

Геллерт Илья
Архиповский Алексей



СПбПУ / Java

Паттерн: Адаптер

1. Введение
2. Типы реализации в Java
3. Практические примеры в JDK
4. Преимущества и Недостатки
5. Заключение
6. Дополнительные материалы для изучения



Введение



Паттерн Адаптер (Adapter) - это структурный шаблон проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он создает класс-обертку, который действует как мост, преобразуя интерфейс одного класса в интерфейс, ожидаемый клиентом. Ключевая особенность - это достигается без изменения исходного кода существующих классов.

Типы реализации в Java



Object
Adapter

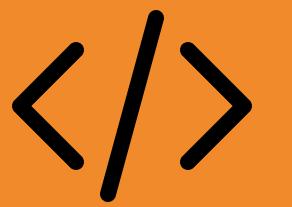
Class
Adapter

Two-Way
Adapter

Interface
Adapter

Адаптер
для
нескольких
Adaptee

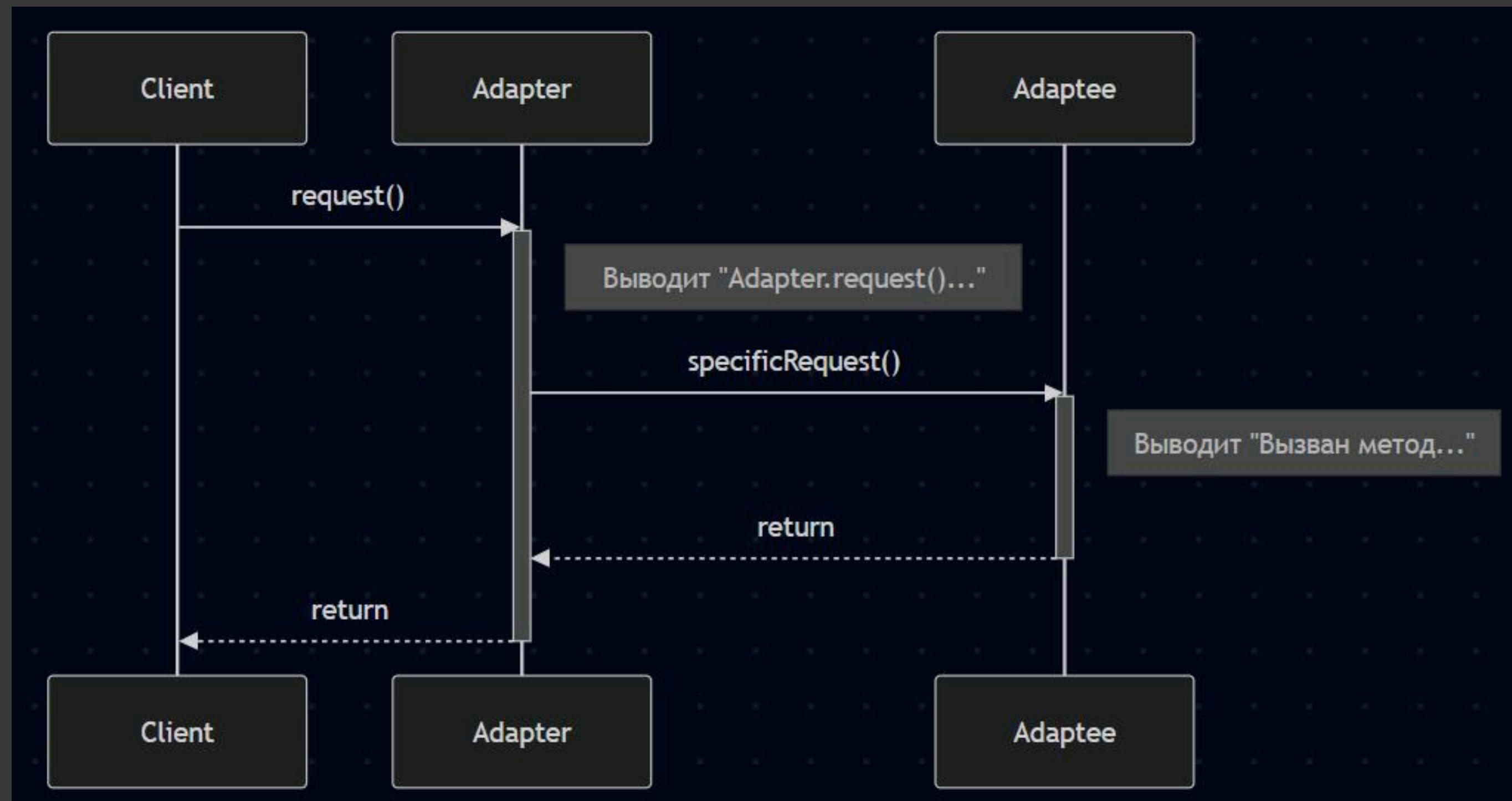
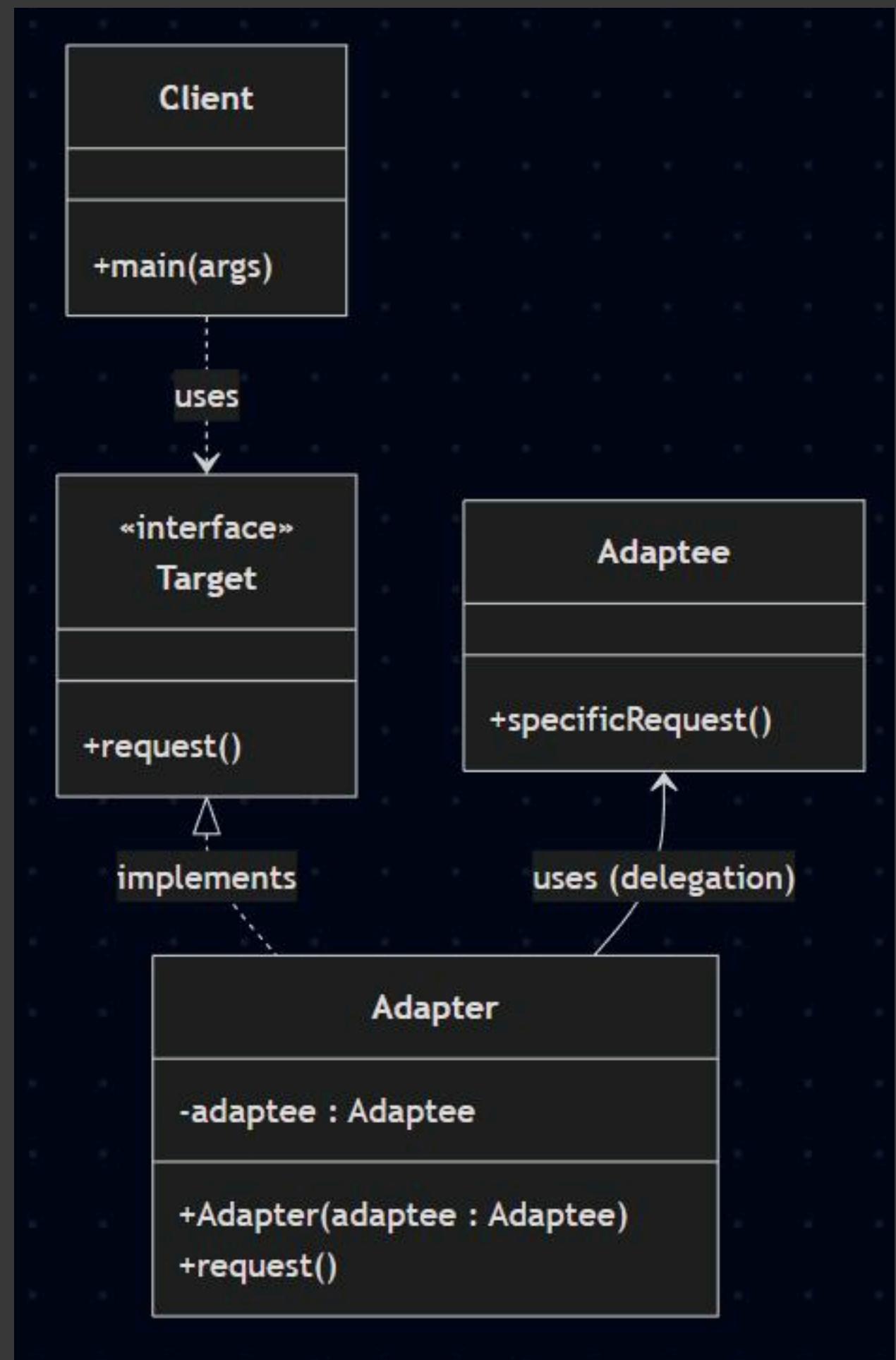
Object Adapter



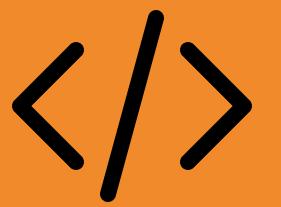
```
1 interface Target {  
2     void request();  
3 }  
4  
5 class Adaptee {  
6     public void specificRequest() {  
7         System.out.println("Вызван метод Adaptee.specificRequest()");  
8     }  
9 }  
10  
11 class Adapter implements Target {  
12     private final Adaptee adaptee;  
13     public Adapter(Adaptee adaptee) {  
14         this.adaptee = adaptee;  
15     }  
16  
17     @Override  
18     public void request() {  
19         System.out.println("Adapter.request() -> транслирует вызов в...");  
20         adaptee.specificRequest();  
21     }  
22 }  
23  
24 class Client {  
25     public static void main(String[] args) {  
26         Adaptee adaptee = new Adaptee();  
27         Target adapter = new Adapter(adaptee);  
28  
29         adapter.request();  
30     }  
31 }
```

Object Adapter

</>



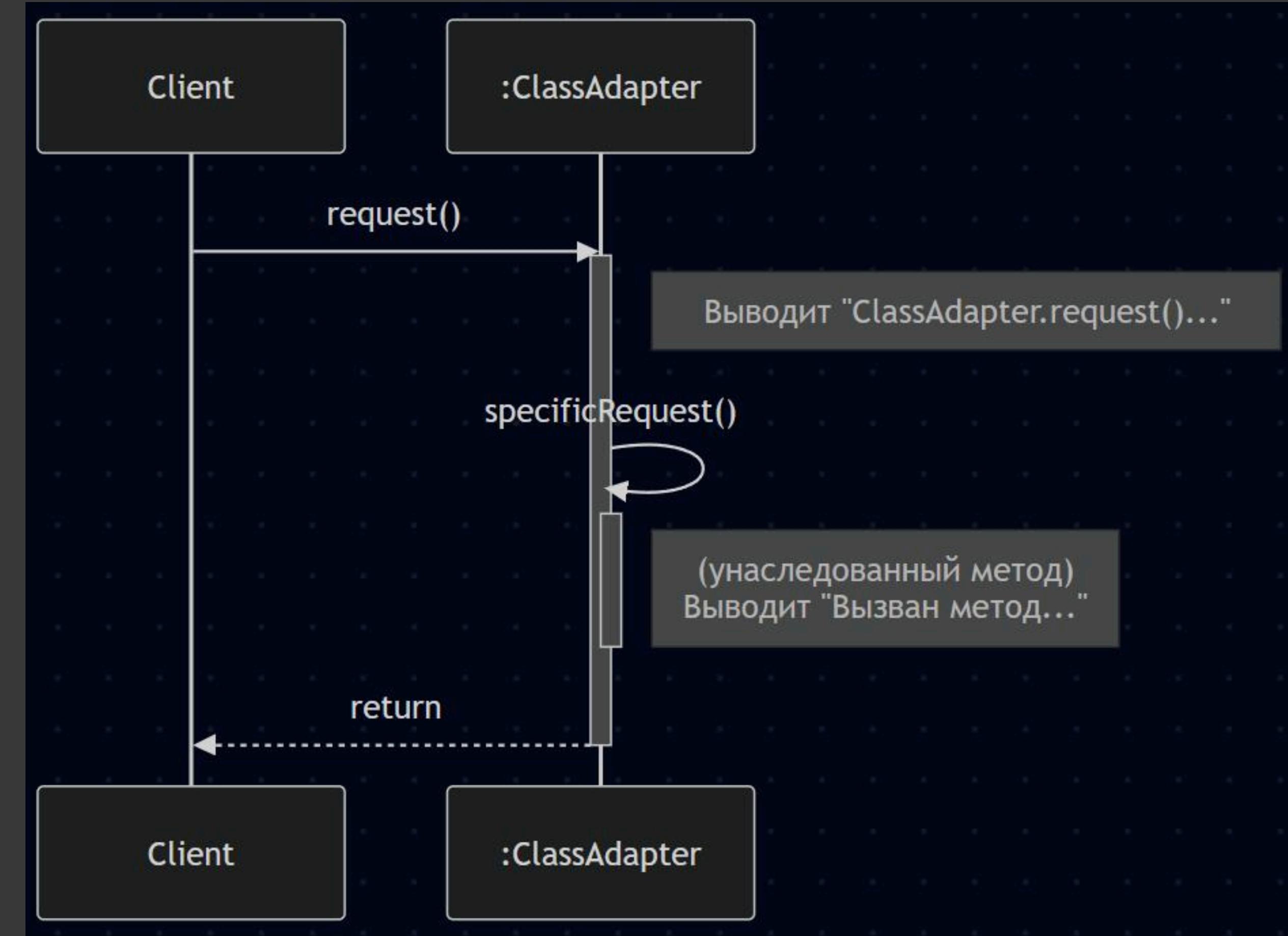
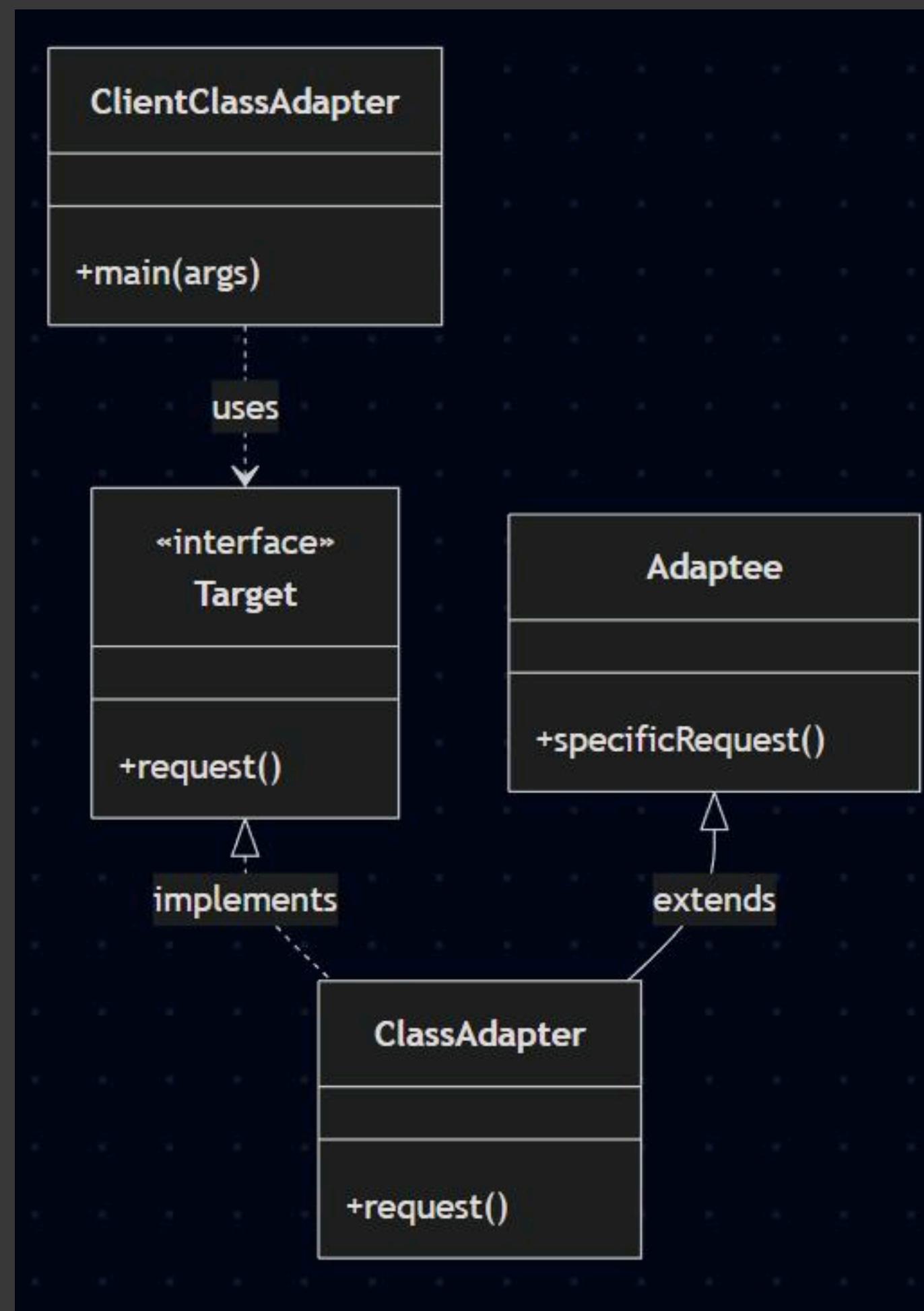
Class Adapter



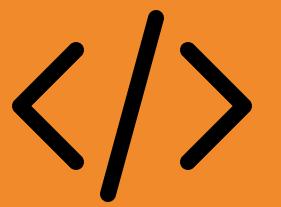
```
1 interface Target {  
2     void request();  
3 }  
4  
5 class Adaptee {  
6     public void specificRequest() {  
7         System.out.println("Вызвын метод Adaptee.specificRequest()");  
8     }  
9 }  
10  
11 class ClassAdapter extends Adaptee implements Target {  
12     @Override  
13     public void request() {  
14         System.out.println("ClassAdapter.request() -> напрямую вызывает specificRequest()");  
15         specificRequest();  
16     }  
17 }  
18  
19 class ClientClassAdapter {  
20     public static void main(String[] args) {  
21         Target adapter = new ClassAdapter();  
22         adapter.request();  
23     }  
24 }
```

</>

Class Adapter



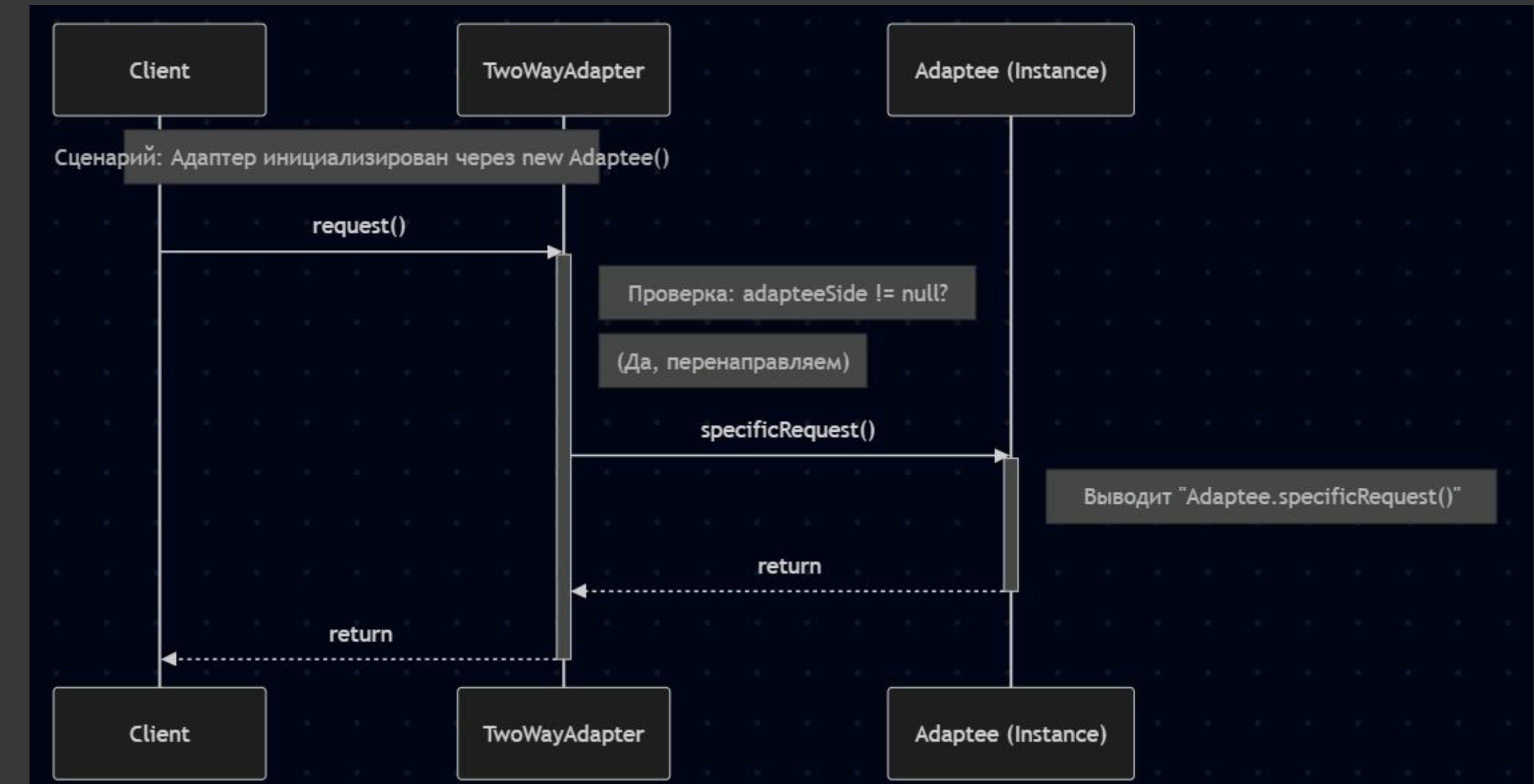
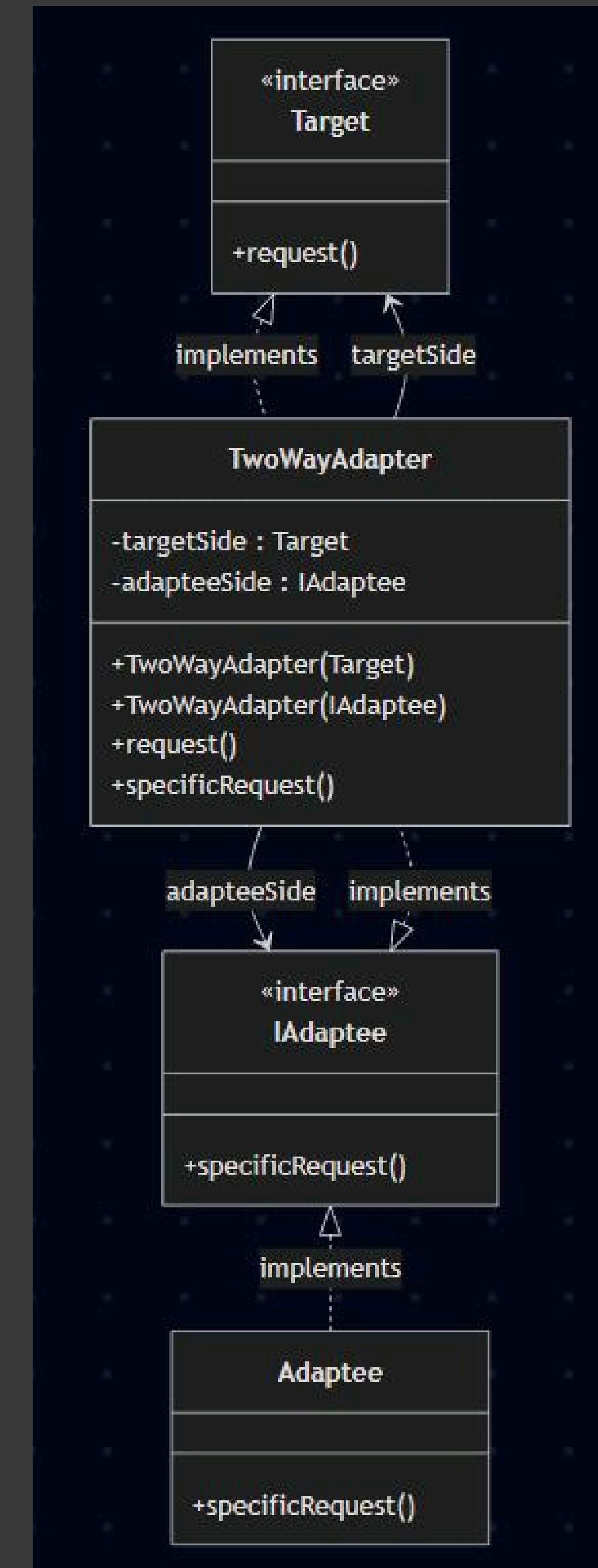
Two-Way Adapter



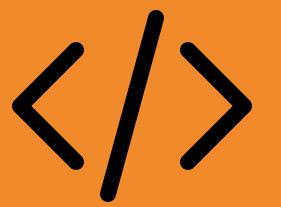
```
1 interface Target {
2     void request();
3 }
4 interface IAdaptee {
5     void specificRequest();
6 }
7 class Adaptee implements IAdaptee {
8     public void specificRequest() {
9         System.out.println("Adaptee.specificRequest()");
10    }
11 }
12 class TwoWayAdapter implements Target, IAdaptee {
13     private final Target targetSide;
14     private final IAdaptee adapteeSide;
15     public TwoWayAdapter(Target t) {this.targetSide = t; this.adapteeSide = null; }
16     public TwoWayAdapter(IAdaptee a) {this.adapteeSide = a; this.targetSide = null; }
17     @Override
18     public void request() {
19         if (adapteeSide != null) {
20             System.out.println("TwoWayAdapter.request() -> вызывает specificRequest()");
21             adapteeSide.specificRequest();
22         } else {
23             System.out.println("TwoWayAdapter.request() -> targetSide.request()");
24             targetSide.request();
25         }
26     }
27     @Override
28     public void specificRequest() {
29         if (targetSide != null) {
30             System.out.println("TwoWayAdapter.specificRequest() -> вызывает targetSide.request()");
31             targetSide.request();
32         } else {
33             adapteeSide.specificRequest();
34         }
35     }
36 }
```

</>

Two-Way Adapter



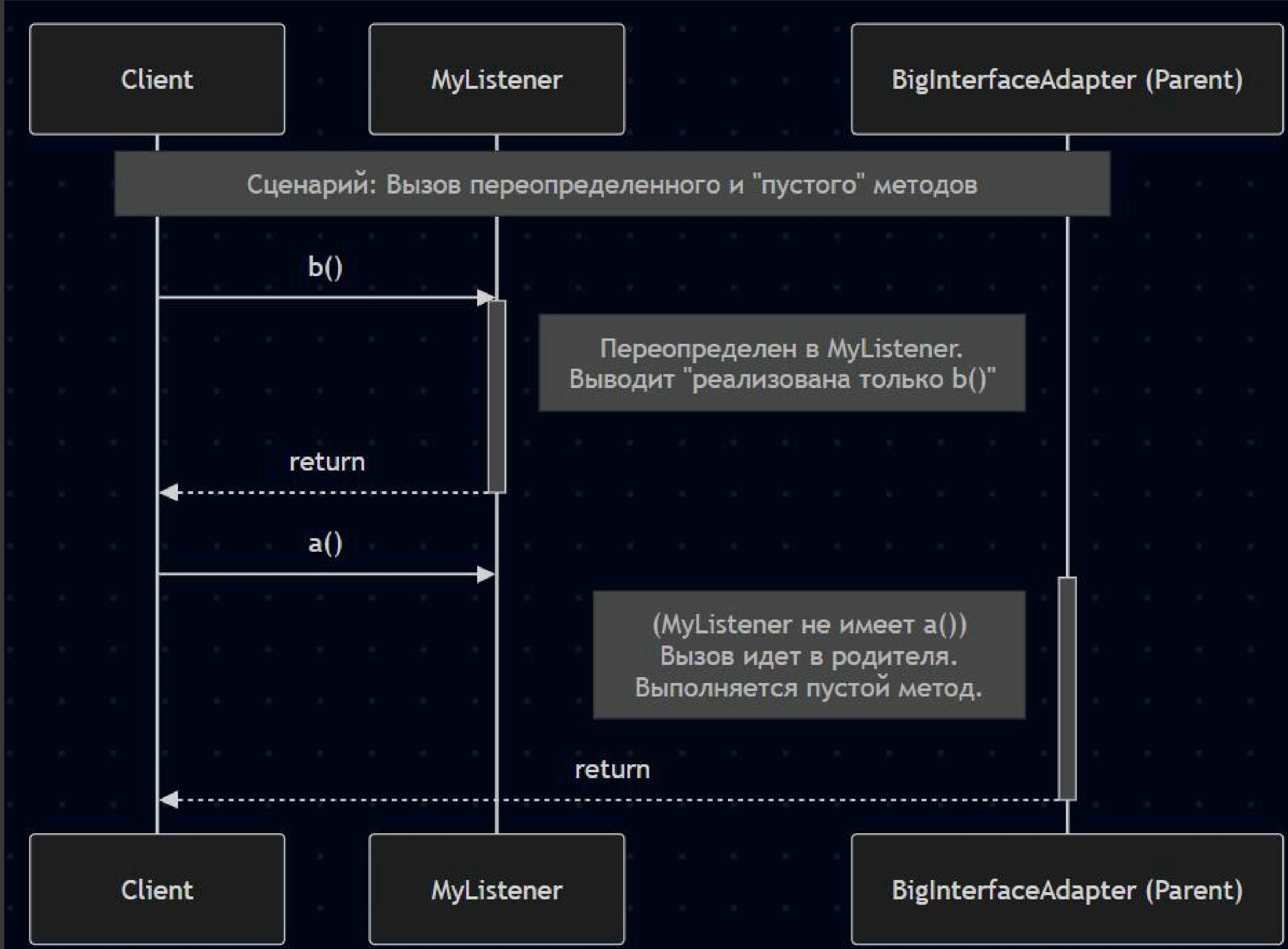
Interface Adapter



```
1 interface BigInterface {  
2     void a(); void b(); void c();  
3 }  
4  
5 abstract class BigInterfaceAdapter implements BigInterface {  
6     @Override public void a() {}  
7     @Override public void b() {}  
8     @Override public void c() {}  
9 }  
10  
11 class MyListener extends BigInterfaceAdapter {  
12     @Override public void b() {  
13         System.out.println("реализована только b()");  
14     }  
15 }
```

Interface Adapter

</>



</>

Адаптер для нескольких Adaptee

Иногда адаптер содержит несколько adaptee (или использует фасет подсистемы) и объединяет их поведение в интерфейс Target. Это — просто расширение Object Adapter: вместо одной ссылки — несколько, и request() распределяет/составляет вызовы.

Практические примеры в JDK

`InputStreamReader /`
`OutputStreamWriter –`
адаптируют байтовые потоки
в символьные и обратно

`Arrays.asList(...) –`
обращивает массив в `List`
(адаптация интерфейса доступа)

`Collections.list(Enumeration) /`
`Collections.enumeration(Collection)`
– преобразование между `Enumeration` и
`Collection/Iterator`



1. Позволяет использовать существующие классы без их изменения — интеграция legacy-кода.
2. Гибкость: адаптер через композицию позволяет подменять реализации во время выполнения.
3. Упрощает повторное использование и тестирование (обёртки легко мокировать).
4. Следует принципам SOLID



1. Дополнительный уровень абстракции — небольшая накладная и усложнение архитектуры.
2. В Java `class adapter` ограничен одиночным наследованием; чаще нужна композиция.
3. Много мелких адаптеров повышает сложность сопровождения и понимания системы.
4. Иногда несовместимости проявляются только во время выполнения (`runtime`)

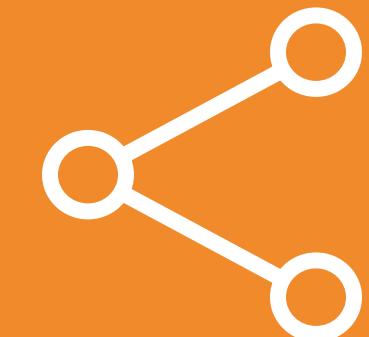
Паттерн «Адаптер» обеспечивает совместимость между несовместимыми интерфейсами и способствует повторному использованию кода. Он изолирует изменения и снижает зависимость компонентов. На практике чаще используется композиционный подход из-за его гибкости и простоты тестирования. Однако чрезмерное количество адаптеров может усложнить архитектуру, поэтому важно применять паттерн осознанно.





"Паттерны проектирования" (Э. Гамма и др.)

"Head First. Паттерны проектирования"
(Э. Фримен и др.)



<https://www.baeldung.com/java-adapter-pattern>

<https://habr.com/ru/articles/85095/>