# Patterns

by

Dr. Günter Kolousek

# Muster (engl. pattern)

- erprobte Lösung in strukturierter Form für ein Problem
  - Lösungsschablone
  - hat einen Namen: $\leadsto$ Kommunikation unter Entwicklern
- Architekt Christopher Alexander
  - $\leadsto$ Häuser (1970 ff.)
- Kent Beck und Ward Cunningham (ca. 1987)
  - Verwendung für GUI in Smalltalk
- James Coplien
  - Advanced C++ Idioms (1991)
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Design Patterns - Elements of Reusable Object-Oriented Software (1994)
  - Gang of Four
- Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann
  - Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects (2000)

# Arten

- Idioms
  - Spezifisch für eine Programmiersprache
  - ”niedrige” Abstraktionsebene
- (Entwurfs)richtlinien und Heuristiken
  - Heuristik: Kunst mit begrenztem Wissen und wenig Zeit zu einer guten Lösung zu kommen
- Entwurfsmuster (engl. design patterns)
  - ”mittlere” Abstraktionsebene

# Arten – 2

- Architekturmuster
  - Organisation und Interaktion zwischen Komponenten
  - z.B. ”Schichtenarchitektur”
  - ”hohe” Abstraktionsebene
- Analysemuster
  - für bestimmte Domäne, z.B. ”Account”
- Business pattern
  - für einen bestimmten Geschäftsbereich
- Anti-Pattern

# Aufbau eines (Entwurfs)Musters

- Pattern Name, Klassifizierung, andere Namen
- Zweck
- Szenario
- Problem/Kontext
- Lösung
- Vorteile
- Nachteile
- Verwendung
- Varianten
- Verweise

# Einteilung gemäß GoF

- Creational patterns
  - z.B. Factory Method
- Structural patterns
  - z.B. Composite
- Behavioral patterns
  - z.B. State, Iterator, Observer
- Aber… noch viele andere Patterns!

# Creational Patterns

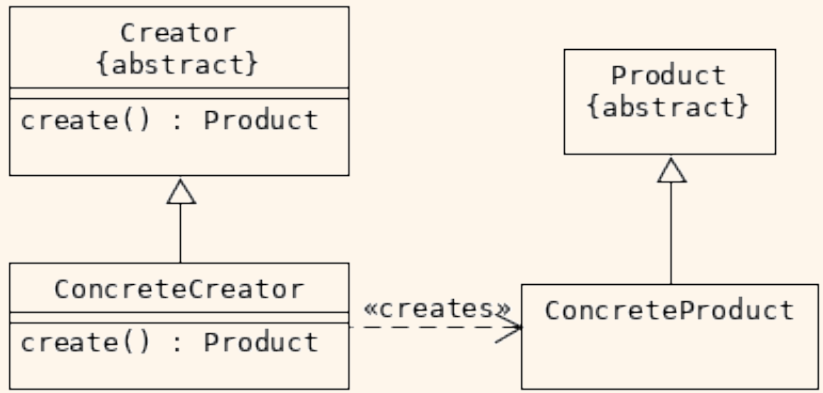
- Factories
  - Factory Method
  - Abstract Factory
- Builder
- Prototype
- Singleton

# Factory Method

**Zweck** Definiere eine Schnittstelle zum Erzeugen von Objekten ohne die konkreten Klassen zu spezifizieren.

**Prob/Kont.** Es soll ein Objekt erzeugt werden, aber die konkrete Klasse ist noch nicht bekannt und wird i.d.R. erst in einer Subklasse festgelegt.

**Lösung**




**Varianten**

- Fabriksmethoden mit Parameter
- Eine Creator-Klasse und viele Produktklassen

# Factory Method – 2

```
#include <iostream>
using namespace std;

struct Button {
    virtual void paint()=0;  };
struct LinuxButton : public Button {
    void paint() {
        cout << ”Linux button” << endl;  } };

struct ButtonCreator {
    virtual Button* create()=0; };
struct LinuxButtonCreator : public ButtonCreator {
    Button* create() {
        return new LinuxButton(); } };

int main() {
    //Button* btn{new LinuxButton()};  // -> hard coded!
    ButtonCreator* factory{new LinuxButtonCreator()};
    Button* btn{factory->create()};
    btn->paint(); }
```

# Factory Method – 3

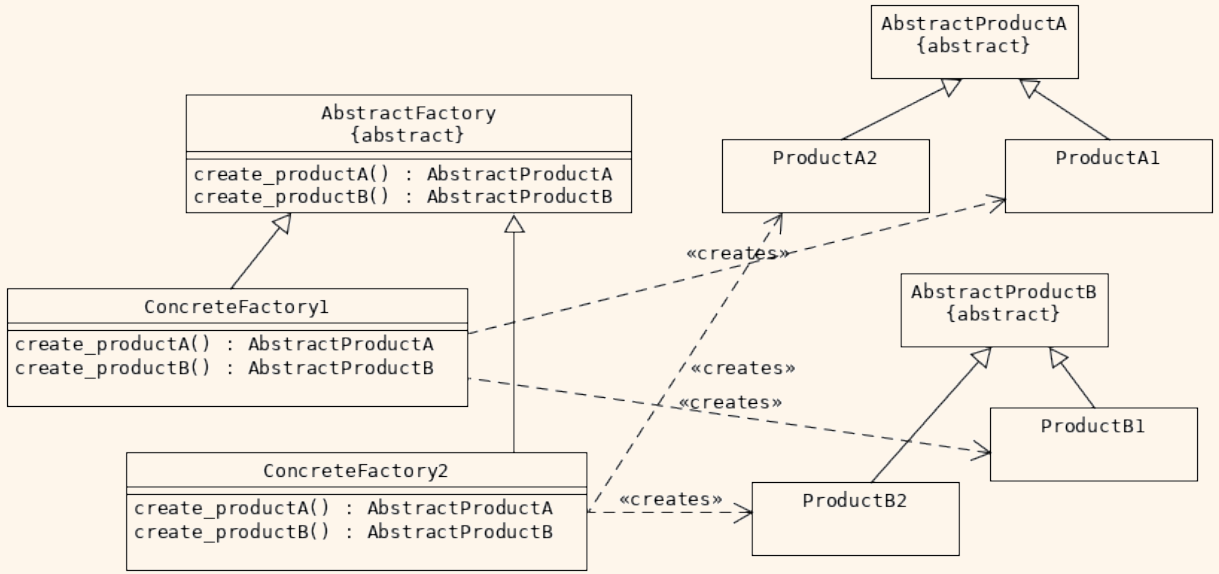
```
#include <iostream>
#include <cmath>
using namespace std;
// alternative Verwendung von ”factory method”!!!
struct Point {
    float x;
    float y;
    // Point(float x, float y) {}
    // Point(float r, float alpha) {}
  protected:
    Point(float x, float y) : x{x}, y{y} {}
  public:
    static Point create_cartesian(float x, float y) {
        return { x, y };  }
    static Point create_polar(float r, float alpha) {
        return { r * cos(alpha), r * sin(alpha) };
    } };

int main() {
    Point p{Point::create_polar(1, atan(1))};
    cout << p.x << ' ' << p.y << endl;  }
```

# Abstract Factory

**Zweck** Definiere eine Schnittstelle zum Erzeugen ganzer Familien von Objekten ohne die konkreten Klassen zu spezifizieren.

**Prob/Kont.** Es sollen ganze Familien von Objekten in einem Zusammenhang erzeugt werden, aber die konkreten Klassen sind noch nicht bekannt.

**Lösung**




**Varianten** Wird nur eine konkrete Fabrik benötigt, kann man auf die *AbstractFactory* verzichten. Dann spricht man von einer Factory (bzw. dem Factory-Pattern).

# Abstract Factory – 2

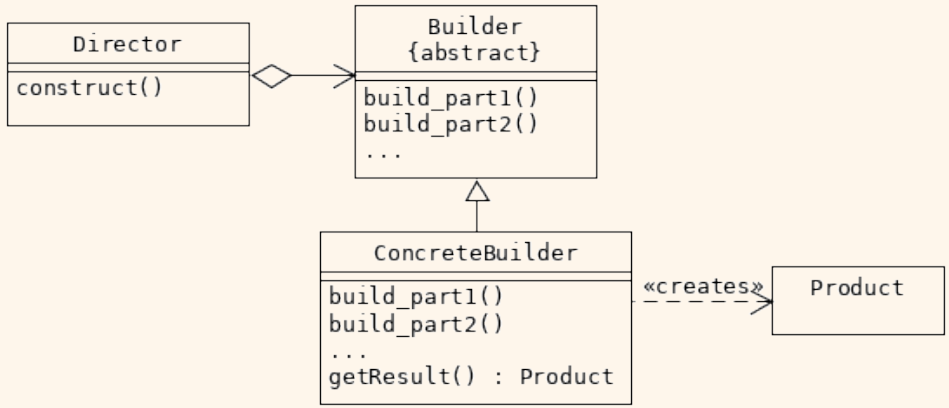
```
#include <iostream>
using namespace std;

struct Button {
    virtual void paint()=0; };
struct LinuxButton : public Button {
    void paint() {
        cout << ”Linux button” << endl;  } };

struct GUIFactory {
    virtual Button* create_button()=0; };
struct LinuxGUIFactory : public GUIFactory {
    Button* create_button() {
        return new LinuxButton();  } };

int main() {
    //Button* btn{new LinuxButton()};  // -> hard coded!
    GUIFactory* factory{new LinuxGUIFactory()};
    Button* btn{factory->create_button()};
    btn->paint();  }
```

# Builder

**Zweck** Erzeugen von komplexen Objekten

**Prob/Kont.** Schrittweises Erzeugen komplexer Objekte, die in unterschiedlichen Repräsentationen vorliegen können

**Lösung**




**Nachteile** Enge Kopplung zwischen Produkt dem konkreten Builder und den am Konstruktionsprozess beteiligten Klassen

# Builder – 2

```
#include <iostream>
#include <vector>
using namespace std;
struct HtmlBuilder;
struct HtmlElement {
    string name;  string text;
    vector<HtmlElement> elements;
    static HtmlBuilder& create(string name);
    string to_string() {
        string res{name + ':' + text};
        if (elements.size() != 0) {
            res += '{';
            for (int i{0}; i < elements.size(); ++i) {
                res += '{';
                res += elements[i].to_string();
                res += '}';
                if (i != elements.size() - 1) res += ", ";
            }
            res += '}';
        }
        return res;  } };
```

# Builder – 3

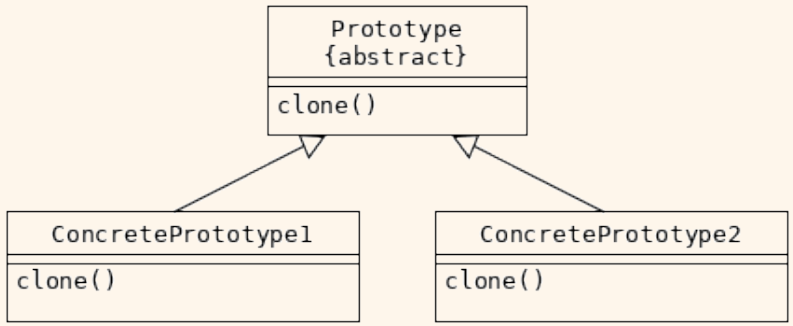
```
struct HtmlBuilder {
    HtmlElement root;
    HtmlBuilder& add_child(string name, string text) {
        HtmlElement elem{name, text};
        root.elements.emplace_back(elem);
        return *this;  }
    HtmlBuilder(string name) {
        root.name = name;  }
    HtmlElement& get_root() { return root; }
};

HtmlBuilder& HtmlElement::create(string name) {
    return *(new HtmlBuilder(name));  }

int main() {
    HtmlElement list{HtmlElement::create("ul")
          .add_child("li", "banana")
          .add_child("li", "apple").
          get_root()};
    cout << list.to_string() << endl;  }
```

# Prototype

**Zweck** Erzeugen von Objekten durch Erstellen einer Kopie eines bestehenden Objektes

**Prob/Kont.** Es werden ähnliche Objekte zu bestehenden Objekten benötigt und das Erzeugen von Grund auf ist zu aufwändig

**Lösung**




**Verwendung** zu beachten ist, dass meist ein *deep copy* vorzunehmen ist, außer es handelt sich um *immutable objects*

# Prototype – 2

```
#include <iostream>
using namespace std;
struct Address {
    string street; string city;
};
struct Contact {
    string name;
    Address* address;
    Contact(string name, Address* address) :
      name{name}, address{address} {}
    Contact(const Contact& other) : name{other.name},
              address{new Address{*other.address}} {}
};
int main() {
    Contact c1 {"Maxi", new Address{"weg 1", "Zwergenstadt"}};
    Contact c2{c1};
    c2.address->street = "weg 42";
    cout << c1.address->street << endl;
    cout << c2.address->street << endl;
}
```

# Singleton

**Zweck** Stellt sicher, dass nur eine Instanz einer Klasse erzeugt wird

**Prob/Kont.** Es ist notwendig, dass von einer Klasse nur maximal eine Instanz vorhanden ist.

**Lösung**

```
#include <iostream>
using namespace std;
class Singleton {
    static Singleton* singleton;
    Singleton() {}
  public:
    static Singleton* get() {
        if (!Singleton::singleton)
             Singleton::singleton = new Singleton();
        return singleton;  } };
Singleton* Singleton::singleton;
int main() {
    Singleton* s{Singleton::get()};
    Singleton* s2{Singleton::get()};
    cout << (s == s2) << endl;  }  // -> 1
```

**Nachteile** thread-sichere Lösung schwierig!

# Singleton – 2

```cpp
#include <iostream>
using namespace std;
struct Singleton {
  protected:
    Singleton() { /*...*/ }
  public:
    static Singleton& get() {
        static Singleton singleton; // thread-safe since C++11
        return singleton;
    }
    Singleton(const Singleton&)=delete;
    Singleton(Singleton&&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    Singleton& operator=(Singleton&)=delete;
};
int main() {
    Singleton& s{Singleton::get()};
    Singleton& s2{Singleton::get()};
    cout << (&s == &s2) << endl;
}
```

# Structural Patterns

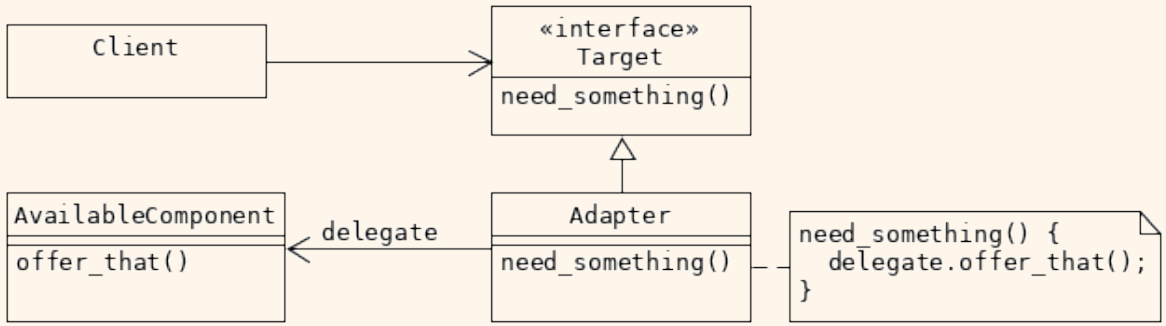
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy
- Flyweight

# Adapter

**Zweck** Anpassen einer Schnittstelle einer Klasse an eine von einem Klienten erwartete Schnittstelle

**Prob/Kont.** Verwenden einer Klasse deren Schnittstelle nicht mit der benötigten Schnittstelle über einstimmt (inkompatibel oder unpassend)

**Lösung**



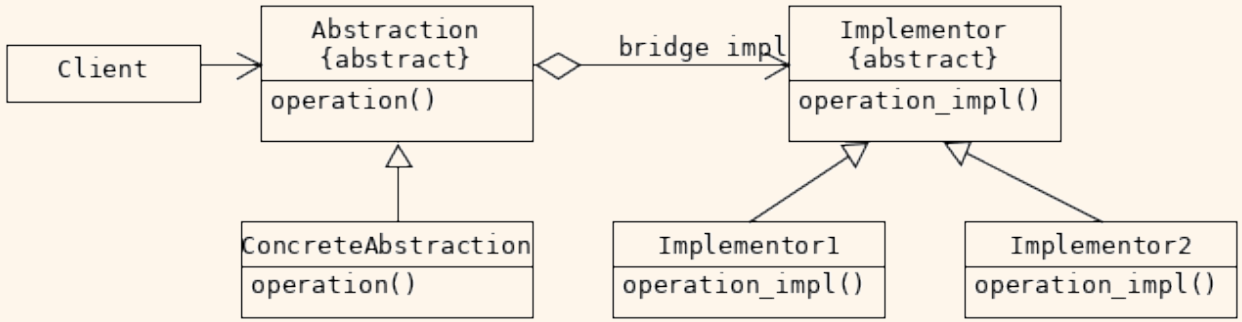

**Nachteile** zusätzliche Indirektion!

# Bridge

**Zweck** Entkoppeln einer Abstraktion von ihrer Implementierung sodass beide unabhängig verändert werden können

**Prob/Kont.** Kapselung zur Erreichung von unabhängigen Änderungen und SoC gefordert

**Lösung**




**Verwendung** Sinnvoll, wenn Abstraktionen mit unterschiedlichen Implementierungen vorliegen

# Bridge – 2

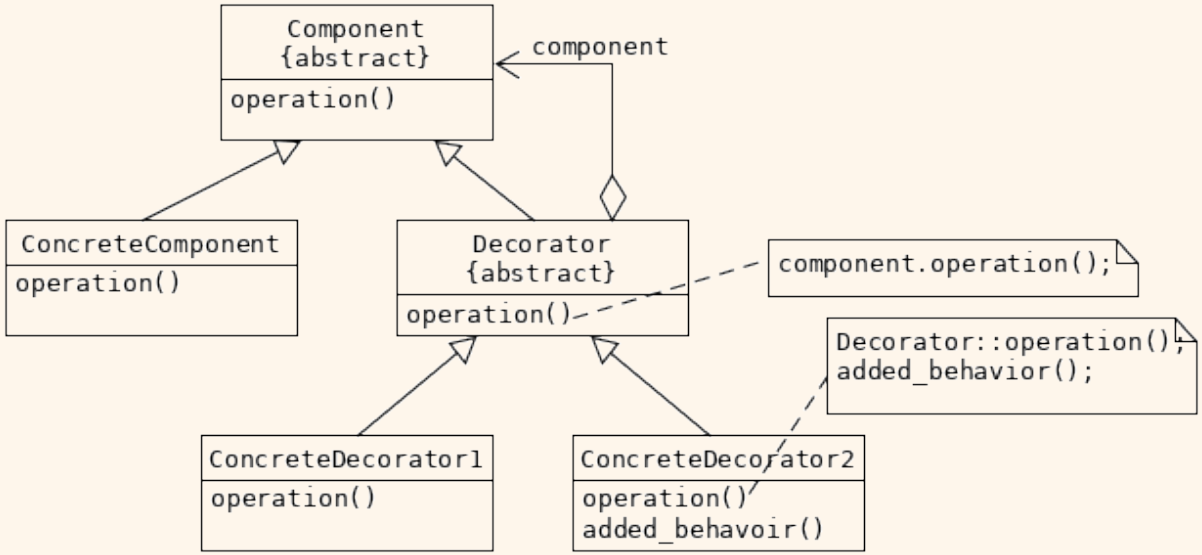
```
#include <iostream>
using namespace std;
struct TransportVehicle { virtual void transport_impl()=0; };
struct RefrigeratorCar : public TransportVehicle {
 void transport_impl() { cout<< ”per refr. car”<< endl; } };
struct TankTruck : public TransportVehicle {
 void transport_impl() { cout << ”per tank truck” << endl; }};

struct Article {
    Article(TransportVehicle* carrier) : carrier{carrier} {}
    TransportVehicle* carrier;
    virtual void transport() { carrier->transport_impl(); } };
struct Fish : public Article {
    Fish(TransportVehicle* carrier) : Article{carrier} {} };
struct Menhir : public Article {
    Menhir(TransportVehicle* carrier) : Article{carrier} {} };

int main() {
    TransportVehicle* carrier{new RefrigeratorCar()};
    Article* fish{new Fish(carrier)};
    fish->transport(); }
```

# Decorator

**Zweck** Fügt einer Komponente neue Funktionalität hinzu

**Prob/Kont.** Funktionalität ist zu ändern ohne Klasse zu verändern

**Lösung**



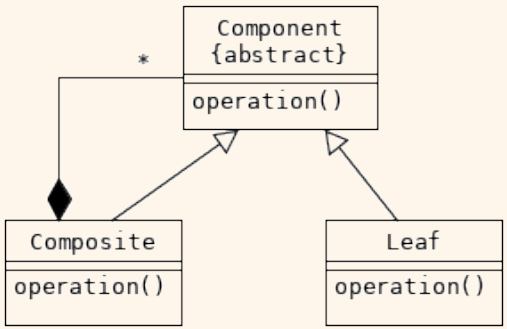

**Vorteile** Verwendet Komposition anstatt Vererbung;
Funktionalität kann zur Laufzeit *verändert* werden

# Composite

**Zweck** Gleichbehandlung von Einzelelementen und Elementgruppierungen in verschachtelter Struktur

**Prob./Kont.** Verschachtelte Struktur, deren Elemente verwaltet/manipuliert werden müssen.

**Lösung**



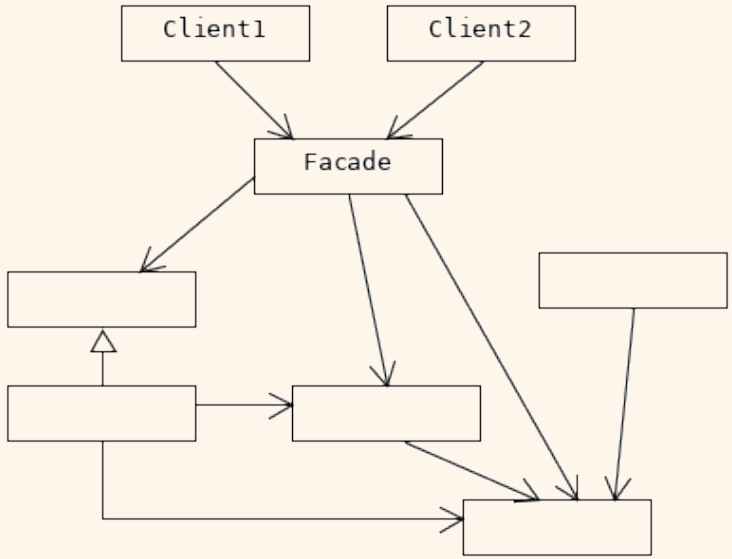

**Verweise** Iterator

# Facade

**Zweck** Vereinfacht Zugriff auf ein komplexes Subsystem

**Prob/Kont.** Benötigt wird ein Zugriff auf Subsystem mit komplexer innerer Struktur und innere Details verborgen bleiben

**Lösung**



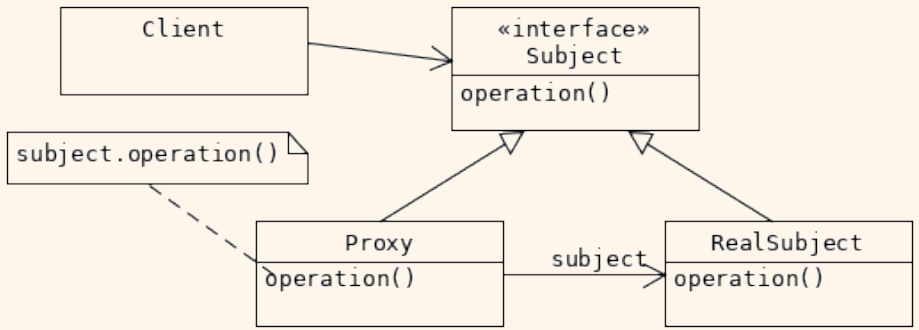

**Verweise** Proxy

# Proxy

**Zweck** Stellvertreter für ein anderes Objekt und kontrolliert Zugang zu dem Objekt

**Prob/Kont.** Zugang zu einem Objekt kann teuer sein oder Zugriff muss geregelt werden

**Lösung**




**Verwendung** *RemoteProxy* oder *ProtectionProxy* oder *VirtualProxy* (enthält Informationen von `RealSubject` falls Zugriff teuer)

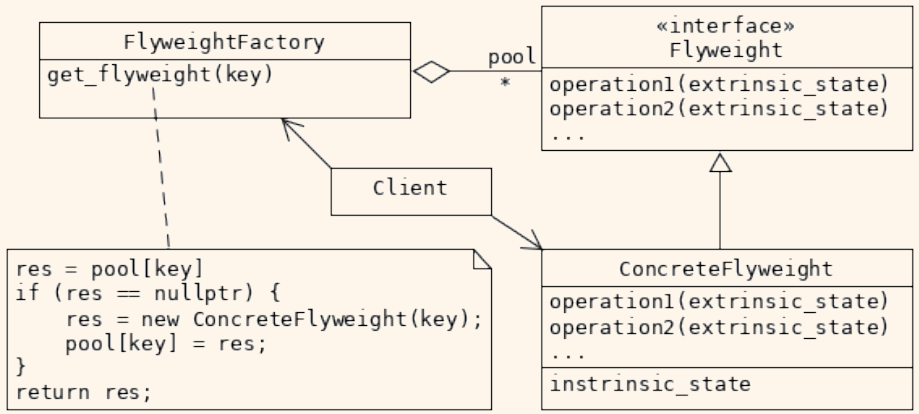
# Flyweight

**Zweck** Eine sehr große Anzahl feingranularer Objekte effizient verwalten

**Prob/Kont.** Es ist eine sehr große Anzahl an ähnlichen Objekten im Speicher zu halten. Der Speicherbedarf ist enorm.

**Lösung**

- *intrinsic* (wesentlich, inhärent, intrinsisch) vs. *extrinsic* (von außen wirkend, extern, extrinsisch)
- simultane Mehrfachverwendung von Objekten mit intrinsischen Zustand




**Verwendung** Flyweight-Objekte müssen immutable sein!

# Flyweight – 2

```
#include <iostream>
#include <map>
using namespace std;

struct Drawable {  // Flyweight
    virtual void draw(int x, int y)=0; };

struct Icon : public Drawable {
    string name;
    int width;
    int height;
    Icon(string name) : name{name} {
        width = 16;  // dependent of...
        height = 16;
    }
    void draw(int x, int y) {
        cout << "icon at " << x << ":" << y << endl;
    }
};
```

# Flyweight – 3

```
struct IconFactory {
    map<string, Icon*> pool;
    Icon* get(string name) {
        auto search = pool.find(name);
        if (search == pool.end()) {
            pool[name] = new Icon(name);
        }
        return pool[name];
    }
};

int main() {
    IconFactory factory;
    Icon* i1{factory.get("hearts")};
    Icon* i2{factory.get("diamonds")};
    Icon* i3{factory.get("hearts")};
    cout << (i1 == i2) << endl;  // -> 0
    cout << (i1 == i3) << endl;  // -> 1
}
```

# Behavioral Patterns

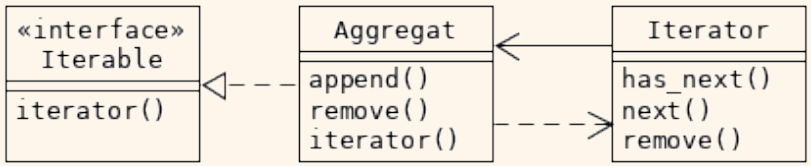
- (Chain of Responsibility)
- (Command)
- (Interpreter)
- Iterator
- Mediator
- (Memento)
- Observer
- State
- Strategy
- Template Method
- Visitor

# Iterator

**Zweck** Sequentieller Zugriff eines Clients auf die Elemente einer Collection, ohne deren internen Aufbau zu kennen.

**Prob/Kont.** Nacheinander auf alle Elemente einer Collection zugreifen, wobei unterschiedliche Traversierungsvarianten notwendig sein können.

**Lösung**




**Nachteile**

- Gleichzeitiges Hinzufügen bzw. Entfernen!
- Transparenz nicht immer möglich

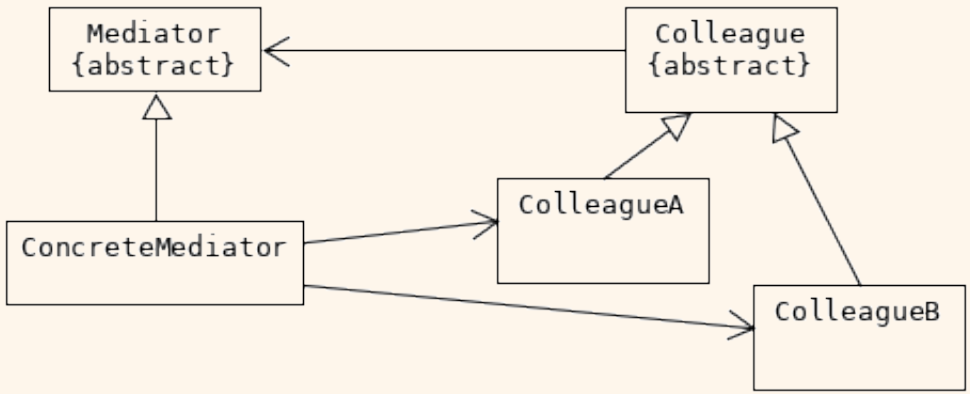
**Verweise** Composite, Factory Method

# Mediator

**Zweck** Entkoppeln vieler miteinander kommunizierender Objekte

**Prob/Kont.** Viele Objekte sind miteinander verbunden und kommunizieren miteinander. Die Abhängigkeiten sind schwer zu warten.

**Lösung**



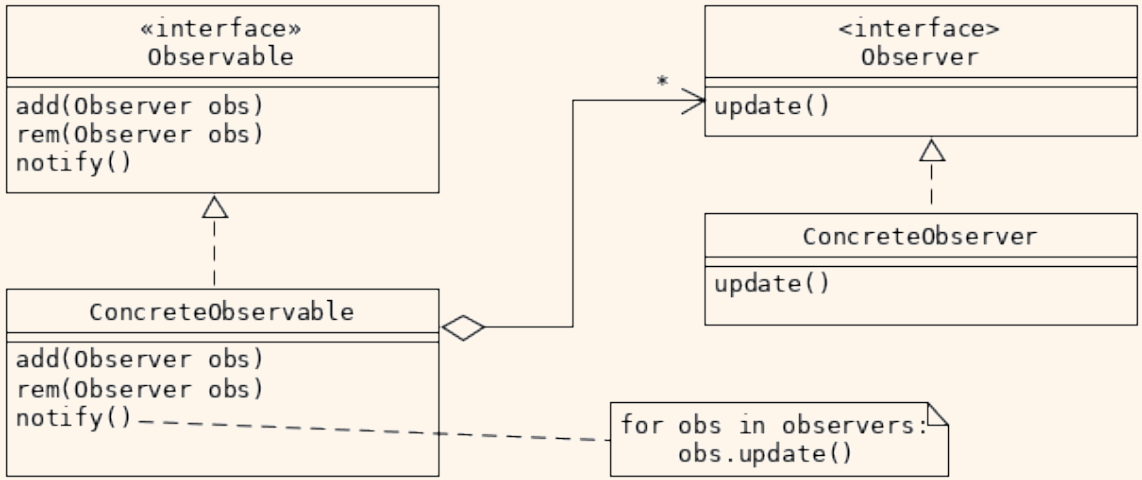

**Nachteile** Single point of failure, Skalierbarkeit

# Observer

**Zweck** Automatische Verteilung der Zustandsänderung eines Objektes auf mehrere andere Objekte.

**Prob./Kont.** Wenn sich Zustand ändert, müssen andere Objekte ihren Zustand automatisch anpassen.

**Lösung**



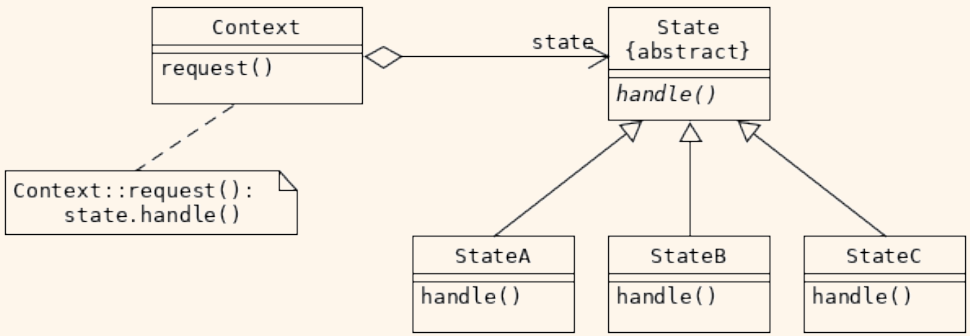

**Verweise** Alternative Namen: ”Publish-Subscribe”, ”Dependents”, ”Publisher-Subscriber”, ”Listener”

# State

**Zweck** Verändert das Verhalten eines Objektes, wenn sich dessen Zustand ändert

**Prob./Kont.** Verhalten eines Objektes hängt von seinem Zustand ab und der Zustand ändert sich im Laufe der Zeit.

**Lösung**



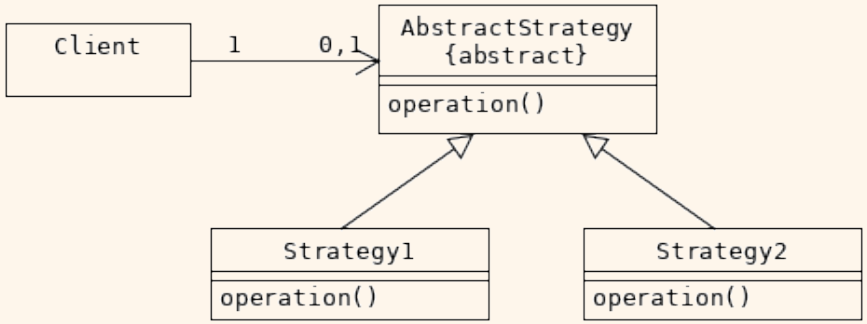

**Vorteil** keine unübersichtlichen `switch` Anweisungen :)

# Strategy

**Zweck** Kapselt einen Algorithmus in eine Klasse

**Prob./Kont.** Ein Algorithmus soll unabhängig von nutzenden Clients ausgetauscht werden können.

**Lösung**



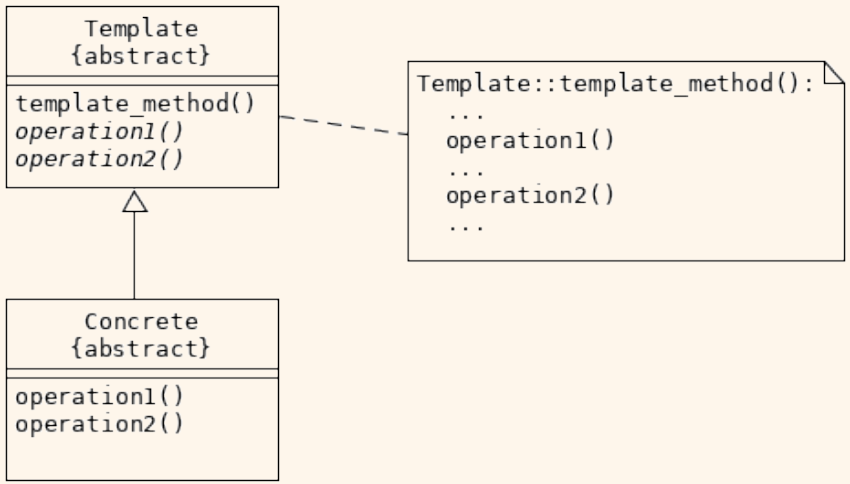

**Verweise** → Template Method

# Template Method

**Zweck** Definition eines Algorithmus, wobei einzelne konkrete Schritte in Unterklassen verlagert werden und damit eine gewisse Flexibilität erlangt wird.

**Prob./Kont.** Man will einen generischen Algorithmus beschreiben, von dem einzelnen Operationen variieren können.

**Lösung**



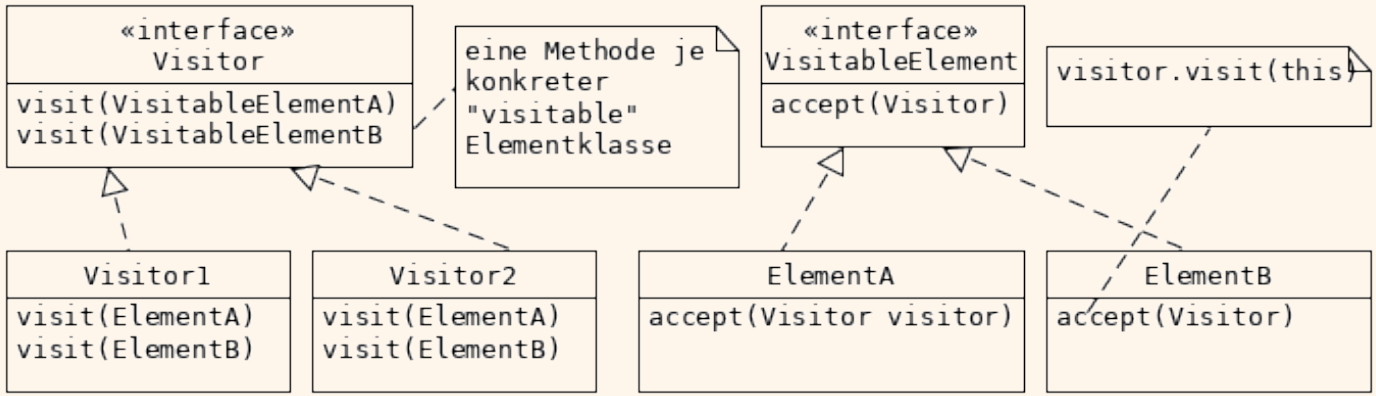

**Verweise** → Strategy

# Visitor

**Zweck** Man neue Operationen auf den Elementen zu definieren, ohne die Elemente selbst anzupassen

**Prob./Kont.** Es liegen unterschiedliche Elemente vor, auf die Operationen angewendet werden sollen, die jedoch nicht die zugehörigen Klassen aufblähen sollen.

**Lösung**




**Verwendung** Die Menge der konkreten ”visitable” Klassen sollte abgeschlossen sein, da auf Grund der starken Kopplung bzgl. der Methoden die Einführung einer neuen Klasse teuer ist.