УДК 004.04

ENUM И SWITCH, И ПРОБЛЕМЫ ИХ ИСПОЛЬЗОВАНИЯ

Д.В. СУЩЕВСКИЙ
(Представлено: канд. техн. наук, доц. И.Б. БУРАЧЕНОК)

Представлен способ решения проблемы использования напрямую типов перечислений в операторе switch для разработчиков, использующих С-подобные языки программирования.

Очень часто разработчикам использующих С-подобные языки, приходится сталкиваться с проблемой не обработанных `CASE`-условий в `switch`. Такие ошибки могут нести огромные убытки для компаний и надёжности самого программного обеспечения (ПО). С появление функциональных языков программирования люди начали понимать, что на многие вопросы стоит посмотреть под другим углом, чтобы увидеть проблему или преимущества.

Для решения проблем использования типов перечислений в операторе `switch`, в функциональных парадигмах используются размеченные объединения (“discrimination union”) и код просто не будет компилироваться, пока не будут рассмотрены все возможные варианты из типа, в то время как, например, C# вынуждает разработчика использовать условие `DEFAULT` чтобы точно обработать все условия, так как `enum` не строго типизирован и можно любое число преобразовать в `enum`, что приносит много проблем в дальнейшей перспективе.

С такими проблемами можно столкнуться. Где возникает такая необходимость? Чаще всего это работа с каким-либо web Application Programming Interface (API). В ситуации, когда к нам приходит сущность с фиксированным количеством полей, а уже понимать какие поля заполнены или актуальны для данной сущности, можно лищь по какому-то типу, который находится в объекте. Тогда код будет обрастать оператором switch и соответствующей логикой обработки актуальных типов на данный момент. Всё может усугубить то, что не весь код обработки таких сущностей находится в одном проекте и по банальной человеческой невнимательности можно забыть доработать код, в других проектах, что повлечет ошибочные ситуации и трудно обнаруживаемые ошибки.

В данной статье хотелось бы поделиться возможными решениями данной проблемы. Ниже рассмотрим реальные примеры на языке программирования C#. Если проблема состоит только в `switch` операторе и отслеживании новых типов, тогда необходимо избавиться от них. Идея состоит в том, чтобы заменить использование `switch` паттерном `visitor`.

Пример 1. Предположим, что у нас есть какой-то API (Application Programming Interface) для работы с документами, от которого мы получаем необходимые данные и определяем его тип, а далее в зависимости от этого типа, необходимо делать различные операции (см. рисунок 1).

```
public enum DocumentType
{
    Invoice,
    PrepaymentAccount
}

public interface IDocumentVisitor<out T>
{
    T VisitInvoice();
    T VisitPrepaymentAccount();
}

public static class DocumentTypeExt
{
    public static T Accept<T>(this DocumentType self, IDocumentVisitor<T> visitor)
    {
        switch (self)
        {
            case DocumentType.Invoice:
                return visitor.VisitInvoice();
            case DocumentType.PrepaymentAccount:
                return visitor.VisitPrepaymentAccount();
            default:
                throw new ArgumentOutOfRangeException(nameof(self), self, null);
        }
    }
}
```

Рисунок 1. – Пример файла `DocumentType.cs`

Предложено определять все связанные типы в одном файле, как показано на рисунке 2, что не является идиоматичным для .Net разработчика, однако иногда это очень упрощает понимание кода.

```
public class DatabaseSearchVisitor : IDocumentVisitor<IDocument>
{
    private ApiId _id;
    private Database _db;

    public DatabaseSearchVisitor(ApiId id, Database db)
    {
        _id = id;
        _db = db;
    }

    public IDocument VisitInvoice() => _db.SearchInvoice(_id);
    public IDocument VisitPrepaymentAccount() => _db.SearchPrepaymentAccount(_id);
}
```

Рисунок 2. – `Visitor`, который производит поиск документа в базе

Пример использования DatabaseSearchVisitor показан на рисунке 3.

```
public void UpdateStatus(ApiDoc doc)
{
    var searchVisitor = new DatabaseSearchVisitor(doc.Id, _db);

    var databaseDocument = doc.Type.Accept(searchVisitor);

    databaseDocument.Status = doc.Status;

    _db.SaveChanges();
}
```

Рисунок 3. – Пример использования `DatabaseSearchVisitor.cs`

Пример 2. Предположим, что у нас имеется список событий по электронному кошельку. Их определяем из очереди в виде типа события и `json` с дополнительными полями. Определение этих типов показано на рисунке 4.

```
public enum PurseEventType
{
    Increase,
    Decrease,
    Block,
    Unlock
}

public sealed class PurseEvent
{
    public PurseEventType Type { get; }
    public string Json { get; }

    public PurseEvent(PurseEventType type, string json)
    {
        Type = type;
        Json = json;
    }
}
```

Рисунок 4. – Описание события по счету

Например, появилась необходимость отправлять уведомления пользователю на определенный тип событий. Тогда реализуем visitor следующим образом, показанным на рисунке 5.

```
public interface IPurseEventTypeVisitor<out T>
{
    T VisitIncrease();
    T VisitDecrease();
    T VisitBlock();
    T VisitUnlock();
}

public sealed class PurseEventTypeNotificationVisitor : IPurseEventTypeVisitor<Missing>
{
    private readonly INotificationManager _notificationManager;
    private readonly PurseEventParser _eventParser;
    private readonly PurseEvent _event;

    public PurseEventTypeNotificationVisitor(PurseEvent @event, PurseEventParser eventParser, I
NotificationManager notificationManager)
    {
        _notificationManager = notificationManager;
        _event = @event;
        _eventParser = eventParser;
    }

    public Missing VisitIncrease() => Missing.Value;

    public Missing VisitDecrease() => Missing.Value;

    public Missing VisitBlock()
    {
        var blockEvent = _eventParser.ParseBlock(_event);
        _notificationManager.NotifyBlockPurseEvent(blockEvent);
        return Missing.Value;
    }

    public Missing VisitUnlock()
    {
        var blockEvent = _eventParser.ParseUnlock(_event);
        _notificationManager.NotifyUnlockPurseEvent(blockEvent);
        return Missing.Value;
    }
}
```

Рисунок 5. – `Visitor` по отправке уведомлений

Для примера, ничего не будем возвращать после отправки уведомления. Для этого используется тип `Missing` из `System.Reflection` или же это можно написать, используя тип `Unit`. В реальном проекте возвращался бы `Result`, например, с информацией об ошибке, если такие имеются (см. рис. 6).

```
public void SendNotification(PurseEvent @event)
{
    var notificationVisitor = new PurseEventTypeNotificationVisitor(@event, _eventParser, _noti
ficationManager);
    @event.Type.Accept(notificationVisitor);
}
```

Рисунок 6. – Пример использования `visitor` по отправке уведомлений

Если нужно использовать такой подход там, где важна производительность, в качестве `visitor` можно использовать структуры (ключевое слово `struct`). Тогда код изменится следующим образом, как показано на рисунке 7.

```
public static T Accept<TVisitor, T>(this DocumentType self, in TVisitor visitor)
    where TVisitor : IDocumentVisitor<T>
    {
        switch (self)
        {
            case DocumentType.Invoice:
                return visitor.VisitInvoice();
            case DocumentType.PrepaymentAccount:
                return visitor.VisitPrepaymentAccount();
            default:
                throw new ArgumentOutOfRangeException(nameof(self), self, null);
        }
    }
```

Рисунок 7. – Изменение в методе расширении

Сам `visitor` остаётся прежним, достаточно изменить `class` на `struct`. Сам код обновления документа будет выглядеть не так удобно (см. рисунок 8), но работает быстро.

```
public void UpdateStatus(ApiDoc doc)
{
    var searchVisitor = new DatabaseSearchVisitor(doc.Id, _db);

    var databaseDocument = doc.Type.Accept<DatabaseSearchVisitor, IDocument>(searchVisitor);

    databaseDocument.Status = doc.Status;

    _db.SaveChanges();
}
```

Рисунок 8. – Изменение в методе обновления документа

При таком использовании `generic`, необходимо уточнять типы самому, так как компилятор не способен вывести их автоматически.

Если нужно реализовать логику только в одном месте, то часто `visitor` – громоздко и не удобно. Поэтому есть альтернативное решение `match` (см. рисунок 9).

```
public static T Match<T>(this DocumentType self, Func<T> invoiceCase, Func<T> prepaymentAccount
Case)
{
    var visitor = new FuncVisitor<T>(invoiceCase, prepaymentCase);
    return self.Accept<FuncVisitor<T>, T>(visitor);
}
```

Рисунок 9. – Пример метода match для структуры

На рисунке 10 представлена реализация visitor, который принимает реализацию логики обработки типа в виде делегата.

```
public readonly struct FuncVisitor<T> : IDocumentVisitor<T>
{
    private readonly Func<T> _invoiceCase;
    private readonly Func<T> _prepaymentAccountCase;

    public FuncVisitor(Func<T> invoiceCase, Func<T> prepaymentAccountCase)
    {
        _invoiceCase = invoiceCase;
        _prepaymentAccountCase = prepaymentAccountCase;
    }

    public T VisitInvoice() => _invoiceCase();
    public T VisitPrepaymentAccount() => _prepaymentAccountCase();
}
```

Рисунок 10. – Реализация FuncVisitor

Пример использования метода расширения `match` (см рисунок 11).

```
public void UpdateStatus(ApiDoc doc)
{
    var databaseDocument = doc.Type.Match(
        () => _db.SearchInvoice(doc.Id),
        () => _db.SearchPrepaymentAccount(doc.Id)
    );

    databaseDocument.Status = doc.Status;

    _db.SaveChanges();
}
```

Рисунок 11. – Пример использования Match

Таким образом. При добавлении нового значения в `enum` необходимо.

– добавить метод в интерфейс;

– добавить его использование в метод расширение.

Для всех реализаций представленного интерфейса необходимо реализовать логику обработки нового типа и пока это не будет сделано, проект просто не соберется. Таким образом можно избавиться от проблемы забытого `case` в `switch`.

ЛИТЕРАТУРА

1. Ploeh blog. [Электронный ресурс] / Ploeh blog. – Режим доступа: https://blog.ploeh.dk/2018/06/25/visitor-as-a-sum-type/. – Дата доступа: 16.09.2020.
2. Wikipedia. [Электронный ресурс] / Wiki. – Режим доступа: https://en.wikipedia.org/wiki/Unit_type. – Дата доступа: 10.09.2020.
3. Microsoft documentation. [Электронный ресурс] / Result – F#. – Режим доступа: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/results. – Дата доступа: 13.09.2020.