CS22510 - Assignment 1
Runners and Riders - ”Out and About”

Samuel Jackson
slj11@aber.ac.uk

March 15, 2013

1 Event Creation Program Documentation

1.1 Code Listing

The following section provides the full code listing for the event creation program. This application is written using C++. Doxygen documentation is available via the provided CD.

Listing 1: eventcreator.h

```
/**
 * @file eventcreator.h
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to create courses, entrants and events.
 */

#ifndef MENU_H
#define MENU_H

#include <vector>
#include "ioscanner.h"
#include "fileio.h"
#include "event.h"

class EventCreator {
public:
    EventCreator();
    virtual ~EventCreator();

    void ShowMainMenu();
private:
    FileIO fio;
    IOScanner scanner;
    std::vector<Event> events;

    void MakeEvent();
    void AddEntrants();
    void CreateCourse();
    int ChooseEvent();
    char ChooseCourse(Event event);
    void ViewEvent();
};

#endif /* MENU_H */
```

Listing 2: eventcreator.cpp

```
/**
 * @file eventcreator.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to create courses, entrants and events.
 * Also outputs and handles user navigation between menus.
 */

#include <iostream>
#include <string>
#include <ctime>
#include <algorithm>

#include "ioscanner.h"
#include "eventcreator.h"
#include "fileio.h"
#include "event.h"

/**
 * Initialises the event creator program and outputs startup message
 */
EventCreator::EventCreator() {
    using namespace std;

    cout << "−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
    cout << "EVENT CREATION PROGRAM" << endl;
    cout << "−−−−−−−−−−−−−−−−−−−−−−−−−" << endl << endl;
}

/**
 * Displays the main menu to the user and processes users choice
 */
void EventCreator::ShowMainMenu() {
    using namespace std;
    int input = 0;

    do {
        cout << "MAIN MENU" << endl;
        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
        cout << "Enter an option: " << endl;
        cout << "1 − Make new event" << endl;
        cout << "2 − Add entrants to event" << endl;
        cout << "3 − Create course for event" << endl;
        cout << "4 − Write an event to file" << endl;
        cout << "5 − View an event in the system" << endl;
        cout << "6 − Exit Program" << endl;

        input = scanner.ReadInt();
        int evt_index;
        switch(input) {
            case 1:
                MakeEvent();
                break;
            case 2:
                AddEntrants();
                break;
            case 3:
                CreateCourse();
                break;
            case 4: //save event to file
                evt_index = ChooseEvent();
                if(evt_index >= 0) {
                    Event e = events[evt_index];
                    fio.WriteEvent(e);
                }
                break;
            case 5:
                ViewEvent();
                break;
        }
    } while (input != 6);
}

/**
 * Member function to create a new event on the system.
 */
void EventCreator::MakeEvent() {
    using namespace std;
    string evt_name;
    tm date, time;

    cout << "Enter name of event:" << endl;
    evt_name = scanner.ReadString(80);

    cout << "Enter event date (DD/MM/YY):" << endl;
    date = scanner.ReadDate();

    cout << "Enter event start time (HH:MM):" << endl;
    time = scanner.ReadTime();

    cout << "Enter location of nodes file for event:" << endl;
    string nodesfile = scanner.ReadString(100);
    vector<int> nodes = fio.ReadNodesList(nodesfile);

    Event e(evt_name, date, time);
    e.SetNodes(nodes);
    events.push_back(e);
}

/**
 * Member function to add a new entrant to an event.
 */
void EventCreator::AddEntrants() {
    using namespace std;
    int eventIndex = ChooseEvent();
    int numEntrants = 0;
    string name;
    int id;
    char course;

    //if user picked an event
    if(eventIndex >= 0) {
        Event event = events[eventIndex];

        //check if we have some courses already.
        if(event.GetCourses().size() > 0) {
            cout << "Enter number of entrants to add: " << endl;

            do {
                numEntrants = scanner.ReadInt();
                if(numEntrants <=0) {
                    cout << "Not a valid number of entrants" << endl;
                } else if (numEntrants > 50) {
                        cout << "Too many entrants to create at once!" << endl;
                }
            } while (numEntrants <= 0);

            for(int i = 0; i < numEntrants; i++) {
                cout << "Enter entrant's name: " << endl;
                name = scanner.ReadString(50);
                course = ChooseCourse(event);
                id = event.GetEntrants().size()+1;
                event.AddEntrant(name, id, course);
                events[eventIndex] = event;
            }
        } else {
            cout << "You must create at least one course first." << endl;
        }
    }
}
```

```
/**
 * Choose an event to work with if there are events on the system.
 * @return the id of the chosen event
 */
int EventCreator::ChooseEvent() {
    using namespace std;
    int index = −1;
    bool validChoice = false;

    if(events.size() > 0 ) {
        cout << "Please choose an event:" << endl;
        for(std::vector<int>::size_type i = 0; i != events.size(); i++) {
            cout << i << " − " << events[i].GetName() << endl;
        }

         do {
            index = scanner.ReadInt();
            if (index >= 0 && index < events.size()) {
                validChoice = true;
            } else {
                cout << "Not a valid event choice." << endl;
            }
        } while(!validChoice);

    } else {
        cout << "You must create at least one event first." << endl;
    }

    return index;
}

/**
 * Choose a course based on the selected event
 * @param event the currently selected event
 * @return the id of the chosen course
 */
char EventCreator::ChooseCourse(Event event) {
    using namespace std;
    bool validChoice = false;
    int index;
    char choice;
    std::vector<Course> courses = event.GetCourses();

    if(courses.size() > 0 ) {
        cout << "Please choose course for the entrant:" << endl;
        for(std::vector<int>::size_type i = 0; i != courses.size(); i++) {
            cout << i << " − " << courses[i].GetId() << endl;
        }

         do {
            index = scanner.ReadInt();
            if (index >= 0 && index < courses.size()) {
                validChoice = true;
            } else {
                cout << "Not a valid course choice." << endl;
            }
        } while(!validChoice);
        choice = courses[index].GetId();
    } else {
        cout << "You must create at least one course first." << endl;
    }

    return choice;
}

/**
 * Create a course based on the selected event
 */
void EventCreator::CreateCourse() {
    using namespace std;
    int eventIndex = ChooseEvent();
    int node;
    vector<int> courseNodes;
    vector<int> allowedNodes;
    if(eventIndex >= 0) {
        Event event = events[eventIndex];
        allowedNodes = event.GetNodes();

        if(event.GetCourses().size() <= 26) {
            cout << "Enter nodes for course. Enter 0 to finish: " << endl;

            do {
                node = scanner.ReadInt();
                if(find(allowedNodes.begin(), allowedNodes.end(), node)!=allowedNodes.end()) {
                    courseNodes.push_back(node);
                } else if (node != 0) {
                    cout << "Not a valid node number!" << endl;
                }
            } while(node != 0);

            //convert numerical index to character index
            // e.g. ASCII 'A' is 65, 'B' is 66 etc.
            char id = (int)event.GetCourses().size()+65;

            event.AddCourse(id, courseNodes);
            events[eventIndex] = event;

        } else {
            cout << "Events can not have more than 26 courses" << endl;
        }
    }
}

/**
 * View an event on the system. This will list all course and
 * entrants associated with the chosen event.
 */
void EventCreator::ViewEvent() {
    using namespace std;
    int eventIndex = ChooseEvent();
    if(eventIndex >= 0) {
        Event event = events[eventIndex];

        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
        cout << event.GetName() << endl;
        cout << event.GetFormattedDate() << endl;
        cout << event.GetFormattedTime() << endl;
        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
        cout << "COURSES" << endl;
        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;

        if(event.GetCourses().size() > 0) {
                        for(std::vector<Course>::iterator it = event.GetCourses().begin();
                                it != event.GetCourses().end(); ++it) {
                            cout << it−>GetId() << " ";
                            cout << it−>GetNodes().size() << " ";

                            std::vector<int> nodes = it−>GetNodes();
                            for(std::vector<int>::iterator jt = nodes.begin();
                                    jt != nodes.end(); ++jt) {
                                cout << *jt << " ";
                            }

                            cout << endl;
                        }
        } else {
                cout << "This event has no courses yet!" << endl;
        }

        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
        cout << "ENRTANTS" << endl;
        cout << "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−" << endl;
        if(event.GetEntrants().size() > 0) {
                    for (vector<Entrant>::iterator it = event.GetEntrants().begin();
                                it != event.GetEntrants().end(); ++it) {
                          cout << it->GetId() << " " << it->GetCourse() << " ";
                          cout << it->GetName() << endl;
                    }
        } else {
              cout << "This event has no entrants yet!" << endl;
        }
    }
}

EventCreator::~EventCreator() {
}

/**
 * Main method and application entry point.
 * Simply shows the main menu.
 *
 * @param argc the number of command line arguments
 * @param argv the char array of command line arguments
 * @return program exit status (0)
 */
int main(int argc, char** argv) {
    EventCreator ec;
    ec.ShowMainMenu();
    return 0;
}
```

Listing 3: event.h

```
/**
 * @file event.h
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about an event.
 */

#ifndef EVENT_H
#define EVENT_H

#include <string>
#include <vector>

#include "entrant.h"
#include "course.h"

class Event {
    public:
        Event(std::string name, tm date, tm time);
        virtual ~Event();

        void AddEntrant(std::string name, int id, char course);
        void AddCourse(char id, std::vector<int> nodes);
        void SetCourses(std::vector<Course> courses);
        std::vector<Course> GetCourses() const;
        void SetEntrants(std::vector<Entrant> entrants);
        std::vector<Entrant> GetEntrants() const;
        void SetName(std::string name);
        std::string GetName() const;
        void SetDate(tm date);
        tm GetDate() const;
        void SetTime(tm time);
        tm GetTime() const;
        void SetNodes(std::vector<int> nodes);
        std::vector<int> GetNodes() const;

        std::string GetFormattedDate();
        std::string GetFormattedTime();
    private:
        tm time;
        tm date;
        std::string name;
        std::vector<Entrant> entrants;
        std::vector<Course> courses;
        std::vector<int> nodes;

        std::string GetDayPostfix(int day);
};

#endif /* EVENT_H */
```

Listing 4: event.cpp

```
/**
 * @file event.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about an event.
 */

#include <string>
#include <sstream>
#include "event.h"
#include "entrant.h"

/**
 * Create a new event and initilise it with a name, date and time.
 * @param name the name of the event
 * @param date the date of the event
 * @param time the time of the event
 */
Event::Event(std::string name, tm date, tm time) {
    this->time = time;
    this->date = date;
    this->name = name;
}

Event::~Event() {
}

/**
 * Add an entrant to this event.
 * @param name the name of the entrant
 * @param id the id of the entrant
 * @param course the if of the entrant's course
 */
void Event::AddEntrant(std::string name, int id, char course) {
    Entrant entrant(id, name, course);
    entrants.push_back(entrant);
}

/**
 * Add a course to this event.
 * @param id the id of the course
 * @param nodes the vector of nodes for the course
 */
void Event::AddCourse(char id, std::vector<int> nodes) {
    Course course(id, nodes);
    courses.push_back(course);
}

/**
 * Set the list of courses for this event
 * @param courses the vector of courses for an event
 */
void Event::SetCourses(std::vector<Course> courses) {
    this->courses = courses;
}

/**
```

```
 * Get the list of courses for this event
 * @return the vector of courses for an event
 */
std::vector<Course> Event::GetCourses() const {
    return courses;
}

/**
 * Set the list of entrants for this event
 * @param entrants the vector of entrants for an event
 */
void Event::SetEntrants(std::vector<Entrant> entrants) {
    this->entrants = entrants;
}

/**
 * Get the list of entrants for this event
 * @return the vector of entrants for an event
 */
std::vector<Entrant> Event::GetEntrants() const {
    return entrants;
}

/**
 * Set the name of this event
 * @param name the name of this event
 */
void Event::SetName(std::string name) {
    this->name = name;
}

/**
 * Get the name of this event
 * @return the name of this event
 */
std::string Event::GetName() const {
    return name;
}

/**
 * Set the date of this event
 * @param date the date of this event
 */
void Event::SetDate(tm date) {
    this->date = date;
}

/**
 * Get the date of this event
 * @return the date of this event
 */
tm Event::GetDate() const {
    return date;
}

/**
 * Set the time of this event
 * @param time the time of this event
 */
void Event::SetTime(tm time) {
    this->time = time;
}

/**
 * Get the time of this event
 * @return the time of this event
 */
tm Event::GetTime() const {
    return time;
}
```

```
/**
 * Set the list of nodes for this event
 * @param nodes the vector of nodes for this event
 */
void Event::SetNodes(std::vector<int> nodes) {
    this->nodes = nodes;
}

/**
 * Get the list of nodes for this event
 * @return the vector of nodes for this event
 */
std::vector<int> Event::GetNodes() const {
    return nodes;
}

/**
 * Get the date of the event as a string in a long format
 * e.g. 1st February 2012
 * @return the date formatted and as a string
 */
std::string Event::GetFormattedDate() {
        using namespace std;
    ostringstream outputDate;
    char monthname[10];
    char year[5];

    strftime(monthname, 10, "%B", &date);
    strftime(year, 5, "%Y", &date);

    outputDate << date.tm_mday;
    outputDate << GetDayPostfix(date.tm_mday) << " ";
    outputDate << monthname;
    outputDate << " ";
    outputDate << year;

    return outputDate.str();
}

/**
 * Get the time of the event as a string
 * e.g. 17:45
 * @return the time as a string
 */
std::string Event::GetFormattedTime() {
        using namespace std;
    ostringstream timeString;
    char outputTime [6];

    strftime(outputTime, 6, "%R", &time);
    timeString << outputTime;

    return timeString.str();
}


/**
 * Member function to get the postfix of the date's day
 * will return a string with either 'st', 'nd' or 'rd'.
 * @param day the day to get the postfix for
 * @return the postfix for the date's day
 */
std::string Event::GetDayPostfix(int day) {
    std::string postfix = "th";
    switch(day) {
        case 1:
        case 21:
        case 31:
            postfix = "st";
            break;
        case 2:
```

```
        case 22:
            postfix = "nd";
            break;
        case 3:
        case 23:
            postfix = "rd";
            break;
    }

    return postfix;
}
```

Listing 5: entrant.h

```
/**
 * @file entrant.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about an entrant in an event.
 */

#ifndef ENTRANT_H
#define ENTRANT_H

#include <string>

class Entrant {
    public:
        Entrant(int id, std::string name, char course);
        virtual ~Entrant();

        void SetCourse(char course);
        char GetCourse() const;
        void SetName(std::string name);
        std::string GetName() const;
        void SetId(int id);
        int GetId() const;
    private:
        int id;
        std::string name;
        char course;
};

#endif /* ENTRANT_H */
```

Listing 6: entrant.cpp

```
/**
 * @file entrant.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about an entrant in an event.
 */

#include "entrant.h"

/**
 * Initilises a new instance of an entrant with an ID,
 * name and course.
 *
 * @param id the ID of the entrant
 * @param name the name of entrant
 * @param course the ID of the course the entrant is registered on.
 */
Entrant::Entrant(int id, std::string name, char course) {
        SetId(id);
        SetName(name);
        SetCourse(course);
}
```

```
Entrant::~Entrant() {
}

/**
 * Set the course the entrant is on.
 * @param course the course id
 */
void Entrant::SetCourse(char course) {
    this->course = course;
}

/**
 * Get the course the entrant is on.
 * @return the course id
 */
char Entrant::GetCourse() const {
    return course;
}

/**
 * Set the name of the entrant.
 * @param name the name of the entrant
 */
void Entrant::SetName(std::string name) {
    this->name = name;
}

/**
 * Get the name of the entrant.
 * @return the name of the entrant
 */
std::string Entrant::GetName() const {
    return name;
}

/**
 * Set the entrant's ID.
 * @param id the entrant id
 */
void Entrant::SetId(int id) {
    this->id = id;
}

/**
 * Get the entrant's ID.
 * @return the id of the entrant
 */
int Entrant::GetId() const {
    return id;
}
```

Listing 7: course.h

```
/**
 * @file course.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about a course in an event.
 */

#ifndef COURSE_H
#define COURSE_H

#include <vector>

class Course {
    public:
        Course(char id, std::vector<int> nodes);
        virtual ~Course();

        void SetNodes(std::vector<int> nodes);
        std::vector<int> GetNodes() const;
        void SetId(char id);
        char GetId() const;
    private:
        char id;
        std::vector<int> nodes;
};

#endif /* COURSE_H */
```

Listing 8: course.cpp

```
/**
 * @file course.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to hold data about a course in an event.
 */

#include "course.h"

/**
 * Initialises an instance of a course with an id
 * and a set of nodes
 * @param id the id of the course
 * @param nodes the nodes in the course
 */
Course::Course(char id, std::vector<int> nodes) {
        SetId(id);
        SetNodes(nodes);
}

Course::~Course() {
}

/**
 * Set the list of nodes in this course
 * @param nodes the vector of nodes.
 */
void Course::SetNodes(std::vector<int> nodes) {
    this->nodes = nodes;
}

/**
 * Get the list of nodes in this course
 * @return the vector of nodes.
 */
std::vector<int> Course::GetNodes() const {
    return nodes;
}

/**
 * Set the ID of this course
 * @param id the ID of the course
 */
void Course::SetId(char id) {
    this->id = id;
}

/**
 * Set the list of nodes in this course
 * @return the ID of the course.
 */
char Course::GetId() const {
    return id;
}
```

Listing 9: fileio.h

```
/**
```

```
 * @file fileio.h
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to read in data files and write out the created event.
 */

#ifndef FILEIO_H
#define FILEIO_H

#include <vector>
#include <string>

#include "event.h"
#include "entrant.h"
#include "course.h"

class FileIO {
public:
    FileIO();
    virtual ~FileIO();

        void WriteEvent(Event event);
    std::vector<int> ReadNodesList(std::string filename);
private:
    bool WriteCoursesFile(std::string filename, std::vector<Course> courses);
    bool WriteEntrantsFile(std::string filename, std::vector<Entrant> entrants);
    bool WriteEventFile(std::string filename, Event event);
};

#endif /* FILEIO_H */
```

Listing 10: fileio.cpp

```
/**
 * @file fileio.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to read in data files and write out the created event.
 */

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <sys/stat.h>

#include "fileio.h"
#include "entrant.h"
#include "course.h"
#include "event.h"

FileIO::FileIO() {
}

/**
 * Write an event to file. This makes the courses, entrants and
 * name files.
 * @param evt the event to be written to file
 */
void FileIO::WriteEvent(Event evt) {
        mkdir (evt.GetName().c_str(), 0755);

    WriteEventFile(evt.GetName() + "/name.txt", evt);
    WriteCoursesFile(evt.GetName() + "/courses.txt", evt.GetCourses());
    WriteEntrantsFile(evt.GetName() + "/entrants.txt", evt.GetEntrants());

}

/**
 * Member function to write a vector of courses to a file
 * @param filename the name and path to create the file
 * @param courses the vector of courses to write to file
 * @return whether the write operation was successful
 */
bool FileIO::WriteCoursesFile(std::string filename,
        std::vector<Course> courses) {
    using namespace std;
    ofstream out(filename.c_str());
    bool success = false;

    if(out.is_open()) {
        for(std::vector<Course>::iterator it = courses.begin();
                it != courses.end(); ++it) {
            out << it->GetId() << " ";
            out << it->GetNodes().size() << " ";

            std::vector<int> nodes = it->GetNodes();
            for(std::vector<int>::iterator jt = nodes.begin();
                    jt != nodes.end(); ++jt) {
                out << *jt << " ";
            }

            out << endl;
        }
    }

    return success;
}

/**
 * Member function to write a vector of entrants to a file
 * @param filename the name and path to create the file
 * @param entrants the vector of entrants to write to file
 * @return whether the write operation was successful
 */
bool FileIO::WriteEntrantsFile(std::string filename,
        std::vector<Entrant> entrants) {
    using namespace std;
    ofstream out(filename.c_str());
    bool success = false;
    if(out.is_open()) {
        for(std::vector<Entrant>::iterator it = entrants.begin();
                it != entrants.end(); ++it) {
            out << it->GetId() << " ";
            out << it->GetCourse() << " ";
            out << it->GetName() << endl;
        }

        out.close();
        success = true;
    }

    return success;
}

/**
 * Member function to read in a list of nodes for a given file
 * @param filename the name and path to the nodes file
 * @return vector of nodes read in from file.
 */
std::vector<int> FileIO::ReadNodesList(std::string filename) {
    using namespace std;
    string input = "";
    ifstream in(filename.c_str());
    int number;
    char buffer[5];
    int line = 0;
    vector<int> nodes;

    if(in.is_open()) {
        while(!in.eof()) {
            line++;
            getline(in, input);
            int matches = sscanf (input.c_str(),"%d %s", &number, buffer);
            if(matches != 2) {
                cout << "Error parsing nodes file on line: " << line << endl;
                exit(-1);
            }

            nodes.push_back(number);
        }
    }

    in.close();

    return nodes;

}

/**
 * Member function to write an event to a file
 * @param filename the name and path to create the file
 * @param event the event to write to file
 * @return whether the write operation was successful
 */
bool FileIO::WriteEventFile(std::string filename, Event event) {
    using namespace std;
    ofstream out(filename.c_str());

    string name = event.GetName();
    string date = event.GetFormattedDate();
    string time = event.GetFormattedTime();

    if (out.is_open()) {

        out << name << endl;
        out << date << endl;
        out << time << endl;

        out.close();
        return true;
    } else {
        return false;
    }

}

FileIO::~FileIO() {
}
```

Listing 11: ioscanner.h

```
/**
 * @file ioscanner.h
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to read user input in from the command line in a variety of formats.
 */

#ifndef IOSCANNER_H
#define IOSCANNER_H

#include <string>

class IOScanner {
public:
    IOScanner();
    virtual ~IOScanner();

    int ReadInt();
    std::string ReadString(int limit);
    tm ReadDate();
    tm ReadTime();
};

#endif /* CONSOLE_INPUT_H */
```

Listing 12: ioscanner.cpp

```
/**
 * @file ioscanner.cpp
 * @author Samuel Jackson (slj11@aber.ac.uk)
 * @date 09 March 2013
 * @brief class to read user input in from the command line in a variety of formats.
 */

#include <iostream>
#include <limits>
#include <string>
#include <iostream>
#include <locale>

#include "ioscanner.h"

IOScanner::IOScanner() {
}

/**
 * Member function to read a single integer from standard in.
 * @return The integer that was read in
 */
int IOScanner::ReadInt() {
    using namespace std;

    int input;
    while (!(cin >> input)) {
        cout << "Input wasn't a number!\n" ;
        cin.clear();
        cin.ignore(std::numeric_limits<streamsize>::max(), '\n') ;
    }
    cin.ignore(std::numeric_limits<streamsize>::max(), '\n');

    return input;
}

/**
 * Member function to read a string from standard in.
 * @param limit the limit of the number of characters to read in.
 * @return The string that was read in
 */
std::string IOScanner::ReadString(int limit) {
    using namespace std;
    string input = "";

    do {
        getline(cin, input);

        if(input.size() >= limit) {
                cout << "Input too long!" << endl;
        }
    } while(input.size() >= limit);

    return input;
}

/**
 * Member function to read a date from standard in. Dates must be entered in
 * the format DD/MM/YY
 * @return time structure containing the date that was read in
 */
tm IOScanner::ReadDate() {
    using namespace std;
    string date;
    tm when;
    bool valid;

    do {
        valid = true;
        date = ReadString(10);

        if(!strptime(date.c_str(), "%d/%m/%y", &when)) {
            cout << "That wasn't a date!\n" << endl;
            valid = false;
        }
    } while (!valid);

    return when;
}

/**
 * Member function to read a time from standard in. Dates must be entered in
 * the format HH:mm
 * @return time structure containing the time that was read in
 */
tm IOScanner::ReadTime() {
    using namespace std;
    string time;
    tm when;
    bool valid;
    do {
        valid = true;
        time = ReadString(7);

        if(!strptime(time.c_str(), "%R", &when)) {
            cout << "That wasn't a time!" << endl;
            valid = false;
        }
    } while(!valid);


    return when;
}

IOScanner::~IOScanner() {
}
```

1.2 Compilation Output

1.3 Session Output

1.4 Generated Output Files

2 Checkpoint Manager Program Documentation

2.1 Code Listing

Listing 13: CheckpointManagerGUI.java

```java
package checkpoint.manager.gui;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.text.ParseException;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;

import javax.swing.DefaultListModel;
import javax.swing.DefaultListSelectionModel;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSpinner;
import javax.swing.SpinnerDateModel;

import checkpoint.manager.FileIO;
import checkpoint.manager.datamodel.CPType;
import checkpoint.manager.datamodel.Checkpoint;
import checkpoint.manager.datamodel.CheckpointManager;
import checkpoint.manager.datamodel.Entrant;
import checkpoint.manager.exceptions.ArgumentParseException;

/**
 * The Class CheckpointManagerGUI.
 */
@SuppressWarnings("serial")
public class CheckpointManagerGUI extends JFrame {

    /** The checkpoint list model to store checkpoints in the GUI. */
    private final DefaultListModel cpListModel;

    /** The checkpoint list to display checkpoints in order. */
    private JList JLCheckpointList;

    /** The entrant list to display entrants in order. */
    private JList JLEntrantList;

    /** The entrant list model to store the entrant list in the GUI. */
    private DefaultListModel entrantListModel;

    /** The checkbox for excluding an entrant. */
    private final JCheckBox chkMCExcluded;

    /** The button to check in and entrant. */
    private final JButton btnCheckIn;

    /** The arrival time of the entrant. */
    private final JSpinner JarrivalTime;

    /** The departure time of the entrant. */
    private final JSpinner JdepartureTime;

    /** The checkpoint manager GUI event listener. */
    private final CheckpointManagerListener chkptListener;

    /** The checkpoint manager to process the data model. */
    private CheckpointManager cpManager;

    /** The current entrant label. */
    private final JLabel currentEntrant;

    /** The current checkpoint label. */
    private final JLabel currentCheckpoint;

    /**
     * Instantiates a new checkpoint manager GUI.
     *
     * @param args the args from the command line
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public CheckpointManagerGUI(HashMap<String, String> args) throws FileNotFoundException, IOException {
        this.setSize(500, 600);

        currentEntrant = new JLabel("Current Entrant: ");
        currentCheckpoint = new JLabel("Current Checkpoint: ");

        try {
            cpManager = new CheckpointManager(args);
            if(!cpManager.updateTimes()) {
                JOptionPane.showMessageDialog(this, "Could not read the times file!", "Error!", JOptionPane.ERROR_MESSAGE);
                System.exit(0);
            }
        } catch (ParseException ex) {
            JOptionPane.showMessageDialog(this, ex, "Could not Parse Text times file!", JOptionPane.ERROR_MESSAGE);
            System.exit(0);
        }

        chkptListener = new CheckpointManagerListener(this);
        cpListModel = new DefaultListModel();
        entrantListModel = new DefaultListModel();
        btnCheckIn = new JButton("Check In");
        chkMCExcluded = new JCheckBox("Exclude entrant for medical reasons");
        JarrivalTime = new JSpinner(new SpinnerDateModel());
        JdepartureTime = new JSpinner(new SpinnerDateModel());

        initGUI();

        JLCheckpointList.setSelectedIndex(0);
        JLEntrantList.setSelectedIndex(0);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(1, 3));
        setVisible(true);
        pack();
    }

    /**
     * Initialises the GUI.
     */
    private void initGUI() {
        JPanel temp = new JPanel();
        JPanel rightPanel = new JPanel();
        JPanel centrePanel = new JPanel();
        JPanel leftPanel = new JPanel();

        //create list of checkpoints
        JLCheckpointList = new JList(cpListModel);
        JLCheckpointList.setSelectionMode(DefaultListSelectionModel.SINGLE_SELECTION);
        JLCheckpointList.setLayoutOrientation(JList.VERTICAL);

        //populate list of checkpoints
        for (Entry<Integer, Checkpoint> entry : cpManager.getCheckpoints().entrySet()) {
            Checkpoint chk = (Checkpoint) entry.getValue();
            cpListModel.addElement(chk.getId() + " " + chk.getType().toString());
        }

        JLCheckpointList.addListSelectionListener(chkptListener);
        JScrollPane listScroller = new JScrollPane(JLCheckpointList);
        listScroller.setPreferredSize(new Dimension(250, 300));

        //layout list of checkpoints
        temp.add(new JLabel("Checkpoints: "));
        leftPanel.setLayout(new BorderLayout());
        leftPanel.add(temp, BorderLayout.NORTH);
        temp = new JPanel();
        temp.add(listScroller);
        leftPanel.add(temp, BorderLayout.SOUTH);

        //create list of entrants
        JLEntrantList = new JList(entrantListModel);
        JLEntrantList.setSelectionMode(DefaultListSelectionModel.SINGLE_SELECTION);
        JLEntrantList.setLayoutOrientation(JList.VERTICAL);
        refreshEntrants();

        JLEntrantList.addListSelectionListener(chkptListener);

        listScroller = new JScrollPane(JLEntrantList);
        listScroller.setPreferredSize(new Dimension(250, 300));

        //layout list of entrants
        rightPanel.setLayout(new BorderLayout());
        temp = new JPanel();
        temp.add(new JLabel("Entrants: "));
        rightPanel.add(temp);
        rightPanel.add(temp, BorderLayout.NORTH);
        temp = new JPanel();
        temp.add(listScroller);
        rightPanel.add(temp, BorderLayout.SOUTH);

        //create centre panel
        JarrivalTime.setModel(new SpinnerDateModel());
        JarrivalTime.setEditor(new JSpinner.DateEditor(JarrivalTime, "HH:mm"));
        JdepartureTime.setModel(new SpinnerDateModel());
        JdepartureTime.setEditor(new JSpinner.DateEditor(JdepartureTime, "HH:mm"));

        btnCheckIn.setActionCommand("CheckIn");
        btnCheckIn.addActionListener(chkptListener);

        //layout elements in centre panel
        centrePanel.setLayout(new BorderLayout());

        temp = new JPanel();

        JPanel first = new JPanel();
        first.add(currentEntrant);
        temp.add(first);
        first = new JPanel();
        first.add(currentCheckpoint);
        temp.add(first);

        JPanel second = new JPanel();
        second.add(new JLabel("Arrival Time: "));
        second.add(JarrivalTime);
        temp.add(second);

        JPanel third = new JPanel();
        third.add(new JLabel("Dpearture Time: "));
        third.add(JdepartureTime);
        temp.add(third);

        JPanel fourth = new JPanel();
        fourth.add(chkMCExcluded);
        temp.add(fourth);

        JPanel fifth = new JPanel();
        fifth.add(btnCheckIn);
        temp.add(fifth);
        centrePanel.add(temp, BorderLayout.CENTER);
        centrePanel.setPreferredSize(new Dimension(300, 100));

        getContentPane().add(leftPanel);
        getContentPane().add(centrePanel);
        getContentPane().add(rightPanel);
    }

    /**
     * Parses the ID from the start of a list box item.
     *
     * @param list the list model
     * @param index the index of the selected item
     * @return the ID
     */
    private int parseIndex(DefaultListModel list, int index) {
        return (Integer.parseInt(list.get(index).toString().split("[a-z ]")[0]));
    }

    /**
     * Check in an entrant in response to a users click.
     */
    public void doCheckIn() {
        int index = JLEntrantList.getSelectedIndex();
        int entrantId = parseIndex(entrantListModel, index);
        index = JLCheckpointList.getSelectedIndex();
        int checkpointId = parseIndex(cpListModel, index);
        Checkpoint checkpoint = cpManager.getCheckpoint(checkpointId);

        Date arrivalTime = (Date) JarrivalTime.getValue();
        Date departureTime = null;
        boolean mcExcluded = chkMCExcluded.isSelected();
        boolean successful = false;
        boolean validInput = true;

        //reload the times file.
        try {
            successful = cpManager.updateTimes();
            if(!successful) {
                JOptionPane.showMessageDialog(this, "Could not reload times! Perhaps file was locked by another process?");
            }
        } catch (FileNotFoundException ex) {
            JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
        } catch (ParseException ex) {
            JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
        }

        if(successful) {
            //check if we're at a medical checkpoint
            if(JdepartureTime.isEnabled()) {
                departureTime = (Date) JdepartureTime.getValue();
            }

            //check if the times entered were valid
            if((checkpoint.getType()==CPType.MC && cpManager.compareTime(arrivalTime, departureTime))
                    || !cpManager.checkValidTime(entrantId, arrivalTime)) {
                JOptionPane.showMessageDialog(this, "Invalid time data!");
                validInput = false;
            }

            if(validInput) {
                //check if the entrant will be excluded with this update
                if(cpManager.willExcludedEntrant(entrantId, checkpointId) || mcExcluded) {
                    //confirm this with the user.
                    int confirm = JOptionPane.showConfirmDialog(this,
                            "This will exclude the entrant. Are you sure?",
                            "Confirm Choice", JOptionPane.YES_NO_OPTION);
                    validInput = (confirm == JOptionPane.YES_OPTION) ? true : false;
                }
            }
        }

        if(validInput) {

            //perform the update
            try {
                successful = cpManager.checkInEntrant(entrantId, checkpointId, arrivalTime, departureTime, mcExcluded);
                refreshEntrants();
            } catch (FileNotFoundException ex) {
                JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
            } catch (ParseException ex) {
                JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
            }

            //feedback to the user if successful
            if(successful) {
                JOptionPane.showMessageDialog(this, "Checked in!");
            } else {
                JOptionPane.showMessageDialog(this, "Could not check in entrant! Perhaps file was locked by another process?");
            }

            try {
                successful = cpManager.updateLog("Checked in entrant " + entrantId + " @ node " + checkpointId);
            } catch (FileNotFoundException ex) {
                JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(this, ex, "Error:", JOptionPane.ERROR_MESSAGE);
            }

            if(!successful) {
                JOptionPane.showMessageDialog(this, "Could not write to log file!");
            }
        }
    }

    /**
     * Update the GUI "currently selected" labels in response to user interaction.
     */
    public void updateOutput() {
        int index = JLCheckpointList.getSelectedIndex();

        if(index >= 0) {
            String currentChkpt = cpListModel.get(index).toString();
            currentCheckpoint.setText("Current Checkpoint: " + currentChkpt);
        }

        index = JLEntrantList.getSelectedIndex();
        if(index >= 0) {
            String entrant = entrantListModel.get(index).toString();
            currentEntrant.setText("Current Entrant: " + entrant);
        }
    }

    /**
     * Toggle input for a medical checkpoint
     */
    public void toggleMedicalCPInput() {
        int index = JLCheckpointList.getSelectedIndex();
        int cpId = (Integer.parseInt(cpListModel.get(index).toString().split("[a-z ]")[0]));
        if(cpManager.getCheckpoint(cpId).getType() == CPType.MC) {
            JdepartureTime.setEnabled(true);
            chkMCExcluded.setEnabled(true);
        } else {
            JdepartureTime.setEnabled(false);
            chkMCExcluded.setEnabled(false);
        }
    }

    /**
     * The main method and entry point to the application.
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        try {
            HashMap<String, String> cmdArgs;
            cmdArgs = FileIO.parseArgs(args);
            new CheckpointManagerGUI(cmdArgs);
        } catch (ArgumentParseException ex) {
            printHelp();
```

```
                System.exit(0);
            } catch (FileNotFoundException ex) {
                JOptionPane.showMessageDialog(null, ex, "Error:", JOptionPane.ERROR_MESSAGE);
                System.exit(0);
            } catch (IOException ex) {
                JOptionPane.showMessageDialog(null, ex, "Error:", JOptionPane.ERROR_MESSAGE);
                System.exit(0);
            }
        }

        /**
         * Prints the help menu to the console.
         */
        private static void printHelp() {
                System.out.println("Checkpoint Manager -- Usage:");
                System.out.println("Please supply the following files using the given flags");
                System.out.println(" -E <entrants file>");
                System.out.println(" -C <courses file>");
                System.out.println(" -K <checkpoints file>");
                System.out.println(" -T <times file>");
                System.out.println(" -L <log file>");
        }

        /**
         * Refresh the list of entrants.
         */
        private void refreshEntrants() {
            entrantListModel = new DefaultListModel();
            Iterator<Entry<Integer,Entrant>> it = cpManager.getEntrants().entrySet().iterator();
            while (it.hasNext()) {
                Entrant e = (Entrant) ((Entry<Integer, Entrant>) it.next()).getValue();
                if(!(e.isExcluded() || e.isFinished())) {
                    entrantListModel.addElement(e.getId() + " " + e.getName());
                }
            }

            JLEntrantList.setModel(entrantListModel);
            JLEntrantList.setSelectedIndex(0);
        }
}
```

Listing 14: CheckpointManagerListener.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package checkpoint.manager.gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

// TODO: Auto-generated Javadoc
/**
 * The listener interface for receiving checkpointManager events.
 * The class that is interested in processing a checkpointManager
 * event implements this interface, and the object created
 * with that class is registered with a component using the
 * component's <code>addCheckpointManagerListener<code> method. When
 * the checkpointManager event occurs, that object's appropriate
 * method is invoked.
 *
 * @author samuel
 */
public class CheckpointManagerListener implements ActionListener, ListSelectionListener {

    /** The parent. */
    private final CheckpointManagerGUI parent;

```

```java
    /**
     * Instantiates a new checkpoint manager listener.
     *
     * @param parent the parent
     */
    CheckpointManagerListener(CheckpointManagerGUI parent) {
        this.parent = parent;
    }

    /* (non-Javadoc)
     * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
     */
    @Override
    public void actionPerformed(ActionEvent ae) {
        if(ae.getActionCommand().equals("CheckIn")) {
            parent.doCheckIn();
        }
    }

    /* (non-Javadoc)
     * @see javax.swing.event.ListSelectionListener#valueChanged(javax.swing.event.ListSelectionEvent)
     */
    @Override
    public void valueChanged(ListSelectionEvent lse) {
        parent.updateOutput();
        parent.toggleMedicalCPInput();
    }

}
```

Listing 15: CheckpointManager.java

```java
package checkpoint.manager.datamodel;

import checkpoint.manager.FileIO;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.PriorityQueue;

/**
 * The Class CheckpointManager.
 * Main management class to the underlying data model.
 * Manages the processing and updating of data from user input via the GUI
 * into the data files.
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class CheckpointManager {

    /** The FileIO object to write to files. */
    private final FileIO fio;

    /** The LinkedHashMap of entrants. Entrant ID used as key. */
    private LinkedHashMap<Integer, Entrant> entrants;

    /** The LinkedHashMap of checkpoints. Checkpoint ID used as key */
    private LinkedHashMap<Integer, Checkpoint> checkpoints;

    /** The HashMap of courses. Course ID used as key */
    private HashMap<Character, Course> courses;

    /** The PriorityQueue of times. Oldest time has highest priority */
    private PriorityQueue<CPTimeData> times;

    /**
     * Instantiates a new checkpoint manager.
     *
     * @param args the arguments supplied via the command line.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     * @throws ParseException the parse exception thrown by failing to parse a date.
     */
    public CheckpointManager(HashMap<String, String> args)
            throws FileNotFoundException, IOException, ParseException {

        fio = new FileIO(args);
        entrants = fio.readEntrants();
        checkpoints = fio.readCheckpoints();
        courses = fio.readCourses(checkpoints);
    }

    /**
     * Check if updating an entrant to the given checkpoint ID will cause the
     * entrant to be excluded.
     *
     * @param entrantId the entrant's id
     * @param chkptId the checkpoint id
     * @return true, if successful
     */
    public boolean willExcludedEntrant(int entrantId, int chkptId) {

        Entrant entrant = getEntrant(entrantId);
        Course course = courses.get(entrant.getCourse());

        if(!entrant.isFinished()) {
            if(course.getNode(entrant.getPosition()+1) != chkptId
                && (!entrant.hasStarted() || entrant.getLatestTime().getUpdateType() != 'A')) {
                return true;
            }
        }

        return false;
    }

    /**
     * Re-read the times file and update all entrants with a new set of times.
     *
     * @return true, if successful in reading the file
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws ParseException the parse exception if a date could not be parsed.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public boolean updateTimes()
            throws FileNotFoundException, ParseException, IOException {
        times = fio.readCheckpointData(entrants, courses);

        //Failed to acquire lock or not
        return (times != null);
    }

    /**
     * Check compare the time part of two instances of a date object
     *
     * @param time the first time to be compared
     * @param time2 the second time to be compared
     * @return true, if the time is valid
     */
    public boolean compareTime(Date time, Date time2) {
        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm");
        return sdf.format(time).compareTo(sdf.format(time2)) >= 0;
    }

    /**
     * Check if the supplied time is a valid time.
     *
     * @param entrantId the entrant ID
     * @param time the time to be checked.
     * @return true, if the time is valid
```

```java
     */
    public boolean checkValidTime(int entrantId, Date time) {
        Entrant entrant = getEntrant(entrantId);
        if(entrant.hasStarted()) {
            if(compareTime(entrant.getLatestTime().getTime(), time)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Check in entrant.
     *
     * @param entrantId the entrant ID
     * @param chkptId the checkpoint ID
     * @param arrivalTime the arrival time of the entrant
     * @param departureTime the departure time of the entrant
     * @param mcExcluded the flag for if the entrant is exlcuded for medical reasons
     * @return true, if successful at writing data to file.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     * @throws ParseException the parse exception if a date could not be parsed.
     */
    public boolean checkInEntrant(int entrantId, int chkptId,
            Date arrivalTime, Date departureTime, boolean mcExcluded)
            throws FileNotFoundException, IOException, ParseException {

        boolean checkedIn = false;
        Date checkInTime;
        Entrant entrant = entrants.get(entrantId);
        Checkpoint chkpoint = checkpoints.get(chkptId);
        Course course = courses.get(entrant.getCourse());
        char updateType = 'T';

        if(!entrant.isExcluded()) {
            checkInTime = arrivalTime;

            //set arrival time if medical checkpoint
            if (chkpoint.getType() == CPType.MC) {
                checkInTime = departureTime;
                addEntrantTime(entrantId, chkptId, arrivalTime, 'A', CPType.MC);
                updateType = 'D';
            }

            CPType type = (updateType == 'D') ? CPType.MC : CPType.CP;

            //exclude entrant if they failed for medical reasons
            if (mcExcluded) {
                entrant.setExcluded(true);
                updateType = 'E';
            }

            //exclude entrant if they came to wrong checkpoint
            if(willExcludedEntrant(entrant.getId(), chkpoint.getId())) {
                entrant.setExcluded(true);
                updateType = 'I';
            }

            //check if the entrant is after this update
            if(entrant.getPosition() >= course.getLength()-2) {
                entrant.setFinished(true);
            }

            addEntrantTime(entrantId, chkptId, checkInTime, updateType, type);
            entrant.incrementPosition();
            checkedIn = fio.writeTimes(times);
        }

        return checkedIn;
    }

    /**
     * Output an update to the log file.
     * @param output the output to add to the log file.
     * @return true, if updating the log file was successful
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     */
    public boolean updateLog(String output) throws FileNotFoundException, IOException {
        return fio.writeLog(output);
    }

    /**
     * Creates a time update and adds it to the list of times and the entrant's
     * time list.
     *
     * @param entrantId the entrant ID
     * @param chkptId the checkpoint ID
     * @param date the time of the update
     * @param updateType the type of update (T, I, A, D, E)
     * @param type the type of checkpoint.
     */
    private void addEntrantTime(int entrantId, int chkptId, Date date, char updateType, CPType type) {
        CPTimeData time = new CPTimeData();
        time.setTime(date);
        time.setEntrantId(entrantId);
        time.setType(type);
        time.setUpdateType(updateType);
        time.setNode(chkptId);
        entrants.get(entrantId).addTime(time);
        times.add(time);
    }

    /**
     * Gets an entrant with the given ID.
     *
     * @param id the ID of the entrant
     * @return the entrant with the given ID
     */
    public Entrant getEntrant(int id) {
        return getEntrants().get(id);
    }

    /**
     * Gets a checkpoint with the given ID
     *
     * @param id the ID of the checkpoint
     * @return the checkpoint with the given ID
     */
    public Checkpoint getCheckpoint(int id) {
        return getCheckpoints().get(id);
    }

    /**
     * Gets the list of entrants.
     *
     * @return the entrant list
     */
    public HashMap<Integer, Entrant> getEntrants() {
        return entrants;
    }

    /**
     * Gets the list of checkpoints.
     *
     * @return the checkpoint list
     */
    public LinkedHashMap<Integer, Checkpoint> getCheckpoints() {
        return checkpoints;
    }
```

```
}
```

Listing 16: Entrant.java

```

package checkpoint.manager.datamodel;

import java.util.ArrayList;

/**
 * The Class Entrant.
 * Holds data about a single entrant in the event.
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class Entrant {

    /** The name of the entrant. */
    private String name;

    /** The course the entrant is on. */
    private char course;

    /** The id of the entrant. */
    private int id;

    /** The list of time updates an entrant has been checked in on. */
    private ArrayList<CPTimeData> times;

    /** Whether the entrant has been exlcuded or not. */
    private boolean excluded;

    /** Whether the entrant has finished or not. */
    private boolean finished;

    /** The position of the entrant on the course. */
    private int position;

    /**
     * Instantiates a new entrant.
     */
    public Entrant() {
        times = new ArrayList<CPTimeData>();
        excluded = false;
        finished = false;
        position = -1;
    }

    /**
     * Gets the name of this entrant.
     *
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * Sets the name of this entrant.
     *
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Gets the course the entrant is on.
     *
     * @return the course
     */
    public char getCourse() {
        return course;
    }

    /**
     * Sets the course the entrant is on.
     *
     * @param course the course to set
     */
    public void setCourse(char course) {
        this.course = course;
    }

    /**
     * Gets the id of the entrant.
     *
     * @return the id
     */
    public int getId() {
        return id;
    }

    /**
     * Sets the id of the entrant.
     *
     * @param id the id to set
     */
    public void setId(int id) {
        this.id = id;
    }

    /**
     * Gets the times the entrant has been check in at.
     *
     * @return the times
     */
    public ArrayList<CPTimeData> getTimes() {
        return times;
    }

    /**
     * Sets the times the entrant has been check in at.
     *
     * @param times the times to set
     */
    public void setTimes(ArrayList<CPTimeData> times) {
        this.times = times;
    }

    /**
     * Adds a time update to the entrant
     *
     * @param cpData the cp data
     */
    public void addTime(CPTimeData cpData) {
        this.times.add(cpData);
    }

    /**
     * Checks if is excluded.
     *
     * @return the excluded
     */
    public boolean isExcluded() {
        return excluded;
    }

    /**
     * Sets the as excluded or not.
     *
     * @param excluded the excluded to set
     */
    public void setExcluded(boolean excluded) {
        this.excluded = excluded;
    }

    /**
     * Gets the position of the entrant.
     *
     * @return the position
     */
    public int getPosition() {
        return position;
    }

    /**
     * Reset position of the entrant.
     */
    public void resetPosition() {
        position = −1;
    }

    /**
     * Increment position of the entrant.
     */
    public void incrementPosition() {
        position++;
    }

    /**
     * Check if the entrant has started.
     *
     * @return true, if entrant has started
     */
    public boolean hasStarted() {
        return (times.size() > 0);
    }

    /**
     * Gets the latest time currently avalible for the entrant.
     *
     * @return the latest time
     */
    public CPTimeData getLatestTime() {
        return times.get(times.size()−1);
    }

    /**
     * Checks if is finished has finished.
     *
     * @return the finished
     */
    public boolean isFinished() {
        return finished;
    }

    /**
     * Sets the finished as been finished or not.
     *
     * @param finished the finished to set
     */
    public void setFinished(boolean finished) {
        this.finished = finished;
    }
}
```

Listing 17: Course.java

```
package checkpoint.manager.datamodel;

import java.util.ArrayList;

/**
```

```
 * The Class Course.
 * Holds data about a single course
 *
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class Course {

    /** The id of the course */
    private char id;

    /** The nodes in the course */
    private ArrayList<Integer> nodes;

    /**
     * Gets the id of the course.
     *
     * @return the id
     */
    public char getId() {
        return id;
    }

    /**
     * Sets the id of the course.
     *
     * @param id the id to set
     */
    public void setId(char id) {
        this.id = id;
    }

    /**
     * Gets the length.
     *
     * @return the length
     */
    public int getLength() {
        return nodes.size();
    }

    /**
     * Gets the nodes in the course.
     *
     * @return the nodes
     */
    public ArrayList<Integer> getNodes() {
        return nodes;
    }

    /**
     * Sets the nodes in the course.
     *
     * @param nodes the nodes to set
     */
    public void setNodes(ArrayList<Integer> nodes) {
        this.nodes = nodes;
    }

    /**
     * Gets the node.
     *
     * @param index the index of the node.
     * @return the node
     */
    public int getNode(int index) {
        return getNodes().get(index);
    }
}
```

Listing 18: Checkpoint.java

```java
package checkpoint.manager.datamodel;

/**
 * The Class Checkpoint.
 * Holds data about a single checkpoint (or medical checkpoint) in an event.
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class Checkpoint {

    /** The id of the checkpoint */
    private int id;

    /** The type of the checkpoint. */
    private CPType type;

    /**
     * Instantiates a new checkpoint.
     *
     * @param id the id of the checkpoint
     * @param type the type of the checkpoint
     */
    public Checkpoint(int id, String type) {
        this.id = id;
        this.type = CPType.valueOf(type);
    }

    /**
     * Gets the id of the checkpoint.
     *
     * @return the id
     */
    public int getId() {
        return id;
    }

    /**
     * Gets the type type of the checkpoint.
     *
     * @return the type
     */
    public CPType getType() {
        return type;
    }
}
```

Listing 19: CPTimeData.java

```java
package checkpoint.manager.datamodel;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

/**
 * The Class CPTimeData.
 * Holds data about a single checkpoint time update.
 *
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class CPTimeData implements Comparable<CPTimeData> {

    /** The entrant id of the entrant. */
    private int entrantId;

    /** The type of checkpoint. */
    private CPType type;

    /** The update type. One of the 5 types of updates allowed (T, I, A, D, E) . */
    private char updateType;

    /** The node that the checkpoint update occurred on. */
    private int node;

    /** The time the update occurred. */
    private Date time;

    /** The date formatter object. */
    private final SimpleDateFormat sdf;

    /**
     * Instantiates a new instance of a checkpoint time data object.
     */
    public CPTimeData() {
        sdf = new SimpleDateFormat("HH:mm");
    }

    /**
     * Gets the entrant's id.
     *
     * @return the entrantId
     */
    public int getEntrantId() {
        return entrantId;
    }

    /**
     * Sets the entrant id.
     *
     * @param entrantId the entrantId to set
     */
    public void setEntrantId(int entrantId) {
        this.entrantId = entrantId;
    }

    /**
     * Gets the type.
     *
     * @return the type
     */
    public CPType getType() {
        return type;
    }

    /**
     * Sets the type of update.
     *
     * @param type the type to set
     */
    public void setType(CPType type) {
        this.type = type;
    }

    /**
     * Gets the node that the update occurred on.
     *
     * @return the cpId
     */
    public int getNode() {
        return node;
    }

    /**
     * Sets the node that the update occurred on.
     *
     * @param checkpointId the cpId to set
     */
    public void setNode(int checkpointId) {
        this.node = checkpointId;
    }

    /**
     * Gets the time as a string.
     *
     * @return the time
     */
    public String getStringTime() {
        return sdf.format(time);
    }

    /**
     * Gets the time (Date) object.
     *
     * @return the time
     */
    public Date getTime() {
        return time;
    }

    /**
     * Sets the time.
     *
     * @param time the new time
     */
    public void setTime(Date time) {

        this.time = time;
    }

    /**
     * Gets the update type. One of the 5 types of updates (T,I,A,D,E)
     *
     * @return the updateType
     */
    public char getUpdateType() {
        return updateType;
    }

    /**
     * Sets the update type. One of the 5 types of updates (T,I,A,D,E)
     *
     * @param updateType the updateType to set
     */
    public void setUpdateType(char updateType) {
        this.updateType = updateType;
    }

    /* (non-Javadoc)
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    @Override
    public int compareTo(CPTimeData t) {
        return sdf.format(time).compareTo(sdf.format(t.getTime()));
    }
}
```

Listing 20: CPType.java

```
package checkpoint.manager.datamodel;

// TODO: Auto-generated Javadoc
/**
 * The Enum CPType.
 * The used to represent the type of a checkpoint, either regular or medical.
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public enum CPType {
    CP,
    MC
}
```

Listing 21: FileIO.java

```java

package checkpoint.manager;

import checkpoint.manager.datamodel.CPTimeData;
import checkpoint.manager.datamodel.Checkpoint;
import checkpoint.manager.datamodel.Course;
import checkpoint.manager.datamodel.Entrant;
import checkpoint.manager.exceptions.ArgumentParseException;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.RandomAccessFile;
import java.nio.channels.FileLock;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map.Entry;
import java.util.PriorityQueue;
import java.util.Scanner;

/**
 * The Class FileIO.
 * Reads and writes files used during a race event.
 *
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
public class FileIO {

    /** The simple date formatter */
    private SimpleDateFormat sdf;

    /** The names of each of the files passed as command line arguements. */
    private HashMap<String, String> filenames;

    /**
     * Instantiates a new instace of FileIO.
     *
     * @param args HashMap of filenames
     */
    public FileIO (HashMap<String, String> args) {
        filenames = args;
        sdf = new SimpleDateFormat("HH:mm");
    }

    /**
     * Parses the command line arguments.
     *
     * @param args the command line arguments
     * @return HashMap of parse arguments
     * @throws ArgumentParseException the argument parse exception thrown if
     * arguments array cannot be parsed.
     */
    public static HashMap<String, String> parseArgs(String[] args)
            throws ArgumentParseException {
        HashMap<String, String> argsList = new HashMap<String, String>();

        if (args.length == 10) { //all arguments are required
            for (int i = 0; i < args.length; i+=2) {
                String key = "";
                switch(args[i].charAt(1)) {
                    case 'E':
                        key = "entrants";
                        break;
                    case 'T':
                        key = "times";
                        break;
                    case 'C':
                        key = "courses";
                        break;
                    case 'K':
                        key = "checkpoints";
                        break;
                    case 'L':
                        key = "log";
                        break;
                    default:
                        throw new ArgumentParseException();
                }

                argsList.put(key, args[i+1]);
            }
        } else {
            throw new ArgumentParseException();
        }

        return argsList;
    }

    /**
     * Read in the entrant's file.
     *
     * @return the linked HashMap of entrant's, identified by an entrant's ID.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public LinkedHashMap<Integer, Entrant> readEntrants()
            throws FileNotFoundException, IOException {
        Scanner in = new Scanner(new File(filenames.get("entrants")));
        LinkedHashMap<Integer, Entrant> entrants = new LinkedHashMap<Integer, Entrant>();

        while(in.hasNext()) {
            Entrant e = new Entrant();
            e.setId(in.nextInt());
            e.setCourse(in.next().charAt(0));
            e.setName(in.nextLine());
            entrants.put(e.getId(),e);
        }

        in.close();

        return entrants;
    }

    /**
     * Read in the courses file.
     *
     * @param checkpoints the HashMap of nodes that are checkpoints (or medical checkpoints).
     * @return HashMap of courses, identified by the course ID.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public HashMap<Character, Course> readCourses(LinkedHashMap<Integer, Checkpoint> checkpoints)
            throws FileNotFoundException, IOException {
        Scanner in = new Scanner(new File(filenames.get("courses")));

        HashMap<Character, Course> courses = new HashMap<Character, Course>();

        while (in.hasNext()) {
            ArrayList<Integer> nodes = new ArrayList<Integer>();
            Course course = new Course();
            course.setId(in.next().charAt(0));

            while(in.hasNextInt()) {
                int node = in.nextInt();
                if(checkpoints.containsKey(node)) {
                    nodes.add(node);
                }
            }
            course.setNodes(nodes);
            courses.put(course.getId(), course);
        }

        in.close();

        return courses;
    }

    /**
     * Read checkpoint data.
     *
     * @param entrants the list of entrants to update.
     * @param courses the list of all courses.
     * @return PriorityQueue of CPTimeData objects, ordered by oldest time first.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws ParseException the parse exception thrown when a date cannot be parsed.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public PriorityQueue<CPTimeData> readCheckpointData(
            LinkedHashMap<Integer, Entrant> entrants, HashMap<Character, Course> courses)
            throws FileNotFoundException, ParseException, IOException {
        RandomAccessFile fis = new RandomAccessFile(filenames.get("times"), "rw");
        FileLock fl = fis.getChannel().tryLock();
        Scanner in = new Scanner(fis.getChannel());

        PriorityQueue<CPTimeData> times = null;
        Entrant entrant;

        //clear out the entrants times and reset
        for (Entry<Integer, Entrant> entry : entrants.entrySet()) {
            entrant = (Entrant) entry.getValue();
            entrant.setTimes(new ArrayList<CPTimeData>());
            entrant.resetPosition();
        }

        //if we have locked the file
        if(fl != null) {
            times = new PriorityQueue<CPTimeData>();

            while (in.hasNext()) {
                CPTimeData chkpoint = new CPTimeData();
                char type = in.next().charAt(0);
                int node = in.nextInt();
                int entrantNo = in.nextInt();
                Date date = sdf.parse(in.next());
                entrant = entrants.get(entrantNo);

                //exclude entrant if necessary
                switch(type) {
                    case 'I':
                    case 'E':
                        entrant.setExcluded(true);
                        break;
                }

                //create checkpoint update data
                chkpoint.setUpdateType(type);
                chkpoint.setNode(node);
                chkpoint.setEntrantId(entrantNo);
                chkpoint.setTime(date);

                Course course = courses.get(entrant.getCourse());
                if(entrant.getPosition() >= course.getLength()-2) {
                    entrant.setFinished(true);
                }

                //update entrant and times list.
                entrant.incrementPosition();
                entrant.addTime(chkpoint);
                times.add(chkpoint);
            }

            fl.release();
        }

        in.close();
        fis.close();

        return times;
    }

    /**
     * Read in the checkpoints file.
     *
     * @return the LinkedHashMap of checkpoints (nodes) identified by ID.
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public LinkedHashMap<Integer, Checkpoint> readCheckpoints()
            throws FileNotFoundException, IOException {
        Scanner in = new Scanner(new File(filenames.get("checkpoints")));

        LinkedHashMap<Integer, Checkpoint> checkpoints = new LinkedHashMap<Integer, Checkpoint>();

        while(in.hasNext()) {
            int id = in.nextInt();
            String type = in.next();

            //ignore junctions
            if(!type.equals("JN")) {
                checkpoints.put(id, new Checkpoint(id, type));
            }
        }

        in.close();

        return checkpoints;
    }

    /**
     * Write out time data to the times file.
     *
     * @param writer the PrintWriter to use to output the time
     * @param data the data to output to file
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    private void writeTimeData(PrintWriter writer, CPTimeData data) throws FileNotFoundException, IOException {
        String time = data.getStringTime();
        String output = data.getUpdateType() + " " + data.getNode() + " " + data.getEntrantId() + " " + time;
        writer.write(output + "\n");
        writer.flush();
    }

    /**
     * Write out the list of times to file.
     *
     * @param times the list of times to output.
     * @return true, if successful at writing
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     */
    public boolean writeTimes(PriorityQueue<CPTimeData> times) throws FileNotFoundException, IOException {
        FileOutputStream fis = new FileOutputStream(new File(filenames.get("times")));
        FileLock fl = fis.getChannel().tryLock();
        PrintWriter writer = new PrintWriter(fis);
        boolean writeSuccess = false;

        //we have file lock
        if(fl != null) {
            while (!times.isEmpty()) {
                //get times in order of priority (oldest first)
                CPTimeData t = times.poll();
                writeTimeData(writer, t);
            }
            fl.release();
            writeSuccess = true;
        }

        fis.close();
        writer.close();


        return writeSuccess;

    }

    /**
     * Write to the log file.
     *
     * @param updateText the message to output to the log file
     * @throws FileNotFoundException exception thrown when file cannot be found.
     * @throws IOException Signals that an unexpected I/O exception has occurred.
     * @return true, if successful at writing
     */
    public boolean writeLog(String updateText) throws FileNotFoundException, IOException {
        String outputStr;
        Date time = new Date();
        FileOutputStream fis = new FileOutputStream(new File(filenames.get("log")), true);
        FileLock fl = fis.getChannel().tryLock();
        PrintWriter writer = new PrintWriter(fis);
        boolean writeSuccess = false;
        //we have file lock
        if(fl != null) {
            outputStr = sdf.format(time) + " CMP: " + updateText + "\n";
            writer.append(outputStr);
            writer.flush();
            writeSuccess = true;
        }
        fis.close();
        writer.close();

        return writeSuccess;
    }
}
```

Listing 22: ArgumentParseException.java

```

package checkpoint.manager.exceptions;

/**
 * The Class ArguementParseException.
 * Thrown if the command line arguments could not be parsed.
 * @author Samuel Jackson (slj11@aber.ac.uk)
 */
@SuppressWarnings("serial")
public class ArgumentParseException extends Exception{

    /* (non-Javadoc)
     * @see java.lang.Throwable#getMessage()
     */
    @Override
    public String getMessage() {
        return "Could not parse command line arguments";
    }
}
```

2.2 Compilation Output

2.3 Example Run

3 Event Manager Program Documentation

3.1 Compilation Output

3.2 Example Run Output

3.3 Example Run Results List

3.4 Output Of Log File

4 Outline of Programs

This section of the document provides a brief outline of each of the three programs included as part of this project. This includes a discussion of the basic structure, design and operation of each application.

4.1 Event Creation Program

The event creation program is a command line based application written in C++. Its purpose is to create the event, courses and entrants file for each event. The design of the application allows the user to create multiple events at the same time, rather than having to make each event in serial. Because entrants need a course and a course needs an event, an event must be created before a course and a course must be created before an entrant. This includes the functionality to create different course and entrants associated with different events. Each event also expects a nodes file to be given when creating the event, allowing different events to work with different sets of allowed nodes. The user is also able to view an event by selecting the relevant option form the main menu.

Since lists of courses and entrants are associated with each event, I decided that the best approach would be to allow the user to create all the data about an event, then write it to file, rather than creating each of the files one at a time. When the user chooses the option to write an event, a new folder is created with the name of the event as the name of the folder. Inside the folder, the event, entrants and courses files are written.

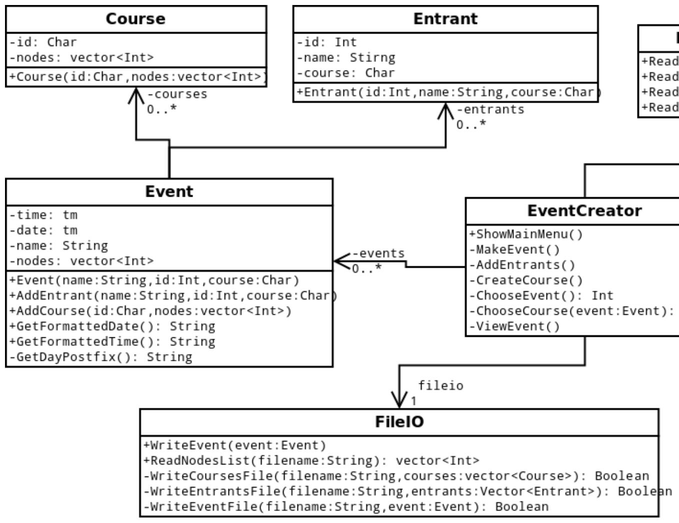
Course
-id: Char
-nodes: vector<Int>
+Course(id:Char,nodes:vector<Int>)
-courses
0..*
Entrant
-id: Int
-name: Stirng
-course: Char
+Entrant(id:Int,name:String,course:Char)
-entrants
0..*
IOScanner
+ReadInt(): Int
+ReadString(): String
+ReadDate(): tm
+ReadTime(): tm
scanner
1
Event
-time: tm
-date: tm
-name: String
-nodes: vector<Int>
+Event(name:String,id:Int,course:Char)
+AddEntrant(name:String,id:Int,course:Char)
+AddCourse(id:Char,nodes:vector<Int>)
+GetFormattedDate(): String
+GetFormattedTime(): String
-GetDayPostfix(): String
-events
0..*
EventCreator
+ShowMainMenu()
-MakeEvent()
-AddEntrants()
-CreateCourse()
-ChooseEvent(): Int
-ChooseCourse(event:Event): Int
-ViewEvent()
fileio
1
FileIO
+WriteEvent(event:Event)
+ReadNodesList(filename:String): vector<Int>
-WriteCoursesFile(filename:String,courses:vector<Course>): Boolean
-WriteEntrantsFile(filename:String,entrants:Vector<Entrant>): Boolean
-WriteEventFile(filename:String,event:Event): Boolean

Figure 1: Class diagram of the Event Creator program. Getters/Setters not shown.

4.2 Checkpoint Manager Program

The checkpoint manager program is written in Java and provides a Swing based GUI to allow the user to easily update entrants out in the field as the JVM allows the program to be executed on a variety of platforms. This program accepts the required files (entrants, courses, nodes, time and log files) as command line arguments using flags for each file. Help instructions are printed when no arguments or incorrect arguments are supplied. An example listing of arguments is supplied below:

```
java −jar checkpoint_manager.jar −E ../../event_3/entrants.txt −C ../../event_3/courses.txt −K ../../event_3/nodes.txt
 −T ../../event_3/times.txt −L ../../event_3/log.txt
```

The checkpoint manager program allows a race marshal to update the location of the entrants as they arrive at the various checkpoints on the course. Entrants are automatically excluded if checked into a checkpoint they should not of visited. The GUI also provides an option for marshals to excluded entrants based on failing a medical checkpoint. When an entrant is excluded, they are automatically removed from the list of available entrants. When an entrant is about to be excluded, the user is asked to confirm the operation, ensuring that they don't accidentally excluded a competitor.

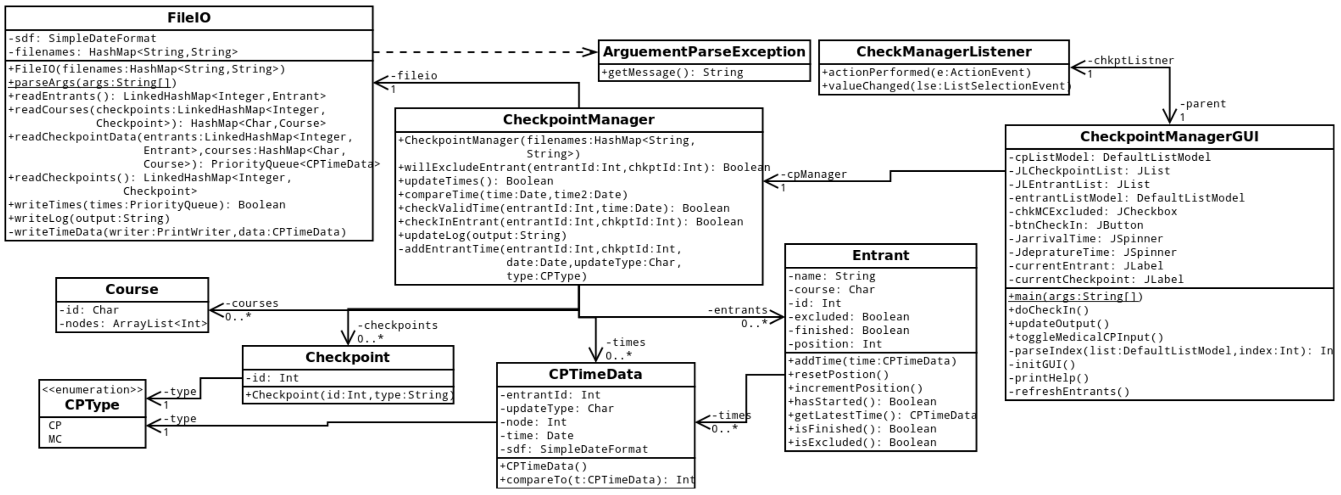
FileIO
-sdf: SimpleDateFormat
-filenames: HashMap<String,String>
+FileIO(filenames:HashMap<String,String>)
+parseArgs(args:String[])
+readEntrants(): LinkedHashMap<Integer,Entrant>
+readCourses(checkpoints:LinkedHashMap<Integer, Checkpoint>): HashMap<Char,Course>
+readCheckpointData(entrants:LinkedHashMap<Integer, Entrant>,courses:HashMap<Char, Course>): PriorityQueue<CPTimeData>
+readCheckpoints(): LinkedHashMap<Integer, Checkpoint>
+writeTimes(times:PriorityQueue): Boolean
+writeLog(output:String)
-writeTimeData(writer:PrintWriter,data:CPTimeData)
ArguementParseException
+getMessage(): String
CheckManagerListener
+actionPerformed(e:ActionEvent)
+valueChanged(lse:ListSelectionEvent)
-chkptListner
1
-parent
1
-fileio
1
CheckpointManager
+CheckpointManager(filenames:HashMap<String, String>)
+willExcludeEntrant(entrantId:Int,chkptId:Int): Boolean
+updateTimes(): Boolean
+compareTime(time:Date,time2:Date)
+checkValidTime(entrantId:Int,time:Date): Boolean
+checkInEntrant(entrantId:Int,chkptId:Int): Boolean
+updateLog(output:String)
-addEntrantTime(entrantId:Int,chkptId:Int, date:Date,updateType:Char, type:CPType)
-cpManager
1
CheckpointManagerGUI
-cpListModel: DefaultListModel
-JLCheckpointList: JList
-JLEntrantList: JList
-entrantListModel: DefaultListModel
-chkMCExcluded: JCheckbox
-btnCheckIn: JButton
-JarrivalTime: JSpinner
-JdepratureTime: JSpinner
-currentEntrant: JLabel
-currentCheckpoint: JLabel
+main(args:String[])
+doCheckIn()
+updateOutput()
+toggleMedicalCPInput()
-parseIndex(list:DefaultListModel,index:Int): In
-initGUI()
-printHelp()
-refreshEntrants()
Course
-id: Char
-nodes: ArrayList<Int>
-courses
0..*
-checkpoints
0..*
-entrants
0..*
-times
0..*
Entrant
-name: String
-course: Char
-id: Int
-excluded: Boolean
-finished: Boolean
-position: Int
+addTime(time:CPTimeData)
+resetPostion()
+incrementPosition()
+hasStarted(): Boolean
+getLatestTime(): CPTimeData
+isFinished(): Boolean
+isExcluded(): Boolean
Checkpoint
-id: Int
+Checkpoint(id:Int,type:String)
<<enumeration>>
CPType
CP
MC
-type
1
-type
1
CPTimeData
-entrantId: Int
-updateType: Char
-node: Int
-time: Date
-sdf: SimpleDateFormat
+CPTimeData()
+compareTo(t:CPTimeData): Int
-times
0..*

Figure 2: Class diagram of the Checkpoint Manager program. Getters/Setters not shown.

The event manager program allows the user to input the time a competitor arrives and, in the case of medical checkpoints, departs. The program automatically checks that the arrival time is greater than the last time the entrant was checked in. In the case of medical checkpoints, it also checks that the arrival time is not greater than the departure time. Correct order of times is tracked using a priority queue.

4.3 Event Manager Program

The event manager program is written in C and handles checking the position and state of entrants as they progress through a course. This includes viewing a list of which entrants have been excluded, finished and are currently out on a track. It also gives the user the ability to query individual competitors and provides an estimate of what track/node they should/are on.

The event manager requires the loading of all the data files for an event. This is done by prompting the user at the start of the application and only needs to be done once. Like the event manager, the application locks the log and times file when reading to prevent multiple applications crashing during file processing.