

GoF-Entwurfsmuster

Design Patterns nach der „Gang of Four“

Handout zum Vortrag – Erzeugungs-, Struktur- und Verhaltensmuster

IT-Umschulung Fachinformatiker/in Anwendungsentwicklung und Systemintegration

So arbeiten Sie mit diesem Handout

Kapitel 1 bis 4 erklären, was Entwurfsmuster sind, woher sie kommen und nach welchen Prinzipien sie funktionieren. Kapitel 5 bis 7 stellen je Kategorie drei Muster ausführlich vor – mit Problem, Lösung, UML-Klassendiagramm, Code und Konsequenzen – und die übrigen GoF-Muster in Kurzform.

Alle Beispiele stammen aus der Demo-Solution [GoF-Entwurfsmuster_Demo.sln](#) (Nordfracht Logistik GmbH, .NET Framework 4.8).

Aussagen mit **(ungeprüft)** sollten Sie vor einer Weiterverwendung gegen eine aktuelle Quelle prüfen.

Inhalt

- 1 Was ist ein Entwurfsmuster?
- 2 Geschichte
- 3 Einordnung und Kategorien
- 4 Prinzipien hinter den Mustern
- 5 Erzeugungsmuster
- 6 Strukturmuster
- 7 Verhaltensmuster
- 8 Muster im .NET-Alltag
- 9 Kritik und Antimuster
- 10 Entwurfsmuster in der Abschlussprüfung
- 11 Ausblick
- 12 Die Demo-Solution
- 13 Glossar
- 14 Wiederholungsfragen
- 15 Weiterführende Quellen

1 Was ist ein Entwurfsmuster?

Ein **Entwurfsmuster** (englisch „design pattern“) ist eine bewährte, wiederverwendbare Lösungsschablone für ein wiederkehrendes Entwurfsproblem in der objektorientierten Softwareentwicklung. Es ist **kein fertiger Code** und keine Bibliothek, die man einbindet, sondern eine Beschreibung: Welche Klassen und Schnittstellen gibt es, welche Aufgaben haben sie, und wie arbeiten sie zusammen?

Ein Vergleich hilft: Ein Kochrezept für „Soße binden“ funktioniert bei Bratensoße genauso wie bei Currysauce. Das Rezept beschreibt das Vorgehen, die konkreten Zutaten wählt man je nach Gericht. Genauso beschreibt das Muster „Beobachter“ das Vorgehen, ob nun Sendungsstatus, Börsenkurse oder Sensorwerte beobachtet werden.

1.1 Die vier Elemente eines Musters

Die „Gang of Four“ nennt vier wesentliche Bestandteile jeder Musterbeschreibung:

Element	Bedeutung
Name	Ein kurzer, einprägsamer Begriff. Er erweitert das Fachvokabular: „Da nehmen wir einen Decorator“ ersetzt eine halbe Seite Erklärung.
Problem	Wann wird das Muster angewendet? Welche Situation, welcher Zielkonflikt liegt vor?
Lösung	Die beteiligten Elemente, ihre Beziehungen und Zuständigkeiten – abstrakt, nicht als fertige Implementierung.
Konsequenzen	Vor- und Nachteile, Kosten und Nutzen. Jedes Muster hat einen Preis, meist zusätzliche Klassen und Indirektion.

Im GoF-Buch wird jedes Muster zusätzlich nach einem festen Schema beschrieben, unter anderem mit Zweck, Motivation, Anwendbarkeit, Struktur (Klassendiagramm), Teilnehmern, Interaktionen, Implementierungshinweisen, bekannten Verwendungen und verwandten Mustern.

1.2 Warum Muster lernen?

- **Gemeinsame Sprache:** Entwicklerinnen und Entwickler verständigen sich schneller und präziser über Entwürfe.
- **Erprobte Lösungen:** Man muss das Rad nicht neu erfinden – und vermeidet Fehler, die andere schon gemacht haben.
- **Code lesen können:** Frameworks wie .NET sind voller Muster. Wer sie erkennt, versteht fremden Code schneller.
- **Prüfungsrelevanz:** In der Abschlussprüfung Teil 2 werden Muster in UML-Diagrammen und Code abgefragt (siehe Kapitel 10).

2 Geschichte

2.1 Christopher Alexander – Muster in der Architektur (1977)

Der Begriff stammt nicht aus der Informatik, sondern aus der Gebäudearchitektur. Der Architekt **Christopher Alexander** veröffentlichte 1977 mit Kollegen das Buch „**A Pattern Language**“ mit 253 Mustern für Städte, Gebäude und Räume – etwa „Fensterplatz“ oder „Licht von zwei Seiten in jedem Raum“. Seine Definition prägt bis heute: Ein Muster beschreibt ein Problem, das immer wieder auftritt, und den Kern einer Lösung, die man beliebig oft verwenden kann, ohne sie je zweimal gleich umzusetzen.

2.2 Der Sprung in die Software (1987)

1987 wurde die Idee auf einer Fachkonferenz erstmals auf die Software übertragen – zunächst mit einer kleinen Mustersammlung für Benutzeroberflächen in Smalltalk.

2.3 Die „Gang of Four“ (1994)

Der Durchbruch kam 1994 mit dem Buch „**Design Patterns: Elements of Reusable Object-Oriented Software**“ von **Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides**. Die vier Autoren werden als „**Gang of Four**“ (GoF) bezeichnet. Das Buch katalogisiert **23 Muster** mit Beispielen in C++ und Smalltalk und ordnet sie in drei Kategorien ein. Die deutsche Ausgabe heißt „Entwurfsmuster“.

2.4 Nach der GoF

Auf die GoF folgten weitere Musterkataloge für andere Ebenen und Einsatzgebiete. Die wichtigsten:

Jahr	Thema	Inhalt
1996	Architekturmuster	Muster für die Grundstruktur ganzer Systeme (Schichten, Pipes & Filters, MVC) und die Einteilung in Architekturmuster, Entwurfsmuster und Idiome
1996	Kritik	In dynamischen Programmiersprachen sind viele GoF-Muster unsichtbar oder deutlich einfacher
1998	Antimuster	Katalog typischer Fehlentwürfe (Gottobjekt, Spaghetti-Code, Goldener Hammer)
2002	Unternehmensanwendungen	Repository, Unit of Work, Active Record, Data Mapper
2004	„Head First Design Patterns“	Didaktisch aufbereitete Einführung, sehr verbreitet in der Ausbildung

3 Einordnung und Kategorien

3.1 Drei Abstraktionsebenen

Muster gibt es auf unterschiedlichen Ebenen. Üblich ist die Unterscheidung in drei:

Ebene	Reichweite	Beispiele
Architekturmuster	Grundstruktur eines ganzen Systems	Schichtenarchitektur, Client-Server, MVC, MVVM, Microservices, Pipes & Filters
Entwurfsmuster	Zusammenspiel weniger Klassen innerhalb eines Teilsystems	die 23 GoF-Muster, z. B. Observer, Strategy, Adapter
Idiome	Lösung in einer bestimmten Programmiersprache	<code>using</code> -Block für <code>IDisposable</code> in C#, <code>yield return</code> , RAII in C++

Die Grenzen sind fließend. MVVM etwa ist ein Architekturmuster, das intern mehrere Entwurfsmuster nutzt: Observer (`INotifyPropertyChanged`) und Command (`ICommand`).

3.2 Die drei GoF-Kategorien nach Zweck

Kategorie	Leitfrage	Muster
Erzeugungsmuster (creational)	Wie entstehen Objekte – ohne dass der Aufrufer die konkrete Klasse kennen muss?	Abstrakte Fabrik (Abstract Factory), Erbauer (Builder), Fabrikmethode (Factory Method), Prototyp (Prototype), Singleton
Strukturmuster (structural)	Wie werden Klassen und Objekte zu größeren Strukturen zusammengesetzt?	Adapter, Brücke (Bridge), Kompositum (Composite), Dekorierer (Decorator), Fassade (Facade), Fliegengewicht (Flyweight), Stellvertreter (Proxy)
Verhaltensmuster (behavioral)	Wie arbeiten Objekte zusammen und wie werden Zuständigkeiten verteilt?	Zuständigkeitskette (Chain of Responsibility), Befehl (Command), Interpreter, Iterator, Vermittler (Mediator), Memento, Beobachter (Observer), Zustand (State), Strategie (Strategy), Schablonenmethode (Template Method), Besucher (Visitor)

3.3 Zweite Achse: Klassen- oder Objektmuster

Die GoF unterscheidet außerdem nach dem **Bereich**: **Klassenumuster** beschreiben Beziehungen zwischen Klassen, die über **Vererbung** zur Compilezeit festgelegt werden. **Objektmuster** beschreiben Beziehungen zwischen Objekten, die über **Komposition** zur Laufzeit veränderbar sind. Die meisten Muster sind Objektmuster.

	Erzeugung	Struktur	Verhalten
Klasse	Fabrikmethode (Factory Method)	Adapter (Klassen-Variante)	Interpreter, Schablonenmethode (Template Method)
Objekt	Abstract Factory, Builder, Prototyp, Singleton	Adapter (Objekt-Variante), Bridge, Composite, Decorator, Facade, Flyweight, Proxy	Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor

3.4 Alle 23 GoF-Muster: deutsche und englische Namen

In Fachliteratur, Stellenanzeigen, Frameworks und auch in Prüfungsaufgaben werden meist die **englischen** Namen verwendet. Lernen Sie beide.

Kategorie	Deutscher Name	Englischer Name	Kurzzweck
Erzeugung	Abstrakte Fabrik	Abstract Factory	Familien verwandter Objekte erzeugen
Erzeugung	Erbauer	Builder	Komplexes Objekt schrittweise aufbauen
Erzeugung	Fabrikmethode	Factory Method	Unterklasse entscheidet, was erzeugt wird
Erzeugung	Prototyp	Prototype	Objekte durch Klonen erzeugen
Erzeugung	Singleton (selten: Einzelstück)	Singleton	Genau eine Instanz
Struktur	Adapter	Adapter (Wrapper)	Schnittstelle anpassen
Struktur	Brücke	Bridge	Abstraktion und Implementierung trennen
Struktur	Kompositum	Composite	Baumstrukturen einheitlich behandeln
Struktur	Dekorierer	Decorator	Verhalten dynamisch ergänzen
Struktur	Fassade	Facade	Einfache Schnittstelle zu einem Subsystem
Struktur	Fliegengewicht	Flyweight	Gemeinsamen Zustand teilen, Speicher sparen
Struktur	Stellvertreter	Proxy	Zugriff auf ein Objekt kontrollieren
Verhalten	Zuständigkeitskette	Chain of Responsibility	Anfrage entlang einer Kette weiterreichen
Verhalten	Befehl	Command	Aktion als Objekt kapseln
Verhalten	Interpreter	Interpreter	Einfache Sprache auswerten
Verhalten	Iterator	Iterator	Sammlung durchlaufen
Verhalten	Vermittler	Mediator	Kommunikation zentral steuern
Verhalten	Memento	Memento	Zustand sichern und wiederherstellen
Verhalten	Beobachter	Observer (Publish-Subscribe)	Abhängige bei Änderung benachrichtigen
Verhalten	Zustand	State	Verhalten hängt vom inneren Zustand ab
Verhalten	Strategie	Strategy	Algorithmen austauschbar machen
Verhalten	Schablonenmethode	Template Method	Ablauf fest, Einzelschritte variabel
Verhalten	Besucher	Visitor	Neue Operationen ohne Klassenänderung

4 Prinzipien hinter den Mustern

Muster sind keine Sammlung zufälliger Tricks. Fast alle beruhen auf wenigen Grundprinzipien. Wer diese versteht, kann Muster ableiten, statt sie auswendig zu lernen.

4.1 Gegen Schnittstellen programmieren

Die GoF formuliert: „**Programmiere gegen eine Schnittstelle, nicht gegen eine Implementierung.**“ Der **Preisrechner** der Demo kennt nur **IPreisStrategie**, nicht **SperrgutTarif**. Dadurch kann jeder neue Tarif eingesetzt werden, ohne den Rechner zu ändern.

4.2 Komposition vor Vererbung

Das zweite GoF-Prinzip lautet: „**Ziehe Objektkomposition der Klassenvererbung vor.**“ Vererbung legt Verhalten zur Compilezeit fest und führt schnell zu starren, tiefen Hierarchien. Komposition – ein Objekt **hat** ein anderes – ist zur Laufzeit veränderbar. Der Dekorierer ist das Paradebeispiel: Statt acht Unterklassen für alle Kombinationen aus Express, Versicherung und Nachnahme gibt es drei kleine Hüllen, die beliebig kombiniert werden.

4.3 Kapsle, was variiert

Man identifiziert die Teile eines Programms, die sich ändern werden, und trennt sie von denen, die stabil bleiben. Bei der Fabrikmethode variiert der erzeugte Typ, beim Strategiemuster der Algorithmus, beim Beobachter die Menge der Interessenten.

4.4 SOLID

Die fünf **SOLID-Prinzipien** passen eng zu den Mustern:

Prinzip	Bedeutung	Muster-Bezug
Single Responsibility	Eine Klasse hat genau einen Grund, sich zu ändern.	Fassade, Befehl
Open/Closed	Offen für Erweiterung, geschlossen für Änderung.	Strategie, Beobachter, Dekorierer
Liskov Substitution	Unterklassen müssen überall anstelle der Oberklasse funktionieren.	Fabrikmethode, alle Interface-basierten Muster
Interface Segregation	Lieber mehrere kleine Schnittstellen als eine große.	Strategie (eine Methode)
Dependency Inversion	Abhängigkeit von Abstraktionen, nicht von Details.	Adapter, Abstrakte Fabrik, Dependency Injection

5 Erzeugungsmuster

Erzeugungsmuster kapseln das Erzeugen von Objekten. Der Aufrufer soll nicht wissen müssen, **welche** konkrete Klasse entsteht, **wie** sie aufgebaut wird oder **wie viele** es davon gibt. Jedes `new KonkreteKlasse()` bindet den Code fest an diese Klasse – Erzeugungsmuster lösen diese Bindung.

5.1 Singleton

Kategorie: Erzeugung, Objekt · **Englisch:** Singleton · **Zweck in einem Satz:** Sicherstellen, dass es von einer Klasse genau eine Instanz gibt, und einen globalen Zugriffspunkt bieten.

Problem

Die Nordfracht-Anwendung liest ihre Konfiguration (Firmenname, maximales Paketgewicht) einmal beim Start. Würde jede Klasse ein eigenes Konfigurationsobjekt erzeugen, würde die Datei mehrfach gelesen, und es könnten widersprüchliche Stände entstehen.

Lösung

Der Konstruktor wird `private` gemacht, sodass niemand außerhalb der Klasse ein Objekt erzeugen kann. Die Klasse selbst hält die einzige Instanz in einem statischen Feld und gibt sie über eine statische Eigenschaft heraus. In C# erledigt `Lazy<T>` die threadsichere, verzögerte Erzeugung.

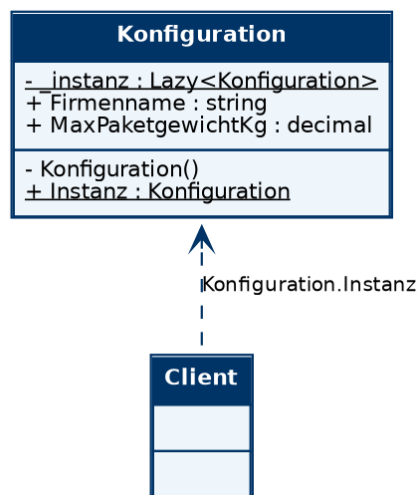


Abb.: Singleton – Klassendiagramm der Demo

Rolle	In der Demo
Singleton	<code>Konfiguration</code>
Client	jede Klasse, die <code>Konfiguration.Instanz</code> aufruft – in der Demo <code>Program</code> und der <code>SendungBuilder</code> , der dort das Höchstgewicht liest

Code-Ausschnitt

```
public sealed class Konfiguration
{
    private static readonly Lazy<Konfiguration> _instanz =
        new Lazy<Konfiguration>(() => new Konfiguration());

    public static Konfiguration Instanz { get { return _instanz.Value; } }

    private Konfiguration() { /* Datei lesen */ }
}
```

Konsequenzen

Vorteile	Nachteile
Garantiert genau eine Instanz Verzögerte Erzeugung (erst bei Bedarf) Einfach umzusetzen	Globaler Zustand – versteckte Abhängigkeiten Schwer testbar (nicht austauschbar) Bei naiver Umsetzung nicht threadsicher Verstößt gegen Single Responsibility (verwaltet sich selbst)

Im .NET-Alltag: In modernen Anwendungen wird statt eines klassischen Singletons meist ein Dienst per Dependency Injection mit der Lebensdauer „Singleton“ registriert (`services.AddSingleton<IKonfiguration, Konfiguration>()`). Die Einzigartigkeit regelt dann der Container, und im Test lässt sich eine Attrappe einsetzen.

Nicht verwechseln

Eine **statische Klasse** ist kein Singleton: Sie kann keine Schnittstelle implementieren, nicht als Parameter übergeben und nicht vererbt werden.

5.2 Fabrikmethode (Factory Method)

Kategorie: Erzeugung, Klasse · **Englisch:** Factory Method · **Zweck in einem Satz:** Eine Schnittstelle zur Objekterzeugung definieren, Unterklassen entscheiden, welche Klasse instanziiert wird.

Problem

Der Ablauf „Sendung versenden“ ist bei Nordfracht immer gleich: Transportmittel bereitstellen, Sendung transportieren. Welches Transportmittel das ist, hängt aber von der Abteilung ab – der Landversand nutzt LKW, der Seeverand Schiffe. Stünde `new Lkw()` direkt im Ablauf, müsste man ihn für jede neue Transportart ändern.

Lösung

Die abstrakte Oberklasse `Versandabteilung` implementiert den festen Ablauf `Versende()` und ruft darin eine abstrakte Methode `ErzeugeTransportmittel()` auf – die **Fabrikmethode**. Jede Unterklasse überschreibt sie und liefert ihr eigenes Produkt. Der Ablauf arbeitet nur mit der Schnittstelle `ITransportmittel`.

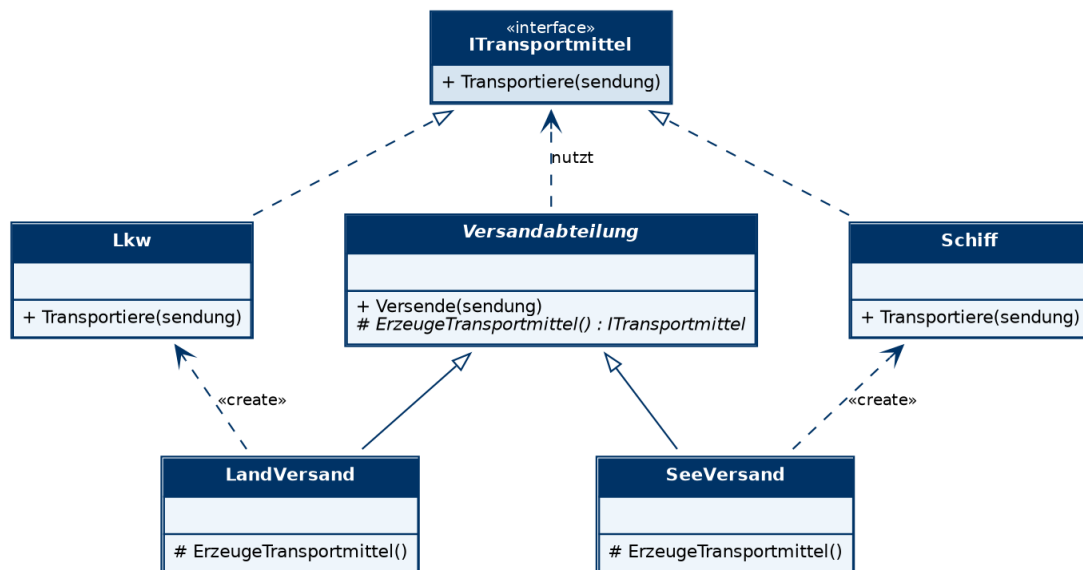


Abb.: Fabrikmethode (Factory Method) – Klassendiagramm der Demo

Rolle	In der Demo
Produkt	<code>ITransportmittel</code>
Konkretes Produkt	<code>Lkw</code> , <code>Schiff</code>
Erzeuger (Creator)	<code>Versandabteilung</code> mit abstrakter Fabrikmethode
Konkreter Erzeuger	<code>LandVersand</code> , <code>SeeVersand</code>

Code-Ausschnitt

```

public abstract class Versandabteilung
{
    protected abstract ITransportmittel ErzeugeTransportmittel(); // Fabrikmethode

    public void Versende(string sendung)
    {
        ITransportmittel transport = ErzeugeTransportmittel();
        transport.Transportiere(sendung);
    }
}

public class SeeVersand : Versandabteilung
{
    protected override ITransportmittel ErzeugeTransportmittel()
    {
        return new Schiff();
    }
}
  
```

Konsequenzen

Vorteile	Nachteile
Neue Produkte ohne Änderung am Ablauf (Open/Closed) Keine Kopplung an konkrete Klassen Erzeugungslogik an einem definierten Ort	Für jedes Produkt eine eigene Erzeuger-Unterklasse Mehr Klassen, mehr Indirektion

Im .NET-Alltag: `DbProviderFactory.CreateConnection()`, `WebRequest.Create()`; in ASP.NET Core `IHttpClientFactory.CreateClient()`.

Nicht verwechseln

Die „**einfache Fabrik**“ (eine statische Methode mit `switch`, die je nach Parameter ein Objekt liefert) ist kein GoF-Muster, wird aber oft so genannt. Die **Abstrakte Fabrik** erzeugt ganze Familien zusammengehöriger Objekte über ein Fabrikojekt, die Fabrikmethode ein einzelnes Produkt über Vererbung.

5.3 Erbauer (Builder)

Kategorie: Erzeugung, Objekt · **Englisch:** Builder · **Zweck in einem Satz:** Die Konstruktion eines komplexen Objekts von seiner Darstellung trennen und schrittweise durchführen.

Problem

Eine Sendung hat zwei Pflichtangaben (Empfänger, Gewicht) und viele optionale (Express, Versicherungswert, Nachnahme, Hinweis). Ein Konstruktor mit acht Parametern ist unlesbar – `new Sendung("Hansen", 2.5m, true, false, 0, 49.90m, null, true)` – und viele Überladungen („Teleskop-Konstruktoren“) sind unübersichtlich.

Lösung

Ein eigener `SendungBuilder` nimmt die Pflichtangaben im Konstruktor entgegen und bietet für jede optionale Angabe eine Methode, die den Builder selbst zurückgibt. So entsteht eine lesbare Methodenkette. Erst `Build()` prüft alle Regeln zentral und liefert das fertige Objekt. Ungültige Sendungen entstehen gar nicht erst.

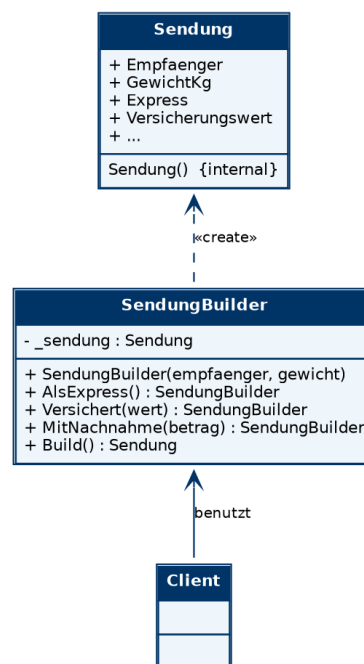


Abb.: Erbauer (Builder) – Klassendiagramm der Demo

Rolle	In der Demo
Erbauer (Builder)	<code>SendungBuilder</code>
Produkt	<code>Sendung</code> (Konstruktor <code>internal</code>)
Client	Program

Code-Ausschnitt

```
Sendung s = new SendungBuilder("MedZentrum Lübeck", 4m)
    .AlsExpress()
    .Versichert(500m)
    .MitHinweis("Kühlware – nicht stapeln")
    .Build(); // prüft: Express nur bis 20 kg
```

Konsequenzen

Vorteile	Nachteile
Lesbarer, selbsterklärender Code Zentrale Validierung in <code>Build()</code> Produkt kann unveränderlich bleiben	Zusätzliche Klasse pro Produkt Bei wenigen Parametern überdimensioniert (in C# oft Objektinitialisierer oder benannte Parameter ausreichend)

Im .NET-Alltag: `StringBuilder`, `UriBuilder`, `SqlConnectionStringBuilder`, `WebApplication.CreateBuilder()` in ASP.NET Core.

Nicht verwechseln

Die GoF-Originalform kennt zusätzlich einen **Direktor**, der die Bauschritte in fester Reihenfolge aufruft. Die heute verbreitete „Fluent Builder“-Variante ohne Direktor ist eine Vereinfachung.

5.4 Weitere Erzeugungsmuster in Kurzform

Muster	Zweck	Beispiel
Abstrakte Fabrik (Abstract Factory)	Familien zusammengehöriger Objekte erzeugen, ohne ihre konkreten Klassen zu nennen.	Eine <code>IUiFabrik</code> liefert Button, Textfeld und Dialog passend zum Design „Hell“ oder „Dunkel“.
Prototyp (Prototype)	Neue Objekte durch Kopieren eines Musterobjekts erzeugen.	Eine vorkonfigurierte Standardsendung wird geklont und nur im Empfänger angepasst (<code>ICloneable</code> , <code>MemberwiseClone</code>).

6 Strukturmuster

Strukturmuster beschreiben, wie Klassen und Objekte zu größeren, flexiblen Strukturen zusammengesetzt werden. Viele von ihnen „umhüllen“ ein anderes Objekt – entscheidend ist, **wozu** sie das tun.

6.1 Adapter

Kategorie: Struktur, Klasse oder Objekt · **Englisch:** Adapter (auch Wrapper) · **Zweck in einem Satz:** Die Schnittstelle einer Klasse an eine andere, erwartete Schnittstelle anpassen.

Problem

Die Nordfracht-Software arbeitet mit der Schnittstelle **IWaage**, die das Gewicht in Kilogramm als **decimal** liefert. Eine neu angeschaffte US-Waage bringt eine Herstellerbibliothek mit, deren Klasse **UsScaleDevice** das Gewicht in Pfund als **double** liefert. Den Code dieser Bibliothek kann man nicht ändern.

Lösung

Ein **UsWaageAdapter** implementiert die eigene Schnittstelle **IWaage** und hält intern ein **UsScaleDevice**. Jeder Aufruf wird übersetzt: Pfund lesen, in Kilogramm umrechnen, als **decimal** zurückgeben. Für den Rest des Programms sieht die US-Waage aus wie jede andere Waage.

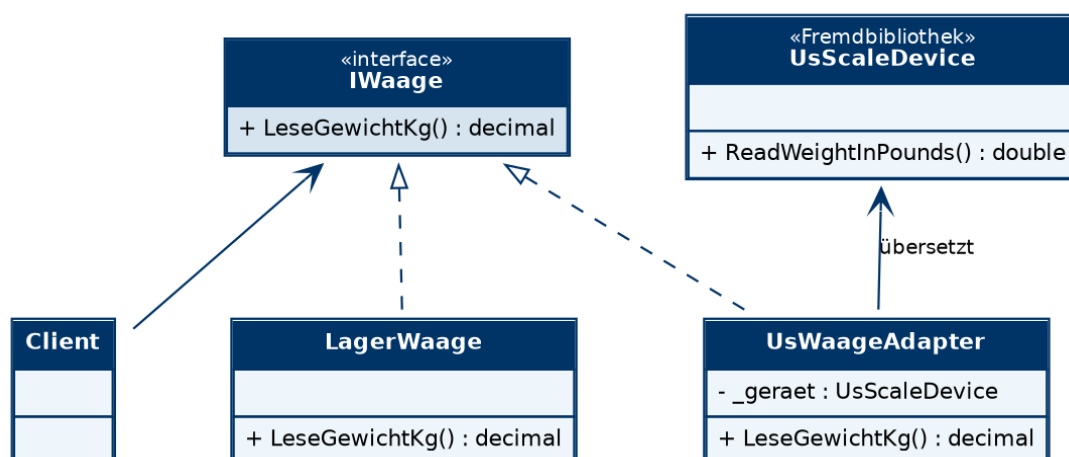


Abb.: Adapter – Klassendiagramm der Demo

Rolle	In der Demo
Ziel (Target)	IWaage
Adaptierte Klasse (Adaptee)	UsScaleDevice (Fremdbibliothek)
Adapter	UsWaageAdapter
Client	Program

Code-Ausschnitt

```
public class UsWaageAdapter : IWaage
{
    private readonly UsScaleDevice _geraet;
    public UsWaageAdapter(UsScaleDevice geraet) { _geraet = geraet; }

    public decimal LeseGewichtKg()
    {
        decimal pfund = (decimal)_geraet.ReadWeightInPounds();
        return decimal.Round(pfund * 0.45359237m, 2);
    }
}
```

Konsequenzen

Vorteile	Nachteile
Fremde oder alte Klassen ohne Änderung nutzbar Übersetzungslogik an einer Stelle	Zusätzliche Klasse je Anpassung Kann schlechtes Design der Altklasse verdecken

Im .NET-Alltag: Datenbank-Adapter ([SqlDataAdapter](#)), Wrapper um COM-Komponenten, Anbindung von Fremd-APIs in jeder Schnittstellenschicht.

Nicht verwechseln

Adapter ändert die Schnittstelle. **Dekorierer** behält die Schnittstelle und erweitert das Verhalten. **Stellvertreter (Proxy)** behält die Schnittstelle und kontrolliert den Zugriff. **Fassade** vereinfacht ein ganzes Subsystem.

6.2 Dekorierer (Decorator)

Kategorie: Struktur, Objekt · **Englisch:** Decorator · **Zweck in einem Satz:** Einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzufügen – als flexible Alternative zur Unterklassenbildung.

Problem

Zum Standardversand können Express-Zuschlag, Versicherung und Nachnahme frei kombiniert werden. Mit Vererbung bräuchte man eine Klasse für jede Kombination: 3 Zusätze ergeben $2^3 = 8$ Klassen, 4 Zusätze bereits 16. Diese **Klassenexplosion** ist nicht wartbar.

Lösung

Alle Bausteine implementieren [IVersandkosten](#). Der [StandardVersand](#) ist die Grundkomponente. Jeder Zusatz ist ein Dekorierer, der ebenfalls [IVersandkosten](#) implementiert und ein anderes [IVersandkosten](#)-Objekt im Konstruktor erhält. Er ruft dessen Methoden auf und ergänzt sein eigenes Verhalten. So lassen sich Zusätze wie Zwiebelschalen beliebig schachteln.

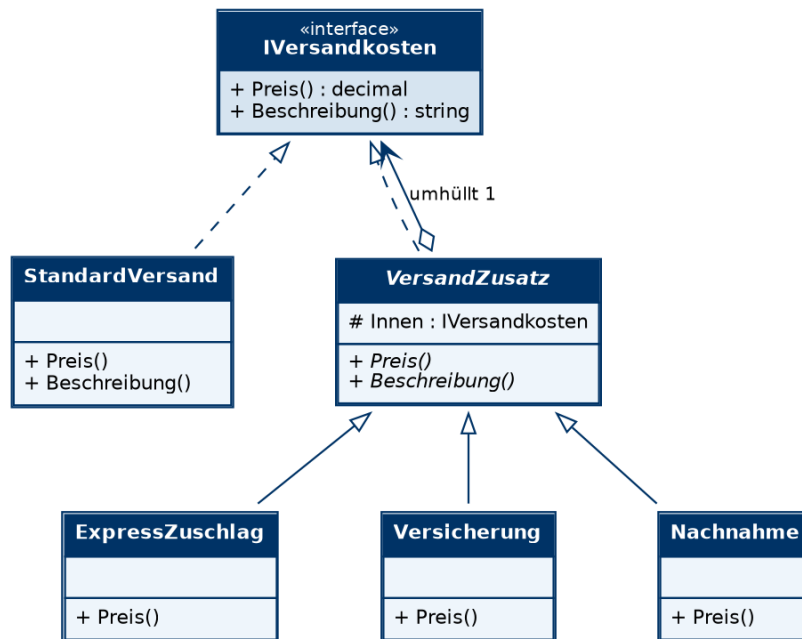


Abb.: Dekorierer (Decorator) – Klassendiagramm der Demo

Rolle	In der Demo
Komponente	IVersandkosten
Konkrete Komponente	StandardVersand
Dekorierer (abstrakt)	VersandZusatz mit Feld Innen
Konkrete Dekorierer	ExpressZuschlag, Versicherung, Nachnahme

Code-Ausschnitt

```

public class ExpressZuschlag : VersandZusatz
{
    public ExpressZuschlag(IVersandkosten innen) : base(innen) { }
    public override decimal Preis() { return Innen.Preis() + 9.00m; }
}

IVersandkosten k = new Nachnahme(
    new Versicherung(
        new ExpressZuschlag(
            new StandardVersand()));
// 5,90 + 9,00 + 3,50 + 4,00 = 22,40 EUR

```

Konsequenzen

Vorteile	Nachteile
- Beliebige Kombinationen zur Laufzeit - Keine Klassenexplosion - Jeder Zusatz einzeln testbar (Single Responsibility)	- Viele kleine Objekte, verschachtelte Aufrufe schwer zu debuggen - Reihenfolge kann eine Rolle spielen (z. B. prozentuale Zuschläge)

Im .NET-Alltag: Die Stream-Klassen: `new GZipStream(new BufferedStream(new FileStream(...)))`. In ASP.NET Core lassen sich Dienste per Scrutor-Bibliothek dekorieren (ungeprüft).

Nicht verwechseln

Dekorierer und Kompositum sehen im Diagramm ähnlich aus. Der Dekorierer umhüllt **genau ein** Objekt und ergänzt Verhalten, das Kompositum enthält **beliebig viele** Kinder und bildet eine Baumstruktur.

6.3 Fassade (Facade)

Kategorie: Struktur, Objekt · **Englisch:** Facade · **Zweck in einem Satz:** Eine einheitliche, vereinfachte Schnittstelle zu einer Menge von Schnittstellen eines Subsystems bieten.

Problem

Um eine Sendung aufzugeben, müssen drei Subsysteme in der richtigen Reihenfolge angesprochen werden: Lager (Artikel reservieren), Abrechnung (Rechnung erstellen), Tracking (Sendungsnummer vergeben). Web-Shop, Filial-Software und App würden diese Logik jeweils wiederholen – und bei jeder Änderung müssten alle angepasst werden.

Lösung

Die **SendungFacade** bietet eine einzige Methode **Aufgeben()**, die das Zusammenspiel intern steuert. Aufrufer kennen nur noch die Fassade. Die Subsysteme bleiben weiterhin direkt erreichbar – die Fassade verbietet nichts, sie vereinfacht nur den häufigsten Anwendungsfall.

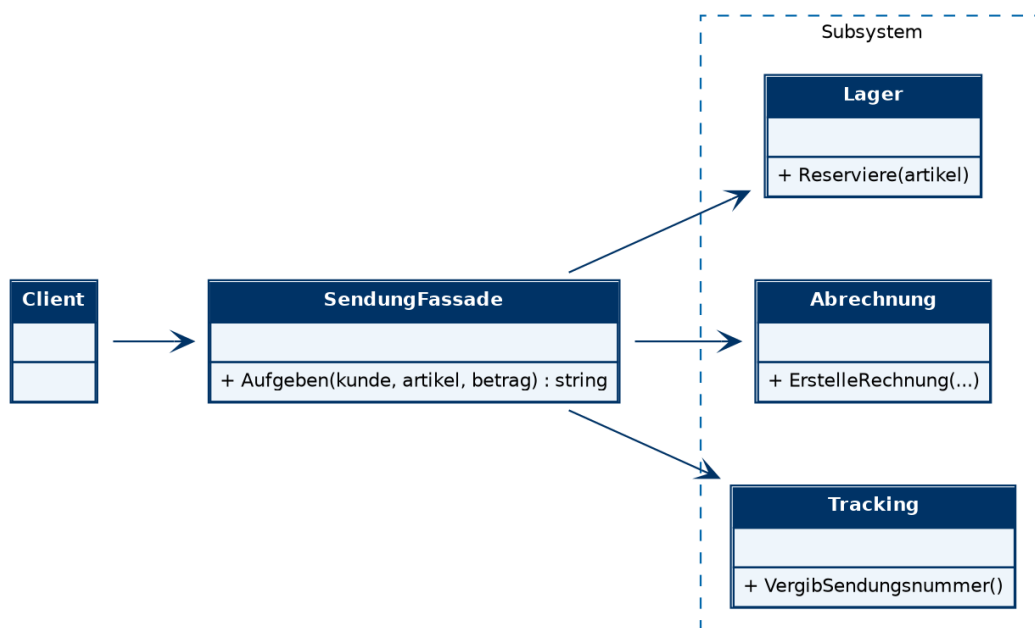


Abb.: Fassade (Facade) – Klassendiagramm der Demo

Rolle	In der Demo
Fassade	SendungFacade
Subsystemklassen	Lager, Abrechnung, Tracking
Client	Web-Shop, Filiale, App (in der Demo: Program)

Code-Ausschnitt

```
public string Aufgeben(string kunde, string artikel, decimal betrag)
{
    if (!_lager.Reserviere(artikel))
        throw new InvalidOperationException("Artikel nicht verfügbar.");
    _abrechnung.ErstelleRechnung(kunde, betrag);
    return _tracking.VergibSendungsnummer();
}
```

Konsequenzen

Vorteile	Nachteile
Einfache Nutzung komplexer Subsysteme Lose Kopplung zwischen Client und Subsystem Ablauf an einer Stelle änderbar	Gefahr eines „Gottobjekts“, wenn die Fassade zu viel übernimmt Zusätzliche Schicht

Im .NET-Alltag: `File.ReadAllText()` verbirgt `FileStream`, `StreamReader` und Encoding. Service-Klassen in Schichtenarchitekturen sind oft Fassaden.

6.4 Weitere Strukturmuster in Kurzform

Muster	Zweck	Beispiel
Brücke (Bridge)	Abstraktion und Implementierung trennen, sodass beide unabhängig variieren.	Berichtsarten (Tages-, Monatsbericht) × Ausgabeformate (PDF, CSV), ohne jede Kombination als Klasse.
Kompositum (Composite)	Teil-Ganzes-Hierarchien als Baum; einzelne Objekte und Gruppen werden gleich behandelt.	Container enthält Paletten, Paletten enthalten Pakete – <code>Gewicht()</code> funktioniert auf jeder Ebene. Dateisystem, WPF-Visual-Tree.
Fliegengewicht (Flyweight)	Viele feingranulare Objekte teilen sich unveränderlichen Zustand, um Speicher zu sparen.	Zeichen in einem Texteditor teilen sich Schriftinformationen; String-Interning in .NET.
Stellvertreter (Proxy)	Platzhalter, der den Zugriff auf ein anderes Objekt kontrolliert.	Lazy-Loading-Proxys in Entity Framework, Schutz-Proxy mit Rechteprüfung, Remote-Proxy für Webdienste.

7 Verhaltensmuster

Verhaltensmuster beschreiben, wie Objekte zusammenarbeiten, wie Zuständigkeiten verteilt werden und wie Algorithmen gekapselt sind. Mit elf Mustern ist es die größte Kategorie.

7.1 Strategie (Strategy)

Kategorie: Verhalten, Objekt · **Englisch:** Strategy · **Zweck in einem Satz:** Eine Familie von Algorithmen definieren, jeden kapseln und austauschbar machen.

Problem

Nordfracht berechnet Preise nach verschiedenen Tarifen: Standard, Geschäftskunde, Sperrgut. Ohne Muster entsteht eine wachsende **switch**-Anweisung. Jeder neue Tarif ändert dieselbe Methode, und die Tarife lassen sich nicht einzeln testen.

Lösung

Jeder Tarif wird eine eigene Klasse, die die Schnittstelle **IPreisStrategie** mit der Methode **Berechne()** implementiert. Der **Preisrechner** (Kontext) hält eine Referenz auf eine Strategie und delegiert die Berechnung. Die Strategie wird von außen übergeben und kann zur Laufzeit gewechselt werden.

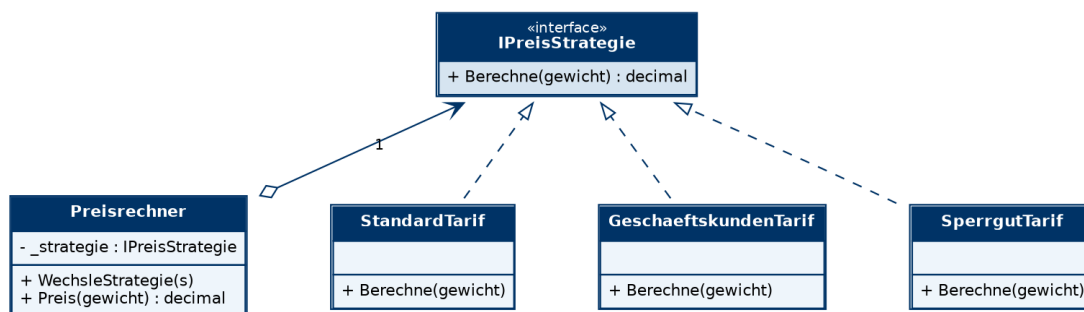


Abb.: Strategie (Strategy) – Klassendiagramm der Demo

Rolle	In der Demo
Strategie	IPreisStrategie
Konkrete Strategien	StandardTarif, GeschaeftskundenTarif, SperrgutTarif
Kontext	Preisrechner

Code-Ausschnitt

```

// Vorher – wächst mit jedem Tarif:
switch (tarif) { case "Standard": ... case "Geschäft": ... case "Sperrgut": ... }

// Nachher:
Preisrechner rechner = new Preisrechner(new StandardTarif());
rechner.WechsleStrategie(new SperrgutTarif()); // Austausch zur Laufzeit
decimal preis = rechner.Preis(10m);
  
```

Konsequenzen

Vorteile	Nachteile
Neue Algorithmen ohne Änderung am Kontext <code>switch</code> -Wüsten verschwinden Jede Strategie einzeln testbar	Aufrufer muss die Strategien kennen und auswählen Mehr Klassen; bei sehr einfachen Algorithmen reicht ein Delegat (<code>Func<decimal, decimal></code>)

Im .NET-Alltag: `IComparer<T>` für `List<T>.Sort()`, `IEqualityComparer<T>` für `Dictionary`, Delegaten und Lambdas als „Strategie light“.

Nicht verwechseln

Zustand (State) hat dieselbe Struktur. Bei der Strategie wählt der **Aufrufer** das Verhalten. Beim Zustand wechselt das **Objekt selbst** sein Verhalten, sobald sich sein innerer Zustand ändert (z. B. Sendung „angelegt“ → „unterwegs“ → „zugestellt“ mit jeweils anderen erlaubten Aktionen).

7.2 Beobachter (Observer)

Kategorie: Verhalten, Objekt · **Englisch:** Observer (auch Publish-Subscribe) · **Zweck in einem Satz:** Eine 1-zu-n-Abhängigkeit definieren, sodass bei einer Zustandsänderung alle abhängigen Objekte benachrichtigt werden.

Problem

Ändert sich der Status einer Sendung, sollen der Empfänger per SMS, der Absender per E-Mail und das Leitstand-Dashboard informiert werden. Ruft die Sendung diese Dienste direkt auf, muss sie jeden kennen – und für jede neue Benachrichtigungsart geändert werden.

Lösung

Die Sendung (das **Subjekt**) verwaltet eine Liste von Beobachtern, die nur die Schnittstelle `ISendungsBeobachter` erfüllen müssen. Beobachter melden sich mit `Anmelden()` an und mit `Abmelden()` ab. Ändert sich der Status, ruft die Sendung bei allen angemeldeten Beobachtern `StatusGeändert()` auf. Die Sendung weiß nicht, was die Beobachter damit tun.

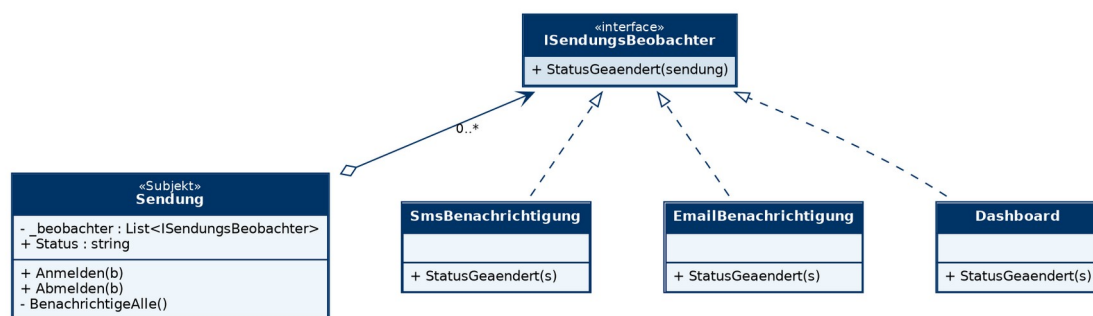


Abb.: Beobachter (Observer) – Klassendiagramm der Demo

Rolle	In der Demo
Subjekt	Sendung
Beobachter	ISendungsBeobachter
Konkrete Beobachter	SmsBenachrichtigung, EmailBenachrichtigung, Dashboard

Code-Ausschnitt

```
public string Status
{
    get { return _status; }
    set
    {
        if (_status == value) return;
        _status = value;
        foreach (ISendungsBeobachter b in _beobachter.ToArray())
            b.StatusGeaendert(this);
    }
}
```

Konsequenzen

Vorteile	Nachteile
Lose Kopplung zwischen Subjekt und Beobachtern Beobachter zur Laufzeit an- und abmeldbar Neue Beobachter ohne Änderung am Subjekt (Open/Closed)	Reihenfolge der Benachrichtigung nicht garantiert Vergessenes Abmelden verhindert die Speicherfreigabe (Speicherleck) Kaskaden von Aktualisierungen schwer zu überblicken

Im .NET-Alltag: C#-Events sind ein in die Sprache eingebautes Beobachtermuster.

[INotifyPropertyChanged](#) in MVVM, [IObservable<T>/IObserver<T>](#) und die Reactive Extensions bauen darauf auf.

Nicht verwechseln

Beim **Vermittler (Mediator)** kommunizieren Objekte nicht direkt, sondern über eine zentrale Stelle, die entscheidet, wer was erfährt. Beim Beobachter entscheidet jeder Beobachter selbst durch seine Anmeldung.

7.3 Befehl (Command)

Kategorie: Verhalten, Objekt · **Englisch:** Command · **Zweck in einem Satz:** Eine Anforderung als Objekt kapseln, um sie zu speichern, zu protokollieren, in eine Warteschlange zu stellen oder rückgängig zu machen.

Problem

Lagerbuchungen (Einlagern, Auslagern) sollen rückgängig gemacht werden können. Ein direkter Methodenaufruf wie `platz.Zugang(50)` ist dafür zu flüchtig – nach dem Aufruf weiß niemand mehr, was getan wurde.

Lösung

Jede Aktion wird ein Objekt, das die Schnittstelle [ILagerBefehl](#) mit [Ausfuehren\(\)](#) und [Rueckgaengig\(\)](#) implementiert und alle nötigen Daten (Lagerplatz, Menge) speichert. Das [Lagerterminal](#) führt Befehle aus und legt sie auf einen Stapel. Rückgängig nimmt den obersten Befehl vom Stapel und ruft [Rueckgaengig\(\)](#) auf.

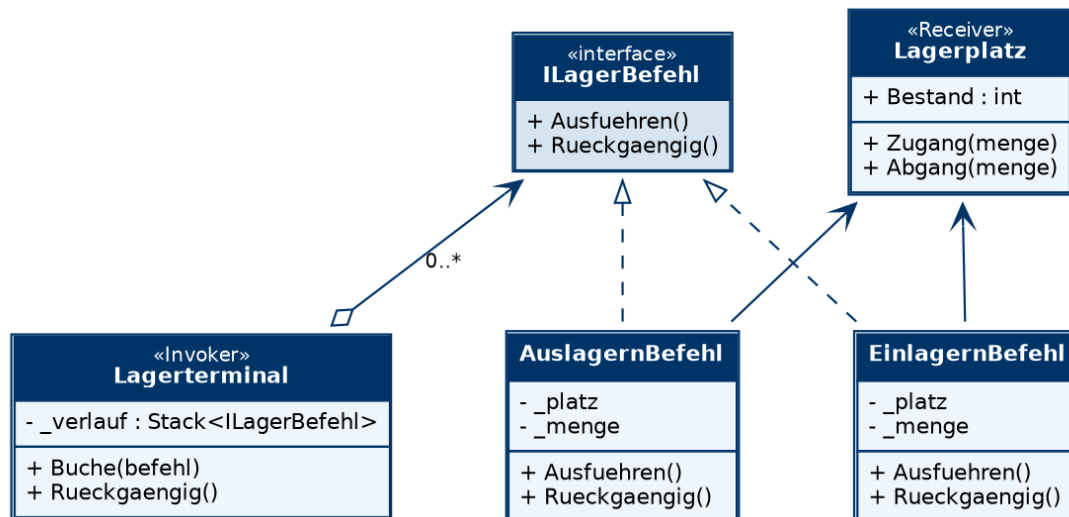


Abb.: Befehl (Command) – Klassendiagramm der Demo

Rolle	In der Demo
Befehl	<code>ILagerBefehl</code>
Konkrete Befehle	<code>EinlagernBefehl</code> , <code>AuslagernBefehl</code>
Empfänger (Receiver)	<code>Lagerplatz</code> – erledigt die eigentliche Arbeit
Aufrufer (Invoker)	<code>Lagerterminal</code> mit <code>Stack<ILagerBefehl></code>

Code-Ausschnitt

```

terminal.Buche(new EinlagernBefehl(platz, 50)); // Bestand 50
terminal.Buche(new AuslagernBefehl(platz, 20)); // Bestand 30
terminal.Rueckgaengig();                       // Bestand 50
terminal.Rueckgaengig();                       // Bestand 0
  
```

Konsequenzen

Vorteile	Nachteile
Rückgängig/Wiederholen einfach umsetzbar Befehle speicherbar, protokollierbar, in Warteschlangen Aufrufer und Empfänger entkoppelt	Eine Klasse pro Aktion Rückgängig muss für jede Aktion korrekt gedacht werden (was, wenn der Bestand inzwischen fehlt?)

Im .NET-Alltag: `ICommand` in WPF und MVVM, Undo/Redo in Editoren, Nachrichten in Warteschlangen (Message Queues), Migrationen in Entity Framework mit `Up()` und `Down()`.

7.4 Weitere Verhaltensmuster in Kurzform

Muster	Zweck	Beispiel
Zuständigkeitskette (Chain of Responsibility)	Eine Anfrage wird entlang einer Kette weitergereicht, bis ein Glied sie bearbeitet.	Freigabe nach Betrag: Sachbearbeiter → Teamleitung → Geschäftsführung; ASP.NET-Core-Middleware.
Interpreter	Grammatik einer einfachen Sprache als Klassen darstellen und Ausdrücke auswerten.	Filterausdrücke wie „Gewicht > 20 UND Express“; Expression Trees.

Muster	Zweck	Beispiel
Iterator	Elemente einer Sammlung nacheinander durchlaufen, ohne ihren Aufbau zu kennen.	<code>IEnumerable<T></code> , <code>foreach</code> , <code>yield return</code> .
Vermittler (Mediator)	Zentrales Objekt steuert die Kommunikation vieler Objekte, die sich nicht kennen.	Leitstand koordiniert Fahrzeuge; Bibliothek MediatR; Tower beim Flugverkehr.
Memento	Inneren Zustand eines Objekts sichern und wiederherstellen, ohne die Kapselung zu verletzen.	Entwurfsstand eines Formulars speichern; Spielstände.
Zustand (State)	Objekt ändert sein Verhalten, wenn sich sein innerer Zustand ändert.	Sendung: „angelegt“ erlaubt Stornieren, „zugestellt“ nicht mehr.
Schablonenmethode (Template Method)	Oberklasse legt den Ablauf fest, Unterklassen füllen einzelne Schritte.	Berichtserstellung: Kopf, Inhalt, Fuß – nur der Inhalt variiert.
Besucher (Visitor)	Neue Operationen auf einer Objektstruktur, ohne deren Klassen zu ändern.	Zollerklärung und Gefahrgutprüfung über alle Pakettypen; <code>ExpressionVisitor</code> in .NET.

8 Muster im .NET-Alltag

Viele Muster sind so verbreitet, dass sie in Sprache und Framework eingebaut sind. Man nutzt sie täglich, oft ohne es zu bemerken.

Muster	Wo in .NET / C#
Iterator	<code>IEnumerable<T></code> / <code>IEnumerator<T></code> , <code>foreach</code> , <code>yield return</code>
Beobachter (Observer)	<code>event</code> / Delegaten, <code>INotifyPropertyChanged</code> , <code>IObservable<T></code>
Befehl (Command)	<code>ICommand</code> in WPF/MAUI
Dekorierer (Decorator)	Stream-Klassen: <code>BufferedStream</code> , <code>GZipStream</code> , <code>CryptoStream</code>
Erbauer (Builder)	<code>StringBuilder</code> , <code>UriBuilder</code> , <code>WebApplication.CreateBuilder()</code>
Fabrikmethode / Abstrakte Fabrik (Factory Method / Abstract Factory)	<code>DbProviderFactory</code> , <code>IHttpClientFactory</code> , <code>ILoggerFactory</code>
Strategie (Strategy)	<code>IComparer<T></code> , <code>IEqualityComparer<T></code> , Lambdas als Parameter
Stellvertreter (Proxy)	Lazy-Loading-Proxys in Entity Framework, WCF-/gRPC-Client-Proxys
Singleton	<code>Lazy<T></code> , DI-Lebensdauer <code>AddSingleton</code>
Zuständigkeitskette (Chain of Responsibility)	ASP.NET-Core-Middleware-Pipeline
Prototyp (Prototype)	<code>ICloneable</code> , <code>MemberwiseClone()</code> , <code>with</code> -Ausdruck bei <code>record</code>
Fliegengewicht (Flyweight)	String-Interning (<code>string.Intern</code>)

9 Kritik und Antimuster

9.1 Kritik an Entwurfsmustern

- **Overengineering:** Muster werden eingesetzt, weil man sie kennt, nicht weil das Problem sie erfordert. Ein Programm mit drei Klassen braucht keine Abstrakte Fabrik.
- **Sprachdefizite:** Schon 1996 wurde gezeigt, dass viele GoF-Muster in dynamischen Sprachen unsichtbar oder deutlich einfacher sind. Auch in C# sind durch Delegaten, Lambdas, Events und `yield` viele Muster zu wenigen Zeilen geschrumpft.
- **Singleton als Antimuster:** Globaler Zustand erschwert Tests und versteckt Abhängigkeiten. Viele Teams verzichten zugunsten von Dependency Injection darauf.
- **Mehr Klassen, mehr Indirektion:** Jedes Muster fügt Schichten hinzu. Code wird flexibler, aber nicht automatisch verständlicher.

Faustregel

Muster sind ein **Werkzeugkasten, kein Bauplan**. Schreiben Sie zuerst die einfachste funktionierende Lösung. Wenn Sie merken, dass sich eine Stelle immer wieder ändert oder Code dupliziert wird, führen Sie das passende Muster per **Refactoring** ein.

9.2 Antimuster

Ein **Antimuster** beschreibt eine häufig gewählte, aber schlechte Lösung – samt Erkennungsmerkmalen und dem Weg heraus.

Antimuster	Beschreibung	Ausweg
Gottobjekt (God Object / Blob)	Eine Klasse weiß und kann alles, oft tausende Zeilen.	Zuständigkeiten aufteilen (Single Responsibility), Fassade + Subsysteme
Spaghetti-Code	Unstrukturierter Code mit verwobenem Kontrollfluss.	Methoden extrahieren, Strategie, Zustand
Goldener Hammer	„Wer einen Hammer hat, sieht überall Nägel“ – ein bekanntes Werkzeug für alles.	Problem zuerst analysieren, dann Lösung wählen
Copy-and-Paste-Programmierung	Duplizierter Code an vielen Stellen.	Gemeinsame Methode, Schablonenmethode
Lavafluss (Lava Flow)	Toter, unverständlicher Code, den sich niemand zu löschen traut.	Tests schreiben, dann entfernen
Magische Zahlen	Unerklärte Konstanten wie <code>31.5</code> mitten im Code.	Benannte Konstanten (<code>MaxGewichtKg</code>)

10 Entwurfsmuster in der Abschlussprüfung

In der Abschlussprüfung Teil 2 für Fachinformatikerinnen und Fachinformatiker Anwendungsentwicklung tauchen Entwurfsmuster regelmäßig auf. Geprüft wird dabei weniger das Auswendiglernen als die **Transferleistung**: ein Muster in einem gegebenen Szenario erkennen, begründen und umsetzen.

10.1 Typische Aufgabenformen

- Ein UML-Klassendiagramm ist gegeben – **welches Muster** liegt vor? Begründen Sie anhand der Beziehungen.
- Ein Szenario wird beschrieben – **welches Muster** löst das Problem? Nennen Sie einen Vor- und einen Nachteil.
- Ein unvollständiges Klassendiagramm soll **ergänzt** werden (fehlende Schnittstelle, Beziehung oder Methode).
- Pseudocode oder C#-Code eines Musters soll **vervollständigt** werden, z. B. die Benachrichtigungsschleife beim Beobachter.

10.2 Erkennungsmerkmale im Klassendiagramm

Muster	Woran erkenne ich es im UML-Diagramm?
Singleton	Eine Klasse, Konstruktor mit - (private), statisches (unterstrichenes) Attribut und statische Methode, die den eigenen Typ zurückgibt
Fabrikmethode (Factory Method)	Abstrakte Klasse mit abstrakter Methode, deren Rückgabotyp ein Interface ist; Unterklassen mit «create»-Abhängigkeit auf konkrete Produkte
Adapter	Klasse realisiert ein Interface und hat eine Assoziation zu einer Klasse mit anders benannten Methoden
Dekorierer (Decorator)	Klasse realisiert ein Interface und hat eine Aggregation (Multiplizität 1) zu demselben Interface
Fassade (Facade)	Eine Klasse mit Assoziationen zu mehreren Klassen eines Subsystems, Client nur mit Fassade verbunden
Strategie (Strategy)	Kontextklasse mit Aggregation zu einem Interface mit (meist) einer Methode; mehrere Realisierungen
Beobachter (Observer)	Subjekt mit Aggregation 0..* zu einem Beobachter-Interface sowie Methoden zum An- und Abmelden
Befehl (Command)	Interface mit <code>execute()</code> (und ggf. <code>undo()</code>), Invoker mit Liste/Stapel davon, konkrete Befehle mit Assoziation zum Receiver

Prüfungstipp

Begründen Sie immer mit der **Struktur** (wer kennt wen, welche Beziehung, welche Multiplizität) **und** dem **Zweck** (welches Problem wird gelöst). „Das ist ein Beobachter, weil es eine Liste gibt“ reicht nicht – „weil das Subjekt eine 0..*-Aggregation zum Beobachter-Interface hat und bei Zustandsänderung alle angemeldeten Beobachter über eine gemeinsame Methode benachrichtigt“ schon.

11 Ausblick

Die GoF-Muster sind über 30 Jahre alt – und dennoch Grundwissen. Die Musteridee hat sich seitdem auf viele weitere Ebenen ausgebreitet:

Bereich	Beispiele
Architekturmuster	Schichtenarchitektur, Hexagonale Architektur (Ports & Adapters), Clean Architecture, MVC, MVVM, MVU
Unternehmensanwendungen	Repository, Unit of Work, Data Mapper; Aggregate, Entity, Value Object (Domain-Driven Design)
Verteilte Systeme / Cloud	Circuit Breaker, Retry, Saga, CQRS, Event Sourcing, API Gateway, Sidecar
Nebenläufigkeit	Producer-Consumer, Async/Await-Muster, Actor-Modell
Funktionale Programmierung	Higher-Order Functions ersetzen Strategie, Befehl und Schablonenmethode; unveränderliche Daten ersetzen Memento

Ein wichtiger Trend: Viele klassische Muster wandern in die **Sprache** (Events, Lambdas, [record](#)) oder in **Frameworks** (Dependency Injection, Middleware). Man implementiert sie seltener von Hand, muss sie aber verstehen, um die Frameworks richtig zu nutzen.

Fazit

Entwurfsmuster sind **Vokabeln für Softwareentwürfe**. Wer sie kennt, erkennt Strukturen in fremdem Code, diskutiert Entwürfe präziser und greift auf erprobte Lösungen zurück. Entscheidend ist nicht, möglichst viele Muster einzusetzen, sondern das **Problem hinter dem Muster** zu erkennen – und dann bewusst zu entscheiden, ob sich der zusätzliche Aufwand lohnt.

12 Die Demo-Solution

Die Solution `GoF-Entwurfsmuster_Demo.sln` enthält drei Konsolenprojekte für .NET Framework 4.8 (C# 7.3), eines je GoF-Kategorie. Jede Musterdatei beginnt mit einem Kommentarblock nach dem Schema **Problem – Lösung – Merkmal – .NET-Beispiel**.

Projekt	Muster	Szenario
Muster1_Erzeugungsmuster	Singleton, Fabrikmethode, Erbauer	Konfiguration; Land- und Seeversand; Sendung mit optionalen Angaben
Muster2_Strukturmuster	Adapter, Dekorierer, Fassade	US-Waage in Pfund; kombinierbare Versandzuschläge; Sendung aufgeben
Muster3_Verhaltensmuster	Strategie, Beobachter, Befehl	Tarife; Statusbenachrichtigung; Lagerbuchungen mit Rückgängig

12.1 Aufgaben zum Selbststudium

1. Ergänzen Sie eine Fabrikmethode für den `LuftVersand` mit dem Transportmittel `Frachtflugzeug`. Welche bestehenden Klassen müssen Sie ändern?
2. Ergänzen Sie einen Dekorierer `Samstagszustellung` (+ 6,00 €). Welche Kombinationen sind jetzt möglich, und wie viele Klassen hätten Sie mit Vererbung gebraucht?
3. Schreiben Sie einen `LogBeobachter`, der jede Statusänderung mit Uhrzeit in eine Datei schreibt. Zeigen Sie, dass `Sendung` dafür nicht geändert werden muss.
4. Bauen Sie den `Preisrechner` so um, dass die Strategie als `Func<decimal, decimal>` übergeben wird. Diskutieren Sie Vor- und Nachteile gegenüber dem Interface.
5. Die Klasse `Konfiguration` ist ein Singleton. Bauen Sie sie so um, dass sie ein Interface `IKonfiguration` implementiert und per Konstruktor übergeben wird. Was gewinnen Sie für Unit-Tests?
6. Ergänzen Sie ein Wiederholen (Redo) für das `Lagerterminal`. Tipp: ein zweiter Stapel.

13 Glossar

Begriff	Erklärung
Aggregation	UML-Beziehung „hat ein/hat mehrere“ mit leerer Raute; das Teil kann ohne das Ganze existieren.
Antimuster	Häufig gewählte, aber nachteilige Lösung für ein wiederkehrendes Problem.
Delegation	Ein Objekt reicht eine Aufgabe an ein anderes weiter, statt sie selbst zu erledigen.
Dependency Injection	Abhängigkeiten werden von außen übergeben statt intern erzeugt.
Gang of Four (GoF)	Die vier Autoren des Buchs „Design Patterns“ (1994), das die 23 klassischen Muster beschreibt.
Idiom	Muster auf der Ebene einer bestimmten Programmiersprache.
Indirektion	Zusätzliche Zwischenschicht zwischen Aufrufer und eigentlicher Arbeit.
Klassensexplosion	Starkes Anwachsen der Klassenzahl durch Vererbung für jede Kombination.
Komposition	Ein Objekt setzt sich aus anderen Objekten zusammen und nutzt deren Fähigkeiten.
Kopplung	Grad der Abhängigkeit zwischen Klassen; lose Kopplung ist das Ziel.
Open/Closed-Prinzip	Software soll offen für Erweiterung, aber geschlossen für Änderung sein.
Realisierung	UML-Beziehung, bei der eine Klasse ein Interface implementiert (gestrichelte Linie, leeres Dreieck).
Refactoring	Verbesserung der inneren Struktur von Code, ohne sein Verhalten zu ändern.
Subjekt	Beim Beobachtermuster das Objekt, dessen Zustand beobachtet wird.

14 Wiederholungsfragen

Die Antworten finden Sie in den angegebenen Kapiteln.

1. Definieren Sie den Begriff „Entwurfsmuster“ und nennen Sie die vier wesentlichen Elemente einer Musterbeschreibung. (Kap. 1)
2. Woher stammt die Idee der Muster, und welches Buch hat sie in der Softwareentwicklung bekannt gemacht? (Kap. 2)
3. Nennen Sie die drei GoF-Kategorien mit ihrer Leitfrage und je zwei Beispielen – jeweils mit deutschem und englischem Namen. (Kap. 3)
4. Worin unterscheiden sich Architekturmuster, Entwurfsmuster und Idiome? (Kap. 3)
5. Erklären Sie das Prinzip „Komposition vor Vererbung“ am Beispiel des Dekorierers. (Kap. 4, 6.2)
6. Warum gilt das Singleton oft als Antimuster? Welche Alternative gibt es? (Kap. 5.1, 9)
7. Grenzen Sie Adapter, Dekorierer und Stellvertreter voneinander ab. (Kap. 6)
8. Ein Onlineshop soll verschiedene Versandkostenberechnungen anbieten, die der Kunde auswählt. Welches Muster eignet sich? Begründen Sie. (Kap. 7.1)
9. Skizzieren Sie das Klassendiagramm des Beobachtermusters mit allen Beziehungen und Multiplizitäten. (Kap. 7.2, 10)
10. Wie lässt sich mit dem Befehlsmuster eine Rückgängig-Funktion umsetzen? (Kap. 7.3)
11. Nennen Sie drei Stellen in .NET, an denen Sie Entwurfsmuster täglich nutzen. (Kap. 8)
12. Worin unterscheiden sich Strategie und Zustand, obwohl ihre Klassendiagramme fast gleich aussehen? (Kap. 7.1)

15 Weiterführende Quellen

- Gamma, Helm, Johnson, Vlissides: „Design Patterns – Elements of Reusable Object-Oriented Software“, Addison-Wesley 1994 (dt. „Entwurfsmuster“).
- Freeman, Robson u. a.: „Head First Design Patterns“, O'Reilly, 2. Auflage 2020 (dt. „Entwurfsmuster von Kopf bis Fuß“).
- refactoring.guru – frei zugängliche, illustrierte Übersicht aller GoF-Muster mit Codebeispielen, auch auf Deutsch.