Assignment 3 – External Documentation

Waleed R. Alhindi – B0091984

Overview:

The program submitted simulates a simplified version control system. It stores and organizes commits made by developers based on when the commit was made, what files (classes) were involved, and whether the commit was to fix a bug or add a new feature. The storage and organization of commits facilitates the program's reporting features which include, but are not limited to, grouping classes into software components based on a defined minimum threshold, finding feature tasks that involve a defined number of components, and identifying "expert" developers that have made commits on a defined number of components. Furthermore, a time window can be set for the reporting operations such that only the commits added in that time window are referenced when carrying out the reporting operations. The ability to set a time window is important because project leads/managers/developers often want some insight into the activity on their project within, for example, the last month or week since that would give more insight into the current state of the project. However, the program does offer the option of reporting on all commits by invoking report operations without having a time window set. Moreover, this program constitutes a limited version control system and simulates how such systems store, organize, and report commits.

Program Components:

- CommitManager.java – this class is the program's primary class and serves as its overarching "single point of control" containing the program's core functionalities. It provides the methods users will utilize to add commits, set time windows, and report on commit activity.

- Commit.java – this class encapsulates the information of individual commits (i.e., time, developer, files, etc.) into an object. In other words, each commit will be stored in an object of this class.

- CommitDatabase.java – this class acts as a "database" for all the commits that are added and provides CommitManager with methods that include, but are not limited to, retrieving some subset of commits, grouping tasks or developers with their associated files, and calculating the number of times files appeared. This class's data consists of three lists that store commits: a list containing all commits, a list containing only bug commits, and a list containing only feature commits. Each commit added in the CommitManager class is stored in these lists as an object of the Commit class.

- CommitFileGraph.java – this class is primarily used by CommitManager to group files into software components. The class is an implementation of a graph where file names are the vertices and the edges between them are stored in a GraphEdges object (see class below). Edges between vertices denote that the files appeared together in commits and store how many times they appeared together.

- GraphEdges.java – this class stores the edges of a vertex in a String-Integer map where the key denotes the vertex's destination vertex, and the value denotes how many time that edge occurred. In other word, the map keeps track of the destination vertex files that appeared with the source vertex file, as the map's keys, and tallies how many times they appeared together, as the map's values. Thus, each vertex in the graph also references an object of this class to track and maintain its edges.

Assumptions:

The assignment instructions grant the assumption that a commit's developer, task identifier, and files are case sensitive. However, additional implicit assumptions were also made during the development of the program:

1. All empty string values should be treated as invalid inputs since they do not convey any meaningful information that can be used to identify entities. Thus, the addCommit method should throw an IllegalArgumentException if developer, task, or at least one file in commitFiles is an empty string. Additionally, a task is also treated as an empty string if it consists of only a task type with no identifying characters proceeding it (i.e., passing just "B-" or "F-").

2. Expanding upon assumption (1), passing an empty set as the commitFiles in addCommit will also be treated as an invalid argument and will result in addCommit throwing an IllegalArgumentException. This is because adding a commit without any files is a meaningless checkpoint.

3. Expanding on assumption (1), a commit's task identifier must start with "B-" or "F-" where inputs like "b-123" or "F123" will be considered invalid since tasks are case sensitive and must adhere to the format described in the assignment instructions.

4. A commit added with a commitTime of 0 is considered valid as it denotes that the commit was added as soon as the version control system was deployed.

5. In the reporting methods repetitionInBugs, broadFeatures, experts, and busyClasses, passing a number less than 1 as the threshold/limit argument will be considered invalid and cause an IllegalArgumentException to be thrown. This is because a value of less than 1 would not produce any meaningful reports and would undermine the point of using these reporting methods. For example, if the experts method is called with a threshold of 0, then every developer that has committed any file would be deemed an "expert" provides no insight into which developers actually are "experts."

6. Setting a time window where the startTime and endTime are equal is valid and denotes that reporting operations should be done on only that single time unit interval.

7. If softwareComponents, broadFeatures, or experts is called before componentMinimum *successfully* sets a minimum, then the software components should be calculated using a component minimum threshold of 1. This is because setting the default minimum to 1

will group any files that occurred together and provide at least some baseline insight into the software's components, albeit with no granularity. Although not ideal, this approach is better than the two alternatives, which either set the default minimum to 0, where every file becomes part of a single component, or set it to infinity, where every file becomes its own component. Both alternatives provide no practical insight into which files are connected.

Data Structures Utilized and Their Efficiencies:

The program utilizes two overarching data structures to facilitate its operations:

- A graph that stores and maintains the relationship between committed files. Each file is a vertex and edges between file vertices denote the files appeared together in commits. These edges also store the number of times the adjacent vertices were committed together. This graph is maintained by the CommitFileGraph class and uses objects of the GraphEdges class. The graph consists of a <string, object> map where each key denotes a file vertex, and its value is a GraphEdges object that references all the other files that it appeared with along with a counter of how many times they appeared together. Moreover, this graph facilitates the grouping of files (classes) into component sets.

  Efficiency: The graph is implemented using two nested maps, meaning the average time complexity for search, retrieval, and insertion operations is O(1). Additionally, similarly to an adjacency list, this implementation saves space when the graph is sparse but ends up taking more space than something like an adjacency matrix or incidence matrix as the graph becomes denser. So, the implementation will always exhibit good time complexity, but starts exhibiting bad space complexity as the graph grows denser.

- A "database" that stores, maintains, and organizes all the commits added. This "database" is maintained by the CommitDatabase class and uses objects of the Commit class to store individual commits. The "database" consists of three lists of Commit objects: one with all the commits, one for only bug commits, and one for only feature commits. This "database" is used to retrieve organized subsets of commits during reporting operations like, for instance, grouping bug tasks that were committed during the defined time window with all their associated files during repetitionInBugs.

  Efficiency: Since this "database" data structure consists of array lists, the time complexity of its operations averages out to O(n). However, during the program's reporting operations, the program frequently needs to iterate over all the commits in an array list, so the time complexity of such operations is O(n) regardless of the of the data structure used. For example, in the repetitionInBugs method, the program iterates through bug task commits array list to group each bug task with the files it committed.

Key Algorithms and Their Efficiencies:

Grouping Software Components

This algorithm uses the graph that links commit files that appeared together to define the components of the software. The graph also denotes how many times files occurred together, which this algorithm uses to calculate whether files are part of the same component based on the minimum threshold defined by the componentMinimum method. In the case that a minimum threshold has not been set, the algorithm uses a default minimum of 1.

```
Instantiate a set of sets to store the components called allComponents
Iterate (traverse) over each file vertex in the graph
    Check if the file already exists in a component set in allComponents
        Operate on that existing component set
    Else
        Create a new component set and add this file to it
    Get the file vertex's GraphEdges object
    Iterate over each <file, occurrence together tally> entry in the object
        If this entry's value (occurrence tally) meets the "threshold"
            Add the entry's key (file) to this component set
    Add this new/updated component set to allComponents
```

Efficiency: At its core, this algorithm is simply traversing through the graph and checking the "weight" of each vertex's edges to determine whether the adjacent vertices should be added to the same component set. Thus, the efficiency of this algorithm is dictated by the data structure used to implement the graph. In this case, that data structure is a map that stores <file vertex, GraphEdges object> pairs. The algorithm loops through each of the map's entries and access that file vertex's GraphEdges object. The algorithm then loops through that vertex's GraphEdges object, which itself is a map. These nested loops become less efficient as the graph grows denser since each file's GraphEdges object becomes larger and more time consuming to iterate through. However, since we need to cover every vertex and edge in the graph when grouping components, the inefficiency of nested loops is somewhat offset by the comprehensive coverage it provides.

Finding Repetitions in Bugs

This algorithm uses the list of bug task commits maintained by the CommitDatabase class to identify and return bug tasks that have been committed with at least one file a "threshold" amount of times. To facilitate this, the CommitDatabase class first groups each bug task with the files it committed. The algorithm then uses those groupings to calculate whether a bug task is a "repeated bug".

```
Group each bug task with the files it committed
//The CommitDatabase object returns a <bug task, list of committed files> map that
groups each bug task with the files it committed
Iterate over each <bug task, list of files> entry in that map
    Iterate over each file in the bug task's list of files
        Tally how many times that file appears in the list
        If the file's tally meets the "threshold"
            Add this bug task to the set of "repeated bugs"
            Break the loop and go to the next bug task
```

Efficiency: Since we need to identify all "repeated bugs," each bug task needs to be checked. Hence, this algorithm iterates over all bug tasks giving it a baseline time complexity of O(n). However, this algorithm also iterates over each bug task's list of committed files, which adds additional overhead. To reduce the number of iterations needed through a bug task's list of committed files, the algorithm breaks that inner loop as soon as one of its files meets the "threshold" and moves on to the next bug task (next outer loop iteration). Thus, the number of iterations the algorithm performs is the minimal amount needed to identify "repeated bugs." This increases the algorithms overall efficiency, especially as more bug tasks and files are committed.

Identifying Broad Features

Similarly to the algorithm used to find repeated bugs, this algorithm uses an object of the CommitDatabase class to group each feature task with the files it committed. Additionally, this algorithm also indirectly relies on the algorithm used to group software components. This algorithm cross references each feature task's list of committed files against the software component sets to identify whether a feature task was involved in a "threshold" number of components.

```
Retrieve the set of software components
//note this will also set the software components if they have not already been set
Group each feature task with the files it was committed with
//The CommitDatabase object returns a <feature task, list of committed files> map that
groups each feature task with the files it committed
Iterate over each <feature task, list of files> entry in that map
    Get this feature task entry's list of committed files
    Iterate over each software component set
        Iterate over this component's set of files
            If this file also appears in this feature task's list of committed files
                Increment this feature task's "touched components" tally
                Break the loop and go to the next component
        If this feature task's tally meets the minimum threshold
            Add this feature to the "broad features" set
            Break the loop and go to the next feature task
```

Efficiency: To identify "broad features," the algorithm iterates over each feature task and then iterates over each component's set of files to check whether at least one file also exists in the feature task's list of files. These nested loops increase the algorithm's time complexity considerably. To combat this, the algorithm reduces the number of iterations performed in the inner loops by breaking them once *any* file is found to exist in both the component and the feature task. Thus, the algorithm moves on to the next component as soon as the feature task is detected to have "touched" the component. Additionally, the algorithm also stops looping over component sets as soon as the feature task's "touched component" tally meets the "threshold." Although these nested loops are still inefficient, the inefficiency introduced is minimized by reducing the time spent iterating through inner loops to the bare minimum.

Identifying Expert Developers

Similarly to the previous two algorithms, this algorithm relies on the CommitDatabase class to group each developer with the files they committed. It also indirectly relies on the algorithm used to group software components. This algorithm cross references each developer's list of committed files against the software component sets to identify "expert" developers that committed files belonging to a "threshold" number of different components.

```
Retrieve the set of software components
//note this will also set the software components if they have not already been set
Group each developer with all the files they committed
//The CommitDatabase object returns a <developer, list of committed files> map that
groups each developer with the files they committed
Iterate over each <developer, list of files> entry in that map
        Iterate over each software component
                Iterate over each file in that component
                        If the file also exists in this developer's list of committed files
                                Increment this developer's "expert counter" tally
                                Break the loop and go to the next component
                If the developer's "expert counter" tally now meets the "threshold"
                        Add this developer to the set of "experts"
                        Break the loop and go to the next developer
```

Efficiency: Similarly to the algorithm used to identify "broad features," this algorithm iterates over each developer with an additional nested loop over each component's files. To decrease the amount of time spent in the nested inner loops, this algorithm breaks inner loops as soon as one of its files are found to also exist in the developer's list of committed files. Thus, the algorithm reduces the inherent inefficiency of nested loops by reducing the number of iterations through them to the minimum number of iterations needed.

Identifying Busiest Classes (Most Frequently Committed Files)

This algorithm returns a sorted list of the most committed files. The size of that list depends on the "limit" argument passed and whether a tie between files occurs at the limit. If a tie occurs at the limit, then each additional tied file is also added to the list of "busy classes." To facilitate these operations, this algorithm relies on the CommitDatabase class to return a <file, number of occurrences> map that is sorted based on its values (number of occurrences) in descending order. The algorithm uses this sorted map to construct the sorted list of "busy classes."

```
Calculate and sort the number of times each file was committed
//The CommitDatabase object returns a <file, number of occurrences> map of that is
sorted by its values (number of occurrences) in descending order
Iterate over each <file, occurrences> entry in that map
    If the size of the "busy classes" list is less than the limit by at least 2
        Add this file to the "busy classes" list
    Else if the size of the "busy classes" list is less than the limit by 1
        Add this file to the "busy classes" list
        Store this file's number of occurrences
        //We store the file number of occurrences so that we can check
        //whether the files after it (right past the limit) are tied with it
    Else if the size of the "busy classes" list is greater than or equal to the limit
        If this file's number of occurrences is equal to the value stored at the limit
            Add this file to the list of "busy classes"
        Else
            Break the loop and stop adding files to the "busy classes" list
```

Efficiency: Since the CommitDatabase object returns a sorted map, the algorithm simply needs to iterate through that map and add the first "limit" number of files to the "busy classes" list. Additionally, iterations over that sorted map are kept to a minimum by breaking the loop once the "limit" is reached and no additional files past the limit tie with the file at the limit. Thus, this algorithm organizes the most frequent files into the "busy classes" list in a simple and efficient manner.

Test Cases:

Input Validation:

- addCommit:
  - developer = null → throws exception
  - task = null → throws exception
  - commitFiles = null → throws exception
  - developer = empty string → throws exception
  - task = empty string → throws exception
  - commitTime < 0 → throws exception
  - commitFiles = empty set → throws exception
  - commitFiles includes at least one null string → throws exception
  - commitFiles includes at least one empty string → throws exception
  - task string's first character is not 'B' or 'F' → throws exception
  - task string's second character is not '-' → throws exception
  - task = 'B-' → throws exception (treated as empty string)
  - task = 'F-' → throws exception (treated as empty string)

- setTimeWindow:
  - startTime < 0 → does not set time window (returns false)
  - endTime < 0 → does not set time window (returns false)
  - endTime < startTime → does not set time window (returns false)

- componentMinimum:
  - threshold < 1 → does not set component minimum (returns false)

- repetitionInBugs:
  - threshold < 1 → throws exception

- broadFeatures:
  - threshold < 1 → throws exception

- experts:
  - threshold < 1 → throws exception

- busyClasses:
  - limit < 1 → throws exception

Boundary Cases:

- addCommit:
  - developer is one-character string → adds the commit
  - commitTime = 0 → adds the commit
  - task is 'B-' followed by one char → adds the commit
  - task is 'F-' followed by one char → adds the commit
  - commitFiles contains only one file string → adds the commit
  - commitFiles includes a file string that is only one character → adds commit

- setTimeWindow:
  - startTime = endTime → sets time window (returns true)
  - startTime = 0 → sets time window (returns true)
  - endTime = 0 → sets time window (returns true)

- componentMinimum:
  - threshold = 1 → sets component minimum (returns true)

- repetitionInBugs:
  - threshold = 1 → returns set of bug tasks that committed one or more files at least once (i.e., all bug tasks since commitFiles cannot be an empty set during addCommit)

- broadFeature:
  - threshold = 1 → returns set of features that "touched" at least one component (i.e., all feature tasks since each file committed is part of some component)

- experts:
  - threshold = 1 → returns set of experts that "touched" at least one component (i.e., all developers since each file committed is part of some component)

- busyClasses:
  - limit = 1 → returns sorted list of most frequently committed files (returned list may contain more than one file if there is a tie at the limit)

Control Flow

- addCommit:
    - Call when a time window is set:
        - commitTime is outside time window → adds commit
        - commitTime is inside time window → adds commit
    - Add a commit when no time window is set → adds commit
    - commitFiles has multiple files → adds commit

- setTimeWindow:
    - Call when no time window is set → sets time window (returns true)
    - Call when a time window has already been set → overwrites previous time window (returns true)

- clearTimeWindow:
    - Call when no time window is set → nothing happens but does not crash
    - Call when a time window has already been set → removes current time window

- componentMinimum:
    - Call when no minimum is set → sets component minimum (returns true)
    - Call when a minimum has already been set → overwrites previous minimum (returns true)

- softwareComponents:
    - Call when the component minimum threshold has been set → returns set of components according to minimum set
    - Call when a time window has been set → returns set of components using the using only the commits added during that time window
    - Call when no time window is set → returns set of components calculated using *all* commits
    - threshold > any two files occurring together → returns set where every file is its own component

- repetitionInBugs:
    - Call when no time window is set → returns set of "repeated bugs" based on *all* commits
    - Call when a time window has been set → returns set of "repeated bugs" based only on commits added in time window

    - threshold > number of times any bug tasks committed an individual file → returns empty set

- broadFeatures:
    - Call when no time window is set → returns set of "broad features" based on *all* commits
    - Call when a time window has been set → returns set of "broad features based only on the commits added in time window
    - threshold > number of times any feature task "touched" different components → returns empty set

- experts:
    - Call when no time window is set → returns set of "experts" based on *all* commits
    - Call when time window has been set → returns set of "experts" based only on the commits added in time window
    - threshold > number of times any developer "touched" different components → returns empty set

- busyClasses:
    - Call when no time window is set → returns list of "busy classes" based on *all* commits
    - Call when a time window has been set → returns list of "busy classes" based only on commits added in time window
    - Call when there is no tie at the limit → returns list of "busy classes" of size (limit)
    - Call when there is one tie at the limit → returns list of "busy classes" of size (limit + 1) [includes that additional file that tied at limit]
    - Call when there is more than one tie at the limit (X ties at the limit) → returns list of "busy classes" of size (limit + X) [includes all additional files that tied at the limit]

Data Flow

- setTimeWindow:
  - *Covered in control flow*

- clearTimeWindow:
  - *Covered in control flow*

- componentMinimum:
  - Call before adding any commits → sets minimum threshold (returns true)
  - *Some covered in control flow*

- softwareComponents:
  - Call before adding any commits → returns empty set
  - Call before setting a minimum component threshold → returns set of components using a minimum threshold of 1
  - *Some covered in control flow*

- repetitionInBugs:
  - Call before adding any commits → returns empty set
  - *Some covered in control flow*

- broadFeatures:
  - Call before adding any commits → returns empty set
  - Call before setting a minimum component threshold → returns set of "broad features" using a component minimum threshold of 1
  - *Some covered in control flow*

- experts:
  - Call before adding any commits → returns empty set
  - Call before setting a minimum component threshold → returns set of "experts" using a component minimum threshold of 1
  - *Some covered in control flow*

- busyClasses:
  - Call before adding any commits → returns empty list
  - *Some covered in control flow*