Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

РЕФЕРАТ

На тему: «Структурные паттерны: Bridge и его использование для разделения абстракции от реализации»

Выполнил:
Диченко Дина Алексеевна
3 курс, группа ИВТ-б-о-21-1,
09.03.01 – Информатика и вычислительная техника, профиль (профиль)
09.03.01 – Информатика и вычислительная техника, профиль «Автоматизированные системы обработки информации и управления», очная форма обучения

_____
(подпись)

Проверил:
Воронкин Р.А., канд. тех. наук, доцент, доцент кафедры инфокоммуникаций Института цифрового развития,

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Структурные паттерны программирования предоставляют эффективные решения для построения сложных структур программного обеспечения. Одним из таких паттернов является Bridge, который играет ключевую роль в разделении абстракции от реализации, повышая гибкость и улучшая структуру кода.

Актуальность: Современные программные проекты становятся все более сложными и масштабными. С развитием технологий и расширением функциональности программ, важно иметь гибкую архитектуру, которая обеспечивает легкость изменений и поддержки. Bridge позволяет эффективно разделять абстракцию от реализации, что упрощает поддержку кода и обеспечивает гибкость в изменении одной из сторон без воздействия на другую.

Цель: подробно изучить паттерн Bridge, его сущности и механизмы действия, разобраться в принципах, которые лежат в основе разделения абстракции и реализации с использованием Bridge-паттерна.

Задачи:

- Анализ проблем разработки без Bridge: Изучить сценарии, где отсутствие Bridge может привести к сложностям в изменении кода, добавлении новых функциональностей и поддержке проекта.
- Рассмотрение структуры и примеров использования Bridge: Разобраться с основной структурой Bridge-паттерна и предоставить примеры его применения в реальных проектах.
- Сравнение с другими паттернами: Сравнить Bridge с другими структурными паттернами, такими как Adapter, чтобы выделить особенности и преимущества каждого.
- Применение в различных областях: Исследовать, как Bridge-паттерн может быть применен в различных областях разработки, таких как разработка пользовательского интерфейса, работа с базами данных и т. д.

1. ОСНОВНЫЕ ПОНЯТИЯ И ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

1.1 Паттерны проектирования. Общее описание паттерна Bridge

Шаблон проектирования программного обеспечения - это общее, многоразовое решение часто встречающейся проблемы в данном контексте при проектировании программного обеспечения. Это не готовый проект, который можно преобразовать непосредственно в исходный или машинный код. Скорее, это описание или шаблон решения проблемы, который может быть использован во многих различных ситуациях. Паттерны проектирования оформляются лучшие практики, которые программист может использовать для решения общих проблем при разработке приложения или системы.

Объектно-ориентированные шаблоны проектирования обычно показывают отношения и взаимодействия между классами или объектами, без указания конечных классов приложения или объектов, которые задействованы. Шаблоны, подразумевающие изменяемое состояние, могут не подходить для языков функционального программирования. Некоторые шаблоны могут оказаться ненужными в языках, которые имеют встроенную поддержку для решения проблемы, которую они пытаются решить, а объектно-ориентированные шаблоны не обязательно подходят для не объектно-ориентированных языков.

Структурные шаблоны проектирования - это шаблоны проектирования, которые упрощают проектирование, определяя простой способ реализации взаимосвязей между объектами.

Паттерн Bridge (Мост) — структурный шаблон проектирования. То есть, его основная задача — создание полноценной структуры из классов и объектов. Bridge решает эту задачу путем разделения одного или нескольких классов на отдельные иерархии — абстракцию и реализацию. Изменение функционала в одной иерархии не влечет за собой изменения в другой.

1.2 Принцип работы паттерна Bridge и его основные компоненты

Принцип работы Bridge основан на разделении абстракции и реализации с целью достижения гибкости и независимости изменений в обеих сторонах. Давайте рассмотрим ключевые аспекты принципа работы этого структурного паттерна:

- Разделение интерфейса:

Bridge разделяет абстракцию (высокоуровневый интерфейс) и реализацию (низкоуровневый интерфейс).

Высокоуровневый интерфейс предоставляет абстракцию и определяет методы, связанные с концептуальной частью функциональности.

Низкоуровневый интерфейс предоставляет конкретную реализацию, связанную с деталями выполнения функциональности.

- Связь через композицию:

Абстракция содержит ссылку на объект реализации, что позволяет ей делегировать часть своих обязанностей объекту реализации.

Делегирование осуществляется через композицию, не создавая жесткой зависимости между абстракцией и реализацией.

- Независимость изменений:

Изменения в абстракции не влияют на реализацию, и наоборот. Это означает, что можно вносить изменения в одну из сторон без изменения другой, обеспечивая гибкость кода.

Например, при добавлении нового функционала в абстракцию не требуется изменять код реализации.

- Совместимость с различными реализациями:

Bridge обеспечивает совместимость между разными реализациями и абстракциями, что позволяет легко добавлять новые реализации или абстракции без изменения существующего кода.

Основные компоненты Bridge-паттерна:

1. Abstraction (Абстракция).

Абстракция представляет высокоуровневый интерфейс или абстрактный класс, который определяет интерфейс для клиентского кода. Этот интерфейс отражает концептуальную часть функциональности. Содержит ссылку на объект реализации, что позволяет ей делегировать выполнение некоторых задач объекту реализации.

2. Refined Abstraction (Уточненная абстракция).

Уточненная абстракция представляет собой конкретный подкласс абстракции и является одним из ключевых компонентов Bridge-паттерна. Этот класс расширяет базовую абстракцию, предоставляя более конкретные реализации или добавляя новые методы. Уточненная абстракция обеспечивает

3. Implementor (Реализатор).

Реализация представляет низкоуровневый интерфейс, который определяет конкретную реализацию. Этот интерфейс связан с деталями выполнения функциональности.

клиентам более точные средства взаимодействия с абстракцией.

4. Concrete Implementor (Конкретный реализатор).

Классы конкретной реализации реализуют интерфейс реализации, предоставляя конкретные детали выполнения функциональности.

5. Client (Клиентский код)

Клиентский код создает объекты абстракции и реализации, связывает их между собой и использует абстракцию для выполнения функциональности.

1.3 Применимость паттерна Bridge

Используйте паттерн мост, когда:

— хотите избежать постоянной привязки абстракции к реализации. Так, на пример, бывает, когда реализацию необходимо выбирать во время выполнения программы;

— и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;

— изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

— (только для C++!) вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;

— число классов начинает быстро расти. Это признак того, что иерархию следует разделить на две части. Для таких иерархий классов Рамбо (Rumbaugh) использует термин «вложенные обобщения»;

— вы хотите разделить одну реализацию между несколькими объектами (быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента. Простой пример – это разработанный Джеймсом Коплиеном класс String, в котором разные объекты могут разделять одно и то же представление строки (StringRep).

2. РЕАЛИЗАЦИЯ ПАТТЕРНА BRIDGE

2.1 Основные вопросы реализации

Если вы предполагаете применить паттерн мост, то подумайте о таких вопросах реализации:

— только один класс Implementor. В ситуациях, когда есть только одна реализация, создавать абстрактный класс Implementor необязательно. Это вырожденный случай паттерна мост – между классами Abstraction и Implementor существует взаимно-однозначное соответствие. Тем не менее разделение все же полезно, если нужно, чтобы изменение реализации класса не отражалось на существующих клиентах (должно быть достаточно заново скомпоновать программу, не перекомпилируя клиентский код). Для описания такого разделения Каролан (Carolan) употребляет сочетание «чеширский кот». В C++ интерфейс класса Implementor можно определить в закрытом заголовочном файле, который не передается клиентам. Это позволяет полностью скрыть реализацию класса от клиентов;

— создание правильного объекта Implementor. Как, когда и где принимается решение о том, какой из нескольких классов Implementor инстанцировать? Если у класса Abstraction есть информация о конкретных классах ConcreteImplementor, то он может инстанцировать один из них в своем кон структуре; какой именно – зависит от переданных конструктору параметров

Так, если класс коллекции поддерживает несколько реализаций, то решение можно принять в зависимости от размера коллекции. Для небольших коллекций применяется реализация в виде связанного списка, для больших – в виде хэшированных таблиц.

Другой подход – заранее выбрать реализацию по умолчанию, а позже изменять ее в соответствии с тем, как она используется. Например, если число элементов в коллекции становится больше некоторой условной величины, то

мы переключаемся с одной реализации на другую, более эффективную. Можно также делегировать решение другому объекту. В примере с иерархия ми Window/WindowImp уместно было бы ввести фабричный объект (см. пат терн абстрактная фабрика), единственная задача которого – инкапсулировать платформенную специфику. Фабрика обладает информацией, объекты WindowImp какого вида надо создавать для данной платформы, а объект Window просто обращается к ней с запросом о предоставлении какого-нибудь объекта WindowImp, при этом понятно, что объект получит то, что нужно. Преимущество описанного подхода: класс Abstraction напрямую не привязан ни к одному из классов Implementor;

— разделение реализаторов. Джеймс Коплиен показал, как в C++ можно применить идиому описатель/тело, чтобы несколькими объектами могла совместно использоваться одна и та же реализация. В теле хранится счетчик ссылок, который увеличивается и уменьшается в классе описателя. Код для присваивания значений описателям, разделяющим одно тело, в общем виде выглядит так:

```
Handle& Handle::operator= (const Handle& other) {
        other._body->Ref();
        _body->Unref();

        if (_body->RefCount() == 0) {
        delete _body;
        }
        _body = other._body;
        return *this;
}
```

— использование множественного наследования. В C++ для объединения интерфейса с его реализацией можно воспользоваться множественным наследованием. Например, класс может открыто наследовать классу Abstraction и закрыто – классу ConcreteImplementor. Но такое решение

зависит от статического наследования и жестко привязывает реализацию к ее интерфейсу. Поэтому реализовать настоящий мост с помощью множественного наследования невозможно, по крайней мере в C++.

2.2 Реализация паттерна с использованием псевдокода

Задача: необходимо нарисовать графические фигуры (круг и квадрат) на различных устройствах (на экране и в файле).

Этот пример демонстрирует, как Bridge-паттерн позволяет отделить абстракцию (графические фигуры) от реализации (методы рисования на экране или в файле). Классы CircleShape и SquareShape предоставляют уточненные абстракции, которые используют конкретные реализации ScreenDrawingAPI и FileDrawingAPI. При необходимости можно легко добавить новые фигуры или новые методы рисования, не затрагивая другие части кода.

Псевдокод:

```
interface DrawingAPI {
    method drawCircle(x, y, radius)
    method drawSquare(x, y, side)
}

class ScreenDrawingAPI implements DrawingAPI {
    method drawCircle(x, y, radius):
        // Логика рисования круга на экране

    method drawSquare(x, y, side):
        // Логика рисования квадрата на экране

class FileDrawingAPI implements DrawingAPI {
    method drawCircle(x, y, radius):
```

```
        // Логика записи круга в файл

    method drawSquare(x, y, side):
        // Логика записи квадрата в файл
}

# Abstraction
abstract class Shape {
    protected DrawingAPI drawingAPI

    constructor(drawingAPI):
        this.drawingAPI = drawingAPI

    abstract method draw()
    abstract method resize(factor)
}

# Refined Abstraction
class CircleShape extends Shape {
    private x, y, radius

    constructor(x, y, radius, drawingAPI):
        super(drawingAPI)
        this.x = x
        this.y = y
        this.radius = radius

    method draw():
        drawingAPI.drawCircle(x, y, radius)
```

```
    method resize(factor):
        radius *= factor
}

# Refined Abstraction
class SquareShape extends Shape {
    private x, y, side

    constructor(x, y, side, drawingAPI):
        super(drawingAPI)
        this.x = x
        this.y = y
        this.side = side

    method draw():
        drawingAPI.drawSquare(x, y, side)

    method resize(factor):
        side *= factor
}

# Пример использования
drawingOnScreen = new ScreenDrawingAPI()
drawingInFile = new FileDrawingAPI()

circleOnScreen = new CircleShape(1, 2, 3, drawingOnScreen)
circleOnScreen.draw()

squareInFile = new SquareShape(5, 6, 4, drawingInFile)
squareInFile.draw()
```

Рассмотрим по подробнее шаги реализации:

1. Создание интерфейса (DrawingAPI).

Этот интерфейс представляет собой "реализатор" (Implementor) в терминах Bridge-паттерна. Он содержит методы для рисования круга и квадрата. Каждый из этих методов принимает координаты и размеры фигуры.

2. Конкретные реализации (ScreenDrawingAPI и FileDrawingAPI).

Эти классы представляют конкретные реализации интерфейса DrawingAPI. Они содержат реальные алгоритмы рисования или записи в файл для каждой фигуры.

3. Абстракция (Shape):

Этот класс представляет собой "абстракцию" (Abstraction). Он содержит ссылку на объект типа DrawingAPI и определяет абстрактные методы draw и resize, которые будут реализованы в конкретных уточненных абстракциях.

4. Уточненные абстракции (CircleShape и SquareShape):

Эти классы представляют уточненные абстракции (Refined Abstraction). Они наследуются от Shape и реализуют методы draw и resize, используя конкретные реализации DrawingAPI для выполнения действий над фигурами.

5. Пример использования.

Этот код создает объект ScreenDrawingAPI, затем создает объект CircleShape, используя ScreenDrawingAPI в качестве реализации, и вызывает метод draw для отображения круга на экране.

Таким образом, структура и взаимодействие компонентов в этом псевдокоде отражают ключевые концепции Bridge-паттерна, где абстракция и реализация разделены, обеспечивая гибкость и возможность легкого расширения системы.

2.3 Реализация паттерна Bridge в Python

В Python нет явного указания типов переменных, что делает код более гибким. Использование паттерна "Bridge" в Python позволяет более свободно комбинировать различные реализации и абстракции.

Пример реализации паттерна в Python:

```
# Implementor
class DrawingAPI:
    def draw_circle(self, x, y, radius):
        pass

    def draw_square(self, x, y, side):
        pass

# Concrete Implementors
class DrawingAPI1(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API1: Drawing a circle at ({x}, {y}) with radius {radius}")

    def draw_square(self, x, y, side):
        print(f"API1: Drawing a square at ({x}, {y}) with side {side}")

class DrawingAPI2(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API2: Drawing a circle at ({x}, {y}) with radius {radius}")

    def draw_square(self, x, y, side):
        print(f"API2: Drawing a square at ({x}, {y}) with side {side}")

# Abstraction
```

```
class Shape:
    def __init__(self, drawing_api):
        self.drawing_api = drawing_api

    def draw(self):
        pass

    def resize(self, factor):
        pass

# Refined Abstraction
class CircleShape(Shape):
    def __init__(self, x, y, radius, drawing_api):
        super().__init__(drawing_api)
        self.x = x
        self.y = y
        self.radius = radius

    def draw(self):
        self.drawing_api.draw_circle(self.x, self.y, self.radius)

    def resize(self, factor):
        self.radius *= factor

# Refined Abstraction
class SquareShape(Shape):
    def __init__(self, x, y, side, drawing_api):
        super().__init__(drawing_api)
        self.x = x
        self.y = y
```

```
        self.side = side

    def draw(self):
        self.drawing_api.draw_square(self.x, self.y, self.side)

    def resize(self, factor):
        self.side *= factor

# Пример использования
api1 = DrawingAPI1()
api2 = DrawingAPI2()

circle = CircleShape(1, 2, 3, api1)
circle.draw()
circle.resize(2)
circle.draw()

square = SquareShape(5, 6, 4, api2)
square.draw()
square.resize(1.5)
square.draw()
```

Основная идея заключается в том, что Shape (абстракция) содержит ссылку на DrawingAPI (реализацию), и эти два аспекта могут развиваться независимо друг от друга. Например, вы можете легко добавить новую форму или новую реализацию, не меняя остальной код.

2.4 Использования паттерна Bridge в других языках программирования

Пример применения паттерна на языке Java:

```
// Implementor
```

```
interface DrawingAPI {
    void drawCircle(int x, int y, int radius);
    void drawSquare(int x, int y, int side);
}

// Concrete Implementors
class DrawingAPI1 implements DrawingAPI {
    // реализация методов drawCircle и drawSquare
}

class DrawingAPI2 implements DrawingAPI {
    // реализация методов drawCircle и drawSquare
}

// Abstraction
abstract class Shape {
    protected DrawingAPI drawingAPI;

    public Shape(DrawingAPI drawingAPI) {
        this.drawingAPI = drawingAPI;
    }

    abstract void draw();
    abstract void resize(int factor);
}

// Refined Abstraction
class CircleShape extends Shape {
    private int x, y, radius;
```

```
    public CircleShape(int x, int y, int radius, DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }

    @Override
    void resize(int factor) {
        radius *= factor;
    }
}

// Пример использования
DrawingAPI api1 = new DrawingAPI1();
DrawingAPI api2 = new DrawingAPI2();

Shape circle = new CircleShape(1, 2, 3, api1);
circle.draw();
circle.resize(2);
circle.draw();
```

Пример применения паттерна на языке C++:

```
// Implementor
class DrawingAPI {
public:
```

```
    virtual void drawCircle(int x, int y, int radius) = 0;
    virtual void drawSquare(int x, int y, int side) = 0;
};

// Concrete Implementors
class DrawingAPI1 : public DrawingAPI {
    // реализация методов drawCircle и drawSquare
};

class DrawingAPI2 : public DrawingAPI {
    // реализация методов drawCircle и drawSquare
};

// Abstraction
class Shape {
protected:
    DrawingAPI* drawingAPI;

public:
    Shape(DrawingAPI* api) : drawingAPI(api) {}

    virtual void draw() = 0;
    virtual void resize(int factor) = 0;
};

// Refined Abstraction
class CircleShape : public Shape {
private:
    int x, y, radius;
```

```
public:
    CircleShape(int x, int y, int r, DrawingAPI* api) : Shape(api), x(x), y(y), radius(r) {}

    void draw() override {
        drawingAPI->drawCircle(x, y, radius);
    }

    void resize(int factor) override {
        radius *= factor;
    }
};

// Пример использования
DrawingAPI1 api1;
DrawingAPI2 api2;

CircleShape circle(1, 2, 3, &api1);
circle.draw();
circle.resize(2);
circle.draw();
```

JavaScript использует прототипное наследование, что может отличаться от классического наследования в языках, таких как Java или C++. Это влияет на способ организации кода с использованием паттерна "Bridge".Он является основным языком для веб-разработки, и паттерн "Bridge" может использоваться для организации сложных веб-интерфейсов и взаимодействия с различными браузерами.

Пример применения паттерна на языке JavaScript:

```
// Implementor
class DrawingAPI {
```

```
    drawCircle(x, y, radius) {
        // реализация метода drawCircle
    }

    drawSquare(x, y, side) {
        // реализация метода drawSquare
    }
}

// Abstraction
class Shape {
    constructor(drawingAPI) {
        this.drawingAPI = drawingAPI;
    }

    draw() {
        // абстрактный метод draw
    }

    resize(factor) {
        // абстрактный метод resize
    }
}

// Refined Abstraction
class CircleShape extends Shape {
    constructor(x, y, radius, drawingAPI) {
        super(drawingAPI);
        this.x = x;
        this.y = y;
```

```
        this.radius = radius;
    }

    draw() {
        this.drawingAPI.drawCircle(this.x, this.y, this.radius);
    }

    resize(factor) {
        this.radius *= factor;
    }
}

// Пример использования
const api1 = new DrawingAPI();
const circle = new CircleShape(1, 2, 3, api1);
circle.draw();
circle.resize(2);
circle.draw();
```

3. ПРЕИМУЩЕСТВА И НЕДОСТАТКИ ИСПОЛЬЗОВАНИЯ ПАТТЕРНА BRIDGE

3.1 Преимущества паттерна Bridge

Паттерн "Bridge" предоставляет ряд преимуществ в проектировании программного обеспечения, обеспечивая гибкость, расширяемость и уменьшая связанность между абстракцией и реализацией. Вот подробное рассмотрение преимуществ этого паттерна:

1. Разделение абстракции и реализации.

гибкость и расширяемость: "Bridge" позволяет изменять абстракцию и реализацию независимо друг от друга. Это позволяет добавлять новые абстракции или реализации без изменения существующего кода, что обеспечивает гибкость системы;

множественное наследование: В языках программирования, поддерживающих множественное наследование, "Bridge" позволяет избежать проблем, связанных с наследованием от нескольких классов, поддерживая при этом различные реализации.

2. Сокрытие деталей реализации.

уменьшение связанности: Паттерн позволяет абстрагироваться от деталей реализации, что уменьшает связанность между абстракцией и конкретными реализациями. Это делает код более модульным и легким для понимания;

улучшение безопасности: Сокрытие деталей реализации способствует повышению безопасности, так как клиентский код имеет ограниченный доступ к внутренней реализации.

3. Совместимость с изменениями.

легкость внесения изменений: Изменение абстракции или реализации не влияет на другую сторону. Это делает систему более устойчивой к

изменениям, обеспечивает простоту внесения изменений и поддерживает принцип открытости/закрытости.

4. Переносимость и поддержка множества платформ.

адаптация под различные платформы: Паттерн "Bridge" упрощает создание переносимого кода, так как различные реализации могут быть легко заменены, не затрагивая абстракцию. Это особенно полезно в случаях, когда код должен работать на разных платформах.

5. Улучшенная поддержка тестирования и сопровождения.

изоляция модулей: Разделение абстракции и реализации упрощает тестирование, так как изменения в одной части системы не влияют на другую. Это также облегчает сопровождение и отладку кода.

6. Обеспечение гибкой архитектуры.

архитектурная гибкость: "Bridge" способствует созданию гибкой архитектуры, позволяя легко добавлять новые абстракции и реализации, а также комбинировать их в различных конфигурациях.

3.2 Недостатки паттерна Bridge

Несмотря на множество преимуществ, паттерн "Bridge" также имеет свои недостатки и ограничения. Рассмотрим некоторые из них

1. Усложнение структуры кода:

Реализация "Bridge" может усложнить структуру кода. Создание дополнительных абстракций и реализаций может привести к увеличению числа классов, что, в свою очередь, может затруднить понимание кода.

2. Дополнительные затраты на разработку:

Внедрение паттерна "Bridge" требует создания дополнительных классов и интерфейсов. Это может привести к увеличению объема кода, а следовательно, к дополнительным затратам на разработку.

3. Потенциальное усложнение вызовов методов:

Клиентский код может быть вынужден вызывать методы как на абстракции, так и на реализации, что может затруднить некоторые случаи использования и сделать код менее наглядным.

4. Не всегда подходит для простых систем:

Паттерн "Bridge" имеет смысл в тех случаях, когда система становится сложной и поддерживает различные платформы или изменения. В простых системах он может быть избыточен и усложнить код без необходимости.

5. Возможные излишества в структуре кода:

Иногда использование паттерна "Bridge" может показаться избыточным, особенно если абстракция и реализация всегда согласованы и меняются вместе. В таких случаях "Bridge" может привести к излишествам в структуре кода.

6. Сложность определения оптимальной структуры:

При проектировании системы с применением паттерна "Bridge" может быть сложно определить оптимальную структуру, особенно когда существует множество абстракций и реализаций.

Необходимо оценивать применимость паттерна "Bridge" с учетом конкретных требований и особенностей проекта, чтобы избежать избыточной сложности и обеспечить понятность и эффективность кода.

ЗАКЛЮЧЕНИЕ

Структурный паттерн "Bridge" представляет собой мощный инструмент в объектно-ориентированном проектировании, предназначенный для разделения абстракции от реализации. Этот паттерн обеспечивает гибкость и расширяемость системы, позволяя изменять абстракцию и реализацию независимо друг от друга. Он поддерживает принцип открытости/закрытости, обеспечивает легкость внесения изменений и улучшает переносимость кода между различными платформами.

Применение паттерна "Bridge" особенно полезно в сценариях, где необходимо поддерживать множество вариаций и комбинаций абстракций и реализаций. Этот подход также способствует снижению связанности между компонентами системы, что улучшает ее обслуживаемость и тестируемость.

Осознанное использование паттерна "Bridge" при проектировании программного обеспечения позволяет создавать более гибкие, поддерживаемые и расширяемые системы, что является ключевым аспектом в современной разработке программного обеспечения. Паттерн "Bridge" продемонстрирован в различных языках программирования и успешно применяется в индустрии, делая его важным инструментом для инженеров-программистов.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. 2. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес Приемы объектно-ориентированного проектирования. Паттерны проектирования = Design Patterns: Elements of Reusable Object- Oriented Software. — СПб: «Питер», 2007. — С. 366.
2. Паттерны проектирования на платформе .NET. — СПб.: Питер, 2015. — 320 с.: ил. ISBN 978-5-496-01649-0.
3. 3. Марк Гранд Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML = Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML. — М.: «Новое знание», 2004. — С. 560.
4. 4. Крэг Ларман Применение UML 2.0 и шаблонов проектирования = Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. — М.: «Вильямс», 2006. — С. 736.
5. 5. Джошуа Кериевски Рефакторинг с использованием шаблонов (паттернов проектирования) = Refactoring to Patterns (Addison-Wesley Signature Series). — М.: «Вильямс», 2006. — С. 400.
6. Швец Александр Погружение в паттерны проектирования. Design patterns. Explained simply — 2018.
7. Шаблоны проектирования по-человечески: структурные паттерны (proglib.io)
8. О структурных шаблонах проектирования простым языком (tproger.ru)
9. Паттерн проектирования Мост (Bridge Pattern) (javarush.com)
10. Andrey on .NET | Структурные шаблоны: Мост (Bridge) (moveax.ru)