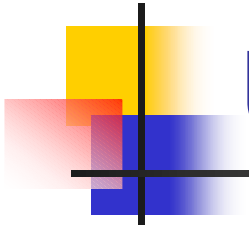
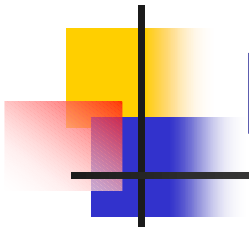
Design Pattern

Adapter



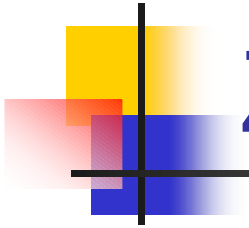
Überblick

- Sinn und Zweck
- Beispielanwendung
- Möglichkeiten der Implementation
- Vor- und Nachteile
- Verwandte Pattern



Das Adapter-Pattern...

- ... ist ein Strukturpattern
- ... ist auch bekannt als „Wrapper“ (Umwickler)

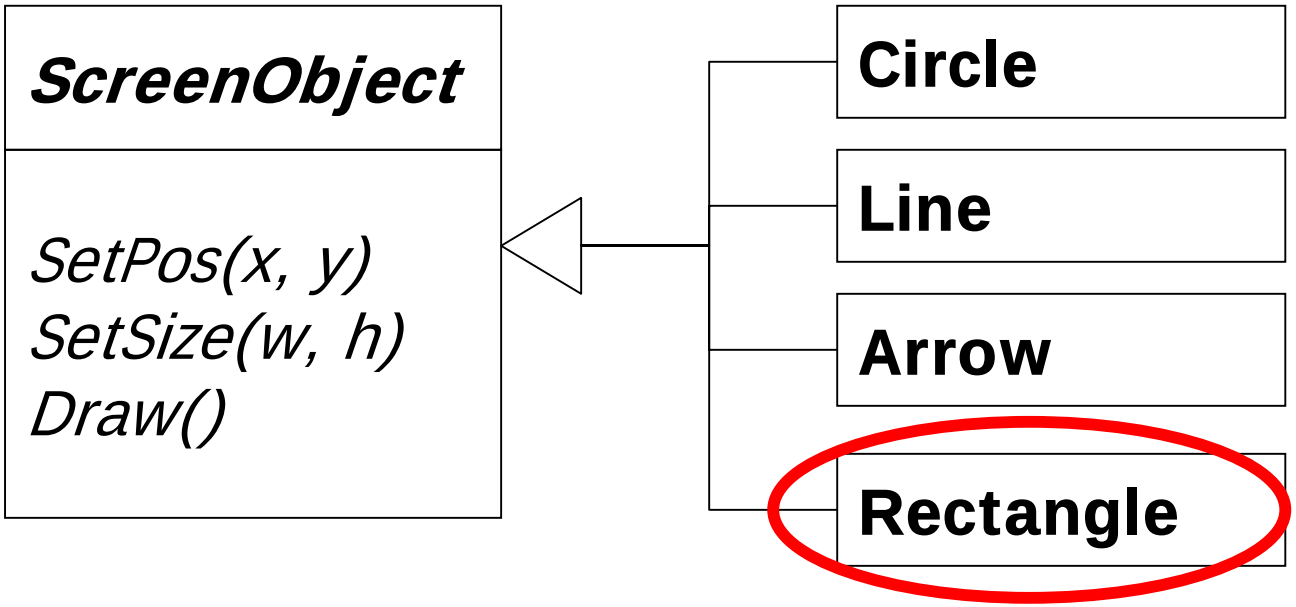


Zweck

- Anpassung der Schnittstelle einer existierenden Klasse an eine andere Schnittstelle
- D.h. die Funktionalität der Klasse ist perfekt, nur das Interface passt nicht.
- Das Interface nur für eine einzelne Anwendung zu ändern wäre unschön
- Quellcode der Klasse liegt evtl. nicht vor

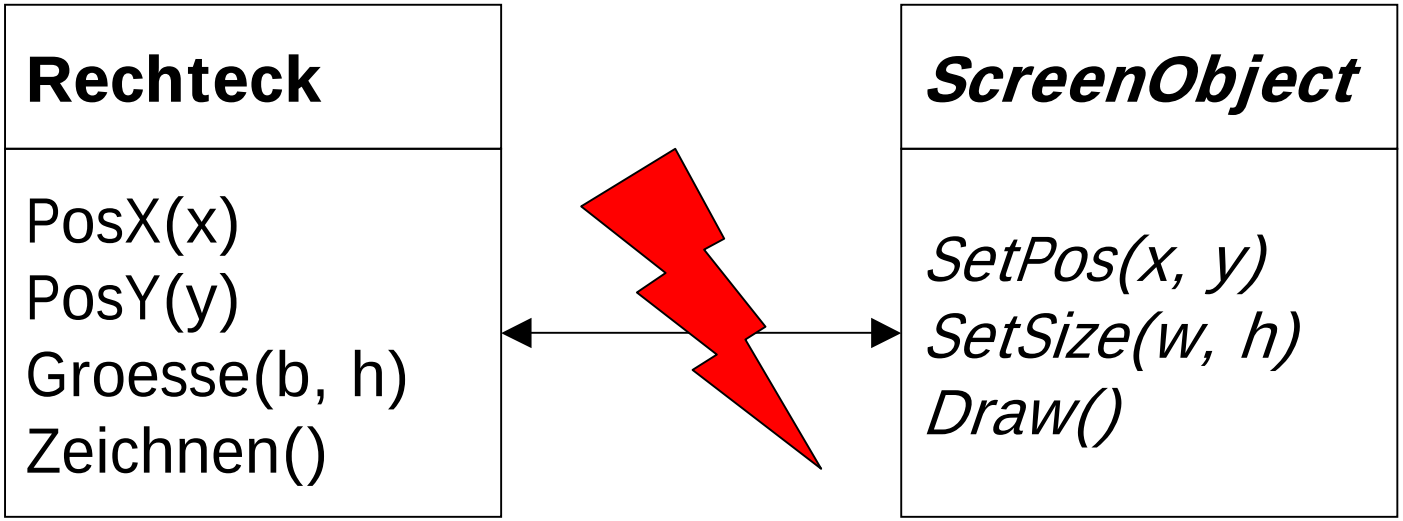
Beispiel: Das Problem

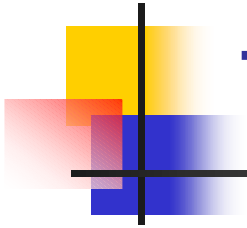
- Es soll ein Präsentationsprogramm geschrieben werden, das verschiedene graphische Elemente mitbringt.



Beispiel: Das Problem

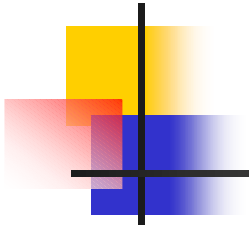
- die Klasse **Rechteck** wäre eine gute Implementation für **Rectangle**





Teilnehmer

- **Ziel** *(ScreenObject)*
  - das gewünschte Interface
- **Klient** *(Präsentationsprogramm)*
  - arbeitet mit Objekten, die Zielinterface haben
- **Adaptierte Klasse** *(Rechteck)*
  - existierende und anzupassende Schnittstelle
- **Adapter** *(Rectangle)*
  - implementiert Ziel mittels adaptierter Klasse



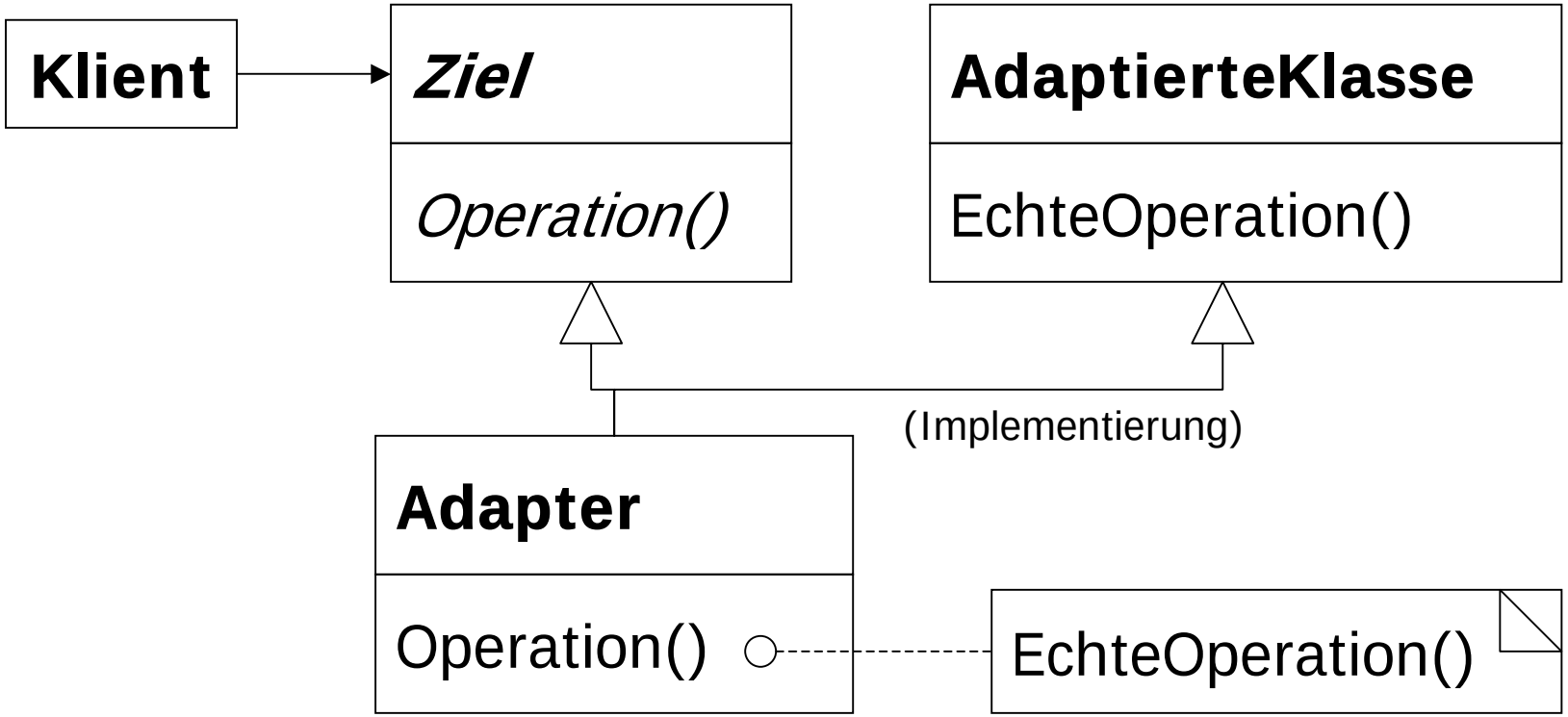
Implementierung des Adapters

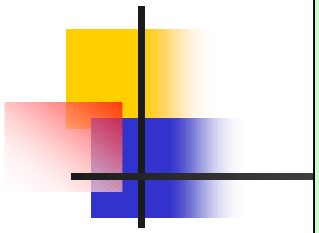
- mittels **Vererbung** oder **Mehrfachvererbung** (Klassenadapter)

- mittels **Aggregation** (Objektadapter)
  - d.h. entweder der Adapter konstruiert eine neue Instanz der zu adaptierenden Klasse oder ihm wird eine Instanz übergeben

Struktur

Implementation mittels (Mehrfach-)Vererbung



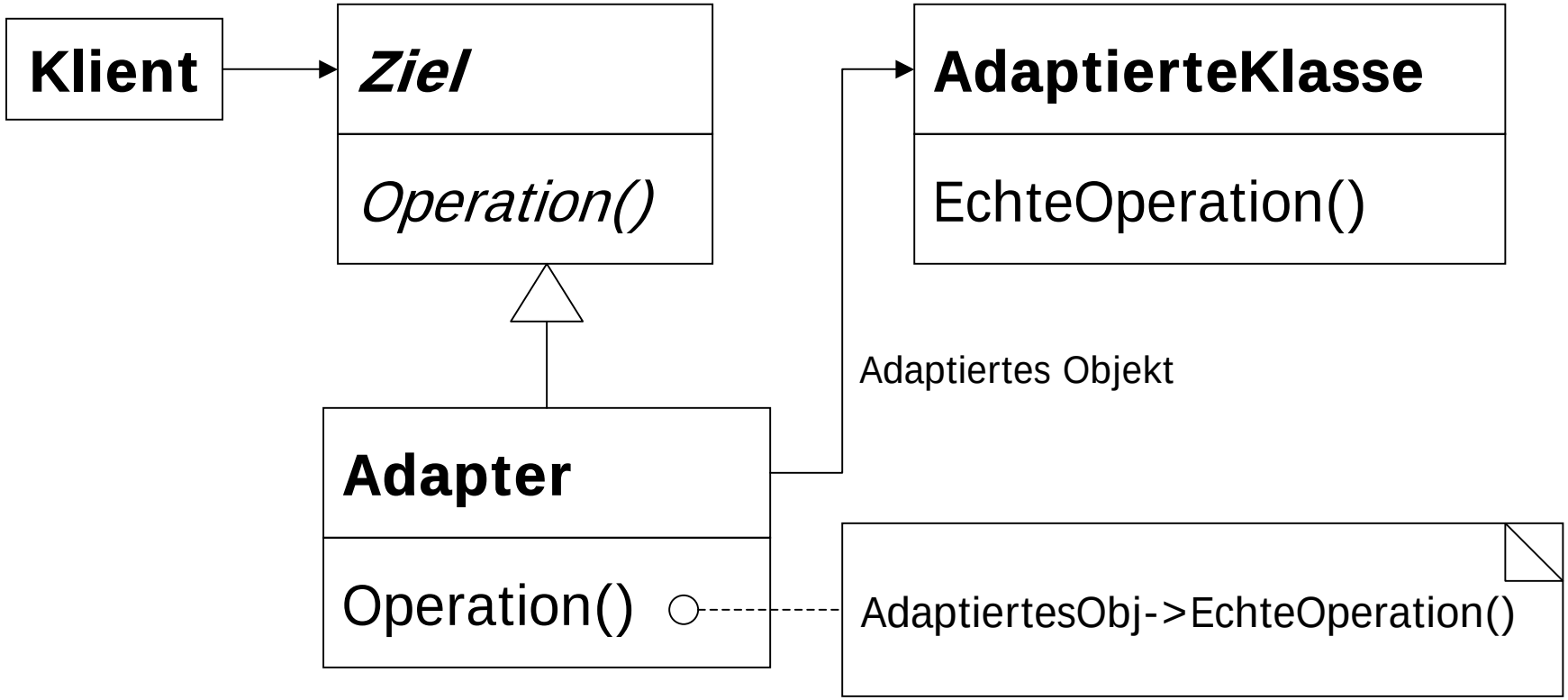


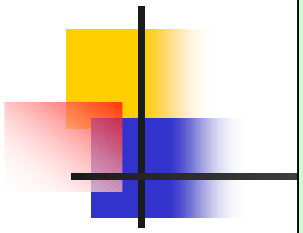
Implementation in C++
(Mehrfachvererbung)

```
class Rectangle : public ScreenObject, private Rechteck
{
public:
        ScreenObject()          { };
        ~ScreenObject()         { };

        void SetPos(int x, int y)
        {
                PosX(X);
                PoxY(Y);
        };

        void SetSize(int w, int h)
        {
                Groesse(w, h);
        };

        void Draw()
        {
                Zeichnen();
        };
};
```

Struktur

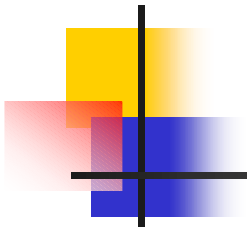
Implementation mittels Aggregation





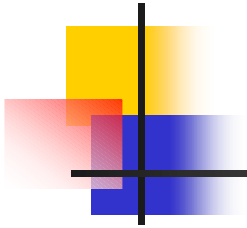
Implementation in Java (Aggregation)

```
public class Rectangle extends ScreenObject
{
        private Rechteck m_Rechteck;

        public Rectangle() {
                m_Rechteck = new Rechteck();
        };

        public void SetPos(int x, int y) {
                m_Rechteck.PosX(X);
                m_Rechteck.PoxY(Y);
        };

        public void SetSize(int w, int h) {
                 m_Rechteck.Groesse(w, h);
        };

        public void Draw() {
                 m_Rechteck.Zeichnen();
        };

};
```



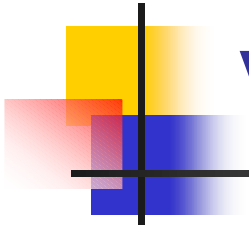
Konsequenzen

- Klassenadapter (Vererbung)
  - (+) nur eine Klasse, keine zusätzliche Indirektion
  - (-)Funktioniert nur mit einer Klasse, alle ihre Unterklassen müssen separat adaptiert werden!
  - (-) Adapter kann leicht Verhalten der adaptierten Klasse überschreiben



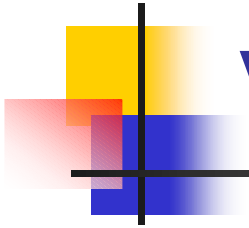
Konsequenzen

- Objektadapter (Aggregation)
  - (+) kann eine Klasse und alle Unterklassen gleichzeitig adaptieren
  - (+) Überschreiben von Funktionen der adaptierten Klasse nicht ohne weiteres möglich
  - (-) Zusätzliche Indirektion



Verwandte Pattern

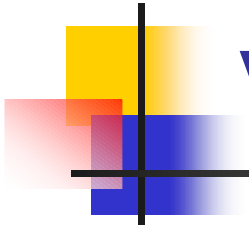
- Brückenmuster (Bridge)
- Dekorierermuster (Decorator)
- Stellvertretermuster (Proxy)



Verwandte Pattern

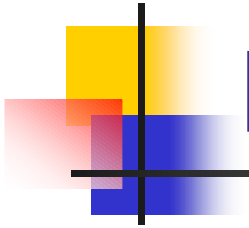
Pattern im Vergleich zur gekapselten Klasse

	Interface	Funktion
Adapter	verschieden	gleich
Proxy	gleich	gleich
Decorator	gleich	verschieden



Verwandte Pattern

- Das Bridge-Pattern ist dem Adapter sehr ähnlich, hat aber eine andere Aufgabe, nämlich Trennung von Interface und Implementation.



Ende

Vielen Dank für‘s Zuhören!

Fragen?