Лабораторная работа №5

Исследование способов применения структурных паттернов проектирования при рефакторинге ПО

1. Цель работы

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

2. Основные положения

2.1. Паттерны проектирования

Паттерны (шаблоны) проектирования [3] представляют собой инструмент, который позволяет документировать опыт разработки объектно-ориентированных программ. В основе использования паттернов лежит следующая идея: при проектировании каждый проект не разрабатывается с нуля, а используется опыт предыдущих проектов. То есть паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем.

Во многих объектно-ориентированных системах встречаются повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед.

Описание каждого паттерна принято разбивать на следующие разделы:

- **Название и классификация паттерна.** Название паттерна должно четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая будет рассмотрена ниже.
- **Назначение.** Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.
- **Известен также под именем.** Другие распространенные названия паттерна, если таковые имеются.
- **Мотивация.** Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.
- **Применимость.** Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций.

- **Структура.** Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT). Могут использоваться также диаграммы взаимодействий для иллюстрации последовательностей запросов и отношений между объектами.

- **Участники.** Классы или объекты, задействованные в данном паттерне проектирования, и их функции.

- **Отношения.** Взаимодействие участников для выполнения своих функций.

- **Результаты.** Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?

- **Реализация.** Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?

- **Пример кода программы.** Фрагмент программного кода, иллюстрирующий вероятную реализацию на языках C++ или Smalltalk.

- **Известные применения.** Возможности применения паттерна в реальных системах. Даются, по меньшей мере, два примера из различных областей.

- **Родственные паттерны.** Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

2.2. Порядок использования паттернов проектирования

1. Прочитать описание паттерна (см. ниже), чтобы получить о нем общее представление. Особое внимание обратить на разделы «Применимость» и «Результаты». Убедиться, что выбранный паттерн действительно подходит для решения данной задачи.

2. Изучить разделы описания паттерна «Структура», «Участники» и «Отношения». Детально проанализировать назначение упоминаемых в паттерне классов и объектов и то, как они взаимодействуют друг с другом.

3. Посмотреть на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе. Изучение программного кода поможет понять, как нужно реализовывать паттерн.

4. Выбрать для участников паттерна подходящие имена. Имена участников паттерна обычно слишком абстрактны, чтобы употреблять их непосредственно в коде. Тем не менее, бывает полезно включить имя участника как имя в программе. Это помогает сделать паттерн более очевидным при реализации. Например, при использовании паттерна ***Стратегия*** в алгоритме размещения текста, классы могли бы называться `SimpleLayoutStrategy` или `TeXLayoutStrategy`.

5. Определить классы. Объявить их интерфейсы, установить отношения наследования и определить переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты. Выявить

имеющиеся в вашем приложении классы, на которые паттерн оказывает влияние, и соответствующим образом модифицировать их.

6. Определить имена операций, встречающихся в паттерне. Здесь, как и в предыдущем случае, имена обычно зависят от приложения. При этом следует руководствоваться теми функциями и взаимодействиями, которые ассоциированы с каждой операцией. Кроме того, нужно быть последовательным при выборе имен. Например, для обозначения **фабричного метода** можно было бы всюду использовать префикс `Create-`.

7. Реализовать операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне. Советы о том, как это лучше сделать, можно найти в разделе «Реализация». Поможет и «Пример кода».

2.3. Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример – использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот паттерн, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Другой пример паттерна уровня класса – ***Адаптер***. В общем случае Адаптер делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

Вместо композиции интерфейсов или реализаций структурные паттерны уровня объекта компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Примером структурного паттерна уровня объектов является ***Компоновщик***. Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать произвольно сложные структуры из примитивных и других составных объектов.

В паттерне ***Заместитель*** объект берет на себя функции другого объекта. У Заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве. Или представлять большой объект, загружаемый по требованию. Или ограничивать доступ к критически важному объекту. Заместитель вводит дополнительный косвенный уровень доступа к отдельным свойствам объекта. Поэтому он может ограничивать, расширять или изменять эти свойства.

Паттерн ***Приспособленец*** определяет структуру для совместного использования объектов. Владельцы разделяют объекты, по меньшей мере, по двум причинам: для достижения эффективности и непротиворечивости. Приспособленец акцентирует внимание на эффективности использования памяти. В приложениях, в которых участвует очень много объектов, должны снижаться накладные расходы на хранение. Значительной экономии можно добиться за счет разделения объектов вместо их дублирования. Но объект может быть разделяемым, только если его состояние не зависит от контекста. У объектов-приспособленцев такой зависимости нет. Любая дополнительная информация передается им по мере необходимости. В отсутствие контекстных зависимостей объекты-приспособленцы могут легко разделяться.

Если паттерн Приспособленец дает способ работы с большим числом мелких объектов, то ***Фасад*** показывает, как один объект может представлять целую подсистему. Фасад представляет набор объектов и выполняет свои функции, перенаправляя сообщения объектам, которых он представляет. Паттерн ***Мост*** отделяет абстракцию объекта от его реализации, так что их можно изменять независимо.

Паттерн ***Декоратор*** описывает динамическое добавление объектам новых обязанностей. Это структурный паттерн, который рекурсивно компонует объекты с целью реализации заранее неизвестного числа дополнительных функций. Например, объект-декоратор, содержащий некоторый элемент пользовательского интерфейса, может добавить к нему оформление в виде рамки или тени либо новую функциональность, например возможность прокрутки или изменения масштаба. Два разных оформления прибавляются путем простого вкладывания одного декоратора в другой. Для достижения этой цели каждый объект-декоратор должен соблюдать интерфейс своего компонента и перенаправлять ему сообщения. Свои функции (скажем, рисование рамки вокруг компонента) декоратор может выполнять как до, так и после перенаправления сообщения.

2.3.1. Паттерн «Адаптер»

Название и классификация паттерна

Адаптер – паттерн, структурирующий классы и объекты.

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. *Адаптер* обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Известен также под именем

Wrapper (обертка).

Мотивация

Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удается использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет изменяемую форму и изображает сам себя. Интерфейс графических объектов определен абстрактным классом `Shape`. Редактор определяет подкласс класса `Shape` для каждого вида графических объектов: `LineShape` для прямых, `PolygonShape` для многоугольников и т.д.

Классы для элементарных геометрических фигур, например `LineShape` и `PolygonShape`, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс `TextShape`, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс `TextView`, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать `TextView` для реализации `TextShape`, но библиотека разрабатывалась без учета классов `Shape`, поэтому заставить объекты `TextView` и `Shape` работать совместно не удается.

Так каким же образом существующие и независимо разработанные классы вроде `TextView` могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса `TextView`, чтобы он соответствовал интерфейсу `Shape`, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять *`TextView`*; библиотека не должна приспосабливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс `TextShape` так, что он будет адаптировать интерфейс `TextView` к интерфейсу `Shape`. Это допустимо сделать двумя способами:

- наследуя интерфейс от `Shape`, а реализацию от `TextView`;
- включив экземпляр `TextView` в `TextShape` и реализовав `TextShape` в терминах интерфейса `TextView`. Два данных подхода соответствуют вариантам паттерна ***Адаптер*** в его классовой и объектной ипостасях. Класс `TextShape` мы будем называть адаптером.

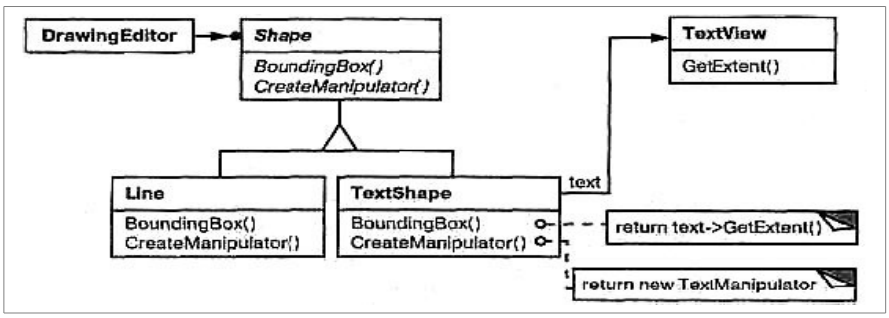

Рисунок 2.1 – Пример использования паттерна Адаптер

На рисунке 2.1 показан адаптер объекта. Видно, как запрос `BoundingBox`, объявленный в классе `Shape`, преобразуется в запрос `GetExtent`, определенный в классе `TextView`. Поскольку класс `TextShape` адаптирует `TextView` к интерфейсу `Shape`, графический редактор может воспользоваться классом `TextView`, хотя тот и имеет несовместимый интерфейс.

Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс. На диаграмме показано, как адаптер выполняет такого рода функции. У пользователя должна быть возможность перемещать любой объект класса `Shape` в другое место, но в классе `TextView` такая операция не предусмотрена. `TextShape` может добавить недостающую функциональность, самостоятельно реализовав операцию `CreateManipulator` класса `Shape`, которая возвращает экземпляр подходящего подкласса `Manipulator`.

`Manipulator` – это абстрактный класс объектов, которым известно, как анимировать `Shape` в ответ на такие действия пользователя, как перетаскивание фигуры в другое место. У класса `Manipulator` имеются подклассы для различных фигур. Например, `TextManipulator` – подкласс для `TextShape`. Возвращая экземпляр `TextManipulator`, объект класса `TextShape` добавляет новую функциональность, которой в классе `TextView` нет, а классу `Shape` требуется.

Применимость

Паттерн *Адаптер* следует применяит, когда:

- необходмио использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- нужно создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектов!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому. Структура адаптера класса показана на рисунке 2.2.

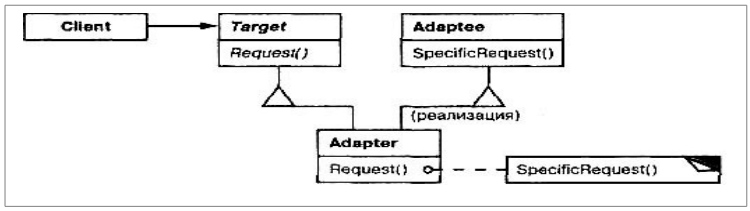

Рисунок 2.2 – Структура адаптера класса

Адаптер объекта применяет композицию объектов. Структура адаптера уровня объектов показана на рисунке 2.3.

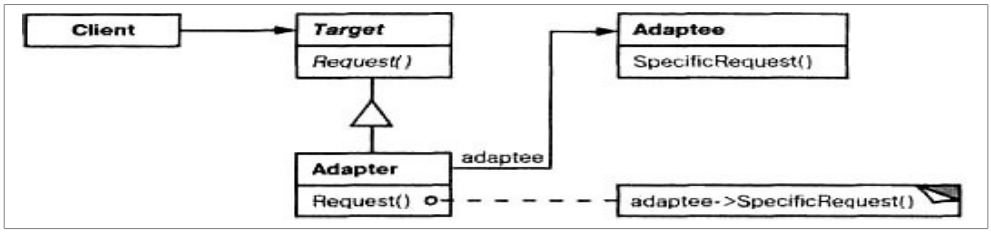

Рисунок 2.3 – Структура адаптера объекта

Участники

- `Target` (`Shape`) – целевой. Определяет зависящий от предметной области интерфейс, которым пользуется `Client`;
- `Client` (`DrawingEditor`) – клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу `Target`;
- `Adaptee` (`Textview`) – адаптируемый: определяет существующий интерфейс, который нуждается в адаптации;
- `Adapter` (`Text Shape`) – адаптер: адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Отношения

Клиенты вызывают операции экземпляра адаптера `Adapter`. В свою очередь адаптер вызывает операции адаптируемого объекта или класса `Adaptee`, который и выполняет запрос.

Результаты

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
- позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

Пример кода программы

Ниже приводится краткий обзор реализации адаптеров класса и объекта для примера, рассмотренного в разделе «Мотивация».

```
class Shape {
    public:
    Shape();
    virtual void BoundingBox(Points bottomLeft, Point& topRight)
const;
    virtual Manipulator* CreateManipulator() const;
};
class TextView {
    public:
    TextView();
    void GetOrigin(Coord& x, Coords y) const;
    void GetExtent(Coord& width, Coords height) const;
    virtual bool IsEmpty() const;
};
```

В классе Shape предполагается, что ограничивающий фигуру прямоугольник определяется двумя противоположными углами. Напротив, в классе TextView он характеризуется начальной точкой, высотой и шириной. В классе Shape определена также операция CreateManipulator() для создания объекта-манипулятора класса Manipulator, который знает, как анимировать фигуру в ответ на действия пользователя. В TextView эквивалентной операции нет. Класс TextShape является адаптером между двумя этими интерфейсами.

Для адаптации интерфейса адаптер класса использует множественное наследование. Принцип адаптера класса состоит в наследовании интерфейса по одной ветви и реализации – по другой. В C++ интерфейс обычно наследуется

открыто, а реализация – закрыто. Мы будем придерживаться этого соглашения при определении адаптера TextShape:

```
class TextShape : public Shape, private TextView {
    public:
    TextShape();
    virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmptyO const;
    virtual Manipulator* CreateManipulator() const;
};
```

Операция BoundingBox преобразует интерфейс TextView к интерфейсу Shape:

```
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

На примере операции IsEmpty демонстрируется прямая переадресация запросов, общих для обоих классов:

```
bool TextShape::IsEmpty () const
{
    return TextView::IsEmpty();
}
```

Наконец, определим операцию CreateManipulator (отсутствующую в классе TextView) с нуля. Предположим, класс TextManipulator, который поддерживает манипуляции с TextShape, уже реализован:

```
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

Адаптер объектов применяет композицию объектов для объединения классов с разными интерфейсами. При таком подходе адаптер TextShape содержит указатель на TextView:

```
class TextShape : public Shape {
    public:
    TextShape(TextView*);
```

```
7
0     virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmptyO const;
    virtual Manipulator* CreateManipulator() const;
    private:
    TextView* _text;
};
```

Объект TextShape должен инициализировать указатель на экземпляр TextView. Делается это в конструкторе. Кроме того, он должен вызывать операции объекта TextView всякий раз, как вызываются его собственные операции.

В этом примере мы предположим, что клиент создает объект TextView и передает его конструктору класса TextShape:

```
TextShape::TextShape (TextView* t)
{
    _text = t;
}
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const
{
    return _text->IsEmpty();
}
```

Реализация CreateManipulator не зависит от версии адаптера класса, поскольку реализована с нуля и не использует повторно никакой функциональности TextView:

```
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

Сравним этот код с кодом адаптера класса. Для написания адаптера объекта нужно потратить чуть больше усилий, но зато он оказывается более гибким. Например, вариант адаптера объекта TextShape будет прекрасно работать и с подклассами TextView: клиент просто передает экземпляр подкласса TextView конструктору TextShape.

3. Порядок выполнения работы

3.1. Ознакомиться с основными преимуществами объектно-ориентированного проектирования на основе паттернов, изучить порядок проектирования с использованием паттернов. Изучить назначение и структуру паттерна ***Адаптер*** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна ***Адаптер***. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент и адаптируемый класс, функциональностью которого должен воспользоваться клиент.

3.3. Выполнить перепроектирование системы, использовав паттерн ***Адаптер***, изменения отобразить на диаграмме классов.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с описанием программного продукта, для которого проводится рефакторинг.

4.3. Словесное описание мотивации применения паттерна ***Адаптер*** при проектировании данной системы.

4.4. UML-диаграммы классов (исходная и модифицированная с использованием паттерна) с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

6. Контрольные вопросы

6.1. Что понимается под паттерном объектно-ориентированного проектирования?

6.2. Из каких разделов состоит описание паттерна?

6.3. Каков общий порядок применения паттернов проектирования?

6.4. Для чего предназначены структурные паттерны проектирования?

6.5. Какие задачи решает паттерн «Адаптер»?

6.6. Какие классы входят в состав паттерна «Адаптер», каковы их обязанности?