Template method

Nơi sử dụng

Template method trong thư mục ui/templatePattern

strategyPattern trong thư mục ui/strategyPattern

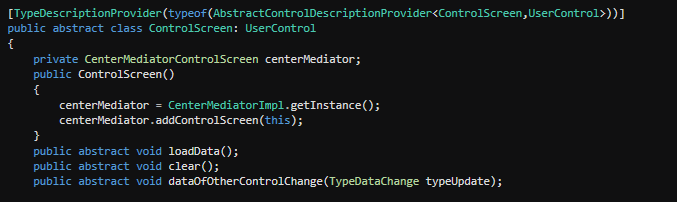
FormHome: đổi màn hình(đổi chiến lược) khi chạy và template method làm lại hoạt động (các child là các control)

Lý do sử dụng

Template Pattern giúp ta tạo nên một bộ khung của một chức năng nào đó và có thể cho lớp con định nghĩa lại một số hành vi trong bộ khung. Mô hình này áp dụng khi thay đổi các UserControl, nó sẽ thực hiện các hành vi trong một bộ khung đã được định nghĩa trước đó và điều đó tránh việc lặp đi lặp lại code ở nhiều nơi, đồng thời đối với việc các lớp con có hành vi khác nhau, nó giúp định nghĩa lại những hành vi riêng biệt đó, tránh ảnh hưởng đến các hành vi có cấu trúc giống nhau giữa lớp con, cũng như việc khởi tạo riêng cho các phương thức gây nên tình trạng dư thừa không đáng có. Về lâu về dài, điều này giúp hạn chế việc thay đổi code ở nhiều nơi mà chỉ cần chỉnh sửa ở một nơi nhất định.

Sử dụng

Template Method được sử dụng trong ứng dụng bằng cách thông qua lớp ControlScreen. Lớp ControlScreen sẽ chứa hai bộ khung code template có sẵn trong hai hàm là screenClick và firstActive. Hàm screenClick được kích hoạt khi các lớp con của ControlScreen được bấm để chuyển đổi với nhau. Và khi hàm screenClick được kích hoạt thì nó thực hiện một bộ khung có 4 hoạt động: đem lớp con của ControlScreen được bấm hiện lên, focus vô lớp đó và cuối cùng là tùy thuộc vào lớp con là lớp gì sẽ thực hiện hai method là loadData và clear. Chính vì 2 hoạt động loadData và clear được thực hiện tùy vào lớp con của ControlScreen nên nó sẽ được định nghĩa ở lớp con. Tương tự vậy thì hàm firstActive nó được dành cho việc hiện các lớp con ngay từ đầu. Và cũng như hàm screenClick thì khi được kích hoạt thì nó cũng sẽ thực hiện theo một bộ khung có sẵn.



Lớp ControlScreen



2 bộ khung

Factory Pattern

Nơi sử dụng

Folder frontController (FormFactory)

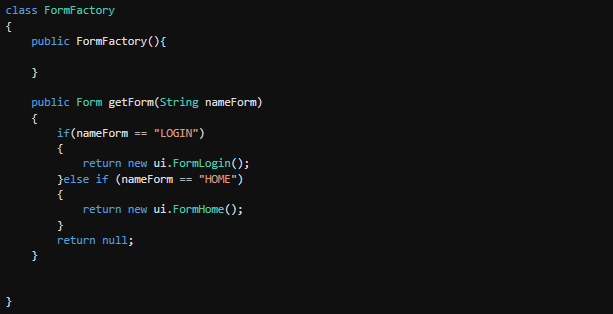
Để tạo ra các màn hình để chuyển

Lý do sử dụng

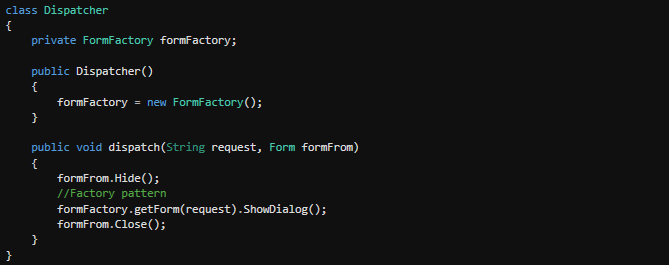
Factory Pattern là mô hình giúp ta tạo ra các đối tượng theo yêu cầu, được áp dụng để tạo ra các đối tượng Form khác nhau phụ thuộc vào Form muốn khởi tạo, đồng thời mẫu thiết kế cũng giúp ta có một cấu trúc code dễ nhìn, dễ quản lý hơn rất nhiều khi muốn tạo ra một đối tượng.

Sử dụng

Factory Pattern được sử dụng như sau. Lớp Dispatcher là lớp được dùng để chuyển đổi giữa các Form với nhau thông qua hàm dispatch. Và ở hàm này chúng ta cần tạo ra một lớp con của lớp Form tùy thuộc vào biến request để xác định lớp con của lớp Form được tạo ra. Và công việc tạo các lớp con của lớp Form sẽ được thực hiện bởi lớp FormFactory. Cho nên khi hàm dispatch được kích hoạt thì nó sẽ gọi hàm getForm ở lớp FormFactory và truyền biến request qua để lớp này có thể tạo ra các lớp con của Form dựa trên biến đó và trả về kết quả về hàm dispatch ở lớp Dispatcher để nó thực hiện các hoạt động dựa trên lớp con của Form đã được tạo ra.



Form Factory



Dispatcher sử dụng

Singleton Pattern:

Nơi sử dụng

Folder helper

@DatabaseMySql

Folder Dao

FrontController

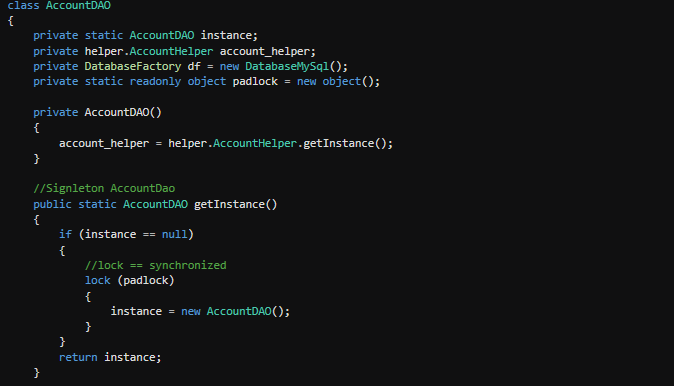
CenterMeidatorImpl

Lý do sử dụng

Singleton Pattern là một trong những mô hình được sử dụng phổ biến nhất, nó giúp cho việc tạo đối tượng chỉ diễn ra đúng một lần từ khi khởi tạo ứng dụng. Mô hình này giúp giữ các instance của những lớp chỉ cần khởi tạo một lần như: lớp Dao, Helper,… điều đó giúp cho việc gọi đến những lớp này dễ dàng và quan trọng hơn là dữ liệu trong những lớp đó sẽ không bị thay đổi khi gọi lại, làm cho quá trình xây dựng ứng dụng được liền mạch và xuyên suốt.

Sử dụng

Singleton được sử dụng như sau. Lấy ví dụ ở lớp AccountDAO khi ta khởi tạo thì ta sẽ không gọi hàm khởi tạo của lớp AccountDAO như bình thường vì khi ta sử dụng Singleton Pattern thì hàm khởi tạo được đặt ở chế độ private. Chính vì vậy khi ta muốn gọi đến lớp AccountDAO thì ta sẽ gọi qua hàm static getInstance và kết quả trả về là một instance AccountDAO. Hàm này sẽ tự động check biến static với tên instance và type là AccountDAO là có khác null không. Nếu bằng null thì hàm getInstance sẽ gọi hàm khởi tạo ở lớp AccountDAO và gán cho biến instance và trả về biến instance. Ngược lại thì nếu biến instance khác null thì hàm getInstance sẽ trả thẳng ra instance mà không gọi hàm khởi tạo của AccountDAO. Từ những điều trên thì nó sẽ giúp project của ta luôn luôn chỉ có một instance duy nhất của lớp AccountDAO. Tương tự như lớp AccountDAO thì các lớp ở thư mục dao, helper, lớp DatabaseMySql, FrontController, CenterMeidatorImpl đều hoạt động như vậy.



Singleton ở lớp AccountDAO

AbstractFactory:

Nơi sử dụng

Folder helper

Folder Dao

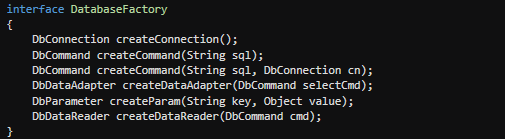
@DatabaseFactory, @DatabaseMySql

Lý do sử dụng

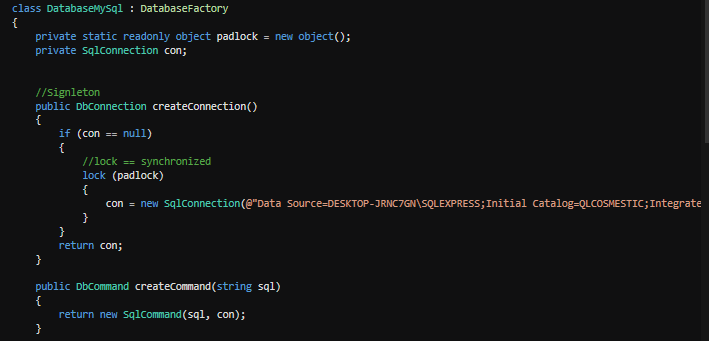
Abstract Factory Pattern là mẫu thiết kế giúp quản lý việc trả về các đối tượng liên quan với nhau. Do đó mẫu thiết kế này tạo ra một nơi thực hiện một việc duy nhất là quản lý, tạo và trả về các đối tượng kết nối database, command, param,… Hiện tại ứng dụng chỉ kết nối với một loại cơ sở dữ liệu chính là SQL nhưng sau này nếu kết nối với một loại cơ sở dữ liệu khác thì mô hình vẫn hỗ trợ để có thể gọi các method trả về đối tượng trước đó đã sử dụng bằng cách thêm một lớp con thừa kế lớp cơ sở dữ liệu của lớp cha mà đã định nghĩa các hàm khởi tạo và điều chỉnh các đối tượng được tạo ra tùy thuộc vào cơ sở dữ liệu đó mà không ảnh hưởng đến lớp cơ sở dữ liệu cũ, cũng như chương trình đã được thực hiện trước đó. Mặt khác, ta cũng giảm thiểu đi việc sử dụng if else quá nhiều để tạo các đối tượng ở cơ sở dữ liệu mà ta muốn.

Sử dụng

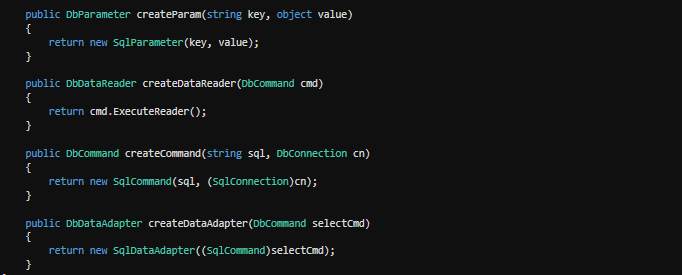
Abstract Factory Pattern được sử dụng như sau. Chúng ta sẽ có một interface DatabaseFactory, lớp này sẽ thực hiện việc định nghĩa các hành vi tạo các đối tượng Db sẽ có ở các lớp con. Và các lớp con sẽ thừa kế lớp DatabaseFactory định nghĩa lại các hành vi đó và tùy thuộc vào lớp con đó muốn trả về giá trị gì thì ở lớp con sẽ thực hiện trả về các giá trị đó. Giống như hiện tại lớp con DatabaseMySql sẽ thực hiện việc thừa kế lớp DatabaseFactory và định nghĩa lại các hành vi và trả về các đối tượng có dạng là Sql. Khi sử dụng ở lớp AccountHelper thì ta chỉ việc gọi lại class DatabaseFactory và phụ thuộc mình muốn khởi tạo các đối tượng loại Database nào thì mình sẽ khởi tạo Database đó ví dụ ta muốn đối tượng là là dạng sql thì ta khởi tạo DatabaseMySql. Sau đó ta chỉ việc gọi các hành vi tạo các đối tượng sử dụng ở các hàm checkAccountExistDb, selectCurrentUser,… Sau này ta có muốn thêm một lớp database nào khác thì ta chỉ cần tạo lớp database đó, thừa kế lại lớp DatabaseFactory, và định nghĩa lại các hành vi tạo đối tượng dưới dạng Db mà lớp đó muốn trả về và chỗ nào muốn sử dụng thì ta chỉ việc khởi tạo lớp Database đó dưới dạng lớp DatabaseFactory là được. Tương tự như các lớp ở trong thực mục dao, helper cũng thực hiện việc khởi tạo các object database như ở lớp AccountHelper.



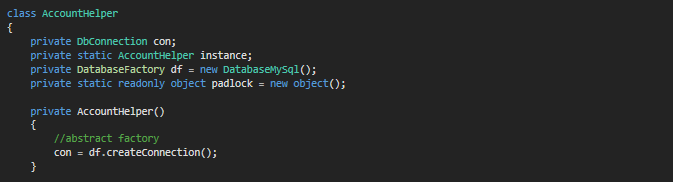
Lớp DatabaseFactory



Lớp DatabaseMySql



Lớp DatabaseMySql



Minh họa một chỗ sử dụng Abstract Factory

Command Pattern

Nơi sử dụng

Folder ui/Command

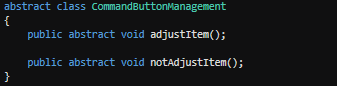
QuanLyKhachHangVaSuKienControl, QuanLyNhanVienControl, QuanLyNhapSanPhamControl, QuanLySanPhamControl

Lý do sử dụng

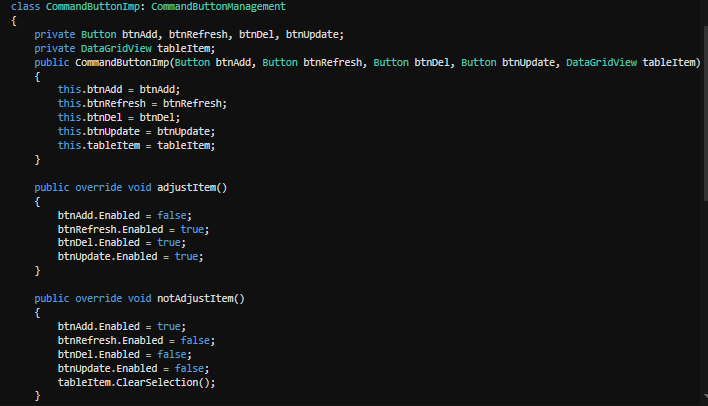
Command Pattern là mẫu thiết kế giúp đóng gói các thông tin cần thiết vào 1 đối tượng để thực hiện một hành động hay sự kiện nào đó. Nên chúng em áp dụng mẫu thiết kế này vào ứng dụng của chúng em để có thể thực hiện các hoạt động ẩn hoặc hiện các nút thêm, xóa, sửa, refresh, bảng dữ liệu ở mỗi màn hình quản lý. Điều đó giúp em có thể sau này muốn thay đổi code thực hiện các hoạt động ẩn hiện các nút cũng không ảnh hưởng đến lớp mà sử dụng nó, và đồng thời vì có nhiều màn hình sử dụng chức năng này nên khi muốn chĩnh sửa cũng không cần phải chĩnh sửa nhiều nơi.

Sử dụng

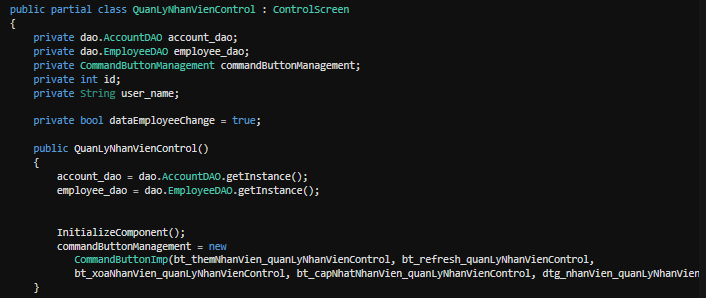
Command Pattern được sử dụng như sau. Chúng ta sẽ có 1 lớp abstract CommandButtonManagement, lớp này sẽ định nghĩa các hoạt động sẽ xảy ra ở lớp con và ở đây ta có hai hoạt động là adjustItem và notAdjustItem. Đồng thời có 1 lớp con là CommandButtonImp, lớp này sẽ thừa kế lại lớp CommandButtonManagement để định nghĩa lại các hành vi ở hai method trên và lớp này sẽ bao gồm các property có thuộc tính Button và 1 property có thuộc tính là DataGridView. Và ý nghĩa của các hành động đó như sau: ở các lớp QuanLyKhachHangVaSuKienControl, QuanLyNhanVienControl, QuanLyNhapSanPhamControl và QuanLySanPhamControl thì khi ta không chọn một item nào trên 1 DataGrideView thì ta sẽ gọi lệnh notAdjustItem ở lớp CommandButtonImp dựa trên lớp CommandButtonManagement để thực thi việc tắt các nút refresh, delete, update, bật nút add và xóa sự lựa chọn trên bảng DataGridView ở các lớp đó. Ngược lại nếu có 1 item nào được chọn để edit, xóa thì ta sẽ thực hiện lệnh adjustItem để bật các nút refresh, delete, update, tắt nút add ở các lớp trên.



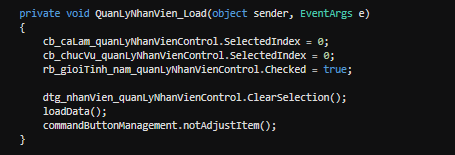
Lớp CommandButtonManagement



Lớp CommandButtonImp



Ví dụ minh họa sài Command Pattern ở lớp QuanLyNhanVienControl



Ví dụ minh họa sài Command Pattern ở lớp QuanLyNhanVienControl

Mediator Pattern

Nơi sử dụng

Thư mục MediatorControlScreen

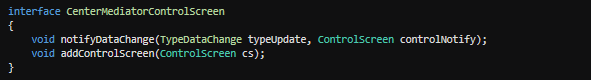
Thư mục ui/control

Lý do sử dụng

Mediator Pattern là mẫu thiết kế giúp các class tương tác, giao tiếp với nhau. Chúng em áp dụng pattern này vào ứng dụng để giúp cho việc khi có một màn hình nào thay đổi data thì có thể nói cho những màn hình khác rằng data đã được thay đổi và khi bấm sang các màn hình được thông báo thì có thể biết là có nên load lại data không và việc này cũng giúp chúng em tránh đi việc lúc nào cũng load lại data khi chuyển giữa các màn hình.

Sử dụng

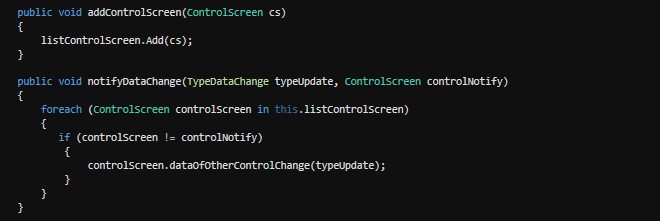
Mediator Pattern được sử dụng như sau. Ta có một lớp interface CenterMediatorControlScreen định nghĩa các hoạt động sẽ có trong lớp con của nó. Và cụ thể ở đây sẽ gồm 2 hoạt động là hoạt động addControlScreen với mục đích là thêm các lớp của ControlScreen vào một mảng và có thể sau này dùng để thông báo đến các ControlScreen đó khi cần và hoạt động thứ hai là hoạt động notifyDataChange với mục đích là khi có một ControlScreen có data thay đổi thì có thể dùng hàm này để thông báo đến các ControlScreen khác rằng có một dạng data nào đó đã thay đổi thông qua biến typeUpdate ví dụ nếu data product thay đổi thì typeUpdate sẽ là PRODUCT và nếu ControlScreen nào liên quan đến typeUpdate đó thì khi chuyển qua ControlScreen đó sẽ load lại data, còn không thì không cần load. Ví dụ như ở màn hình QuanLySanPhamControl có thay đổi về product thì sẽ gọi đến hàm notifyOtherControlDataChange ở lớp cha ControlScreen, lớp này sẽ gọi đến class CenterMediatorImpl để thông báo đến các ControlScreen khác là data product có sự thay đổi, trong đó có ControlScreen BanHangControl nên khi quay lại màn hình BanHangControl thì nó sẽ tự load lại data product.



Lớp CenterMediatorControlScreen



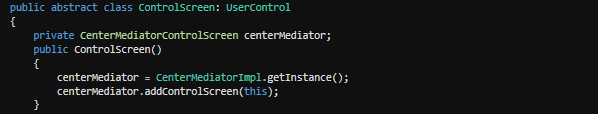
Lớp CenterMediatorImpl



Các hàm xử lý trong lớp CenterMediatorImpl

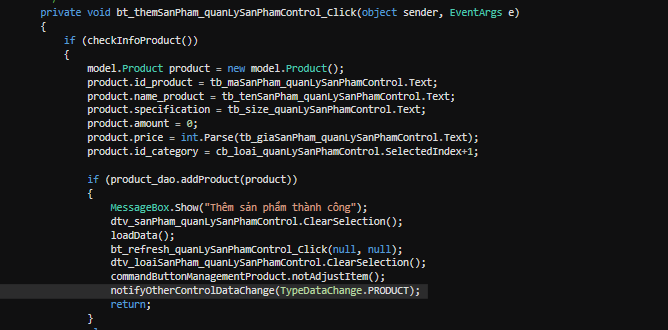


Kiểu dữ liệu TypeDataChange



Nơi sử dụng





Ví dụ thông báo đến các màn hình khác từ màn hình QuanLySanPhamControl với type là product khi sản phẩm có sự thay đổi

Memento Pattern

Nơi sử dụng

BanHangControl

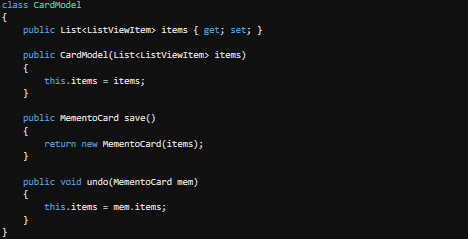
MementoBanHang

Lý do sử dụng

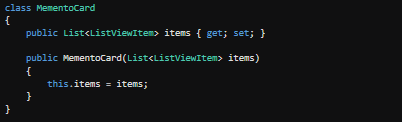
Memento Pattern là pattern giúp lưu lại trạng thái của một đối tượng để sau này có thể khôi phục lại trạng thái của đối tượng đó. Nên chúng em áp dụng pattern này vào ứng dụng để có thể lưu lại trạng thái giỏ hàng sau mỗi hoạt động để có thể thực hiện chức năng hoàn tác giỏ hàng trên ứng dụng và pattern này giúp em đơn giản code hơn trong việc sao lưu trạng thái của một đối tượng trong ứng dụng.

Sử dụng

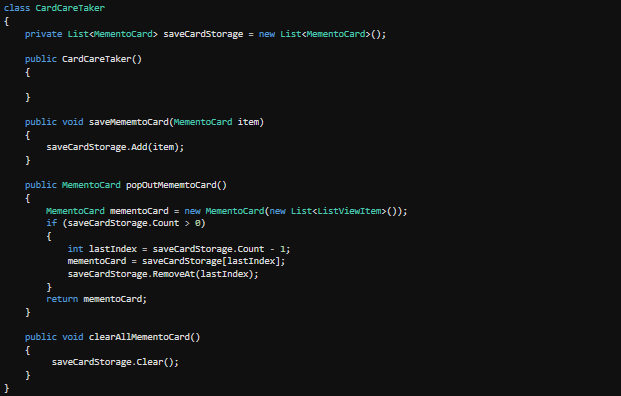
Memento Pattern được sử dụng như sau. Ta có 3 lớp chính là CardModel là model được dùng để lưu các item (sản phẩm) có trong giỏ hàng, lớp MementoCard là lớp được dùng để thể hiện trạng thái của CardModel và cuối cùng là lớp CardCareTaker được dùng để lưu trữ các trạng thái của CardModel và cũng là lớp giúp khôi phục lại trạng thái (MementoCard ) của CardModel. Cách hoạt động của Memento Pattern trong ứng dụng như sau là khi trong lớp BanHangControl có sự thay đổi trạng thái của giỏ hàng (CardModel), ta sẽ dùng lớp CardCareTaker lưu lại trạng thái củ thông qua method saveMememtoCard rồi mới update giỏ hàng (CardModel). Trong BanHangControl có một nút hoàn tác, khi ta bấm nút này, thì BanHangControl sẽ gọi đến method popOutMememtoCard để trả về trạng thái gần nhất và ta có thể gắn trạng thái (MementoCard) cho giỏ hàng hiện tại. Nếu thanh toán xong hoặc bấm nút refresh thì ta sẽ xóa mọi trạng thái của giỏ hàng thông qua hàm clearAllMementoCard trong lớp CardCareTaker.



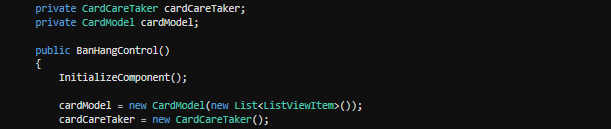
Lớp CardModel



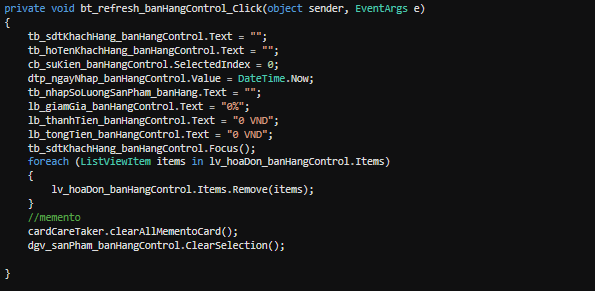
Lớp MementoCard



Lớp CardCareTaker



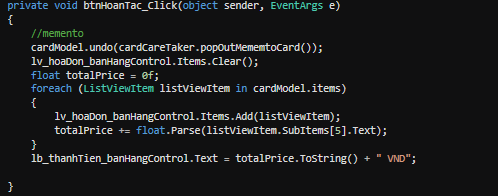
Sử dụng ở lớp BanHangControl để lưu lại giỏ hàng



Xóa mọi trạng thái giỏ hàng khi bấm nút refresh



Dùng CardCareTaker để save lại trạng thái cũ của CardModel và update trạng thái mới cho CardModel



Hoàn tác lại trạng thái cho CardModel khi bấm nút hoàn tác