Математическое, алгоритмическое и программное обеспечение

DOI 10.66032/2221-2574-2025-1-2-59-68 УДК 004.421

СИСТЕМНАЯ СТРУКТУРИЗАЦИЯ ШАБЛОНА MODEL-VIEW-PRESENTER

Дубов Илья Ройдович
доктор технических наук, профессор, профессор кафедры вычислительной техники и систем управления ФГБОУ ВО «Владимирский государственный университет имени А.Г. и Н.Г. Столетовых».
E-mail: dubov@vlsu.ru

Адрес: 600000, Российская Федерация, г. Владимир, ул. Горького, д. 87.

Аннотация: В работе показано, что элементы архитектурного шаблона Model-View-Presenter могут рассматриваться как части программной системы, состоящей из подсистемы пользовательского интерфейса и подсистемы бизнес-логики. В первую подсистему входит представление (View), во вторую — модель (Model). Менеджер (Presenter) входит в обе подсистемы одновременно, являясь их границей. Он в процедурном виде реализует взаимодействие представления и модели в соответствии со сценарием, заданным функциональными требованиями к системе. Гранулярность структуры представления и структуры модели определяется структурой блоков кода менеджера, так как каждый блок его кода служит основой для формирования триады MVP. Отношение «включение» варианта использования программно реализуется как процедурный вызов соответствующего менеджера. Показана возможность применения различных представлений, например, на основе консольного ввода-вывода или GUI, без изменения менеджера.

Ключевые слова: система, декомпозиция, вариант использования, сценарий, представление, бизнес-логика.

Введение

Шаблон Model-View-Presenter (MVP) широко используется в разработке программного обеспечения для выделения основных компонентов архитектуры приложения. Однако в известных рекомендациях элементы шаблона и принципы их взаимодействия рассматриваются на основе интуитивных, неформальных понятий. Поэтому применение шаблона вызывает трудности в условиях отсутствия теоретически обоснованной методики. Цель данной работы — выявить фундаментальные свойства элементов MVP и, используя их, получить теоретически обоснованную методику реализации шаблона в разработке приложений.

Архитектурные шаблоны, в состав которых входят представление (View) и модель (Model), обладают общими чертами и могут рассматриваться как единое семейство Model-View-* (MV*) [1]. Модель реализует бизнес-логику приложения, выполняя обработку и хранение данных, а представление обеспечивает пользователя данными в понятной ему форме. Исторически первыми были варианты шаблона MVC с пассивным [2] и активным [3] представлением. В них элемент контроллер (Controller) получает управление от пользователя или активного представления и управляет моделью, изменяя её состояние. В шаблоне MVP [4], в отличие от MVC, представление и модель не взаимодействуют напрямую, между ними располагается менеджер (Presenter). В этой работе будем использовать термин «менеджер» вместо «презентер», чтобы подчеркнуть равную значимость его действий в управлении и представлением, и моделью. Он инициирует изменение состояния модели в соответствии с данными и командами, поступающими от представления, извлекает данные из модели для передачи их представлению и последующей демонстрации пользователю (рис. 1).

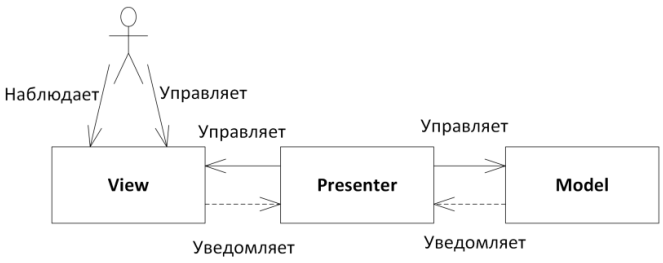

Рис. 1. Шаблон Model-View-Presenter

К семейству MV* следует отнести также шаблон HMVC (Hierarchical Model-View-Controller), основанный на иерархии триад MVC, и шаблон PAC (Presentation-Abstraction-Control), основанный на иерархии триад MVP (см. [3, 5]). Для формирования иерархии триад сформулированы некоторые эмпирические правила, основанные на общем понимании функциональных особенностей и сложности контейнера визуальных элементов [3].

Имеется ряд работ, в которых предлагается разрабатывать менеджер в MVP так, чтобы в нём действия соответствовали непосредственно функциональным требованиям, задаваемым в форме сценариев. В Rational Unified Process такие требования называют вариантами использования, в Microsoft Solution Framework — сценариями, а в Agile-подходе — пользовательскими историями [6, 7]. Это делает процесс проектирования программной системы более определённым и позволяет внедрить в разработку подходы «Presenter First» и «Test-Driven Design» [1, 8].

В рамках концепции «Model-Driven Architecture» имеются работы, в которых предлагаются подходы к автоматическому преобразованию вариантов использования в код на языке программирования [9, 10, 11] в соответствии с шаблоном MVP. Идея заключается в использовании некоторого предметно-ориентированного языка для написания требований и правил преобразования конструкций этого языка в код на языке программирования. Следует иметь в виду, что в этом подходе субъективная процедура выявления элементов шаблона MVP фактически совмещается с процессом составления требований.

Системная интерпретация MVP

Процесс проектирования программной системы представляет собой синтез кода программы с использованием архитектурных шаблонов, в том числе MVP. Это прямая задача проектирования. Однако здесь рассмотрим обратную ей задачу — из готовой программной системы путём декомпозиции выделим шаблон MVP. Для этого используем простейшее экспериментальное приложение, написанное в процедурном стиле на языке C#. Оно изначально не содержит каких-либо объектных архитектурных решений. Выполняя трансформацию кода приложения, добьёмся того, чтобы в нём появились черты шаблона MVP. Анализ процесса трансформации позволит наглядно показать природу элементов MVP и логику их структурирования.

Предлагаемая экспериментальная программа напоминает психологический тест для исследования внимания человека: пациенту демонстрируются случайные числа, и он должен повторять их. Назначение программы — подсчитать количество попыток, выполненных без ошибок (функция CheckAttentionFlow на рис. 2). До начала тестирования следует определить по состоянию здоровья, готов ли пациент к участию в исследовании. Тест проводится, если пациент оценивает своё самочувствие достаточно высоко (функция InspectFlow на рис. 3). В точке входа в программу Main (рис. 4) выполняется CheckAttenstionFlow. В программе используется заданный набор операций консольного ввода-вывода, составляющий основу пользовательского интерфейса (класс UiCore на рис. 4).

Термин «система» в одном из своих значений определяется как совокупность элементов и набор правил их взаимодействия, образующих единое целое. Такому пониманию системы в программировании соответствует конструкция «блок». Локальные переменные блока — это элементы, значения которых в совокупности представляют состояние системы. Операторы блока являются правилами функ-

ционирования и изменяют состояние системы. Экземпляр системы, задаваемый блоком, создаётся в момент передачи управления в блок и перестаёт существовать при передаче управления за пределы блока. Блок, как система, может иерархически содержать вложенные блоки, представляющие подсистемы. Блок тела функции также является системой, в которой формальные параметры дают доступ к элементам надсистемы.

```
public static void CheckAttentionFlow() // class CheckAttention
{
    var title = "Check Attention";
    var counter = 0;
    if (Inspect.InspectFlow())
    {
        var testTitle = "Test";
        for (int i = 0; i < 3; i++)
        {
            var sample = DateTime.Now.Microsecond.ToString();
            UiCore.WriteLine(testTitle, sample);
            var answer = UiCore.ReadLine(testTitle);
            if (sample.Equals(answer)) ++counter;
        }
    }
    UiCore.WriteLine(title, $"Counter: {counter}");
    UiCore.ReadLine(title);
}
```

Рис. 2. Основная функция CheckAttentionFlow экспериментального приложения

```
public static bool InspectFlow() // class Inspect
{
    var title = "Inspect";
    var maxRate = 10;
    int rate;
    bool result;
    string message;
    message = $"Rate up to {maxRate}:";
    UiCore.WriteLine(title, message);
    rate = int.Parse(UiCore.ReadLine(title));
    result = rate > maxRate / 2;
    return result;
}
```

Рис. 3. Вызываемая функция InspectFlow экспериментального приложения

Декомпозиция программной системы на подсистему пользовательского интерфейса и подсистему бизнес-логики подразумевает соответствующую декомпозицию всех её подсистем. В экспериментальной программе системе верхнего уровня соответствует блок тела функции Main. В ней выполняется подсистема (функция) CheckAttentionFlow, которая, в свою очередь, инициирует подсистему (функцию) InspectFlow. В составе CheckAttentionFlow имеются вложенные блоки, которые тоже являются подсистемами, но в данном случае они анонимные.

Основную идею декомпозиции блока поясним на примере с телом функции InspectFlow. В блок тела этой функции (рис. 3) входит несколько локальных переменных и операторов. Этот блок, как систему, необходимо разделить на две подсистемы: пользовательский интерфейс и бизнес-логику. При разнесении элементов функции по этим двум подсистемам получим три группы (рис. 5). В группу view (представление) отнесём те элементы, которые однозначно относятся к пользовательскому интерфейсу и не используются в бизнес-логике. В группу model (модель) отнесём элементы, необходимые для вычисления степени готовности пациента, и не используемые в пользовательском интерфейсе. Остаются переменные и операторы, которые не могут быть однозначно отнесены только к одной из двух подсистем, так как они используются как в подсистеме пользовательского интерфейса, так и в подсистеме бизнес-логики, т.е. являются общими. Группа общих элементов presenter (менеджер) образует границу подсистем.

В реальных языках программирования отсутствуют механизмы для поддержки рассматриваемой теоретической декомпозиции блока

```
static class Program { static void Main() => CheckAttention.CheckAttentionFlow(); }
static class UiCore
{
  public static void WriteLine(string title, string? value) =>
Console.WriteLine($"{title}: {value}");
  public static string? ReadLine(string title) { Console.Write($"{title}: ");
return Console.ReadLine(); }
}
```

Рис. 4. Точка входа в приложение Main и операции ввода-вывода в статическом классе UiCore

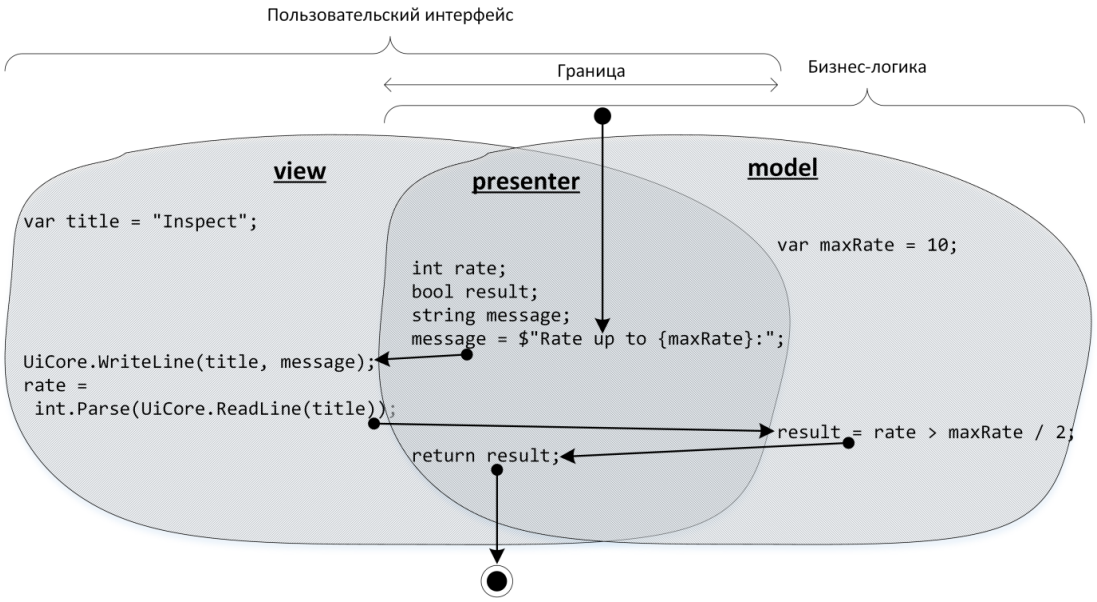

Рис. 5. Декомпозиция блока тела функции InspectFlow на смежные блоки (подсистемы)

на смежные блоки с общей границей. Поэтому декомпозицию моделируем средствами объектно-ориентированного программирования. Из элементов подсистем, за исключением границы, формируются объекты представления и модели. Локальные переменные блока становятся полями объекта, исполняемые операторы включаются в методы объекта. При этом операторы группируются по методам таким образом, чтобы при вызове методов менеджером обеспечивалась та же последовательность действий, которая имела место в исходном блоке. Объекты представления и модели должны создаваться и уничтожаться одновременно с началом и завершением работы менеджера, поскольку все три группы элементов выделены логически, но в совокупности образуют одну систему с определённым началом и концом существования. Для этого воспользуемся оператором using, реализуя интерфейс IDisposable в представлении и модели.

Для объекта представления получаем класс на рис. 6. В нём имеется одно поле, которое в исходном блоке было локальной переменной, и один метод, состоящий из двух последовательных операторов, полученных из исходного блока. Аналогично, класс модели (рис. 7) содержит одно поле и метод, выполняющий оператор из исходного блока. Функция на рис. 8 начинает исполнять блок менеджера (тело оператора using) с одновременным созданием объектов представления и модели. Менеджер исполняет методы представления и модели, а завершается блок менеджера одновременным уничтожением объектов представления и модели.

```
class Inspect2View : IDisposable
{
    string title = "Inspect";
    public int GetRate(string message)
    {
        UiCore.WriteLine(title, message);
        return int.Parse(UiCore.ReadLine(title));
    }
    void IDisposable.Dispose() { }
}
```

Рис. 6. Класс представления, полученный в результате декомпозиции InspectFlow

```
class Inspect2Model : IDisposable
{
    int maxRate = 10;
    public int MaxRate => maxRate;
    public bool GoodRate(int rate) => rate > maxRate / 2;
    void IDisposable. Dispose() { }
}
```

Рис. 7. Класс модели, полученный в результате декомпозиции InspectFlow

Выполненное преобразование позволяет уточнить природу элементов шаблона MVP. Представление и модель — это искусственно выделенные объекты, необходимые для программной реализации декомпозиции системы на подсистему пользовательского интерфейса и подсистему бизнес-логики. Подсистема пользовательского интерфейса состоит из представления и менеджера, а подсистема бизнес-логики состоит из модели и менеджера. Таким образом, менеджер входит одновременно в обе подсистемы, являясь их границей.

```
public static bool Inspect2Flow() // class Inspect2
{
    using (var view = new Inspect2View())
    using (var model = new Inspect2Model())
    {
        int rate;
        bool result;
        string message;
        message = $"Rate up to {model.MaxRate}:";
        rate = view.GetRate(message);
        result = model.GoodRate(rate);
        return result;
    }
}
```

преставление

менеджер

модель

Рис. 8. Функция Inspect2Flow с MVP, соответствующая исходной InspectFlow

Выполним декомпозицию функции CheckAttentionFlow на подсистемы пользовательского интерфейса и бизнес-модели аналогичным образом. В блоке тела функции выделяются элементы для представления view и для модели model (рис. 9). В функции имеется вложенный блок, для которого тоже нужно выполнить декомпозицию. В нём операторы ввода-вывода зависят от локальной переменной testTitle, следовательно, взаимодействие пользователя с этим блоком отличается от взаимодействия с внешним блоком. Поэтому соответствующие элементы образуют дополнительное представление testView. Во вложенном блоке нет переменных, характеризующих состояние бизнес-логики в этом блоке, поэтому его операторы бизнес-логики входят в объект model внешнего блока. Имеется ещё один вложенный блок — тело цикла, но его декомпозиция ничего не даёт, так как в нем нет переменных, которые делали бы необходимым создание дополнительного объекта представления или модели. Используя объектный подход, из локальных переменных и операторов исходной функции создаём классы для объектов представления view и testView (рис. 10), а также класс для объекта модели model (рис. 11). На рис. 12 представлена функция

```
public static void CheckAttentionFlow() // class CheckAttention
{
    var title = "Check Attention";
    var counter = 0;
    if (Inspect.InspectFlow())
    {
        var testTitle = "Test";
        for (int i = 0; i < 3; i++)
        {
            var sample = DateTime.Now.Microsecond.ToString();
            UiCore.WriteLine(testTitle, sample);
            var answer = UiCore.ReadLine(testTitle);
            if (sample.Equals(answer)) ++counter;
        }
    }
    UiCore.WriteLine(title, $"Counter: {counter}");
    UiCore.ReadLine(title);
}
```

вложенный блок

view

testView

model

Рис. 9. Декомпозиция блока тела функции CheckAttentionFlow на объекты представлений view, testView и объект модели model

CheckAttention2Flow с двумя триадами MVP, соответствующая исходной CheckAttention Flow.

Код каждого менеджера, входящего в функции CheckAttention2Flow и Inspect2Flow,

```
class CheckAttention2View : IDisposable
{
    string title = "Check Attention";
    public void Inform(string message)
    {
        UiCore.WriteLine(title, message);
        UiCore.ReadLine(title);
    }
    void IDisposable.Dispose() { }
}

class Test2View : IDisposable
{
    string testTitle = "Test";
    public string GetAnswer(string sample)
    {
        UiCore.WriteLine(testTitle, sample);
        return UiCore.ReadLine(testTitle);
    }
    void IDisposable.Dispose() { }
}
```

Рис. 10. Классы представлений: CheckAttention2View — для блока тела функции, Test2View — для вложенного блока

```
class CheckAttention2Model : IDisposable
{
    int counter = 0;
    public int Counter => counter;
    public string GetSample() => DateTime.Now.Microsecond.ToString();
    public void Count(string text, string answer)
    {
        if (text.Equals(answer)) ++counter;
    }
    void IDisposable.Dispose() { }
}
```

Рис. 11. Класс объекта модели, полученного при декомпозиции CheckAttentionFlow

```
public static void CheckAttention2Flow()
{
    using (var view = new CheckAttention2View())
    using (var model = new CheckAttention2Model())
    {
        if (Inspect2.Inspect2Flow())
        {
            using (var testView = new Test2View())
            {
                for (int i = 0; i < 3; i++)
                {
                    var sample = model.GetSample();
                    var answer = testView.GetAnswer(sample);
                    model.Count(sample, answer);
                }
            }
        }
        view.Inform($"Counter: {model.Counter}");
    }
}
```

триада MVP

вложенная триада MVP

Рис. 12. Функция CheckAttention2Flow с триадами MVP, соответствующая исходной CheckAttentionFlow

содержит последовательность вызова методов представления и модели, т.е. осуществляет взаимодействие между этими двумя объектами. Поэтому менеджер может рассматриваться как программное воплощение сценария взаимодействия пользователя и подсистемы бизнес-логики, где пользователь — это система, состоящая из реального пользователя и подсистемы пользовательского интерфейса, которая обеспечивает интеграцию реального пользователя в программную среду. Руководствуясь кодом менеджеров (рис. 8, 12), запишем функциональные требования к программной системе в виде вариантов использования (рис. 13 и 14). Эту процедуру можно считать реверс-инжинирингом экспериментальной программы.

Как видно из восстановленных вариантов использования, отношения между ними отображаются в отношения между менеджерами в программной реализации. Отношению включения (include) варианта использования соответствует вызов функции менеджера. Вложенному блоку сценария соответствует вложенный менеджер (вложенный блок кода) в программной реализации. В данной работе не рассматривается реализация других отношений между вариантами использования, так как это требует отдельного обсуждения.

Вариант использования «Исследовать внимание». Актант: пациент.
1. Система определяет готовность пациента к участию в исследовании (вариант использования «Определить готовность пациента»).
2. Если пациент готов участвовать, система три раза опрашивает пациента:
2.1. Пациенту предъявляется случайное число.
2.2. Пациент вводит запомненное число.
2.3. Система учитывает правильно введенное число.
3. Пациенту демонстрируется количество правильно воспроизведенных чисел.

Рис. 13. Вариант использования, восстановленный из менеджера в CheckAttention2Flow

Вариант использования «Определить готовность». Актант: пациент.
1. Пациенту предлагается ввести оценку своего самочувствия, не превышающую заданную величину.
2. Система считает, что пациент готов к участию, если оценка самочувствия достаточно высокая.

Рис. 14. Вариант использования, восстановленный из менеджера в Inspect2Flow

Подход «Presenter First»

Авторами работы [8] предлагается подход «Presenter First» для организации разработки программных систем. Он заключается в том, что создание приложения в первую очередь начинается с работы над менеджерами, в которых реализуется логика сценариев из функциональных требований. В процессе разработки менеджера определяются сигнатуры вызываемых функций и необходимых событий, которые составляют интерфейс обратного вызова для представления и интерфейс обратного вызова для модели (рис. 15). Во вторую очередь разрабатываются конкретные классы, реализующие соответствующие интерфейсы. Изоляция менеджера от конкретных классов представления и модели позволяет создавать тестирующие классы с указанными интерфейсами и, подставляя их вместо представлений и моделей, организовать тестирование приложения в виде модульных тестов в соответствии с принципом TDD [12] до разработки целевых классов.

Рис. 15. Шаблон MVP со связями через интерфейсы обратного вызова IView и IModel

После тестирования для полученных на первом этапе интерфейсов разрабатываются классы представлений на базе определённой библиотеки пользовательского интерфейса (например, Windows Forms для C#) и классы моделей, выполняющие бизнес-логику приложения.

В менеджерах CheckAttention2Flow (рис. 12) и Inspect2Flow (рис. 8) эксперимен-

```
public static bool Inspect3Flow(
    IInspect3ViewFactory viewFactory, IInspect3ModelFactory modelFactory) // class Inspect3
{
    using (IInspect3View view = viewFactory.CreateView())
    using (IInspect3Model model = modelFactory.CreateModel())
    {
        int rate = 0;
        bool result;
        string message;
        message = $"Rate up to {model.MaxRate}:";
        rate = view.GetRate(message);
        result = model.GoodRate(rate);
        return result;
    }
}
```

Рис. 16. Функция Inspect3Flow, полученная из Inspect2Flow введением интерфейсов представления (IInspect3View), модели (IInspect3Model) и фабрик (IInspect3ViewFactory, IInspect3ModelFactory)

```
public static void CheckAttention3Flow(ICheckAttention3ViewFactory
    viewFactory, ICheckAttention3ModelFactory modelFactory) // class CheckAttention3
{
    using (ICheckAttention3View view = viewFactory.CreateView())
    using (ICheckAttention3Model model = modelFactory.CreateModel())
    {
        if (Inspect3.Inspect3Flow(view.CreateInspect3ViewFactory(),
            model.CreateInspect3ModelFactory()))
            using (ITest3View testView = view.CreateTest3View())
            {
                for (int i = 0; i < 3; i++)
                {
                    var sample = model.GetSample();
                    var answer = testView.GetAnswer(sample);
                    model.Count(sample, answer);
                }
            }
        view.Inform($"Counter: {model.Counter}");
    }
}
```

Рис. 17. Функция CheckAttention3Flow, полученная из CheckAttention2Flow введением интерфейсов представлений (ICheckAttention3View, ITest3View), модели (ICheckAttention3Model) и фабрик (ICheckAttention3ViewFactory, ICheckAttention3ModelFactory)

```
public class InspectWinViewFactory(Control control) : IInspect3ViewFactory
{
    public IInspect3View CreateView() => new View(control);
    class View : IInspect3View
    {
        readonly Control _control;
        readonly TextBox rateTextBox = new ();
        readonly Button okButton = new ();
// ...
        public int GetRate(string message)
        {
            var waitUi = new AutoResetEvent(false);
            _control.Invoke(() => messageLabel.Text = message);
            void okButtonClick(object? sender, EventArgs e) => waitUi.Set();
            okButton.Click += okButtonClick;
            waitUi.WaitOne();
            okButton.Click -= okButtonClick;
            return int.Parse(rateTextBox.Text);
        }
    }
}
```

Рис. 18. Реализация метода GetRate для преставления, использующего библиотеку Windows Forms

тального приложения экземпляры объектов представления и модели создаются прямо в теле функций. Чтобы исключить зависимость этих менеджеров от конкретных классов, выделим из классов интерфейсы и заменим действия с объектами классов на работу с интерфейсами. Исключить зависимость менеджеров от механизма конструирования объектов можно различными способами: глобальная фабрика, инверсия зависимостей и т.п. В данном примере используем шаблон «Abstract Factory» [13]. На рис. 16 и 17 показаны менеджеры после выделения интерфейсов для представлений и моделей. Код для интерфейсов приводить не будем, так как набор методов для каждого интерфейса без труда определяется по вызовам в коде менеджера.

Через входные параметры в обе функции передаются фабрики для создания представления и модели. Представление с интерфейсом ICheckAttention3View, в свою очередь, выступает в роли фабрики для представления с интерфейсом ITest3View во вложенном блоке. При вызове Inspect3Flow представление с интерфейсом ICheckAttention3View выступает в роли фабрики для создания фабрики представления, передаваемой в вызываемую функцию Inspect3Flow. Там же модель с интерфейсом ICheckAttention3ModelFactory выступает в роли фабрики для создания фабрики модели внутри функции Inspect3Flow.

Следует обратить внимание на то, что все методы представлений вызываются в менеджерах на рис. 16 и 17 обычным образом, без механизма обработки событий. Это существенное отличие от конструирования менеджера с использованием подписки на события представления, описанного в [8]. Представление, построенное на консольном вводе-выводе, показано на рис. 6 и 10. Однако реализация тех же методов представления с использованием графического пользовательского интерфейса (GUI) требует дополнительных разъяснений.

При использовании GUI визуальная часть представления выполняется в специальном потоке для GUI, а менеджер — в другом потоке. Методы модели могут выполняться в потоке менеджера, хотя при необходимости они могут выполняться и в отдельных потоках. На рис. 18 показаны фрагменты кода для класса фабрики представления InspectWinViewFactory и класса самого представления View, которые реализуют соответствующие интерфейсы, используемые в менеджере на рис. 16.

Метод представления GetRate после нажатия кнопки okButton должен вернуть число, введённое пользователем в визуальный элемент rateTextBox. Эти элементы управления агрегируются элементом _control на форме пользовательского интерфейса. Метод GetRate выполняется в потоке менеджера, так как он вызывается непосредственно из него. В GetRate имеется элемент синхронизации потоков — событие класса AutoResetEvent. Выполнение GetRate блокируется на вызове метода WaitOne события в ожидании сигнала. В это время пользователь записывает число в элемент текстового ввода, после чего нажимает на кнопку для продолжения работы. Обработчик

нажатия кнопки подаёт сигнал элементу синхронизации, и поток менеджера продолжает выполнять метод GetRate, возвращая введённое пользователем значение. Способ согласования потока менеджера с потоком GUI при помощи элемента синхронизации AutoResetEvent известен, но найти его автора по литературным источникам не удалось.

Заключение

В работе показано, что шаблон MVP может рассматриваться как инструмент декомпозиции программной системы на две подсистемы: подсистему пользовательского интерфейса и подсистему бизнес-логики. В этой декомпозиции менеджер является границей подсистем, так как он входит в обе из них. Представление совместно с менеджером образует подсистему пользовательского интерфейса, а модель совместно с менеджером образует подсистему бизнес-логики. Такая интерпретация представления, модели и менеджера позволяет сформулировать следующие результаты:

1) Менеджер реализует взаимодействие представления и модели в соответствии со сценарием из функциональных требований к системе.

2) Гранулярность структуры представления и модели определяется структурой блоков кода менеджера. Для вложенного блока должен создаваться дополнительный экземпляр представления и/или модели, если у представления и/или модели должны появиться дополнительные поля, характеризующие его состояние более детально, чем во внешнем блоке. Экземпляру представления и/или модели приписываются методы, содержащие операторы работы с дополнительными полями и вызываемые во вложенном блоке.

3) Отношение «включение» варианта использования реализуется как вызов соответствующей функции менеджера.

4) Различные представления, например, на основе консольного ввода-вывода или GUI, могут быть использованы без изменения менеджера.

Таким образом, благодаря системной интерпретации элементов MVP, получена обоснованная методика реализации данного шаблона в разработке приложений. В соответствии с ней функции менеджеров исполняют сценарии функциональных требований, а структура представления и модели определяется блочной структурой сценариев.

Литература

1. *Syromiatnikov A., Weyns D.* A Journey through the Land of Model-View-Design Patterns // IEEE/IFIP Conference on Software Architecture, Sydney, NSW, Australia, 2014. pp. 21-30.

2. *Фаулер М., Райс Д., Фоммел М.* Шаблоны корпоративных приложений. М: Диалектика, 2020. 544 с.

3. *Karagkasidis A.* Developing GUI applications: architectural patterns revisited // The Thirteenth Annual European Conference on Pattern Languages of Programming (EuroPLoP 2008), Irsee, Germany, 2008.

4. *Fowler M.* GUI Architectures. [Электронный ресурс]: URL: https://martinfowler.com/eaaDev/uiArchs.html (дата обращения: 16.11.2024).

5. *Wendler S., Streitferdt D.* A Software Category Model for Graphical User Interface Architectures // The Ninth International Conference on Software Engineering Advances (ICSEA 2014). IARIA, 2014. Pp. 123–133.

6. Гибкая методология разработки программного обеспечения. Москва: ИНТУИТ, 2016. 153 с. [Электронный ресурс]: URL: https://e.lanbook.com/book/100590 (дата обращения: 16.11.2024).

7. *Wiegers K., Hokanson C.* Software Requirements Essentials: Core Practices for Successful Business Analysis. Addison-Wesley, 2023. 208 p.

8. *Alles M., Crosby D., Erickson C. et al.* Presenter first: organizing complex GUI applications for test-driven development. // AGILE 2006 (AGILE'06). Minneapolis, MN, USA, 2006. Pp 276–288.

9. *Smialek M., Nowakowski W.* From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice. Springer International Publishing, 2015. 352 p.

10. *Esbai R., Erramdani M.* Model-to-model transformation in approach by modeling: From uml model to model-view-presenter and dependency injection patterns // Information and Communication Technologies (WICT), 2015 5th World Congress on. IEEE, 2015. Pp. 1–6.

11. *Hue1 C.T.M., Hanh D.D., Binh N.N. Duc L.M.* USL: A Domain-Specific Language for Precise Specification of Use Cases and Its Transformations // Informatica (Slovenia), 2018. Vol. 42. Pp.325–343.

12. *Бек К.* Экстремальное программирование: разработка через тестирование. СПб.: Питер, 2022. 224 с.

13. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приёмы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2015. 368 с.

Поступила 17 января 2025 г.

English

SYSTEM STRUCTURING OF THE MODEL-VIEW-PRESENTER PATTERN

Ilya Roidovich Dubov — Grand Dr. in Engineering, Professor, Federal State Budgetary Educational Institution of Higher Professional Education “Vladimir State University named after A.G. and N.G. Stoletovs”.

E-mail: dubov@vlsu.ru

Address: 602264, Russian Federation, Vladimir region, Murom, Orlovskaya St., 23.

Abstract: The work demonstrates that the components of the Model-View-Presenter (MVP) architectural pattern can be viewed as parts of a software system consisting of the user interface subsystem and the business logic subsystem. The View is included in the first subsystem, while the Model belongs to the second. The Presenter simultaneously belongs to both subsystems, acting as their boundary. It implements the interaction between the View and the Model in a procedural manner, in accordance with the scenario defined by the functional requirements to the system. The granularity of the structure of the View and the structure of the Model is determined by the code blocks of the Presenter, with each code block serving as a foundation for forming the MVP tri-ad. The "Include" relationship of a use case is programmatically realized as a procedural call to the corresponding Presenter. The possibility of applying various views, such as those based on console input-output or GUI, is demonstrated without altering the Presenter.

Keywords: system, decomposition, use case, scenario, view, business logic.

References

1. *Syromiatnikov A., Weyns D.* A Journey through the Land of Model-View-Design Patterns. IEEE/IFIP Conference on Software Architecture, Sydney, NSW, Australia, 2014. Pp. 21–30.

2. *Fowler M., Rice D., Fommel M.* Enterprise Application Patterns. Moscow: Dialectics, 2020. 544 p.

3. *Karagkasidis A.* Developing GUI applications: architectural patterns revisited. The Thirteenth Annual European Conference on Pattern Languages of Programming (EuroPLoP 2008), Irsee, Germany, 2008.

4. *Fowler M.* GUI Architectures. [Electronic Source]: URL: https://martinfowler.com/eaaDev/uiArchs.html (Access Date: 16.11.2024).

5. *Wendler S., Streitferdt D.* A Software Category Model for Graphical User Interface Architectures. The Ninth International Conference on Software Engineering Advances (ICSEA 2014). IARIA, 2014. Pp. 123–133.

6. Agile methodology of software development. Moscow: INTUIT, 2016. 153 p. [Electronic Source]: URL: https://e.lanbook.com/book/100590 (Access Date: 16.11.2024).

7. *Wiegers K., Hokanson C.* Software Requirements Essentials: Core Practices for Successful Business Analysis. Addison-Wesley, 2023. 208 p.

8. *Alles M., Crosby D., Erickson C. et al.* Presenter first: organizing complex GUI applications for test-driven development. AGILE 2006 (AGILE’06). Minneapolis, MN, USA, 2006. Pp. 276–288.

9. *Smialek M., Nowakowski W.* From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice. Springer International Publishing, 2015. 352 p.

10. *Esbai R., Erramdani M.* Model-to-model transformation in approach by modeling: From uml model to model-view-presenter and dependency injection patterns. Information and Communication Technologies (WICT), 2015 5th World Congress on. IEEE, 2015. Pp. 1–6.

11. *Hue1 C.T.M., Hanh D.D., Binh N.N. Duc L.M.* USL: A Domain-Specific Language for Precise Specification of Use Cases and Its Transformations. Informatica (Slovenia), 2018. Vol. 42. Pp. 325–343.

12. *Beck K.* Extreme programming: development through testing. SPb.: Piter, 2022. 224 p.

13. *Gamma E., Helm R., Johnson R., Vlissides J.* Object-oriented design techniques. Design patterns. St. Petersburg: Piter, 2015. 368 p.