

Guide line for “Traditional Feast Order Management” 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 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 xác định

- a. Danh từ: **Customer, Set Menu, Order, 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*), **Place a feast order, 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 một số danh từ
 - i. **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*)
 - ii. **Customer**: là đối tượng chính phục vụ cho việc quản lý thông tin của khách hàng khi sử dụng dịch vụ của nhà hàng với các thông tin thể hiện đặc điểm cho từng đối tượng độc lập bao gồm các danh từ phụ thuộc: **ID** of the customer, **Name** of the customer, **Phone, Email** ... là những thông tin liên lạc của từng cá nhân
 - iii. **SetMenu**: Với đối tượng dữ liệu này, xem xét cấu trúc của tập tin *FeastMenu.csv* được cung cấp sẵn kèm theo đề bài, ta thấy danh sách các SetMenu (*Thực đơn bao gồm một số món ăn không bị trùng nhau, thường dùng cho đặt tiệc*) trong danh mục có thể lựa chọn để sử dụng tại ngày mong muốn tổ chức sự kiện (*Sinh nhật, Đám cưới, Họp mặt, ...*), mỗi dòng là thông tin của 1 Set Menu với các thuộc tính phụ thuộc như: **Code** of SetMenu, **Name** of SetMenu, **Price** of SetMenu, **Ingredients** of SetMenu.
 - iv. **Order**: Các đơn hàng do khách hàng đặt trên hệ thống (*yêu cầu cung cấp dịch vụ ăn uống dựa trên sự kiện muốn tổ chức*), với các thông tin chi tiết bao gồm các danh từ phụ thuộc vào danh từ chính nhằm làm rõ thông tin đặt hàng của khách như: **Order ID, ID of Customer, ID of SetMenu, Preferred event date, Number of Tables**,
- b. Động từ: trong danh sách các động từ đã liệt kê, có những động từ có ý nghĩa tương tác với một 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, Place** (*a feast order*)
 - ii. Động từ tác động lên một nhóm đối tượng: **Add, Update, Search, ShowAll, 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. Customer class

Customer
- id: String - name: String - phone: String - email: String
+ Customer() + Customer(fields) + Getters/Setters + toString()

b. SetMenu class

SetMenu
- menuId: String - menuName: String - price: double - ingredients: String
+ SetMenu() + SetMenu(fields) + Getters/Setters + toString()

c. Order class

Order
- orderCode: String - customerId: String - province: String - menuId: String - numTables: int - eventDate: Date
+ Mountain() + Mountain(fields) + Getters/Setters + toString()

- d. 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 SetMenu được cung cấp cho dịch vụ đặt tiệc*, quản lý *nhiều thông tin* của các Khách hàng, quản lý *nhiều thông tin đặt hàng* của khách, ... 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: **Customer**) với business logics 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

Customers
- pathFile: String - isSaved: boolean
+ Customers() + isSaved() : boolean + addNew (Customer x):void + update (Customer x): void + searchById (String id): Customer + filterByName (String name): List<Customer> + showAll () : void + readFromFile () : void + saveToFile () : void

Orders
- pathFile: String - isSaved: boolean
+ Orders() + isSaved() : boolean + isDuplicate (Order x): boolean + addNew (Order x):void + update (Order x): void + searchById (String id): Order + showAll () : void + readFromFile () : void + saveToFile () : void

SetMenus
- pathFile: String
+ SetMenus() + isValidMenuID(String menuId): boolean + dataToObject(String text): SetMenu + readFromFile():void + showMenuList()

- e. 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 <<final>>
+ 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 **Acceptable** 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 để sử dụng trong chương trình.

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

```

8  /**
9   * Phục vụ cho việc validation dữ liệu theo yêu cầu dựa trên regular expression
10  * @author [REDACTED]
11  * https://www.w3schools.com/java/java\_regex.asp
12  */
13  public interface Acceptable {
14      public final String CUS_ID_VALID = "[CcGgKk]\\d{4}$";
15      public final String NAME_VALID = "^. {2,25}$";
16      public final String PHONE_VALID = "^0\\d{9}$";
17      public final String INTEGER_VALID = "[1-9]\\d*";
18      public final String POSITIVE_INT_VALID = "[1-9]\\d*";
19      public final String DOUBLE_VALID = "[0-9]+\\.?[0-9]*";
20      public final String POSITIVE_DOUBLE_VALID = "[0-9]+\\.?[0-9]*";
21      public final String EMAIL_VALID = "[a-zA-Z0-9_+&quot;]+@[a-zA-Z0-9_+&quot;]+\\.([a-zA-Z]{2,4})$";
22  }
23
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  }

```

i. Kiểm tra mã khách hàng

Để thiết lập mẫu kiểm tra dữ liệu cho mã khách hàng, phải bắt đầu bởi một trong các ký tự: CGK (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 “^[CcGgKk]”. Bốn ký tự còn lại là chữ số, ta sử dụng meta character “\d” và “\$” kết hợp với quantifier, ta có mẫu hoàn chỉnh như mô tả sau: “^[CcGgKk]\\d{4}\$” – (Dòng 14 trong hình minh họa)

ii. Kiểm tra tên có tối thiểu 2 ký tự và tối đa 25 ký tự

Tương tự, để kiểm tra tên với yêu cầu: không được để trống, số ký tự tối thiểu: 2 và tối đa 25, ta xây dựng mẫu mô tả dữ liệu cho trường hợp này bằng cách kết hợp meta character “.” (chấp nhận bất cứ ký tự nào có thể gõ từ bàn phím, ngoại trừ dấu new line được tạo bởi enter), kết hợp với quantifier, xây dựng được mẫu sẽ dùng làm điều kiện như sau: “^. {2,25}\$” – (Dòng 15 trong hình minh họa)

iii. Kiểm tra số điện thoại

Số điện thoại là một chuỗi có chiều dài 10 ký số, luôn bắt đầu bởi chữ số “0”, theo sau là 9 ký số bất kỳ. Sử dụng meta characters: “^”, “\$”, kết hợp với quantifier (Có thể dùng hoặc không dùng \$), ta thiết lập được điều kiện kiểm tra tương ứng như mẫu sau: “^0\\d{9}\$” – (Dòng 16)

iv. Kiểm tra số nguyên dương

Số nguyên dương (positive integer) là một số nguyên lớn hơn không, tức là mẫu mô tả chỉ cho phép ký tự đầu tiên là ký số bất kỳ, ngoại trừ ký số “0”, ta lập thành chuỗi điều kiện “^[1-9]” (bao gồm các ký số trong khoảng từ 1 đến 9, hoặc bạn cũng có thể dùng: “^[123456789]”), các ký số còn lại có thể có hoặc không có (quantifier: “*”), ta dùng: “[0-9]*” hoặc “\\d*”. Như vậy, ta có mẫu kiểm tra như sau: “^[1-9]\\d*” – (Dòng 18)

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

v. static method và hằng số

Vì việc kiểm tra dữ liệu có thể được sử dụng “**hiều lần**” tại “**hiều nơi**” khác nhau trong chương trình (*Main class, Add customer 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 xử lý, và “**meaningful**” về mặt ngữ nghĩa thuật toán, cũng như 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- Sinh mã tự động – Unique key generator

Nhu cầu về sinh mã tự động để đại diện cho một đối tượng nào đó trong chương trình và đảm bảo mã được tạo ra, tồn tại trong hệ thống là duy nhất, không bị trùng lặp là một nhu cầu thường gặp. Tùy theo bối cảnh của bài toán mà thuật toán cho mục tiêu này có nhiều phương pháp khác nhau, nhưng thường gặp nhất là hai dạng sau:

- Cách 1: Phát sinh mã mới dựa trên thư viện mã đã tồn tại
- Cách 2: Phát sinh mã mới dựa vào một “**cột mốc**” xác định trong thực tế (*milestone*)

Trong ví dụ minh họa dưới đây, chúng ta sẽ sử dụng cách thứ hai để tạo mã: **dựa trên một “mốc xác định” trong thực tế** – cụ thể là **thời gian**, hay còn gọi là **TimeStamp**.

Như bạn đã biết, một thời điểm trong đời sống hàng ngày thường được xác định bằng các thông số như: **giờ, phút, giây**, kết hợp với **ngày, tháng, năm**. Những mốc thời gian này gần như **không bao giờ lặp lại** trong hàng nghìn năm, nên rất phù hợp để dùng làm cơ sở sinh mã số duy nhất.

Ví dụ: tại thời điểm viết hướng dẫn này là **8 giờ 15 phút 45 giây, ngày 30 tháng 04 năm 2025**. Nếu chuyển thời gian này sang định dạng số theo chuẩn quy ước (ví dụ: “**yyyyMMddhhmmss**”), ta sẽ được chuỗi: **20250430081545**.

Đây là một dãy số gần như **không thể trùng lặp** trong thời gian rất dài, do đó có thể tin cậy để dùng làm mã nhận diện duy nhất trong một hệ thống. Bạn cũng có thể tìm hiểu thêm về khái niệm **Epoch time** – một kỹ thuật phổ biến khác cũng dựa trên thời gian để tạo mã không trùng lặp (Tham khảo thêm: <https://www.epochconverter.com/>).

Đối với yêu cầu của chức năng đặt hàng – “**Place a feast order**”, cần phải phát sinh một mã đơn hàng, mã đơn hàng phải là duy nhất trong hệ thống, ta có thể áp dụng phương pháp trên để giải quyết vấn đề. Thuật toán được mô tả như sau:

- B1 : Đọc thời gian hiện hành từ hệ thống
- B2 : Chuyển đổi thời gian thành dạng thức quy ước (yyyyMMddhhmmss)
- B3 : Tạo mã tự động khi có đối tượng của Order class được tạo

```
private String generateOrderCode() {
    Date now = new Date();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddhhmmss");
    return sdf.format(now);
}
```

Mã nguồn của Order class mô tả cho việc phát sinh order code như minh họa sau

```
13  /**
14   * Lớp mô tả thông tin đơn hàng được đặt bởi khách muốn tổ chức sự kiện
15   * với mã hàng là duy nhất dựa vào Time stamp [yyyyMMddhhmmss]
16   * @author Huy Nguyễn Mai
17   */
18  public class Order implements Serializable{
19      private String orderCode;
20      private String customerId;
21      private String menuId;
22      private int numOfTables;
23      private Date eventDate;
24  }
25  /**
26   * Tạo mã đơn hàng duy nhất dựa vào Time Stamp [epoch time]
27   * @return
28   */
29  private String generateOrderCode() {
30      Date now = new Date();
31      SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddhhmmss");
32      return sdf.format(now);
33  }
34  /** Default constructor ...3 lines */
35  public Order() {
36      this.orderCode = generateOrderCode();
37      this.customerId=""; this.menuId=""; this.eventDate=new Date();
38  }
39  /** Constructor with attributes ...7 lines */
40  public Order(String customerId, String menuId, int numOfTables, Date eventDate) {
41      this.orderCode = generateOrderCode();
42      this.customerId = customerId;
43      this.menuId = menuId;
44      this.numOfTables = numOfTables;
45      this.eventDate = eventDate;
46  }
47  public String getOrderCode() { ...3 lines }
48  public void setOrderCode(String orderCode) { ...3 lines }
49  public String getCustomerId() { ...3 lines }
50  public void setCustomerId(String customerId) { ...2 lines }
```

3- Inputter: Ứng dụng Coupling – Cohesion

Một nhược điểm khá phổ biến đối với những người mới học lập trình, đó là “*ngĩ 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 yêu cầu, 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 }
```

Ta 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**”, cùng với mục tiêu “**reusable**”; bởi vì nó chỉ dùng được cho duy nhất mục tiêu nhập và kiểm tra tên của khách hàng. Còn những tình huống tương tự:

- Nhập và kiểm tra mã khách hàng
- 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 vừa đề cập, 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 dưới đây:

```
18 /**
19  * Class phục vụ cho việc nhập và kiểm tra dữ liệu từ bàn phím
20  * @author [redacted]
21  */
22 public class Inputter {
23     private Scanner ndl;
24     /** Default constructor ...3 lines */
25     public Inputter() {
26         this.ndl = new Scanner(System.in);
27     }
28     /** 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 */
34     public int getInt(String mess){
35         int result = 0;
36         String temp = getString(mess);
37         if (Acceptable.isValid(temp, Acceptable.INTEGER_VALID))
38             result = Integer.parseInt(temp);
39         return result;
40     }
41     /** Nhập dữ liệu là số double từ bàn phím bởi người sử dụng ...5 lines */
42     public double getDouble(String mess){...7 lines }
43 }
```

```

63  /**
64  * Phương thức cho phép nhập và kiểm tra dữ liệu, nhập lại nếu "không khớp"
65  * @param mess Thông báo hướng dẫn nhập dữ liệu
66  * @param pattern Điều kiện kiểm tra dữ liệu
67  * @param isLoop Yêu cầu lặp để nhập dữ liệu cho tới khi đúng
68  * @return String
69  */
70  public String inputAndLoop(String mess, String pattern, boolean isLoop){
71      String result = "";
72      boolean more = true;
73      do {
74          result = getString(mess);
75          more = !Acceptable.isValid(result, pattern);
76          if (more && (isLoop && result.length()>0))
77              System.out.println("Data is invalid !. Re-enter ...");
78      } while (isLoop && more);
79      return result.trim();
80  }
81  /** Phương thức cho phép nhập mới [hoặc cập nhật] dữ liệu của đối tượng Khách
82  * @param isUpdate
83  * @return Customer
84  */
85  public Customer inputCustomerInfo(boolean isUpdate){...13 lines }
86  /** Hỗ trợ cho việc đặt tiệc của khách hàng ...5 lines */
87  public Order placeFeastOrder(boolean isUpdate){...34 lines }
88  }

```

* Phương thức **getString**(String mess) [Dòng 35-38] 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 46, 74)

** 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 47, 75)

*** 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 70-dòng 80). Với phương thức này, bạn có thể dùng chung đồng thời cho các công việc

+ Nhập và kiểm tra mã khách hàng

```
inputAndLoop("Customer ID: ", Acceptable.CUS_ID_VALID);
```

+ Nhập và kiểm tra tên khách hàng

```
inputAndLoop("Customer 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)
```

4- 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 **Customers** trong phần **thiết kế** (3.c); phương thức này, ban đầu được thiết kế để hiển thị thông tin của tất cả các khách hàng có trong danh sách.

* Các công việc cần thực hiện 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

```
55  /**
56   * Phương thức hiển thị thông tin của tất cả các khách hàng có trong hệ thống
57   */
58  @Override
59  public void showAll() {
60      System.out.println(TABLE_HEADER);
61      for(Customer i: this)
62          System.out.println(i);
63      System.out.println(TABLE_FOOTER);
64  }
```

*** 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 tính năng mà chương trình phải có theo requirements của project:

+ 3. Search for customer information by name

+ 8. Display Customer or Order lists.

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

```
55  /**
56   * Phương thức hiển thị thông tin của tất cả các khách hàng có trong hệ thống
57   */
58  @Override
59  public void showAll() {
60      showAll(this);
61  }
62  /**
63   * Hiển thị danh sách khách hàng theo yêu cầu
64   * @param l Chứa danh sách khách hàng mong muốn hiển thị thông tin
65   */
66  public void showAll(List<Customer> l) {
67      System.out.println(TABLE_HEADER);
68      for(Customer i: l)
69          System.out.println(i);
70      System.out.println(TABLE_FOOTER);
71  }
```

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

+ Hiển thị toàn bộ danh sách khách hàng có trong hệ thống

```
cl.showAll();
```

+ Hiển thị danh sách khách hàng bằng cách lọc theo tên (*Hoặc một phần của tên*)

```
temp = ndl.getString("Enter customer name [or part of name]: ");
```

```
List<Student> list = cl.filterByName(temp);
```

```
cl.showAll(list);
```

5- Áp dụng interface cho mục tiêu đảm bảo sự nhất quán giữa thiết kế và triển khai

a. Định nghĩa *interface* **Workable**

Theo như thiết kế ban đầu (xem lại mục 3. Thiết kế), ta thấy các lớp **Customers**, **Orders** đều cần có các phương thức như: *addNew*, *update*, *searchById*, *showAll*, *readFromFile*, *saveToFile*. Nói cách khác, theo như thiết kế ban đầu thì đối tượng của các lớp **Customers** và **Orders** phải “*có đủ các hành vi*” trên để có thể gọi sử dụng trong quá trình lập trình, phục vụ cho “*các nghiệp vụ tương ứng theo yêu cầu của project*”.

Một giả thiết có thể xảy ra trong thực tế, là nếu **Customers** class được triển khai bởi “*Thành viên X*” và **Orders** class được thực hiện bởi “*Thành viên Z*” trong nhóm dự án thì, liệu có đảm bảo cả hai người “*luôn triển khai*” đúng với yêu cầu được đưa ra bởi người thiết kế *hay không* ? (Giả sử thiết kế chương trình do một *thành viên A* trong nhóm thực hiện). Đây chính là vấn đề về *nhất quán từ khâu phân tích – thiết kế chương trình, cho đến triển khai* mã nguồn trong thực tế.

Để giải quyết vấn đề trên, kỹ thuật OOP cho phép định nghĩa *interface* để mô tả các nghiệp vụ (*methods*) mà một class cần phải có, theo cách trừu tượng (*abstract method*) và khi triển khai mã nguồn cho class tương ứng, nhà phát triển cần phải implements *interface* tương ứng, ví dụ đối với lớp **Customers**, ta xây dựng interface *CustomerWorkable* như sau

```
public interface CustomerWorkable {
    void addNew(Customer x);
    void update(Customer x);
    Customer searchById(String id);
    void showAll();
}
```

Tương tự, *OrderWorkable* interface cũng được định nghĩa như mã nguồn dưới đây

```
public interface OrderWorkable {
    void addNew(Order x);
    void update(Order x);
    Order searchById(String id);
    void showAll();
}
```

Khi xây dựng **Customers** class, ngoài việc *extends* từ **ArrayList** để **Customers** object trở thành một **List** collection thì cần phải *implements CustomerWorkable* vừa tạo ở trên, mã nguồn minh họa như sau:

```
public class Customers extends ArrayList<Customer>
    implements CustomerWorkable{
    ...
}
```

b. Kỹ thuật *Generic type*

Việc định nghĩa và triển khai *interface*, *class* như mô tả ở trên có thể dẫn đến một số bất khoản, nhất là đối với sinh viên, những người chưa quen lắm (*hoặc chưa hiểu thấu đáo, cũng như còn ít kinh nghiệm*) trong việc áp dụng OOP để phát triển mã nguồn chương trình. Một số thắc mắc thường thấy như:

- Tại sao phải định nghĩa *interface* với các *abstract methods* như vậy rồi **override** trong *class* sau khi *implements* ?!, *sao không viết luôn trong class* cho gọn !
- Mỗi *class* lại có một *interface* tương ứng, vậy thì mã nguồn hình như phức tạp hơn bình thường thì phải ?!!!
- ...

Xét một hành vi cụ thể của Customers và Orders là: addNew

```
void addNew(Customer x); và  
void addNew(Order x);
```

Nếu xét ở góc độ *trừu tượng hóa* (chung chung, tổng quát) thì là hành vi thêm mới một đối tượng: “giống nhau”, xét ở góc độ *cụ thể hóa* (chi tiết), thì một cái là thêm mới **Customer** và cái còn lại là thêm mới **Order**: “khác nhau”. Vậy các phương thức: `addNew()`, `update()`, `searchById()`, `showAll()`; khi khai báo trong *interface* là những *phương thức trừu tượng* có thể khai báo theo cách để có thể dùng chung cho bất cứ class nào cũng được hay không ?!.

Câu trả lời là có thể, kỹ thuật này được gọi là Generic types (Tham khảo thêm ở đây: <https://docs.oracle.com/javase/tutorial/java/generics/types.html>). Lúc này thay vì phải tạo 2 interface **CustomerWorkable** và **OrderWorkable**, ta chỉ cần 1 interface **Workable** là đủ. Mã nguồn như minh họa sau:

```
6 package business;  
7  
8 /**  
9  * Chứa các mô tả về những hành vi mà lớp tương ứng cần phải triển khai  
10 * theo yêu cầu về thiết kế của chương trình [Áp dụng: Customers, Orders]  
11 * @author [redacted]  
12 * @param <T> Kiểu dữ liệu sẽ sử dụng để triển khai trong tình huống cụ thể  
13 */  
14 public interface Workable <T>{  
15     void addNew(T x);  
16     void update(T x);  
17     T searchById(String id);  
18     void showAll();  
19 }
```

Để *implements* cho **Customers** class, ta chỉ cần thay tham số <T> bằng <Customer> khi muốn chi tiết hóa theo dữ liệu của tình huống cụ thể là được

```
11 /**  
12 * Lớp mô tả thông tin về các khách hàng đăng ký dịch vụ tại nhà hàng  
13 * Thừa kế từ ArrayList collection và triển khai Workable interface  
14 * nhằm đảm bảo cho việc nhất quán trong thiết kế  
15 * @author [redacted]  
16 */  
17 public class Customers extends ArrayList<Customer>  
18     implements Workable<Customer>{  
19  
20     @Override  
21     public void addNew(Customer x) { ...3 lines }  
22  
23     @Override  
24     public void update(Customer x) { ...3 lines }  
25  
26     @Override  
27     public Customer searchById(String id) { ...3 lines }  
28  
29     @Override  
30     public void showAll() { ...3 lines }  
31  
32 }  
33  
34  
35  
36  
37  
38  
39  
40 }
```

Áp dụng cũng tương tự khi *implements* cho **Orders** class

6- Khác biệt khi dùng List và Set

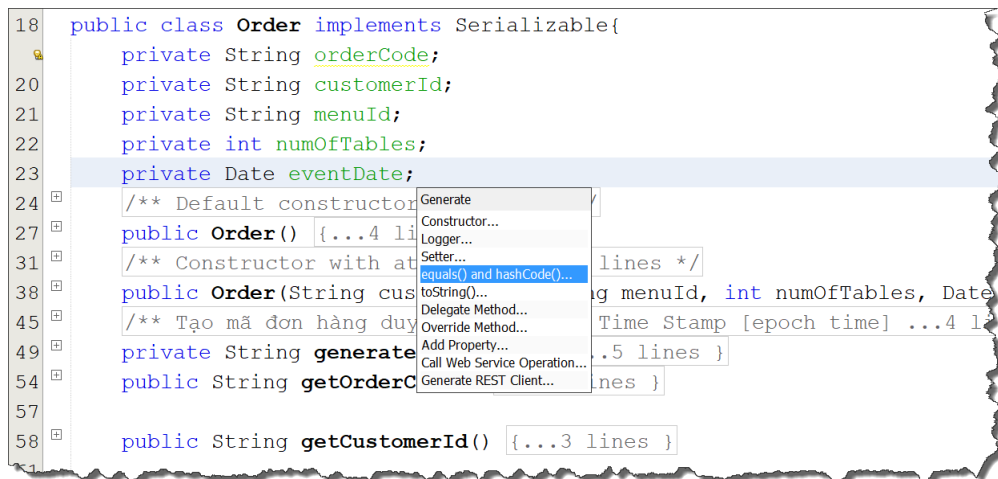
Đối với nghiệp vụ đặt hàng ở chức năng số 6 : “**Place a feast order**”, yêu cầu chỉ cho phép đặt hàng khi tổ hợp 3 thuộc tính **CustomerId**, **SetMenuID** và **eventDay** phải đồng thời khác nhau.

Nếu tạo Orders class thừa kế từ ArrayList class (giống như cách tạo Customers), thì buộc phải viết thêm phần xử lý yêu cầu “đơn hàng có bị trùng hay không?”, việc này sẽ làm phát sinh thêm mã nguồn, rồi trước khi thêm một đơn hàng mới, phải gọi phần xử lý để đảm bảo không bị trùng theo yêu cầu nghiệp vụ của project. Ta thấy trong tình huống này, vấn đề logic dường như lại “*bị phức tạp hóa*” lên một chút.

Để giải quyết vấn đề trên, Java Collection Framework cung cấp một dạng cấu trúc gọi là **Set** collection, thường dùng để chứa một tập các phần tử cùng loại (giống như **List**) nhưng các phần tử chứa trong **Set** là duy nhất. Nói cách khác, **Set** không chứa các đối tượng giống nhau (Đây là sự khác biệt chính so với **List** collection), và để sử dụng **Set**, ta cần định nghĩa lại phương thức **equals()** đối với lớp mô tả dữ liệu cần chứa trong **Set** (trong trường hợp này là **Order** class); sau đó, trước khi thêm phần tử vào set, ta chỉ cần gọi phương thức contains để kiểm tra trùng là đã giải quyết vấn đề một cách đơn giản

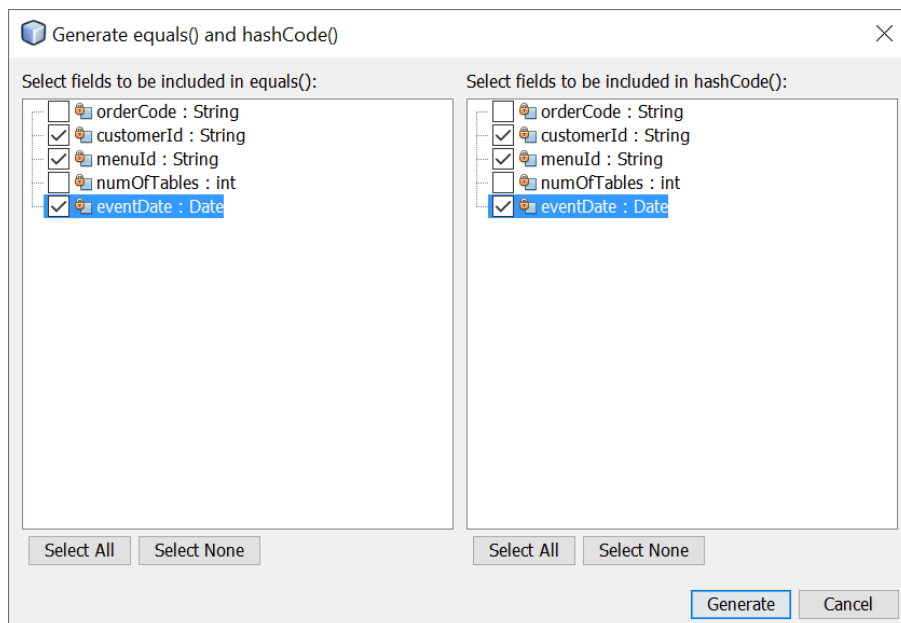
Định nghĩa lại phương thức **equals()** trong **Order** class

- Right mouse -> **Insert code**



```
18 public class Order implements Serializable{
19     private String orderCode;
20     private String customerId;
21     private String menuId;
22     private int numOfTables;
23     private Date eventDate;
24     /** Default constructor
25     public Order() {...4 lines}
26     /** Constructor with attributes
27     public Order(String customerId, String menuId, int numOfTables, Date eventDate) {...5 lines}
28     /** Tạo mã đơn hàng duy nhất
29     private String generateOrderCode() {...4 lines}
30     public String getOrderCode() {...3 lines}
31     public String getCustomerId() {...3 lines}
```

- Chọn **equals()** and **hashCode()** ...



Generate equals() and hashCode()

Select fields to be included in equals():

- ☐ orderCode : String
- ☒ customerId : String
- ☒ menuId : String
- ☒ numOfTables : int
- ☒ eventDate : Date

Select All Select None

Select fields to be included in hashCode():

- ☐ orderCode : String
- ☒ customerId : String
- ☒ menuId : String
- ☒ numOfTables : int
- ☒ eventDate : Date

Select All Select None

Generate Cancel

- Đánh dấu vào những thuộc tính sẽ sử dụng làm điều kiện để xét trùng dữ liệu (Ta chọn **customerId**, **menuId**, **eventDate**). Mã nguồn của **Order** class sẽ phát sinh thêm các phương thức **hashCode()** và **equals()** như sau

```

18 public class Order implements Serializable{
19     private String orderCode;
20     private String customerId;
21     private String menuId;
22     private int numofTables;
23     private Date eventDate;
24
25     @Override
26     public int hashCode() {
27         int hash = 7;
28         hash = 47 * hash + Objects.hashCode(this.customerId);
29         hash = 47 * hash + Objects.hashCode(this.menuId);
30         hash = 47 * hash + Objects.hashCode(this.eventDate);
31         return hash;
32     }
33
34     @Override
35     public boolean equals(Object obj) {
36         if (this == obj) {
37             return true;
38         }
39         if (obj == null) {
40             return false;
41         }
42         if (getClass() != obj.getClass()) {
43             return false;
44         }
45         final Order other = (Order) obj;
46         if (!Objects.equals(this.customerId, other.customerId)) {
47             return false;
48         }
49         if (!Objects.equals(this.menuId, other.menuId)) {
50             return false;
51         }
52         if (!Objects.equals(this.eventDate, other.eventDate)) {
53             return false;
54         }
55         return true;
56     }
57     /** Default constructor ...3 lines */

```

Định nghĩa phương thức **isDuplicate()** [Dòng 57-59] và xét không trùng trước khi thêm một đơn hàng mới [Dòng 65-68] trong **Orders** class

```

26 /**
27  * Lớp mô tả thông tin đơn hàng mà khách hàng đăng ký dịch vụ tại nhà hàng
28  * Thừa kế từ HashSet collection và triển khai Workable interface nhằm
29  * dảm bảo cho việc nhất quán trong thiết kế
30  * @author [REDACTED]
31  */
32 public class Orders extends HashSet<Order> implements Workable<Order>{
33     private final String TABLE_HEADER =
34         "-----\n" +
35         "Order ID| Event date | Customer code| Set Menu| Price    | Tables | Cost    \n" +
36         "-----";
37     private final String TABLE_FOOTER =
38         "-----\n";
39     private final String TABLE_ROW_FORMAT = "%-8s|%-12s|%-14s|%-9s|%-10s|%-8d|%-14s\n";
40     private boolean saved;
41     private String pathFile;
42     /** Default constructor ...3 lines */
43     public Orders(){...6 lines }

```



```

51  /**
52   * Kiểm tra trùng đơn hàng. Nếu 2 đơn hàng có 3 thuộc tính giống nhau là trùng
53   * Customer ID - SetMenu ID - Event date
54   * @param x Order object
55   * @return true : trùng đơn hàng
56   */
57  public boolean isDuplicate(Order x) {
58      return this.contains(x);
59  }
60
61  /**
62   * Thêm một đơn hàng mới vào danh sách đơn hàng hiện có nếu không bị trùng
63   * @param x Đơn hàng mới cần thêm
64   */
65  @Override
66  public void addNew(Order x) {
67      if (!this.isDuplicate(x))
68          this.add(x);
69  }
70  /** Trả về trạng thái của danh sách là đã lưu hay chưa lưu vào file ...4 lines */
71  public boolean isSaved() {...3 lines }
72
73  /** Cập nhật thông tin đơn hàng dựa vào dữ liệu được nhập mới dựa theo ...5 lines */
74

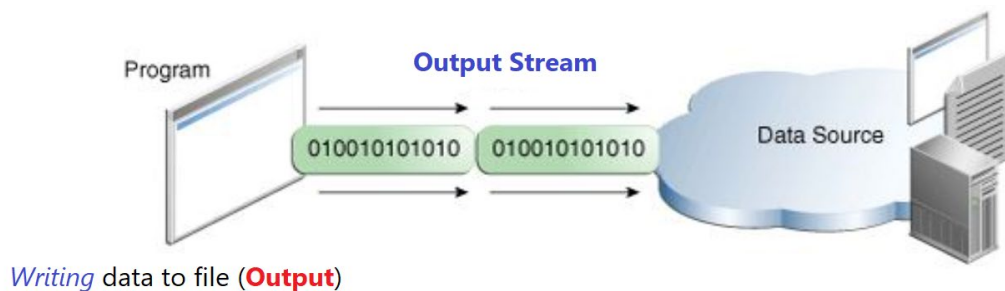
```

Lưu ý: Có thể không cần định nghĩa thêm phương thức **isDuplicate()** mà sử dụng phương thức **contains()** khi xét trùng tại dòng 66 của phương thức **addNew()**

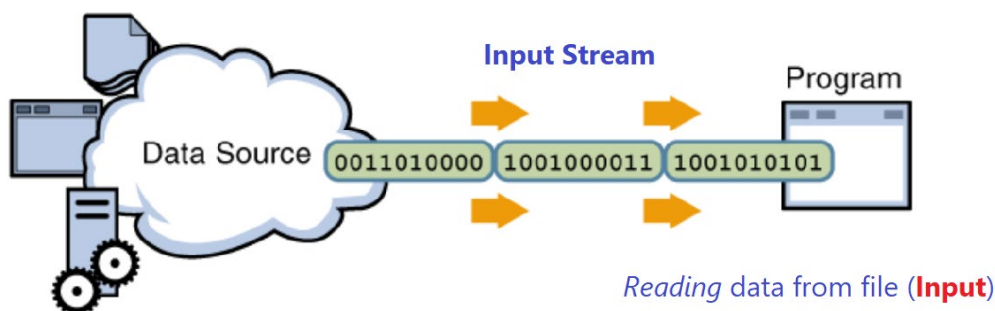
7- Đọc ghi file

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

```
56 //--- B1. Tạo đối tượng File để ánh xạ lên tập tin MountainList.csv --
57 File f = new File(this.pathFile);
58 //--- 1.1 Kiểm tra sự tồn tại của file và thông báo nếu không có ----
59 if (!f.exists()){
60     System.out.println("FeastMenu.csv file not found !.");
61     return ;
62 }
63 //--- B2. Tạo đối tượng đọc dữ liệu, trỏ tới file đã tạo -----
64 FileReader fr = new FileReader(f);
65 //--- B3. Tạo Buffer để đọc dữ liệu từ File -----
66 BufferedReader br = new BufferedReader(fr);
67 String temp = "";
68 //--- B4. Lặp khi còn đọc được dữ liệu từ file -----
69 while ((temp = br.readLine())!=null){
70     SetMenu i = dataToObject(temp);
71     if (i!=null) this.put(i.getMenuId(),i);
72 }
73 //--- B5. Đóng đối tượng sau khi hoàn thành
74 br.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 text file, trên thực tế sẽ như mô tả sau

```
52 /**
53  * Read SetMenu list from FeastMenu.csv file
54  */
55 public void readFromFile() {
56     FileReader fr = null;
57     try {
58         //--- B1. Tạo đối tượng File để ánh xạ lên tập tin MountainList.csv-
59         File f = new File(this.pathFile);
60         //--- 1.1 Kiểm tra sự tồn tại của file và thông báo nếu không có ---
61         if (!f.exists()){
62             System.out.println("FeastMenu.csv file not found !.");
63             return ;
64         } //--- B2. Tạo đối tượng đọc dữ liệu, trỏ tới file đã tạo -----
65         fr = new FileReader(f);
66         //--- B3. Tạo Buffer để đọc dữ liệu từ File -----
67         BufferedReader br = new BufferedReader(fr);
68         String temp = "";
69         //--- B4. Lặp khi còn đọc được dữ liệu từ file -----
70         while ((temp = br.readLine())!=null){
71             SetMenu i = dataToObject(temp);
72             if (i!=null) this.put(i.getMenuId(),i);
73         }
74         //--- B5. Đóng đối tượng sau khi hoàn thành
75         br.close();
76     } catch (FileNotFoundException ex) {
77         Logger.getLogger(SetMenus.class.getName()).log(Level.SEVERE, null, ex);
78     } catch (IOException ex) {
79         Logger.getLogger(SetMenus.class.getName()).log(Level.SEVERE, null, ex);
80     } catch (Exception ex) {
81         Logger.getLogger(SetMenus.class.getName()).log(Level.SEVERE, null, ex);
82     } finally {
83         try {
84             fr.close();
85         } catch (IOException ex) {
86             Logger.getLogger(SetMenus.class.getName()).log(Level.SEVERE, null, ex);
87         }
88     }
89 }
```

* Lưu ý: Phương thức **dataToObject(...)** tại dòng 71 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 **SetMenu** 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: *SetMenu*, *Customer*, ...) 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 của khách hàng, thì class **Customer** phải được bổ sung thêm khai báo như sau

```
14 public class Customer implements Serializable{
```

i. Ghi dữ liệu dạng object file

Ta vẫn áp dụng 5 bước như đã hướng dẫn ở trước đó, 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 sau

```
159 /**
160  * Phương thức phục vụ cho việc lưu dữ liệu xuống file ở trên đĩa
161  */
162 public void saveToFile() {
163     //--- 0. Nếu đã lưu rồi thì thôi, không ghi nữa
164     if (this.saved) return ;
165     FileOutputStream fos = null;
166     try {
167         //--- 1. Tạo File object -----
168         File f = new File(this.pathFile);
169         //--- 2. Tạo FileOutputStream ánh xạ tới File object -----
170         fos = new FileOutputStream(f);
171         //--- 3. Tạo ObjectOutputStream để chuyển dữ liệu xuống thiết bị -----
172         ObjectOutputStream oos = new ObjectOutputStream(fos);
173         //--- 4. Lặp để ghi dữ liệu -----
174         for (Customer i: this)
175             oos.writeObject(i);
176         //--- 5. Đóng các object tương ứng sau khi xử lý -----
177         oos.close();
178         //--- 6. Ghi nhận trạng thái là lưu thành công -----
179         this.saved = true;
180     } catch (FileNotFoundException ex) {
181         Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
182     } catch (IOException ex) {
183         Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
184     } finally {
185         try {
186             fos.close();
187         } catch (IOException ex) {
188             Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
189         }
190     }
191 }
```

ii. Đọc từ object file

Tương tự với tình huống đọc dữ liệu từ Object file, tuy nhiên ở bước 4, để xác định lập 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)

```
119  /**
120   * Đọc và nạp dữ liệu từ file Customers.dat vào danh sách Khách hàng
121   */
122  public final void readFromFile() {
123      FileInputStream fis = null;
124      try {
125          //--- 1. Tạo File object để ánh xạ lên thiết bị -----
126          File f = new File(this.pathFile);
127          if (!f.exists()) {
128              System.out.println("Customers.dat file not found !.");
129              return ;
130          }
131          //--- 2. Tạo luồng ánh xạ tới file để đọc dữ liệu từ thiết bị -----
132          fis = new FileInputStream(f);
133          //--- 3. Tạo đối tượng mang dữ liệu từ luồng đã tạo ở trên -----
134          ObjectInputStream ois = new ObjectInputStream(fis);
135          //--- 4. Lặp và đọc dữ liệu từ file, gán vào đối tượng hiện hành khi còn dữ liệu
136          while (fis.available()>0) {
137              Customer x = (Customer) ois.readObject();
138              this.add(x);
139          }
140          //--- 5. Đóng đối tượng, sau khi đọc xong
141          ois.close();
142          this.saved = true;
143      } catch (FileNotFoundException ex) {
144          Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
145      } catch (IOException ex) {
146          Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
147      } catch (ClassNotFoundException ex) {
148          Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
149      } catch (Exception ex) {
150          Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
151      } finally {
152          try {
153              fis.close();
154          } catch (IOException ex) {
155              Logger.getLogger(Customers.class.getName()).log(Level.SEVERE, null, ex);
156          }
157      }
158  }
```

iii. Generic Methods :: trong triển khai mã nguồn

Trong phần trước, chúng ta đã làm quen với kỹ thuật “*Generic types*” (Mục 5.b) khi khai báo *interface* với mô tả về dữ liệu sẽ dùng là “*chung chung*”, do đó chỉ cần định nghĩa *interface Wrokable* để *implements* cho nhiều class khác nhau (Mặc dù dữ liệu cụ thể là khác biệt). Điều này đem lại 2 ý nghĩa chính rất quan trọng là “*tổng quát hóa*” và “*đơn giản hóa*” trong triển khai mã nguồn chương trình

Ở phần này, chúng ta sẽ áp dụng một kỹ thuật tương tự, đó là “*generic methods*”. Như đã biết, theo yêu cầu của chương trình, có 2 loại dữ liệu **Customer** và **Order** cần phải đọc, ghi dưới dạng *Object file*. Vậy là mã nguồn của phương thức *readFromFile()* cũng sẽ phải được thực hiện 2 lần, tương tự với phương thức *saveToFile()* cũng vậy; điều này làm cho sự dư thừa mã nguồn chương trình có thể sẽ tăng lên rất cao. Để giảm bớt sự trùng lặp mã nguồn khi phải thực hiện cùng một hành vi, đối với nhiều loại dữ liệu khác nhau; Java cho phép người lập trình có thể áp dụng kỹ thuật “*generic methods*”. Do đó, việc “*giảm dư thừa*”, đồng thời tăng cường tính “*tổng quát hóa*” của thuật toán xử lý trong chương trình trở nên khả thi (Tham khảo: <https://docs.oracle.com/javase/tutorial/java/generics/methods.html>)

Vì các lớp **Customers** (*List collection*) và **Orders** (*Set collection*), do đó để thuận lợi cho việc triển khai kỹ thuật generic methods, ta sẽ xây dựng 2 phương thức *readFromFile()* và *SaveToFile()*, trong một Class riêng biệt có tên là **FileUtils** có prototype như sau

```
Public static <T> List<T> readFromFile(String filePath); và  
Public static <T> void saveToFile(List<T> li, String filePath);
```

- Chúng ta dùng **List** để truyền dữ liệu vào khi ghi; hoặc nhận về sau khi đọc file để cho đơn giản, dễ dùng (*đối với sinh viên*); Do đó, khi đọc hoặc ghi ở những cấu trúc không phải **List** (ví dụ: **Set**, **Map**) thì cần phải có bước trung gian để chuyển đổi

Phương thức **readFromFile(...)** được viết ở dạng generic methods

```
27  /** Phương thức cho phép đọc dữ liệu có trong tập tin yêu cầu ...6 lines */  
33  public static <T> List<T> readFromFile(String filePath){  
34      List<T> result = new ArrayList<>();  
35      FileInputStream fis = null;  
36      try {  
37          //--- 1. Tạo File object để ánh xạ lên thiết bị -----  
38          File f = new File(filePath);  
39          if (!f.exists()){  
40              System.out.println("File not found !.\""+filePath+"\"");  
41              return result;  
42          }  
43          //--- 2. Tạo luồng ánh xạ tới file để đọc dữ liệu từ thiết bị -----  
44          fis = new FileInputStream(f);  
45          //--- 3. Tạo đối tượng mang dữ liệu từ luồng đã tạo ở trên -----  
46          ObjectInputStream ois = new ObjectInputStream(fis);  
47          //--- 4. Lặp và đọc dữ liệu từ file, gán vào đối tượng hiện hành khi còn dữ liệu  
48          while (fis.available()>0){  
49              T x = (T) ois.readObject();  
50              result.add(x);  
51          }  
52          //--- 5. Đóng đối tượng, sau khi đọc xong  
53          ois.close();  
54      } catch (FileNotFoundException ex) {  
55          Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
56      } catch (IOException ex) {  
57          Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
58      } catch (ClassNotFoundException ex) {  
59          Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
60      } finally {  
61          try {  
62              fis.close();  
63          } catch (IOException ex) {  
64              Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
65          }  
66      }  
67      return result;  
68  }
```

Quan sát các dòng 33, 34 và 49 để hiểu cách khai báo và sử dụng **generic data**

Tương tự, phương thức **saveToFile(...)** được viết như sau:

```
69  /** Phương thức cho phép ghi dữ liệu vào object file ...6 lines */  
75  public static <T> void saveToFile(List<T> li, String filePath){  
76      FileOutputStream fos = null;  
77      try {  
78          //--- 1. Tạo File object -----  
79          File f = new File(filePath);  
80          //--- 2. Tạo FileOutputStream ánh xạ tới File object -----  
81          fos = new FileOutputStream(f);  
82          //--- 3. Tạo ObjectOutputStream để chuyển dữ liệu xuống thiết bị -----  
83          ObjectOutputStream oos = new ObjectOutputStream(fos);  
84          //--- 4. Lặp để ghi dữ liệu -----  
85          for(T i: li)  
86              oos.writeObject(i);  
87          //--- 5. Đóng các object tương ứng sau khi xử lý -----  
88          oos.close();  
89      } catch (FileNotFoundException ex) {  
90          Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
91      } catch (IOException ex) {  
92          Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
93      } finally {  
94          try {  
95              fos.close();  
96          } catch (IOException ex) {  
97              Logger.getLogger(FileUtils.class.getName()).log(Level.SEVERE, null, ex);  
98          }  
99      }  
100  }
```

Các phương thức phục vụ cho việc đọc, ghi file trong **Customers** class, mã nguồn gọi sử dụng các phương thức tương ứng trong **FileUtils** class dưới dạng tường minh như sau:

```
120  /**
121   * Đọc và nạp dữ liệu từ file Customers.dat vào danh sách Khách hàng
122   */
123  public final void readFromFile() {
124      this.addAll(FileUtils.readFile(this.pathFile));
125      this.saved = true;    //-- Ghi nhận trạng thái là lưu thành công ---
126  }
127  /**
128   * Phương thức phục vụ cho việc lưu dữ liệu xuống file ở trên đĩa
129   */
130  public void saveToFile() {
131      FileUtils.saveToFile(this, pathFile);
132      this.saved = true;    //-- Ghi nhận trạng thái là lưu thành công ---
133  }
```

Tương tự khi gọi trong **Orders** class

```
134  /**
135   * Đọc và nạp dữ liệu từ file feast_order_service.date vào danh sách Đơn hàng
136   */
137  public void readFromFile() {
138      List<Order> l = FileUtils.readFile(this.pathFile);
139      for (Order i : l)
140          addNew(i);
141      this.saved = true;
142  }
143  /** Phương thức cho phép chuyển đổi từ HashSet thành List collection ...4 line
144  private List<Order> toList() {
145      return new ArrayList<>(this);
146  }
147  /**
148   * Phương thức phục vụ cho việc lưu dữ liệu xuống file feast_order_service.date
149   */
150  public void saveToFile() {
151      //-- 0. Nếu đã lưu rồi thì thôi, không ghi nữa
152      if (this.saved) return ;
153      FileUtils.saveToFile(toList(), pathFile);
154  }
```

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 doanh nghiệp, người dùng*)

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:

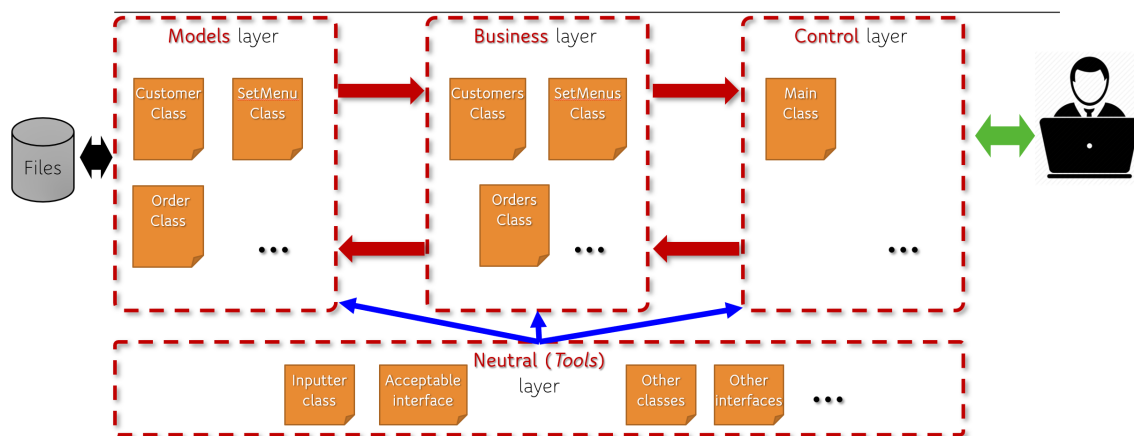
- Những mã nguồn chỉ phục vụ cho việc mô tả dữ liệu chính như: **Customer** class, **SetMenu** class, **Order** class thì quản lý bởi **model** package, gọi là mức thứ nhất: **model layer**.
- Những mã nguồn chuyên thực hiện các nghiệp vụ (*add, showAll, searchByName, update, delete, filter, ...*) có liên quan đến **model layer**, như **Customers** class,

SetMenus class, **Orders** class, **Workable** interface 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** package, tạm gọi là **neutral layer**.

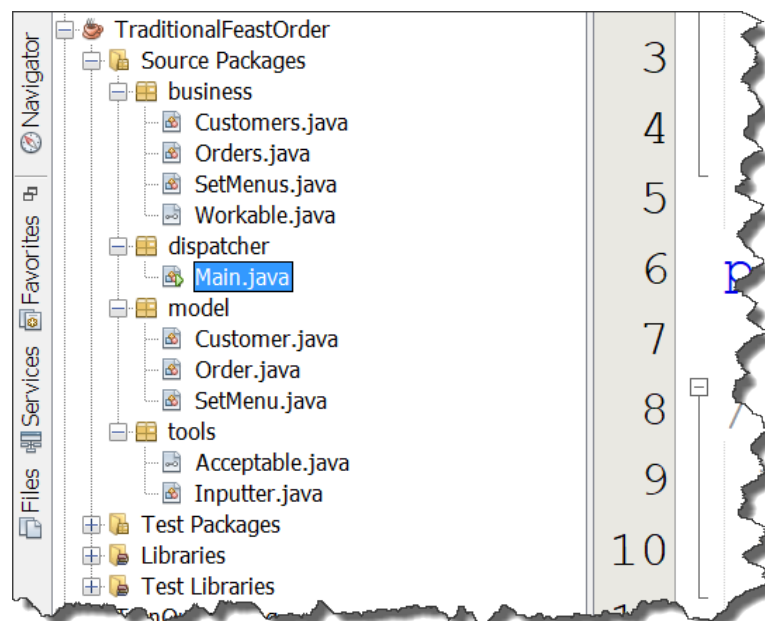
Hãy quan sát sơ đồ thiết kế được dùng cho project có dạng như sau

Core flow



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 “**TraditionalFeastOrder**” 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.

c. 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 rồi 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.

d. View (Giao diện người dùng):

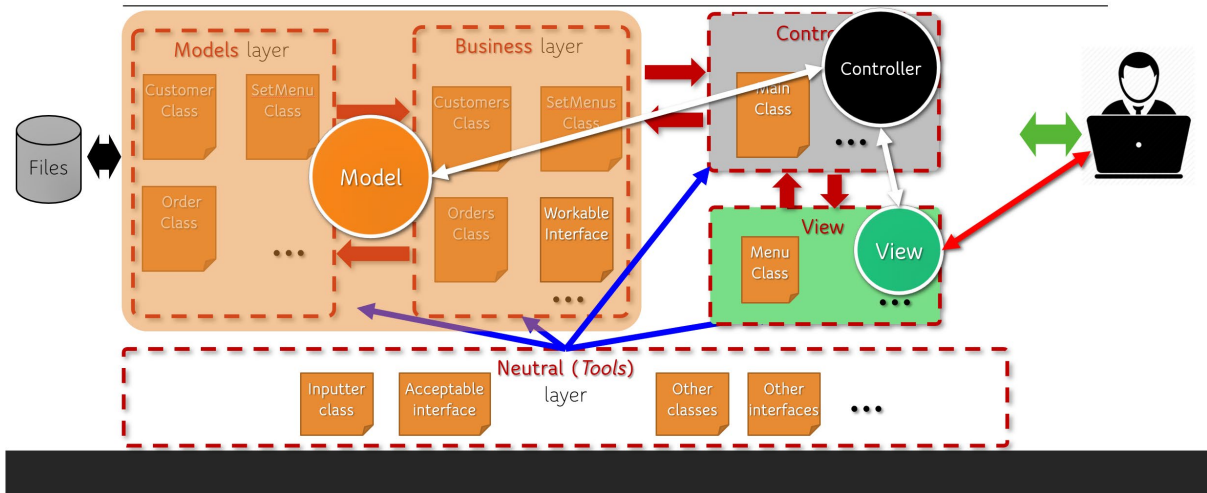
- View chịu trách nhiệm hiển thị dữ liệu (**render**) được cung cấp bởi Model cho người dùng
 - quan sát (*kết quả thực thi, màn hình kết quả, Menu, kết quả tìm kiếm*)
 - tiếp nhận các tương tác do người dùng thực hiện trên thiết bị. Nó là phần mà người dùng tương tác trực tiếp, VD: Menu cho phép chọn 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 của nó là
 - nhận dữ liệu từ Model và
 - hiển thị kết quả dưới dạng mà người dùng có thể hiểu và tương tác.

e. 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

Model-View-Controller design pattern



**Chúc các bạn vận dụng tốt không những trong môn học này
mà cả ở các môn học tiếp theo !.**