УДК 004.622

Применение объектно-ориентированных практик программирования для решения задач построения программных объектов, основанных на разборе текстовых данных

Королёнок Александр Юрьевич
Приамурский государственный университет им. Шолом-Алейхема
Студент

Аннотация
В данной статье рассмотрен способ построения программного модуля, задачей которого является создание объекта, обладающего набором сущностей, полученных путем разбора (парсинга) входных данных, предоставляемых сторонним прикладным программным интерфейсом (Web API). Данный способ реализует так называемую гибкую архитектуру, что позволяет в перспективе развития проекта, или изменения формата входных данных, без чрезмерной сложности вносить изменения в проект, гарантированно не нарушая работоспособность уже существующего кода. Для реализации метода применялась LINQPad – интерактивная программная утилита для разработки приложений и объектно-ориентированные техники программирования. Для реализации такой архитектуры приложения применялись такие паттерны проектирования как: фабричный метод (Factory Method), строитель (Builder) и адаптер (Adapter). Решение позволяет программным сущностям быть открытым для расширения, путем создания новых типов сущностей, и закрытыми для изменения. Такой подход к разработке имеет значительные преимущества в коммерческой среде, когда изменения исходного кода влекут за собой дополнительные затраты человеко-часов на рефакторинг, код-ревью, тестирование и прочие производственные расходы.
Ключевые слова: Паттерны проектирования, фабричный метод, строитель, принцип открытости-закрытости, объектно-ориентированное программирование, регулярные выражения.

Application of object-oriented programming practices for solving problems of constructing software objects based on parsing textual data

Korolenok Alexandr Yurievich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article describes a method for building a software module, the task of which is to create an object that has a set of entities obtained by parsing (parsing) the input data provided by a third-party application programming interface (Web API). This

method implements the so-called flexible architecture, which allows, in the future, the development of the project, or changes in the format of the input data, without excessive complexity to make changes to the project, guaranteed not to break the performance of the existing code. To implement the method, we used LINQPad, an interactive software utility for developing applications and object-oriented programming techniques. To implement such an application architecture, design patterns such as Factory Method, Builder, and Adapter were used. The solution allows programmatic entities to be open for extension by creating new entity types and closed for modification. This approach to development has significant advantages in a commercial environment, when changes to the source code entail additional man-hours spent on refactoring, code review, testing, and other production costs.

Keywords: Windows Management Instrumentation, patterns design, factory method, open-closed principle, Liskov substitution principle, object-oriented programming, reflection of types, information about the system.

Научный руководитель:
Глаголев Владимир Александрович
Приамурский государственный университет имени Шолом-Алейхема
к.г.н., доцент кафедры информационных систем, математики и правовой информатики

1. Введение

1.1. Актуальность исследования

Разбор текстовой информации является довольно частой задачей в коммерческой разработке программного обеспечения. Так, например, при налаживании взаимодействия между различными сервисами (как собственных, так и сторонних поставщиков) приходится организовывать обмен данными, производить импорт, анализировать разного рода информацию, зачастую не имея конкретной спецификации протокола обмена или описания формата данных. Это влечет за собой необходимость разработки средств разбора и анализа данных в условиях неопределенности. И даже имея спецификацию на момент разработки, нет никаких гарантий, что в ближайшем будущем эта спецификация не обновится, что неизбежно повлечет за собой необходимость внесения соответствующих изменений в кратчайшие сроки.

В настоящей статье описан метод решения разбора импортируемых данных от стороннего сервиса бронирования авиабилетов, предоставляемых web-сервисом авиакомпании.

Данные файлы представляют из себя позиционные текстовые данные, без строгой структуры и не имеют спецификации. Задача заключается в разработке такого решения, которое позволит осуществлять разбор импорта и конвертацию полученных данных в программные объекты. При этом, в случае изменения (равно как добавления) в текущем формате не должны

повлечь за собой необходимость внесения кардинальных исправлений в существующий, уже работающий исходный код.

1.2. Обзор исследований

Многие исследователи касались вопроса связанного с парсингом данных. А.Д. Колесов [1] описал разработку парсинг-системы бизнес-аналитики. Р.П Гудзенко и С.Г. Синица [2] в своей работе описали разработку программы для парсинга информации на многостарничных веб-ресурсах. А.Е. Кривоногова [3] описали разработку WEB-приложения для автоматизации поиска наименований коммерческой продукции. Ученные И.А. Писарев и Л.К. Бабенко [4] разработали и описали парсер на языке программирования C# для извлечения структуры криптографических протоколов из исходного кода. Исследователь А.Ю. Королёнок [5] привел пример использования инструментария управления Windows (WMI) для получения системной информации с применением техник объектно-ориентированного программирования. В.А. Кононов [6] разработал и описал программу парсинга сайтов на языке программирования Python. Д.К. Карпов [7] привел пример обработка больших данных с использованием средств языка python.

1.3. Цель исследования

Целью исследования является разработка гибкой архитектуры модуля приложения (набора классов), которая позволит расширять функциональность путем добавления новых классов, не затрагивая при этом работу уже существующего взаимодействия программных объектов.

2. Методы исследования

Для реализации поставленной задачи применим интерактивную программную утилиту LINQPad, работающую на платформе .NET Framework.

Отправной точкой разработки новой архитектуры является анализ предоставляемых данных. Web-сервис авиакомпании по запросу предоставляет ответ в виде файла, в котором находится информация о заявке на бронирование авиабилета. Эти файлы хранят в себе сведения о пассажире, дополнительных услугах, которые пассажир заказал на момент подачи заявки на бронь, сведения о рейсе, стоимости конкретных услуг, налогов и сборов, а также иная, в том числе сервисная информация. Важным обстоятельством для нас является тот факт, что сведения о любой сущности в этих файлах размещаются каждый на одной строке файла, а набор атрибутов внутри описания самой сущности разделен символами точки с запятой, пробельными символами, знаком тире и некоторыми другими знаками пунктуации. Но в некоторых сущностях эти же знаки могут иметь значения не разделителя атрибутов, а являться частью семантики.

Пример содержимого файла, разбор которого необходимо произвести:

```
FLIGHT-BLK207;7D;;157;0000000000;;001001
AMD 0000000000;1/1;
;;FLIGHT
MUC1A VGT677000;0101;MOWD528AB;40379957;;;;;MOWD528AB;40379957;;;;;;;;;;;;;;;;;;;;;D5 VGT677
A-TURKISH-AIRLINES;D5 4211
B-TTM/ET
C-9505/ 10542AS -bcg     ----
D-210818;210818;210818
G- /S;;DMENJC;RU
U-000;000EMD;D5;0421;TURKISH-AIRLINES;18AUG;D0;1;D5;;;MOW;NJC;TO-;AT-;A;C;EXCESS BAG FIRST AT CHEC;0IE;;;;;;1;P;;;;;;RUB
EMD000;000S7;0421;TURKISH-AIRLINES;18AUG;D0;1;D5;;;MOW;NJC;TO-;AT-;A;C;EXCESS BAG FIRST AT CHEC;0IE;;;;;;1;P;;;;;;RUB
ICW4212429730775E1;D0;L0
K-FRUB4000       ;RUB4000       ;;;;;;;;;;;RUB4000       ;;;
Q-MOW D5 NJC4000RUB4000END;
I-000;02STEPANOV/SERGEI VLADIMIROVICH MR(ADT)(IDPPRU2254499451);;APM-79246484898;;
FOI D5  FOID-PPRU2254499451;P2
SSR FOID D5  HK1/PPRU2254499451;P2
SSR DOCS D5  HK1/P/RUS/2254499451/RUS/17NOV75/M/22NOV31/STEPANOV/SERGEI VLADIMIROVICH;P2
SSR CTCM D5  HK1/9246484898;P2
SSR ABGS D5  HK1/UP TO 23 KG AND 203 CM BAGGAGE;S1;P2
TMCX421-9902997953;4212429730775;D0
FEINCL VAT363.64RUB
MFPCASH;D0
ENDX
```

Рисунок 1 – пример содержимого входного файла

Выходной информацией разрабатываемого модуля будет является класс, содержащий в себе некоторый целостный набор сущностей, интересующей нас информации о бронировании (заявках на бронирование), с целью выбора лучшего предложения, исходя из предпочтений клиента.

Все вышеописанное требует, во-первых, создания конкретных методов разбора для каждой конкретной сущности, а во-вторых ограничение целостности (в контексте единого представления информации выходных данных) вынуждает создавать выходной объект, не обладающий избыточностью неиспользуемых сущностей, который при этом будет возможно расширять, посредством введения в классовую структуру новых сущностей.

Определим некоторые термины для предметной области:

- Секция – любая строка файла импорта, которая представляет какую-либо сущность.
- Атрибут – атомарный элемент сущности.
- Ключ (идентификатор сущности) – паттерн регулярного выражения, по которому можно выявить однозначное сопоставление конкретной строки к конкретной сущности.
- Парсер (Parser) – основной класс, осуществляющий разбор импортируемых данных, и выдающий соответствующий этому набору результат в виде программного объекта.

Исходя из того, что результат заведомо является сложным составным объектом, состоящим из определенных сущностей, отличающихся друг от друга алгоритмом разбора своих собственных данных, лучшим способом создания такого объекта будет применение паттерна проектирования «Строитель». Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Он делает возможным использование одного и того же кода строительства для получения разных представлений объектов [7].

С другой стороны, секции представляют из себя набор разных сущностей, обладающих различными атрибутами, но имеющих между собой некоторые общие черты: имеют метод разбора (отличающегося лишь алгоритмом реализации), метод для предоставления информации о себе и ключ, предназначенный для идентификации соответствия конкретных данных конкретной секции.

Таким образом, можно выделить основу: абстрактный базовый класс для всех сущностей.

```
public abstract class SectionBase
{
    public abstract string Key { get; }
    public abstract SectionBase Parse(string context);
    public virtual void ShowDetails() { this.Dump(); }
}
```

Рисунок 2 – Фрагмент кода абстрактного класса

На этом этапе мы могли бы начать создавать набор классов-наследников, но то условие, что мы не можем знать заранее какие именно секции будут создаваться во время выполнения программы затруднит процесс построения результирующего объекта, что потребует введения какого-то механизма распознавания секций по типу оператора выбора “switch”. Однако лучшим способом создания таких однотипных объектов-секций будет создание фабрики. Паттерн «Фабричный метод» — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в базовом классе, позволяя подклассам изменять тип создаваемых объектов.

Для реализации паттерна фабричный метод, сначала создадим абстрактный создатель объектов.

```
public abstract class CreatorBase
{
    public abstract SectionBase Create();
    public SectionBase Create(string ctx)
    {
        var product = Create();
        return product.Parse(ctx);
    }
}
```

Рисунок 3 – Абстрактный создатель объектов

Далее создадим обобщенную фабрику создания секций. На рисунке 3 изображен фрагмент кода обобщенной фабрики создания коллекции

```
public sealed class SectionFactory<T> : CreatorBase
where T : SectionBase, new()
{
    public override SectionBase Create() ⇒ new T();
}
```

Рисунок 4 – фрагмент кода обобщенной фабрики создания коллекции

Таким образом, становится возможным порождение объектов (секций) полиморфным способом, прямо во время выполнения программы.

Теперь можно создать конкретные секции, например, «Авиакомпания» и «Багаж». На рисунках 4-5 изображен фрагмент кода секции «Авиакомпания».

```
class AviaCompany : SectionBase
{
    public override string Key ⇒ @"^A-";
    public string Name { get; set; }
    public string Prefix { get; set; }

    public override SectionBase Parse(string context)
    {
        var pattern = Key;
        Name = Regex.Replace(context.Split(';')[0], Regex.Match(context, pattern).Value, "");
        Prefix = Regex.Match(context.Split(';')[1], @"\w+").Value;
        return this;
    }
}
```

Рисунок 5 – фрагмент кода AviaCompany

```
class Baggage : SectionBase
{
    public override string Key ⇒ @"(^SSR ABAG\s+)|(^SSR ABGS\s+)";
    public double Weight { get; set; }

    public override SectionBase Parse(string context)
    {
        var text = Regex.Match(context, @"(TO\s\d+\sKG)|(TO\s\d+.?\d+\sKG)").Value;
        text = Regex.Match(text, @"(\d+.\d+)|(\d+)").Value;
        if (double.TryParse(text, NumberStyles.Float,
        CultureInfo.InvariantCulture, out double val))
            Weight = val;
        return this;
    }
}
```

Рисунок 6 – фрагмент кода Baggage

Далее определим результирующий класс, который будет содержать в себе все секции из разобранного файла импорта. На рисунке 6 изображен фрагмент кода TiketData – билет.

```
public class TicketData
{
    public List<SectionBase> Sections { get; } = new List<SectionBase>();
    public TicketData(List<SectionBase> sections)
    {
        Sections = sections;
    }
}
```

Рисунок 7 – Фрагмент кода TicketData

Чтобы построить экземпляр данного класса реализуем паттерн строитель. На рисунке 8 изображен фрагмент кода патерна «Строителя».

```
public class TicketDataBuilder
{
    private List<SectionBase> sections = new List<SectionBase>();

    public Dictionary<string, Type> Handlers { get; private set; }
                        = new Dictionary<string, System.Type>();

    private Dictionary<Type, Func<CreatorBase>> BuildMap
                    = new Dictionary<Type, Func<CreatorBase>>();

    public void AddHandler<T>() where T : SectionBase, new()
    {
        if (!BuildMap.ContainsKey(typeof(T)))
        {
            BuildMap.Add(typeof(T), () => new SectionFactory<T>());
            Handlers.Add(new T().Key, typeof(T));
        }
    }
    private void BuildPart(Type handler, string context) =>
    sections.Add(BuildMap[handler].Invoke().Create(context));

    public void Build(Type handler, string context)
    {
        if (BuildMap.ContainsKey(handler))
            BuildPart(handler, context);
        else
            throw new ArgumentException("Отсутствует конфигурирация " + handler);
    }
    public void Build<T>(string context) => Build(typeof(T), context);
    public TicketData GetResult() => new TicketData(sections);
}
```

Рисунок 8 – Фрагмент кода TicketDataBuilder

Рассмотрим данный класс более подробно.

- Закрытое поле sections предназначено для хранения создаваемых секций, из которых будет составлен результирующий объект.

- Открытое свойство Handlers, основанное на словаре, будет хранить пары ключ-значение для ключей секций и соответствующих типов секций, для этих значений. Иными словами, имея ключ секции авиакомпании (регулярное выражение), мы можем получить объект AviaCompany, который сможет разобрать строку файла, представляющую эту секцию.
- Закрытое свойство BuildMap также является словарем, но в качестве ключа используется тип порождаемого объекта, а в качестве значения обобщенный делегат, который строит объект заданного типа.
- Открытый Метод AddHandler<T> - конфигурирует класс строитель новым типом секции. Тем самым позволяя строить такие секции при использовании.
- Закрытый метод BuildPart создает необходимую секцию путем вызова делегата создания этой секции из словаря BuidMap и передает построенному объекту (секции) в метод Parse контекст, который необходимо разобрать и заполнить соответствующие свойства этой секции.
- Открытый метод Build вызывает предыдущий метод и добавляет созданный объект в результирующий список sections.
- Открытый, обобщенный метод Build – является перегрузкой вышеуказанного метода, перегруженной по выводимому типу.
- Открытый метод GetResult() – позволяет получить построенный объект.

Добавим новый статический класс с методом расширения Use, что позволит в более удобной форме (Fluent design) конфигурировать класс строитель. На рисунке 9 изображен фрагмент кода, реализующего хэлпер-класс с методом расширения.

```
public static class BuilderHelper
{
    public static TicketDataBuilder Use<T>(this TicketDataBuilder tb)
    where T : SectionBase, new()
    {
        tb.AddHandler<T>();
        return tb;
    }
}
```

Рисунок 9 – Фрагмент кода BuilderHelper

Теперь, когда классовая структура готова, остается написать класс Parser, который будет производить разбор всего файла. На рисунке 10 представлен фрагмент кода этого класса.

```
class Parser
{
    public static void Parse(string airFile)
    {
        var content = File.ReadAllText(airFile);
        var builder = new TicketDataBuilder();
        builder
        .Use<Baggage>()
        .Use<AviaCompany>();

        var lines = content.Split(new[] { '\n' }, StringSplitOptions.RemoveEmptyEntries);

        foreach (var line in lines)
            foreach (var handler in builder.Handlers)
                if (Regex.IsMatch(line, handler.Key))
                    builder.Build(handler.Value, line);

        foreach (var section in builder.GetResult().Sections)
            section.ShowDetails();
    }
    public static void Main(string[] args)
    {
        var fileName = Path.Combine(
        Environment.GetFolderPath(Environment.SpecialFolder.Desktop),
        @"4212429730775.ttm");
        Parse(fileName);
    }
}
```

Рисунок 10 – Фрагмент кода Parser

Его статический метод Parse принимает имя входного файла, затем производит чтение этого файла. После чего создается объект – строитель, который конфигурируется классами Baggage и AviaCompany. Далее создается набор массив строк, загруженных из файла и в нижеследующем цикле производится их поочередный перебор. Внутренний цикл, в свою очередь, производит перебор всех ключей, которые были добавлены при конфигурировании строителя, и в случае если найдено соответствие регулярного выражения (ключа) в текущей строке будет произведено построение соответствующей секции, которая примет эту строку и произведет разбор данных. В результате чего объект builder будет заполнен распознанными секциями. Последний цикл предназначен для наглядного отображения построенных секций.

В нашем случае вывод будет выглядеть, как показано на рисунке 11.

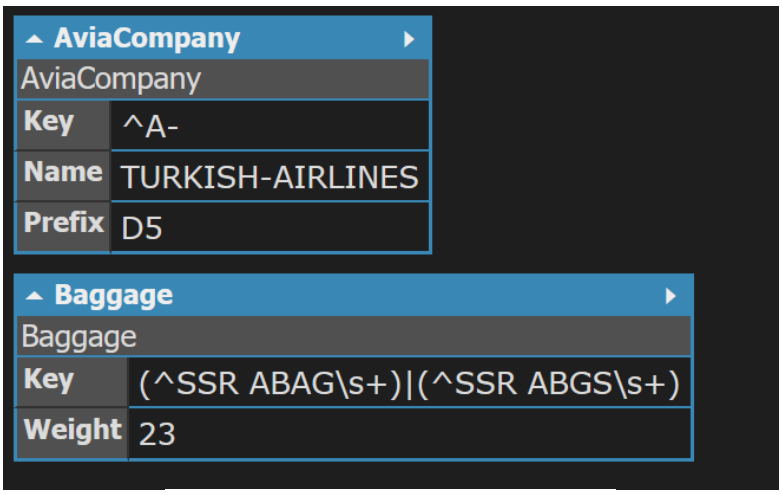

Рисунок 11 – вывод данных

Но вся польза данного подхода заключается в гибкости, которую предоставляет такая архитектура. Если по определенным обстоятельствам понадобится получать из исходных файлов дополнительную информацию, например сведения о налоге на добавленную стоимость, необходимо сделать следующее:

1. Добавить класс, представляющий новую сущность (VatInfo), который будет унаследован от SectionBase.
2. Определить регулярное выражение для этой секции ((^FEINCL\s+)|(^FE\w+/INCL\s+)). Оно будет являться переопределенным полем базового класса Key.
3. Реализовать метод разбора.

Таким образом, получим класс, как представлен на рисунке 12.

```
class VatInfo : SectionBase
{
    public override string Key => @"(^FEINCL\s+)|(^FE\w+/INCL\s+)";
    public double Value { get; set; }
    public string Currency { get; set; }
    public override SectionBase Parse(string context)
    {
        var vatPtrn = @"\d+.?\d+";
        var patt = @"VAT" + vatPtrn;
        var vatStr = Regex.Replace(context, Regex.Match(context, Key).Value, "");
        var value = Regex.Match(Regex.Match(vatStr, patt).Value, vatPtrn).Value;
        Currency = Regex.Replace(vatStr, patt, "");
        if (double.TryParse(value, NumberStyles.Currency, CultureInfo.InvariantCulture, out double res))
            Value = res;
        return this;
    }
}
```

Рисунок 12 – фрагмент кода VatInfo

4. Добавить полученный класс при конфигурировании строителя.

На рисунке 13 представлен фрагмент кода конфигурирования строителя.

```
var builder = new TicketDataBuilder();
builder.Use<Baggage>()
       .Use<AviaCompany>()
       .Use<VatInfo>();
```

Рисунок 13 – Блок кода конфигурирования строителя

Таким образом, мы получаем новый результат, как представлено на рисунке 14.

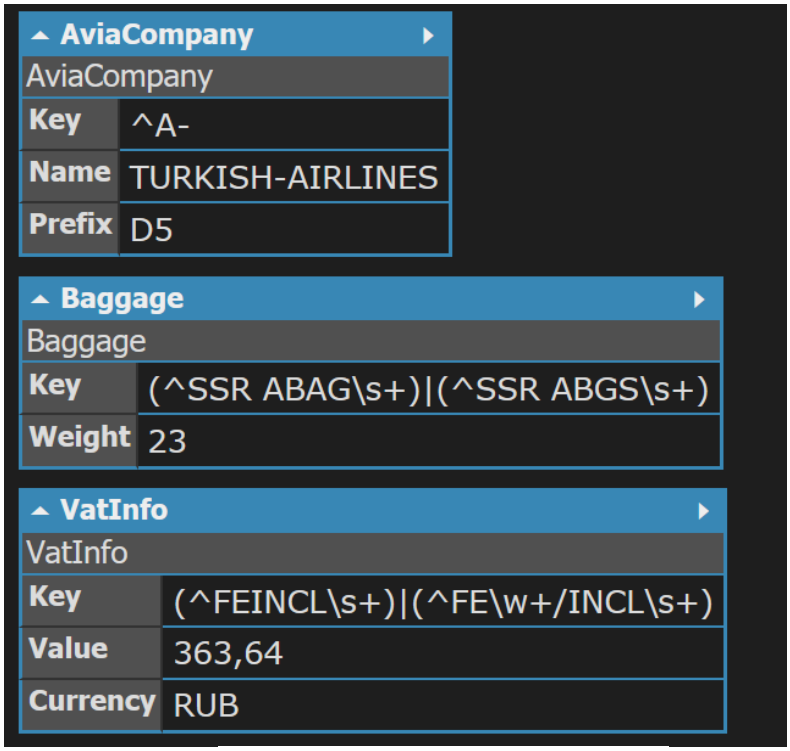

Рисунок 14 – Вывод данных

3. Выводы

В данной статье рассмотрен способ построения гибкой архитектуры приложения, что позволяет в перспективе развития проекта, или изменения входных данных, без чрезмерной сложности вносить изменения в проект, гарантированно не нарушая работоспособность уже существующего кода.

Библиографический список

1. Мартин Р. С., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. СПб: Символ-Плюс, 2013. 768 с.
2. Шевчук А., Охрименко Д., Касьянов А. Design Patterns via C# Приемы объектно-ориентированного проектирования. Киев: ITVDN, 2015. 288 с.
3. Nesteruk D. Design Patterns in Modern C++. NY: APRESS, 2018. 314 с. (49-60) URL: https://www.elibrary.ru/item.asp?id=35750959&selid=35750960
4. Design Patterns: Elements of Reusable Object-Oriented Software / Gamma E., Helm R., Johnson R., Vissides J., Vissides J. и др.; под ред. Gamma E. 1 изд.

NY: Addison-Wesley Professional, 1994. 416 с.
5. SOLID Design Principles Explained: The Single Responsibility Principle // https://stackify.com/ URL: https://stackify.com/solid-design-principles/ (дата обращения: 20.12.2020).
6. Tony Stubblebine, T., 2007. Regular Expression Pocket Reference, 2nd Edition. O'Reilly Media, Inc., pp: 38-49.
7. Refactoring.Guru URL: https://refactoring.guru/ru (дата обращения: 1.09.2021).