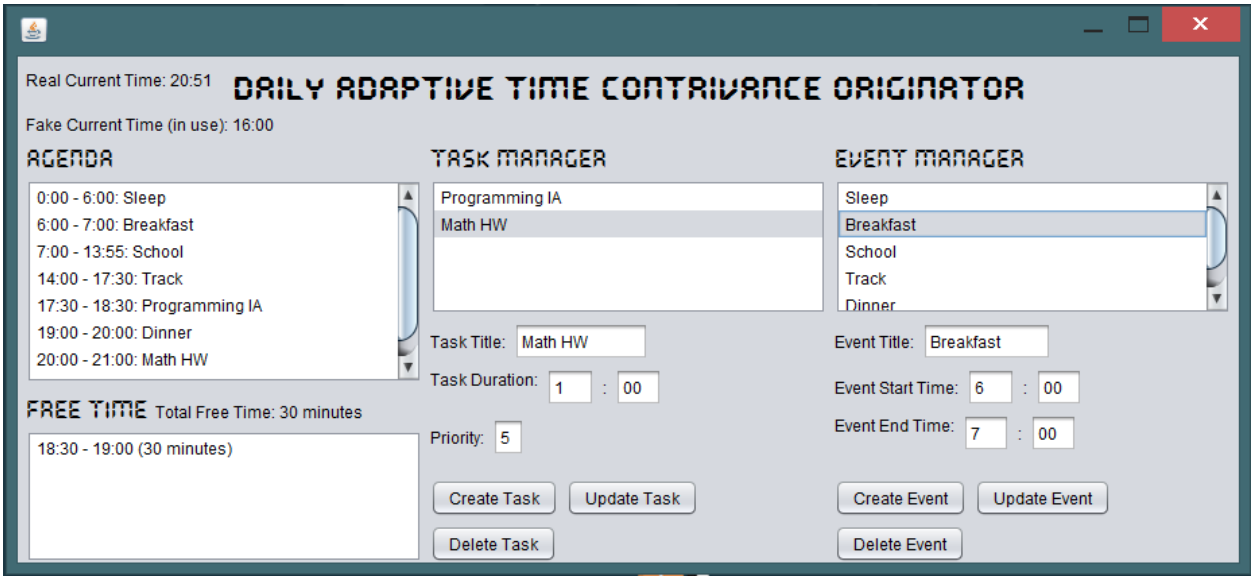
CRITERION C: DEVELOPMENT

OVERVIEW

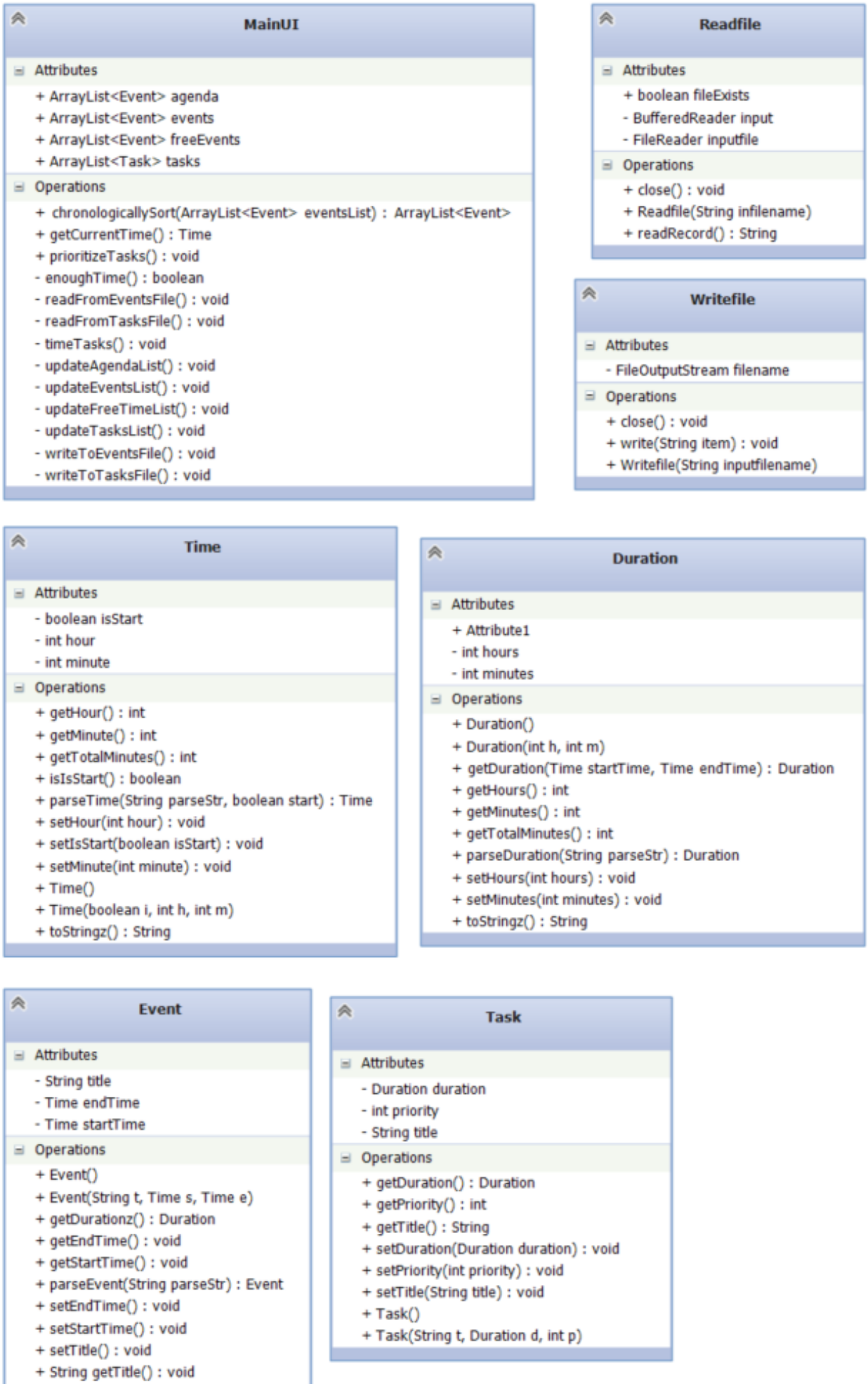
Currently, the only component of the project functioning properly is the desktop text IO agenda program. This simplistic program fulfills my initial goals, minimizing user interface and emphasizing functionality. Moreover, the initial goal of assisting in prioritization delves into the world of artificial intelligence; a simple, more brute-force and greedy algorithm can easily be integrated later, however, it is not crucial to the crux of the project: adaptive scheduling. The time span of the agenda is currently applicable to a single, 24 hour day with individual continuous tasks, eliminating the need for "task categories." In effect, the data structures have been greatly simplified yet the algorithms are still complex but not enigmatic.

USER INTERFACE



DATA STRUCTURES & ORGANIZATION

There are seven classes involved in the program:


MainUI
Attributes
+ ArrayList<Event> agenda
+ ArrayList<Event> events
+ ArrayList<Event> freeEvents
+ ArrayList<Task> tasks
Operations
+ chronologicallySort(ArrayList<Event> eventsList) : ArrayList<Event>
+ getCurrentTime() : Time
+ prioritizeTasks() : void
- enoughTime() : boolean
- readFromEventsFile() : void
- readFromTasksFile() : void
- timeTasks() : void
- updateAgendaList() : void
- updateEventsList() : void
- updateFreeTimeList() : void
- updateTasksList() : void
- writeToEventsFile() : void
- writeToTasksFile() : void
Readfile
Attributes
+ boolean fileExists
- BufferedReader input
- FileReader inputfile
Operations
+ close() : void
+ Readfile(String infilename)
+ readRecord() : String
Writefile
Attributes
- FileOutputStream filename
Operations
+ close() : void
+ write(String item) : void
+ Writefile(String inputfilename)
Time
Attributes
- boolean isStart
- int hour
- int minute
Operations
+ getHour() : int
+ getMinute() : int
+ getTotalMinutes() : int
+ isIsStart() : boolean
+ parseTime(String parseStr, boolean start) : Time
+ setHour(int hour) : void
+ setIsStart(boolean isStart) : void
+ setMinute(int minute) : void
+ Time()
+ Time(boolean i, int h, int m)
+ toStringz() : String
Duration
Attributes
+ Attribute1
- int hours
- int minutes
Operations
+ Duration()
+ Duration(int h, int m)
+ getDuration(Time startTime, Time endTime) : Duration
+ getHours() : int
+ getMinutes() : int
+ getTotalMinutes() : int
+ parseDuration(String parseStr) : Duration
+ setHours(int hours) : void
+ setMinutes(int minutes) : void
+ toStringz() : String
Event
Attributes
- String title
- Time endTime
- Time startTime
Operations
+ Event()
+ Event(String t, Time s, Time e)
+ getDurationz() : Duration
+ getEndTime() : void
+ getStartTime() : void
+ parseEvent(String parseStr) : Event
+ setEndTime() : void
+ setStartTime() : void
+ setTitle() : void
+ String getTitle() : void
+ toStringz() : String
Task
Attributes
- Duration duration
- int priority
- String title
Operations
+ getDuration() : Duration
+ getPriority() : int
+ getTitle() : String
+ setDuration(Duration duration) : void
+ setPriority(int priority) : void
+ setTitle(String title) : void
+ Task()
+ Task(String t, Duration d, int p)

IMPORTANT ALGORITHMS

Chronologically Sort

The "chronologicallySort" method orders the events list according to time, from soonest to latest. This is used all throughout the program.

```
public static ArrayList<Event> chronologicallySort(ArrayList<Event> eventsList)
{
    for (int i = 0; i < eventsList.size(); i++)
        {
            for (int j = 0; j < eventsList.size(); j++)
            {
                if (eventsList.get(i).getStartTime().getTotalMinutes() < eventsList.get(j).getStartTime().getTotalMinutes())
                {
                    Event soonerEvent = eventsList.get(i);
                    Event laterEvent = eventsList.get(j);

                    eventsList.set(i, laterEvent);
                    eventsList.set(j, soonerEvent);
                }
            }
        }
    return eventsList;
}
```

Prioritize Tasks

The "prioritzeTasks" method orders the tasks according to priority, from highest to lowers priority. This method is used quintessentially in scheduling tasks.

```
public static void prioritizeTasks()
{
    for (int i = 0; i < tasks.size(); i++)
    {
        for(int j = 0; j< tasks.size(); j++)
        {
            if (tasks.get(i).getPriority() > tasks.get(j).getPriority())
            {
                Task betterPriority = tasks.get(i);
                Task lowerPriority = tasks.get(j);

                tasks.set(i, lowerPriority);
                tasks.set(j, betterPriority);
            }
        }

    }

}
```

Update Free Time

The “getFreeTime” method involves both updating the freeEvents list and the user interface. This is the first part of the algorithm:

```
private void updateFreeTimeList()
{
    if (!agenda.isEmpty())
    {
        ArrayList<Event> freeTimes = new ArrayList<Event>();

        Time zeroTime = new Time(true, 0,0);
        //sort events chronologically

         agenda = chronologicallySort(agenda);


        if (agenda.get(0).getStartTime() != zeroTime )
        {
            Event firstFreeEvent = new Event("Free Time",zeroTime,agenda.get(0).getStartTime());
            if (firstFreeEvent.getDurationz().getTotalMinutes() != 0)
                freeTimes.add(firstFreeEvent);
        }

        for (int i = 0; i < agenda.size() - 1; i++)
        {

             String freeTitle = "Free Time";
             Time startTime = agenda.get(i).getEndTime();
             Time endTime = agenda.get(i + 1).getStartTime();
             Event newFreeEvent = new Event(freeTitle, startTime, endTime);
             if (newFreeEvent.getDurationz().getTotalMinutes() != 0)
                freeTimes.add(newFreeEvent);

        }
        Time lastTime = new Time(false,24,0);
        if (agenda.get(agenda.size() -1).getEndTime() != lastTime)
        {
            Event lastFreeEvent = new Event("Free Time", agenda.get(agenda.size() -1).getEndTime(), lastTime);
            if (lastFreeEvent.getDurationz().getTotalMinutes() != 0)
                freeTimes.add(lastFreeEvent);
        }

        freeEvents = new ArrayList<Event>();
        for (Event freeTime : freeTimes)
            freeEvents.add(freeTime);
        freeEvents = chronologicallySort(freeEvents);

        for (int index = 0; index < freeEvents.size(); index++)
        {
            if (freeEvents.get(index).getEndTime().getTotalMinutes() <= getCurrentTime().getTotalMinutes())
            {
                //if the freeTime has already passed
                freeEvents.remove(index);
            }
            else if (freeEvents.get(index).getStartTime().getTotalMinutes() < getCurrentTime().getTotalMinutes()
                    && freeEvents.get(index).getEndTime().getTotalMinutes() > getCurrentTime().getTotalMinutes())
            {
                //if in the middle of free Time
                //replace with the free event's starting time as current time
                Time endTime = freeEvents.get(index).getEndTime();
                freeEvents.remove(index);
                Event currentFreeTime = new Event("Free Time", getCurrentTime(), endTime);
                freeEvents.add(0, currentFreeTime);
            }
        }

        freeEvents = chronologicallySort(freeEvents);
```

This part of the method updates the freeEvents list, which contains a list of events that are free time. Since busy events in the agenda have a start times and an end times, this algorithm creates free events from end times to start times. First the program acquires the first space of free time: from the beginning of the day to the first event. Then, the program loops through the agenda to get all the space in between events. Finally, it gets the last space from the end of the last event to the end of the day.

At this point, the method has acquired all the free time for the entire day. However, when timing tasks, one needs to time tasks in the future, not in the past. Therefore, the free time list needs to be truncated to only contain free time in the future to properly schedule tasks. This is done by the rest of the algorithm shown above.

Time Tasks

This is the central method of the program. It operates by first clearing the agenda, adding only the static events, then chronologically sorting it. Then, it loops through each task, placing the task at the soonest slot of free time that fits.

```java
private void timeTasks()
{
    agenda = new ArrayList<Event>();
    for (Event event : events)
    {
        agenda.add(event);
    }
    prioritizeTasks();
    events = chronologicallySort(events);
    updateFreeTimeList();
    freeEvents = chronologicallySort(freeEvents);
    agenda = chronologicallySort(agenda);

   for (Task task : tasks)
   {
        updateFreeTimeList();
        freeEvents = chronologicallySort(freeEvents);

        //get the title
        String title = task.getTitle();

        //get the start time, if there is enough time for the task
        Time startTime = new Time();
        boolean fits = false;
        for (Event freeEvent : freeEvents)
        {
            if (freeEvent.getDurationz().getTotalMinutes() >= task.getDuration().getTotalMinutes())
            {
                startTime = freeEvent.getStartTime();
                fits = true;
                break;
            }
        }

        //adding duration to get the ending time of the task
        if (fits)
        {
            int startTotalTime = (startTime.getHour()*60) + startTime.getMinute();
            int taskTotalTime = task.getDuration().getTotalMinutes();
            int totalTime = startTotalTime + taskTotalTime;
            int newHour = totalTime/60;
            int newMinute = totalTime %60;

            //Get endTime
            Time endTime = new Time(false, newHour, newMinute);
            Event placeTaskEvent = new Event(title, startTime, endTime);
            boolean duplicate = false;
            for (int index = 0; index < agenda.size(); index++)
            {
                if (agenda.get(index).getTitle().equalsIgnoreCase(title))
                {
                    agenda.remove(index);
                    agenda.add(index, placeTaskEvent);
                    duplicate = true;
                }

            }

            if (!duplicate)
            {
                agenda.add(placeTaskEvent);
                updateFreeTimeList();
            }
        }

   }



}
```

Find If Enough Time

This method checks to see if the tasks are compatible with the amount of free time left. There will not be enough free time if the largest space of free time is less than the largest task duration, or if the total free time is less than the total tasks time.

```
private boolean enoughTime()
{
    int longestTaskTotalMinutes = 0;
        for (Task task: tasks)
        {
            if (task.getDuration().getTotalMinutes() > longestTaskTotalMinutes)
                longestTaskTotalMinutes = task.getDuration().getTotalMinutes();
        }

        agenda = new ArrayList<Event>();
        for (Event event : events)
        {
            agenda.add(event);
        }
        prioritizeTasks();
        events = chronologicallySort(events);
        updateFreeTimeList();
        freeEvents = chronologicallySort(freeEvents);
        agenda = chronologicallySort(agenda);

        int longestFreeTimeTotalMinutes = 0;
        for(int i = 0; i < freeEvents.size(); i++)
        {
            if (freeEvents.get(i).getDurationz().getTotalMinutes() > longestFreeTimeTotalMinutes)
                longestFreeTimeTotalMinutes = freeEvents.get(i).getDurationz().getTotalMinutes();
        }
        int totalTaskTime = 0;
        for (Task task: tasks)
        {
            totalTaskTime += task.getDuration().getTotalMinutes();
        }

        int totalFreeTime = 0;
        for (Event freeTime : freeEvents)
        {
            totalFreeTime += freeTime.getDurationz().getTotalMinutes();
        }

        if (longestTaskTotalMinutes > longestFreeTimeTotalMinutes || totalTaskTime > totalFreeTime)
        {
            return false;
        }
        else
        {
            return true;
        }

}
```