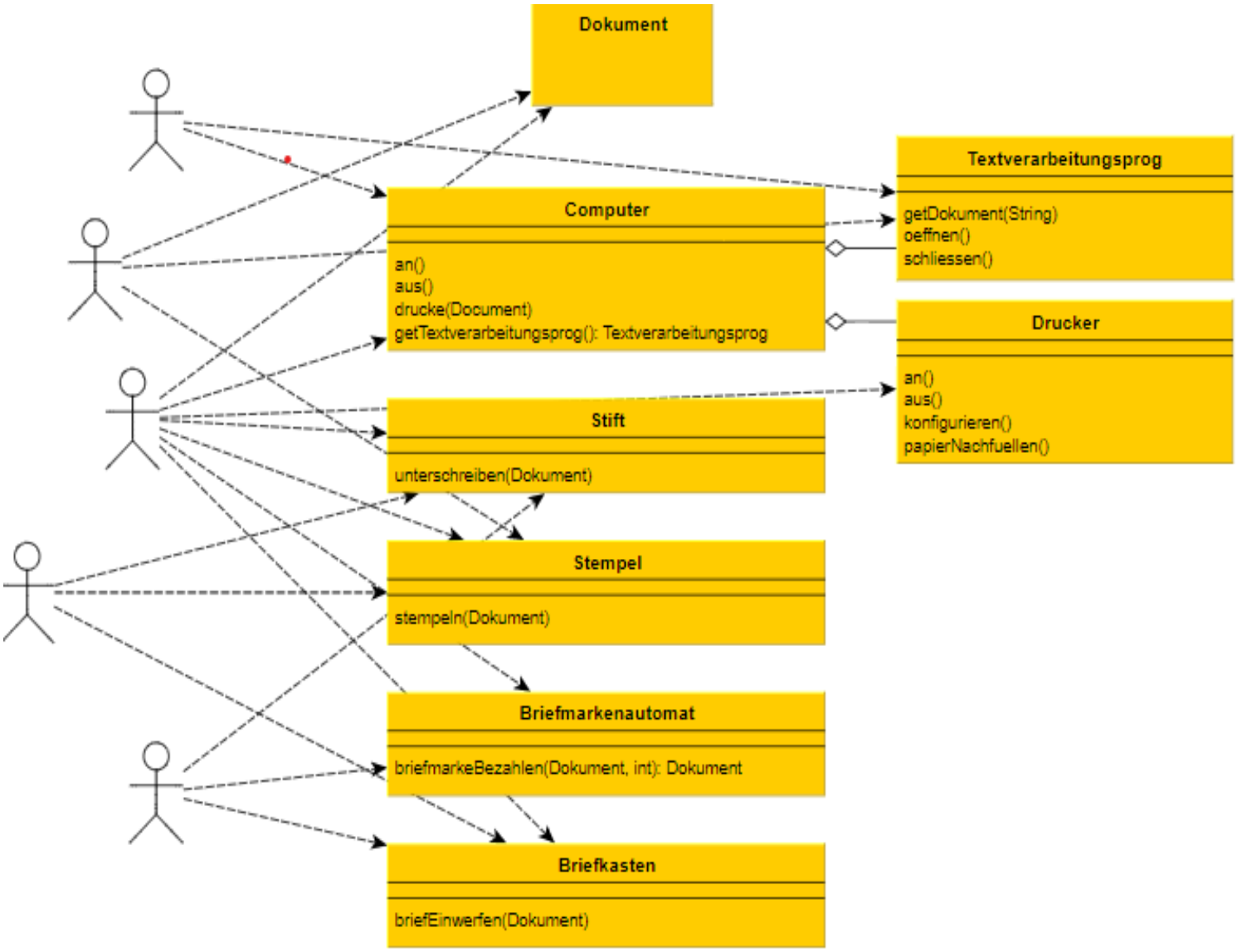
Das Adapter-Facade Design Pattern

Gegeben sei die Modellierung einer Firmenverwaltung.

Die Verwaltung nutzt dabei eine Reihe von Klassen: Dokumente, Computer, Textverarbeitungsprogramme, Drucker, Stifte, Stempel, Briefkästen und Briefmarkenautomaten.

Das Klassensystem wird von verschiedenen Benutzern (Clients) verwendet: Einer nutzt lediglich den Computer, um Texte zu schreiben, andere benötigen nur den Stift, andere wollen sich Stempel herstellen lassen und wieder andere sind nur mit dem Frankieren und Versenden von Briefen beschäftigt.



Die meisten Benutzer des Systems wollen jedoch ein Schreiben aufsetzen, drucken und gleich verschicken. Dazu sind alle Systemklassen und eine Reihe von immer gleichen Befehlen notwendig:

1. Computer muss angeschaltet werden.
2. Zugriff auf das Textverarbeitungsprogramm über den Computer.
3. Das Textverarbeitungsprogramm muss geöffnet werden.
4. Mit dem Textverarbeitungsprogramm muss das Dokument erstellt werden
5. Drucker muss angeschaltet werden
6. Drucker muss konfiguriert werden
7. Papier muss in den Drucker gelegt werden
8. Das Dokument muss gedruckt werden
9. Der Drucker sollte ausgeschaltet werden
10. Der Computer sollte ausgeschaltet werden
11. Das Dokument muss mit dem Stift unterschrieben werden
12. Das Dokument muss mit dem Stempel gestempelt werden
13. Das Dokument muss mit Hilfe des Briefmarkenautomaten frankiert werden
14. Das Dokument muss schließlich in den Briefkasten geworfen werden.

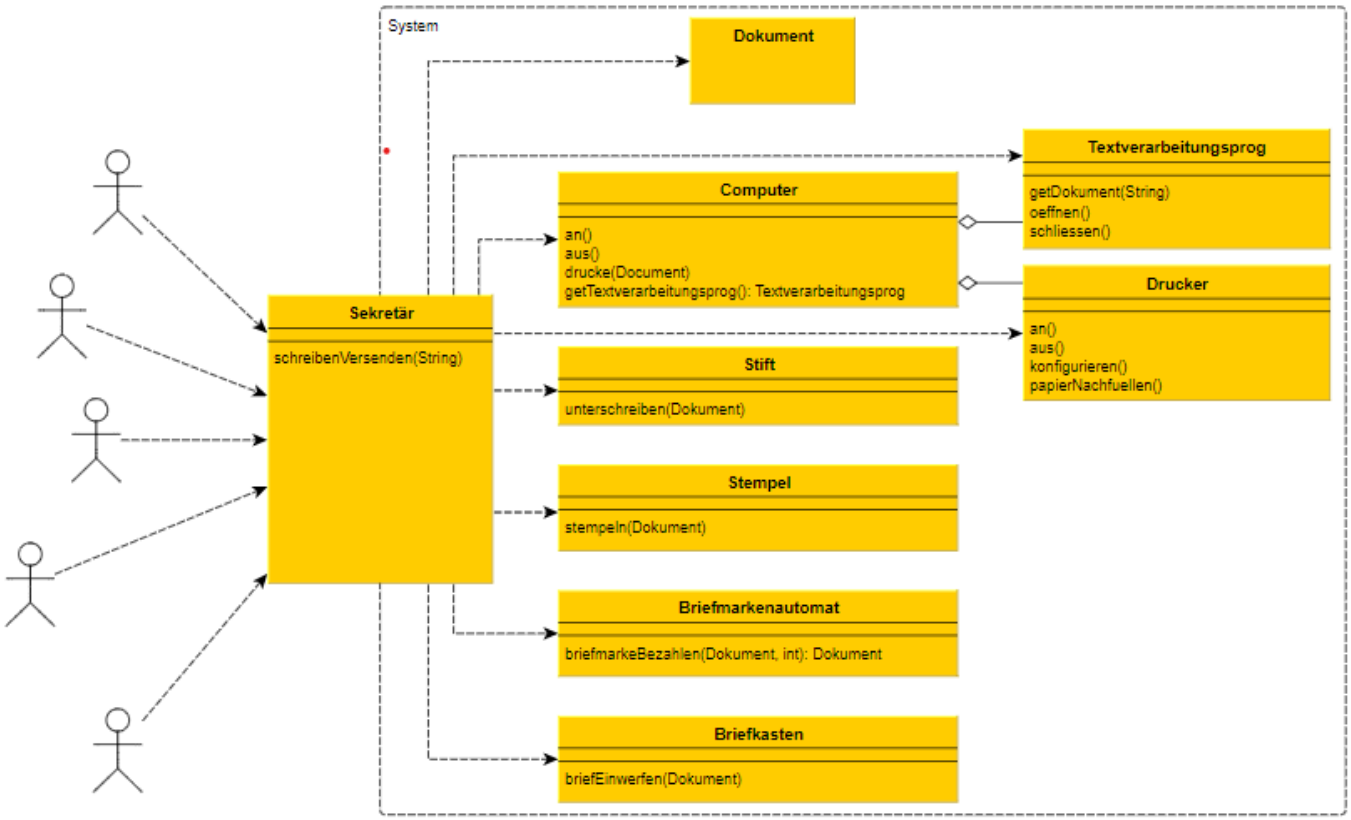
```
//Gegeben: Alle benötigten Klassen sind korrekt instanziiert,
initialisiert und bekannt
   String text = "Dieser Text soll verschickt werden";

   //Computer muss angeschaltet werden.
   computer.an();
   //Zugriff auf das Textverarbeitungsprogramm über den
   Computer.
   Textverarbeitungsprog textverarbeitungsprog =
   computer.getTextverarbeitungsprog();
   //Das Textverarbeitungsprogramm muss geöffnet werden.
   textverarbeitungsprog.oeffnen();
   //Mit dem Textverarbeitungsprogramm muss das Dokument
   erstellt werden
   Dokument dokument = textverarbeitungsprog.getDokument(text);
   //Drucker muss angeschaltet werden
   drucker.an();
   //Drucker muss konfiguriert werden
   drucker.konfigurieren();
   //Papier muss in den Drucker gelegt werden
   drucker.papierNachfuellen();
   //Das Dokument muss gedruckt werden
   computer.drucke(dokument);
   //Der Drucker sollte ausgeschaltet werden
   drucker.aus();
   //Der Computer sollte ausgeschaltet werden
   computer.aus();
   //Das Dokument muss mit dem Stift unterschrieben werden
   stift.unterschreiben(dokument);
   //Das Dokument muss mit dem Stempel gestempelt werden
   stempel.stempel(dokument);
   //Das Dokument muss mit Hilfe des Briefmarkenautomaten
   frankiert werden
   briefmarkenautomat.briefmarkeBezahlen(dokument, 2);
   //Das Dokument muss schließlich in den Briefkasten geworfen
   werden.
   briefkasten.briefEinwerfen(dokument);
```

Schauen wir uns diese Vorgehensweise genau an:

- **Systemwissennotwendig.** Jeder Client muss nicht nur jede benötigte Klasse des Systems kennen, sondern auch ihr Zusammenspiel und ihre Funktionsweise, um sie nutzen zu können.

- **Abhängigkeiten und geringe Änderungsstabilität**. Da jeder Client viele verschiedene Klassen kennen muss, steigen seine Abhängigkeiten. Er ist hart an das System gekoppelt.. Die Folge ist hoher Wartungsaufwand.
- **Coderedundanz** (überlaufen)und **Gefahr von Inkonsistenz** (nicht miteinander vereinbar). Alle Clients, die ein Schreiben aufsetzen, drucken und verschicken wollen, müssen immer den gleichen Code schreiben. Dabei kann es sehr schnell passieren, dass Schritte ausgelassen werden oder die APIs falsch verwendet wird.

Wie wäre es, wenn wir **eine Instanz zwischen System und Benutzern des Systems schalten**? Diese Instanz stellt eine einheitliche und vereinfachte Schnittstelle (API) zum Schreiben, Drucken und Verschicken von Dokumenten bereit. Also, führen wir diese Instanz ein: der Sekretär - unsere *Fassade*!



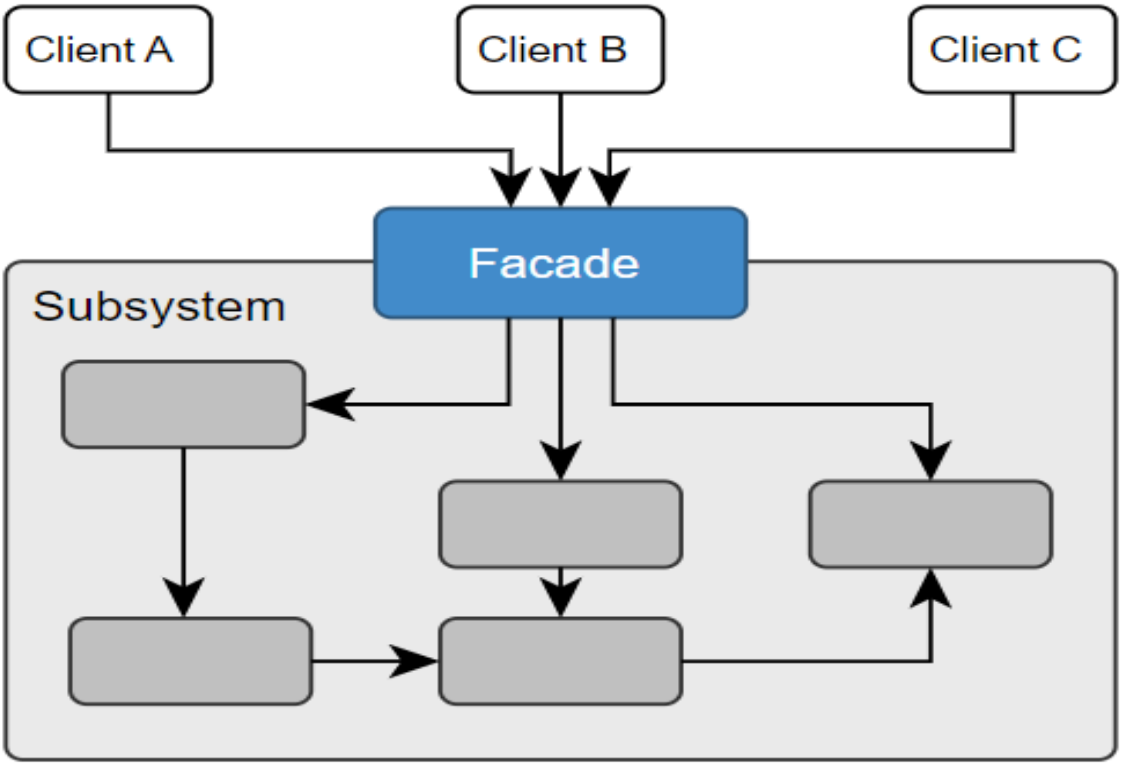
Der Sekretär stellt eine vereinfachte Schnittstelle zur Nutzung des Systems bereit !!!!!!

Der Client kann die Funktionalität des Systems fortan bequem über den Sekretär nutzen !!!!!!!!

Vorteile:

- Der Sekretär **vereinfacht die Benutzung** unseres Klassensystems durch den Client.
- Clients, die nur ausgewählte Klassen unseres Systems nutzen wollen oder **anspruchsvollere Aufgaben mit dem System erledigen** möchten, für die die Schnittstelle des Sekretärs nicht ausreicht, können **die Klassen weiterhin direkt nutzen**.

- **Wartbarkeit** und **Änderungsstabilität** durch **verringerte Abhängigkeiten** und **lose Kopplung zwischen Client und System**.
- Betrachten wir den Sekretär als **Einstiegspunkt zu unserem System.**
- **Keine Coderedundanz** und Gefahr von **Inkonsistenz** beim Client.



Das Adapter-Facade Design Pattern definiert eine vereinfache Schnittstelle zur Benutzung eines Systems oder einer Menge von Objekten.

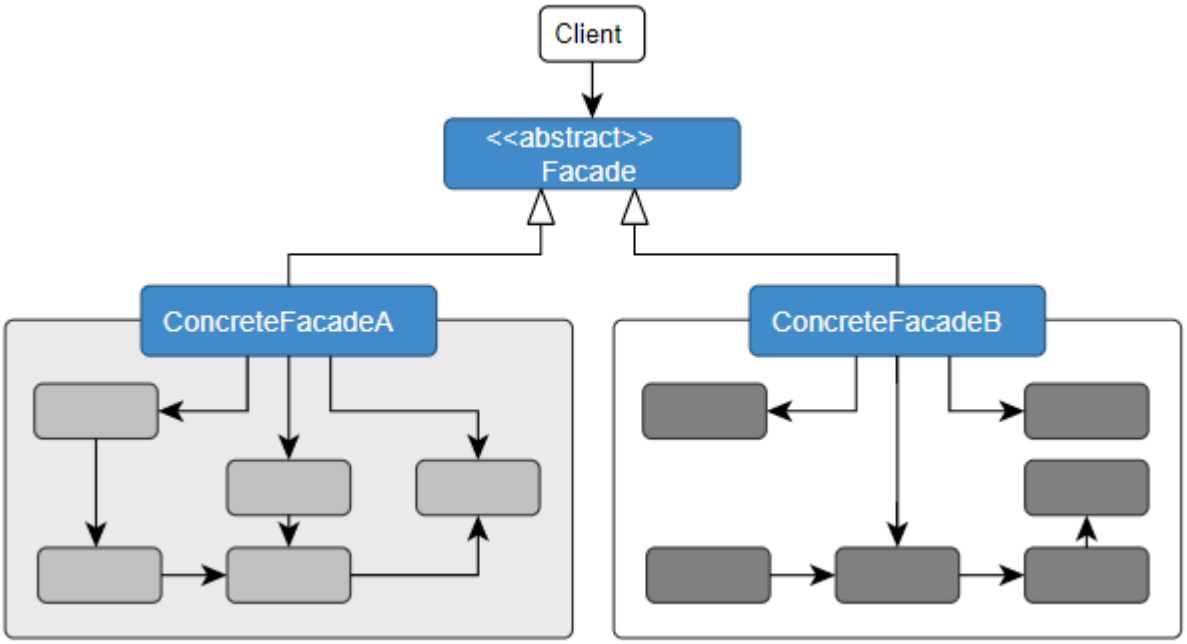
Die Fassade ist eine Klasse mit ausgewählten Methoden, die eine häufig benötigte Untermenge an Funktionalität des Subsystems umfasst.

Variationen

Schichten durch Fassaden

Betrachtet man das Facade Design Pattern als Mittel um ganze Systeme zu kapseln, so ist es auch möglich, diese Systeme selbst wiederum aus Teilsystemen aufzubauen, die über Fassaden angesprochen werden können.

Minimierung der Kopplung zwischen Client und Subsystem



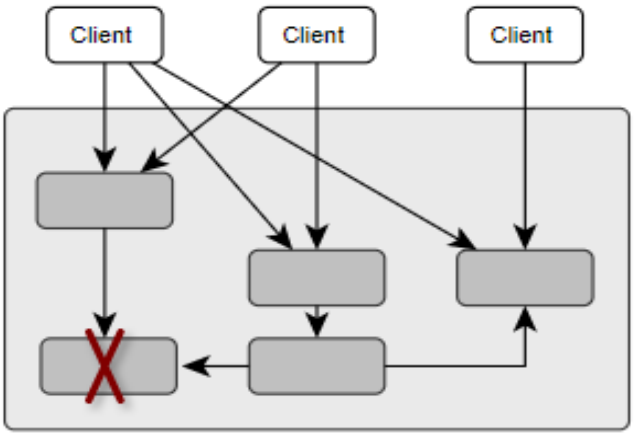
Anwendungsfälle

Das **Adapter-Facade Design Pattern** kann angewandt werden, wenn...

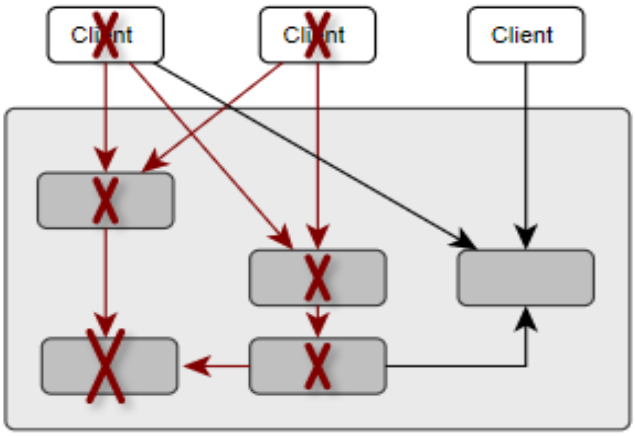
- ... eine vereinfachte **Schnittstelle** zur Nutzung eines komplizierten Subsystems oder Menge von Objekten benötigt wird.
- ... die **Reduzierung der Abhängigkeiten** zwischen dem Client und dem benutzten Subsystem angestrebt wird.
- ... ein System in **Schichten unterteilt werden** soll

Vorteile

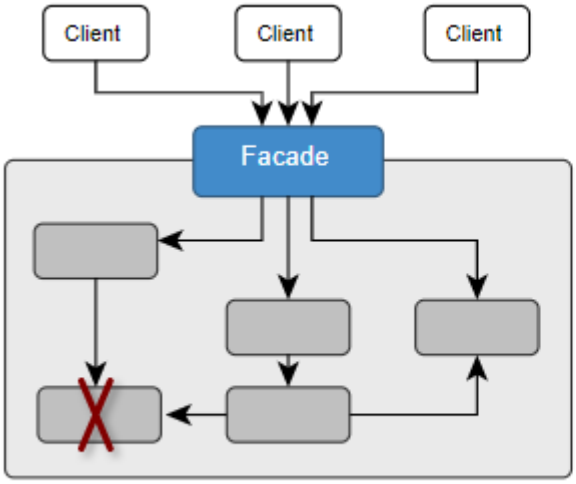
- **Vereinfachte Schnittstelle**. Der Client kann ein komplexes System einfacher verwenden, ohne die Klassen des Systems zu kennen
- **Entkopplung** des Client vom Subsystem.
- **Wartungen** und **Modifikationen** am Subsystem innerhalb des Systems. Die Schnittstelle der Fassade nach außen bleibt davon unbetroffen. Kein Code der Clients bricht !!!!!!



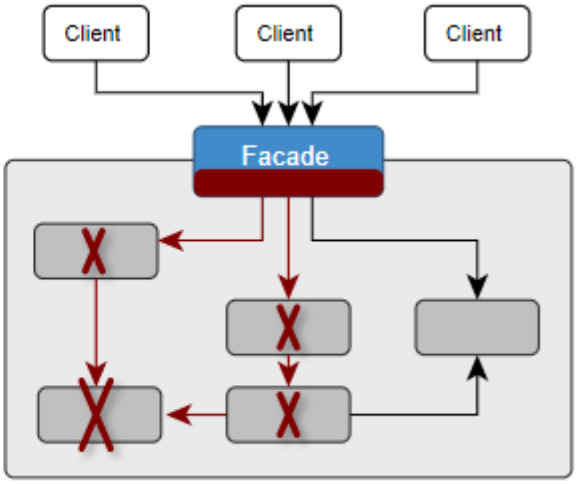
Änderung einer Componente



Clients betroffen!



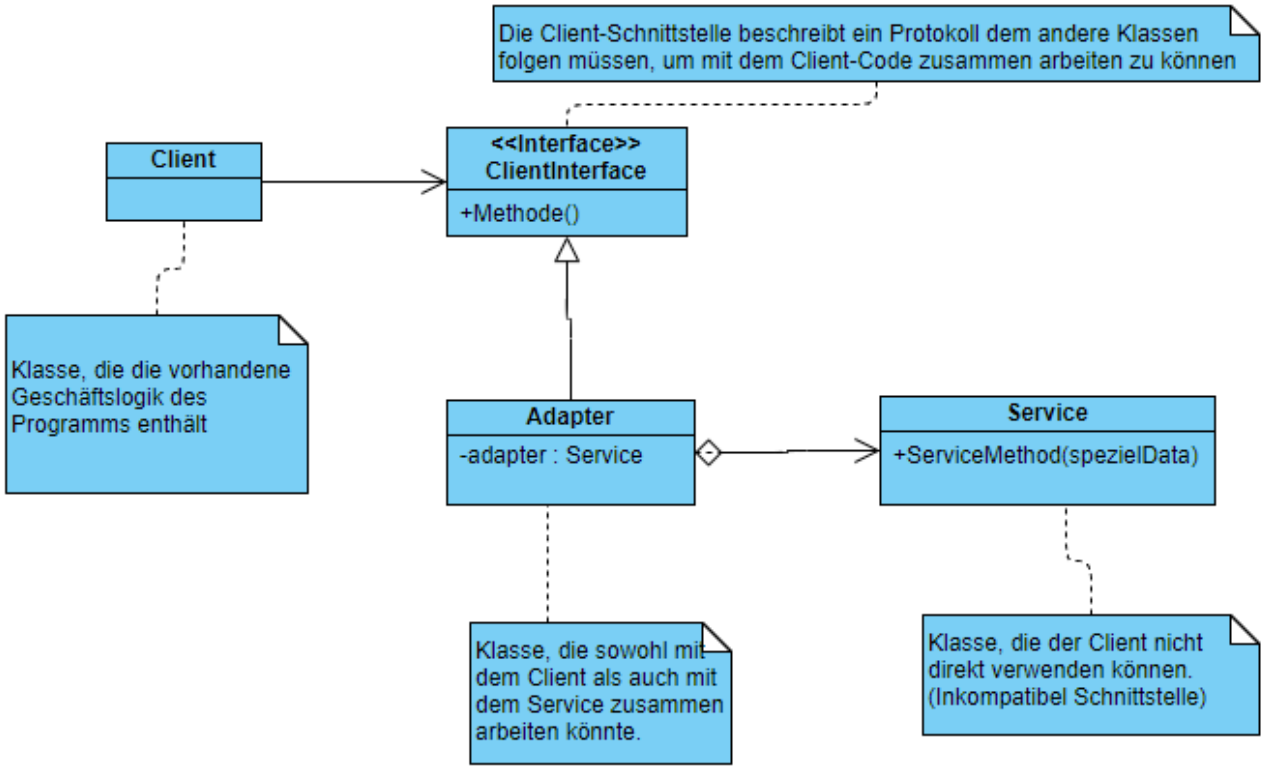
Änderung einer Componente



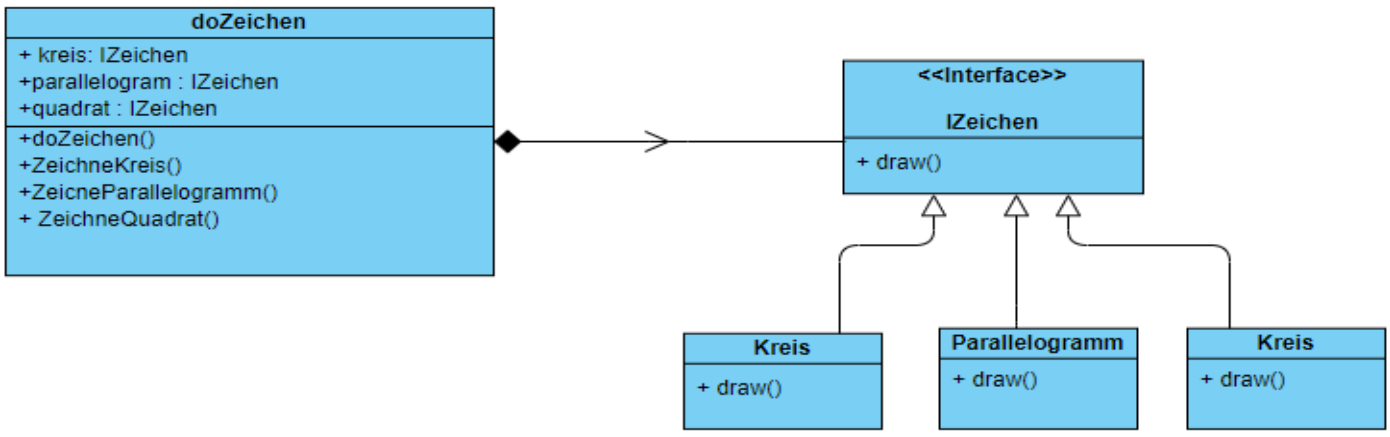
Clients nicht betroffen

- Eine Fassade verringerte die Abhängigkeiten des Clients, da die Anzahl der Objekte, die vom Client gehändelt werden müssen, signifikant gesenkt werden kann.
- Jedoch können wenn gewünscht anspruchsvolle Clients weiterhin die Fassade übergehen und die Objekte des Subsystems direkt verwenden, sollte die bereitgestellte Schnittstelle der Fassade einmal nicht ausreichen.

Adapter-Facade-Muster



Adapter-Facade Muster (Beispiel)




```
// Interface - IZeichen

public interface IZeichen
  {
    void draw();
  }
```

```
// Klasse - Kreis

public class Kreis :IZeichen
  {
    public void draw()
    {
      Console.WriteLine("Circle::draw()");
    }
  }
```

```
// Klasse - Parallelogramm

public class Parallelogramm : IZeichen
  {

   public void draw()
    {
      Console.WriteLine("Rectangle::draw()");
    }
  }
```

```
// Klasse - Quadrat

public class Quadrat :IZeichen
  {
    public void draw()
    {
      Console.WriteLine("Square::draw()");
    }
  }
```

```
// Klasse - DoZeichen
public class DoZeichen
  {
      public IZeichen kreis;
      public IZeichen parallelogramm;
      public IZeichen quadrat;

      public DoZeichen()
      {
        kreis = new Kreis();
        parallelogram = new Parallelogramm();
        quadrat = new Quadrat();
      }

      public void ZeichneKreis()
      {
```

```
            kreis.draw();
        }
        public void ZeichneParallelogramm()
        {
            parallelogram.draw();
        }
        public void ZeichneQuadrat()
        {
            quadrat.draw();
        }
    }
```

Main

```
public static void Main(string[] args)
    {
        // Obj der Klasse DoZeichen erzeugen

        DoZeichen my_Zeichen = new DoZeichen();

       // die Methoden ZeichneKreis(), ZeichneParallelogramm() und ZeichneQuadrat()
        my_Zeichen.ZeichneKreis();
        my_Zeichen.ZeichneParallelogramm();
        my_Zeichen.ZeichneQuadrat();
        Console.WriteLine("Hallo World!");
    }
```