Guide line for "**Mountain Hiking Challenge Registration**" Project

I – Analysis and design

1- Phân tích dữ liệu

Sau khi đọc kỹ và tìm hiểu về ngữ cảnh của bài toán cùng với các yêu cầu có liên quan, lựa chọn và phân loại đối tượng (*danh từ*) cùng các hành vi (*động từ*) có thể có của đối tượng trong chương trình cần thực hiện, ta có

a. Danh từ: **Student**, **Campus**, **Mountain**, **Menu**

b. Động từ: ***Input*** (*get data from keyboard by user*), ***Add***, ***Update***, ***Search***, ***Get Info*** (object Information), ***ShowAll*** (*display all data in the list*), ***Delete***, ***Filter***, ***Statisticalize***, ***Read*** (*get data from file*), ***Save*** (*store data to file*)

2- Lựa chọn và chi tiết hóa

a. Danh từ: Trong số các danh từ đã xác định, có 2 danh từ

i. **Campus**: tạm hiểu là cơ sở đào tạo phục vụ cho việc quản lý sinh viên (*ở mức thấp hơn trường*), mục tiêu này không nằm trong yêu cầu về quản lý dữ liệu của đề bài, mà sử dụng các ký hiệu dựa trên mã sinh viên. Do đó, để đơn giản, ta sử dụng thông tin này dưới dạng hằng số quy ước có trong mã sinh viên (*Loại khỏi class diagram*)

ii. **Menu**: mô tả cho đối tượng phục vụ cho mục đích hiển thị các lựa chọn của chương trình và cho phép tương tác để điều khiển thực thi theo yêu cầu của người dùng. Đối tượng này mang tính optional, nếu bạn muốn triển khai theo MVC pattern, thì có thể xem Menu như là đối tượng thuộc loại View để triển khai trong chương trình. Nếu không áp dụng design pattern, có thể loại bỏ khỏi class diagram khi thiết kế (*Code mục tiêu này trực tiếp trong Main method*)

iii. **Student**: là đối tượng chính phục vụ cho việc quản lý thông tin đăng ký của sinh viên với các thông tin thể hiện đặc điểm cho từng đối tượng độc lập bao gồm các danh từ phụ thuộc: **ID** of the student, **Name** of the student, **Phone**, **Email**, **Mountain Code**, **Tuition Fee**

iv. **Mountain**: Với đối tượng dữ liệu này, xem xét cấu trúc của tập tin *MountainList.csv* được cung cấp sẵn kèm theo đề bài, ta thấy danh sách các ngọn núi trong danh mục có thể lựa chọn để chinh phục mô tả mỗi với nhiều dòng, mỗi dòng là thông tin của 1 ngọn núi với các thuộc tính phụ thuộc như: **Code** of mountain, Name of **mountain**, Name of the **province** containing the mountain, **Description**

b. Động từ: trong danh sách các động từ đã liệt kê, có những động từ có ý nghĩa tương tác với 1 đối tượng dữ liệu đơn lẻ, một số khác mang ý nghĩa tương tác với một tập các đối tượng dữ liệu, như vậy ta có 2 nhóm động từ được phân chia như sau

i. Động từ tác động lên một đối tượng độc lập: ***Input, Get_Info***

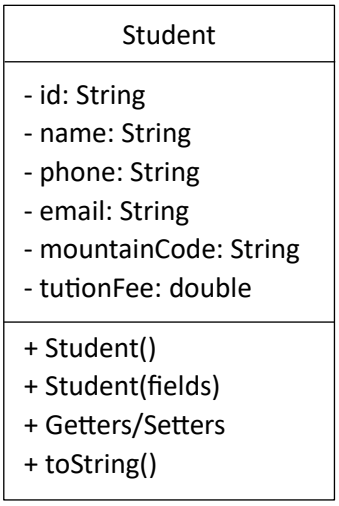
ii. Động từ tác động lên một nhóm đối tượng: ***Add, Update***, ***Search, ShowAll, Delete***, ***Filter***, ***Statisticalize***, ***Read, Save***

3- Thiết kế

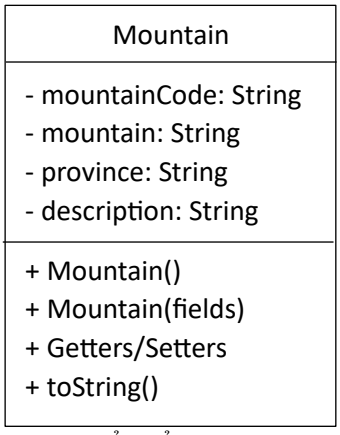
Để cho đơn giản hóa và phù hợp với yêu cầu ***low coupling*** trong thiết kế (*tốt nhất khi đạt tới Atomicity*), ta ***xây dựng class*** dựa trên *danh từ chính*, với các ***thuộc tính*** là những *danh từ phụ thuộc*, cùng các ***phương thức*** là những *động từ* thuộc các đối tượng độc lập.

Mặt khác, để xác định modifier phù hợp, ta dựa trên đặc điểm: ***thuộc tính*** của đối tượng *hiển nhiên* thì ***che giấu*** và mọi ***hành vi nên*** được thực hiện *dựa trên phương thức của đối tượng* tương ứng (*nhằm đảm bảo sự tương tác là linh hoạt, dễ nâng cấp và bảo trì khi có sự thay đổi trong việc hiện thực cùng với việc bảo mật thông tin với bên ngoài đối tượng, chống lại việc truy cập tùy tiện làm ảnh hưởng trạng thái của đối tượng*). Ta có các thiết kế như sau:

a. Student class



b. Mountain class



c. Dựa trên quan hệ "***is A***" – "***has A***" để triển khai business logic class

Vì chương trình trong quá trình hoạt động, cần lưu trữ và thao tác trên nhiều đối tượng cùng loại.

VD: quản lý *nhiều đỉnh núi cần chinh phục*, quản lý *nhiều thông tin đăng ký* của các sinh viên, … Do đó, chương trình cần có một cấu trúc dữ liệu phù hợp để đáp ứng nhu cầu này, các cấu trúc có thể liệt kê như: Mảng, Danh sách liên kết, … là những lựa chọn phù hợp.

Mặt khác, có 2 loại quan hệ có thể sử dụng: "***has A***" và "***is A***", tức là khai báo cấu trúc đã chọn *như một thành phần có sẵn* trong đối tượng

– "***has A***", hoặc định nghĩa *đối tượng chính là cấu trúc* phục vụ cho nhu cầu này

– "***is A***". Để tối ưu hóa bộ nhớ cho quá trình lưu trữ (*cấp phát động khi có nhu cầu, và giảm kích thước bộ nhớ khi dư thừa*), và thuận tiện cho các thao tác trên danh sách, sử dụng quan hệ "***is A***" bằng cách áp dụng kỹ thuật thừa kế, ta chọn **ArrayList** class (*có sẵn trong collection framework*) làm lớp cơ sở (***Super class***) để tạo ra những class mới có khả năng chứa nhiều đối tượng cùng loại (VD: ***Student***, ***Mountain***) với các business logic dựa trên *các động từ tác động lên một nhóm đối tượng* đã phân tích ở trên (*mục số 2*).

Ta có thêm một số class với thiết kế như sau

Mountains
- pathFile: String
+ ***Mountains***() + ***get***(String *mountainCode*): Mountain + ***isValidMountainCode***(String *mountainCode*): boolean + ***dataToObject***(String text): Mountain + ***readFromFile***():void

Students
- pathFile: String - isSaved: boolean
+ ***Students***() + ***isSaved***(): boolean + ***add***(Student *x*):void + ***update***(student *x*): void + ***delete***(String *id*): void + ***searchById***(String *id*): Student + ***searchByName***(String *name*): void + ***showAll***(): void + ***filterByCampusCode***(String: *campusCode*): List<Student> + ***statisticalizeByMountainPeak***():void + ***readFromFile***():void + ***saveToFile***(): void

d. Công cụ phục vụ việc nhập và kiểm tra dữ liệu
Dữ liệu trước khi đưa vào object nên được kiểm tra để đảm bảo tính đúng đắn và hợp lệ theo yêu cầu của project, ta cần xây dựng interface **Acceptable** như thiết kế dưới đây

<<interface>> Acceptable
+ STUDENT_ID : String << final>> + NAME_VALID : String << final>> + DOUBLE_VALID : String <<final>> + INTEGER_VALID : String <<final>> + PHONE_VALID : String <<final>> + VIETTEL_VALID : String <<final>> + VNPT_VALID : String <<final>> + EMAIL_VALID : String <<finall>>
+ ***isValid***(String *data, String pattern*): boolean <<static>>

Mặt khác, đối với các dữ liệu thuộc loại primitive data, nên xây dựng class chứa các phương thức phục vụ cho việc hiển thị thông báo và nhập trực tiếp từ bàn phím, đồng thời kết hợp với **Accepable** nhằm đáp ứng mục đích kiểm tra dữ liệu trước khi sử dụng cho đối tượng của lớp. Nên thiết kế thêm **Inputter** class có class diagram như mô tả sau

Inputter
- ndl: Scanner
+ ***Inputter***() + ***getString***(String *mess*): String + ***getInt***(String *mess*): int + ***getDouble***(String *mess*): double

II – Triển khai kỹ thuật và điều chỉnh thiết kế

1- Validation: Kiểm tra dữ liệu

Về nguyên tắc, dữ liệu khi được đưa vào chương trình, lưu trữ phải đảm bảo tính đúng đắn, hợp lệ, … Vì vậy, sau khi được nhập bởi người sử dụng, dữ liệu cần phải được kiểm tra.

VD: *Số lượng mua phải lớn hơn 0*, *điểm phải thuộc khoảng giá trị từ 0 .. 10*, … Thông thường, để kiểm tra dữ liệu, người lập trình sẽ sử dụng các phép logic, để xác định, hoặc có đôi khi dùng try … catch … (*trong trường hợp kiểm tra dữ liệu có phải là số, ngày tháng, ...*). Những cách này hoàn toàn đúng, tuy nhiên lại tiềm ẩn một số nhược điểm như:

- Mã lệnh với thuật giải phức tạp, rắc rối (*trong những trường hợp xét logic chặt chẽ*)
- Không đem lại hiệu suất thực thi tốt (*khi lạm dụng try ... catch ...*)
- Chỉ có thể kiểm tra những dữ liệu đặc thù (*Số, ngày tháng, ...*)
- ….

a. Regular expression

Nhằm đơn giản hóa và thuận lợi cho việc kiểm tra dữ liệu mà không gia tăng độ phức tạp của thuật toán đối với mã lệnh cho mục tiêu này, người lập trình có thể xây dựng các "***mẫu mô tả dữ liệu***" đối với dạng thức của dữ liệu mong muốn, dựa trên các ký hiệu đặc biệt (*Meta characters*), kỹ thuật này thường được gọi là Regular expression.

Ưu điểm của kỹ thuật này có một số điểm chính như:

- Có thể dùng ở hầu hết các ngôn ngữ và công nghệ phát triển ứng dụng (*JavaScript, C#, Python, Java, ...Web app, Desktop app, Mobile app, ...*)
- Áp dụng cho hầu hết các dạng thức dữ liệu cũng như điều kiện kiểm tra theo cách đơn giản (*thay vì xử lý bằng logic, chỉ cần liệt kê để xét tính hợp lệ*)
- Hỗ trợ tốt cho tính tổng quát hóa của thuật toán
- Có thể dùng cho yêu cầu "**pre-input**" nhằm tăng cường tính dễ dùng cho ứng dụng (*Ngăn chặn dữ liệu nhập sai*)

Để có thể xây dựng được các "***mẫu mô tả dữ liệu***", bạn cần phải nắm vững 3 khái niệm sau

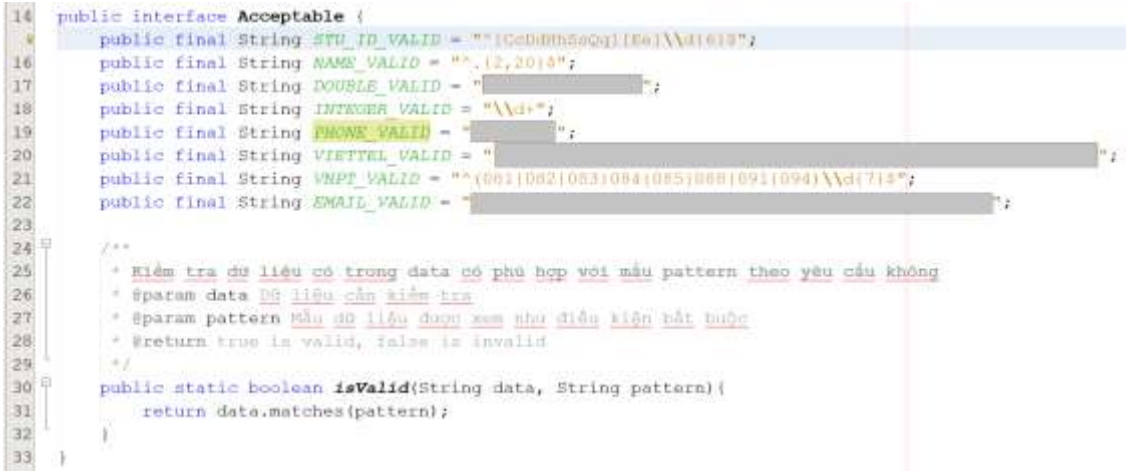
- Regular Expression Patterns
- Metacharacters
- Quantifiers

Thông tin có liên quan đến regular expression có thể *tham khảo thêm ở đây* *https://www.w3schools.com/java/java_regex.asp*)

b. Interface với hằng số và static methods

Như vậy, thay vì phải viết mã lệnh để kiểm tra dữ liệu dựa trên các logic phức tạp, ta chỉ cần mô tả *mẫu dữ liệu* sẽ dùng cho việc kiểm tra, đồng thời kết hợp với phương thức ***matches*** của lớp **String** để kiểm tra chuỗi dữ liệu có *khớp với mẫu* đã thiết lập hay không?! trước khi đưa vào, hoặc chuyển đổi và sử dụng trong chương trình.

Do đó, interface **Acceptable** được triển khai như minh họa dưới đây:



i. Kiểm tra mã sinh viên

Để thiết lập mẫu kiểm tra dữ liệu cho mã sinh viên của FU, phải bắt đầu bởi một trong các ký tự: CDHSQ (có thể là IN HOA hoặc chữ thường), ta sử dụng meta character "^", kết hợp với liệt kê các ký tự cho phép trong cặp dấu "[" và "]". Ta được chuỗi "**^[CcDdHhSsQq]**"; ký tự thứ 2 là "E" hoặc "e" dẫn đến mẫu được mô tả thành "**^[CcDdHhSsQq][Ee]**". Sáu ký tự còn lại là chữ số, ta sử dụng meta character "**\d**" và "**$**" kết hợp với quantifier ta có mẫu hoàn chỉnh như sau: "`^[CcDdHhSsQq][Ee]\\d{6}$`" – (*Dòng 15 trong hình minh họa*)

ii. Kiểm tra tên có tối thiểu 2 ký tự và tối đa 20 ký tự

Tương tự, để kiểm tra tên với yêu cầu: không được để trống, số ký tự tối thiểu: 2 và tối đa 20, ta xây dựng mẫu mô tả dữ liệu cho trường hợp này bằng cách kết hợp với meta character "**.**" với các ký hiệu đã giới thiệu, xác định được mẫu sẽ dùng làm điều kiện như sau: "`^.{2,20}$`" – (*Dòng 16 trong hình minh họa*)

iii. Kiểm tra số điện thoại có thuộc mạng VNPT hay không

Số điện thoại là một chuỗi, có chiều dài 10 ký số, trong đó mỗi nhà mạng sẽ quản lý "***các đầu số***" khác nhau. Để tìm hiểu thông tin này, bạn nên sử dụng hỗ trợ bằng cách google với keyword: "*Các đầu số của mạng di động ở việt nam*" hoặc tham khảo chi tiết ở link sau: https://www.thegioididong.com/hoi-dap/danh-sach-cac-dau-so-moi-cua-cac-nha-mang-vitettel-1263877

Dựa vào danh sách đầu số của mạng VNPT, ta liệt kê các đầu số và kết hợp với quantifier, từ đó thiết lập được điều kiện kiểm tra tương ứng như mẫu sau: `"^(081|082|083|084|085|088|091|094)\\d{7}$"` – (*Dòng 21*)

Hãy tự mình cố gắng tập cách thiết lập các mẫu kiểm tra dữ liệu còn lại, theo yêu cầu của project được giao

iv. static method và hằng số

Vì việc kiểm tra dữ liệu có thể được sử dụng "***nhiều lần***" tại "***nhiều nơi***" khác nhau trong chương trình (*Main class, Add student information, Update, ...*). Do vậy, việc khai báo các mẫu đã xây dựng như là những hằng số trong interface sẽ tăng cường việc "***tái sử dụng***" khi có nhu cầu, đồng thời tạo ra sự "***nhất quán***" trong thuật toán xử lý, và thuận lợi cho việc điều chỉnh sau này. Tương tự, phương thức ***isValid*** được xây dựng dưới dạng **static**, cho phép có thể gọi thi hành trực tiếp thông qua interface **Acceptable** mà không cần tạo đối tượng của lớp. Phương thức này chỉ làm duy nhất một nhiệm vụ là gọi phương thức ***matches***(…) của **data** để kiểm tra dữ liệu chứa trong nó có "*khớp*" với dạng thức của dữ liệu "*mô tả trong mẫu đã thiết lập*" hay không?!

```
24      /**
25       * Kiểm tra dữ liệu có trong data có phù hợp với mẫu pattern theo yêu cầu không
26       * @param data Dữ liệu cần kiểm tra
27       * @param pattern Mẫu dữ liệu được xem như điều kiện bắt buộc
28       * @return true is valid, false is invalid
29       */
30      public static boolean isValid(String data, String pattern){
31          return data.matches(pattern);
32      }
```

2- Inputter: Ứng dụng Coupling – Cohesion

Một nhược điểm khá phổ biến đối với những người mới học lập trình, đó là "*nghĩ sao, viết vậy*", hoặc gặp những phần mã lệnh có thuật toán gần tương đồng thì "*sao chép, rồi sửa điều kiện xử lý*", tất cả những điều này vô tình tạo nên sự "***dư thừa mã lệnh***" trong chương trình và nguy cơ "***phức tạp hóa***" khi chương trình có quy mô lớn với nhiều xử lý phức tạp. Để cải thiện vấn đề này, các thành phần khi thiết kế trong chương trình cần phải đạt được 2 đặc tính "***low coupling – high cohesion***".

- **low coupling**: giúp giảm "*mã dư thừa*", hướng đến sự "*đơn giản, chuyên nghiệp*". Mỗi chức năng hay phương thức chỉ giải quyết duy nhất một vấn đề.
- **high cohesion**: tăng cường tính hữu dụng khi có sự "***gắn kết cao***". Các chức năng hay phương thức có kết gọi hay tương tác một cách thuận lợi tùy theo trình tự sắp xếp khác nhau theo ý muốn của người làm phần mềm

Một số minh họa

a. Nhập và kiểm tra mã sinh viên
Phân tích các bước cần thực hiện
   a.1 – Tạo Scanner phục vụ cho việc nhận dữ liệu từ bàn phím
   a.2 – In thông báo, hướng dẫn nhập tên
   a.3 – Nhận dữ liệu đã nhập vào biến trung gian
   a.4 – Kiểm tra tính hợp lệ của dữ liệu
   a.5 – Thông báo và lặp lại từ bước a.2 nếu dữ liệu không hợp lệ

Mã lệnh của hàm phục vụ viết theo các phân tích trên như sau:

```
111     public String inputName(){
112         String temp = ""; boolean loopMore=true;
113         //-- a.1 - Tạo Scanner phục vụ cho việc nhận dữ liệu từ bàn phím
114         Scanner sc = new Scanner(System.in);
115         do {
116             //-- a.2 - In thông báo, hướng dẫn nhập tên
117             System.out.print("Input name [min:2 - max:20 characters]: ");
118             //-- a.3 - Nhận dữ liệu đã nhập vào biến chuỗi
119             temp = sc.nextLine();
120             //-- a.4 - Kiểm tra tính hợp lệ của dữ liệu
121             if (temp.length()<2 || temp.length()>20){
122                 System.out.println("Name is invalid !. Re-enter ...");
123             }else
124                 loopMore=false;
125             //-- a.5 - Thông báo và lặp lại từ bước a.2 nếu dữ liệu không hợp lệ
126         } while (loopMore);
127         return temp;
128     }
```

Có thể thấy hàm ***inputName***() trong minh họa ở trên có thể giải quyết được yêu cầu khi nhập tên sinh viên. Nhưng nếu xét về mặt kỹ thuật, thì thiết kế này *không tốt*. Vì không đạt được yêu cầu "***low coupling***" đồng thời cũng không đạt được tính "***high cohesion***", cũng như mục tiêu "***reusable***" bởi vì nó chỉ dùng được cho duy nhất mục tiêu nhập và kiểm tra tên của sinh viên. Còn những tình huống tương tự:

- Nhập và kiểm tra mã sinh viên
- Nhập và kiểm tra số điện thoại
- Nhập và kiểm tra email
- …

Thì **không** thể *tái sử dụng* được.

Thông thường, những người mới học lập trình thường chọn giải pháp copy để tạo ra hàm mới. Lúc này số mã lệnh (*10 lệnh*) tương ứng trong hàm ***inputName***() ở trên, lại được sao chép và điều chỉnh điều kiện kiểm tra cho phù hợp với nhu cầu mới. Dẫn đến, số mã lệnh sẽ dư thừa rất nhiều

b. Để cải thiện vấn đề trên, ta cần phân tích tất cả các *mục tiêu cần thực hiện*, tìm kiếm những công việc (*bên trong mỗi mục tiêu*) thường xuyên phải làm (VD: *Nhập chuỗi, Kiểm tra có hợp lệ hay không, ...*). Từ đó xây dựng các phương thức, hàm chỉ giải quyết cho một mục tiêu cụ thể; những phương thức này thường sẽ đơn giản, tính chuyên nghiệp cao, do đó cơ hội được gọi sử dụng lại ở trong những phương thức khác sẽ khả thi hơn. Hãy quan sát mã nguồn của lớp **Inputter** như sau:

```
12  /**
13   * Class phục vụ cho việc nhập và kiểm tra dữ liệu từ bàn phím
14   * @author Huy Nguyễn Mai
15   */
16  public class Inputter {
        private Scanner ndl;
18      /** Default constructor ...3 lines */
21      public Inputter(){
22          this.ndl = new Scanner(System.in);
23      }
24      /** Nhập dữ liệu chuỗi trực tiếp từ bàn phím bởi người sử dụng ...5 lines */
29      public String getString(String mess){
30          System.out.print(mess);
31          return ndl.nextLine();
32      }
33      /** Nhập dữ liệu là số nguyên từ bàn phím bởi người sử dụng ...5 lines */
38      public int getInt(String mess){
39          int result = 0;
40          String temp = getString(mess);
41          if (Acceptable.isValid(temp, Acceptable.INTEGER_VALID))
42              result = Integer.parseInt(temp);
43          return result;
44      }
45      /** Nhập dữ liệu là số double từ bàn phím bởi người sử dụng ...5 lines */
50      public double getDouble(String mess){...7 lines }
57      /** Phương thức cho phép nhập dữ liệu và yêu cầu nhập lại nếu dữ liệu không đúng
64      public String inputAndLoop(String mess, String pattern){
65          String result = "";
            boolean more = true;
67          do {
68              result = getString(mess);
69              more = !Acceptable.isValid(result, pattern);
70              if (more ) System.out.println("Data is invalid !. Re-enter ...");
71          } while (more);
72          return result.trim();
73      }
74      /** Phương thức cho phép nhập dữ liệu của đối tượng ...5 lines */
79      public Student enterStudentInfo(boolean isUpdate){...29 lines }
108 }
```

* Phương thức ***getString***(String mess) [Dòng 29-32] đạt được "***low coupling***" và "***high cohesion***" vì chỉ thực hiện 1 công việc duy nhất là nhập chuỗi, và phương thức này được gọi, sử dụng lại nhiều lần (*các dòng 40, 68*)

** Tương tự, phương thức ***isValid***(String *data*, String *pattern*) đã xây dựng trước đó trong interface **Acceptable** cũng đạt được "***low coupling***" và "***high cohesion***" vì chỉ thực hiện 1 công việc duy nhất là kiểm tra chuỗi (*tham số* ***data***) theo mẫu (*tham số* ***pattern***), và phương thức này được gọi, sử dụng trong lớp **Inputter** nhiều lần (*các dòng 41, 69*)

*** Nếu thiết kế của các phương thức, hàm trong chương trình đạt được yêu cầu "***low coupling***" và "***high cohesion***" thì khả năng triển khai thuật toán theo hướng ***tổng quát hóa*** sẽ rất khả thi. Hãy quan sát phương thức ***inputAndLoop***(…) trong lớp **Inputter** ở trên (*dòng 64-dòng 73*). Với phương thức này, bạn có thể dùng chung đồng thời cho các công việc

+ Nhập và kiểm tra mã sinh viên

```
inputAndLoop("Student ID: ", Acceptable.STU_ID_VALID)
```

+ Nhập và kiểm tra tên sinh viên

```
inputAndLoop("Student name: ", Acceptable.NAME_VALID)
```

+ Nhập và kiểm tra số điện thoại

```
inputAndLoop("Phone number [10 digits]: ", Acceptable.PHONE_VALID)
```

+ Nhập và kiểm tra địa chỉ email

```
inputAndLoop("Email address: ", Acceptable.EMAIL_VALID)
```

3- Overloading method

Một vấn đề khác trong triển khai mã nguồn, đó là luôn phải kiểm tra và hoàn thiện thiết kế của mình. Xem xét tình huống cụ thể đối với phương thức ***showAll***() của lớp **Students** trong phần ***thiết kế*** (3.c); phương thức này, ban đầu được thiết kế để hiển thị tất cả các thông tin đã đăng ký của sinh viên có trong danh sách.

* Các công việc của hàm này được mô tả như sau:

- In tiêu đề của danh sách (**table_header**)

- Lặp trên tập dữ liệu và in ra thông tin của từng đối tượng (*tương ứng với dòng*)

- In dòng kết thúc (**table_footer**)

** Mã lệnh triển khai như sau

```
122     /**
123      * Hiển thị tất cả thông tin đã đăng ký của sinh viên trong hệ thống
124      */
125     public void showAll(){
126         System.out.println(HEADER_TABLE);
            for (Student i : this)
128             System.out.println(i);
129         System.out.println(FOOTER_TABLE);
130     }
```

*** Tuy nhiên, khi thực hiện các chức năng theo yêu cầu của project, ta thấy các công việc kể trên có xu hướng lặp lại nhiều lần tại một số mục trong menu chính:

+ 3. Display Registered List

+ 6. Filter Data by Campus

**** Áp dụng kỹ thuật Overloading đã học trong PRO192, xây dựng phương thức showAll như mô tả dưới đây:

```
122     /**
123      * Hiển thị tất cả thông tin đã đăng ký của sinh viên trong hệ thống
124      */
125     public void showAll(){
126         showAll(this);
127     }
128     /**
129      * Hiển thị thông tin của tất cả các sinh viên đã đăng ký trong danh sách
130      */
131     public void showAll(List<Student> l){
132         System.out.println(HEADER_TABLE);
            for (Student i : l)
134             System.out.println(i);
135         System.out.println(FOOTER_TABLE);
136     }
```

Lúc này, việc hiển thị danh sách sinh viên đã linh hoạt và hiệu quả hơn (*Ghi chú:* ***rl*** *trong phần minh họa là đối tượng của lớp* ***Students***, ***ndl*** *là đối tượng của lớp* ***Inputter***)

+ Hiển thị toàn bộ danh sách

```
rl.showAll();
```

+ Hiển thị danh sách sinh viên đã lọc theo campus

```
temp = ndl.getString("Enter campus code [HE-CE-DE-SE-QE]: ");
List<Student> list = rl.filterByCampusCode(temp);
rl.showAll(list);
```

4- Định nghĩa dữ liệu, bổ sung thuộc tính để đáp ứng mục tiêu

a. Định nghĩa thêm cấu trúc để hỗ trợ xử lý

Với yêu cầu thống kê dữ liệu dựa trên số liệu sinh viên đã đăng ký tham gia theo yêu cầu của tính năng số "7" trong đề bài, tạm diễn giải như sau:

a.1 – Xác định những ngọn núi, sinh viên đã đăng ký tham gia

a.2 – Dựa trên thông tin đã xác định (a.1), tiến hành: \

a.2.1 – Đếm số người tham gia

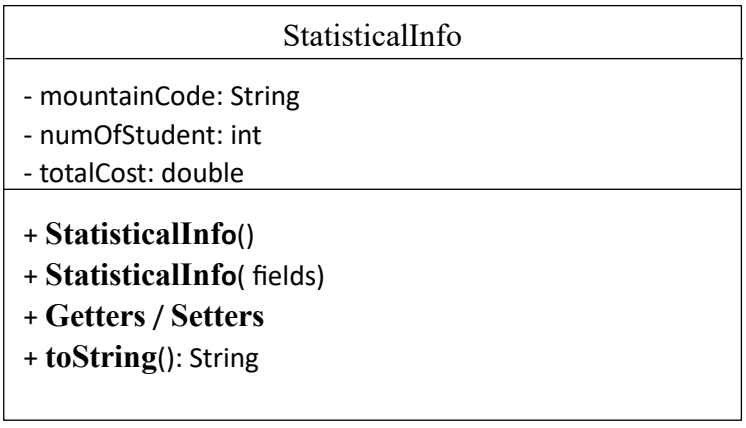
a.2.2 – Cộng dồn để tính tổng số chi phí cần thu

* Có một thách thức khi triển khai mã nguồn theo thuật toán trên, đó là "*các ngọn núi mà sinh viên đăng ký tham gia*", trong quá trình vận hành chương trình hoàn toàn ***không*** được ***biết trước***. Do đó, việc xét logic để thực hiện các tính toán ở bước a.2 sẽ bị ảnh hưởng, dẫn đến không dễ triển khai mã nguồn cho mục đích này.

** Nếu có một cấu trúc "*đóng gói các thông tin có liên quan*" đồng thời có thể "*phát sinh động*" trong quá trình thực thi thuật toán thì vấn đề sẽ được giải quyết. Để giải quyết vấn đề trên, ta ứng dụng tính "***encapsulation***" đã học trong PRO192 để tạo class phục vụ cho mục đích này, đồng thời, trong quá trình duyệt danh sách chứa thông tin đăng ký của sinh viên, đối tượng tương ứng sẽ được tạo ra để phục vụ cho việc thống kê (*lưu trữ điều kiện và cập nhật kết quả*) sẽ đáp ứng được các mong đợi vừa đề cập ở trước đó.

*** Cách làm này cho thấy, nếu "*có một cấu trúc dữ liệu đủ tốt*", thì việc triển khai thuật toán cho bất cứ nhiệm vụ nào trong chương trình đều rất khả thi.

i. StatisticalInfo class



ii. Statistics class

Statistics *extends* **HashMap**<String, StatisticalInfo>
+ **Statistics**() + **Statistics**(List<Student> list) + **show**(): void + **statisticalize**(List<Student> list): List<StatisticalInfo>

iii. Mã nguồn: Statistics class

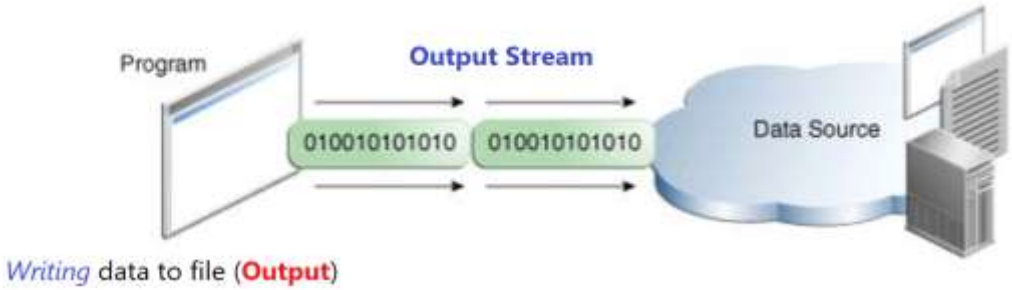
```
/** Lớp phục vụ cho việc thống kê dữ liệu theo yêu cầu của chức năng "7" ...4 li
public class Statistics extends HashMap<String, StatisticalInfo>{

    private final String HEADER_TABLE =
            "|------------------------------------------------------------|\n"+
            "|   Peak Name      | Number of Participants | Total Cost     |\n"+
            "|------------------|------------------------|----------------|";
    private final String FOOTER_TABLE =
            "|------------------------------------------------------------|";
    /** Default constructor ...3 lines */
    public Statistics(){
        super();
    }
    /** Constructor with Student list to update Statistical Information ...4 lir
    public Statistics(List<Student> l){
        super();
        statisticalize(l);
    }
    /** Phương thức thực hiện thống kê dữ liệu dựa trên danh sách ...6 lines */
    public final void statisticalize(List<Student> l){
        for(Student i: l)
            if (this.containsKey(i.getMountainCode())){
                StatisticalInfo x = this.get(i.getMountainCode());
                x.setNumOfRegistration(x.getNumOfRegistration()+1);
                x.setTotalCost(x.getTotalCost()+i.getTutionFee());
            }else{
                StatisticalInfo z = new StatisticalInfo(i.getMountainCode(),
                                                        1, i.getTutionFee());
                this.put(i.getMountainCode(), z);
            }
    }
    /** Hiển thị thông tin thống kê ...3 lines */
    public void show(){
        System.out.println(HEADER_TABLE);
        for (StatisticalInfo i : this.values())
            System.out.println(i);
        System.out.println(FOOTER_TABLE);
    }
}
```

5- Đọc ghi file

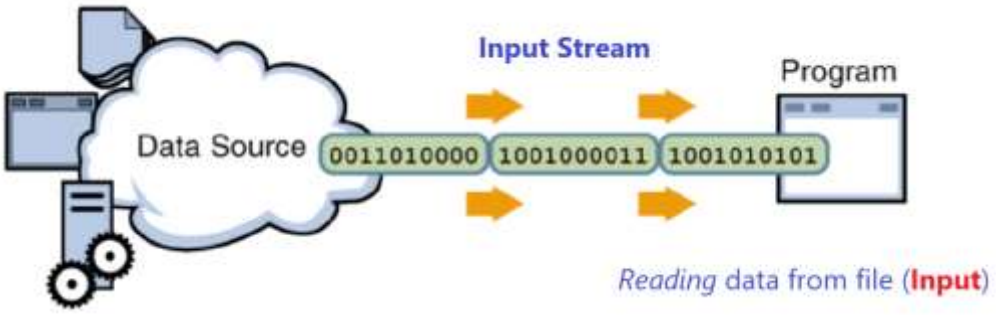
Khi chương trình thi hành, toàn bộ mã lệnh của chương trình sẽ được nạp vào "***vùng bộ nhớ chương trình***" (*program memory*) thuộc vùng nhớ trong (*Internal memory*) của máy tính để thực thi, dữ liệu mà người dùng nhập vào thông qua thiết bị (*input device*) cũng tạm thời được lưu trữ trong vùng nhớ này. Như vậy, nếu *nguồn điện bị mất*, ***dữ***

liệu cũng sẽ không còn !. Để thuận lợi cho quá trình hoạt động, cũng như thuận tiện cho việc sử dụng; dữ liệu cần được lưu vào trong thiết bị lưu trữ (*Bộ nhớ ngoài – External memory*) dưới dạng tập tin (*file object*). Một nguyên lý được phân định rất rõ ràng khi lập trình trong Java, đó là:

+ Với các thao tác nhằm thực hiện để đưa dữ liệu từ "*bộ nhớ trong*" ra "*bộ nhớ ngoài*" thì được gọi là **Output** (*VD: save, write data to file, ...*).



+ Ngược lại, nếu dữ liệu được nạp từ "*bên ngoài*" vào "*bộ nhớ trong*" thì gọi là **Input** (*VD: read data from file, loading, ...*)



Các bước để thực hiện thao tác trên file ở mức trừu tượng: Đọc (*input*) hay Ghi (*output*) rất đơn giản, bao gồm (*Sử dụng các class thuộc* **java.io package**):

- B1. Tạo đối tượng File để ánh xạ lên thiết bị lưu trữ
- B2. Tạo "*luồng*" (*input/output stream*) sẽ tương tác với file
- B3. Tạo đối tượng chịu trách nhiệm vận chuyển dữ liệu (*Buffered hoặc Object*)
- B4. Lặp để tiến hành đọc (*hay ghi*) dữ liệu
- B5. Đóng "*luồng*" và đối tượng sau khi xử lý

Tham khảo thêm ở tài liệu "***File Input – Output***", môn học PRO192

a. **Text file**

Ứng dụng các bước trên cho trường hợp đọc dữ liệu dựa trên các đối tượng: **File**(B1), **FileReader** (B2), **BufferedReader** (B3) ta có mã lệnh của các bước theo

trình tự như sau

```
125         //--- B1. Tạo đối tượng File để ánh xạ lên tập tin MountainList.csv ---
126         File f = new File(this.pathFile);
127         //--- B2. Tạo đối tượng đọc dữ liệu, trỏ tới file đã tạo -------------
128         FileReader fr = new FileReader(f);
129         //--- B3. Tạo Buffer để đọc dữ liệu từ File --------------------------
130         BufferedReader br = new BufferedReader(fr);
131         String temp ="";
132         //--- B4. Lặp khi còn đọc được dữ liệu từ file -----------------------
133         while ((temp = br.readLine())!=null){
134             Mountain i = dataToObject(temp);
135             if (i!=null) this.add(i);
136         }
137         //--- B5. Đóng đối tượng sau khi hoàn thành --------------------------
138         br.close();
139         fr.close();
```

Tuy nhiên, vì quá trình đọc file có thể xảy ra Exception (*VD: Thiết bị lưu trữ bị hư, tập tin không tồn tại trên đĩa, ...*). Do đó, Java sẽ yêu cầu bạn xử lý exception, như vậy mã nguồn cho mục đích đọc file text, trên thực tế sẽ như mô tả sau

```
84      /**
85       * Read Mountain list from csv file
86       */
87      public void readFromFile(){
88          FileReader fr = null;
89          try {
90              //--- B1. Tạo đối tượng File để ánh xạ lên tập tin MountainList.csv-
91              File f = new File(this.pathFile);
92              //--- 1.1 Kiểm tra sự tồn tại của file và thông báo nếu không có ---
93              if (!f.exists()){
94                  System.out.println("MountainList.csv file not found !.");
95                  return ;
96              }
97              //--- B2. Tạo đối tượng đọc dữ liệu, trỏ tới file đã tạo -----------
98              fr = new FileReader(f);
99              //--- B3. Tạo Buffer để đọc dữ liệu từ File ----------------------
100             BufferedReader br = new BufferedReader(fr);
101             String temp ="";
102             //--- B4. Lặp khi còn đọc được dữ liệu từ file --------------------
103             while ((temp = br.readLine())!=null){
104                 Mountain i = dataToObject(temp);
105                 if (i!=null)
106                     this.add(i);
107             }
108             //--- B5. Đóng đối tượng sau khi hoàn thành
109             br.close();
110         } catch (FileNotFoundException ex) {
111             Logger.getLogger(Mountains.class.getName()).log(Level.SEVERE, null, ex);
112         } catch (IOException ex) {
113             Logger.getLogger(Mountains.class.getName()).log(Level.SEVERE, null, ex);
114         } catch (Exception ex) {
115             Logger.getLogger(Mountains.class.getName()).log(Level.SEVERE, null, ex);
116         } finally {
117             try {
118                 fr.close();
119             } catch (IOException ex) {
120                 Logger.getLogger(Mountains.class.getName()).log(Level.SEVERE, null, ex);
121             }
122         }
123     }
```

* <u>Lưu ý:</u> Phương thức ***dataToObject***t(…) tại dòng 104 là phương thức có nhiệm vụ tách các thành phần có trong chuỗi temp đọc được từ file để chuyển thành các thuộc tính tương ứng của **Mountain** object

** Tương tự, thao tác ghi file bạn cần phải tự mình làm lấy. Hãy nhớ, ở B2 đối tượng cần dùng để ghi là **FileWriter**, và ở B3 bộ đệm cần dùng cho ghi dữ liệu là **BufferedWriter** (*có thể tham khảo tài liệu của môn học*).

b. **Object file**

Việc đọc và ghi text file khá đơn giản vì xét về bản chất, dữ liệu văn bản đơn thuần chỉ là tập các ký tự được lưu trữ một cách tuần tự với kích thước không đổi (*1 ký tự tương đương 2 bytes đối với UTF-8*). Nhưng trở ngại lớn nhất khi sử dụng kỹ thuật này để lưu thông tin của các đối tượng phức tạp, có nhiều thuộc tính (*VD: Student, Mountain, ...*) thì phải viết mã lệnh để phân tích dữ liệu chuỗi đã đọc được, sau đó phải chuyển thành "*đối tượng*" tương ứng. Phương thức ***dataToObject***(…) trong phần minh họa ở trên là một ví dụ, và nếu vừa phải đọc, rồi phân tích, kiểm tra, chuyển đổi, … trong khi đó, nếu quá trình đọc và phân tích gặp các vấn đề như "*dữ liệu đặc biệt*", có thể phát sinh lỗi khi chuyển đổi hay kiểm tra … thì thuật toán sẽ không đơn giản, làm tốn kém nhiều thời gian và công sức của người lập trình.

Java cung cấp kỹ thuật cho phép tương tác trực tiếp với tập tin chứa đối tượng, gọi là object file. Trong đó, việc tính toán và chuyển đổi để xác định từng object cùng với thuộc tính tương ứng sẽ do trình biên dịch tự tính toán sao cho phù hợp với cấu trúc đã được định nghĩa bởi class tương ứng. Tuy nhiên, class của loại đối tượng muốn thực hiện đọc, ghi file ở dạng object bắt buộc phải được triển khai bởi interface **Serializable** (*cung cấp sẵn trong **java.lang** package*). Như vậy, theo yêu cầu ghi dữ liệu đăng ký của các sinh viên, theo đề bài thì class **Student** phải được bổ sung thêm khai báo như sau

```
14 public class Student implements Serializable{
```

i. Ghi dữ liệu dạng object file

Ta vẫn áp dụng 5 bước như đã hướng dẫn ở trên, tuy nhiên ở B2, sử dụng đối tượng của lớp **FileOutputStream**, B3 sử dụng đối tượng **ObjectOutputStream** và ở B4, sử dụng phương thức ***writeObject***(…) để ghi xuống. Mã lệnh minh họa như mô tả sau

```
151     /**
152      * Phương thức phục vụ cho việc lưu dữ liệu xuống file ở trên đĩa
153      */
154     public void saveToFile(){
155         //--- 0. Nếu đã lưu rồi thì thôi, không ghi nữa
156         if (this.saved) return ;
157         FileOutputStream fos = null;
158         try {
159             //--- 1. Tạo File object ------------------------------------------------
160             File f = new File(this.pathFile);
161             //--- 2. Tạo FileObjectStream ánh xạ tới File object ---------------------
162             fos = new FileOutputStream(f);
163             //--- 3. Tạo ObjectOutputStream để chuyển dữ liệu xuống thiết bị ---------
164             ObjectOutputStream oos = new ObjectOutputStream(fos);
165             //--- 4. Lặp để ghi dữ liệu ---------------------------------------------
166             for(Student i: this)
167                 oos.writeObject(i);
168             //--- 5. Đóng các object tương ứng sau khi xử lý  -----------------------
169             oos.close();
170             //--- 6. Ghi nhận trạng thái là lưu thành công --------------------------
171             this.saved = true;
172         } catch (FileNotFoundException ex) {
173             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
174         } catch (IOException ex) {
175             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
176         } finally {
177             try {
178                 fos.close();
179             } catch (IOException ex) {
180                 Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
181             }
182         }
183     }
```

ii. Đọc từ object file

Tương tự với tình huống đọc dữ liệu từ Object file, tuy nhiên ở bước 4, để xác định lặp đến khi nào thì ngưng, ta có thể gọi phương thức ***available***(), phương thức này trả về giá trị kiểu nguyên, mô tả số bytes đọc được từ thiết bị lưu trữ. Nếu bằng không tức là hết rồi, không còn gì để đọc nữa. Lưu ý: Để đọc dữ liệu, dùng phương thức ***readObject***() của đối tượng **ObjectInputStream** (*dòng 202*)

```
184     /**
185      * Đọc và nạp dữ liệu từ file Registration.dat vào danh sách sinh viên đã đăng ký
186      */
187     public void readFromFile(){
188         FileInputStream fis = null;
189         try {
190             //--- 1. Tạo File object để ánh xạ lên thiết bị ----------------------------
191             File f = new File(this.pathFile);
192             if (!f.exists()){
193                 System.out.println("registration.dat file not found !.");
194                 return ;
195             }
196             //--- 2. Tạo luồng ánh xạ tới file để đọc dữ liệu từ thiết bị ---------------
197             fis = new FileInputStream(f);
198             //--- 3. Tạo đối tượng mang dữ liệu từ luồng đã tạo ở trên -----------------
199             ObjectInputStream ois = new ObjectInputStream(fis);
200             //--- 4. Lặp và đọc dữ liệu từ file, gắn vào đối tượng hiện hành khi còn dữ liệu
201             while (fis.available()>0){
202                 Student x = (Student) ois.readObject();
203                 this.add(x);
204             }
205             //--- 5. Đóng đối tượng, sau khi đọc xong
206             ois.close();
207             this.saved = true;
208         } catch (FileNotFoundException ex) {
209             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
210         } catch (IOException ex) {
211             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
212         } catch (ClassNotFoundException ex) {
213             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
214         }catch (Exception ex) {
215             Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
216         } finally {
217             try {
218                 fis.close();
219             } catch (IOException ex) {
220                 Logger.getLogger(Students.class.getName()).log(Level.SEVERE, null, ex);
221             }
222         }
223     }
```

6- **Collection:** dùng **"Map – Set"** hay dùng **"List"**
Các thầy cô sẽ guideline phù hợp với cách thức phù hợp với sinh viên

III – Mô hình triển khai mã nguồn

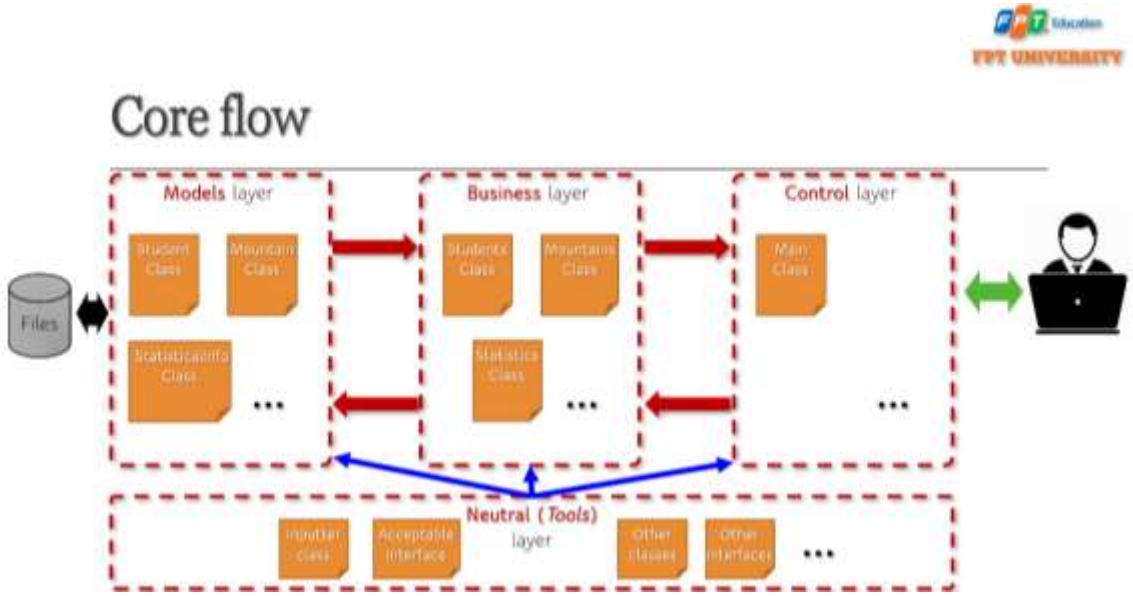
1- nLayers

Một chương trình trong thực tế, có thể có rất nhiều tập tin mã nguồn (*class, interface, ...*). Như vậy, số lượng mã nguồn gia tăng, thì độ phức tạp cũng tăng theo, nhà phát triển phần mềm phải đối mặt với một vấn đề khác: "*Làm sao để quản lý mã nguồn chương trình một cách hiệu quả ?!*". Điều này rất quan trọng, vì để phát triển một phần mềm (*trong thực tế*) thường không phải chỉ do một người mà là một nhóm, có thể có rất nhiều người tham gia. Hơn nữa, sau khi xây dựng chương trình thành công, công tác bảo trì chương trình cũng hết sức quan trọng (*nhà phát triển phải điều chỉnh thuật toán, mở rộng chương trình khi có những thay đổi trong quá trình hoạt động của ứng dụng*)

Để giải quyết vấn đề trên, người ta thường áp dụng kỹ thuật ***breakdown*** kết hợp với tổ chức mã nguồn "*hướng mục tiêu*" ở các ***mức độ*** (*layer*) khác nhau và quản lý chúng dựa vào package. Ví dụ:

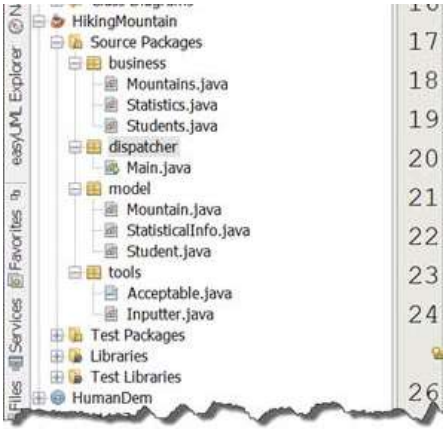
- Những mã nguồn chỉ phục cho việc mô tả dữ liệu chính như: **Student** class, **Mountain** class thì quản lý bởi **model** pakage, gọi là mức thứ nhất: ***model layer***.
- Những mã nguồn chuyên thực hiện các nghiệp vụ (*add, showAll, searchByName, update, delete, filter, ...*) có liên quan đến **model layer**, như Students class, Moutants class sẽ quản lý bởi **business** package, gọi là mức thứ hai: ***business layer***.
- Mã nguồn chính: **Main** class có nhiệm vụ xây dựng giao diện, xử lý tương tác với người dùng đồng thời gọi **business layer** để thực thi yêu cầu sẽ được quản lý bởi **dispatcher** package, tạm gọi là ***control layer***.
- Những mã nguồn còn lại, không tham gia vào quy trình xử lý theo các mức đã thiết kế, nhưng có thể tham gia ở bất kỳ nơi nào trong chương trình để phục vụ cho quá trình xử lý được tổ chức trong **tools** pakage, tạm gọi là ***neutral layer***.

Hãy quan sát sơ đồ thiết kế được dùng cho project có dạng như sau



Trong thực tế, mô hình nLayers vẫn chưa đủ tốt và chuyên nghiệp để áp dụng cho việc phát triển những chương trình lớn, với số lượng chức năng đa dạng, phức tạp có nhiều nhóm (*chuyên gia*) cùng làm việc chung. Tuy nhiên, việc ứng dụng mô hình này để sinh viên tập làm quen như một bước chuyển tiếp, trước khi tiếp cận các mô hình có tính chuyên nghiệp cao hơn, phù hợp hơn ở những môn học tiếp theo là hoàn toàn phù hợp.

Với "**HikingMountain**" project, áp dụng thiết kế trên sẽ có cấu trúc mô tả sau



2- M-V-C design pattern

Đây là một thiết kế rất nổi tiếng và được áp dụng phổ biến trong việc phát triển các ứng dụng "*có quy mô vừa, hoặc lớn*" (VD: *Quản lý đào tạo ở các trường đại học, Sàn giao dịch thương mại điện tử, ...*). Để làm quen với thiết kế này, bạn phải hiểu rõ nLayers, và nên tập làm quen từ project thứ 2 trở đi (*Tổng số project của môn LAB211 là 3*).

Thiết kế MVC phân chia mã nguồn của chương trình cần phát triển thành ba thành phần chính, mỗi thành phần có nhiệm vụ và chức năng riêng, giúp việc phát triển, bảo trì và mở rộng ứng dụng trở nên dễ dàng hơn dựa trên Model-View-Controller, đặc biệt là làm việc theo dạng teamwork.

a. **Model (Mô hình dữ liệu)**:

- Bao gồm các lớp phục vụ cho mục tiêu xử lý dữ liệu của chương trình. Model đại diện cho
  - dữ liệu (*model layer*) và
  - các xử lý nghiệp vụ (*business logic layer*) của ứng dụng.
- Model có trách nhiệm
  - tương tác với "dữ liệu thô", được lưu trữ trên thiết bị (VD: csv file, object file, cơ sở dữ liệu …) và
  - xử lý dữ liệu và trả về kết quả cho người dùng thông qua View hoặc Controller.
- Cần lưu ý: Model không trực tiếp tương tác với giao diện người dùng, nó chỉ tập trung vào việc quản lý và xử lý dữ liệu.

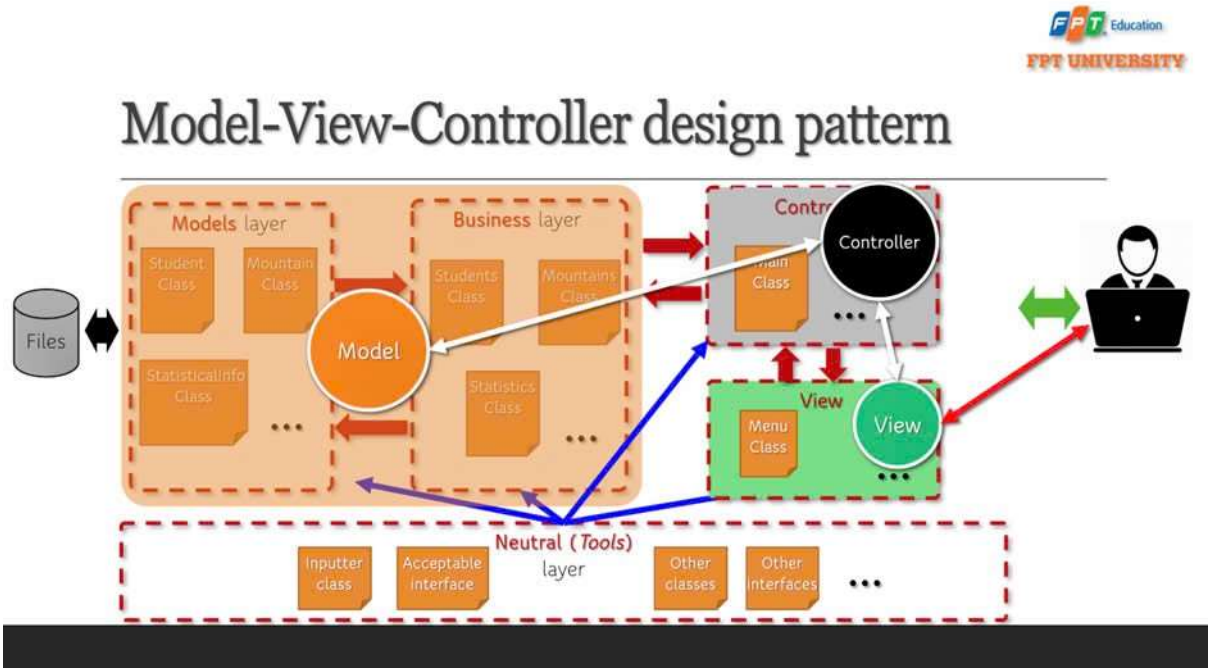
b. **View (Giao diện người dùng)**:

- View chịu trách nhiệm hiển thị dữ liệu (**render**) được cung cấp bởi Model cho người dùng
  - quan sát (kết quả thực thi, màn hình kết quả, Menu, kết quả tìm kiếm)
  - tiếp nhận các tương tác do người dùng thực hiện trên thiết bị. Nó là phần mà người dùng tương tác trực tiếp, VD: Menu cho phép chọn chức năng và hiển thị kết quả trong console application, các trang web, form (*cửa sổ làm việc*), bảng điều khiển, hoặc bất kỳ giao diện đồ họa nào.
- View không chứa bất kỳ logic nghiệp vụ nào, nhiệm vụ
  - nhận dữ liệu từ Model và
  - hiển thị kết quả dưới dạng mà người dùng có thể hiểu và tương tác.

c. **Controller (Điều khiển thực thi)**:

- Controller làm nhiệm vụ điều khiển, dựa trên kết quả
  - tiếp nhận các tương tác (*sự kiện*) từ người dùng (*như nhấn nút, chọn lựa trong menu, gửi biểu mẫu...*)
  - xử lý các yêu cầu và quyết định cách thức phản hồi cho người dùng.
  - Chuyển yêu cầu đến Model
  - Chuyển kết quả xử lý đến View
- Thông thường, controller sẽ yêu cầu Model thực hiện các hành động cần thiết (*như lấy dữ liệu, cập nhật dữ liệu, xử lý, ...*) và sau đó sẽ cung cấp cho View để hiển thị lại kết quả mà người dùng mong đợi.

- Tổng quát hóa hình ảnh áp dụng MVC trong bài như sau



Chúc các bạn vận dụng tốt không những trong môn học này và các môn học tiếp theo.