen is drawn at any given time, allowing structured flow between the home, input, and mutation pages without external GUI libraries.

7. 2D Array for Protein Sequence Comparison

```
if (proteinOutput != null && !proteinOutput.trim().equals("")) {
  proteinComparison = new String[2][];
  proteinComparison[0] = before.split(" ");
  proteinComparison[1] = proteinOutput.trim().split(" ");
} else {
```

and

```
// Protein comparison using 2D array
if (
  proteinComparison != null &&
  proteinComparison.length == 2 &&
  proteinComparison[0] != null &&
  proteinComparison[1] != null &&
  !isProteinSelected
) {
```

Data Structure Used: Abstract Data Type – 2D array

Explanation: A String[][] array is used to store and compare the original and mutated protein sequences. This data structure allows parallel traversal of the "before" and "after" states and is used to highlight differences in red. This demonstrates understanding of multidimensional arrays and their role in visual sequence comparison.

References

National Center for Biotechnology Information (NCBI), 2024. *Genetic Codes*. [online] Available at: https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi [Accessed 18 Apr. 2025].

Oracle, 2024. *Java Platform SE 8 API Specification*. [online] Available at: https://docs.oracle.com/javase/8/docs/api/ [Accessed 18 Apr. 2025].

Processing Foundation, 2024. *Processing Reference*. [online] Available at: https://processing.org/reference/ [Accessed 18 Apr. 2025].