TECHNOLOGICAL UNIVERSITY DUBLIN

TALLAGHT CAMPUS

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRONIC ENGINEERING

PROJECT REPORT

YEAR 2

ACADEMIC YEAR 2020/2021

SOFTWARE DEVELOPMENT 2 ASSIGNMENT ON A KARATE SCHOOL MANAGEMENT SYSTEM

Name: Tim Jäger

Student number: x00170190

Submission Date: 14/05/2021

tudublin.ie

Programmes: TA_ERAHUM_B

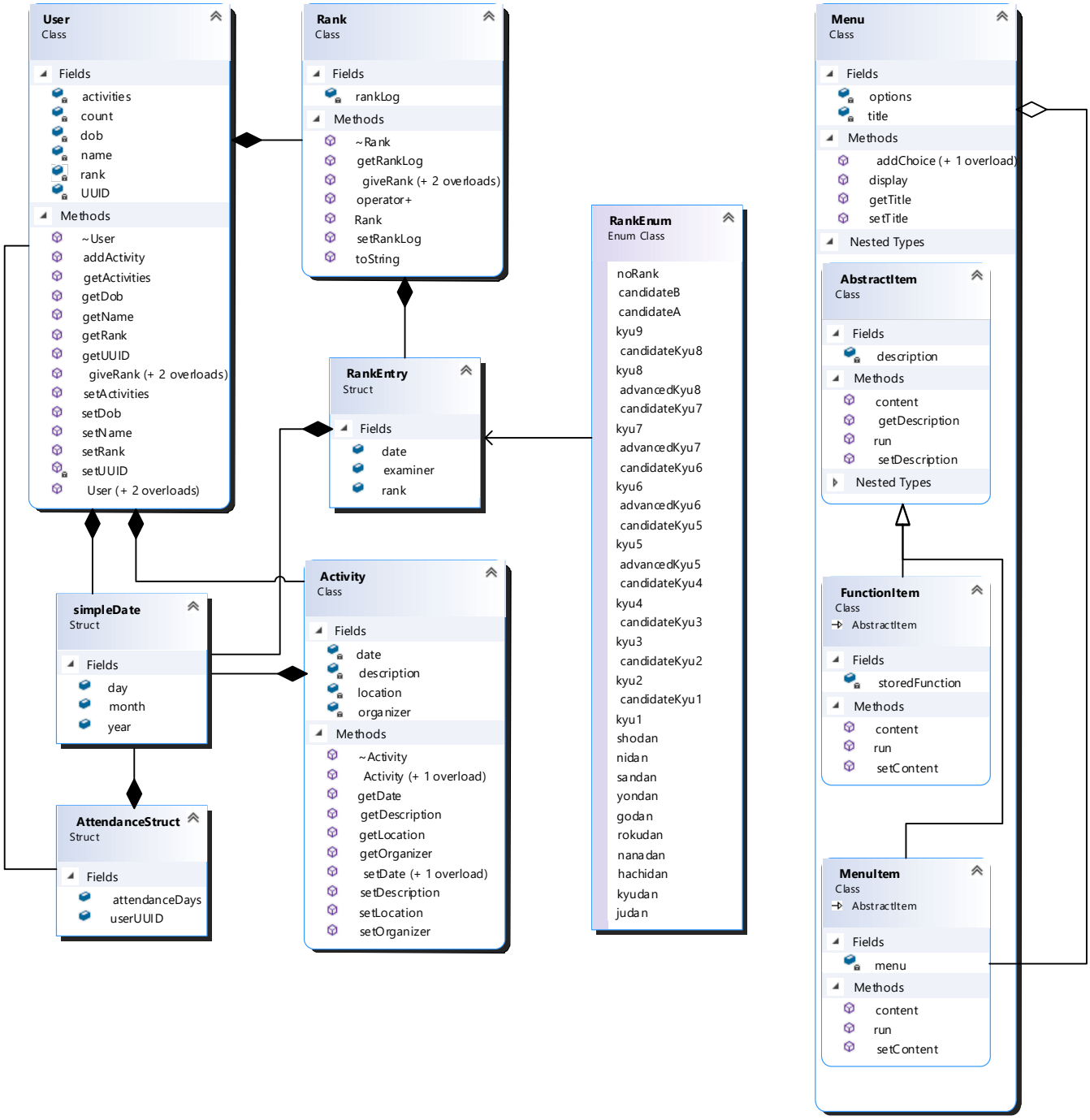
0. Description of your project application area.
    a. Description of your project application area

The Karate School Management System provides functionality for the day to day activities a karate school deals with. This includes member management and keeping track of the attendance of the students.

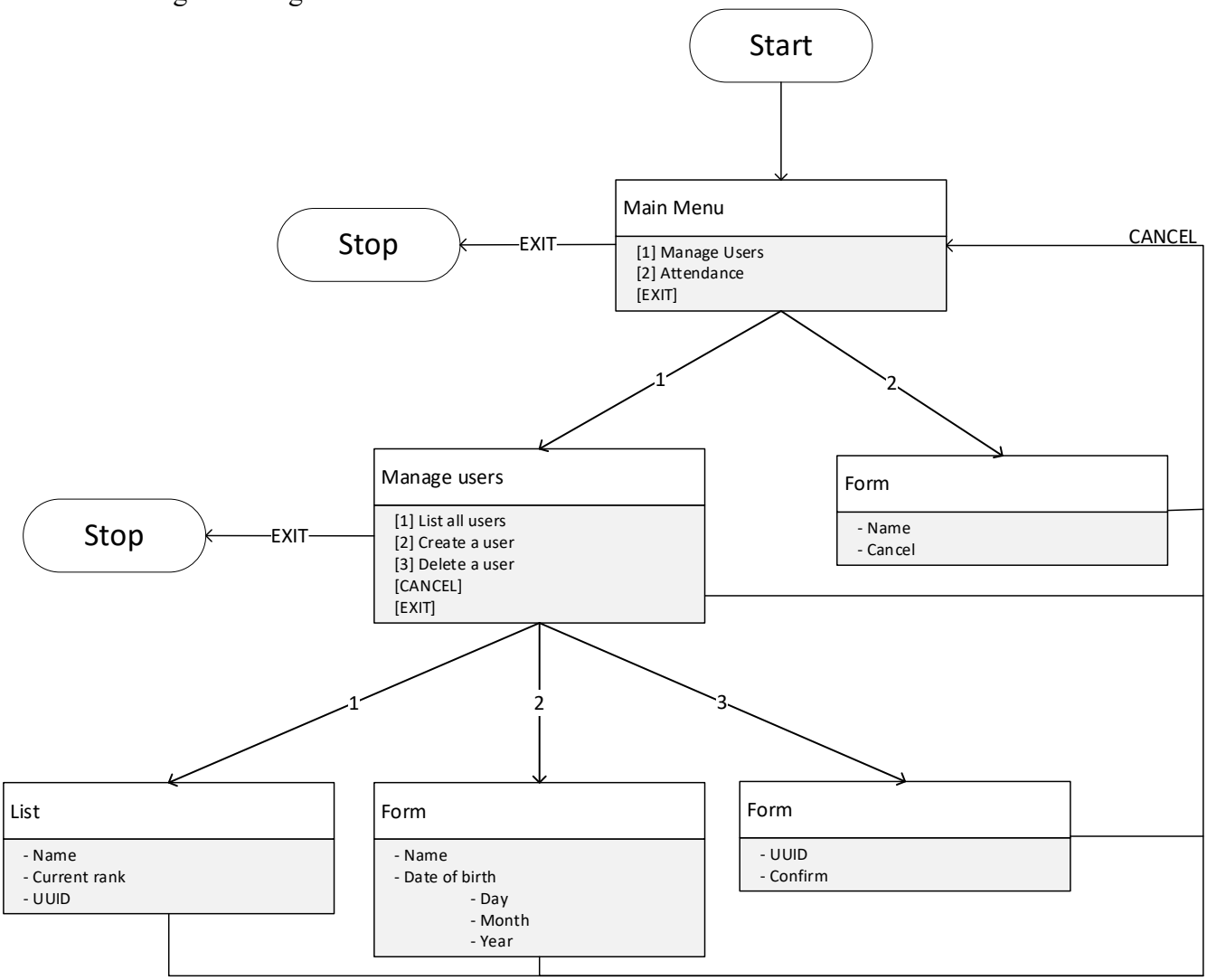
As a karate practitioner myself I found that especially in a large school as ours, with over 200 registered students, it becomes hard to keep track of all attendances and members in general. At the start of each class every student has to sign their attendance. This takes a long time as every student needs to find their name on the paper list, which can cause some delays at the start of the training as getting the attendance takes approximately 5 to 10 minutes. Registering this digitally would reduce the time needed for everyone to find their name and also be more convenient.

    b. Class Diagram, and menu and navigation options

Class diagram:



Menu and navigation diagram



1. A basic class, with access member functions, Constructors, Static member data
    a. Give the examples code that works and tested, etc. Fully comment thecode.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

User.h

```
0.  #pragma once
1.
2.  /**
3.  * Author: Tim Jäger
4.  * E-Mail: tim.jager2001@gmail.com
5.  *
6.  * Description:
7.  *      This file contains a user class, used to store
8.  *      the details of a karate practitioner.
9.  *
10. * External dependencies:
11. *      - iostream
12. *      - ctime
13. *      - Vector
14. *      - iomanip
15. *      - string
16. *
```

```
17. * User-defined dependencies:
18. *       - TJ (namespace)
19. *       - Activity
20. *       - Rank
21. */
22.
23. #include <iostream>
24. #include "TJ.h"
25.
26. #include <ctime>
27. #include <vector>
28. #include <iomanip>
29. #include <string>
30. #include "Activity.h"
31. #include "Rank.h"
32.
33. class User {
34. private:
35.         static long count; // Amount of users that have been initialized
36.         long UUID; // This is set during the object initialization, hence the set function is
    private
37.         std::string name; // This is the users real name, first and last name
38.         TJ::simpleDate dob; // Date of birth
39.         Rank rank; // The karate rank, in our style this is a set of every rank you ever got,
    see Rank.h for details
40.         std::vector<Activity> activities; // This stores all mandatory/voluntary karate
    activities
41.
42.         /* Private getters and setters */
43.         void setUUID(long UUID);
44. public:
45.         User();
46.         User(std::string name, long clubID, TJ::simpleDate dob); // New practitioners have no
    rank, so this sets an empty rank
47.         User(std::string name, long clubID, TJ::simpleDate dob, Rank rank,
    std::vector<Activity> activities);
48.
49.         ~User(); // This is currently the default destructor
50.
51.         /* Getters and setters */
52.         long getUUID(); // Only getter is public for the afore mentioned reason
53.
54.         void setName(std::string name);
55.         std::string getName();
56.
57.         void setDob(TJ::simpleDate dob);
58.         TJ::simpleDate getDob();
59.
60.         void setRank(Rank rank);
61.         void giveRank(RankEntry rankEntry); // giveRank(...) adds a rank to the list
62.         void giveRank(RankEnum rank, std::string examiner);
63.         void giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner);
64.         Rank getRank();
65.
66.         void setActivities(std::vector<Activity> activities);
67.         void addActivity(Activity activity); // This adds an activity to the list
68.         std::vector<Activity> getActivities();
69. };
```

User.cpp: functions “void getName()” and “User()”:

```
User::User() {
        User::count++;        // Everytime the constructor is called, this increases, therefore
counting the
        UUID = User::count;  // instances created and ensuring that the UUID stays unique
```

```
	this->name = "Unnamed";
	this->dob = { 1,1,1 };
}

std::string User::getName() {
	return this->name;
}
```

b. Explain how the code behaves.
c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.
I have combined points b and c, as they are too closely related in this case.

Above is the header file, which can be recognised by the ".h" extension. This contains all that is needed to answer this question. If you want the full implementation file, which is the ".cpp" file, you can find it attached at the end. To use a class, you can call on a function by using "ClassName::functionName(…)" or you can create an object, using code like below.

```
User user;
```

Here the variable "user" is an instance of "User", which is acting as a type. Functions like:

```
user.getName()
```

Will use the "getName()" function as specified in "User.h" and "User.cpp". In this case it will return "this->name" which means the variable "name" stored inside the object "user".

In C++ and object oriented programming in general, there is also something called accessibility. In this case it is implemented using the keywords in the class. In any class the date written behind a "private:" keyword, will only be accessible to anything inside that class (exceptions may apply for things like inheritance, which is discussed later on). The "public:" keyword means that anything outside the function can also access it. For this the most recent keyword counts. Therefore in the aforementioned code "void setUUID(long UUID);" is a private member and for instance can't be used in the "main()" function, while "long getUUID()" is public and can be used in the "main()" function.

Constructors are the functions that get called when you declare an object. When running this line:

```
User user;
```

The object "user" is declared and the function "User()" in the "User" class will be called. Usually this is to initialize variables and prepare everything for further use. If you look at its content in part "a.", you can see that the variable "User::count" is increased by 1, that the UUID is set to "User::count" and so on.

Then there is static member data, this data is scope independent, with the exception of its name. This is specified by writing the keyword "static" in front of the variable. In the user class this is "static long count;". When called anywhere this static data will give the same result. In this case the accessibility is private, therefore a limitation has been posed, that only objects of the same class can access it. But if I were to create multiple users, like I do in "main.cpp":

```
std::vector<User> users;
```

Here, when "User::count" is set to 3 in one user, it will be like that for all of them. And if we leave the scope in which this data was created or deleted, it will stay the same, next time you call that variable again, until the program shuts down.

In my opinion the User class is a good example for my project area, as every karate school has members, which are part of the computer system. It also does not cover to much data outside of the question.

2. Inheritance and polymorphism.
    a. Give the examples code that works and tested, etc. Fully comment the code.

Snippet of menu.h

```
/**
* This class is an interactive menu.
* It stores options (ie. functions or other menus) and can display and run them.
* The menu will look as follow:
*
* ================================================
* Title
* ------------------------------------------------
* [1] Description for 1
* [A] Description for A
* [OTHER] Description for OTHER
* ================================================
* // You can then type here your answer
*
**/
class Menu {
public:
	// These classes are defined in Menu to avoid to need to maintain backwards compatibility
whenever the Menu class is updated

	/**
	* All options in the Menu are stored in so "items". These contain all needed details, like
	* its description, a function to execute its content and a function to get its content
	*
	* As items can store different things like other menus and functions there is an abstract item
class
	* and some derived classes which contain the specifics. These derived classes their content
can be identified
	* by calling the "content()" function.
	**/
	class AbstractItem {
	private:
		std::string description; // This contains a description of the content, which the menu
class uses to dislay info

	public:
		// To make the content type clearer there is an enumerator, this has little to no other
impact
		enum class ItemType { fuction = 0, menu = 1 };

		/* Getters and setters */
		void setDescription(std::string description);
		std::string getDescription();

		/* Pure virtual functions */
		virtual ItemType content() = 0; // Function to identify the content type
		virtual void run() = 0; // This executes the centent, ie. a function will be executed
or a menu will be shown
	};

	/**
	* This is the derived class from "AbstractItem" that stores functions.
	**/
	class FunctionItem : public AbstractItem {
	private:
		std::function<void(void)> storedFunction;

	public:
		ItemType content() override { return ItemType::fuction; }
```

```
		void run() override { storedFunction(); }

		void setContent(std::function<void(void)> func) { storedFunction = func; }
	};

	/**
	* This is the derived class from "AbstractItem" that stores menus.
	**/
	class MenuItem : public AbstractItem {
	private:
		Menu* menu;

	public:
		ItemType content() override { return ItemType::menu; }

		void run() override { this->menu->display(); }

		void setContent(Menu* menu) { this->menu = menu; }
	};

private:
	std::string title; // The title of the meny that will be displayed
	std::map<std::string, std::unique_ptr<Menu::AbstractItem>> options; // This stores all options
the menu will show

public:

	/* Getters and setters */
	void setTitle(std::string title);
	std::string getTitle();

	/* Other functions */
	void addChoice(std::string name, std::string description, std::function<void(void)> func); //
Function to add a function as option
	void addChoice(std::string name, std::string description, Menu* menu); // Function to add a
menu as option

	void display(); // Function to display the meny in the console. It has an interface, making it
also handle the option selection and execution.
};
```

Snippet of menu.cpp

```
void Menu::addChoice(std::string name, std::string description, std::function<void(void)> func) {
	FunctionItem* item = new FunctionItem();
	item->setDescription(description);
	item->setContent(func);
	this->options.emplace(name, item);

}

void Menu::addChoice(std::string name, std::string description, Menu* menu) {
	MenuItem* item = new MenuItem();
	item->setDescription(description);
	item->setContent(menu);
	this->options.emplace(name, item);
}

void Menu::display() {
	/**
	* Display the actual menu in the format:
	*
```

```
        * ================================================
        * Title
        * ------------------------------------------------
        * [1] Description for 1
        * [A] Description for A
        * [OTHER] Description for OTHER
        * ================================================
        **/
        TJ::clearScreen();
        TJ::breakSection('=');

        std::cout << this->getTitle() << std::endl;

        TJ::breakSection();

        // Loop through all options and display them
        for (auto& option : this->options) {
                std::cout << "[" << option.first << "] " << option.second->getDescription() <<
std::endl;
        }

        TJ::breakSection('=');

        // Get the users choise
        std::string choice;

        bool valid = false;
        while (!valid) {
                std::cin >> choice;

                while (std::cin.fail()) {
                        std::cin.clear(); // Reset the Cin flags
                        std::cin.ignore(100, '\n'); // Clear the buffer
                        std::cout << "Invalid input." << std::endl;
                        std::cin >> choice;
                }

                if (this->options.find(choice) != this->options.end())
                        valid = true;
                else
                        std::cout << "Please choose one of the given options." << std::endl;
        }


        // Preform the option that the user chose
        this->options[choice]->run();
}
```

b. Explain how the code behaves.

"AbstractItem" is the abstract class and "FunctionItem" and "MenuItem" are its children, which means they inherit its content. This means that they all have an "std::string description", the enumerator "ItemType", the function "content()", etc.

Functions with the keyword virtual in front of it are, as the keyword suggests, virtual functions. This means that they can be overridden by classes which inherit this function. This is demonstrated by the function "run()" in "AbstractItem" which is overridden in both "FunctionItem" and "MenuItem", using the override keyword to prevent confusion with the compiler. If they have "function = **0**" behind it, they are pure virtual functions which are required to be overridden in the child class. This makes it also impossible for the base class to be instantiated, which is still possible with non-pure virtual functions.

c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

Inheritance: Inheritance is a process in which something inherits properties from something else. In C++ the most common occurrence of this is where attributes and methods of one class, the parent class, are inherited by a child class. This is denoted by the child class referencing its parent in the definition as followed:

```
class AbstractItem {}
class FunctionItem : public AbstractItem {}
```

Here "AbstractItem" is the parent class and "FunctionItem" is the child class. Also note that the reference to the parent class is preceded by an access specifier, which as discussed above in point 1, defines how the access to the inherited member functions is handeled.

Another advantage of inheritance is how an array of different child classes, derived from one parent can be stored.

```
std::map<std::string, std::unique_ptr<Menu::AbstractItem>> options;
```

The above for instance stores many different items of different objects which are from derived classes of AbstractItem. As they all share functions like "run()" and "content()" they can be used as if they are from the same class. Therefore regardless of the derived class it points to, these functions can be executed, as can be seen below.

```
this->options[choice]->run(); // The "this->" is because it is located inside the class
```

To emplace an instance of the "FunctionItem" or "MenuItem" classes it can be done as followed.

```
MenuItem* item = new MenuItem();
this->options.emplace(name, item);
```

Polymorphism: Polymorphism is a process in which something changes its behaviour depending on the situation it is used in. In C++ it is commonly seen as virtual functions, as described above in point b.

3. Abstract base classes and concrete derived classes
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

Snippet of Menu.h

```
class AbstractItem {
private:
	std::string description; // This contains a description of the content, which the menu class uses to dislay info

public:
		// To make the content type clearer there is an enumerator, this has little to no other impact
	enum class ItemType { fuction = 0, menu = 1 };

	/* Getters and setters */
	void setDescription(std::string description);
	std::string getDescription();

	/* Pure virtual functions */
	virtual ItemType content() = 0; // Function to identify the content type
	virtual void run() = 0; // This executes the centent, ie. a function will be executed or a menu will be shown
};

/**
* This is the derived class from "AbstractItem" that stores functions.
**/
class FunctionItem : public AbstractItem {
private:
	std::function<void(void)> storedFunction;

public:
	ItemType content() override { return ItemType::fuction; }

	void run() override { storedFunction(); }

	void setContent(std::function<void(void)> func) { storedFunction = func; }
};
```

    b. Explain how the code behaves.

“AbstractItem” is the abstract class and “FunctionItem” and “MenuItem” are its children, which means they inherit its content. This means that they all have an “std::string description”, the enumerator “ItemType”, the function “content()”, etc.

Functions with the keyword virtual in front of it are, as the keyword suggests, virtual functions. This means that they can be overridden by classes which inherit this function. This is demonstrated by the function “run()” in “AbstractItem” which is overridden in both “FunctionItem” and “MenuItem”, using the override keyword to prevent confusion with the compiler. If they have “function = **0**” behind it, they are pure virtual functions which are required to be overridden in the child class. This makes it also impossible for the base class to be instantiated, which is still possible with non-pure virtual functions.

c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

Abstract classes: an abstract class is a class that contains at least one pure virtual function. As a pure virtual function has no definition, that the class in itself can't be instantiated. For this we need a concrete derived class.

Concrete derived classes: this is a class with no virtual functions and provides functionality for all the inherited pure virtual functions.

An application can be seen in point a. Here the class "AbstractItem" is has pure virtual functions like "run" and "content". Therefor it is an abstract class. The derived class "FunctionItem" defines all functions and therefor is a concrete derived class. The main advantage of this system is that we ensure that all derived classes define the pure virtual functions and share the name, parameters and return type. Other implications of this can be seen in 2.1 in regards to inheritance and polymorphism.

4. Queue link list
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully working, but had no application in the final project. It is however directly integrable.

```
class Queue
{
public:
	Queue(); // Empty constructor
	Queue(const Queue& queue); // Constructor, to copy a queue
	~Queue(); // Destructor, makes sure all nodes are cleared

	void push(User user); // Push a student to the end of the queue
	User peak(void); // Peak at the first node, without deleting it from the queue
	User pop(void); // Peak and then delete the node


	/**** Getters and setters ****/
	void setHead(Node* head);
	Node* getHead(void);

	void setTail(Node* tail);
	Node* getTail(void);

private:
	Node* head; // The head is the address of the first node
	Node* tail; // This is the address of the last node, used to get O(1)
};
```

```
class Node
{
public:
    Node(); // Constructor
    ~Node(); // Destructor

    /**** Getters and setters ****/
    void setNext(Node* next);
    Node* getNext(void);

    void setUser(User user);
    User getUser(void);


private:
    User user; // The data
    Node* next; // The pointer to the next node, is a null pointer if there is no next node
};
```

```
int main(void)
{
	User user1(“Name1”, 1010, { 3, 8, 1960 }), user2(“Name2”, 1010, { 4, 6, 2040 }), userOut();
	Queue q1;
	q1.push(user1);
	q1.push(user2);
	std::cout << "Peak 1: " << q1.peak().getName() << "Pop  1: " << q1.pop().getName() << "\nPop
2: " << q1.pop().getName();
	return 0;
}
```

```
Output:
Peak 1: Name1
Pop  1: Name1
Pop  2: Name2
```

b. Explain how the code behaves.
c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

I have combined points b and c, as they are too closely related in this case.

A queue link list is a way to store data in a FIFO system, meaning that what goes in first comes out first. This is also the behaviour we see demonstrated in the main function. There users are added to the queue, which is usually called pushing an object/item. This user is then processed by the "Queue" class and stored in a "Node" object. These nodes are chained by referencing each other and the links are maintained by the queue class. The main things the queue class does is keep track of the first node of the list and push or pop data to the list. Popping data refers to the act of removing the first element, also known as a node, of the list and returning it. Some lists support peaking, which will return the first element of the list without removing it from the list.

This specific queue linked list also keeps track of the last node. Although this adds an extra variable, it gives O(1) for everything except the destructor, meaning that the time it takes to preform any operation is constant and independent of the amount of data that tis stored. If the class didn't keep track of it, every time a new item were to be pushed, the class would have to look what the first item is and then follow the entire chain till it finds the last node, after which it can store the data. This would give O(n), in other words the execution time is directly proportional to the amount of data stored. In this case it wouldn't be the end of the world, but in large datasets or systems that are time critical it can pose a problematic bottleneck.

Analysing the output of the code above it is clear that the pushing and popping act as intended. "user1" goes in first, followed by "user2". According to the FIFO principle, this means that "user1", which was the first in, must be the first to go out. The peak function shows the first element without removing it. This is indeed the case as the output shows "Name1" as the name of the returned user. Then the pop instruction is used, this does the same but removes the first node, which is the case as the next pop returns "user2", resulting in "Name2" being shown.

5. Operator Overloading, as a member function
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

Snippet of Rank.h

```
class Rank {
private:
	std::vector<RankEntry> rankLog; // Contains your rank, which consists of all exams and ranks you ever had (see VKA regulations for details how this works in practice)
public:
    .
    .
    .
	/* Operator overloading */
	Rank operator+(const Rank& rank);
    .
    .
    .
};
```

Snippet of Rank.cpp

```
Rank Rank::operator+(const Rank& rank) {
	Rank tempRank;
	std::vector<RankEntry> tempRankLog;

	tempRankLog.insert(tempRankLog.end(), rank.rankLog.begin(), rank.rankLog.end());
	tempRankLog.insert(tempRankLog.end(), this->rankLog.begin(), this->rankLog.end());

	tempRank.setRankLog(tempRankLog);

	return tempRank;
}
```

    b. Explain how the code behaves.

The '+' operator is overloaded in this case. When you use the '+' operation, it would act as if you were to call the overloader as a function. In other words "rank1 + rank2" is the same as "rank1.operator+(rank2)". So in this case when adding the two "Rank" objects, it would take the "rank2" as the parameter, which can be accessed by its name. And "rank1" is can be accessed using the "this" pointer. The return type can be any type, but it is also of the "Rank" type in this case.

    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

Operator overloading changes and/or defines the behaviour of operators with the specified classes, making it a form of polymorphism. An example of this would be when you want to add two classes which don't have behaviour defined for the operation. In this case you would overload the '+' operator, by defining the "operator+()" function in the class. In the code shown in "a." it is a member function as it is completely defined within the "Rank" class. The alternative definition is shown in the next

6. Operator Overloading, as a non-member function
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

Snippet of Rank.h

```
class Rank {
private:
	std::vector<RankEntry> rankLog; // Contains your rank, which consists of all exams and ranks you ever had (see VKA regulations for details how this works in practice)
public:
    .
    .
    .
	/* Operator overloading */
	friend std::ostream& operator<<(std::ostream& os, const Rank& rank);
};

std::ostream& operator<<(std::ostream& os, const Rank& rank);
```

Snippet of Rank.cpp

```
std::ostream& operator<<(std::ostream& os, const Rank& rank) {
	os << Rank::toString(rank.rankLog.at(rank.rankLog.size()-1).rank);
	return os;
}
```

    b. Explain how the code behaves.
    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

I have combined points b and c, as they are too closely related in this case.

In this case the object on the left hand side of the operator, which would become the “this” pointer, is not accessible for us to change. In order to overcome that, it is possible to write the operator overloader for “rank1 + rank2” as “operator+(rank1, rank2)”. This makes it a function outside the class, which allows us to define it and then include it in the class as a “friend” function, which means that the function can also access private members, which makes it act extremely similar to a member function, while still being defined outside the function. For the code in “a.” it is the ‘<<’ operator that is overloaded, which is nothing different from the ‘+’ operator, except for the character that is used as operator.

One important note is that in case there is no need for the operator to access private members, it is better to not include the “friend” keyword, which also improves encapsulation.

7. Composition
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

```
class User {
private:
	.
	.
	.
	Rank rank; // The karate rank, in our style this is a set of every rank you ever got, see
Rank.h for details
	.
	.
	.
};
```

    b. Explain how the code behaves.

The "User" class has a member named "rank, which is an instance of the class "Rank". This instance is called an object and can be used to call functions of the "Rank" class. For instance "rank.getRankLog()" would return the rank log which is stored within the object.

    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

Composition describes the relation between two objects, which can be called a "has-a" relation, as in, a "a car has an engine". For a relation to count as composition four criteria have to be met:

- The object (which is "rank" in "a.") is a part of the other object (the class, which is User in "a.")
- It does not know about its existence inside the object.
- It is managed by the object.
- It belongs to only one object at a time.

8. Basic Association
    a. Give the examples code that works and tested, etc. Fully comment the code.

The city example is only for illustration and not for use in the project.

This menu code is fully functional without known bugs. Testing code can be requested from the version control database.

City example:

```
class City
{
private:
   City * neighbouringCity;
};
```

Snippet of Menu.h

```
class Menu {

    .
    .
    .

public:
       class AbstractItem {

         .
         .
         .

       };

       /**
       * This is the derived class from "AbstractItem" that stores menus.
       **/
       class MenuItem : public AbstractItem {
       private:
              Menu* menu;

       public:
              ItemType content() override { return ItemType::menu; }

              void run() override { this->menu->display(); }

              void setContent(Menu* menu) { this->menu = menu; }
       };

    .
    .
    .

};
```

    b. Explain how the code behaves.

For the “City” class, there is a pointer to another city class object. This can be used as any normal pointer, but as it is pointing to its own class there is no extra dependency on an external class.

    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

A basic association or a reflexive association is used when objects in the same class can be associated. In my project I used a variant of this for the menu as can be seen above, where the reflective object was stored in a nested class. This variant isn't too different, but doesn't qualify for being a reflective association. A proper example is the City class in "a.". Here the city is associated with the neighbouring cities, which are of the same class. Because of that it is called a reflexive association.

In my project I used association multiple times, but there was never a need for this type.

9. Qualifier/Qualified Association
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

User.h

```
    .
    .
    .

class User {
private:

        .
        .
        .

      std::vector<Activity> activities; // This stores all mandatory/voluntary karate activities

        .
        .
        .
};
```

Activity.h

```
    .
    .
    .

class Activity {
private:

        .
        .
        .

};
```

    b. Explain how the code behaves.

A user can attend multiple activities, these activities have to be stored. The way this is done, is by pushing each activity the user attends in the std::vector.

    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

If an association has a one to many relation, a qualifier can be used to store the associated class. Which is the “std::vector<>” in the code in “a.”. This is the best way to do so, as it allows dynamic storage of associated objects and prevents the need to store each of them under a separate denominator.

10. Association Class
    a. Give the examples code that works and tested, etc. Fully comment the code.

Example code, not implemented in the project.

```
Class Customer {
  string name;
};

class Product {
  string name;
  float price;
};

class Order {
  Customer * customerPtr;
  Product * productPtr;

  int quantity;
  long referenceNumber;
};
```

    b. Explain how the code behaves.

Instead of having the customer directly interact with the product and store the order inside of if the customer, a separate class for the order is made, which has pointers to reference the customer and the product, while carrying extra members for the order details.

    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

Association classes are used to store information about an association without needing to directly interact with the classes. This approach offers a high amount of encapsulation for the association. In my program there was no real use for this,

11. Dependency
    a. Give the examples code that works and tested, etc. Fully comment the code.

This code is fully functional without known bugs. Testing code can be requested from the version control database.

TJUserFunctions.h

```
#pragma once

#include "User.h"

namespace TJ {
	User createUser();
	void listUsers(std::vector<User> users);
	void deleteUser(std::vector<User>& users);
}
```

TJUserFunctions.cpp

```
User TJ::createUser() {
	User user;

	  .
	  .
	  .

	return user;
}

void TJ::listUsers(std::vector<User> users) {

	.
	.
	.

}
```

    b. Explain how the code behaves.
    c. Explain the topic and how the code covers the topic. Why it is a good example for your project area.

I have combined points b and c, as they are too closely related in this case.

A dependency is a relation between two objects which depend on each other.

In this code the functions “createUser”, “listUsers” and “deleteUser” are dependent on the “User” class. This is not an association or any stronger relation, as the objects are only temporarily used inside the functions.

When the function “createUser” is called, an instance of the “User” class will be temporarily created. After the value is returned, this instance is destructed and then the allocated memory is released again.

12. Detail best thing that you did i.e what was most novel, interesting or challenging piece of code, integration issue, etc that you solved

My favourite part of this code and at the same time also the most interesting and challenging was was the menu system. As I have been annoyed by the messy code that usually came with it, I decided to take the proper OOP approach and designed a streamlined system. Moving away from the usual switch statements, I created a menu class which stores a list of options in a map. While at first this seemed a simple task, this ended up being quite complex as the options in a menu can lead to either a function being executed or a next menu. Since this was a system to optimise the way menus are handled it was in my opinion no good solution to simply allow functions to be added and then have this function create and display a next menu.

To solve this problem I looked into many different options but quickly came to the conclusion that the currently implemented version would be the best option with my current knowledge. In this approach I have a default option item, which is an abstract class with the basic necessities and a virtual function to set the content of the option and to run that content. Then I derived two separate classes from it, one for storing menus and one for functions. Then I could simply store an array of the abstract parent class and have all options in there. Although this ended up not being as easy as I thought.

The first challenge I ran into was looking for a way to store externally defined functions in a class. While I did come across lambda-functions, this was something I had to investigate, as variables had to be passed, but this couldn't be done through the parameters as in most functions, as this would make the functions of a different type. Then I found the option to use the function as if it was inside the actual main script and not running from inside an external class. This was done be writing the lambda-function as followed "[&](){}", where the '&' means that all present variables are being captured by reference, and not just being passed as arguments. While this solved the issue for a while, it proved to be quite limiting, and after looking around I found the "std::function< >" class template, which ended up solving all issues. Now I had the functions stored, I could just run the function by calling the variable in which they are stored. For the menu it was quite simple, as I could just store them by their pointer and then use the general "Menu.display()" function to run them, which is a recursive process, so had to be handled carefully to prevent the system from getting stuck in an infinite loop and potentially even crashing due to a stack overflow.

So in the end, making a menu led my to learning about many interesting topics, of which the most prominent were lambda-functions and the "std::function<>" template class.

The end result in which I can simply call a function to add an option and have a single function open the menu and handle everything worked even better than I could ever imagine and allows clean, readable code which can be understood with very little effort.

CODE:

Main.cpp

```
#include <iostream>

#include "TJ.h" // TJ namespace

#include <string>
#include <vector>
#include <functional>

#include"Menu.h"
#include "User.h"
#include "TJUserFunctions.h"
#include "Attendance.h"

int main() {
	/* Variable declarations*/
	std::vector<User> users;
	std::vector<AttendanceStruct> attendance;

	// Menus
	Menu manageUsersMenu;
	Menu mainMenu;
	Menu attendanceMenu;

	/* Menu setup */
	mainMenu.setTitle("Main menu");
	mainMenu.addChoice("1", "Manage users", &manageUsersMenu);
	mainMenu.addChoice("2", "Attendance", [&]() { getAttendance(attendance, users);
mainMenu.display(); }); // after the attendance function is run, start a new mainMenu
	mainMenu.addChoice("EXIT", "", []() {}); // If nothing happens, no new menu is shown and you
just exit the menu

	manageUsersMenu.setTitle("Manage users");
	manageUsersMenu.addChoice("1", "List all users", [&]() { TJ::listUsers(users);
manageUsersMenu.display(); });
	manageUsersMenu.addChoice("2", "Create a user", [&]() { users.push_back(TJ::createUser());
manageUsersMenu.display(); });
	manageUsersMenu.addChoice("3", "Delete a user", [&]() { TJ::deleteUser(users);
manageUsersMenu.display(); });
	manageUsersMenu.addChoice("CANCEL", "", &mainMenu);
	manageUsersMenu.addChoice("EXIT", "", []() {});

	/* Run the program, just a menu in this case */
	mainMenu.display();

	return 0;
}
```

Activity.h

```
#pragma once

/**
* Author: Tim Jäger
* Email: tim.jager2001@gmail.com
*
* Content:
*	Activity class definition
*
* Dependencies:
*	- iostream
*	- TJ
```

```
*
* Activity is a class that is used to store any karate activities one partakes in, but is general
enough
* to be reused for most activities in general.
* This class keeps track of the following details:
*	- description: this stores what the activity was about
*	- location: this is where the activity took place
*	- date: this is the date on which the activity took place
*	- organizer: this is the person or organization that was responsible for the activity
*
*/

#include <iostream>
#include "TJ.h"

class Activity {
private:
	/* Private variables */
	std::string description;
	std::string location;
	TJ::simpleDate date;
	std::string organizer;

public:
	/* Constructors and destructors */
	Activity();
	Activity(std::string description, std::string location, TJ::simpleDate date, std::string
organizer);
	~Activity();

	/* Getters and setters */
	void setDescription(std::string description);
	std::string getDescription();

	void setLocation(std::string location);
	std::string getLocation();

	void setDate(TJ::simpleDate date);
	void setDate(int day, int month, int year);
	TJ::simpleDate getDate();

	void setOrganizer(std::string organizer);
	std::string getOrganizer();
};
```

Attendance.h

```
#pragma once

/**
* Author: Tim Jäger
* Email: tim.jager2001@gmail.com
*
* Content:
*	- AttendanceStruct (struct)
*	- getAttendance
*
* Dependencies:
*	- iostream
*	- vector
*	- string
*
*	- TJ
```

```
*      - User
*
* AttendanceStruct is a structure that is used to link all the days a user was pressent to
* their UUID.
*
* The getAttendance function is used for the interactive menu, to register the users attendance.
*/

#include <iostream>
#include "TJ.h"

#include <vector>
#include <string>

#include "User.h"

struct AttendanceStruct {
	long userUUID;
	std::vector<TJ::simpleDate> attendanceDays;
};

void getAttendance(std::vector<AttendanceStruct>& attendance, std::vector<User>& users);
```

Menu.h

```
#pragma once

/**
* This class is an interactive menu.
* It stores options (ie. functions or other menus) and can display and run them.
* The menu will look as follow:
*
* ================================================
* Title
* ------------------------------------------------
* [1] Description for 1
* [A] Description for A
* [OTHER] Description for OTHER
* ================================================
* // You can then type here your answer
*
**/

#include <iostream>
#include "TJ.h"

#include <map>
#include <vector>

#include <functional>

class Menu {
public:
	// These classes are defined in Menu to avoid to need to maintain backwards compatibility
whenever the Menu class is updated

	/**
	* All options in the Menu are stored in so "items". These contain all needed details, like
	* its description, a function to execute its content and a function to get its content
	*
	* As items can store different things like other menus and functions there is an abstract item
class
	* and some derived classes which contain the specifics. These derived classes their content
can be identified
	* by calling the "content()" function.
```

```
	**/
	class AbstractItem {
	private:
		std::string description; // This contains a description of the content, which the menu class uses to dislay info

	public:
		// To make the content type clearer there is an enumerator, this has little to no other impact
		enum class ItemType { fuction = 0, menu = 1 };

		/* Getters and setters */
		void setDescription(std::string description);
		std::string getDescription();

		/* Pure virtual functions */
		virtual ItemType content() = 0; // Function to identify the content type
		virtual void run() = 0; // This executes the centent, ie. a function will be executed or a menu will be shown
	};

	/**
	* This is the derived class from "AbstractItem" that stores functions.
	**/
	class FunctionItem : public AbstractItem {
	private:
		std::function<void(void)> storedFunction;

	public:
		ItemType content() override { return ItemType::fuction; }

		void run() override { storedFunction(); }

		void setContent(std::function<void(void)> func) { storedFunction = func; }
	};

	/**
	* This is the derived class from "AbstractItem" that stores menus.
	**/
	class MenuItem : public AbstractItem {
	private:
		Menu* menu;

	public:
		ItemType content() override { return ItemType::menu; }

		void run() override { this->menu->display(); }

		void setContent(Menu* menu) { this->menu = menu; }
	};

private:
	std::string title; // The title of the meny that will be displayed
	std::map<std::string, std::unique_ptr<Menu::AbstractItem>> options; // This stores all options the menu will show

public:

	/* Getters and setters */
	void setTitle(std::string title);
	std::string getTitle();

	/* Other functions */
```

```
        void addChoice(std::string name, std::string description, std::function<void(void)> func); // Function to add a function as option
        void addChoice(std::string name, std::string description, Menu* menu); // Function to add a menu as option

        void display(); // Function to display the meny in the console. It has an interface, making it also handle the option selection and execution.
};
```

Rank.h

```
#pragma once

/**
* Author: Tim Jäger
* Email: tim.jager2001@gmail.com
*
* Content:
*       - RankEnum (enum)
*       - RankEntry (struct)
*       - Rank (class)
*
* Dependencies:
*       - iostream
*       - vector
*
*       - TJ
*
* RankEnum: this is a list of all available karate ranks in the right order.
*
* RankEntry: this is a struct that stores what rank has been obtained when and under whos supervision.
*
* Rank: this is a class that keeps track of all ranks anyone has ever obtained and provides basic
* interaction with the list. It also has a function to convert RankEnum to a human readable format
* and has a stream operator.
*/

#include <iostream>
#include "TJ.h"

#include <vector>


/* Enumerators */
enum class RankEnum {
        noRank = 0, candidateB = 1, candidateA,
        kyu9,
        candidateKyu8, kyu8, advancedKyu8,
        candidateKyu7, kyu7, advancedKyu7,
        candidateKyu6, kyu6, advancedKyu6,
        candidateKyu5, kyu5, advancedKyu5,
        candidateKyu4, kyu4,
        candidateKyu3, kyu3,
        candidateKyu2, kyu2,
        candidateKyu1, kyu1,
        shodan, nidan, sandan, yondan,
        godan, rokudan, nanadan, hachidan,
        kyudan, judan
};

/* Structures */
struct RankEntry {
        RankEnum rank;
        TJ::simpleDate date;
```

```
	std::string examiner;
};


/* Classes */
class Rank {
private:
	std::vector<RankEntry> rankLog;
public:
	Rank();
	~Rank();

	void giveRank(RankEntry rankEntry);
	void giveRank(RankEnum rank, std::string examiner);
	void giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner);

	static std::string toString(RankEnum rank);

	void setRankLog(std::vector<RankEntry> rankLog);
	std::vector<RankEntry> getRankLog();

	/* Operator overloading */
	Rank operator+(const Rank& rank);

	friend std::ostream& operator<<(std::ostream& os, const Rank& rank);
};

std::ostream& operator<<(std::ostream& os, const Rank& rank);
```

TJ.h

```
#ifndef TJ_H    // To prevent multiple instatiations
#define TJ_H

/**
* Author: Tim Jäger
* Email: tim.jager2001@gmail.com
*
* Content:
*	- simpleDate (struct)
*
*	- ignore (function)
*	- clearScreen (function)
*	- breakSection (function)
*
* Dependencies:
*	- iostream
*
* simpleData: this is a struct that doesn't have any fancy things going on, simple a day, month and
year as integers
*
* ignore: this is a function that implements the std::cin.ignore function without having to type it
everytime
*
* clearScrean: this function prints 50 empty lines which virtually clears the screen
*
* breakSection: this is a function with multiple different parameters, which is used to print one or
more lines of the same characters
*/

#include <iostream>

namespace TJ {
	struct simpleDate {
		int day,
```

```
                    month,
                    year;
        };

        void ignore(void);
        void clearScreen(void); // Prints 50 empty lines, which virtually clears the screen
        void breakSection(void); // Print a line of 50 '-' signs
        void breakSection(char sign); // Print a line of 50 times the specified sign
        void breakSection(int lines, char sign); // Print an amount of lines using specific signs
        void breakSection(int lines, int width, char sign); // Print an amount of lines using a
specific sign, with the option to choose the length
}

#endif
```

TJUserFunction.h

```
#pragma once

/**
* Author: Tim Jäger
* Email: tim.jager2001@gmail.com
*
* Content:
*       - createUser (function)
*       - listUsers (function)
*       - deleteUser (function)
*
* Dependencies:
*       - User
*
* createUser: this function contains an interactive menu to create a user and returns it
*
* listUsers: this function creates and displays all users in a vector
*
* deleteUser: this function shows an interactive menu to select and delete a user from
*                       a vector that has been passed
*/

#include "User.h"

namespace TJ {
        User createUser();
        void listUsers(std::vector<User> users);
        void deleteUser(std::vector<User>& users);
}
```

Use.h

```
#pragma once

/**
* Author: Tim Jäger
* E-Mail: tim.jager2001@gmail.com
*
* Description:
*       This file contains a user class, used to store
*       the details of a karate practitioner.
*
* External dependencies:
*       - iostream
*       - ctime
*       - Vector
*       - iomanip
*       - string
*
```

```
* User-defined dependencies:
*       - TJ (namespace)
*       - Activity
*       - Rank
*/

#include <iostream>
#include "TJ.h"

#include <ctime>
#include <vector>
#include <iomanip>
#include <string>
#include "Activity.h"
#include "Rank.h"

class User {
private:
	static long count; // Amount of users that have been initialized
	long UUID; // This is set during the object initialization, hence the set function is private
	std::string name; // This is the users real name, first and last name
	TJ::simpleDate dob; // Date of birth
	Rank rank; // The karate rank, in our style this is a set of every rank you ever got, see
Rank.h for details
	std::vector<Activity> activities; // This stores all mandatory/voluntary karate activities

	/* Private getters and setters */
	void setUUID(long UUID);
public:
	User();
	User(std::string name, long clubID, TJ::simpleDate dob); // New practitioners have no rank, so
this sets an empty rank
	User(std::string name, long clubID, TJ::simpleDate dob, Rank rank, std::vector<Activity>
activities);

	~User(); // This is currently the default destructor

	/* Getters and setters */
	long getUUID(); // Only getter is public for the afore mentioned reason

	void setName(std::string name);
	std::string getName();

	void setDob(TJ::simpleDate dob);
	TJ::simpleDate getDob();

	void setRank(Rank rank);
	void giveRank(RankEntry rankEntry); // giveRank(...) adds a rank to the list
	void giveRank(RankEnum rank, std::string examiner);
	void giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner);
	Rank getRank();

	void setActivities(std::vector<Activity> activities);
	void addActivity(Activity activity); // This adds an activity to the list
	std::vector<Activity> getActivities();
};
```

Activity.cpp

```
#include "Activity.h"

/* Constructors and destructors*/

Activity::Activity() {
	this->description = "Unknown";
```

```
	this->location = "Unknown";
	this->date = { 0, 0, 0 };
	this->organizer = "Unknown";
}

Activity::Activity(std::string description, std::string location, TJ::simpleDate date, std::string
organizer) {
	this->description = description;
	this->location = location;
	this->date = date;
	this->organizer = organizer;
}

Activity::~Activity() {
}

/* Getters and setters */

void Activity::setDescription(std::string description) {
	this->description = description;
}

std::string Activity::getDescription() {
	return this->description;
}

void Activity::setLocation(std::string location) {
	this->location = location;
}

std::string Activity::getLocation() {
	return this->location;
}

void Activity::setDate(TJ::simpleDate date) {
	this->date = date;
}

void Activity::setDate(int day, int month, int year) {
	this->date = { day, month, year };
}

TJ::simpleDate Activity::getDate() {
	return this->date;
}

void Activity::setOrganizer(std::string organizer) {
	this->organizer = organizer;
}

std::string Activity::getOrganizer() {
	return this->organizer;
}
```

Attendance.cpp

```
#include "Attendance.h"

void getAttendance(std::vector<AttendanceStruct>& attendance, std::vector<User>& users) {
	std::vector<User> tmpUsers; // This is used to list all users with that same name that is
selected
	std::string name; // This stores the name of the user that has been chosen
	int choice; // This will store the index in the temporary array of the selected user
```

```
	/* Get the name */
	TJ::clearScreen();
	TJ::breakSection('=');
	std::cout << "Name (or anything to cancel): ";
	TJ::ignore();
	std::getline(std::cin, name);

	/* Store all users with the chosen name in the tmpUsers vector */
	for (User user : users)
		if (user.getName() == name)
			tmpUsers.push_back(user);

	/* Show a list of all users that fit the chosen name and the cancel option */
	TJ::breakSection();

	if (tmpUsers.size() > 0)
		for (int i{ 0 }; i < tmpUsers.size(); i++)
			if (tmpUsers.at(i).getName() == name)
				std::cout << "[" << i << "] " << tmpUsers.at(i).getName() << "\t" <<
((tmpUsers.at(i).getRank().getRankLog().size() > 0) ?
(Rank::toString(tmpUsers.at(i).getRank().getRankLog().back().rank)) : "") << "\t" <<
tmpUsers.at(i).getUUID() << std::endl;
	std::cout << "[-1] CANCEL" << std::endl;

	TJ::breakSection('=');

	/* Get the chosen user from the previously generated list */
	std::cin >> choice;

	while (std::cin.fail() || ((choice < -1 || choice >= tmpUsers.size()) && choice != -1)) {
		std::cin.clear(); // Reset the Cin flags
		std::cin.ignore(100, '\n'); // Clear the buffer
		std::cout << "Invalid input." << std::endl;
		std::cin >> choice;
	}

	/* Get the current date */
	struct tm currentDate;
	time_t now = time(0);
	localtime_s(&currentDate, &now);

	/* Go trough the temporary list and update the attendance list */
	bool match{ false };
	for (int i{ 0 }; i < attendance.size() && choice < tmpUsers.size(); i++) { // Look if the user
has already been registered in the attendance list and add them if they were
		std::cout << "LOOP";
		if (attendance.at(i).userUUID == tmpUsers.at(choice).getUUID()) {
			std::cout << "IF";
			match = true;

			attendance.at(i).attendanceDays.push_back({ currentDate.tm_mday,
currentDate.tm_mon + 1, currentDate.tm_year + 1900 });

			getAttendance(attendance, users);

			std::cout << "next attendance";

		}
	}
	if (match == false && choice != -1) { // If the user didn't find a match and there was no
cancel instruction, add the user to the attendance list
		attendance.push_back({ tmpUsers.at(choice).getUUID(), {{ currentDate.tm_mday,
currentDate.tm_mon + 1, currentDate.tm_year + 1900 }} });
```

```
                getAttendance(attendance, users);
        }
};
```

Menu.cpp

```
#include "Menu.h"

/**
* Display the actual menu in the format:
*
* ================================================
* Title
* ------------------------------------------------
* [1] Description for 1
* [A] Description for A
* [OTHER] Description for OTHER
* ================================================
**/
void Menu::display() {

        /* Clear the screen and display the title */
        TJ::clearScreen();
        TJ::breakSection('=');

        std::cout << this->getTitle() << std::endl;

        TJ::breakSection();

        /* Loop through all options and display them */
        for (auto& option : this->options) {
                std::cout << "[" << option.first << "] " << option.second->getDescription() <<
std::endl;
        }

        TJ::breakSection('=');

        /* Get the users choise */
        std::string choice;

        bool valid = false;
        while (!valid) {
                std::cin >> choice;

                while (std::cin.fail()) {
                        std::cin.clear(); // Reset the Cin flags
                        std::cin.ignore(100, '\n'); // Clear the buffer
                        std::cout << "Invalid input." << std::endl;
                        std::cin >> choice;
                }

                if (this->options.find(choice) != this->options.end())
                        valid = true;
                else
                        std::cout << "Please choose one of the given options." << std::endl;
        }


        // Preform the option that the user chose
        this->options[choice]->run();
}

/* Getters and setters */

void Menu::setTitle(std::string title) {
```

```
	this->title = title;
}

std::string Menu::getTitle() {
	return this->title;
}

void Menu::AbstractItem::setDescription(std::string description) {
	this->description = description;
}

std::string Menu::AbstractItem::getDescription() {
	return description;
}

void Menu::addChoice(std::string name, std::string description, std::function<void(void)> func) {
	FunctionItem* item = new FunctionItem();
	item->setDescription(description);
	item->setContent(func);
	this->options.emplace(name, item);

}

void Menu::addChoice(std::string name, std::string description, Menu* menu) {
	MenuItem* item = new MenuItem();
	item->setDescription(description);
	item->setContent(menu);
	this->options.emplace(name, item);
}
```

Rank.cpp

```
#include "Rank.h"

/* Constructors and destructors */

Rank::Rank() {
	this->rankLog = {};
}

Rank::~Rank() {
}

/* Other functions */
void Rank::giveRank(RankEntry rankEntry) {
	this->rankLog.push_back(rankEntry);
}

void Rank::giveRank(RankEnum rank, std::string examiner) {
	struct tm currentDate;
	time_t now = time(0);
	localtime_s(&currentDate, &now);

	this->giveRank(rank, {currentDate.tm_mday, currentDate.tm_mon + 1, currentDate.tm_year+1900},
examiner);
}

void Rank::giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner) {
	this->rankLog.push_back({ rank, date, examiner });
}

std::string Rank::toString(RankEnum rank) {
	switch (rank) { // This list has to be hardcoded
		case RankEnum::noRank: return "no rank"; break;
		case RankEnum::candidateB: return "candidate B"; break;
```

```
                case RankEnum::candidateA: return "candidate A"; break;
                case RankEnum::kyu9: return "9th kyu"; break;
                case RankEnum::candidateKyu8: return "candidate 8th kyu"; break;
                case RankEnum::kyu8: return "8th kyu"; break;
                case RankEnum::advancedKyu8: return "advanced 8th kyu"; break;
                case RankEnum::candidateKyu7: return "candidate 7th kyu"; break;
                case RankEnum::kyu7: return "7th kyu"; break;
                case RankEnum::advancedKyu7: return "advanced 7th kyu"; break;
                case RankEnum::candidateKyu6: return "candidate 6th kyu"; break;
                case RankEnum::kyu6: return "6th kyu"; break;
                case RankEnum::advancedKyu6: return "advanced 6th kyu"; break;
                case RankEnum::candidateKyu5: return "candidate 5th kyu"; break;
                case RankEnum::kyu5: return "5th kyu"; break;
                case RankEnum::advancedKyu5: return "advanced 5th kyu"; break;
                case RankEnum::candidateKyu4: return "candidate 4th kyu"; break;
                case RankEnum::kyu4: return "4th kyu"; break;
                case RankEnum::candidateKyu3: return "candidate 3rd kyu"; break;
                case RankEnum::kyu3: return "3rd kyu"; break;
                case RankEnum::candidateKyu2: return "candidate 2nd kyu"; break;
                case RankEnum::kyu2: return "2nd kyu"; break;
                case RankEnum::candidateKyu1: return "candidate 1st kyu"; break;
                case RankEnum::kyu1: return "1st kyu"; break;
                case RankEnum::shodan: return "shodan"; break;
                case RankEnum::nidan: return "nidan"; break;
                case RankEnum::sandan: return "sandan"; break;
                case RankEnum::yondan: return "yondan"; break;
                case RankEnum::godan: return "godan"; break;
                case RankEnum::rokudan: return "rokudan"; break;
                case RankEnum::nanadan: return "nanadan"; break;
                case RankEnum::hachidan: return "hachidan"; break;
                case RankEnum::kyudan: return "kyudan"; break;
                case RankEnum::judan: return "judan"; break;
                default: return "No existing rank"; break;
        }
}

/* Getters and setters */

void Rank::setRankLog(std::vector<RankEntry> rankLog) {
        this->rankLog = rankLog;
}

std::vector<RankEntry> Rank::getRankLog() {
        return this->rankLog;
}

/* Operator overloading */

Rank Rank::operator+(const Rank& rank) {
        Rank tempRank;
        std::vector<RankEntry> tempRankLog;

        tempRankLog.insert(tempRankLog.end(), rank.rankLog.begin(), rank.rankLog.end());
        tempRankLog.insert(tempRankLog.end(), this->rankLog.begin(), this->rankLog.end());

        tempRank.setRankLog(tempRankLog);

        return tempRank;
}

std::ostream& operator<<(std::ostream& os, const Rank& rank) {
        os << Rank::toString(rank.rankLog.at(rank.rankLog.size()-1).rank);

        return os;
```

```
}
```

TJ.cpp

```
#include "TJ.h"

namespace TJ {
	void ignore() {
		std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
	}

	void clearScreen(void) {
		std::cout << std::string(50, '\n');
	}

	void breakSection(void) {
		std::cout << std::string(50, '-') << std::endl;
	}

	void breakSection(char sign) {
		breakSection(1, sign);
	}

	void breakSection(int lines, char sign) {
		breakSection(lines, 50, sign);
	}

	void breakSection(int lines, int width, char sign) {
		for (int i{ 0 }; i < lines; i++)
			std::cout << std::string(width, sign) << std::endl;
	}
}
```

TJUserFunctions.cpp

```
#include "TJUserFunctions.h"

User TJ::createUser() {
	User user;
	std::string tmpStr;

	/* Get and set the users name */
	TJ::breakSection('=');
	std::cout << "Name: ";
	TJ::ignore();
	std::getline(std::cin, tmpStr);
	user.setName(tmpStr);

	/* Get and set the date of birth of the user */
	int tmpDay;
	std::cout << "Date of birth:\nDD: ";
	std::cin >> tmpDay;

	while (std::cin.fail()) {
		std::cin.clear(); // Reset the Cin flags
		std::cin.ignore(100, '\n'); // Clear the buffer
		std::cout << "Invalid input." << std::endl;
		std::cin >> tmpDay;
	}

	int tmpMonth;
	std::cout << "MM: ";
	std::cin >> tmpMonth;

	while (std::cin.fail()) {
		std::cin.clear(); // Reset the Cin flags
```

```
		std::cin.ignore(100, '\n'); // Clear the buffer
		std::cout << "Invalid input." << std::endl;
		std::cin >> tmpMonth;
	}

	int tmpYear;
	std::cout << "YYYY: ";
	std::cin >> tmpYear;

	while (std::cin.fail()) {
		std::cin.clear(); // Reset the Cin flags
		std::cin.ignore(100, '\n'); // Clear the buffer
		std::cout << "Invalid input." << std::endl;
		std::cin >> tmpYear;
	}

	user.setDob({ tmpDay, tmpMonth, tmpYear });

	// The user can be returned as is from this point
	return user;
}

void TJ::listUsers(std::vector<User> users) {
	TJ::clearScreen();

	std::cout << std::setw(26) << "Name" << std::setw(19) << "Current rank" << std::setw(5) <<
"UUID" << std::endl;

	TJ::breakSection();

	for (User user : users) {
		std::cout << std::setw(26) << user.getName() << std::setw(19) <<
((user.getRank().getRankLog().size() > 0) ? (Rank::toString(user.getRank().getRankLog().back().rank))
: "") << std::setw(5) << user.getUUID() << std::endl;
	}

	TJ::breakSection('=');

	/* Wait till the user is done reading */
	std::cout << "Press enter to continue...";
	std::cin.ignore(100, '\n');
	std::getchar();
}

void TJ::deleteUser(std::vector<User>& users) {

	/* Get the users UUID */
	TJ::clearScreen();
	TJ::breakSection('=');
	std::cout << "User UUID: ";

	long UUID;
	std::cin >> UUID;

	while (std::cin.fail()) {
		std::cin.clear(); // Reset the Cin flags
		std::cin.ignore(100, '\n'); // Clear the buffer
		std::cout << "Invalid input." << std::endl;
		std::cin >> UUID;
	}

	/* List all users that match the UUID, this should at any time only be one but goes through
all users to make sure */
	TJ::breakSection();
```

```
	std::cout << "Are you sure you want to delete this user:" << std::endl;
	std::cout << std::setw(26) << "Name" << std::setw(19) << "Current rank" << std::setw(5) <<
"UUID" << std::endl;

	User user;

	for (User tmpUser : users) {
		if (tmpUser.getUUID() == UUID) {
			std::cout << std::setw(26) << tmpUser.getName() << std::setw(19) <<
((tmpUser.getRank().getRankLog().size() > 0) ?
(Rank::toString(tmpUser.getRank().getRankLog().back().rank)) : "") << std::setw(5) <<
tmpUser.getUUID() << std::endl;
			user = tmpUser;
		}
	}

	/* Get confirmation */
	TJ::breakSection();
	std::cout << "Type \"YES\" if you are sure: ";

	std::string tmpStr;
	TJ::ignore();
	std::getline(std::cin, tmpStr);

	/* If the answer is YES, remove the user */
	std::string yesStr = "YES";
	if (tmpStr == yesStr) {
		for (int i = 0; i < users.size(); i++) {
			if (users.at(i).getUUID() == UUID) {
				users.erase(users.begin() + i);
				std::cout << "User has been removed" << std::endl;
			}
		}
	}

	/* Wait for user input to make sure they can read the deletion confirmation */
	TJ::breakSection('=');

	std::cout << "Press enter to continue..." << std::endl;
	TJ::ignore();
	std::getchar();
}
```

User.cpp

```
#include "User.h"

long User::count = 0; // On launch, initialize the usercount

/* Constructors and destructor */

User::User() {
	User::count++;			// Everytime the constructor is called, this increases, therefore
counting the
	UUID = User::count;  // instances created and ensuring that the UUID stays unique
	this->name = "Unnamed";
	this->dob = { 1,1,1 };
}

User::User(std::string name, long clubID, TJ::simpleDate dob) {
	User::count++;
	this->UUID = User::count;
	this->name = name;
	this->dob = dob;
```

```
	this->rank = Rank();
	this->activities = {};
}

User::User(std::string name, long clubID, TJ::simpleDate dob, Rank rank, std::vector<Activity>
activities) {
	User::count++;
	this->UUID = User::count;
	this->name = name;
	this->dob = dob;
	this->rank = rank;
	this->activities = activities;
}

User::~User() {
}

/* Getters and setters and additive functions */

void User::setUUID(long UUID) {
	this->UUID = UUID;
}

long User::getUUID() {
	return this->UUID;
}

void User::setName(std::string name) {
	this->name = name;
}

std::string User::getName() {
	return this->name;
}

void User::setDob(TJ::simpleDate dob) {
	this->dob = dob;
}

TJ::simpleDate User::getDob() {
	return this->dob;
}

void User::setRank(Rank rank) {
	this->rank = rank;
}

void User::giveRank(RankEntry rankEntry) {
	this->rank.giveRank(rankEntry);
}

void User::giveRank(RankEnum rank, std::string examiner) {
	this->rank.giveRank(rank, examiner);
}

void User::giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner) {

	this->rank.giveRank(rank, date, examiner);
}

Rank User::getRank() {
	return this->rank;
}
```

```
void User::setActivities(std::vector<Activity> activities) {
	this->activities = activities;
}

void User::addActivity(Activity activity) {
	this->activities.push_back(activity);
}

std::vector<Activity> User::getActivities() {
	return activities;
}
```