Министерство образования и науки Российской Федерации

Севастопольский государственный университет

Кафедра ИС

Отчет

По дисциплине: “Рефакторинг программного обеспечения”

Лабораторная работа №6

“Исследование способов применения поведенческих паттернов

проектирования при рефакторинге ПО”

Выполнил:

ст.гр. ИС/б-17-2

Долженко И.А.

Проверил:

Петраков В.А.

Севастополь

2020

1 ЦЕЛЬ РАБОТЫ

Исследовать возможность использования поведенческих паттернов проектирования. Получить практические навыки применения паттернов поведения при объектно-ориентированном проектировании и рефакторинге ПО.

2 ХОД РАБОТЫ

Представим, что мы делаем систему для уведомлений в зависимости от приоритетов входных задач. Фрагмент кода данной системы представлен ниже:

static void Main(string[] args)

{

Priority[] arrayOfPriority = { Priority.ROUTINE, Priority.IMPORTANT, Priority.ASAP };

foreach (Priority priority in arrayOfPriority)

{

if (priority == Priority.ROUTINE)

{

Console.WriteLine("Sending simple report.");

}

else if (priority == Priority.IMPORTANT)

{

Console.WriteLine("Sending email notification.");

}

else if (priority == Priority.ASAP)

{

Console.WriteLine("Sending SMS notification.");

} else if (......)

{

....

}

}

Console.ReadKey();

}

enum Priority

{

ROUTINE,

IMPORTANT,

ASAP

}

Со временем добавляются новые приоритеты задачам и фрагмент кода с условными операторами становится огромным. При изменении одного правила приходится трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось бы продублировать их код в других классах.

Воспользуемся паттерном “Цепочка обязанностей” и перенесем каждую проверку в отдельный класс. Каждый из них будет иметь ссылку на следующий объект в цепи:

static void Main(string[] args)

{

Notifier reportNotifier = new ReportNotifier();

Notifier emailNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifier();

reportNotifier.SetNext(emailNotifier);

emailNotifier.SetNext(smsNotifier);

reportNotifier.NotifyManager("Sending simple report.", Priority.ROUTINE);

reportNotifier.NotifyManager("Sending email notification.", Priority.IMPORTANT);

reportNotifier.NotifyManager("Sending SMS notification.", Priority.ASAP);

Console.ReadKey();

}

enum Priority

{

ROUTINE,

IMPORTANT,

ASAP

}

abstract class Notifier

{

protected Notifier successor;

public void SetNext(Notifier successor)

{

this.successor = successor;

}

public abstract void NotifyManager(string message, Priority level);

}

class ReportNotifier : Notifier

{

public override void NotifyManager(string message, Priority level)

{

if (level == Priority.ROUTINE)

{

Console.WriteLine(message);

}

else if (successor != null)

{

successor.NotifyManager(message, level);

}

}

}

class EmailNotifier : Notifier

{

public override void NotifyManager(string message, Priority level)

{

if (level == Priority.IMPORTANT)

{

Console.WriteLine(message);

}

else if (successor != null)

{

successor.NotifyManager(message, level);

}

}

}

class SMSNotifier : Notifier

{

public override void NotifyManager(string message, Priority level)

{

if (level == Priority.ASAP)

{

Console.WriteLine(message);

}

else if (successor != null)

{

successor.NotifyManager(message, level);

}

}

}

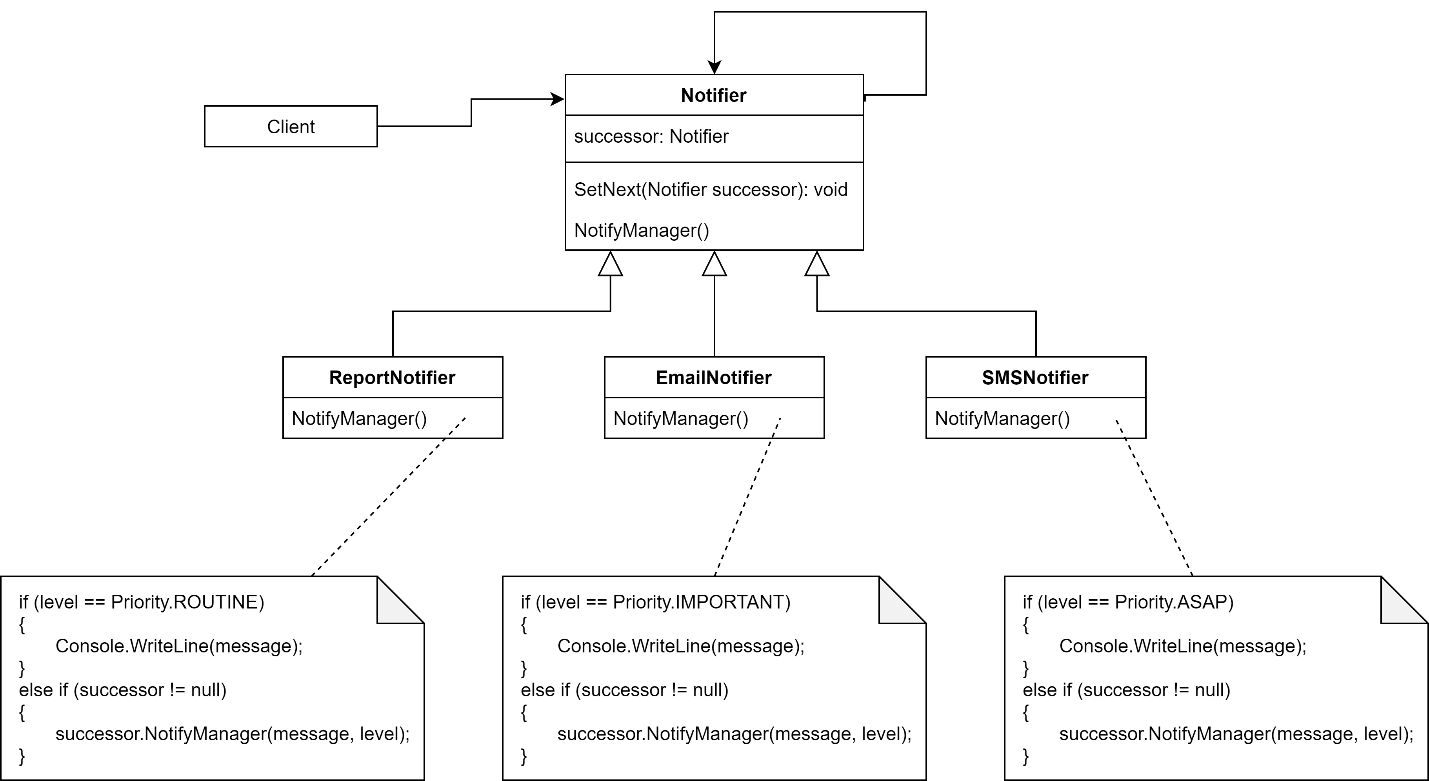


Рисунок 1 – Диаграмма классов после применения паттерна

ВЫВОДЫ

В ходе выполнения лабораторной работы была исследована возможность использования поведенческих паттернов проектирования. Получены практические навыки применения паттернов поведения при объектно-ориентированном проектировании и рефакторинге ПО.

Был применен паттерн “Цепочка обязанностей”, что позволило избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. А также связывать объекты-получатели в цепочку и передавать запрос вдоль этой цепочки, пока его не обработают.