TRƯỜNG ĐẠI HỌC GIA ĐỊNH

KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN

ỨNG DỤNG QUẢN LÝ NHÀ ĐẤT

MÔN **MẪU THIẾT KẾ CHO PHẦN MỀM**

Ngành **CÔNG NGHỆ THÔNG TIN**

Chuyên ngành **KỸ THUẬT PHẦN MỀM**

Giảng viên hướng dẫn **LÊ HUỲNH PHƯỚC**

Sinh viên thực hiện **ĐÀO TUẤN AN**

MSSV **22150316**

Lớp **221405**

TP. Hồ Chí Minh, tháng 08 năm 2024

Khoa/viện: Công Nghệ Thông Tin

NHẬN XÉT VÀ CHẤM ĐIỂM CỦA GIẢNG VIÊN
TIỂU LUẬN MÔN MẪU THIẾT KẾ CHO PHẦN MỀM

1. **Họ và tên sinh viên: Đào Tuấn An**
2. **Tên đề tài**: **Ứng dụng quản lý nhà đất**
3. **Nhận xét**:

   a) Những kết quả đạt được:

   .....

   .....

   .....

   .....

   .....

   b) Những hạn chế:

   .....

   .....

   .....

   .....

   .....

4. **Điểm đánh giá** *(theo thang điểm 10, làm tròn đến 0.5):*

   Sinh viên: Đào Tuấn An

   Điểm số: …… Điểm chữ: ……...

TP. HCM, ngày 31 tháng 07 năm 2024

Giảng viên chấm thi

(Ký và ghi rõ họ tên)

MỤC LỤC

CHƯƠNG 1 LÝ THUYẾT VỀ MẪU THIẾT KẾ

1. The Model-View-Controller Pattern (MẪU MVC)

1.1. vấn đề

Logic trình bày thường đề cập đến giao diện người dùng của một ứng dụng. Logic nghiệp vụ đề cập đến các thủ tục và dữ liệu của miền ứng dụng. Ví dụ: trong lĩnh vực ngân hàng, chúng tôi mong muốn tìm thấy dữ liệu đại diện cho các tài khoản, giao dịch và chủ tài khoản cũng như việc thực hiện các thủ tục chuyển tiền giữa các tài khoản.

Nhiều ứng dụng có thể sử dụng cùng một dữ liệu và quy trình: ngân hàng trực tuyến cho khách hàng, ứng dụng cho giao dịch viên, ứng dụng cho cố vấn tài chính và máy ATM. Tất cả các ứng dụng này sẽ có giao diện người dùng khác nhau. Ngoài ra, theo quy tắc chung, logic nghiệp vụ ổn định hơn so với logic trình bày. Giao diện người dùng luôn thay đổi, nhưng logic của ngân hàng không thay đổi nhiều trong 100 năm qua!

Nếu logic nghiệp vụ phụ thuộc vào logic trình bày, ví dụ như việc khai báo một lớp tài khoản bao gồm cả cách hiển thị trong cửa sổ ngân hàng trực tuyến, thì sẽ rất khó để tái sử dụng lớp này trong ứng dụng của giao dịch viên. Hơn nữa, bất kỳ thay đổi nào về cách hiển thị tài khoản sẽ yêu cầu thay đổi trực tiếp trong lớp tài khoản. Nói cách khác, lớp tài khoản không thể tái sử dụng trong các ứng dụng khác hoặc với các giao diện người dùng khác nhau.

Đây là một ví dụ đơn giản :

```
double square(double x) {
  double result = x * x;
  println("the square of " + x + " = " + result);
  return result;
}
```

Hàm này chứa một sự pha trộn độc hại của logic nghiệp vụ—tính bình phương của x—và logic trình bày—hiển thị kết quả trong cửa sổ console. Rõ ràng chúng ta không thể sử dụng hàm này với một ứng dụng máy tính mà hiển thị câu trả lời trong một cửa sổ đặc biệt, cũng như không thể sử dụng nó trong một ứng dụng kỹ thuật mà gọi hàm bình phương như một chức năng mức thấp.

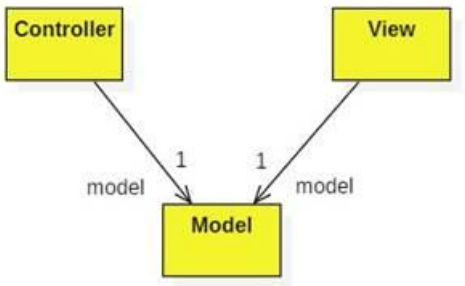
1.2. giải pháp

Mẫu kiến trúc Model-View-Controller (MVC) chia các ứng dụng thành ba phần: model chứa logic nghiệp vụ, views hiển thị logic nghiệp vụ cho người dùng, và controllers thực thi các lệnh của người dùng bằng cách cập nhật model. Cùng nhau, views và controllers (có thể có nhiều hơn một) tạo thành logic trình bày hoặc giao diện người dùng.

Chúng trình bày cho người dùng danh sách các lệnh có sẵn (thông qua các mục menu, nút bấm, bảng điều khiển và các điều khiển khác) và hiển thị cho người dùng những gì đang diễn ra trong model (thông qua màn hình, biểu đồ, đồng hồ đo và các thành phần đầu ra khác).

Đặc điểm quan trọng của kiến trúc Model-View-Controller là model không phụ thuộc, tham chiếu, hoặc sử dụng views hoặc controllers. Chúng ta có thể thay thế views và controllers mà không cần thay đổi bất kỳ điều gì trong model.

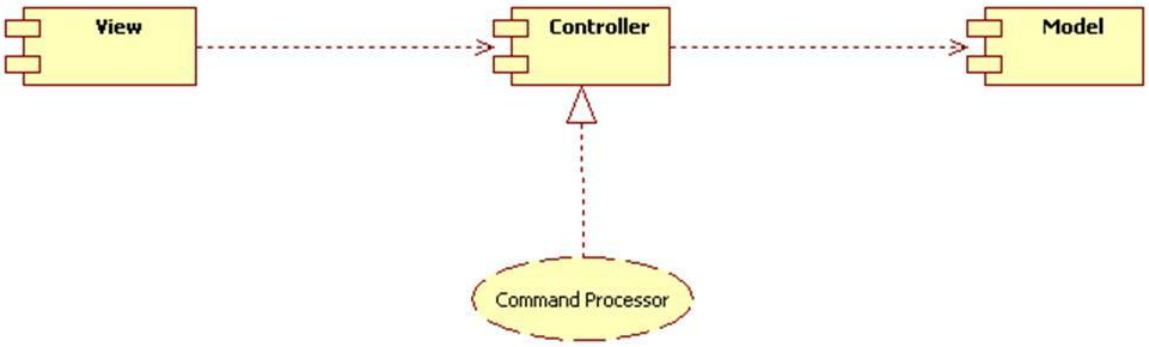
Dưới đây là một bản phác thảo UML về kiến trúc Model-View-Controller:



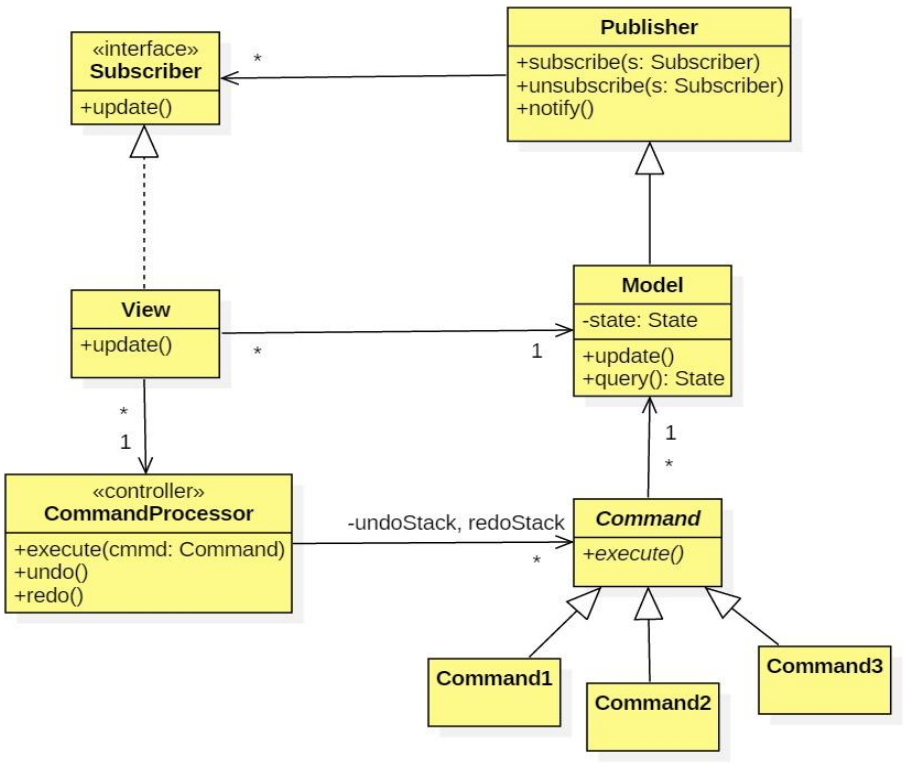
Lưu ý rằng controller và view chứa các tham chiếu một chiều đến model, nhưng ngược lại thì không.

2. Command Processor

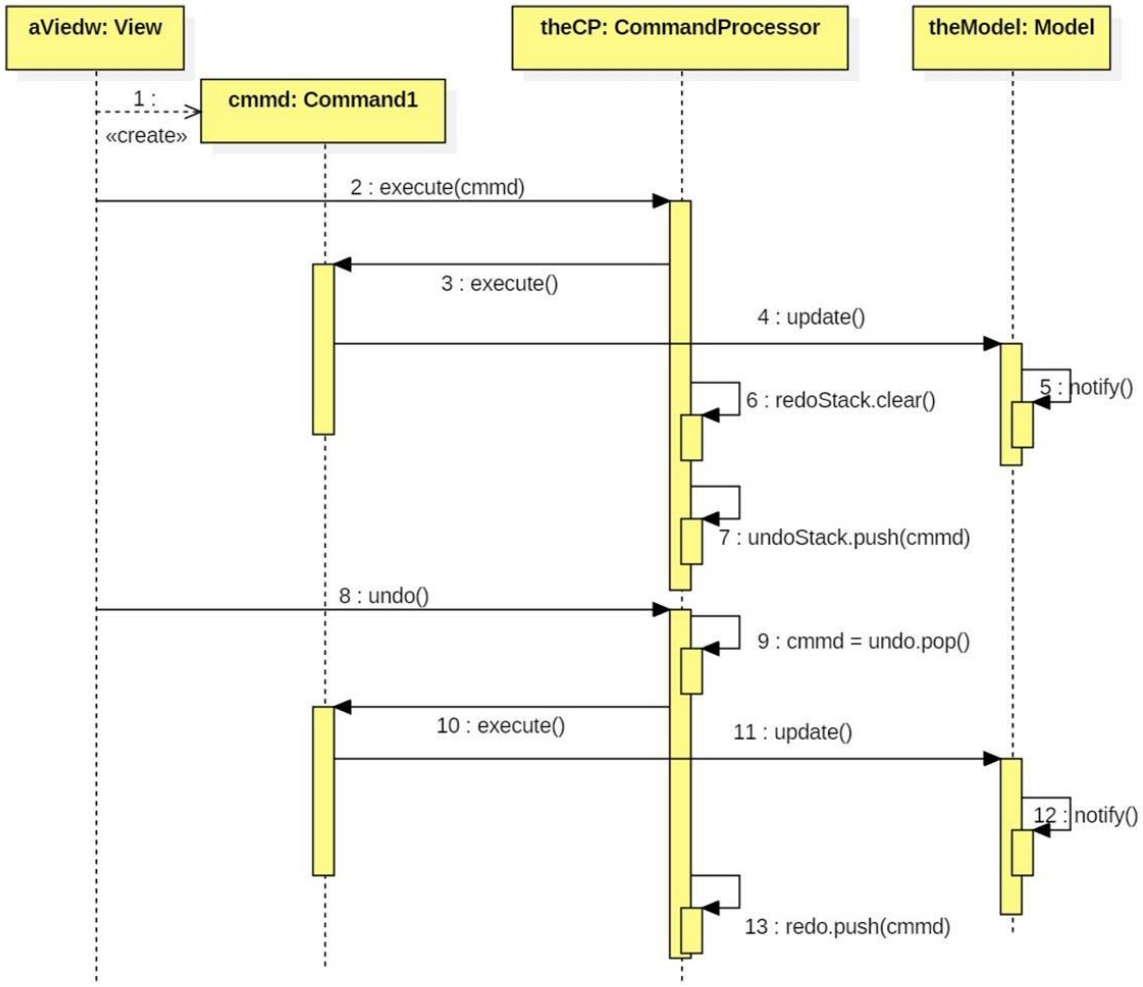
Mẫu Command Processor có thể được sử dụng để hiện thực hóa thành phần controller trong Kiến trúc Model-View-Controller:



Đây là mẫu :



Đây là một tương tác đối tượng điển hình:



Có nhiều biến thể của mẫu này. Biến thể được trình bày ở trên được gọi là biến thể "smart command" vì các lệnh biết cách thực thi chính chúng.

Trong biến thể "smart processor", các lệnh có thể là các chuỗi đơn giản. Phương thức `execute` của bộ xử lý lệnh là một điều kiện đa chiều khổng lồ.

```
if (cmmd instanceof ConcreteCommand1) {
   executeConcreteCommand1();
} else if cmmd instanceof ConcreteCommand2) {
   executeConcreteCommand2();
} else if (cmmd instanceof ConcreteCommand3) {
   executeConcreteCommand3();
} else ...
```

3. Singleton Pattern

3.1. vấn đề

Nhiều phiên bản của một lớp có thể không có ý nghĩa hoặc có thể gây nhầm lẫn.

3.2. giải pháp

Ẩn tất cả các hàm tạo. Cung cấp một phương thức xuất xưởng tĩnh để trả về một con trỏ tới một thể hiện duy nhất của lớp.

3.3. thảo luận

Giả sử một đối tượng của lớp Manager sẽ chịu trách nhiệm quản lý tất cả các thành phần của một ứng dụng. Việc có nhiều đối tượng quản lý có thể gây ra sự nhầm lẫn, vì vậy chúng ta ẩn tất cả các hàm khởi tạo. Các lớp trong Java và C++ đều có hàm khởi tạo mặc định công khai (tức là hàm khởi tạo không tham số) mà phải được chỉ định rõ ràng và đặt là private. Các lớp trong C++ cũng có hàm khởi tạo sao chép ngầm định mà phải được đặt là public.

```
class Manager
{
private:
   static Manager* theManager;
   Manager() { ... }
   Manager(const Manager& m) { }
   ~Manager() { }
public:
   // factory method:
   static Manager* makeManager()
   {
      if (!theManager) { theManager = new
Manager(); }
      return theManager;
   }
   // etc.
};
```

Rõ ràng phương thức xuất xưởng và con trỏ phải được đặt ở dạng tĩnh. Trong C++ bộ nhớ phải được cấp phát và khởi tạo riêng cho con trỏ:

```
Manager* Manager::theManager = 0;
```

Đây là một số mã khách hàng mẫu:

```
Manager* m1 = Manager::makeManager();
Manager* m2 = Manager::makeManager(); // m1 == m2
delete m1; // error, destructor can't be called here
m1 = new Manager(); // error, constructor can't be
called here
```

Giả sử hàm sau được khai báo:

```
void display(Manager m) { ... }
```

Tuy nhiên, việc gọi hàm không được phép:

```
display(*m1); // lỗi,hàm tạo sao chép không thể được gọi ở
đây
```

Các đối tượng quản lý phải được truyền bằng tham chiếu:

```
void display(Manager& m) { ... }
```

3. the composite pattern

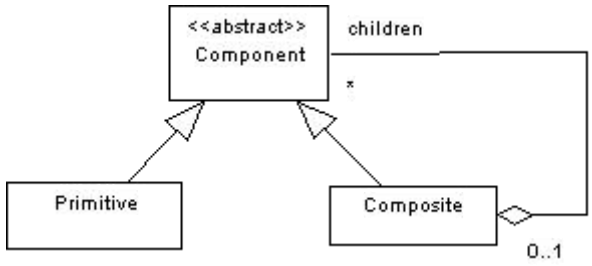
3.1. vấn đề

Cây thường được sử dụng để đại diện cho các cấu trúc lắp ráp và các loại hệ thống phân cấp khác.

3.2. giải pháp

Mẫu Composite giới thiệu một giao diện `Component` trừu tượng. Có hai loại thành phần: thành phần nguyên thủy hoặc đơn giản và thành phần hợp thành (còn gọi là lắp ráp hoặc tổng hợp). Một thành phần hợp thành duy trì một danh sách các tham chiếu (hoặc bản sao) của các phần hoặc con của nó. Những phần này có thể là nguyên thủy hoặc hợp thành.

3.3. cấu trúc



3.4. thảo luận

Dưới đây là cách triển khai Java giới thiệu `Component` như một giao diện trống:

```
interface Component { }
```

Một cách khác, lớp cơ sở (base class) Component có thể duy trì một tham chiếu đến lớp cha (parent) của nó.

```
class Component {
   protected Composite parent = null;
   // etc.
}
```

Các thành phần nguyên thủy là các loại tổng quát có thể chứa bất kỳ loại dữ liệu nào:

```
class Primitive<Data> implements Component {
   private Data data;
   public Primitive(Data data) {
      this.data = data;
   }
   public Data getData() { return data; }
   public void setData(Data data) { this.data = data; }
}
```

Một đối tượng composite (phức hợp) duy trì một tập hợp các đối tượng con, chúng có thể là các đối tượng nguyên thủy (primitives) hoặc các đối tượng phức hợp khác (composites).

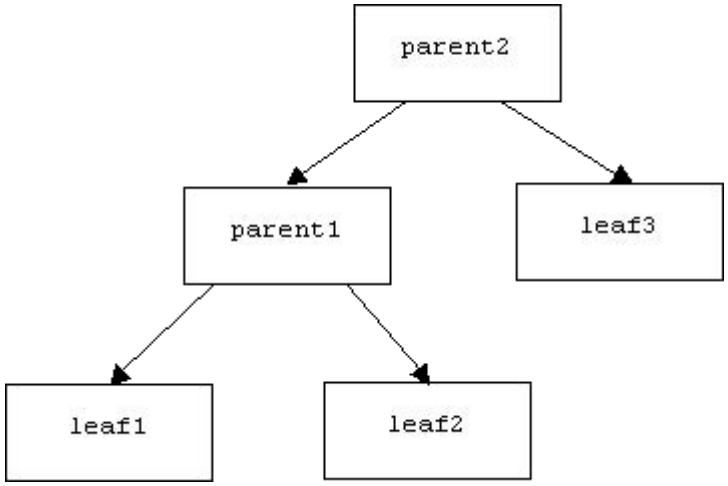
```
public class Composite implements Component {
   private Set<Component> children = new
HashSet<Component>();
   public void add(Component child)
{ children.add(child); }
   public void remove(Component child)
{ children.remove(child); }
   public Iterator iterator() { return
children.iterator(); }
};
```

đây là mẫu ví dụ:

```
      Primitive<String> leaf1 = new
Primitive<String>("Leaf 1");
      Primitive<String> leaf2 = new
```

```
Primitive<String>("Leaf 2");
 Primitive<String> leaf3 = new
Primitive<String>("Leaf 3");
 Composite parent1 = new Composite();
 Composite parent2 = new Composite();
 parent1.add(leaf1);
 parent1.add(leaf2);
 parent2.add(parent1);
 parent2.add(leaf3);
```

Đây là hình vẽ cái cây mà khách hàng đã tạo ra:



4. The Publisher-Subscriber Pattern

Nhà phát hành (Publishers) còn được gọi là người gửi (senders), đối tượng quan sát được (observables), chủ thể (subjects), đài phát thanh (broadcasters), và thông báo (notifiers). Người đăng ký (Subscribers) còn được gọi là người nhận (receivers), người nghe (listeners), người quan sát (observers), và hàm gọi lại (callbacks).

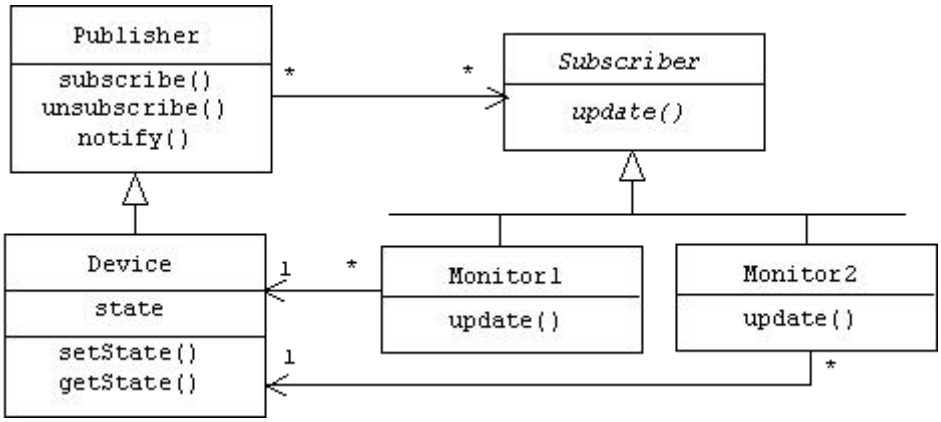
4.1. vấn đề

Nhiều bộ giám sát cần được thông báo khi trạng thái của một thiết bị thay đổi, nhưng số lượng và loại bộ giám sát có thể thay đổi linh hoạt. Chúng ta muốn tránh việc thăm dò liên tục (polling), và chúng ta không muốn đưa ra giả định trong mã thiết bị về số lượng và loại bộ giám sát.

4.2. giải pháp

Thiết bị nên duy trì một danh sách các con trỏ đến các bộ giám sát đã đăng ký. Các bộ giám sát có thể khác nhau, nhưng mỗi bộ phải triển khai cùng một giao diện, bao gồm một hàm update() mà thiết bị sẽ gọi khi trạng thái của nó thay đổi. Danh sách các con trỏ bộ giám sát có thể được quản lý bởi một lớp cơ sở Publisher có thể tái sử dụng. Một lớp cơ sở trừu tượng Subscriber định nghĩa giao diện mà các bộ giám sát phải triển khai.

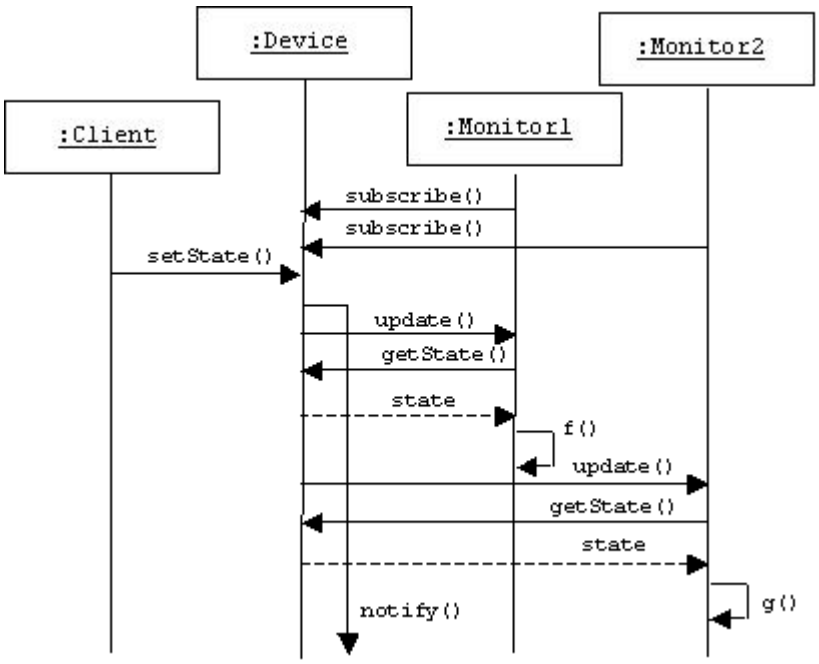
4.3. cấu trúc



Subscriber là một lớp trừu tượng vì update() là một hàm ảo thuần túy sẽ được triển khai khác nhau trong các lớp dẫn xuất khác nhau. Ngoài ra, chúng ta có thể định kiểu Subscriber như một giao diện (interface).

4.4. hành vi

Giả sử có hai bộ giám sát đăng ký vào một thiết bị. Khi thiết bị thay đổi trạng thái, monitor1.f() và monitor2.g() phải được gọi:



4.5. thảo luận

Các mục trong danh mục mẫu thường không bao gồm các triển khai cụ thể, vì chúng được thiết kế để độc lập với ngôn ngữ lập trình. Nhiệm vụ của lập trình viên là cung cấp triển khai phù hợp.

Một số mẫu rất hữu ích và được cung cấp trong các thư viện nổi tiếng. Ví dụ, các lớp nhà phát hành (publisher) và người đăng ký (subscriber) được cung cấp trong gói tiện ích Java, nơi chúng được gọi là Observable và Observer. Các chuyên biệt của các lớp nhà phát hành và người đăng ký cũng được cung cấp trong thư viện Microsoft Foundation Class, nơi chúng được gọi là CDocument và CView, và thư viện Visual Age của IBM, nơi chúng được gọi là Notifier và Subscriber.

Đáng tiếc là thư viện C++ tiêu chuẩn không bao gồm Publisher và Subscriber, vì vậy chúng ta sẽ phải tự mình triển khai chúng. Có nhiều cách tiếp cận, và ở đây chúng ta sẽ theo một cách tiếp cận dựa trên cách triển khai của Java.

4.6. pubsub.h

Chúng ta đặt các khai báo lớp nhà phát hành (publisher) và người đăng ký (subscriber) trong một tệp tiêu đề có tên là pubsub.h. Đây là cấu trúc của tệp:

```
/* File:            pop\util\pubsub.h
* Programmer:      Pearce
* Copyright (c):   2000, all rights reserved.
*/
#ifndef PUBSUB_H
#define PUBSUB_H
#include <list> // for our subscriber list
using namespace std;
class Publisher; // forward reference

class Subscriber { ... };
class Publisher { ... };

#endif
```

Ngoài một hàm hủy ảo (virtual destructor) không làm gì, giao diện subscriber (người đăng ký) chứa một hàm thuần ảo (pure virtual function) update. Để cho phép khả năng một subscriber có thể đăng ký nhiều publisher (nhà phát hành), chúng ta truyền một con trỏ tới publisher đang thông báo. Để cho phép các tin nhắn được gửi từ publisher, chúng ta cung cấp một con trỏ "message" (tin nhắn) tùy chọn và chung chung (generic):

```
class Subscriber
{
public:
 virtual ~Subscriber() {}
 virtual void update(Publisher* who, void* what = 0)
= 0;
};
```

Một publisher (nhà phát hành) duy trì một danh sách các con trỏ đến các subscriber (người đăng ký). Các bộ giám sát (monitor) triển khai giao diện subscriber có thể tự thêm mình vào danh sách bằng cách sử dụng hàm subscribe(), và tự xóa mình bằng cách sử dụng hàm unsubscribe(). Các thiết bị có thể gọi hàm update() của mỗi monitor đã đăng ký bằng cách gọi hàm notify().

Một client có thể cần thay đổi trạng thái của một thiết bị mà không thông báo cho các monitor. Điều này có thể được thực hiện bằng cách đầu tiên đặt cờ notifyEnabled thành false (sai).

```
class Publisher
{
public:
   Publisher() { notifyEnabled = true; }
   virtual ~Publisher() {}
   void subscribe(Subscriber* s)
{ subscribers.push_back(s); }
   void unsubscribe(Subscriber* s)
{ subscribers.remove(s); }
   void notify(void* what = 0, Subscriber *s = 0);
   void setNotifyEnabled(bool flag) { notifyEnabled =
flag; }
   bool getNotifyEnabled() const { return
notifyEnabled; }
private:
   list<Subscriber*> subscribers;
   bool notifyEnabled;
};
```

4.7. pubsub.cpp

Một publisher (nhà phát hành) duy trì một danh sách các con trỏ đến các subscriber (người đăng ký). Các bộ giám sát (monitor) triển khai giao diện subscriber có thể tự thêm mình vào danh sách bằng cách sử dụng hàm subscribe(), và tự xóa mình bằng cách sử dụng hàm unsubscribe(). Các thiết bị có thể gọi hàm update() của mỗi monitor đã đăng ký bằng cách gọi hàm notify().

Một client có thể cần thay đổi trạng thái của một thiết bị mà không thông báo cho các monitor. Điều này có thể được thực hiện bằng cách đầu tiên đặt cờ notifyEnabled thành false (sai).

```
/*
* File:          pop\util\pubsub.cpp
* Programmer:    Pearce
* Copyright (c):  2000, all rights reserved.
*/
#include "pubsub.h"
```

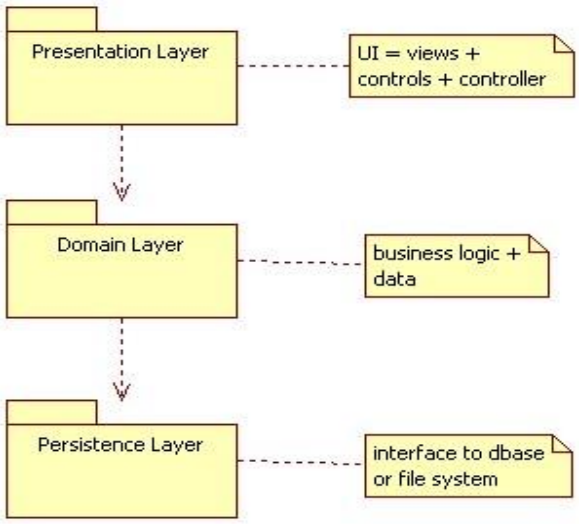
Đây là triển khai của hàm notify(). Lưu ý rằng notifyEnabled tự động được đặt thành true (đúng) ở cuối hàm. Điều này bảo vệ chống lại các trường hợp client quên đặt cờ này trở lại true sau khi đặt nó thành false.

Hàm notify có hai tham số. Tham số đầu tiên là một tin nhắn (message) tùy chọn và chung chung (generic). Tham số thứ hai trỏ đến một subscriber (người đăng ký) không cần được thông báo. Tính năng này hữu ích khi một subscriber muốn gửi một tin nhắn đến các subscriber khác thông qua publisher (nhà phát hành). Trong trường hợp này, subscriber gửi (sender) gọi hàm notify, truyền vào một tin nhắn và địa chỉ của nó. Mọi subscriber đều nhận được tin nhắn ngoại trừ subscriber gửi. Lưu ý rằng cả hai tham số đều có con trỏ null làm đối số mặc định.

```
void Publisher::notify(void* what, Subscriber* s)
{
   if (notifyEnabled)
   {
      list<Subscriber*>::iterator p;
      for(p = subscribers.begin(); p !=
subscribers.end(); p++)
         if (*p != s) (*p)->update(this, what);
   }
   notifyEnabled = true;
}
```

5. The Three-Layer Architecture

Mẫu thiết kế này còn được gọi là kiến trúc Ba Tầng (Three-Tier architecture) và mẫu Bùn thành Cấu trúc (Mud-to-Structure pattern).



Hãy lưu ý cẩn thận rằng các phụ thuộc là một chiều. Điều này ngụ ý rằng tầng persistence (duy trì dữ liệu) có thể được tái sử dụng với các tầng domain (lĩnh vực) khác nhau và tầng domain có thể được tái sử dụng với các tầng presentation (trình bày) khác nhau.

Thuật ngữ "tiers" (tầng) thường được sử dụng khi các tầng đang chạy trên các máy tính khác nhau. Trong trường hợp này, chúng ta có thể nghĩ tầng presentation là một thin client (ứng dụng khách mỏng), tầng domain là một máy chủ ứng dụng, và tầng persistence là một máy chủ cơ sở dữ liệu.

Chúng ta có thể khái quát hóa mẫu này thành kiến trúc N tầng bằng cách chia các tầng presentation và domain thành các tầng nhỏ hơn.

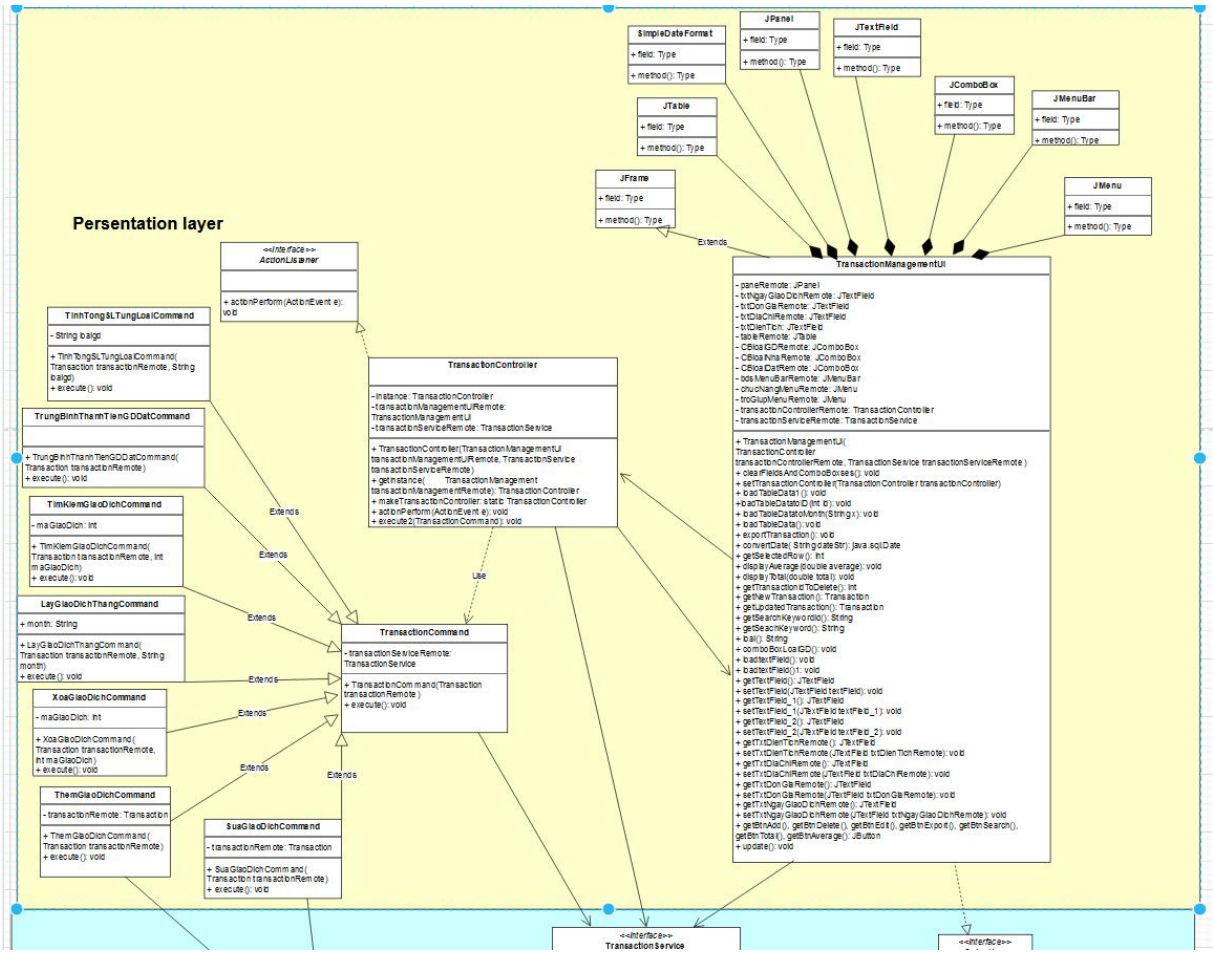
Kiến trúc này trở nên phổ biến nhờ kiến trúc "onion" (hành tây) của UNIX và kiến trúc tham chiếu ISO OSI.

Ví dụ phổ biến nhất là kiến trúc 3.5 tầng, trong đó một máy chủ web thực hiện một số công việc của tầng presentation và domain.

CHƯƠNG 2 PHÂN TÍCH, THIẾT KẾ CHƯƠNG TRÌNH

Sơ đồ thiết kế chương trình

Presentation layer (Tầng trình bày)



TransactionManagementUI: Đây là lớp chính quản lý giao diện người dùng liên quan đến giao dịch.

Chứa các thành phần giao diện như JFrame, JPanel, JTable, JTextField, JComboBox, JMenuBar, SimpleDateFormat và JMenu.

Giao diện này sẽ hiển thị các thông tin giao dịch, và cho phép người dùng tương tác như thêm, sửa, xóa, tìm kiếm giao dịch.

TransactionController: Lớp điều khiển chính cho các hành động liên quan đến giao dịch.

Sử dụng các dịch vụ từ TransactionService để thực hiện các hành động như thêm, sửa, xóa,tìm kiếm giao dịch.

Tương tác với TransactionManagementUI để cập nhật giao diện dựa trên các thay đổi trong giao dịch.

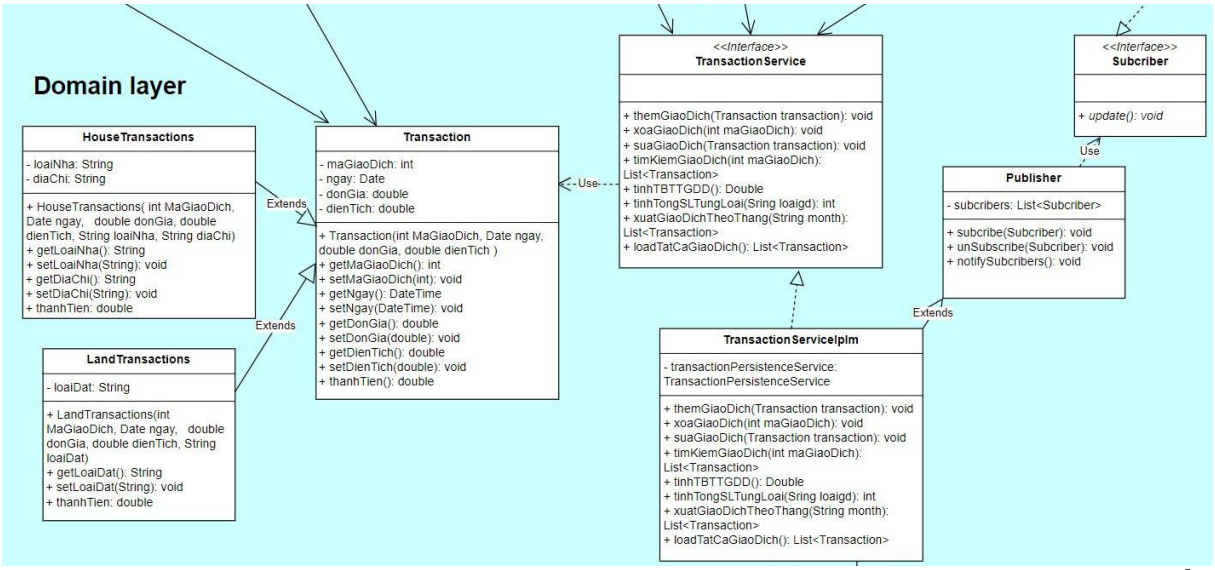
TransactionCommand: Lớp cơ sở cho các lệnh giao dịch.

Các lớp con như, ThemGiaoDichCommand, SuaGiaoDichCommand, XoaGiaoDichCommand, TimKiemGiaoDichCommand, LayGiaoDichThangCommand , TrungBinhThanhTienGDDatCommand, TinhTongSLTungLoaiCommandkế thừa từ lớp **TransactionCommand** . Và triển khai phương thức execute() để thực hiện hành động cụ thể.

TransactionActionListener: Lớp lắng nghe các hành động từ giao diện người dùng.

Khi người dùng thực hiện các hành động như thêm, sửa, xóa, lớp này sẽ chuyển tiếp các hành động đến TransactionController.

Domain layer (Tầng miền)



Transaction : là lớp đại diện cho giao dịch , có thuộc tính và phương thức cần thiết để thực thi lệnh giao dịch

Có các phương thức getter và setter để truy cập và thay đổi các thuộc tính này.

Lớp mô tả đối tượng giao dịch với các thuộc tính như mã giao dịch, đơn giá, ngày, diện tích

Chứa các phương thức để thao tác với đối tượng giao dịch như tính toán, lấy thông tin chi tiết

TransactionService: giao diện cho các dịch vụ giao dịch, định nghĩa cho phương thức thêm, sửa, xóa, tìm kiếm giao dịch

Lớp dịch vụ cung cấp các chức năng liên quan đến giao dịch.

Tương tác với lớp Transaction để thực hiện các hành động như thêm, sửa, xóa, tìm kiếm thông tin giao dịch

HouseTransactions: Kế thừa từ Transaction và thêm các thuộc tính đặc trưng riêng cho giao dịch nhà và đất.

Lớp mô tả các giao dịch liên quan đến bất động sản.

Chứa các thuộc tính và phương thức cụ thể cho các giao dịch này như thông tin nhà đất, diện tích, loại nhà, địa chỉ, thành tiền

LoandTransactions: Kế thừa từ Transaction và thêm các thuộc tính đặc trưng riêng cho giao dịch nhà và đất.

Lớp mô tả các giao dịch liên quan đến loại đất

Chứa các thuộc tính và phương thức cụ thể cho các giao dịch này như mã giao dịch, ngày, diện tích, thành tiền

TransactionServiceImpl: Triển khai cụ thể của TransactionService.

Thực hiện phương thức để quản lý giao dịch

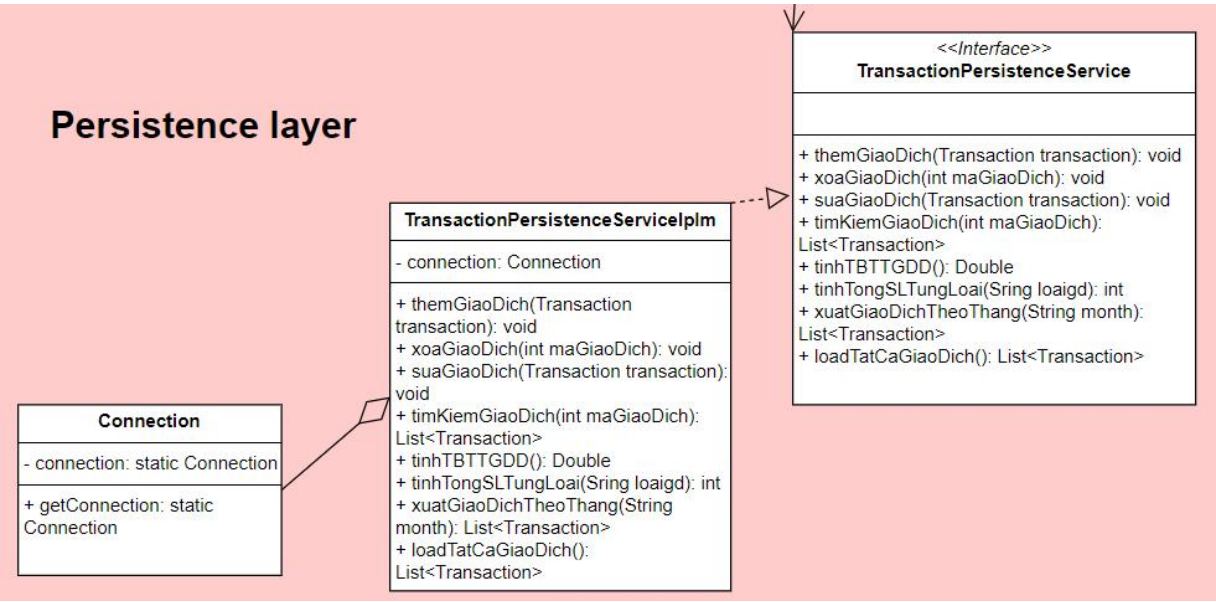
Lớp này để kiểm tra tính hợp lệ của giao dịch và lưu nó vào cơ sở dữ liệu thông qua TransactionPersistenceService.

Publisher và Subscriber:

Publisher quản lý danh sách các Subscriber và thông báo cho họ khi có thay đổi.

Subscriber sẽ thực hiện hành động khi nhận thông báo từ Publisher.

Persistence layer (Tầng lưu trữ)



TransactionPersistenceService: Giao diện cho các dịch vụ lưu trữ giao dịch, định nghĩa các phương thức để thêm, sửa và xóa giao dịch.

Lớp dịch vụ quản lý lưu trữ và truy xuất dữ liệu giao dịch.

Sử dụng lớp Connection để kết nối với cơ sở dữ liệu và thực hiện các thao tác lưu trữ, truy xuất dữ liệu.

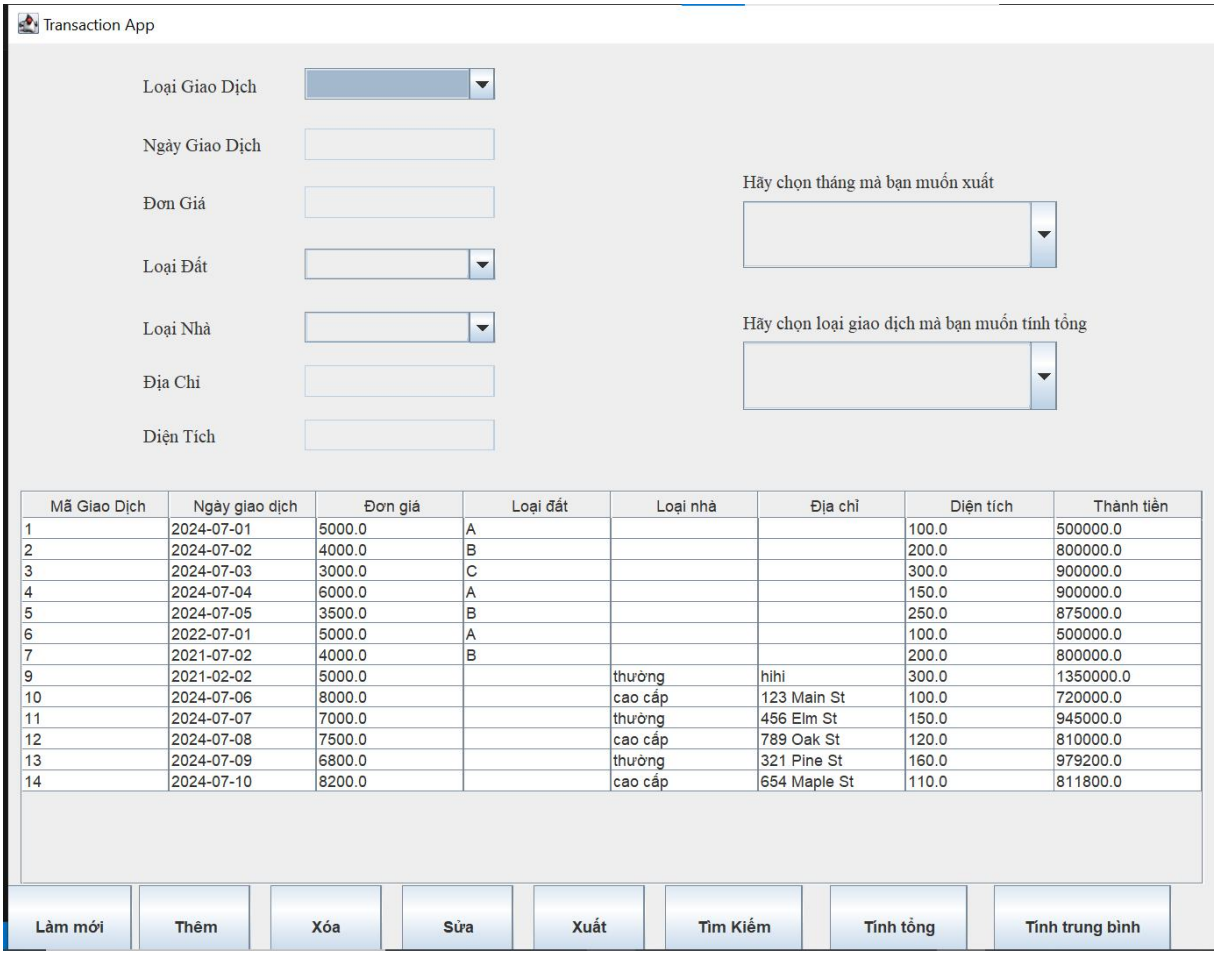
TransactionPersistenceServiceImpl: Lớp triển khai TransactionPersistenceService.

Connection: Đại diện cho kết nối cơ sở dữ liệu.

Lớp quản lý kết nối với cơ sở dữ liệu.

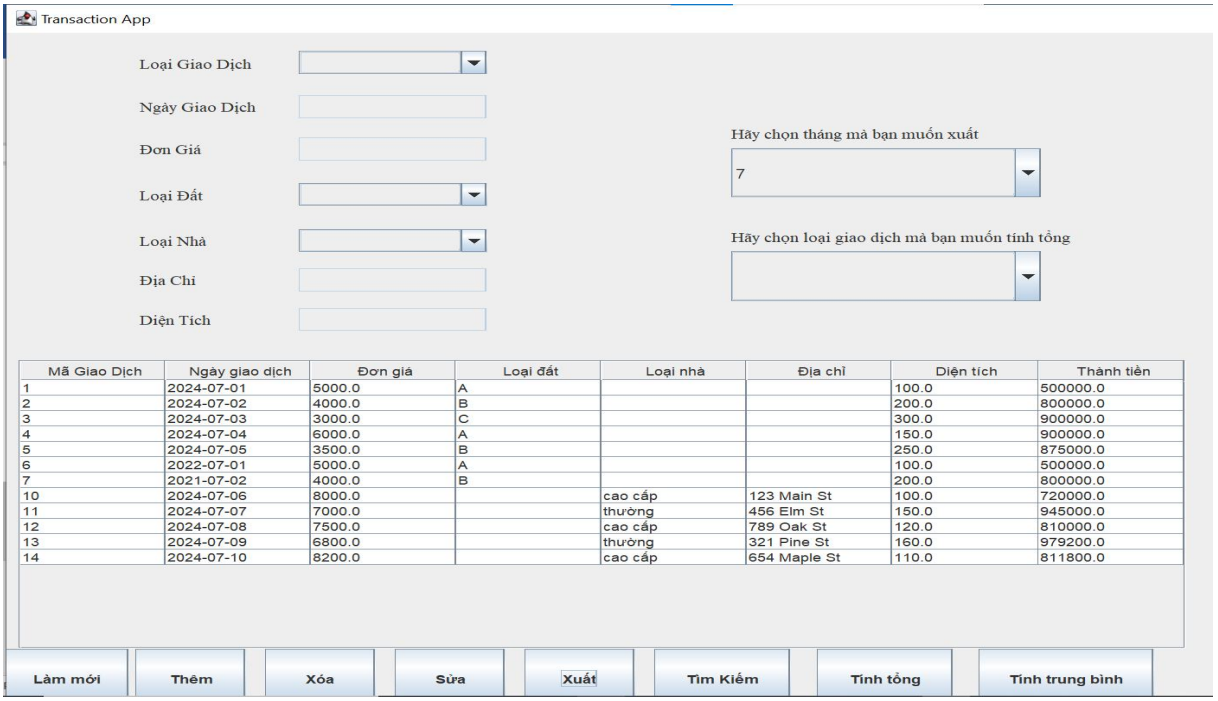
Cung cấp các phương thức để mở, đóng kết nối, thực hiện các truy vấn SQL.

CHƯƠNG 3 KẾT QUẢ CHƯƠNG TRÌNH



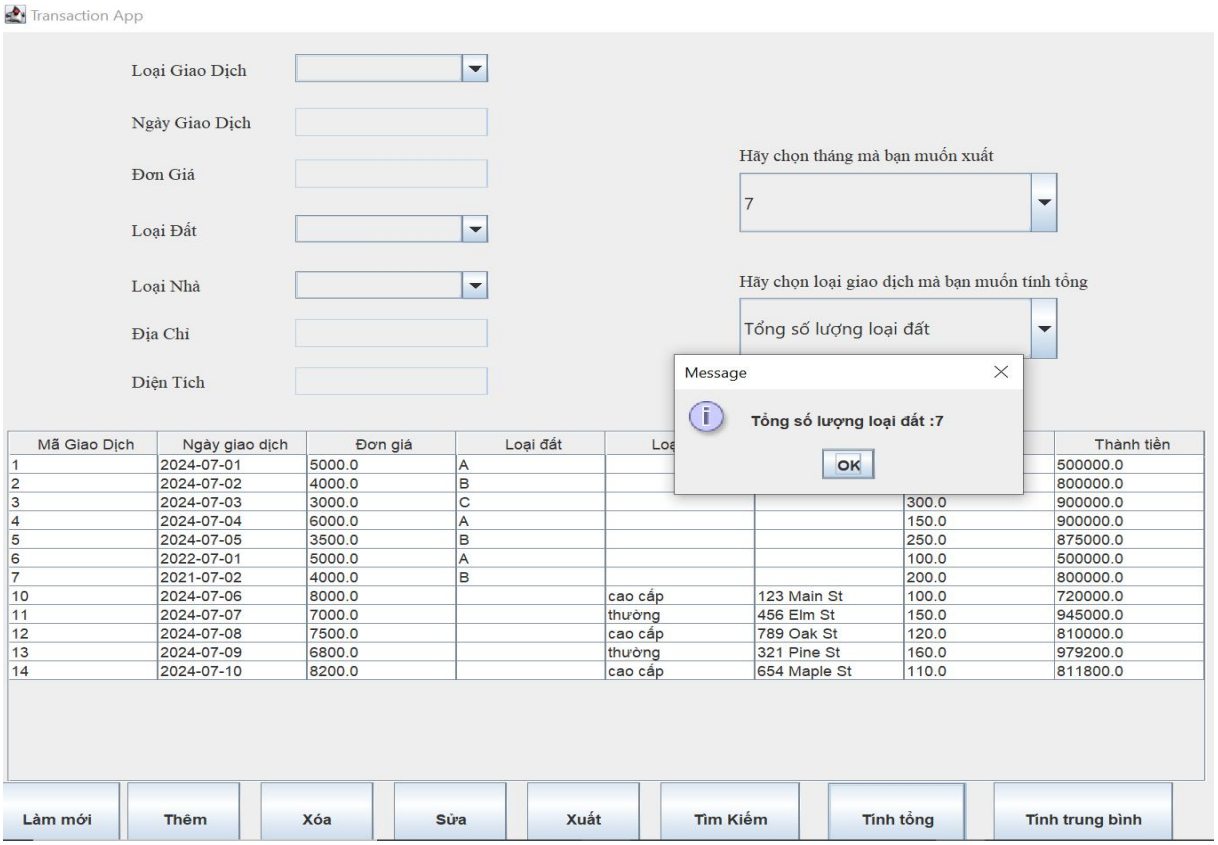
Mã Giao Dịch	Ngày giao dịch	Đơn giá	Loại đất	Loại nhà	Địa chỉ	Diện tích	Thành tiền
1	2024-07-01	5000.0	A			100.0	500000.0
2	2024-07-02	4000.0	B			200.0	800000.0
3	2024-07-03	3000.0	C			300.0	900000.0
4	2024-07-04	6000.0	A			150.0	900000.0
5	2024-07-05	3500.0	B			250.0	875000.0
6	2022-07-01	5000.0	A			100.0	500000.0
7	2021-07-02	4000.0	B			200.0	800000.0
9	2021-02-02	5000.0		thường	hihi	300.0	1350000.0
10	2024-07-06	8000.0		cao cấp	123 Main St	100.0	720000.0
11	2024-07-07	7000.0		thường	456 Elm St	150.0	945000.0
12	2024-07-08	7500.0		cao cấp	789 Oak St	120.0	810000.0
13	2024-07-09	6800.0		thường	321 Pine St	160.0	979200.0
14	2024-07-10	8200.0		cao cấp	654 Maple St	110.0	811800.0

Hình 1: giao diện chính của chương trình

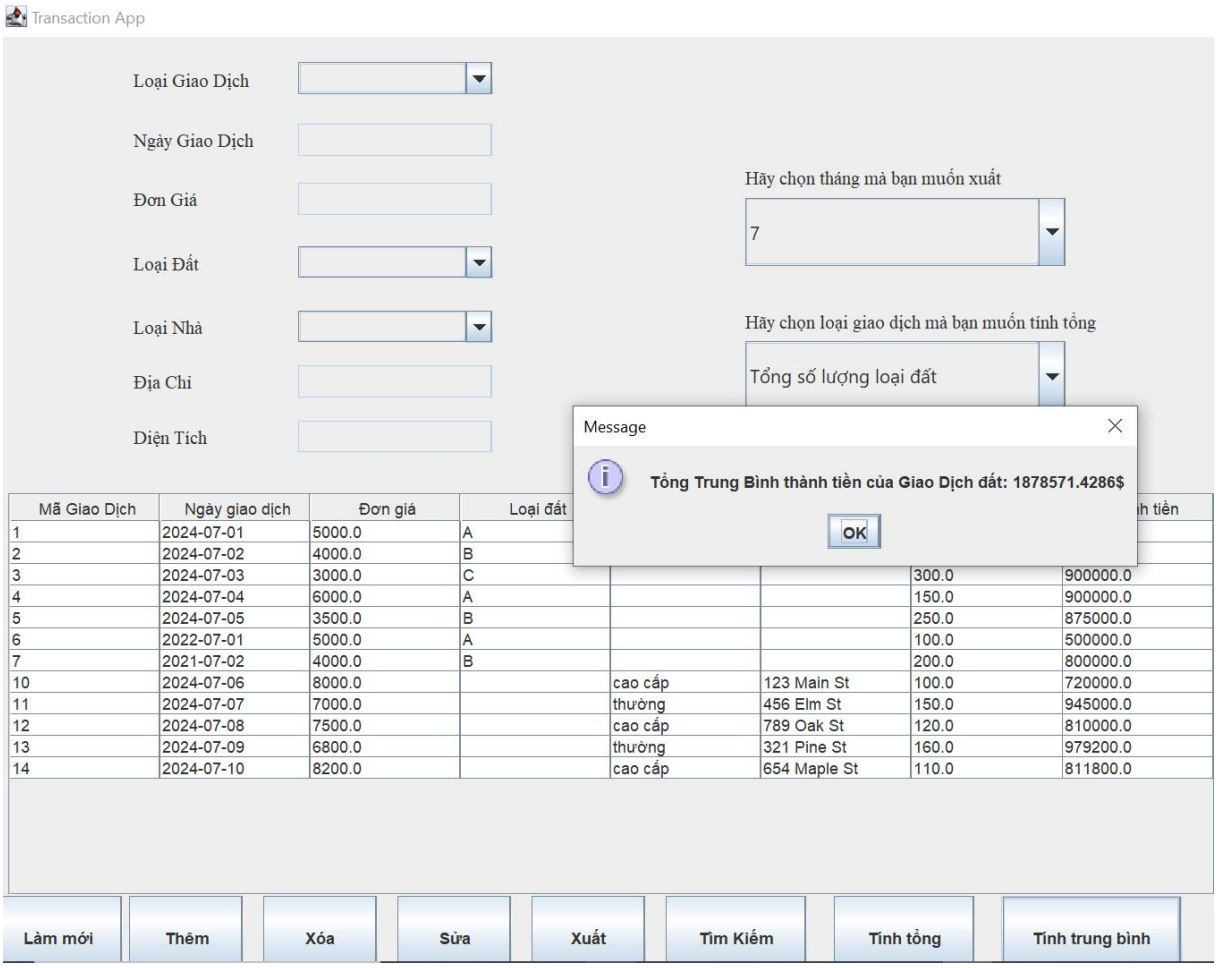


Mã Giao Dịch	Ngày giao dịch	Đơn giá	Loại đất	Loại nhà	Địa chỉ	Diện tích	Thành tiền
1	2024-07-01	5000.0	A			100.0	500000.0
2	2024-07-02	4000.0	B			200.0	800000.0
3	2024-07-03	3000.0	C			300.0	900000.0
4	2024-07-04	6000.0	A			150.0	900000.0
5	2024-07-05	3500.0	B			250.0	875000.0
6	2022-07-01	5000.0	A			100.0	500000.0
7	2021-07-02	4000.0	B			200.0	800000.0
10	2024-07-06	8000.0		cao cấp	123 Main St	100.0	720000.0
11	2024-07-07	7000.0		thường	456 Elm St	150.0	945000.0
12	2024-07-08	7500.0		cao cấp	789 Oak St	120.0	810000.0
13	2024-07-09	6800.0		thường	321 Pine St	160.0	979200.0
14	2024-07-10	8200.0		cao cấp	654 Maple St	110.0	811800.0

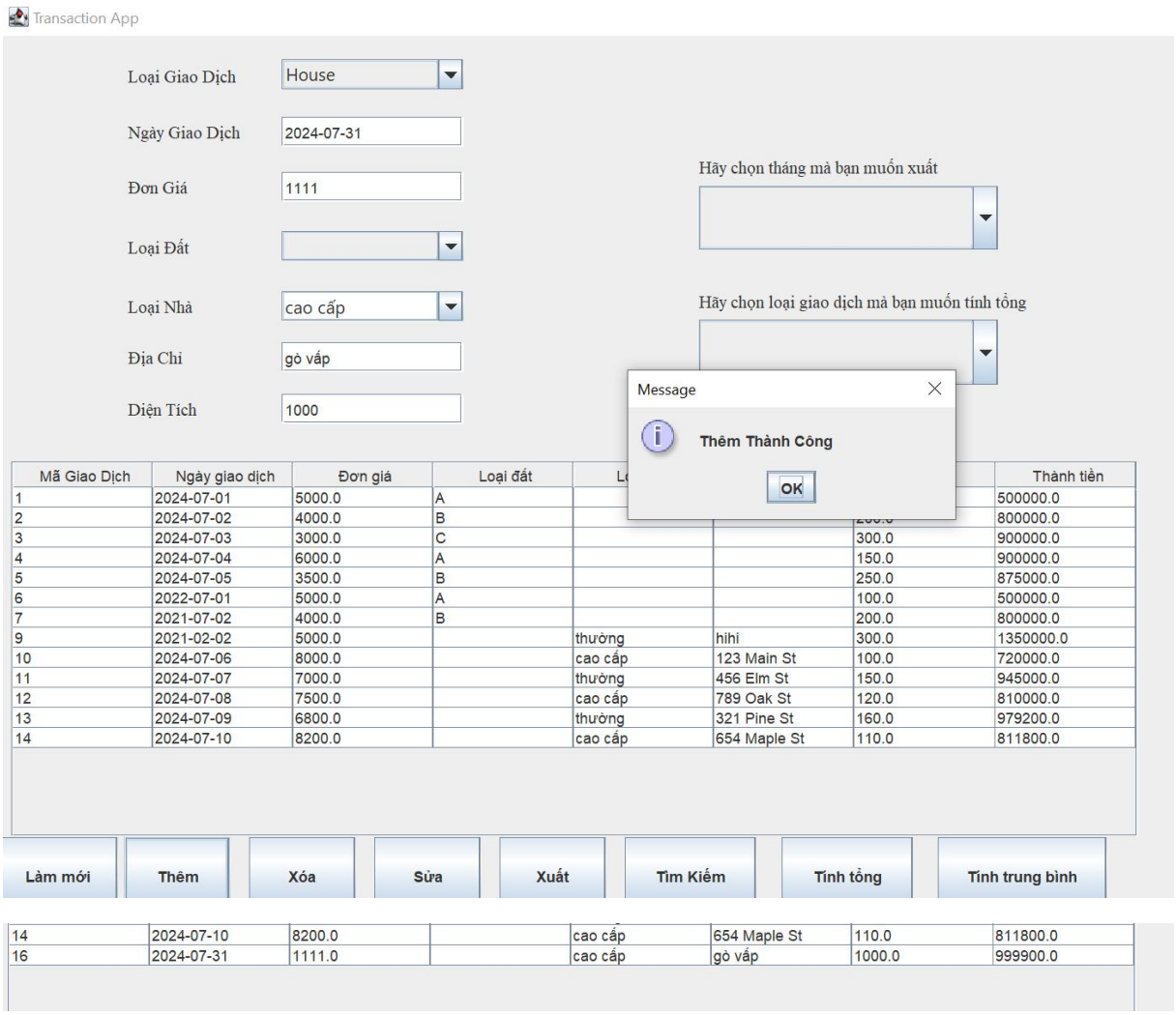
Hình 2 : cho ví dụ nhập tháng 7 vào combo box và nhấn button xuất thì kết quả hiện thị tất cả giao dịch vào tháng 7



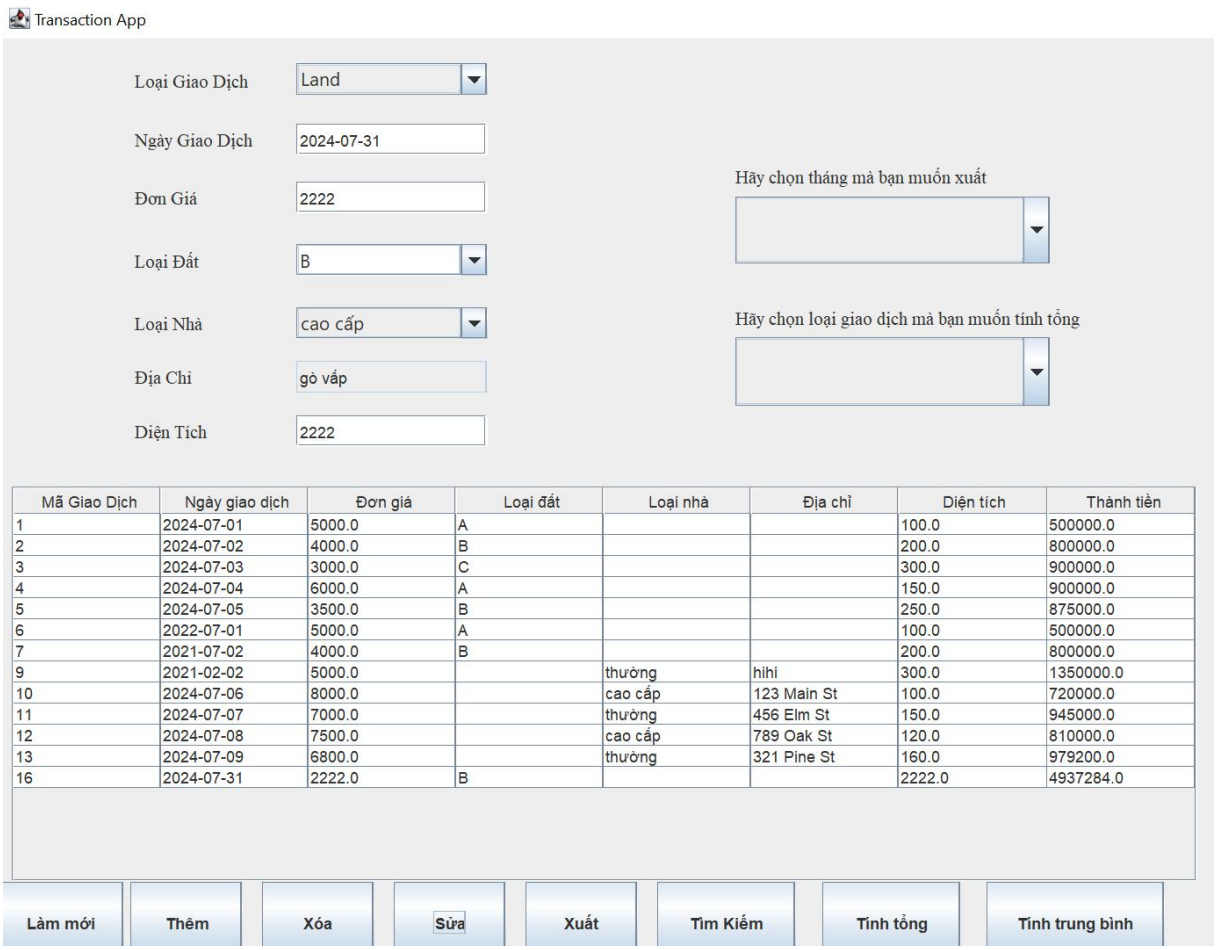
Hình 3: chọn tổng số lượng lạo đất ở combo box và nhấn nút tính tổng thì kết quả hiển thị message thông báo tổng số lượng loại đất đó



Hình 4: để tính tổng trung bình thành tiền của giao dịch đất thì chỉ cần nhấn button tính trung bình thì kết quả sẽ hiển thị message tổng trung bình



Khi thêm giao dịch mới



Khi sửa giao dịch có mã giao dịch 16 từ giao dịch nhà thành giao dịch đất

CHƯƠNG 4 KẾT LUẬN VÀ ĐÁNH GIÁ

Kết luận: Em đã tạo dựng chương trình ứng dụng nhà đất với những chức năng như thêm, sửa, xóa thông tin , tìm kiếm , tính tổng số lượng từng loại , tính tổng trung bình giao dịch đất để có thể phục vụ việc quản lý nhà đất.

Đánh giá: Vẫn chưa hoàn thiện chi tiết về mặt giao diện chương trình và tuổi thọ của chương trình