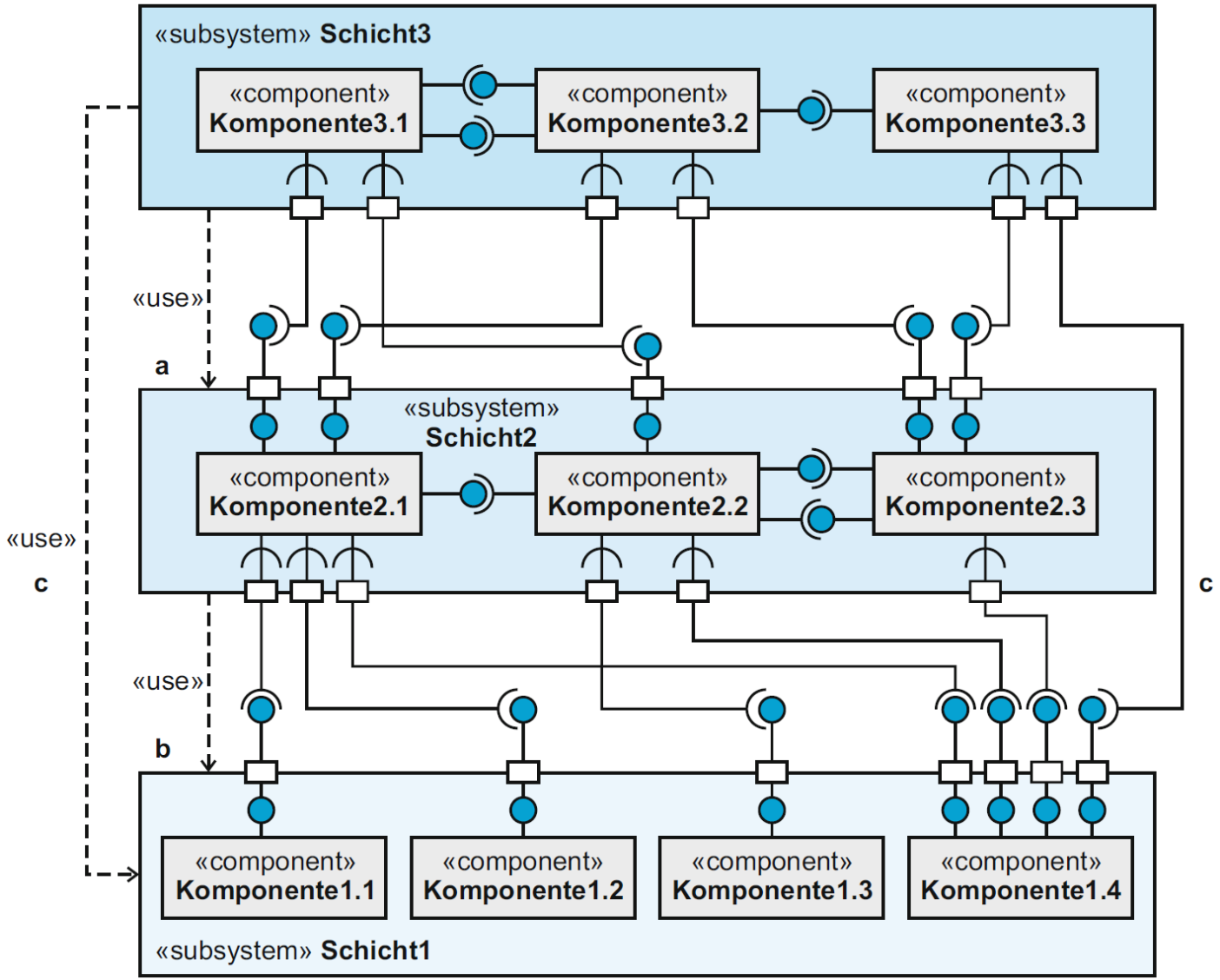
Entwurfsmuster

Design-Patterns

- Unter Design-Patterns (Entwurfsmuster) werden Programmiertechniken verstanden, um bestimmte Abläufe, die häufig wiederkehren, immer auch in ähnlicher Implementierung abzubilden
- Es existieren ein Reihe verschiedener Entwurfsmuster, die gemäß der jeweiligen Aufgabe gewählt werden, es macht keinen Sinn, alle Muster zu verwenden, in der Regel sind bei bestimmten Aufgabenstellungen 4-5 Muster vollkommen ausreichend
- Entwurfsmuster sichern einen hohen Grad an
  - Wiederverwendbarkeit
  - Wiedererkennung und
  - Sicherheit durch erprobte Lösungen
- Entwurfsmuster adressieren verschiedene Bereiche der Softwareentwicklung
  - die Erzeugung von Objekten (z.B. Klassenfabriken)
  - Softwarestrukturen und –komponenten (z.B. Schnittstellen)
  - Verhalten und Bearbeiten von Objekten (z.B. Iteratoren)

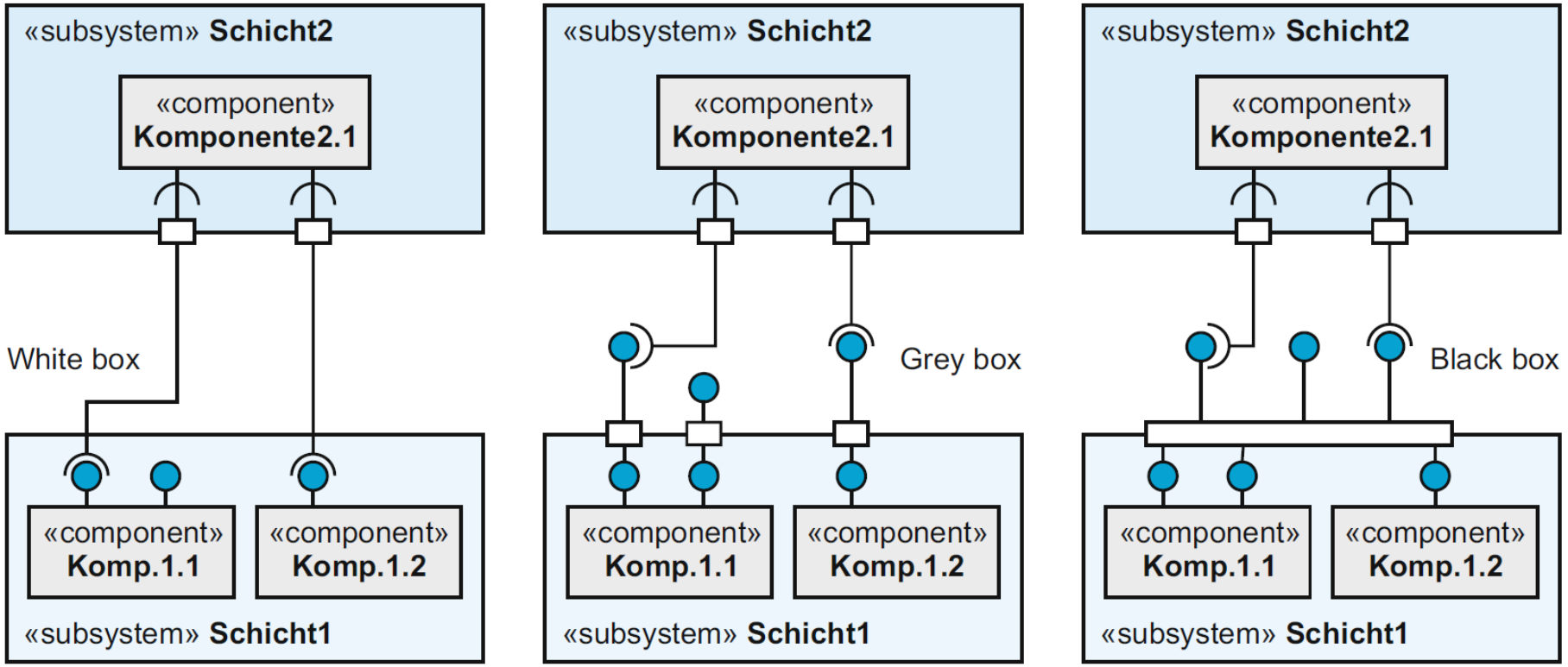
Das Schichten – Muster



Quelle: Balzert

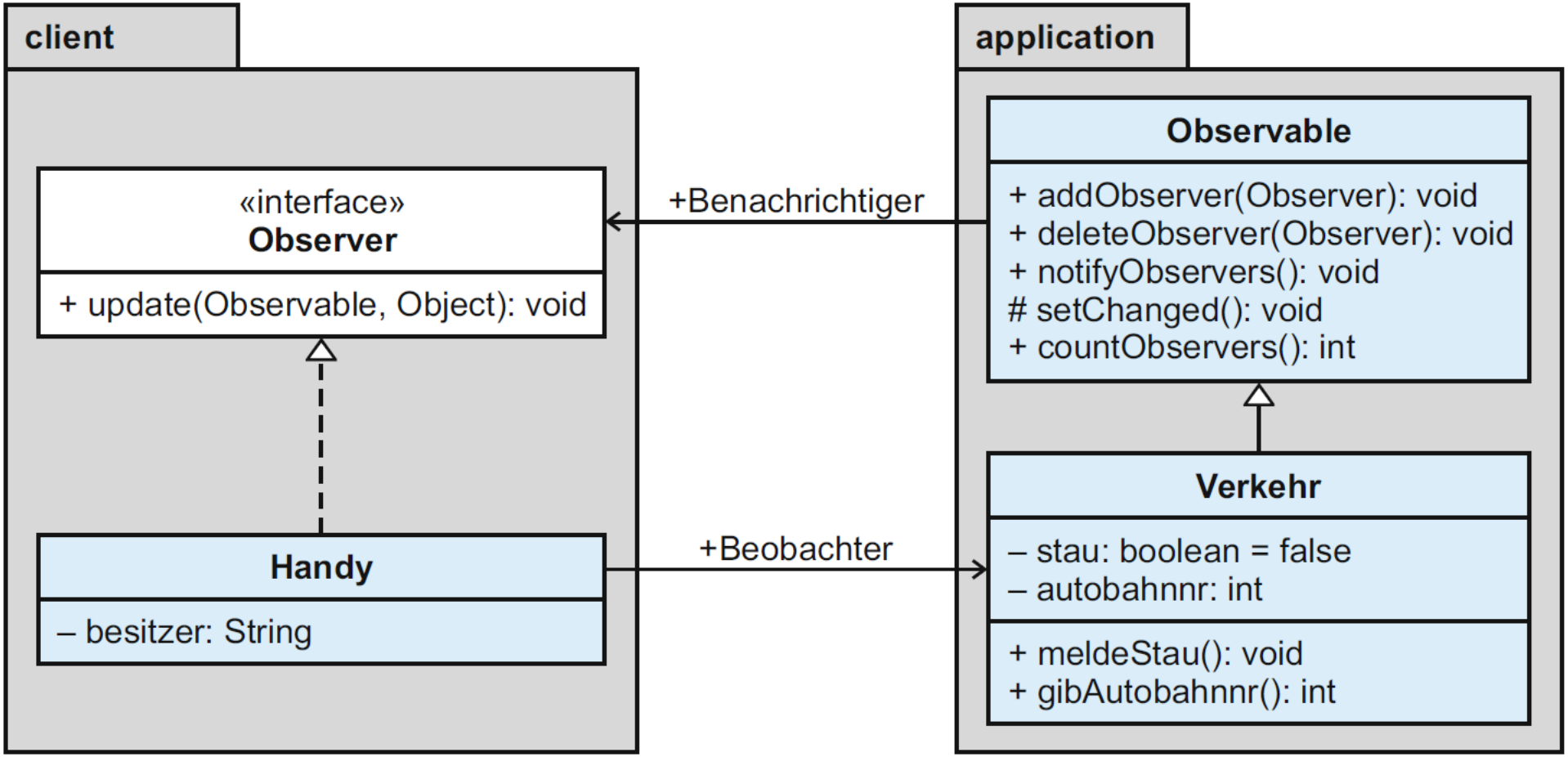
Schnittstellen beim Schichten – Muster

- White Box: Das Moduls liegt offen (implementierungstechnisch ist die Klasse z.B. als Datei verfügbar)
- Black Box: Es sind nur die Schnittstellen (Funktionalität) der Box, nicht aber die interne Struktur sichtbar (z.B. eine Fassadenklasse einer Dll)
- Grey Box: Mischen von White und Grey Box (z.B. Klassenbibliothek)



Quelle: Balzert

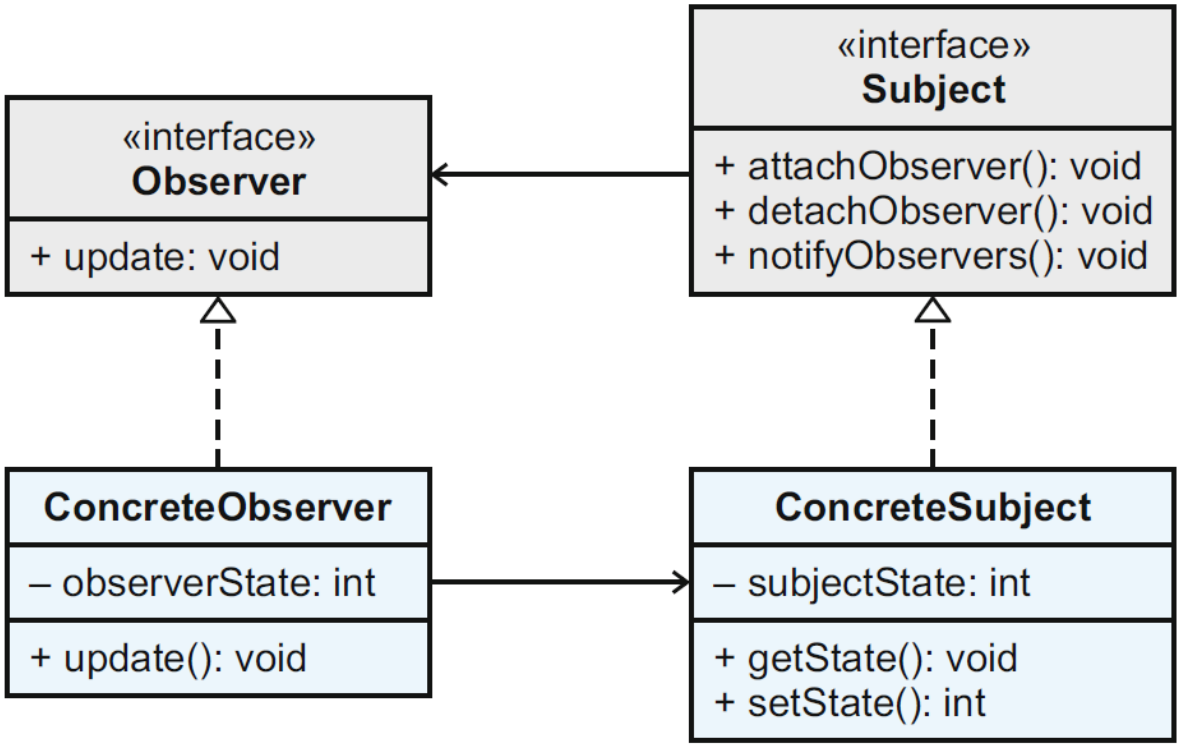
Das Beobachter – Muster



Quelle: Balzert

Das Beobachter – Muster

- Ein Beobachter (Observer) meldet sich beim Subject-Objekt an (mittels attachObserver), in der Regel mit einer Angabe, bei welchen Ereignissen der Observer benachrichtigt werden möchte (siehe Ereigniskonzept C#)
- Damit das Subject-Objekt den Observer ohne genaue Kenntnis des Observers benachrichtigen kann, muß eine Oberserver-Basisklasse die aufzurufende Methode exemplarisch deklarieren, der tatsächliche Oberserver ist immer eine Ableitung dieser Basisklasse



Quelle: Balzert

Das Delegat in C# - ein funktionsbasiertes Beobachter-Muster

- In C# kann mit Hilfe eines Delegates auf eine Funktion in allgemeiner Form verwiesen werden:
```
public delegate void Upd(string message);
```
- Diese Definition beschreibt eine Klasse von Methoden, die einen String als Parameter erhalten und nichts (void) zurückgeben
- Da der Typ unabhängig von einer real vorhandenen Methode deklariert werden kann, kann z.B. ein Observer eine durch das Delegat definierte Methode update implementieren:
```
public static void Update(string message)
{
    System.Console.WriteLine(message);
}
```
- Update entspricht hinsichtlich des Rückgabetyps und der Übergabeparametern der Delegatdefinition Upd
- Um einem Subject-Objekt nun zu sagen, was bei einem Update aufzurufen ist, kann im Unterschied zum Observer-Objekt mit der geerbten Methode update direkt die Funktion für den Aufruf übergeben werden. Das Subject-Objekt muß dazu nur die gemerkten Delegate mit entsprechenden Übergabeparametern rufen

Einfaches Beispiel eines Delegats

- Definition vom Delegat und einer Methode mit dessen Signatur

```
public delegate void Upd(string message);
public static void Update(string message)
{
    System.Console.WriteLine(message);
}
```

- Möglichkeit schaffen, dass sich Funktionsaufrufe in eine Liste eintragen

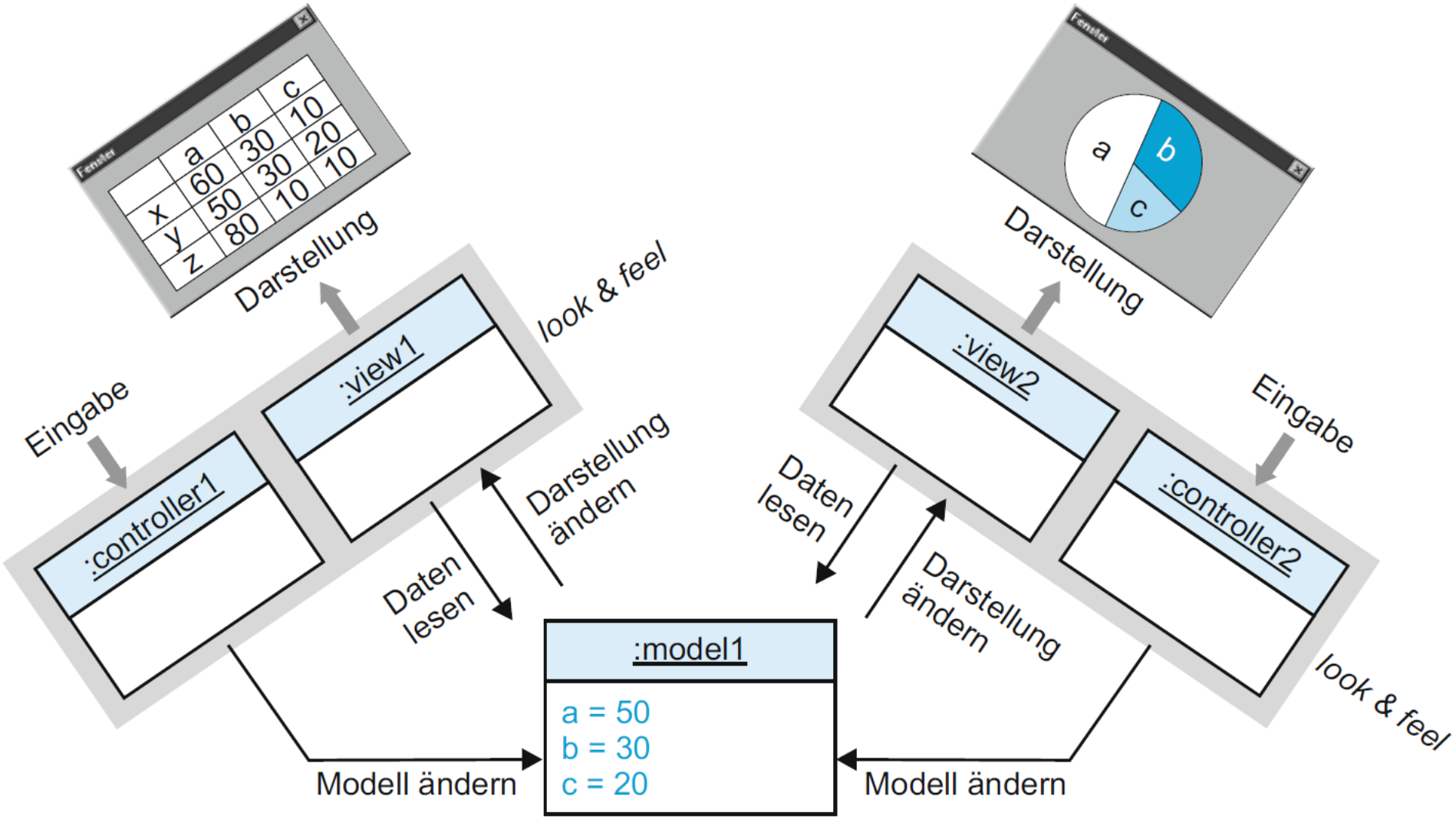
```
static Dictionary<string, Upd> m_functionsToCall = new Dictionary<string, Upd>();
public static void Add(Upd functionToCall, string msg)
{
    m_functionsToCall[msg] = functionToCall;
}
```

- Beispielmethode, die alle registrierten Methoden aufruft

```
public static void DoIt()
{
    foreach (KeyValuePair<string,Upd> kvp in m_functionsToCall)
    {
        kvp.Value(kvp.Key);
    }
}
```

```
Add(Update, "erster");
Add(Update, "zweiter");
Add(Update, "dritter");
DoIt();
```

Das Model – View – Controller – Muster (MVC-Muster)

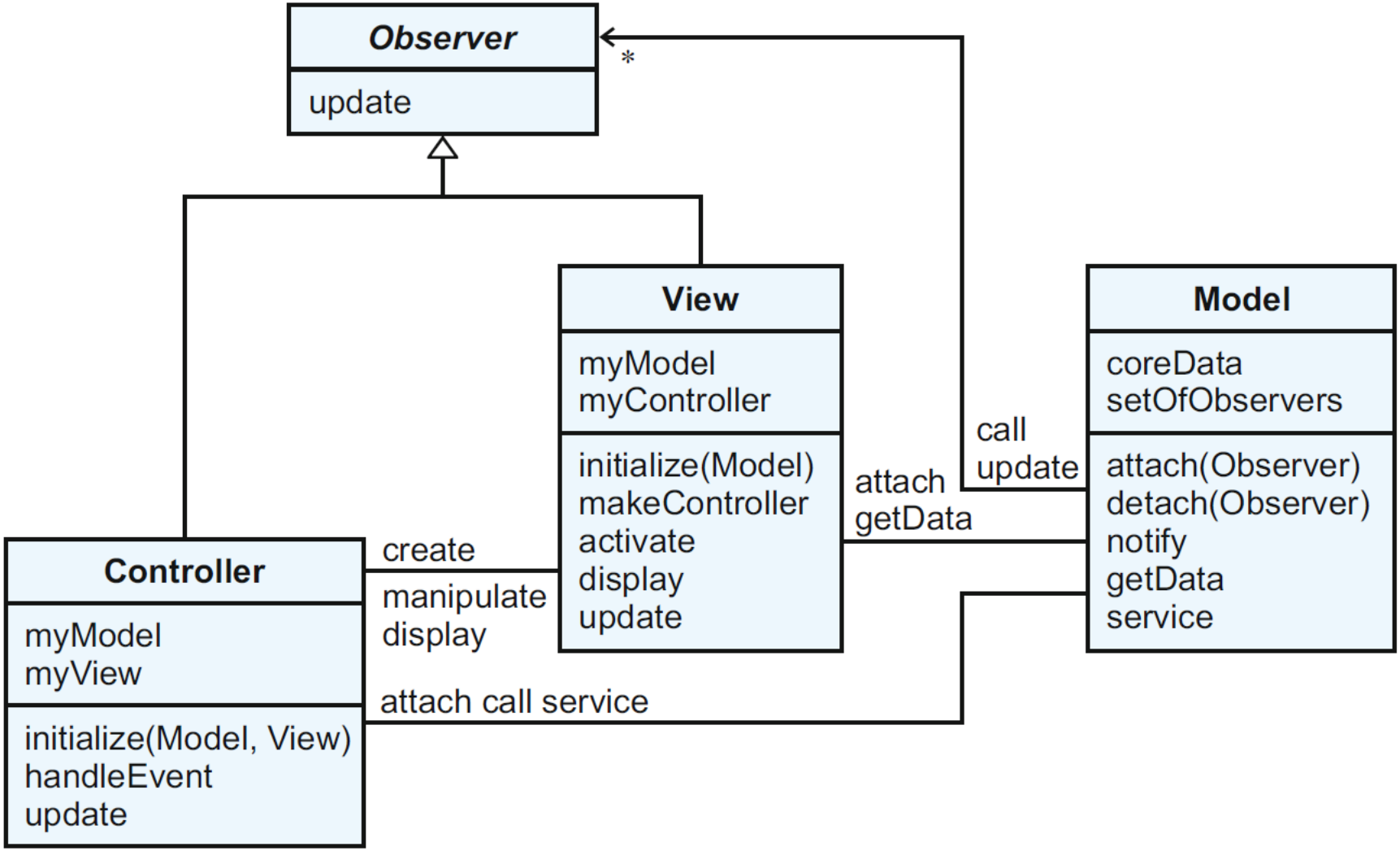


Quelle: Balzert

Das Model – View – Controller – Muster (MVC-Muster)

- Verfeinerung des Beobachter Musters
  - Benachrichtigung an Clients (und Controller) bei Änderung der Modell-Daten
  - Controller modifiziert das Modell
  - Entwurfs-, aber auch Architektur-Muster
  - Modell-Ebene auf Persistenz-Schicht häufig durch eine DB abgebildet
- Modell
  - Subsystem für die Logik des Systems
  - konsistente Modelldaten, die von den View-Objekten gelesen und durch den Controller gesteuert werden
- View
  - Subsystem für die GUI
  - lesender Zugriff auf die Modelldaten
- Controller
  - Subsystem für die Steuerung
  - Abbildung des Ereignis-Konzeptes
  - schreibender und lesender Zugriff auf die Modelldaten
- Grundlagen von vielen Web-Anwendungen mit der Aufteilung in 3 Subsysteme
- Bei grafischen Oberflächen sind View und Controller zusammengefasst

Das Model – View – Controller – Muster (MVC-Muster)

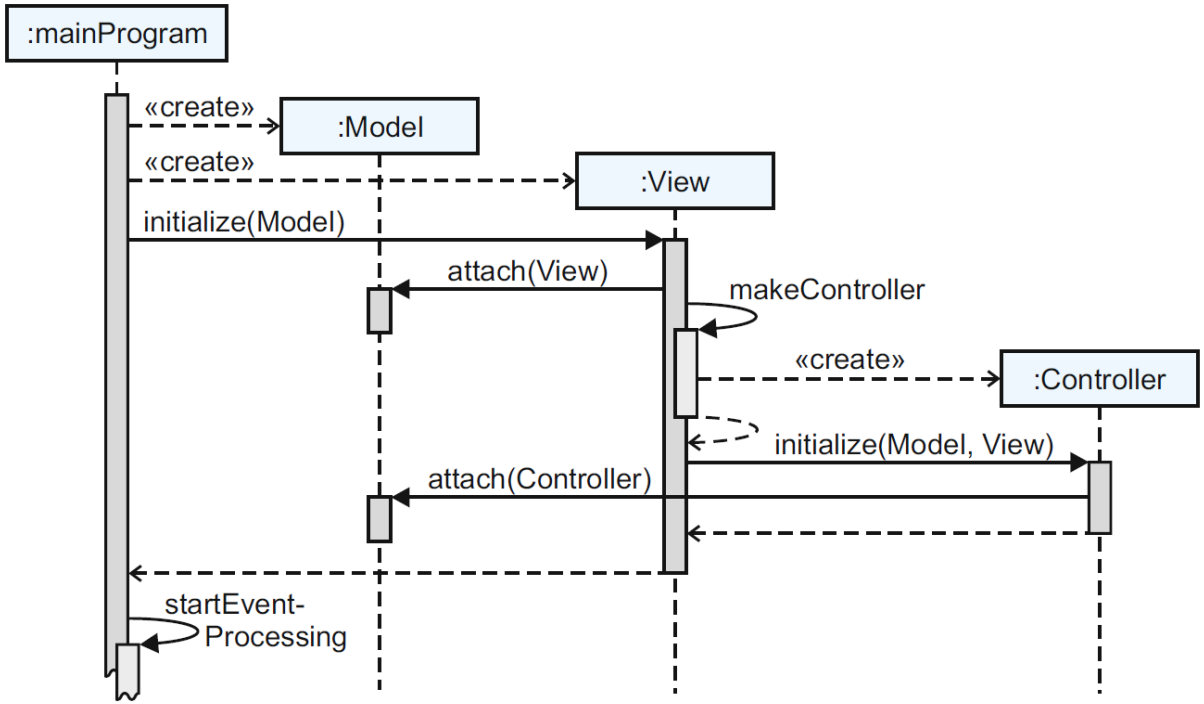


Quelle: Balzert

Das Model – View – Controller – Muster (MVC-Muster)

- Ein model-Objekt wird erzeugt und initialisiert seine internen Datenstrukturen.
- Ein view-Objekt wird erzeugt und erhält als Parameter bei der Initialisierung eine Referenz auf das model-Objekt. Das view-Objekt registriert sich bei dem model-Objekt.
- Das view-Objekt erzeugt sein controller-Objekt und übergibt Referenzen auf das model-Objekt und sich selbst.
- Das controller-Objekt registriert sich beim model-Objekt.

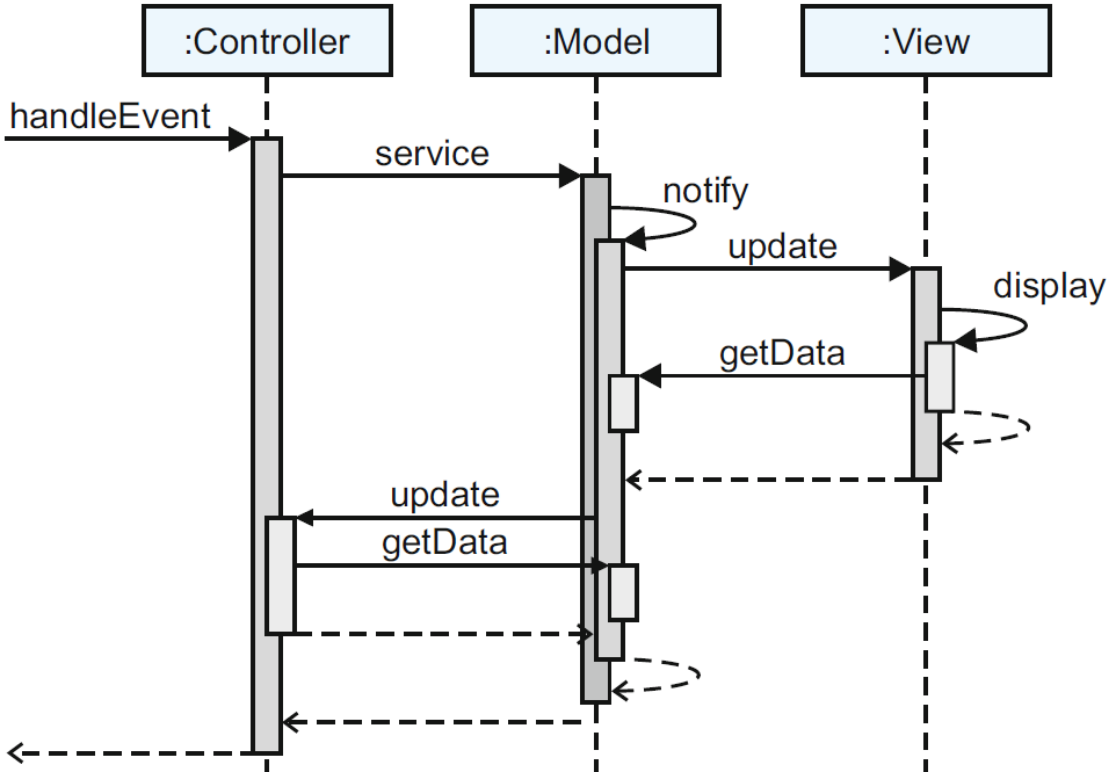
- .



Quelle: Balzert

Das Model – View – Controller – Muster (MVC-Muster)

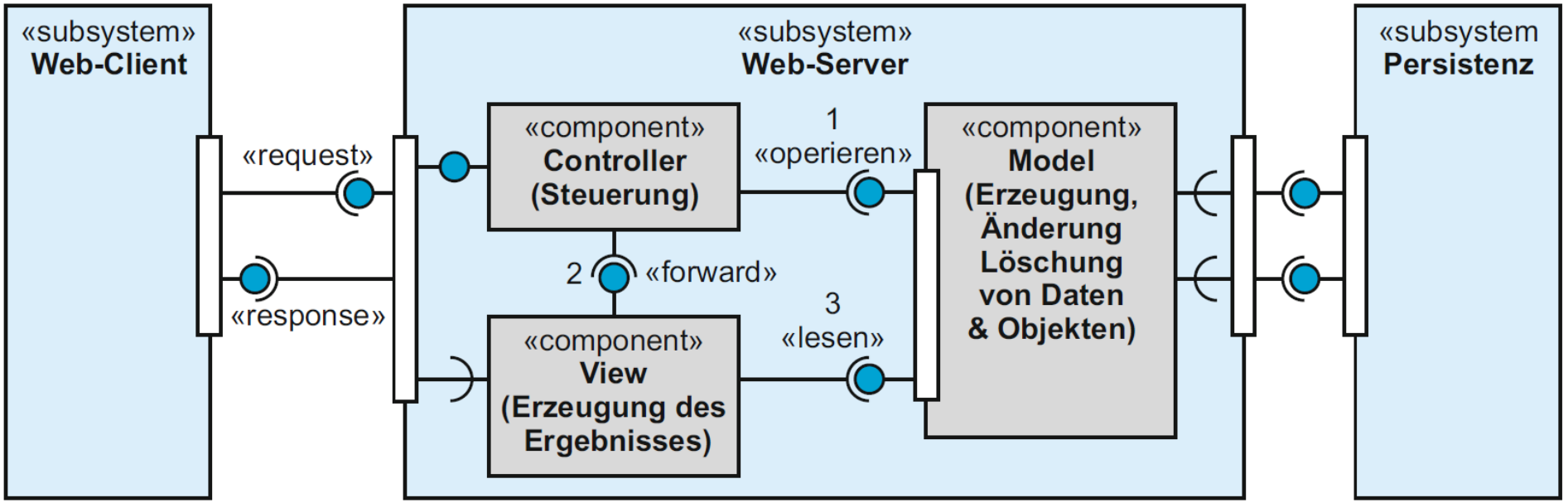
- Der Controller verarbeitet ein Event und schickt seine Meldung an das Modell (service)
- Hier wird intern das notify angestoßen, wodurch das View-Objekt aktualisiert wird (display) und sich dabei aktualisierte Daten aus dem Modell holt (getData)
- Gleiches geschieht mit dem Update vom Controller, der ebenfalls den aktuellen Status vom Modell abfragt.
- An den Observer-Klassen interessiert nur das Update mit festgelegter Signatur. Bei der Modell-Klasse das Registrieren/Unregistrieren der Observer, das Empfangen der Aktualisierung (service) und das Holen der Daten



Quelle: Balzert

Verwendung des MVC-Musters

- Wo bietet sich ein MVC-Muster an?
- Warum eignet sich das MVC-Muster für grafische Oberflächen oder für Web-Anwendungen?
- Wie funktioniert der abgebildete Web-Server? Erklären Sie die dargestellte Architektur.
- Welche Web-Anwendungen basieren auf einer derartigen Architektur?



Quelle: Balzert

Das Fassaden – Muster

- Grundidee
  - Bereitstellung eines exklusiven und/oder zusätzlichen, vereinfachten Zugriffs auf ein komplexes Subsystem oder auf eine Menge zusammengehöriger Objekte
- Der Einsatz des Fassaden-Musters ist in folgenden Fällen sinnvoll:
  - Es soll eine einfache, häufig benötigte Schnittstelle zu einem komplexen Subsystem angeboten werden. Nur Nutzer, die spezielle Methoden benötigen, müssen oder dürfen hinter die Fassade schauen (Grey Box)
  - Bei Verwendung des Schichten-Musters (siehe »Das Schichten-Muster kann durch eine Fassade die Schnittstelle zu jeder Subsystemschicht definiert werden. Dadurch können die Abhängigkeiten zwischen den Subsystemen vereinfacht werden, wenn sie ausschließlich über ihre Fassaden miteinander kommunizieren (Black Box).
  - Fassaden können auch weitere Aufgaben übernehmen, zum Beispiel die Zugangskontrolle, das Management von Transaktionen oder die Verwaltung von Zustandsinformationen für Clients oder Server

Das Fassaden – Muster

- Umsetzung
  - Eine Fassade ist eine Klasse mit ausgewählten Methoden, die eine häufig benötigte Untermenge an Methoden des Subsystems umfasst
  - Dadurch wird der Umgang mit dem Subsystem für Standardfälle vereinfacht
  - Das Subsystem bleibt trotzdem ein White Box-Subsystem, da auch weiterhin auf alle Methoden des Subsystems direkt zugegriffen werden kann
  - das Subsystem kann auch zu einem Black Box-Subsystem oder einem Grey Box-Subsystem gemacht werden
- Vorteile
  - Das Fassaden-Muster fördert die lose Kopplung, weil mehrere, häufig benötigte Methoden in einer Fassaden-Klasse zusammengefasst werden, und dadurch die innere Struktur für den Zugriff nicht bekannt sein muss.
  - Durch die lose Kopplung kann das Subsystem leichter erweitert werden.
  - Die Anzahl der Objekte, die vom Nutzer gehandhabt werden müssen, werden reduziert.
  - Das Subsystem kann durch die Fassade leichter benutzt werden.
- Nachteil
  - Werden während der Entwicklung häufig interne Schnittstellen geändert, dann muss die Fassade häufig angepasst werden.

Das Fassaden – Muster

- Eine Applikation soll Personendaten verwalten, die in einer Datenbank abgelegt werden sollen
- Die Anwendungsfälle dazu sind: Suchen anhand verschiedener Kriterien, Hinzufügen von Personen, Löschen von Einträgen
- Für den Zugriff auf die Datenbank muss diese zunächst geöffnet werden
- Wie könnte eine Fassadenklasse für das Handling der Datenbankzugriffe aussehen?
- Welche Klassen müssen der Fassade bekannt sein?

- Zeichnen Sie ein UML-Diagramm, mit dem Sie die Fassade vollständig beschreiben
  - welche Methoden besitzt die Fassade
  - welche Rückgabetypen und Übergabeparameter sind erforderlich
  - wie soll das Öffnen und Schließen der Datenbank gehandhabt werden
  - Der Ablauf der Methoden soll durch kurze Aktivitätendiagramme dargestellt werden, der direkte Zugriff auf die Datenbank nur verbal beschrieben sein

Das Singleton – Muster

- Häufig wird nur ein Objekt einer Fassaden-Klasse benötigt. Ein Muster, das nur genau ein Objekt einer Klasse bereitstellt, ist das Singleton-Muster
- Das Singleton-Muster kann in verschiedener Weise implementiert werden. Dabei ist insbesondere in Multi-Threaded-Umgebungen die Umsetzung thread-sicher zu gestalten
- Die dargestellte Singleton-Implementierung zeigt:
  - Einen privaten Konstruktor, damit kein Erzeugen eine Objektes außerhalb der Klasse möglich ist
  - Ein privates statisches Member-Objekt, das das einzig vorhandene Objekt darstellt
  - Eine öffentliche, statische Eigenschaft, die einen lediglich lesenden Zugriff auf das Objekt ermöglicht

```
#region singleton
private users()
{
}
static private users instance;
static public users Instance
{
    get
    {
        if (instance == null)
        {
            instance = new users();
        }
        return instance;
    }
}
```

Das Kommando – Muster

- Es entkoppelt Sender (Auslöser, invoker) vom Empfänger (receiver).
- Ein Sender ist ein Objekt, das ein Kommando bzw. einen Befehl erteilt bzw. aufruft (Methodenaufruf), damit eine Anforderung ausgeführt wird. Ein Empfänger ist ein Objekt, das weiß, wie die Anforderung auszuführen ist.
- Entkopplung bedeutet, dass der Sender kein Wissen über die Schnittstelle des Empfängers hat. Das Kommando bzw. der Befehl wird als ein Objekt gekapselt. Dadurch ist es möglich zu variieren, wann und wie eine Anforderung erfüllt werden soll. Anforderungen können dadurch rückgängig gemacht oder wieder ausgeführt werden.
- Anwendungsbereich(e)
  - Der Einsatz des Kommando-Musters ist in folgenden Fällen sinnvoll: Strukturierung eines Systems in abstrakte Operationen, die auf einfachen Operationen basieren.
  - Entkopplung des Objekts, das eine Aktion aufruft, von dem Objekt, das die Aktion ausführt. Wird dieses Muster für diesen Zweck eingesetzt, dann wird es auch als Produzenten-Konsumenten-Muster bezeichnet.
  - Anforderungen sollen nicht sofort, sondern zeitlich verzögert oder asynchron ausgeführt werden. Kommandos können beispielsweise in einer Warteschlange eingeordnet werden
- Das Beobachter Muster in der Delegate-Umsetzung könnte in weitestem Sinne auch als Kommando-Muster verstanden werden

Das Kommando – Muster

- Folgende Klasse bietet zwei Kommandos an, die fernsteuerbar sein sollen

```
//Die zu steuernde Klasse
class Beleuchtung
{
    public void anschalten()
    {
        Console.WriteLine("Beleuchtung wird angeschaltet");
    }
    public void ausschalten()
    {
        Console.WriteLine("Beleuchtung wird ausgeschaltet");
    }
}
```

Quelle: nach Balzert

Das Kommando – Muster

- Mit einer allgemeinen Schnittstelle kann eine konkrete Kommandoklasse implementiert werden:

```
//Die Schnittstelle Command spezifiziert eine abstrakte Methode execute():
//Schnittstelle Command
public interface Command
{
    public void execute();
}

//Konkretes Kommando
class BeleuchtungAnKommando : Command
{
    private Beleuchtung meineBeleuchtung;
    public BeleuchtungAnKommando(Beleuchtung meineBeleuchtung)
    {
        this.meineBeleuchtung = meineBeleuchtung;
    }
    public void execute()
    {
        meineBeleuchtung.anschalten();
    }
}
```

Quelle: nach Balzert

Das Kommando – Muster

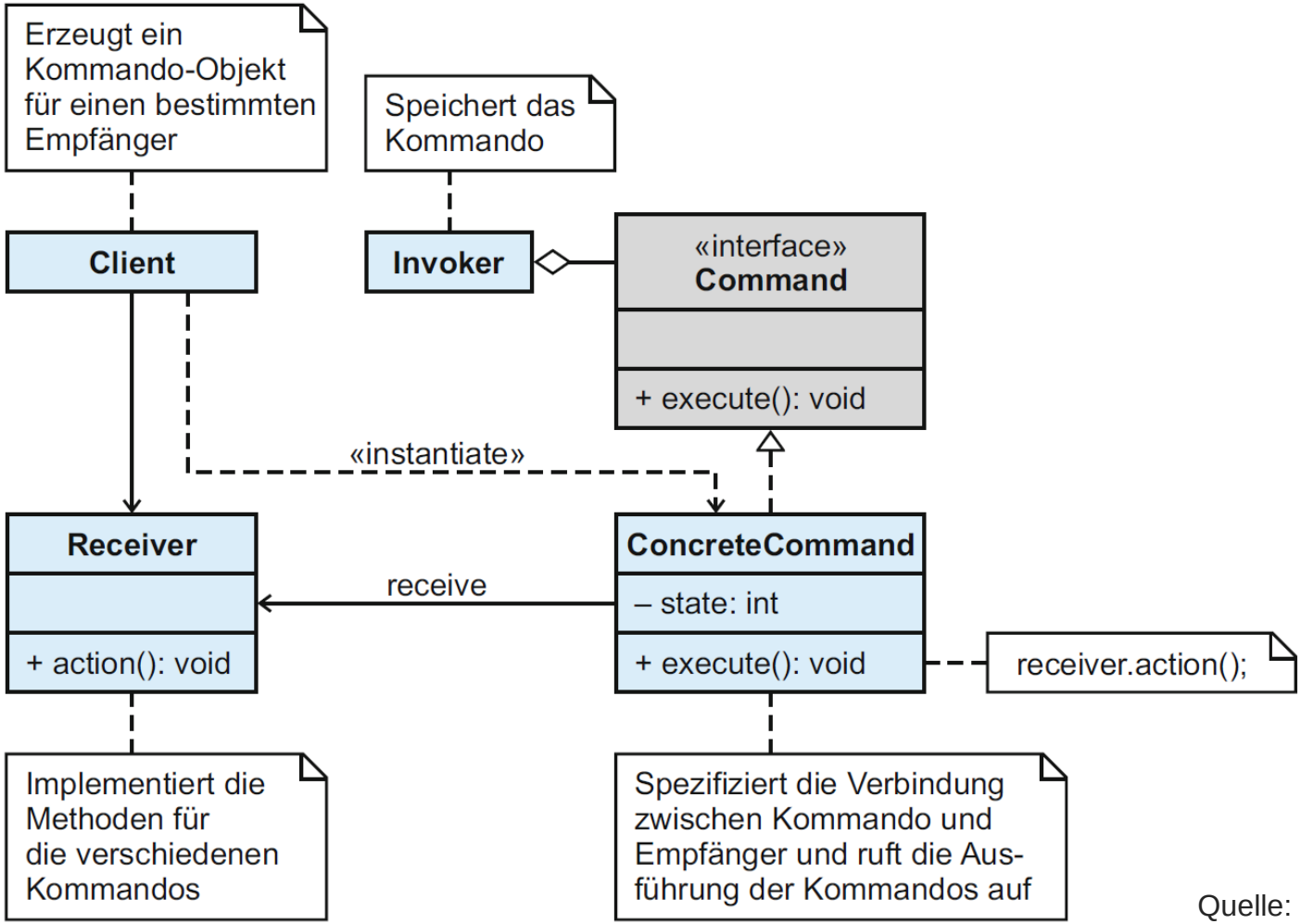
- In einer Invoker-Klasse werden nur die Command-Objekte benutzt

```
//Aufrufer des Kommandos (Invoker)
class Fernbedienung
{
    private Command einKommando, ausKommando;
    public Fernbedienung(Command einKommando, Command ausKommando)
    {
        //Ein konkretes Kommando registriert sich
        //selbst beim Aufrufer
        this.einKommando = einKommando;
        this.ausKommando = ausKommando;
    }
    void ein()
    {
        //Der Aufrufer ruft das konkrete Kommando auf (call back),
        //das das Kommando beim Empfänger ausführt
        einKommando.execute();
    }
    void aus()
    {
        ausKommando.execute();
    }
}
```

Quelle: nach Balzert

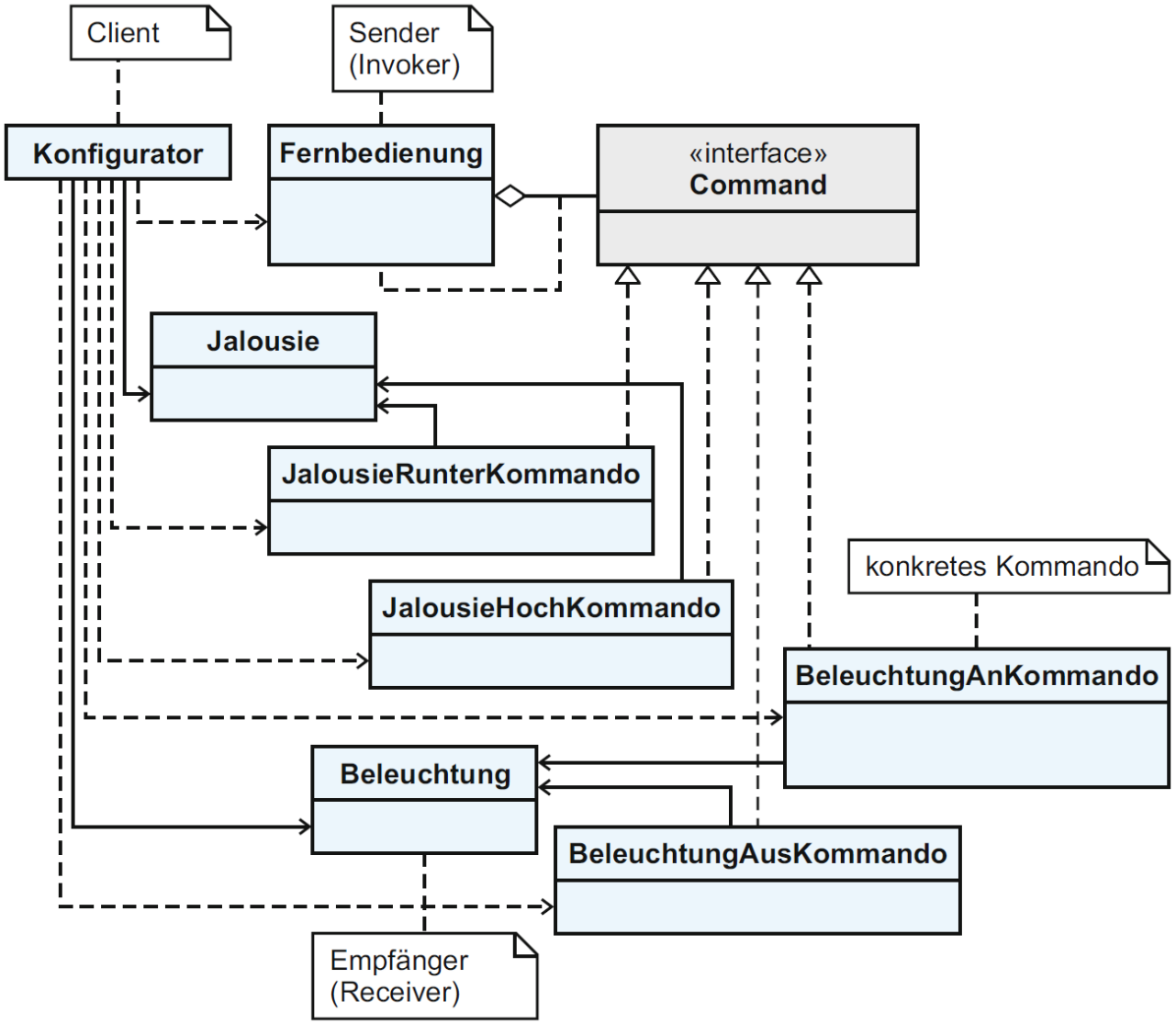
Das Kommando – Muster

- Beim Kommando-Muster erzeugt ein Client eine Instanz einer Kommando-Klasse, das wiederum die Ausführung seiner Aktion an eine Receiver-Klasse delegiert (aber auch selbst implementieren kann). Der Invoker ruft das Kommando auf



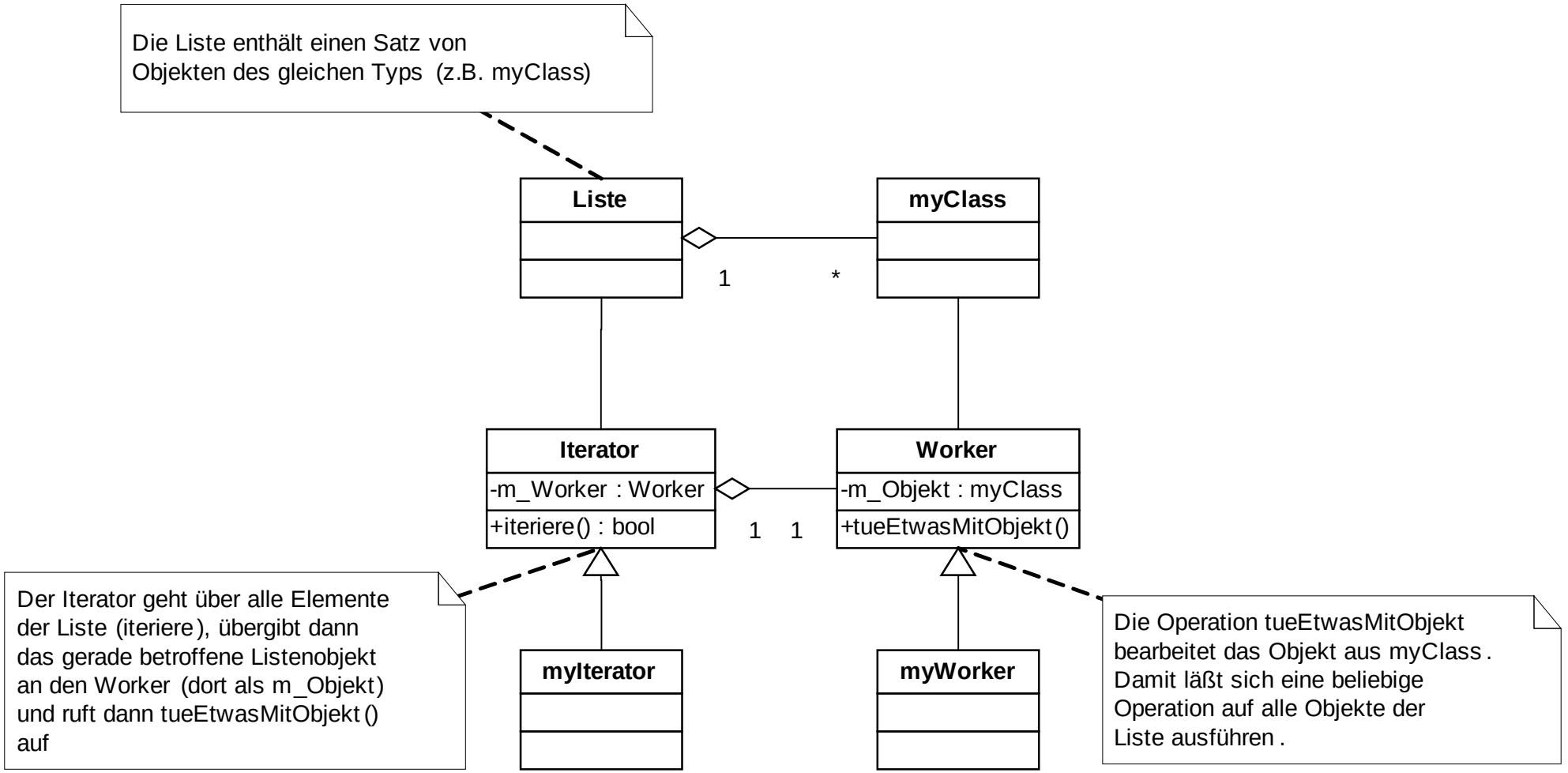
Quelle: Balzert

Das Kommando – Muster



Quelle: Balzert

Das Iterator – Worker – Muster



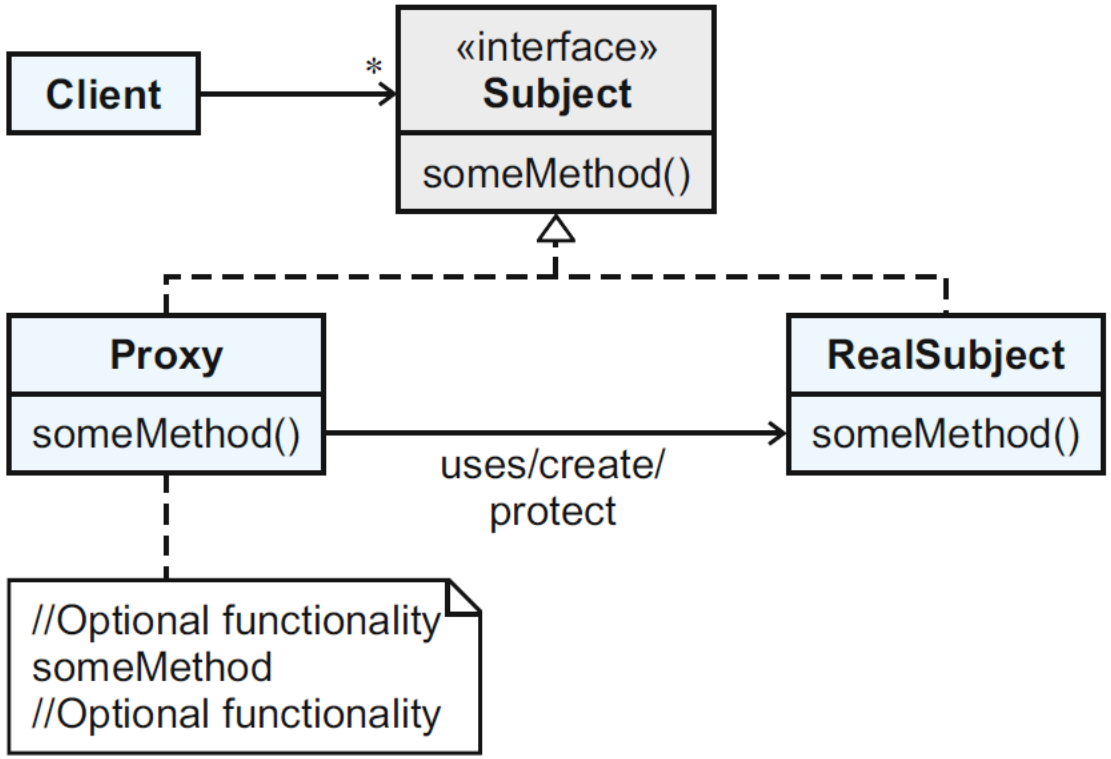
Das Iterator – Worker – Muster

- Der Iterator durchläuft eine beliebige Menge von Objekten und sucht bei entsprechend vorliegenden Suchmustern zu bearbeitende Elemente
  - Ob die Menge der Objekte in einer Datenbank, in Verzeichnissen, in einer Datei selbst wie bspw. ein Suchmuster in einer Bitmap vorliegen beeinflußt nur den Typ des Iterators

- Der Iterator filtert aus der Menge die gesuchten Objekte heraus und übergibt diese einem Worker, dieser arbeitet ähnlich wie das Kommando-Muster
- Durch polymorphes Verhalten sind Iterator und Worker in unterschiedlichen Varianten kombinierbar, solange gleichartige Objekte bearbeitet werden
- Ein häufiger Anwendungsfall ist die Verwendung eines Iterators, der flexibel hinsichtlich der Suchkriterien konfigurierbar ist und mit verschiedenen Worker-Objekten je nach erforderlicher Umsetzung kombiniert werden kann
- Einsatzgebiet dieses Musters ist häufig die Massendatenverarbeitung

Das Proxy – Muster

- Die Grundidee des Proxy – Musters ist die Kontrolle des Zugriffs auf andere Objekte
- Diese Kontrolle kann verschiedenartig ausgeprägt sein:
  - Schutzmechanismus vor unberechtigtem Zugriff,
  - Caching-Werkzeug zum effizienten Zugriff auf entfernte Objekte
  - virtueller Proxy zur Maskierung von Objekten dienen, die erst zu bestimmter Zeit konkret vorhanden sein sollen (z.B. Bild auf Webpage in unsichtbarem Bereich)
  - Synchronisation bei asynchronen Zugriffen
  - …



Quelle: Balzert

Das Proxy – Muster

- Für jede Klasse, die der Kontrolle eines Proxy unterliegen soll, muss ein jeweils der Klasse entsprechender Proxy implementiert werden. Dies ist bei größeren Systemen durchaus aufwendig
- Mit Hilfe von dynamischen Proxies, die zur Laufzeit erstellt werden, kann der Zugriff auf Objekte ohne konkreten Bezug zur zu überwachenden Klasse kontrolliert werden, indem Ereignisse bei Start und Ende eines Methodenaufrufs das Eingreifen in den Programmablauf erlauben
- Der Proxy delegiert Aufrufe an ein Objekt der überwachten Klasse. Damit dieser Mechanismus nicht umgangen werden kann, muss die überwachte Klasse als nur innerhalb des Namensraums zu verwenden deklariert werden (internal), der Proxy bietet die einzige Schnittstelle nach draußen, wobei der Client nicht in der gleichen Assembly liegen darf, um eine Erzeugung am Proxy vorbei nicht zu ermöglichen.
- In der Regel ist der Proxy auch für das Erzeugen und Löschen der zu überwachenden Objekte verantwortlich, zur Vereinfachung erhält er die gleiche Schnittstelle, die Aufgaben an das zu überwachende Objekt delegiert, dabei aber in Managementaufgaben einbettet

Das Proxy – Muster

- Abwandlungen des Proxy – Musters zeigt folgende Tabelle

Proxy	Dekorator	Adapter
gleiche Schnittstelle	gleiche Schnittstelle	unterschiedliche Schnittstelle
kontrolliert Zugriff	zusätzliche / veränderte Funktion	ähnlich Schutz-Proxy
»Implementierungs-Detail«	»konzeptionelle Eigenschaft«	

- Der Nachteil von Proxies liegt im erhöhten Entwicklungsaufwand, da jede Erweiterung der zu überwachenden Klasse eine Erweiterung im Proxy nach sich zieht, zumindest, wenn diese neue Funktionalität nach außen verfügbar gemacht werden soll
- Dies gilt auch für die Varianten Dekorator oder Adapter

Quelle: Balzert

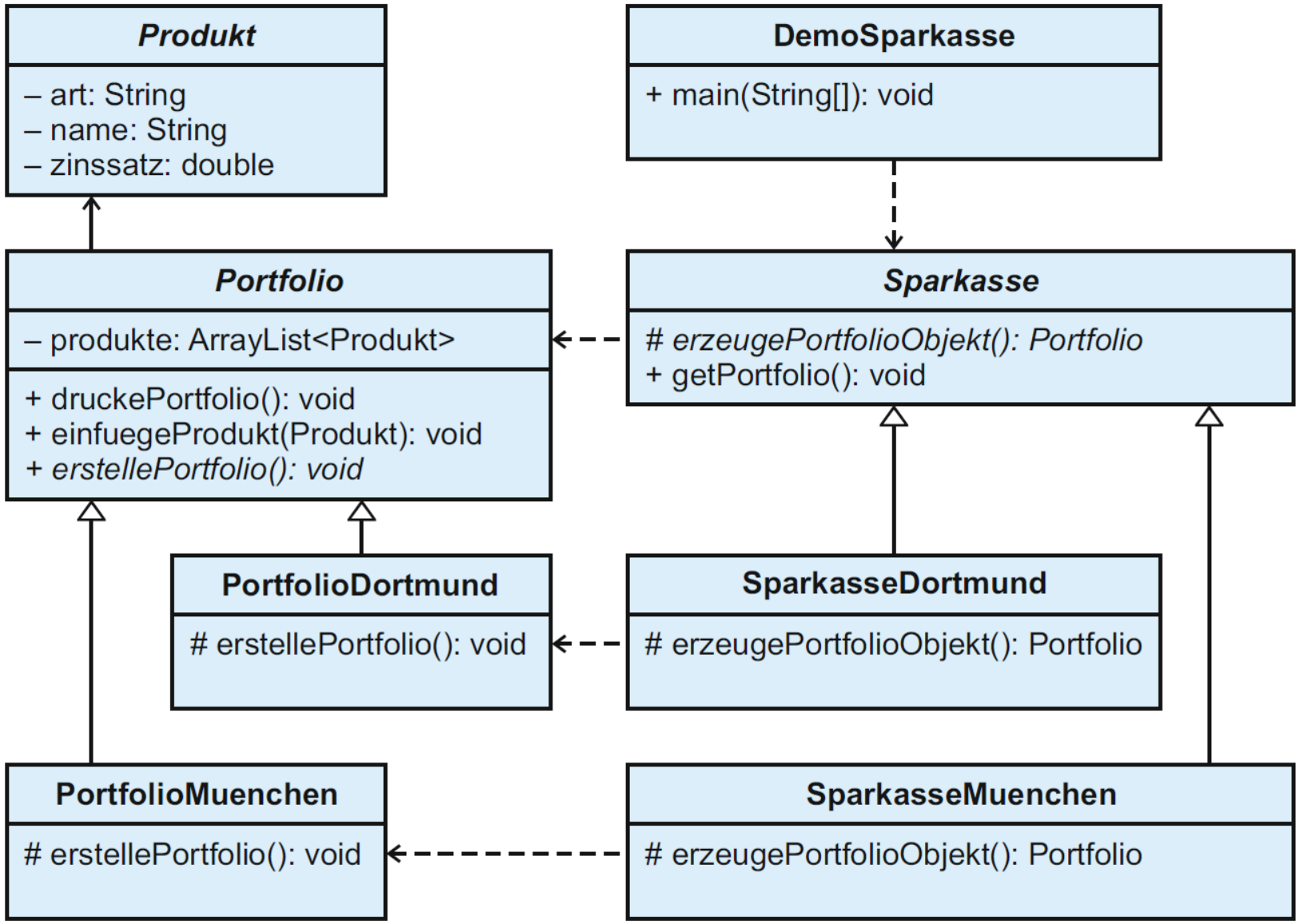
Das Fabrikmethoden – Muster

- Das Fabrikmethoden-Muster (factory method pattern) ist ein klassenbasiertes Erzeugungsmuster. Es ist auch als virtueller Konstruktor (virtual constructor) bekannt.
- In der Regel wird durch den Aufruf eines Konstruktors einer Klasse ein neues Objekt erzeugt. Das hat verschiedene Nachteile:
  - Der Name des Konstruktors ist identisch mit dem Klassennamen.
  - Der problembezogene Kontext ist dadurch oft nicht sichtbar. Beispielsweise kann eine Klasse Punkt nicht zwei beschreibende Erzeugungs-Methoden vonKartesischenKoordinaten(double real, double im) und vonPolarkoordinaten(double real, double im) bereitstellen, sondern nur den Konstruktur Punkt(double real, double im).
  - Durch den Aufruf eines Konstruktors wird immer ein neues Objekt erzeugt. Ist bereits ein Objekt vorhanden, dann kann es nicht wieder verwendet werden.
- Fabrikmethoden kapseln die Erzeugung von Objekten. Ist ein geeignetes Objekt bereits vorhanden, dann kann es wieder verwendet werden. Der Name der Erzeugungsmethode kann frei gewählt werden. Dadurch werden die beiden genannten Nachteile vermieden.
- Das Singleton – Muster kann in weitem Sinne auch als Fabrikmethoden – Muster gesehen werden
- Das Fabrikmethoden – Muster kann die Erzeugung der Objekte an Unterklassen delegieren, die Fabrikmethode stellt damit nur eine Schnittstelle bereit.

Das Fabrikmethoden – Muster

- Mit dem Fabrikmethoden – Muster kann auch festgelegt werden, wie etwas zu erzeugen bzw. herzustellen ist. So können für unterschiedliche Aufgaben Objekte unterschiedlicher abgeleiteter Klassen erzeugt werden, ohne daß das äußere Behandeln der unterschiedlichen Objekte transparent wird, da die erzeugten Objekte nur als Basisklassen zurückgegeben werden
- Fabrikmethoden sind i.d.R. statische Methoden, können aber auch im Falle von Strukturen objektbezogen an den Container-Klassen implementiert werden
- Der Einsatz des Fabrikmethoden-Musters ist in folgenden Fällen sinnvoll:
  - Wenn eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll, oder wenn Unterklassen bestimmen sollen, welche Objekte erzeugt werden.
  - Wenn die Erzeugungsmethode einen beschreibenden Namen besitzen soll, anstelle des Klassennamens. Es kann mehrere Fabrikmethoden mit unterschiedlichen Namen und unterschiedlicher Semantik geben.
  - Wenn die Erzeugungsmethode die Erzeugung von Objekten kapseln soll. Dies kann sinnvoll sein, wenn der Erzeugungsprozess sehr komplex ist, zum Beispiel wenn er von Konfigurationsdateien oder von Benutzereingaben abhängig ist. Dieser Weg wird z.B. bei den COM-Objektbibliotheken beschritten. Eine Arbeitsmappe in Excel wird nicht über einen Konstruktor erzeugt, sondern über eine Add-Methode am darüber liegenden Workbooks-Container. Add ist hier also eine Fabrikmethode

Das Fabrikmethoden – Muster



Quelle: Balzert

Weitere Muster

- Aufgrund der Vielzahl unterschiedlicher Handlungssituationen werden in der Literatur weitere Entwurfsmuster vorgestellt wie z.B.
  - Das Fabrikmethoden-Muster zur Erzeugung von Objekten
  - Das Brücken-Muster zur abstrakten Schnittstellendefinition, um Änderungen an Schnittstellen zu minimieren
  - Das Strategie-Muster, um zur Laufzeit zwischen verschiedenen Implementierungen für gleiche Probleme aufgrund von Randbedingungen unterscheiden zu können

- Entwurfsmuster sind
  - Wiedererkennbar
  - Für unterschiedlichste Anforderungen verfügbar
  - Eine gute Voraussetzung für strukturierte, objektorientierter Implementierungen

- Häufig weisen gute Programme auch ohne Kenntnis von Entwurfsmustern deren typische Strukturen auf