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: /05/2021

tudublin.ie

Module: PROJ H2066, SWPR H2002

Programmes: TA\_ERAHUM\_B

1. Description of your project application area.
   1. IDK
   2. More
2. A basic class, with access member functions, Constructors, Static member data
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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

User.h

|  |
| --- |
| 1. #pragma once 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. #include <iostream> 23. #include "TJ.h" 24. #include <ctime> 25. #include <vector> 26. #include <iomanip> 27. #include <string> 28. #include "Activity.h" 29. #include "Rank.h" 30. class User { 31. private: 32. static long count; // Amount of users that have been initialized 33. long UUID; // This is set during the object initialization, hence the set function is private 34. std::string name; // This is the users real name, first and last name 35. TJ::simpleDate dob; // Date of birth 36. Rank rank; // The karate rank, in our style this is a set of every rank you ever got, see Rank.h for details 37. std::vector<Activity> activities; // This stores all mandatory/voluntary karate activities 39. /\* Private getters and setters \*/ 40. void setUUID(long UUID); 41. public: 42. User(); 43. User(std::string name, long clubID, TJ::simpleDate dob); // New practitioners have no rank, so this sets an empty rank 44. User(std::string name, long clubID, TJ::simpleDate dob, Rank rank, std::vector<Activity> activities); 45. ~User(); // This is currently the default destructor 47. /\* Getters and setters \*/ 48. long getUUID(); // Only getter is public for the afore mentioned reason 49. void setName(std::string name); 50. std::string getName(); 51. void setDob(TJ::simpleDate dob); 52. TJ::simpleDate getDob(); 53. void setRank(Rank rank); 54. void giveRank(RankEntry rankEntry); // giveRank(...) adds a rank to the list 55. void giveRank(RankEnum rank, std::string examiner); 56. void giveRank(RankEnum rank, TJ::simpleDate date, std::string examiner); 57. Rank getRank(); 58. void setActivities(std::vector<Activity> activities); 59. void addActivity(Activity activity); // This adds an activity to the list 60. std::vector<Activity> getActivities(); 61. }; |

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;  } |

* 1. Explain how the code behaves.
  2. 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.

1. Inheritance and polymorphism.
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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();  } |

* 1. 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.

* 1. 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.

1. Abstract base classes and concrete derived classes
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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; }  }; |

* 1. 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.

* 1. 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.

1. Queue link list
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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

|  |
| --- |
| 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 |

* 1. Explain how the code behaves.
  2. 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.

1. Operator Overloading, as a member function
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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;  } |

* 1. 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.

* 1. 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

1. Operator Overloading, as a non-member function
2. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.

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;  } |

1. Explain how the code behaves.
2. 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.

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

|  |
| --- |
| 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  .  .  .  }; |

* 1. 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.

* 1. 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.

1. Basic Association
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.
   2. Explain how the code behaves.
   3. 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, where the reflective object was stored in a nested class.

1. Qualifier/Qualified Association
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.
   2. Explain how the code behaves.
   3. Explain the topic and how the code covers the topic. Why it is a good example for your project area.
2. Association Class
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.
   2. Explain how the code behaves.
   3. Explain the topic and how the code covers the topic. Why it is a good example for your project area.
3. Dependency
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.
   2. Explain how the code behaves.
   3. Explain the topic and how the code covers the topic. Why it is a good example for your project area.
4. Detail best thing that you did i.e what was most novel, interesting or challenging piece of code, integration issue, etc that you solved
   1. Give all the fully working and tested code and it should be fully commented. State at the top if there are any problems with the code – such as it does not work fully.
   2. Explain how the code behaves.
   3. Explain the topic and how the code covers the topic. Why it is a good example for your project area.