TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KÌ MÔN

MẪU THIẾT KẾ

PHẦN MỀM QUẢN LÝ BÁN HÀNG CHO CỬA HÀNG TIỆN LỢI

Người hướng dẫn: **THẦY VŨ ĐÌNH HỒNG**

Người thực hiện: **VŨ LƯƠNG NGỌC BAN – 52000010**

LƯU KIẾN VĂN – 51800832

ĐẶNG PHÙNG THIÊN ÂN – 52000622

NGUYỄN KHƯƠNG VIỆT TIẾN - 52000474

Lớp **: 20050201**

Khoá **: 24**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2023

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KÌ MÔN

MẪU THIẾT KẾ

PHẦN MỀM QUẢN LÝ BÁN HÀNG CHO CỬA HÀNG TIỆN LỢI

Người hướng dẫn: **THẦY VŨ ĐÌNH HỒNG**

Người thực hiện: **VŨ LƯƠNG NGỌC BAN – 52000010**

LƯU KIẾN VĂN – 51800832

ĐẶNG PHÙNG THIÊN ÂN – 52000622

NGUYỄN KHƯƠNG VIỆT TIẾN - 52000474

Lớp **: 20050201**

Khoá **: 24**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2023

LỜI CẢM ƠN

Chúng em gửi lời cảm ơn chân thành đến thầy Vũ Đình Hồng đã nhiệt tình giảng dạy, hỗ trợ chúng em trong môn học Mẫu thiết kế. Qua môn học này, chúng em đã học được những kiến thức mới về các mẫu thiết kế, cách áp dụng chúng vào các dự án thực tế.

Chúng em cũng muốn gửi lời cảm ơn đến Ban giám hiệu và các giảng viên khác đã hỗ trợ chúng em có môi trường học tập và nghiên cứu. Chúng em tin rằng những kiến thức học được từ môn học này sẽ giúp chúng em phát triển trong kỹ năng của mình trong tương lai.

Em xin chân thành cảm ơn !

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của thầy Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào chúng xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày tháng năm

Tác giả

(ký tên và ghi rõ họ tên)

Ban

Văn

Ân

Tiến

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Phần đánh giá của GV chấm bài

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

TÓM TẮT

Dưới đây là đồ án cuối kỳ môn Mẫu thiết kế, đồ án trình bày về cách áp dụng các mẫu thiết kế vào phần mềm quản lý bán hàng cho cửa hàng tiện lợi

Để thực hiện đồ án này, chúng em đã sử dụng các kiến thức về ngôn ngữ lập trình C#, hệ cơ sở dữ liệu Microsoft SQL Server và các design pattern để nâng cao chất lượng phần mềm.

MỤC LỤC

DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ

DANH MỤC HÌNH

CHƯƠNG 1 – MỞ ĐẦU

Phần mềm quản lý bán hàng cho cửa hàng tiện lợi là một dự án nhằm tạo ra một phần mềm hỗ trợ những nhân viên ở cửa hàng tiện lợi có thể dễ dàng hơn để hoàn thành công việc bán hàng của mình.

Phần mềm được thiết với giao diện thân thiện và dễ dàng sử dụng với đa số người dùng. Nhân viên cửa hàng có thể đăng nhập vào phần mềm thực hiện các chức năng chức năng thanh toán, quản lý sản phẩm cũng như doanh thu cửa hàng.

Tuy nhiên phần mềm ban đầu còn đơn giản, thiếu các chức năng và chưa tối ưu, cho nên chúng em sẽ áp dụng các mẫu thiết kế phù hợp để nâng cao chất lượng cho phần mềm.

Yêu cầu phần mềm sau khi áp dụng các mẫu thiết kế là phải đảm bảo phần mềm hoạt động tốt, giao diện thân thiện dễ tương tác, có đủ các chức năng cần thiết cũng như tối ưu chất lượng cho phần mềm.

CHƯƠNG 2 – CÁC MẪU THIẾT KẾ ĐƯỢC ÁP DỤNG

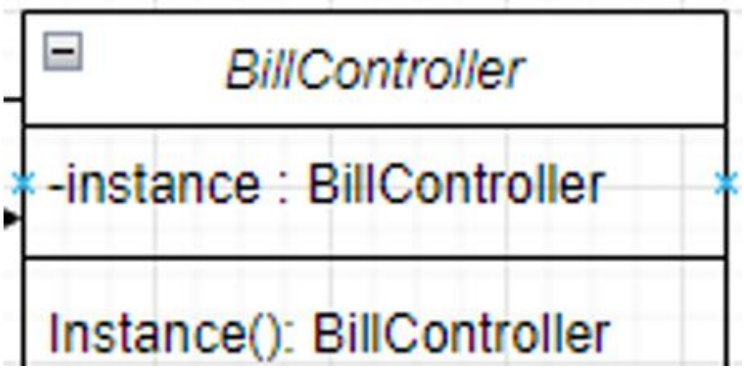
2.1 Singleton pattern

2.1.1 Lý do áp dụng

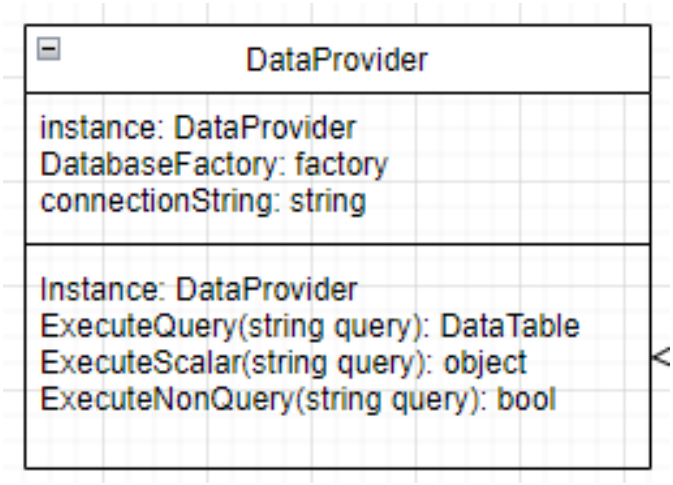
Dự án ban đầu chưa được xây dựng theo mô hình MVC và chưa có singleton, dẫn tới lãng phí tài nguyên hệ thống, ảnh hưởng hiệu suất hoạt động, cũng như có thể gây khó khăn trong việc kiểm soát truy cập đến các đối tượng, nguy cơ xảy ra lỗi và vấn đề bảo mật.

Vì vậy, sau khi tổ chức lại mô hình MVC thì chúng em đã áp dụng mẫu thiết kế singleton vào các hàm khởi tạo controller để đảm bảo tính duy nhất của đối tượng, quản lý tài nguyên hệ thống hiệu quả hơn cũng như dễ dàng kiểm soát và quản lý đối tượng trong hệ thống.

2.1.2 Sơ đồ lớp



Hình 1.1 Một trong những sơ đồ lớp singleton pattern(1)



Hình 2.2 Một trong những sơ đồ lớp singleton pattern (2)

2.1.3 Code áp dụng mẫu thiết kế:

```
namespace DAO
{
    28 references
    public class DataProvider
    {
        private static DataProvider instance;
        private static string connectionString = "Data Source=.\\SQLEXPRESS
                                                 "Initial Catalog=QuanLy711;" +
                                                 "Integrated Security=True";
        // Change this factory if migrating to another database
        private DatabaseFactory factory = new SqlDatabaseFactory(connection

        /*
            Singleton pattern ensures that only 1 instance of db controller
         */
        24 references
        public static DataProvider Instance
        {
            get { if (instance == null) instance = new DataProvider(); retu
            private set { instance = value;}
        }
```

Hình 3.3 Code áp dụng singeton pattern

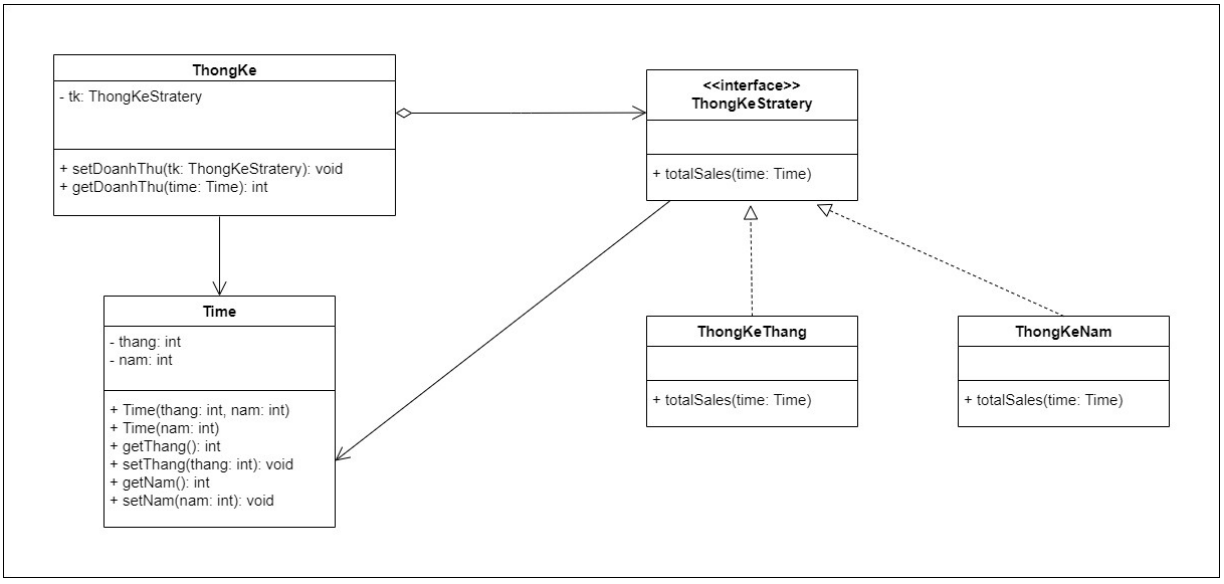
2.2 Strategy pattern

2.2.1 Lý do áp dụng

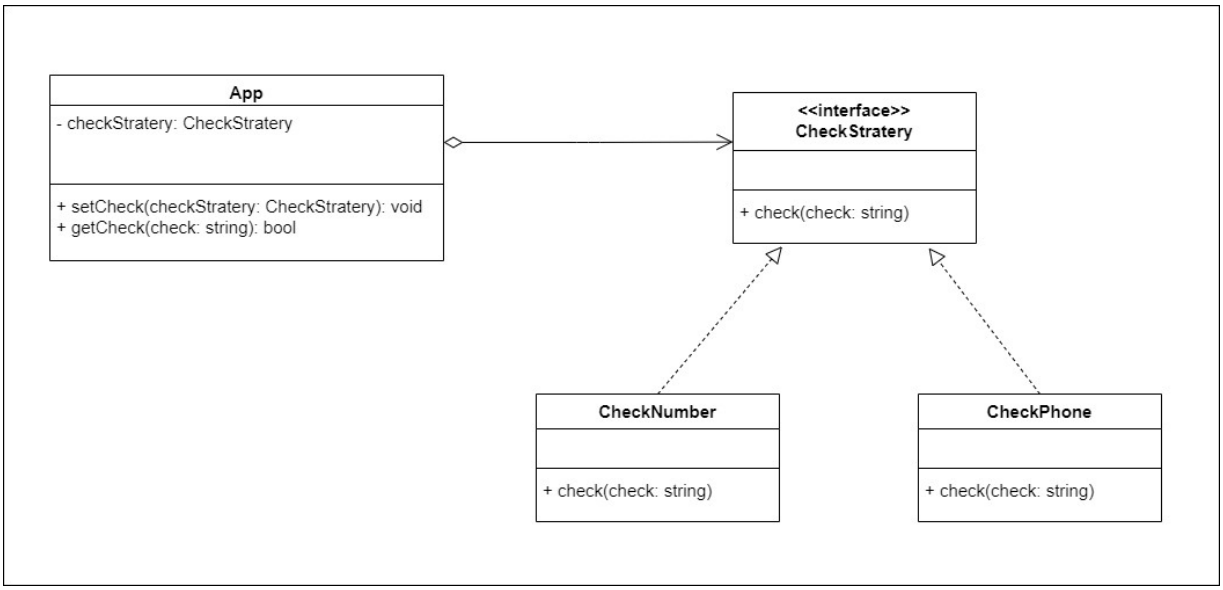
Trong dự án cũ có sử dụng các thuật toán để tính toán doanh thu và kiểm tra các số liệu đầu vào. Khi không có mẫu thiết kế strategy, việc bổ sung hoặc thay đổi các thuật toán sẽ khó khăn hơn vì phải sửa code ở mã nguồn thay vì chỉ cần tạo các lớp con, hoặc không thể tái sử dụng các thuật toán đó trong các tình huống khác nhau, làm cho việc phát triển hệ thống trở nên khó khăn và chậm chạp hơn.

Vì vậy, chúng em áp dụng mẫu thiết kế strategy vào chức năng tính doanh thu và kiểm tra số liệu đầu vào để giúp cho việc quản lý và thay đổi thuật toán trở nên dễ dàng, linh hoạt hơn, giảm sự phụ thuộc giữa các lớp trong hệ thống, cũng như khả năng tái sử dụng code và giải quyết các vấn đề liên quan đến thiết kế của hệ thống.

2.2.2 Sơ đồ lớp



Hình 2.4. Sơ đồ lớp strategy pattern doanh thu



Hình 2.5 Sơ đồ lớp strategy pattern kiểm tra số liệu đầu vào

2.2.3 Code áp dụng mẫu thiết kế:

```
using System.Text;
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Stratery.Check
{
    6 references
    public class App
    {
        private CheckStratery checkStratery;

        4 references
        public void setCheck(CheckStratery checkStratery)
        {
            this.checkStratery = checkStratery;
        }

        6 references
        public bool getCheck(string check)
        {
            return checkStratery.check(check);
        }
    }
}
```

Hình 2.6 Code áp dụng Strategy pattern (1)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Stratery.Check
{
    2 references
    public class CheckNumber : CheckStratery
    {
        2 references
        public bool check(string check)
        {
            if (int.Parse(check) > 0)
                return true;
            return false;
        }
    }
}
```

Hình 2.7 Code áp dụng Strategy pattern (2)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Stratery.Check
{

    public class CheckPhone : CheckStratery
    {

        public bool check(string check)
        {
            if (check.Length == 10)
                return true;
            return false;
        }
    }
}
```

Hình 2.8 Code áp dụng Strategy pattern (3)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Stratery.Check
{
    7 references
    public interface CheckStratery
    {
        3 references
        bool check(string check);
    }
}
```

Hình 2.9 Code áp dụng Strategy pattern (4)

```
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Stratery
{
    2 references
    public class ThongKe
    {
        private ThongKeStratery tk;

        2 references
        public void setDoanhThu(ThongKeStratery tk)
        {
            this.tk = tk;
        }

        2 references
        public int getDoanhThu(Time time)
        {
            return tk.totalSales(time);
        }
    }
}
```

Hình 2.10 Code áp dụng Strategy pattern (5)

```
using DAO;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Stratery
{
    1 reference
    public class ThongKeNam : ThongKeStratery
    {
        2 references
        public int totalSales(Time time)
        {
            int nam = time.Nam;
            return SalesReportController.Instance.getTotalSalesOfYear(nam);
        }
    }
}
```

Hình 2.11 Code áp dụng Strategy pattern (6)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Stratery
{

    public interface ThongKeStratery
    {

        int totalSales(Time time);
    }
}
```

Hình 2.12 Code áp dụng Strategy pattern (7)

```
using DAO;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Stratery
{
    1 reference
    public class ThongKeThang : ThongKeStratery
    {
        2 references
        public int totalSales(Time time)
        {
            int thang = time.Thang;
            int nam = time.Nam;

            return SalesReportController.Instance.getTotalSalesOfMonth(thang,nam);
        }
    }
}
```

Hình 2.13 Code áp dụng Strategy pattern (8)

```
namespace Pattern.Stratery
{
    8 references
    public class Time
    {
        private int thang;
        private int nam;

        1 reference
        public Time(int thang, int nam)
        {
            this.thang = thang;
            this.nam = nam;
        }

        1 reference
        public Time(int nam)
        {
            this.nam = nam;
        }

        1 reference
        public int Thang { get => thang; set => thang = value; }
        2 references
        public int Nam { get => nam; set => nam = value; }
    }
}
```

Hình 2.14 Code áp dụng Strategy pattern (9)

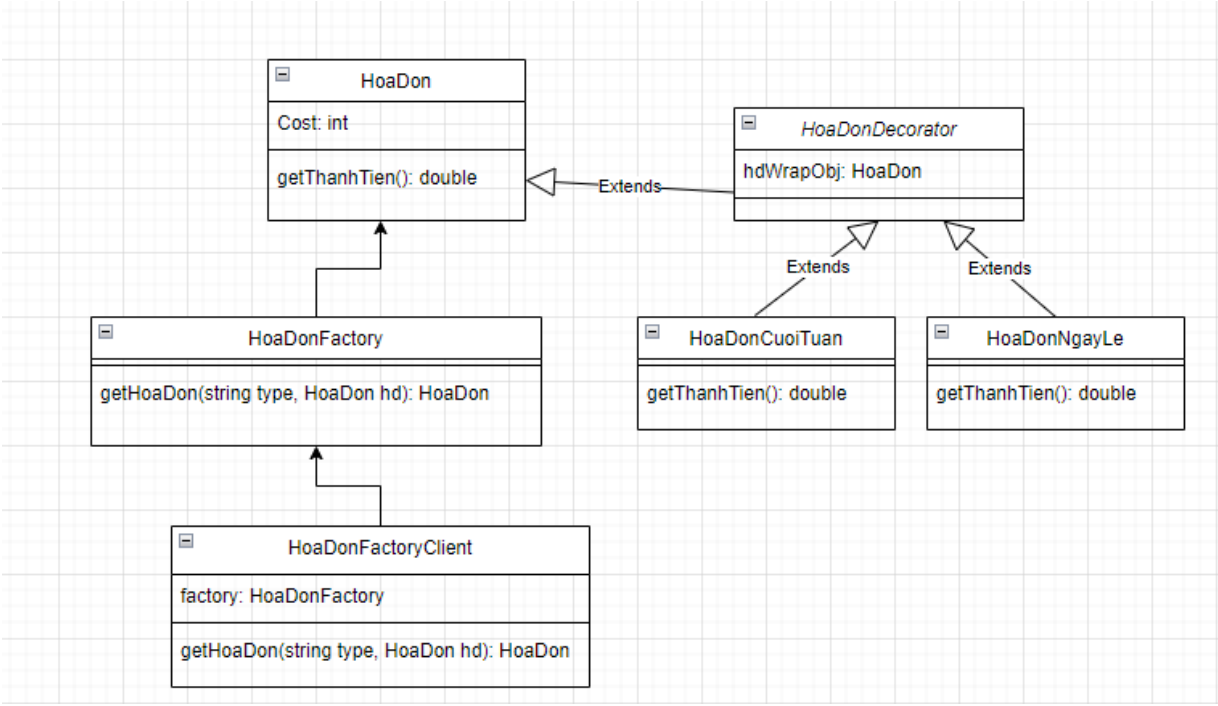
2.3 Simple Factory

2.3.1 Lý do áp dụng

Trong quá trình hoạt động, hệ thống cần tạo ra nhiều loại đối tượng khuyến mãi, trong dự án ban đầu không áp dụng Simple Factory, các đối tượng đó được tạo ra trực tiếp trong mã nguồn khiến cho mã nguồn trở nên phức tạp, khiến cho hệ thống trở nên khó bảo trì hoặc mở rộng.

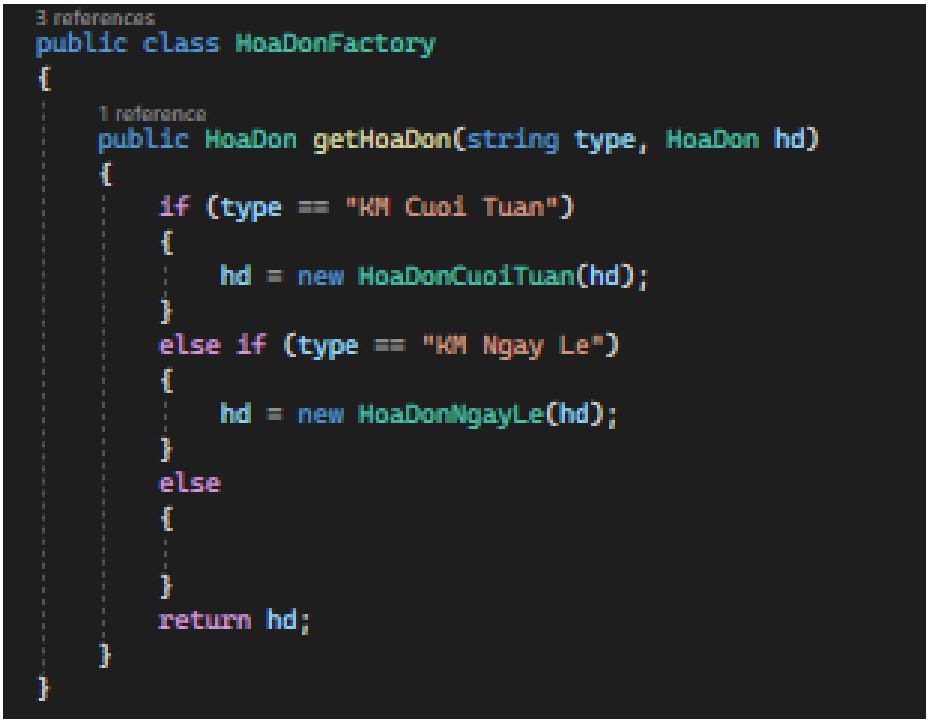
Vì vậy chúng em đã áp dụng Simple Factory vào dự án để có thể tạo ra các đối tượng khuyến mãi một cách linh hoạt hơn, cũng như dễ dàng kiểm thử và mở rộng hơn. Ngoài ra, việc áp dụng Simple Factory cũng hỗ trợ cho việc phát triển Decorator Pattern trong dự án.

2.3.2 Sơ đồ lớp



Hình 2.15 Sơ đồ lớp Simple factory

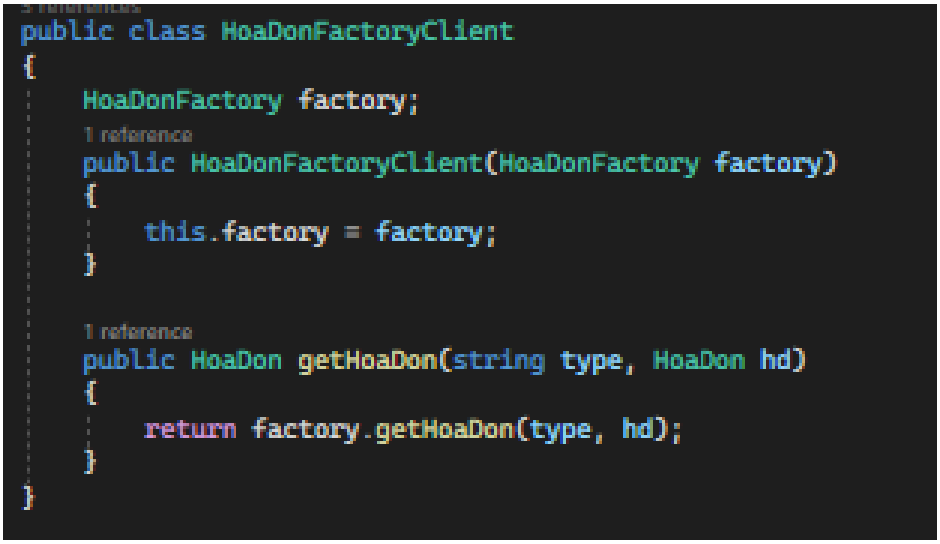
2.3.3 Code áp dụng mẫu thiết kế:



```
3 references
public class HoaDonFactory
{
    1 reference
    public HoaDon getHoaDon(string type, HoaDon hd)
    {
        if (type == "KM Cuoi Tuan")
        {
            hd = new HoaDonCuoiTuan(hd);
        }
        else if (type == "KM Ngay Le")
        {
            hd = new HoaDonNgayLe(hd);
        }
        else
        {

        }
        return hd;
    }
}
```

Hình 2.16 Code áp dụng Simple factory (1)



```
public class HoaDonFactoryClient
{
    HoaDonFactory factory;
    1 reference
    public HoaDonFactoryClient(HoaDonFactory factory)
    {
        this.factory = factory;
    }

    1 reference
    public HoaDon getHoaDon(string type, HoaDon hd)
    {
        return factory.getHoaDon(type, hd);
    }
}
```

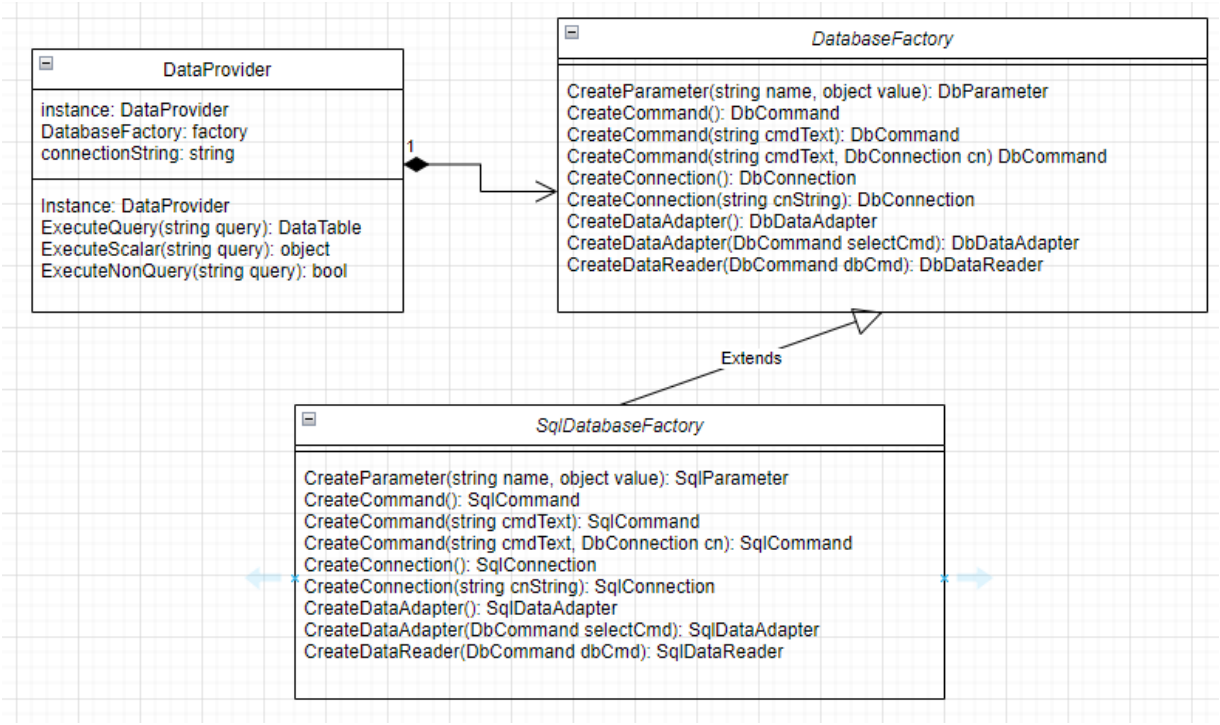
Hình 2.17 Code áp dụng Simple factory (2)

2.4 Abstract Factory

2.4.1 Lý do áp dụng

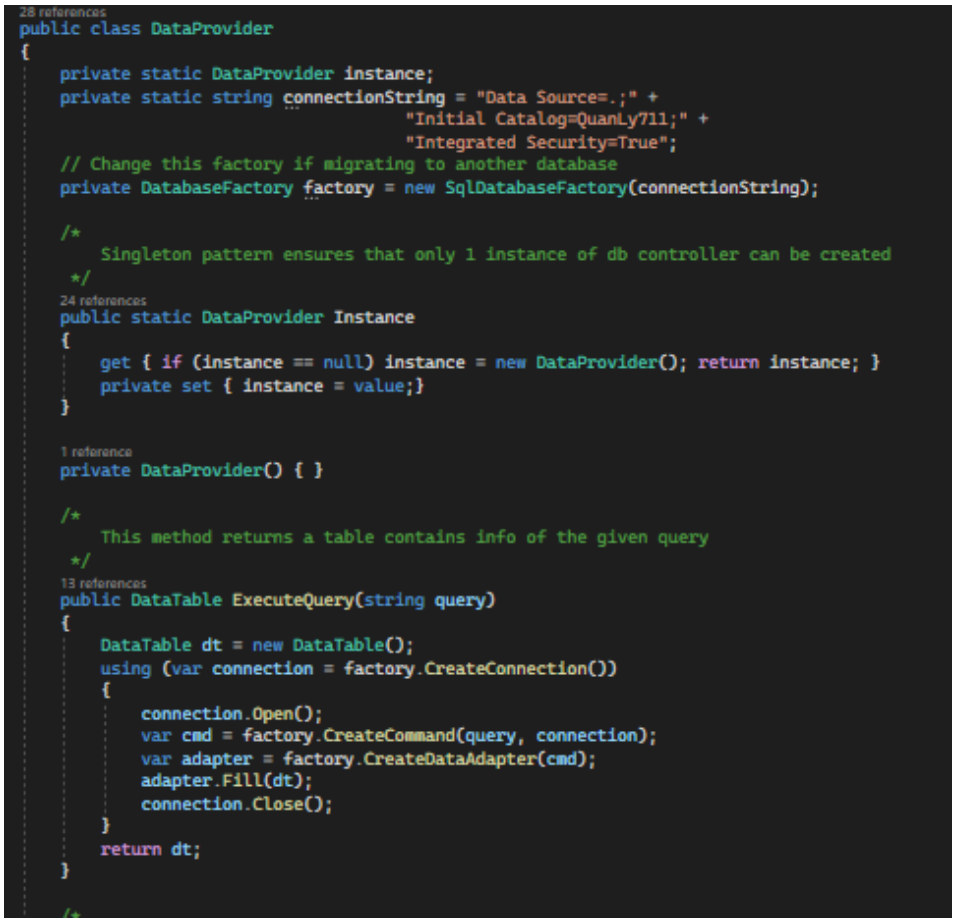
Trong dự án phần mềm, chúng em đã sử dụng Abstract Factory để giảm thiểu việc lặp lại các dòng code và tập trung tất cả các thao tác liên quan đến cơ sở dữ liệu vào một lớp. Đồng thời giúp việc chuyển đổi sang một hệ cơ sở dữ liệu khác dễ dàng hơn.

2.4.2 Sơ đồ lớp



Hình 2.18 Sơ đồ lớp Abstract factory

2.4.3 Code áp dụng mẫu thiết kế:



```
28 references
public class DataProvider
{
    private static DataProvider instance;
    private static string connectionString = "Data Source=.;" +
                                             "Initial Catalog=QuanLy711;" +
                                             "Integrated Security=True";
    // Change this factory if migrating to another database
    private DatabaseFactory factory = new SqlDatabaseFactory(connectionString);

    /*
        Singleton pattern ensures that only 1 instance of db controller can be created
     */
    24 references
    public static DataProvider Instance
    {
        get { if (instance == null) instance = new DataProvider(); return instance; }
        private set { instance = value;}
    }

    1 reference
    private DataProvider() { }

    /*
        This method returns a table contains info of the given query
     */
    13 references
    public DataTable ExecuteQuery(string query)
    {
        DataTable dt = new DataTable();
        using (var connection = factory.CreateConnection())
        {
            connection.Open();
            var cmd = factory.CreateCommand(query, connection);
            var adapter = factory.CreateDataAdapter(cmd);
            adapter.Fill(dt);
            connection.Close();
        }
        return dt;
    }

    /*
```

Hình 2.19 Code áp dụng Abstract factory (1)

```
3 references
public class SqlDatabaseFactory : DatabaseFactory
{
    private string cnString;
    0 references
    public SqlDatabaseFactory() { this.cnString = ""; }
    1 reference
    public SqlDatabaseFactory(string cnString)
    {
        this.cnString = cnString;
    }
    1 reference
    public override DbCommand CreateCommand()
    {
        return new SqlCommand();
    }

    1 reference
    public override DbCommand CreateCommand(string cmdText)
    {
        return new SqlCommand(cmdText);
    }

    4 references
    public override DbCommand CreateCommand(string cmdText, DbConnection cn)
    {
        return new SqlCommand(cmdText, (SqlConnection)cn);
    }

    4 references
    public override DbConnection CreateConnection()
    {
        if (this.cnString != null && this.cnString != "")
            return new SqlConnection(cnString);
        return new SqlConnection();
    }

    1 reference
    public override DbConnection CreateConnection(string cnString)
```

Hình 2.20 Code áp dụng Abstract factory (2)

```
public abstract class DatabaseFactory
{

    public abstract DbParameter CreateParameter(string name, object value);

    public abstract DbCommand CreateCommand();

    public abstract DbCommand CreateCommand(string cmdText);

    public abstract DbCommand CreateCommand(string cmdText, DbConnection cn);

    public abstract DbConnection CreateConnection();
    1 reference
    public abstract DbConnection CreateConnection(string cnString);
    1 reference
    public abstract DbDataAdapter CreateDataAdapter();
    2 references
    public abstract DbDataAdapter CreateDataAdapter(DbCommand selectCmd);
    1 reference
    public abstract DbDataReader CreateDataReader(DbCommand dbCmd);
}
```

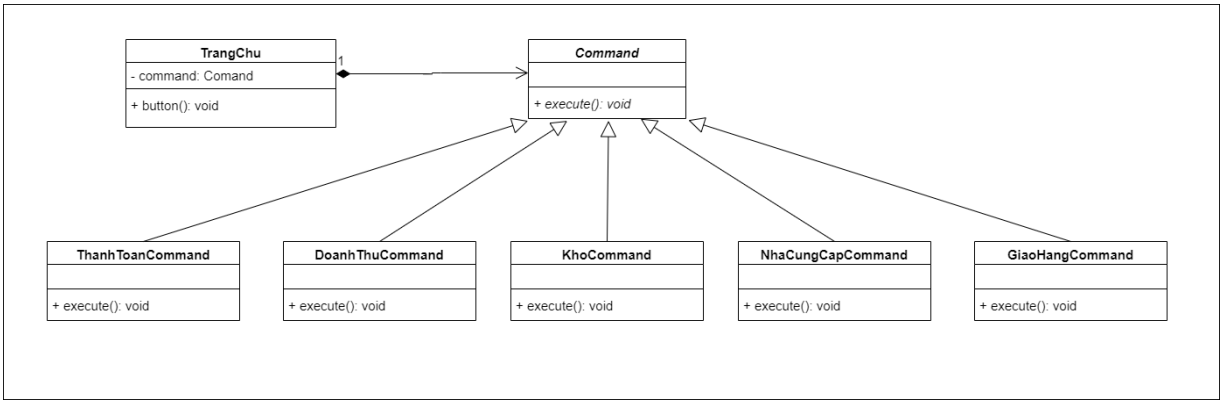
Hình 2.21 Code áp dụng Abstract factory (3)

2.5 Command Pattern

2.5.1 Lý do áp dụng

Trong quá trình hoạt động, phần mềm thường gặp khó khăn trong việc quản lý các yêu cầu và các hoạt động được thực hiện khi yêu cầu được gửi đến.Vì vậy chúng em sử dụng Command pattern để tăng tính linh hoạt của ứng dụng giúp tách biệt yêu cầu và hoạt động được thực hiện, giúp cho mã nguồn ứng dụng trở nên dễ dàng bảo trì hơn.

2.5.2 Sơ đồ lớp



Hình 2.22 Sơ đồ Command pattern

2.5.3 Code áp dụng mẫu thiết kế:

```
namespace Pattern.Command
{
    10 references
    public abstract class Command
    {
        10 references
        public abstract void execute();
    }

    2 references
    public class ThanhToanCommand : Command
    {

        1 reference
        public ThanhToanCommand()
        {

        }
        6 references
        public override void execute()
        {
            FormLoading thanhtoan = new ThanhToanForm();
            thanhtoan.loadForm();
        }
    }
```

Hình 2.23 Code áp dụng Command pattern

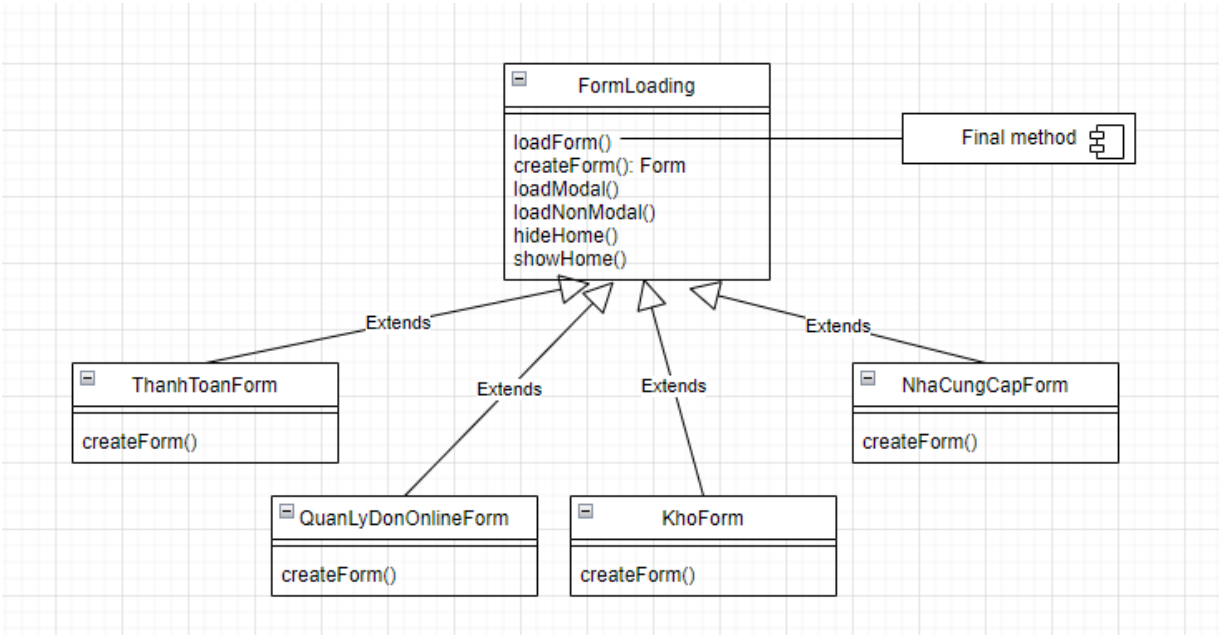
2.6 Template Method

2.6.1 Lý do áp dụng

Trong quá trình hoạt động của phần mềm, mỗi khi load một giao diện, hệ thống cần phải chạy một khối lượng code lặp lại, điều này ảnh hưởng đến hiệu suất của phần mềm và dễ gây ra lỗi.

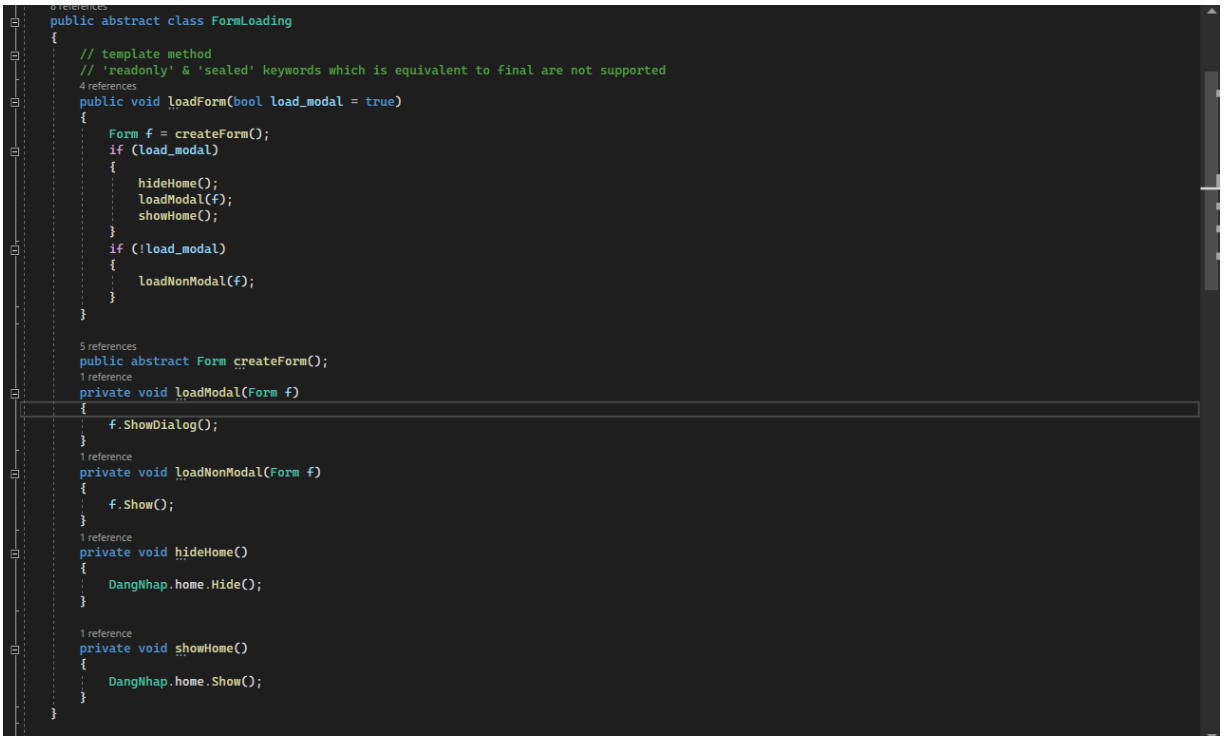
Vì vậy chúng em sử dụng Template Method để giúp giảm thiểu code lặp lại, tăng tính tái sử dụng, tránh các lỗi không đáng có và giúp dễ bảo trì mã nguồn hơn.

2.6.2 Sơ đồ lớp



Hình 2.24 Sơ đồ lớp Template Method

2.6.3 Code áp dụng mẫu thiết kế:

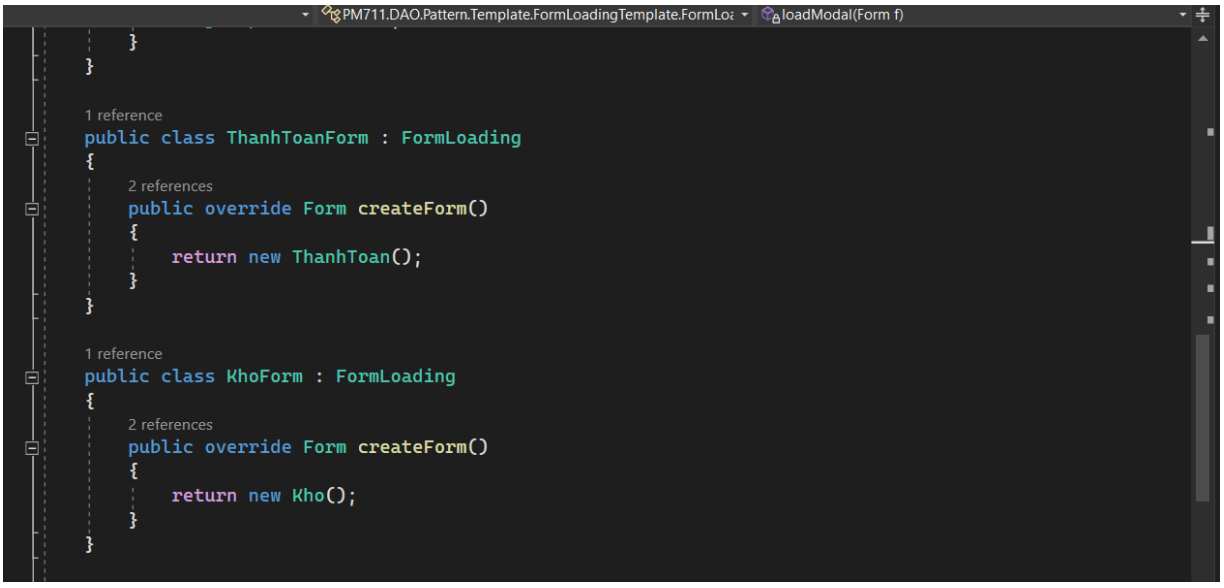


```
public abstract class FormLoading
{
    // template method
    // 'readonly' & 'sealed' keywords which is equivalent to final are not supported
    4 references
    public void loadForm(bool load_modal = true)
    {
        Form f = createForm();
        if (load_modal)
        {
            hideHome();
            loadModal(f);
            showHome();
        }
        if (!load_modal)
        {
            loadNonModal(f);
        }
    }

    5 references
    public abstract Form createForm();
    1 reference
    private void loadModal(Form f)
    {
        f.ShowDialog();
    }
    1 reference
    private void loadNonModal(Form f)
    {
        f.Show();
    }
    1 reference
    private void hideHome()
    {
        DangNhap.home.Hide();
    }

    1 reference
    private void showHome()
    {
        DangNhap.home.Show();
    }
}
```

Hình 2.25 Code áp dụng Template Method (1)



```
PM711.DAO.Pattern.Template.FormLoadingTemplate.FormLoa
loadModal(Form f)
    }
}

1 reference
public class ThanhToanForm : FormLoading
{
    2 references
    public override Form createForm()
    {
        return new ThanhToan();
    }
}

1 reference
public class KhoForm : FormLoading
{
    2 references
    public override Form createForm()
    {
        return new Kho();
    }
}
```

Hình 2.26 Code áp dụng Template method (2)

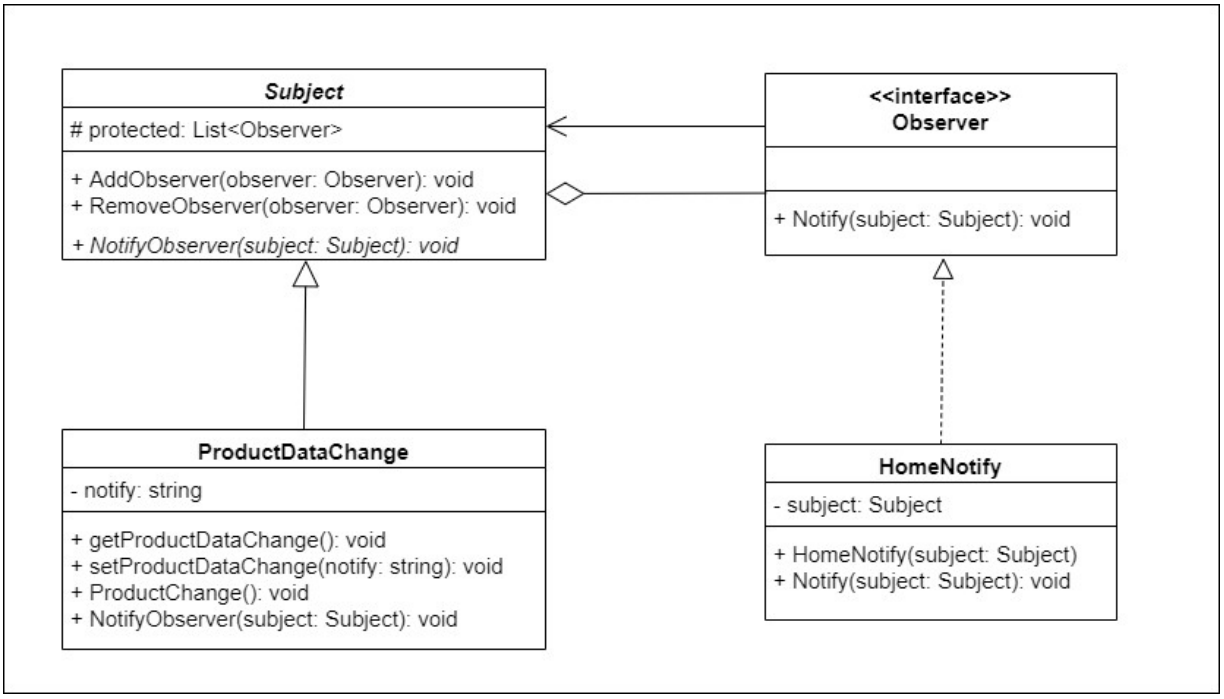
2.7 Observer Pattern

2.7.1 Lý do áp dụng

Trong dự án ban đầu, khi thêm sản phẩm mới vào kho, thì nhân viên sử dụng hệ thống sẽ không được thông báo tự động mà phải tự truy cập vào kho để kiểm tra, dẫn tới các thiếu sót không đáng có.

Vì vậy chúng em đã áp dụng observer pattern, để khi có sản phẩm mới được thêm vào kho, thì hệ thống sẽ tự động gửi thông báo đến trang chủ cho nhân viên, giúp đảm bảo quá trình làm việc của nhân viên được thuận tiện và hiệu quả, cũng như mở rộng sự tương tác của các đối tượng trong phần mềm.

2.7.2 Sơ đồ lớp



Hình 2.27 Sơ đồ Observer pattern

2.7.3 Code áp dụng mẫu thiết kế:

```
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Observer
{
    2 references
    public class HomeNotify : Observer
    {
        private Subject subject;
        1 reference
        public HomeNotify(Subject subject)
        {
            this.subject = subject;
            this.subject.AddObserver(this);
        }

        2 references
        public void Notify(Subject subject)
        {
            if (subject is ProductDataChange productNumber)
                productNumber.getProductDataChange();
        }
    }
}
```

Hình 2.28 Code áp dụng Observer pattern (1)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace PM711.DAO.Pattern.Observer
{
    5 references
    public interface Observer
    {
        2 references
        void Notify(Subject subject);
    }
}
```

Hình 2.29 Code áp dụng Observer pattern (2)

```
3 references
public class ProductDataChange : Subject
{
    private string notify;

    1 reference
    public void getProductDataChange()
    {
        DangNhap.home.Show();

        MessageBox.Show(notify, "Thông báo", MessageBoxButtons.OK);
    }

    1 reference
    public void setProductDataChange(string notify)
    {
        this.notify = notify;
        ProductChange();
    }

    1 reference
    public void ProductChange()
    {
        NotifyObserver(this);
    }

    2 references
    public override void NotifyObserver(Subject subject)
    {
        foreach (var obs in observer)
        {
            obs.Notify(subject);
        }
    }
}
```

Hình 2.30 Code áp dụng Observer pattern (3)

```
using System.Text;
using System.Threading.Tasks;

namespace PM711.DAO.Pattern.Observer
{
    7 references
    public abstract class Subject
    {
        protected List<Observer> observer = new List<Observer>();

        1 reference
        public void AddObserver(Observer observer)
        {
            this.observer.Add(observer);
        }

        0 references
        public void RemoveObserver(Observer observer)
        {
            this.observer.Remove(observer);
        }

        2 references
        public abstract void NotifyObserver(Subject subject);
    }
}
```

Hình 2.31 Code áp dụng Observer pattern (4)

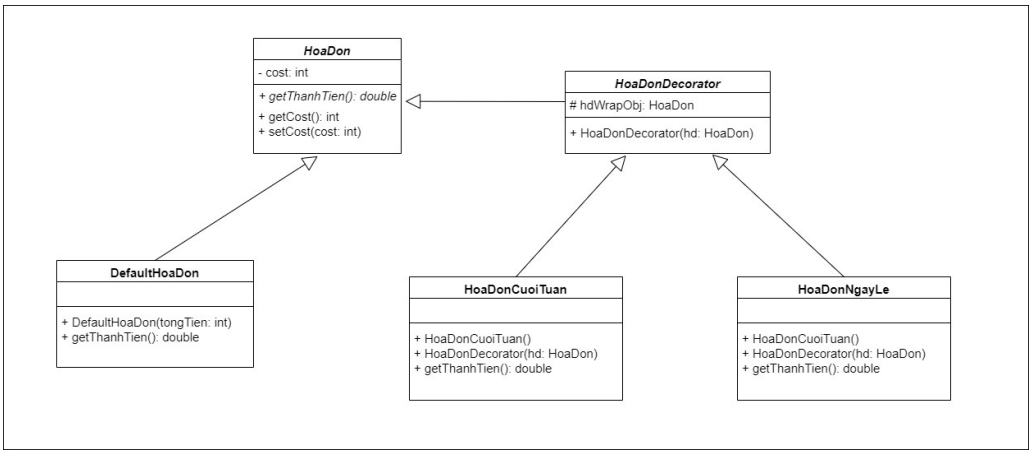
2.8 Decorator Pattern

2.8.1 Lý do áp dụng

Trong chức năng thanh toán của hệ thống, đôi khi chúng ta cần thêm các đối tượng khuyến mãi vào hóa đơn, nếu không sử dụng Decorator pattern, việc thêm các đối tượng đó vào hóa đơn sẽ phức tạp, tốn nhiều công sức hay khó mở rộng.

Vì vậy chúng em áp dụng mẫu thiết kế Decorator để giúp cho các đối tượng khuyến mãi có thể thêm vào hóa đơn linh hoạt hơn, đa dạng và dễ dàng mở rộng hơn.

2.8.2 Sơ đồ lớp



Hình 2.32 Sơ đồ lớp Decorator pattern

2.8.3 Code áp dụng mẫu thiết kế:

```
namespace Pattern.Decorator
{
    2 references
    public class DefaultHoaDon : HoaDon
    {
        1 reference
        public DefaultHoaDon(int tongTien)
        {
            base.Cost = tongTien;
        }


        9 references
        public override double getThanhTien()
        {
            return Cost;
        }
    }
}
```

Hình 2.33 Code áp dụng Decorator pattern (1)

```
using DTO;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Decorator
{

    public abstract class HoaDon
    {
        private int cost;


        public int Cost { get => cost; set => cost = value; }


        public abstract double getThanhTien();
    }
}
```

Hình 2.34 Code áp dụng Decorator pattern (2)

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Decorator
{
    2 references
    public class HoaDonCuoiTuan : HoaDonDecorator
    {
        1 reference
        public HoaDonCuoiTuan(HoaDon hd) : base(hd)
        {
        }

        9 references
        public override double getThanhTien()
        {
            // giam gia 10%
            return hdWrapObj.getThanhTien() - hdWrapObj.getThanhTien() * 0.
        }
    }
}
```

Hình 2.35 Code áp dụng Decorator pattern (3)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Decorator
{

    public abstract class HoaDonDecorator : HoaDon
    {
        protected HoaDon hdWrapObj;


        public HoaDonDecorator(HoaDon hd)
        {
            this.hdWrapObj = hd;
        }
    }
}
```

Hình 2.36 Code áp dụng Decorator pattern (4)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pattern.Decorator
{
    2 references
    public class HoaDonNgayLe : HoaDonDecorator
    {
        1 reference
        public HoaDonNgayLe(HoaDon hd) : base(hd)
        {
        }
        9 references
        public override double getThanhTien()
        {
            // giam gia 20%
            return hdWrapObj.getThanhTien() - hdWrapObj.getThanhTien() * 0.
        }
    }
}
```

Hình 2.37 Code áp dụng Decorator pattern (5)

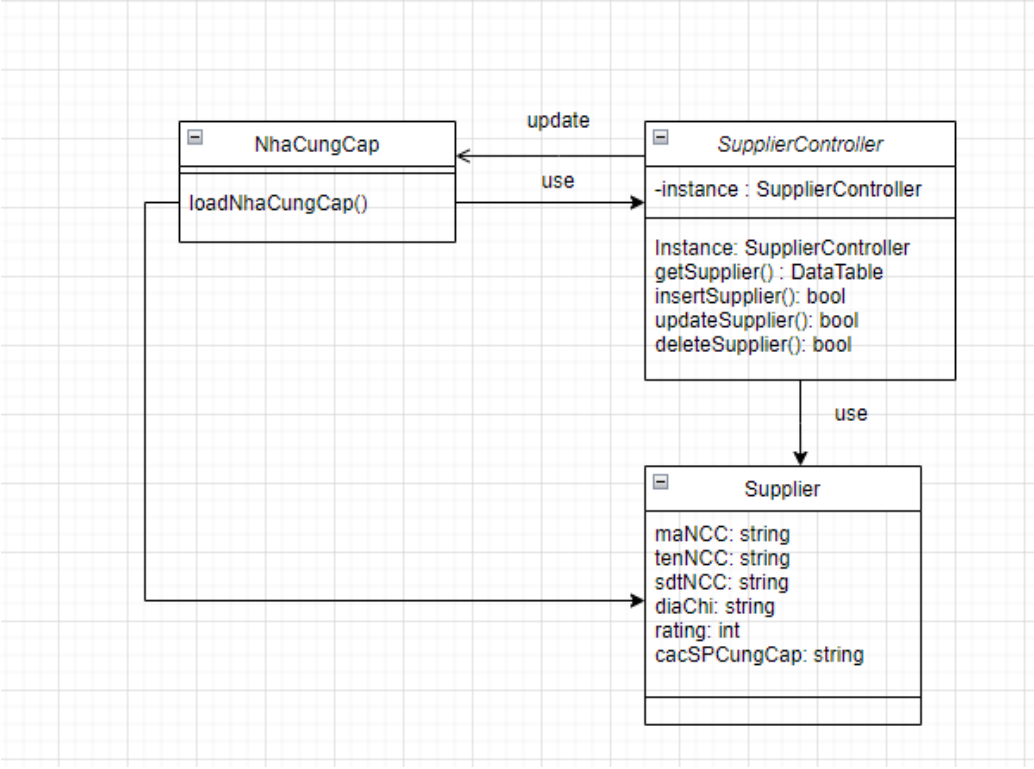
2.9 MVC Pattern

2.9.1 Lý do áp dụng

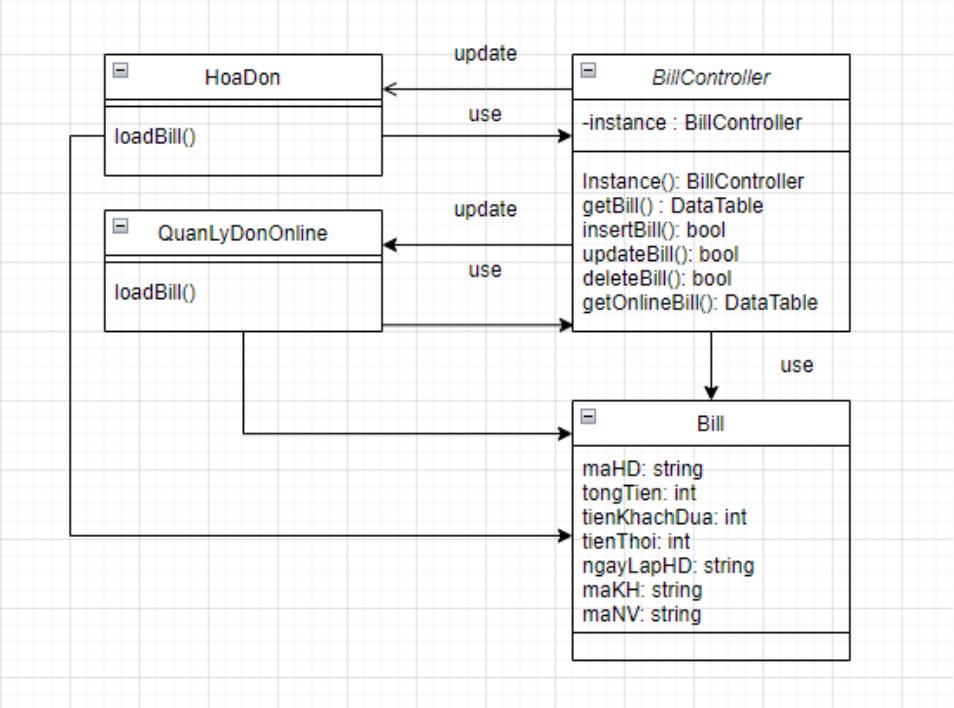
Dự án ban đầu không sử dụng mô hình MVC, khiến cho các thành phần của phần mềm không được phân tách rõ ràng, mã nguồn phức tạp, khó bảo trì và mở rộng phần mềm, làm giảm hiệu suất hoạt động của ứng dụng và dễ dẫn đến các lỗi không mong muốn.

Vì vậy chúng em đã áp dụng mô hình MVC vào dự án để giúp phân tách rõ ràng các thành phần của phần mềm, đơn giản hóa mã nguồn, tăng hiệu suất hoạt động phần mềm, giảm các lỗi ngoài ý muốn và làm cho phần mềm trở nên dễ bảo trì và mở rộng

2.9.2 Sơ đồ lớp



Hình 2.38 Một trong những sơ đồ lớp MVC (1)



Hình 2.39 Một trong những sơ đồ lớp MVC (2)

2.9.3 Code áp dụng mẫu thiết kế:

```
4 references
public class Bill
{
    private string maHD;
    private int tongTien;
    private int tienKhachDua;
    private int tienThoi;
    private string ngayLapHD;
    private string maKH;
    private string maNV;
    private double giamGia = 0;

    0 references
    public Bill() { }

    1 reference
    public Bill(int tongTien, int tienKhachDua, int tienThoi, string ngayLapHD, string maKH, string maNV, double giamGia)
    {
        this.tongTien = tongTien;
        this.tienKhachDua = tienKhachDua;
        this.tienThoi = tienThoi;
        this.ngayLapHD = ngayLapHD;
        this.maKH = maKH;
        this.maNV = maNV;
        this.GiamGia = giamGia;
    }
```

Hình 2.40 Code áp dụng MVC (1)

```
3 references
public class BillController
{
    private static BillController instance;

    2 references
    public static BillController Instance
    {
        get { if (instance == null) { instance = new BillController(); } return instance; }
        private set { instance = value; }
    }

    0 references
    public DataTable getBill()
    {
        // view all bills method
        string query = "select * from HOADON";
        DataTable dt = DataProvider.Instance.ExecuteQuery(query);
        return dt;
    }

    1 reference
    public bool insertBill(int tongTien, int tienKhachDua, int tienThoi, string ngayLap, string maKH, string maNV)
    {
        // insert method
        string query = "insert into HOADON values('', '" + tongTien + "', '" + tienKhachDua + "', '" + tienThoi + "', '" + ngayLap
        return DataProvider.Instance.ExecuteNonQuery(query);
    }
```

Hình 2.41 Code áp dụng MVC (2)

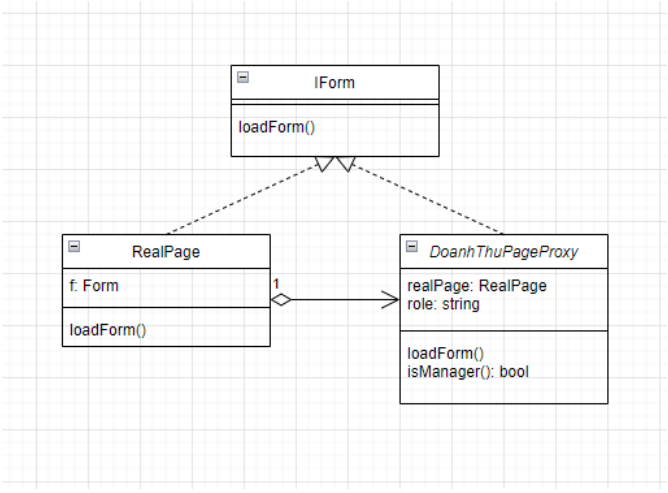
2.10 Proxy pattern

2.10.1 Lý do áp dụng

Trong phần mềm ban đầu, việc phân quyền giữa nhân viên và quản lý chưa được thực hiện rõ ràng, nên đôi khi xảy ra các lỗi về bảo mật.

Vì vậy chúng em sử dụng Proxy để hiện thực chức năng phân quyền cho hệ thống, chỉ cho phép quản lý được phép xem doanh thu của cửa hàng. Giúp cho phần mềm tăng tính bảo mật đồng thời sẵn sàng cho các chức năng cần tới phân quyền sau này.

2.10.2 Sơ đồ lớp



Hình 2.42 Sơ đồ lớp Proxy pattern

2.10.3 Code áp dụng mẫu thiết kế:

```
public class DoanhThuPageProxy : IForm
{
    private RealPage realPage;
    private string role;

    1 reference
    public DoanhThuPageProxy(string role)
    {
        this.role = role;
    }

    2 references
    public void loadForm()
    {
        if (isManager())
        {
            if (realPage == null) { realPage = new RealPage(new DoanhThu()); }
            realPage.loadForm();
        }
        else { MessageBox.Show("Không đủ quyền truy cập"); }
    }

    // phan quyen
    1 reference
    private bool isManager()
    {
        if (this.role == "Quản lý" || this.role == "Admin")
        {
            return true;
        }
        return false;
    }
}
```

Hình 2.43 Code áp dụng Proxy pattern (1)

```
3 references
public interface IForm
{
    4 references
    void loadForm();
}


//
// real service (Form)
//
3 references
public class RealPage : IForm
{
    private Form f;
    1 reference
    public RealPage(Form f)
    {
        this.f = f;
    }
    3 references
    public void loadForm()
    {
        f.Show();
    }
}
```

Hình 2.44 Code áp dụng Proxy pattern (2)

TÀI LIỆU THAM KHẢO

- Strategy Design Pattern: Áp dụng vào code (C#) https://www.youtube.com/watch?v=rXS6q0pG9Ow&list=PLoaAbmGPgTSOrVuxwbnDJ14U9J6CXJXUk&index=4
- Observer Design Pattern: Áp dụng vào Code (C#) https://www.youtube.com/watch?v=zc2smrSk5I0&list=PLoaAbmGPgTSOrVuxwbnDJ14U9J6CXJXUk&index=7
- Decorator Design Pattern: Áp dụng vào code, thử vài trường hợp (C#) https://www.youtube.com/watch?v=U2XUrAefHME&list=PLoaAbmGPgTSOrVuxwbnDJ14U9J6CXJXUk&index=13
- Proxy Design Pattern - Trợ thủ đắc lực của Developers https://viblo.asia/p/proxy-design-pattern-tro-thu-dac-luc-cua-developers-RQqKLB2bl7z
- Design Patterns - Singleton pattern https://viblo.asia/p/design-patterns-singleton-pattern-maGK7zra5j2