HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Objektorientierte Analyse und Design
Entwurfsmuster

Prof. Dr.-Ing. Michael Uelschen
Hochschule Osnabrück
Sommersemester 2021

Objektorientierte Analyse und Design
Entwurfsmuster

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 00 Organisatorisches
- 01 Einführung[1]
- 02 Anforderungsanalyse[3]
- 03 Design[4]
- 04 Entwurfsmuster[3]
- 05 Sonstiges[1]

- Einleitung Entwurfsmuster
- SOLID
- BCE & MVC
- GRASP
- GoF
- Ausblick

Objektorientierte Analyse und Design

GESTALTUNGSMUSTER (DESIGN PATTERN)

Objektorientierte Analyse und Design
Gestaltungsmuster

- Ziel: Gestaltung des zu entwickelnden Systems auf Basis von Mustern.
- Frage: Was ist ein Muster?
  1. Vorlage, Zeichnung, **nach der etwas hergestellt**, gemacht wird.
  2. Etwas in seiner Art Vollkommenes, nachahmenswertes, **beispielhaftes Vorbild** in Bezug auf etwas Bestimmtes.
  3. …

- Vorteile bei Verwendung der Muster:
  - Man erhält gut strukturierte Programme.
  - Die Verständlichkeit der Programme wird deutlich erhöht (zumindest für jemanden, der die Muster kennt).
  - Man spart Aufwand, da für gewisse Problemsituationen keine eigenen Lösungen gesucht werden müssen.
  - Dient der Kommunikation mit anderen Informatikern/-innen.

Objektorientierte Analyse und Design
Design Stamina Hypothesis (M. Fowler)

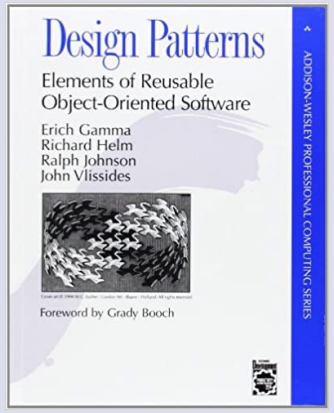
https://martinfowler.com/bliki/DesignStaminaHypothesis.html

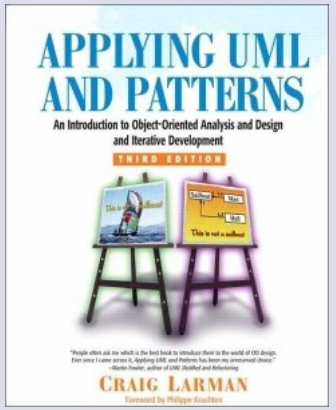
Objektorientierte Analyse und Design
Gestaltungsmuster

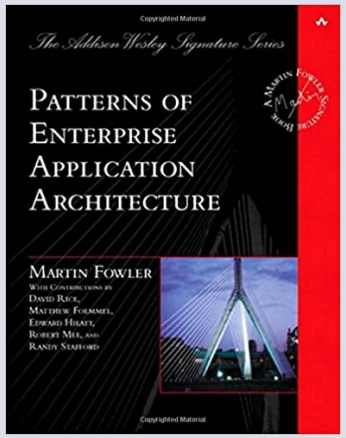
- Beschreiben Schablonen und bewährte Lösungsansätze zur Lösung von Entwurfs–problemen.
- Beruhen auf Erfahrung von Expert[inn]en – Softwarearchitekt[inn]en.
- Erfordern im praktischen Einsatz sehr viel Erfahrung.
- Kommen ursprünglich aus der Architektur und Stadtplanung (Christopher Alexander, 1978).
- Vorhanden in vielen Bereichen auch z.B. in der Gestaltung von Benutzeroberflächen.
- Vorhanden sind unterschiedlich akzentuierte Kataloge/Mustersammlungen.
- Werden von Tools unterstützt.

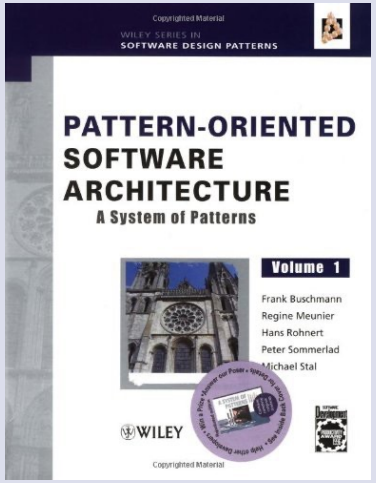
Objektorientierte Analyse und Design
Bekannte Mustersammlungen

- **GRASP** (General Responsibility Assignment Software Patterns) nach C. Larman
- **GoF-Pattern** (Gang of Four) nach Gamma, Helm, Johnson, Vlissides (Standard in der Softwareentwicklung)
- POSA (Pattern Oriented Software Architecture) nach Buschmann et al.
- DB-Pattern Patterns of Enterprise Application Architecture nach Martin Fowler









Objektorientierte Analyse und Design
Einteilung der Muster

- **Verhaltensmuster**: befassen sich mit der Verantwortlichkeit der Klassen (behavioral).
- **Kreationsmuster**: befassen sich mit der Objektgenerierung (creational).
- **Strukturmuster**: befassen sich mit der Zusammen–setzung (Komposition) von Klassen (structural).

- Muster werden üblicherweise beschrieben durch:
  - einen Namen, der allgemein bekannt und auch in der Dokumentation verwendet werden sollte.
  - die zugrunde liegende Regel bzw. eine Kurzbeschreibung.
  - der Hintergrund, d. h. Angaben zur Motivation und zum betreffenden Problembereich.

Objektorientierte Analyse und Design
Basisregeln

KISS

- **Keep It Simple Stupid**
- Wähle die einfachste, mögliche Realisierung, die das Problem vollständig löst und gut nachvollziehbar ist.
- Kein „Quick and Dirty“, sondern eine klare Entscheidung für einen einfachen Entwicklungsstil.

YAGNI

- **You Ain’t Gonna Need It**
- Entwickle keine Verallgemeinerungen, die das Design für theoretisch in der Zukunft vielleicht gewünschte Erweiterungen vereinfachen.

Objektorientierte Analyse und Design

BEISPIEL CQS-PRINCIPLE

Objektorientierte Analyse und Design
Command-Query-Separation Principle

- Prinzip:
  - Strikte Trennung der Schnittstelle (Methoden) einer Klasse in
    - **Command**: Objekt ändert den internen Zustand (schreibender Zugriff auf die Attribute).
    - **Query**: Objekt liefert einen Wert o.ä. zurück (lesender Zugriff).
  - Ein Kommando ändert den Zustand, liefert aber keinen Wert zurück; oder liefert einen Wert, ändert aber nicht den Zustand. Das Stellen einer Frage sollte nicht die Antwort beeinflussen.
- Hintergrund:
  - Die Schnittstelle einer Klasse sollte möglichst übersichtlich und verständlich sein. Insbesondere muss transparent sein, wie Attribute verändert werden.
- Hinweis: in C++ gibt es `const`-Schlüsselwort, das verdeutlicht, ob Methode command oder query ist. Java?

Objektorientierte Analyse und Design
Command-Query Separation in C++

Ohne Command-Query-Separation

```
class Wuerfel {
public:

    explicit Wuerfel(unsigned int n=6);

    unsigned int rollen();

private:
    unsigned int nside = 0;
    unsigned int value = 1;
};


unsigned int Wuerfel::rollen() {
    return value=1+rand()%nside;
}
```

Mit Command-Query-Separation

```
class Wuerfel {
public:

    explicit Wuerfel(unsigned int n=6);

    void rollen();
    unsigned int augenzahl() const;

private:
    unsigned int nside = 0;
    unsigned int value = 1;
};


void Wuerfel::rollen() {
    value=1+rand()%nside;
}
```

Objektorientierte Analyse und Design
Command-Query Separation in C++ 2


```
class Wuerfel {
public:

    explicit Wuerfel(unsigned int n=6);

    /* --Variante. */
    Wuerfel& rollen();

    unsigned int augenzahl() const;

private:
    unsigned int nside = 0;
    unsigned int value = 1;
};
```

```
Wuerfel& Wuerfel::rollen() {
    value=1+rand()%nside;
    return *this;
}



/* --Aufruf

   Wuerfel w;
   unsigned int augen;

   augen=w.rollen().augenzahl();

*/
```

Objektorientierte Analyse und Design

SOLID

Objektorientierte Analyse und Design
SOLID

- Sammlung von 5 grundlegenden **Prinzipien** für **objektorientiertes Design**.
- Beschrieben von Robert C. Martin („Uncle Bob“) in 2000; https://blog.cleancoder.com.
- (Erfolgreiche) Anwendungen werden über Jahre/Jahrzehnte weiterentwickelt.
- Grundidee: **Entwicklung von wartbarem Code**.

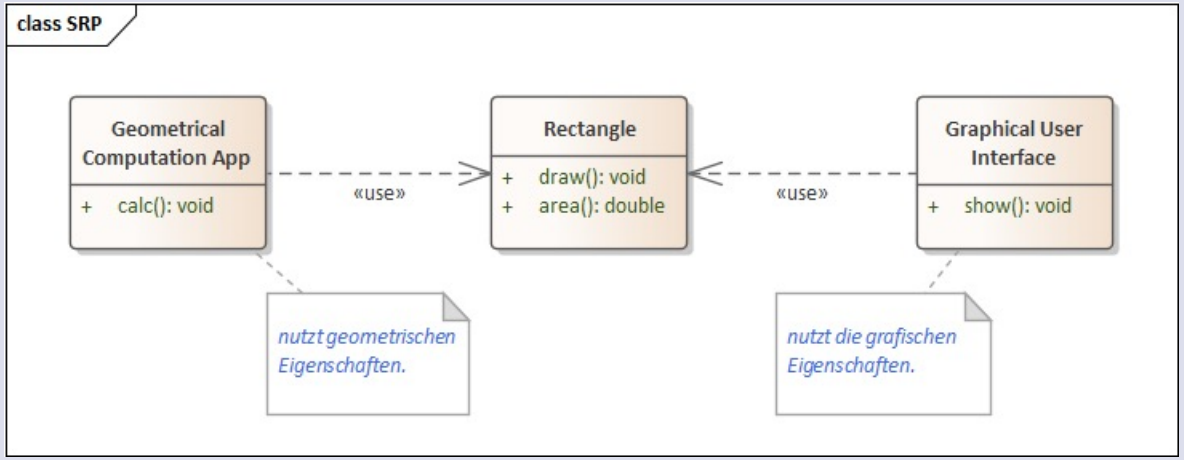
- **Abhängigkeiten** erschweren die Wartung von Anwendungen:
  - **Quelltextebene**: gegenseitiges Einbinden von Header-Dateien.
    - Aufwendig, aber lösbar.
  - **Designebene**: Austausch/Erweiterung von Klassen erfordert Neuentwicklung und Test (!) großer Teile der Anwendung.
    - Sehr ärgerlich, führt zu hohen Kosten.

Objektorientierte Analyse und Design
Single-responsibility principle

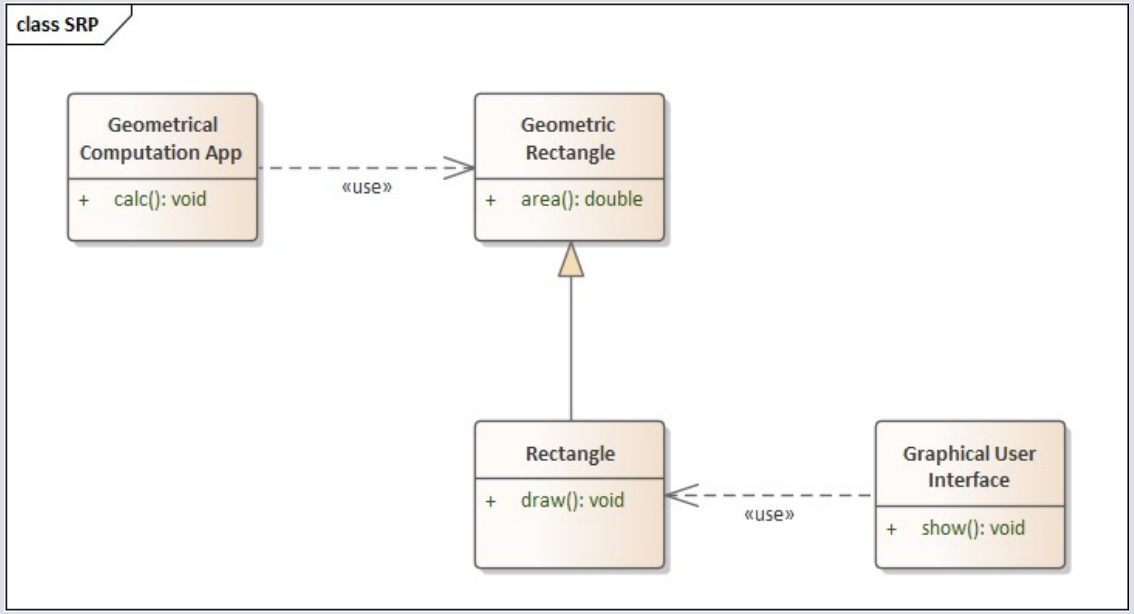
- Prinzip: **„A class should have only one reason to change.“**
- Idee: Eine Klasse sollte nur eine **Verantwortlichkeit** haben.
- Das Vermischen von mehreren Verantwortlichkeiten oder Aufgaben in einer einzelnen Klasse führt zu **schwer änderbaren Abhängigkeiten** im Design und Code.
- Auslöser einer Veränderung entspricht der Verantwortlichkeit.
- Einfaches Prinzip, aber schwer umzusetzen. Wie lassen sich unterschiedliche Verantwortlichkeiten definieren und finden?
- Problem: **Needless Complexity** (Overdesign).

Objektorientierte Analyse und Design
Single-responsibility principle

Mehrere Verantwortlichkeiten 👎



Geteilte Verantwortlichkeiten 👍

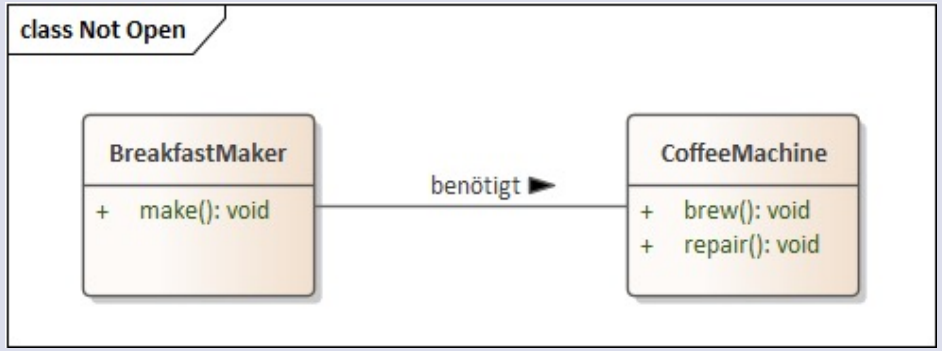


Objektorientierte Analyse und Design
Open–closed principle

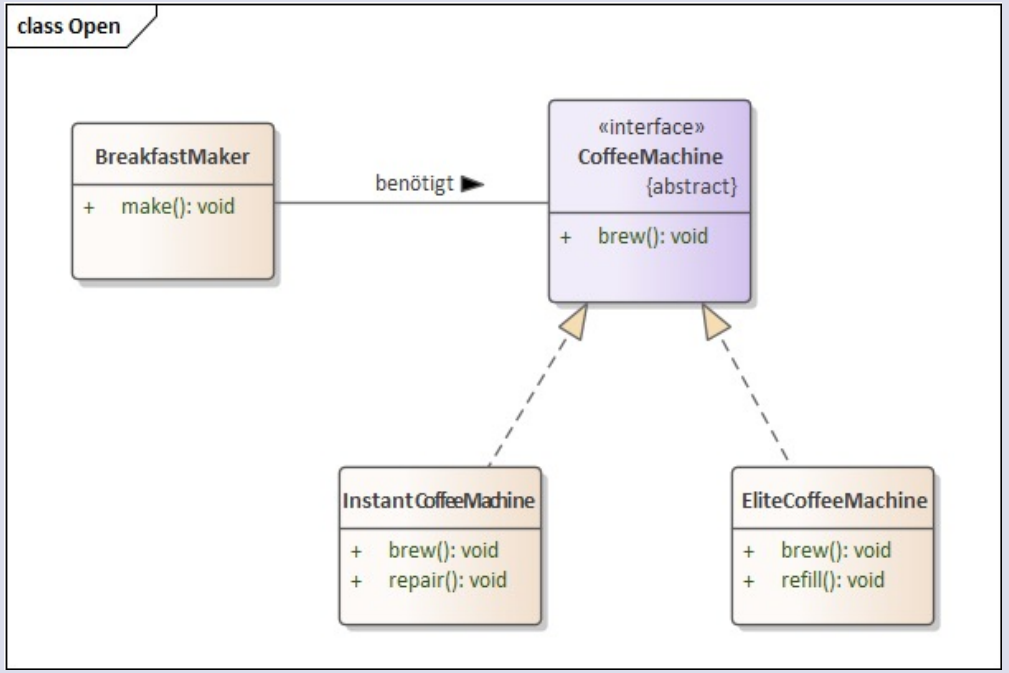
- Prinzip: **„Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.“**
- Ursprünglich von Bertrand Meyer (1988) beschrieben.
- Idee: Klassen so entwerfen, so dass diese nicht mehr geändert werden müssen.
- **Offen für Erweiterungen**: Neue Anforderungen werden durch Ergänzungen berücksichtigt.
- **Geschlossen für Änderungen**: Der entwickelte und getestete Quelltext wird nicht verändert.

Objektorientierte Analyse und Design
Open–closed principle

Not Open 👎



Open 👍



Objektorientierte Analyse und Design
C++: Erweiterungen nicht möglich


```cpp
class BreakfastMaker {
public:

    BreakfastMaker();

    void makeBreakfast(unsigned int nperson);

private:

    std::unique_ptr<CoffeeMachine> machine;

};
```

```cpp
class CoffeeMachine {
public:

    CoffeeMachine() = default;

    /* --Main functionality. */
    void brew(unsigned int ncups);

    /* --Specific methods to this type. */
    void repair();
    bool isCoffeeAvailable() const;

private:
    /* --Capacity in ml. */
    unsigned int capacity = 0;
};
```

Objektorientierte Analyse und Design
C++: Offen für Erweiterungen


```cpp
class CoffeeMachineInterface {
public:

    /* --Interface to every coffee machine. */
    virtual void brew(unsigned int ncups) = 0;

    /* --Virtual destructor is required. */
    virtual ~CoffeeMachineInterface()=default;
};
```

```cpp
class OpenBreakfastMaker {
public:

    explicit OpenBreakfastMaker(
       std::shared_ptr<CoffeeMachineInterface> m);

    void makeBreakfast(unsigned int nperson);

private:
    std::shared_ptr<CoffeeMachineInterface>
       machine;
};
```

Objektorientierte Analyse und Design
C++: Offen für Erweiterungen 2


```
/* --InstantCoffeeMaker. */
class InstantCoffeeMaker :
                public CoffeeMachineInterface {
public:

    InstantCoffeeMaker() = default;

    void brew(unsigned int nperson) override {
        std::cout
          << "Preparing instant coffee for “
          << nperson << "." << std::endl;
    }

};
```

```
class EliteCoffeeMaker :
                public CoffeeMachineInterface {
public:

    explicit EliteCoffeeMaker(
         const std::string& b):beans(b) {}

    void brew(unsigned int nperson) override {
        std::cout << "Best coffee ever with “
              << beans << "." << std::endl;
    }

private:
    std::string beans;
};
```

Objektorientierte Analyse und Design
C++: Offen für Erweiterungen 3


```
std::shared_ptr<InstantCoffeeMaker>
instant=std::make_shared<InstantCoffeeMaker>();
OpenBreakfastMaker weekday(instant);

std::shared_ptr<EliteCoffeeMaker>
elite=std::make_shared<EliteCoffeeMaker>(
                                    "Kopi Luwak");
OpenBreakfastMaker sunday(elite);

/* --Weekdays. */
weekday.makeBreakfast(2);
/* --Sunday. */
sunday.makeBreakfast(2);
```

```
Let's brew some coffee...
Preparing instant coffee for 2.
Done.
Let's brew some coffee...
Best coffee ever with Kopi Luwak.
Done.
```

Objektorientierte Analyse und Design
Open–closed principle – Heuristiken

1. Alle Attribute sind als `private` zu deklarieren.
2. Keine (niemals!) Verwendung von globalen Variablen.
3. Verwendung von RTTI (Typinformationen zur Laufzeit) oder `dynamic_cast` sind zu vermeiden.

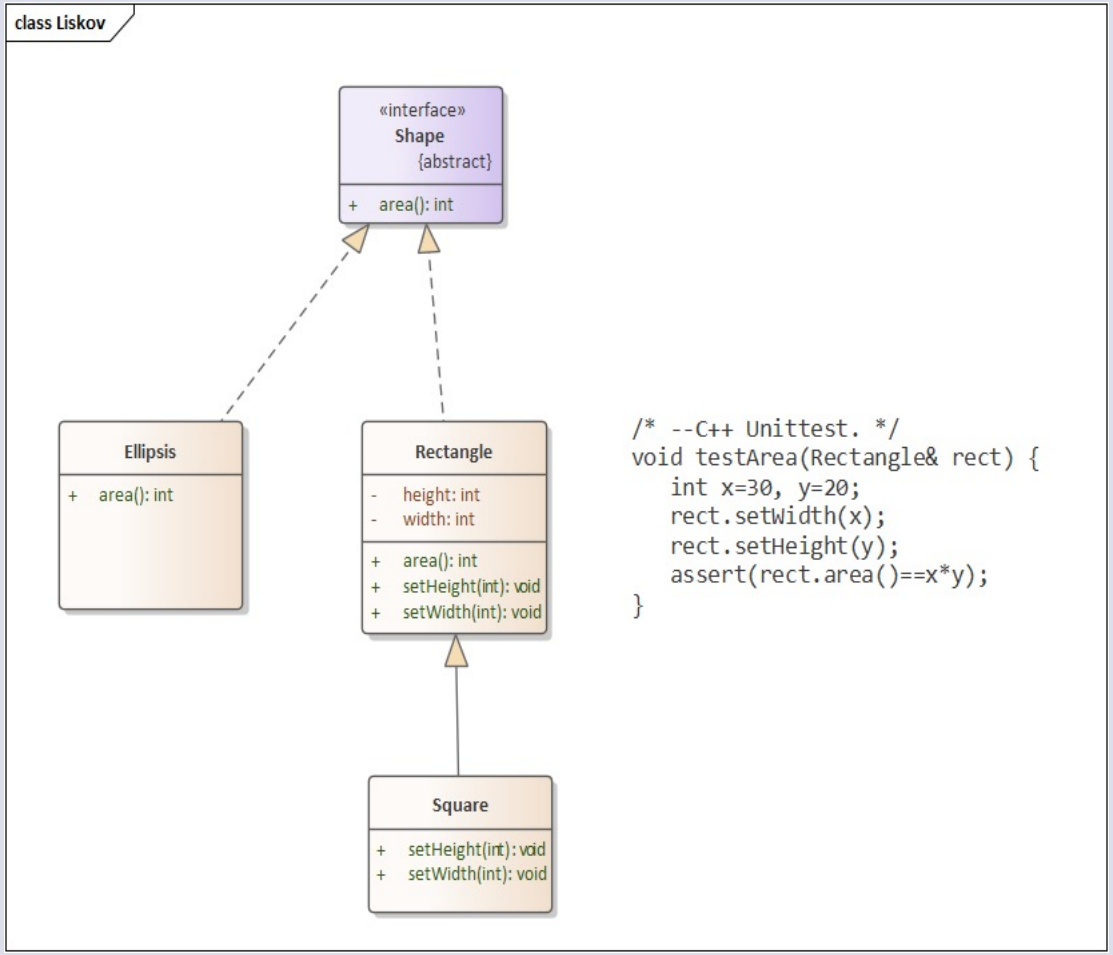
Objektorientierte Analyse und Design
Liskov substitution principle

- Prinzip: **„Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.“**
- Ursprünglich von Barbara Liskov (1988) beschrieben.
- Idee: Die Eigenschaften der Basisklasse müssen in der abgeleiteten Klasse erhalten bleiben.

- „What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.“

Objektorientierte Analyse und Design
Liskov substitution principle

- Beispiel: Geometrie – Berechnung Flächeninhalt.
- Klasse `Square` als Spezialisierung von `Rectangle`: „Quadrat ist ein Rechteck“
  - Problem: Attribute sind nicht mehr unabhängig voneinander: Höhe==Breite!
  - Folge: Eigenschaften der Basisklasse gelten nicht mehr!
  - Lösung?



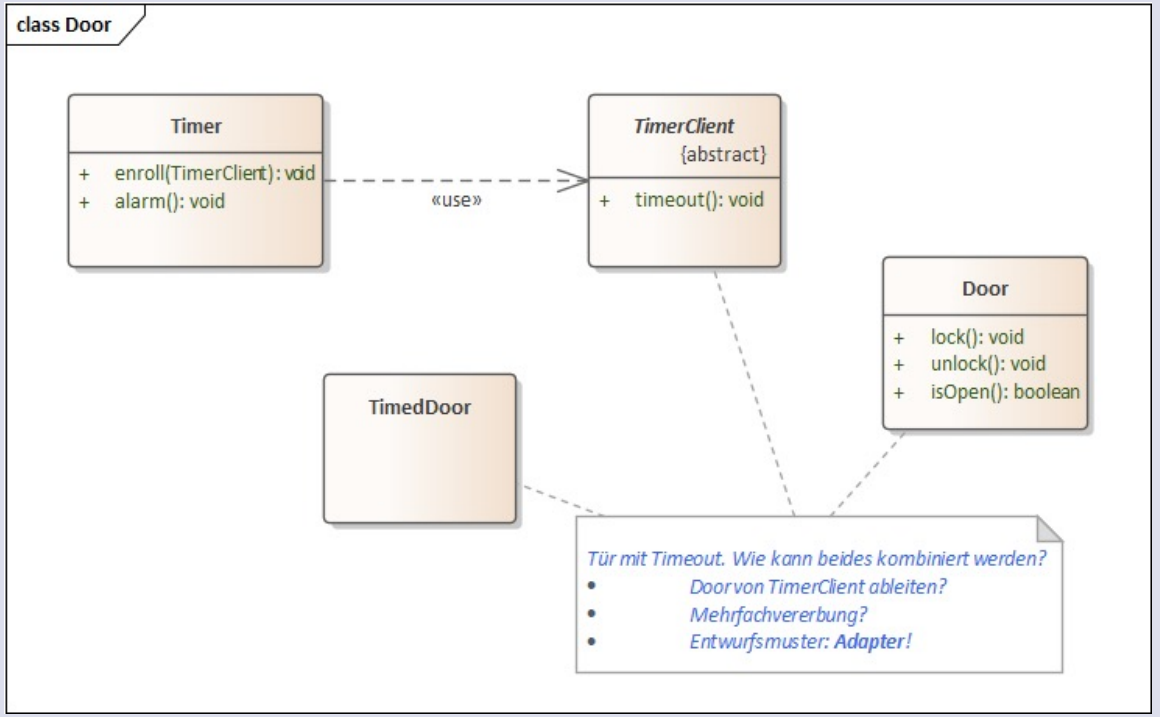
Objektorientierte Analyse und Design
Design by Contract

- Idee: Funktionen/Methoden deklarieren
  - **Vorbedingung** (precondition): Bedingung muss gelten bevor die Funktion/Methode ausgeführt wird.
  - **Nachbedingung** (postcondition): Bedingung gilt, nachdem die Funktion/Methode ausgeführt wurde.
- In C/C++ über `assert`-Makro (eingeschränkt) unterstützt.

- Vererbung
  - Vorbedingungen in abgeleiteten Klassen können schwächer, aber nicht stärker formuliert sein.
  - Nachbedingungen in abgeleiteten Klassen können stärker, aber nicht schwächer formuliert sein.

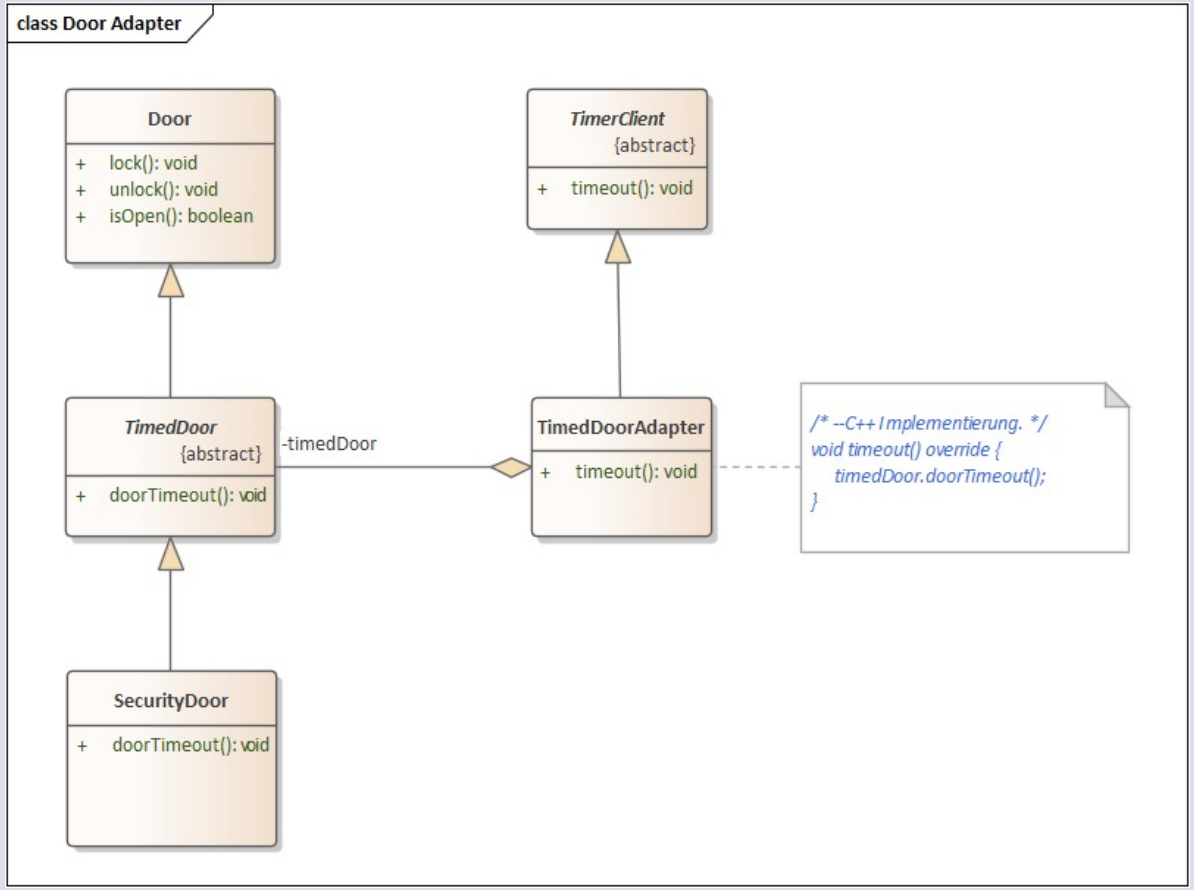
Objektorientierte Analyse und Design
Interface segregation principle

- Prinzip: **„Client should not be forced to depend upon interfaces that they do not use.“**
- Idee: Eine Klasse, die eine Schnittstelle (Interface) nicht nutzt, sollte nicht von der Schnittstelle abhängig sein.
- Änderungen der Schnittstelle betreffen auch unbeteiligte Klassen.



Objektorientierte Analyse und Design
Interface segregation principle: Beispiel

- Kombination der `Door`-Klasse mit einer Zeitsteuerung (`Timer`) zu einer `TimedDoor`-Klasse.
- Realisierungsmöglichkeiten:
  1. Door als Spezialisierung von TimerClient: Jede Tür muss timeout realisieren (Interface Pollution)
  2. Mehrfachvererbung: Muss durch Sprache (C++: 👍 Java: 👎) unterstützt werden.
  3. **Delegation mit Adapter**: Kopplung unterschiedlicher Konzepte.



Objektorientierte Analyse und Design
C++: Getrennte Konzepte vereinen


```cpp
class DoorInterface {
public:
    /* --Abstract methods. */
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual bool isOpened() const = 0;

    virtual ~DoorInterface() = default;
};
```

```cpp
class TimerClient {
public:
    /* --Abstract method. */
    virtual void timeout() = 0;
    virtual ~TimerClient() = default;
};

class Timer {
public:
    /* --Register object to get alarmed. */
    void enroll(unsigned int t,
                std::shared_ptr<TimerClient> c);
    /* --Alarm. */
    void alarm();
protected:
    std::shared_ptr<TimerClient> client;
    unsigned int timeout;
};
```

Objektorientierte Analyse und Design
C++: Getrennte Konzepte vereinen 2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```cpp
class TimedDoorInterface : public
                           DoorInterface {
public:

    virtual void doorTimeout() = 0;

};
```

```cpp
class TimedDoorAdapter : public TimerClient {
public:

    TimedDoorAdapter(
      std::shared_ptr<TimedDoorInterface> td):
                                  timedDoor(td) {}

    /* --Delegation. */
    void timeout() override {
        if (timedDoor)
            timedDoor->doorTimeout();
    }

private:
    std::shared_ptr<TimedDoorInterface>
                                      timedDoor;
};
```

Objektorientierte Analyse und Design
C++: Getrennte Konzepte vereinen 3


```
class SecurityDoor : public
                     TimedDoorInterface {
public:
    /* --Concrete implementation. */
    void lock() override;
    void unlock() override;
    bool isOpened() const override;

    void doorTimeout() override;

private:
    bool open=true;
};
```

```
Timer clock;

auto door=std::make_shared<SecurityDoor>();
auto adapter=
     std::make_shared<TimedDoorAdapter>(door);

clock.enroll(100,adapter);

door->lock();
door->unlock();

clock.alarm();
```

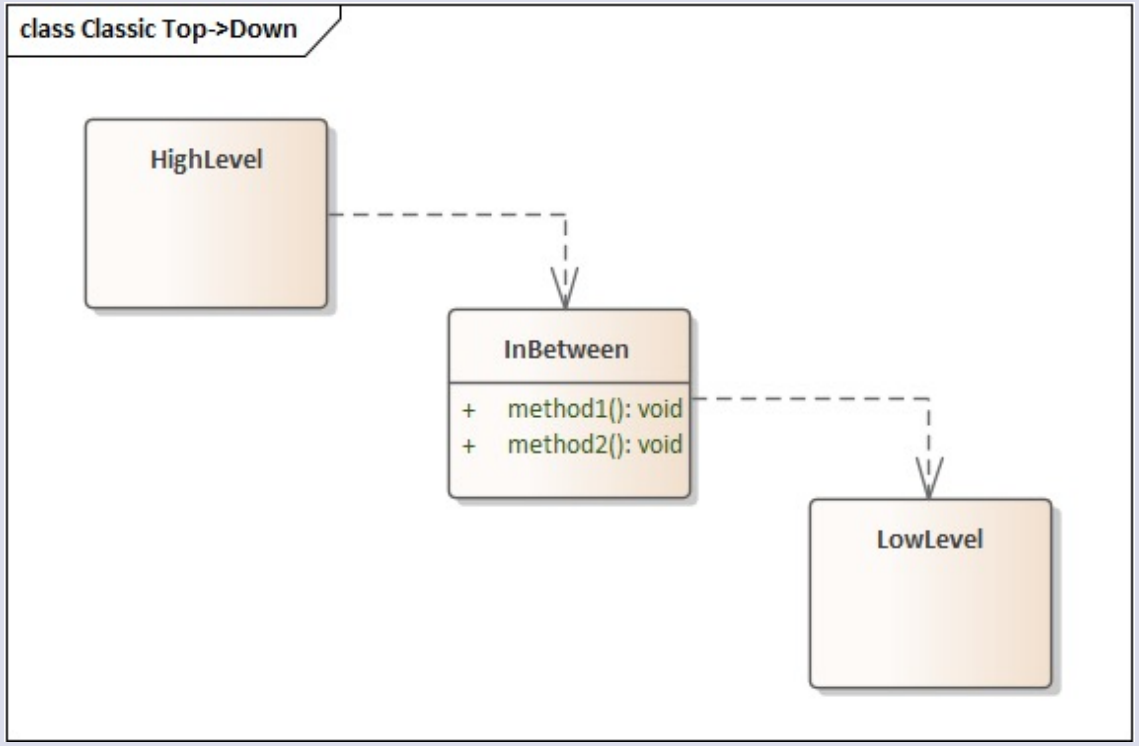
Objektorientierte Analyse und Design
Dependency inversion principle

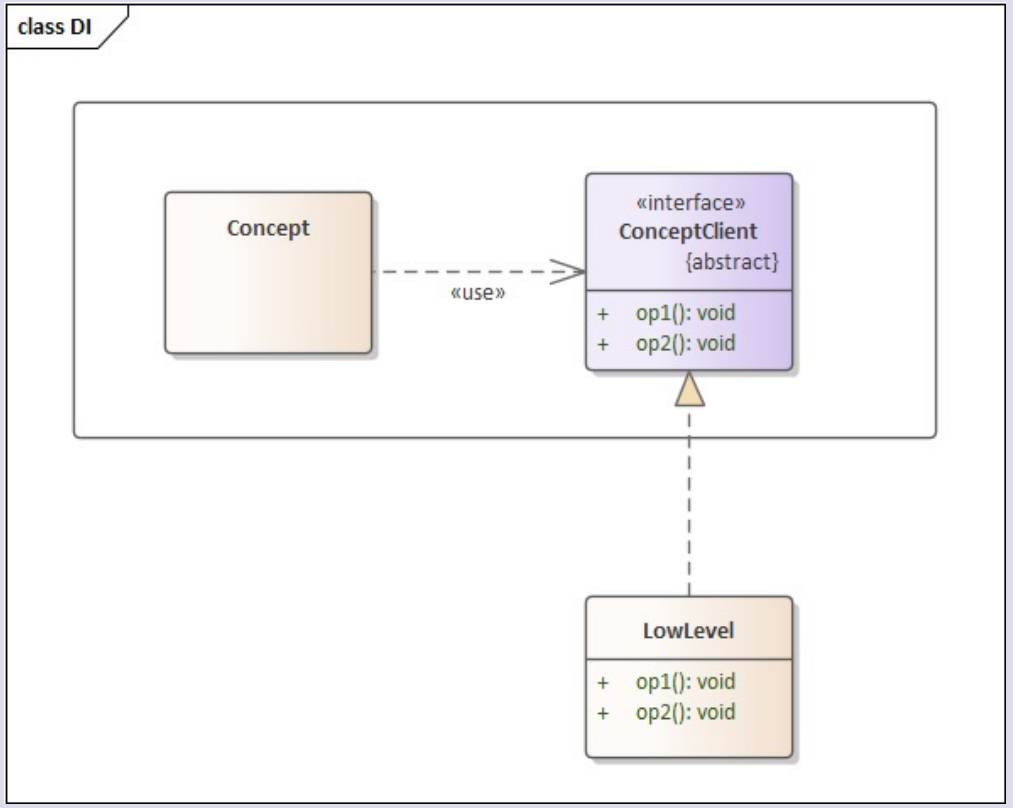
- Prinzip: **„A. High level modules should not depend upon low level modules. Both should depend upon abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions.“**
- Idee: Abhängigkeiten durch abstrakte Klassen oder Interfaces (Schnittstellen) auflösen.

- Bisher
  - Höhere Softwareschichten (Klassen, Komponenten, …) greifen auf die Schnittstelle der unteren Schichten zu.
  - Software-Architektur: höhere Schichten haben höheres Abstraktionsniveau als darunterliegende, z.B. hardware-naher Schichten.
- Jetzt
  - Vermeidung der Abhängigkeiten durch Umkehrung.

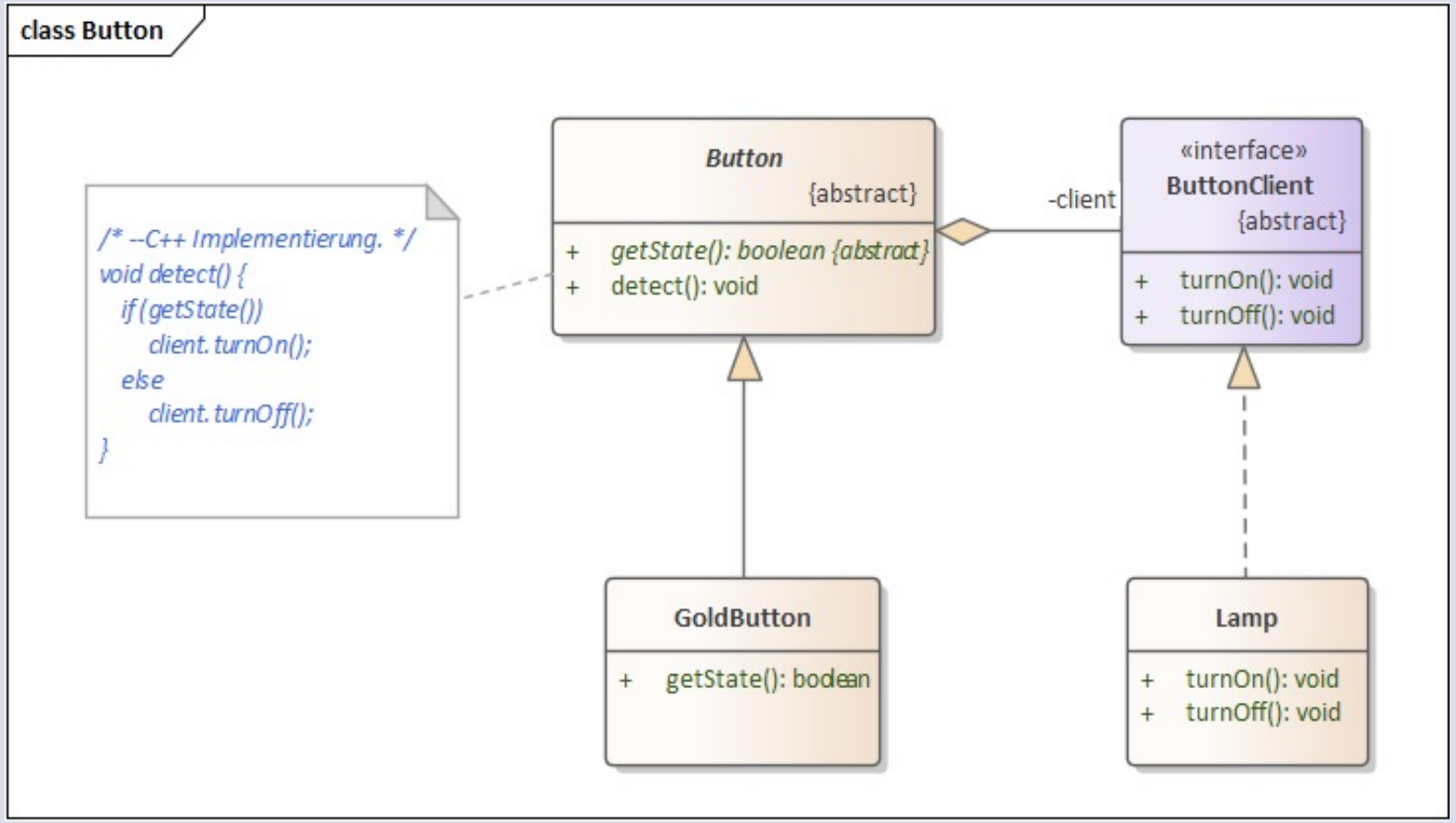
Objektorientierte Analyse und Design
Dependency inversion principle

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES





Objektorientierte Analyse und Design
Dependency inversion principle: Beispiel



Objektorientierte Analyse und Design
C++: Getauschte Abhängigkeiten


```cpp
class ButtonClient {
public:
    /* --Abstract interface. */
    virtual void turnOn() = 0;
    virtual void turnOff() = 0;

    virtual ~ButtonClient() = default;
};
```

```cpp
class Lamp : public ButtonClient {
public:

    Lamp() = default;

    /* --Concrete implementation. */
    void turnOn() override;
    void turnOff() override;

};
```

Objektorientierte Analyse und Design
C++: Getauschte Abhängigkeiten 2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```cpp
class AbstractButton {
public:

    explicit AbstractButton(
             std::unique_ptr<ButtonClient> c);

    /* --Abstract method to get user input. */
    virtual bool getState() = 0;

    /* --Switch client depending on state. */
    void detect();

protected:
    std::unique_ptr<ButtonClient> client;
};
```

```cpp
AbstractButton::AbstractButton(
         std::unique_ptr<ButtonClient> c):
                        client(std::move(c)) {}

void AbstractButton::detect() {
    /* --Check the user input. */
    if (getState())
        client->turnOn();
    else
        client->turnOff();
}
```

Objektorientierte Analyse und Design
C++: Getauschte Abhängigkeiten 3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
class GoldButton : public AbstractButton {
public:

    explicit GoldButton(
            std::unique_ptr<ButtonClient> c);

    /* --Get user input. */
    bool getState() override;

};
```

```
GoldButton::GoldButton(
            std::unique_ptr<ButtonClient> c) :
                AbstractButton(std::move(c)) {}

bool GoldButton::getState() {
    /* --Ask the user. */
    std::cout << "(0) to switch off, (1) to
switch on:... " << std::flush;
    int input;
    std::cin >> input;
    return input;
}
```

Objektorientierte Analyse und Design
C++: Getauschte Abhängigkeiten 4

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```cpp
int main() {
    std::cout << "Hello, Button World!"
              << std::endl;

    auto lamp=std::make_unique<Lamp>();
    auto button=std::make_unique<GoldButton>(
                              std::move(lamp));

    for(int i=0;i<3;i++)
        button->detect();

    return 0;
}
```

```
Hello, Button World!
(0) to switch off, (1) to switch on:... 1
Lamp is on.
(0) to switch off, (1) to switch on:... 0
Lamp is off.
(0) to switch off, (1) to switch on:... 1
Lamp is on.
```

Objektorientierte Analyse und Design

BCE & MVC

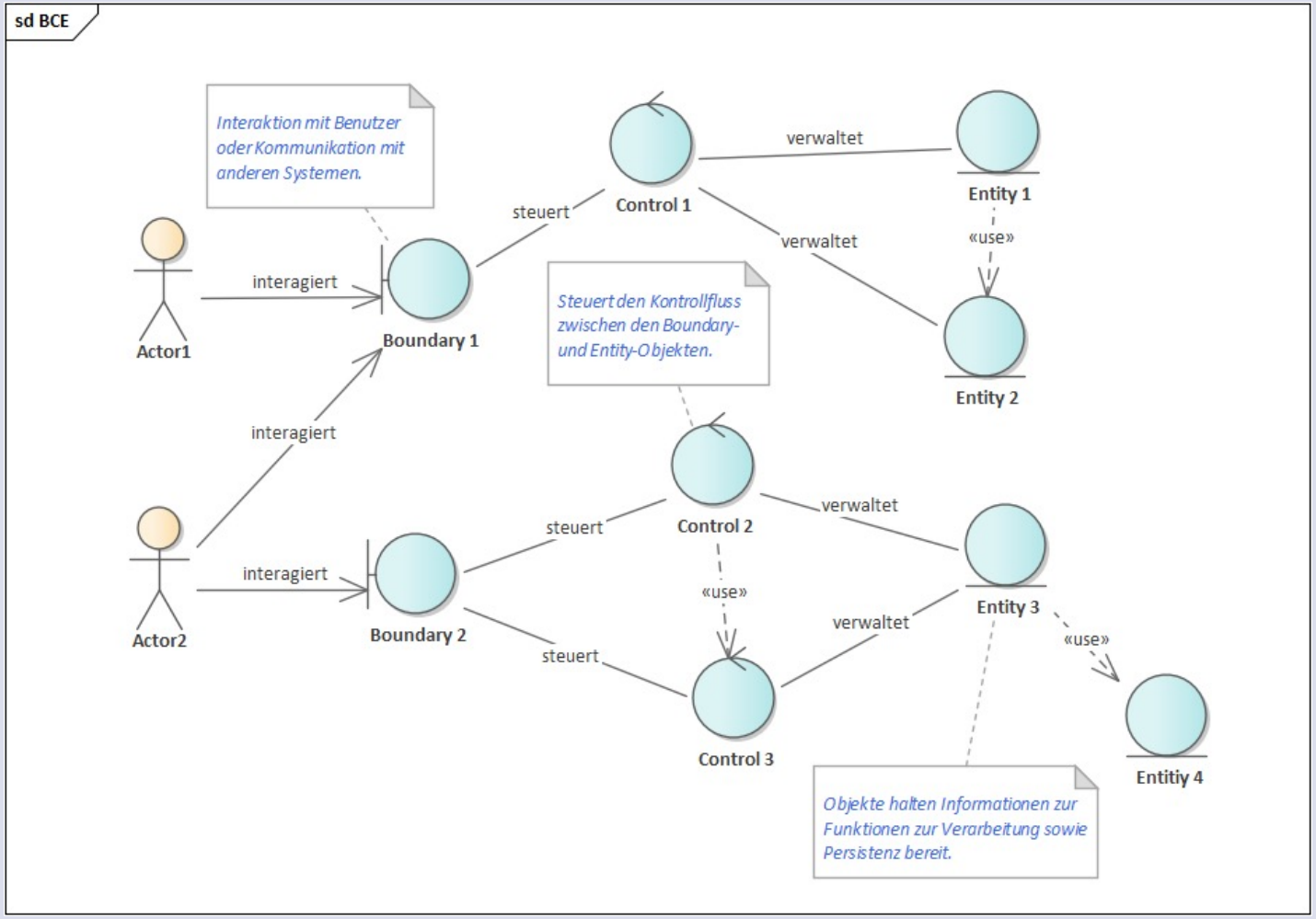
Objektorientierte Analyse und Design
Robustheitsanalyse

- Strukturierung der Klassen in drei **Verantwortlichkeiten**: **Boundary**, **Control** und **Entity**.
- Ursprünglich von Ivar Jacobson (einer der „drei Amigos“, 1992) beschrieben.
- Idee: Klare **Aufgabenteilung** und Definition der Schnittstellen.

- **Boundary**: Interaktion mit den Akteuren (HMI) und anderen Systemen.
- **Control**: Kontrollfluss (Ablaufsteuerung) der Aktivitäten/Anwendungsfälle.
- **Entity**: Informationshaltung und Konzepte aus der Fachdomäne ggfs. mit Persistenz.

Objektorientierte Analyse und Design
Boundary – Control – Entity (BCE)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



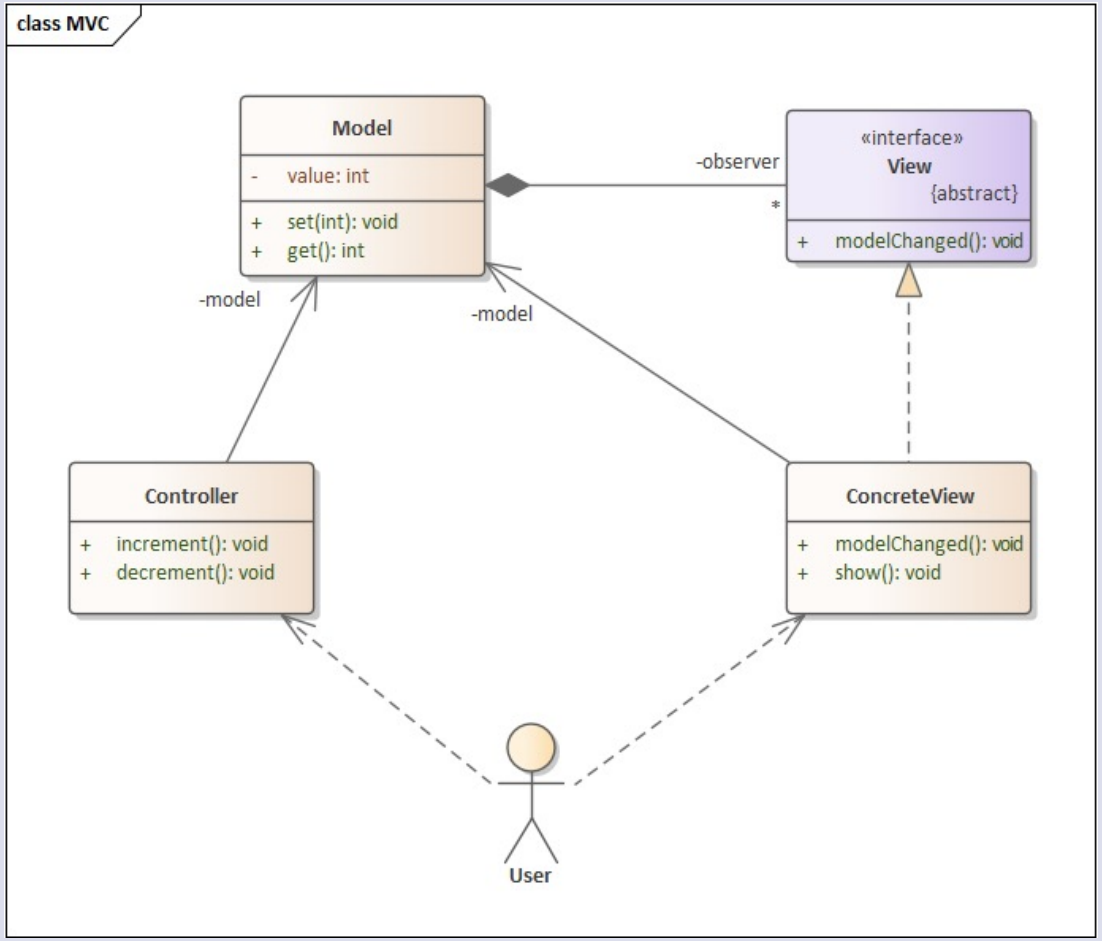
Objektorientierte Analyse und Design
Vom BCE zum Model-View-Controller

- Eine Variante des BCE-Prinzips ist das **MVC**-Muster.
- Typisch für graphische Oberflächen ist, dass es **Objekte zur Eingabe** gibt, die zur Bearbeitung der eigentlichen **Inhaltsklasse** führen, die dann eventuell zu **Änderung der Anzeige** führen.
- Die Aufteilung in die drei genannten Aufgaben führt zum Model-View-Controller-Ansatz.

- MVC wurde zuerst in Smalltalk Ende der 80'er des vorigen Jahrhunderts eingesetzt:
  - **Model**: Zustandsinformation der Komponente (Inhaltsklasse der Domäne.)
  - **View**: Beobachter des Zustands, um diesen darzustellen; es kann viele Views geben.
  - **Controller**: Legt das Verhalten der Komponente auf Benutzereingaben fest.

Objektorientierte Analyse und Design
Model-View-Controller – Varianten

- Kommunikationswege hängen von konkreter Umsetzung ab
  - Model-Delegate (Controller und View zusammen)
  - Model-View-ViewModel (eigenes Model für Darstellung)
  - Model-View-Presenter
  - Model-View-Adapter
- Eine Umsetzung: Controller steuert Änderungen des Modells, Modell teilt allen Views mit, dass eine Änderung aufgetreten ist.



Objektorientierte Analyse und Design
Model-View-Controller in C++ (View,Ctrl)


```cpp
class ViewInterface {
public:
    /* --Abstract Interface. */
    virtual void modelChanged() = 0;
    /* --Virtual method implies virtual
        destructor. */
    virtual ~ViewInterface() = default;
};
```

```cpp
class Controller {
public:

    explicit Controller() = default;

    /* --Set the model. */
    void set(Model* m);

    /* --User input. */
    void loop();

protected:
    Model* model = nullptr;
};
```

Objektorientierte Analyse und Design
Model-View-Controller in C++ (Model)


```
class Model {
public:
    using ViewPtr = std::unique_ptr<ViewInterface>;
    explicit Model() = default;
    /* --Add new view to the model. The model owns
     * the view (lifetime ownership)! */
    void add(ViewPtr v);
    /* --Modify data. */
    void modify(int d);
    /* --Read data. */
    int read() const { return data; }
protected:
    /* --The views. */
    std::vector<ViewPtr> views;
    /* --Inform the views after changes. */
    void updateViews() const;
    /* --Model data. */
    int data = 42;
};
```

```
void Model::add(ViewPtr v) {
    views.push_back(std::move(v));
}

void Model::updateViews() const {
    /* --Loop on the views. */
    for(auto& v: views)
        v->modelChanged();
}

void Model::modify(int d) {
    data=d;
    updateViews();
}
```

Objektorientierte Analyse und Design
Model-View-Controller in C++ (Views)


```cpp
class ValueView : public ViewInterface {
public:

    explicit ValueView(const Model* m);

    /* --Concrete implementation . */
    void modelChanged() override;

protected:
    /* --Link to the model. */
    const Model* model = nullptr;
    /* --Show formatted value. */
    void show(int value, int width =4) const;
};
```

```cpp
class FancyView : public ViewInterface {
public:

    explicit FancyView(const Model* m);

    /* --Concrete implementation . */
    void modelChanged() override;

protected:
    /* --Link to the model. */
    const Model* model = nullptr;
    /* --Show the value. */
    void show();
    /* --Local data of the view. */
    int current = -1;
};
```

Objektorientierte Analyse und Design
Model-View-Controller in C++ (Impl.)


```
void ValueView::modelChanged() {
    /* --Show the value. */
    show(model->read());
}

void ValueView::show(int value, int width)
const {
    std::cout << "* The current value is: "
        << std::setw(width) << value
        << std::endl;
}

ValueView::ValueView(const Model *m):model(m)
{}
```

```
FancyView::FancyView(const Model *m):model(m) {
    /* --Get initial value. */
    current=model->read();
}

void FancyView::show() {
    /* --Get new value. */
    int newvalue=model->read();
    std::cout << "* The value changed: "
        << current << " --> " << newvalue << "."
        << std::endl;
    current=newvalue;
}

void FancyView::modelChanged() {
    show();
}
```

Objektorientierte Analyse und Design
Model-View-Controller in C++ (Ctrl-Impl.)


```
void Controller::set(Model *m) {
    model=m;
}

void Controller::loop() {
    bool done=false;
    do {
        std::cout << "Enter +/- to increase; x to exit: " << std::flush;
        char c;
        std::cin >> c;
        int value=model->read();
        switch (c) {
            case '+' : model->modify(value+1); break;
            case '-' : model->modify(value-1); break;
            case 'x' : done=true; break;
            default  : ;
        }
    } while (!done);
}
```

Objektorientierte Analyse und Design
Model-View-Controller in C++ (main)


```
auto model=std::make_unique<Model>();
auto controller=std::make_unique<Controller>();

Model::ViewPtr
view=std::make_unique<ValueView>(model.get());
model->add(std::move(view));

Model::ViewPtr
fancy=std::make_unique<FancyView>(model.get());
model->add(std::move(fancy));

controller->set(model.get());
controller->loop();
```

```
Enter +/- to increase; x to exit: +
* The current value is:   43
* The value changed: 42 --> 43.
Enter +/- to increase; x to exit: +
* The current value is:   44
* The value changed: 43 --> 44.
Enter +/- to increase; x to exit: -
* The current value is:   43
* The value changed: 44 --> 43.
Enter +/- to increase; x to exit: x
```

Objektorientierte Analyse und Design

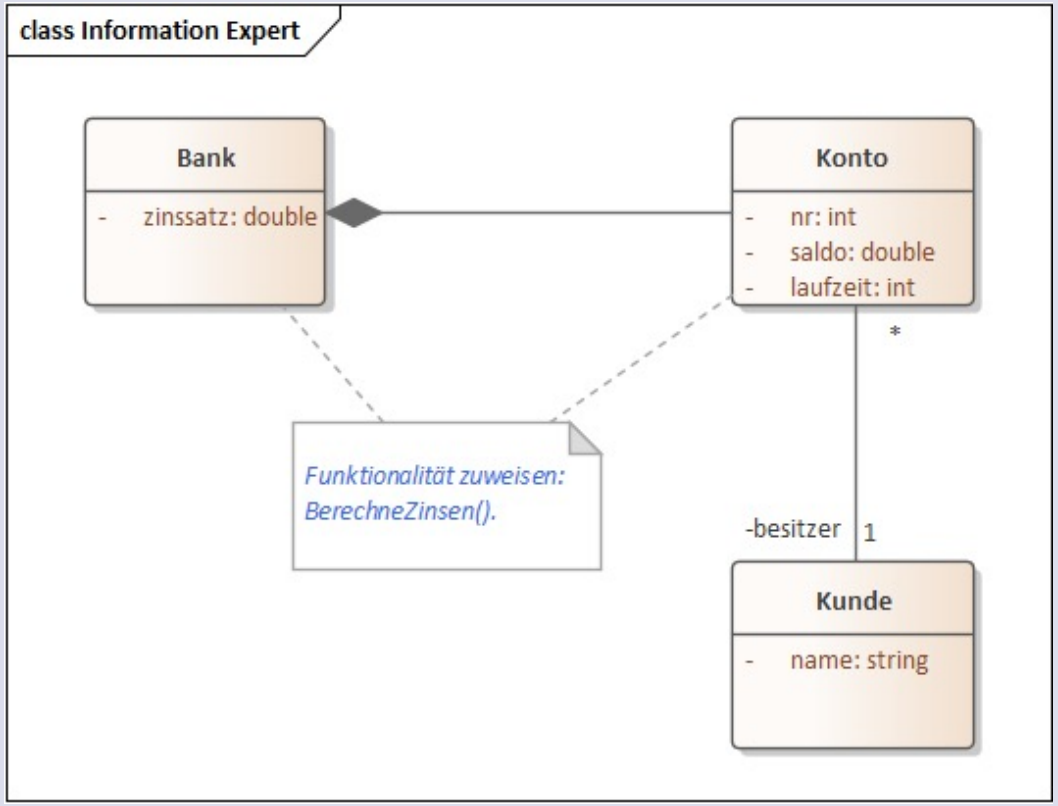
VERANTWORTLICHKEITSMUSTER (GRASP)

Objektorientierte Analyse und Design
GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.
- Denken in Verantwortlichkeiten:
  - **Doing** (Tun)
    - Objekt erstellen oder Berechnung
    - Aktionen anderer Objekte anstoßen, steuern oder kontrollieren
  - **Knowing** (Wissen)
    - Eingekapselte Daten kennen
    - Verwandte Objekte kennen
    - Dinge kennen, die es ableiten oder berechnen kann.
- Insgesamt 9 Muster:
  - Controller
  - **Information Expert**
  - **Creator**
  - Indirection
  - **Low coupling** (Geringe Kopplung)
  - **High cohesion** (Hoher Zusammenhalt)
  - **Pure Fabrication** (Kunstgebilde)
  - Polymorphismus
  - Protected Variations
    - Don‘t talk to strangers
- Command Query Separation

Objektorientierte Analyse und Design
Muster: Information Expert

- Prinzip:
  - Man übertrage eine gegebene Aufgabe/eine Verantwortlichkeit auf diejenige Klasse, die das notwendige Wissen besitzt!
- Hintergrund:
  - Man hat einerseits eine Vielzahl von Klassen und andererseits eine Vielzahl zu vergebender Aufgaben/Verantwortlichkeiten.



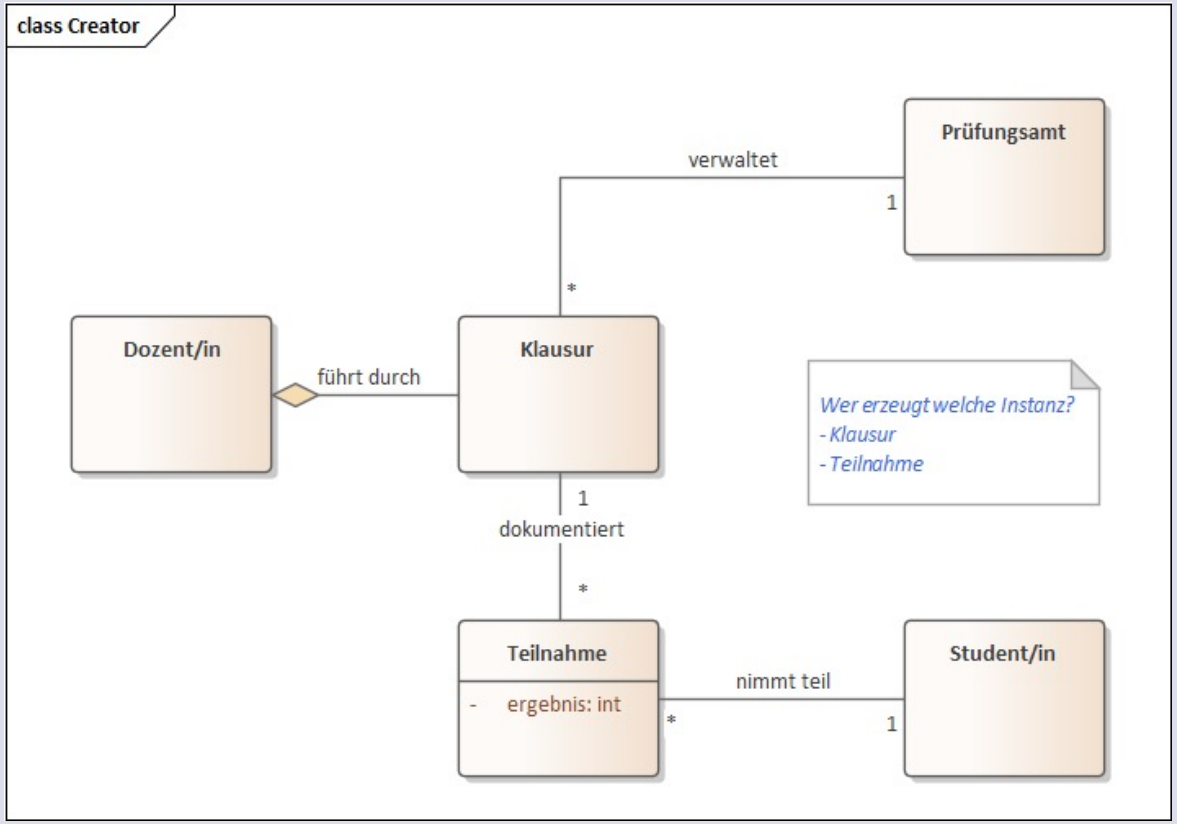
Objektorientierte Analyse und Design
Muster: Creator

Prinzip:

- Die Aufgabe, Objekte einer Klasse A zu erzeugen, wird an eine Klasse B übergeben, wenn
  - B enthält A (Komposition)
  - B steht in enger Beziehung zu A
  - B besitzt das notwendige Wissen (Initialisierung), um A zu erzeugen.

Hintergrund:

- Objekterzeugung ist daher eine wichtige Aufgabe; die Zuständigkeit dafür sollte sorgfältig vergeben werden.

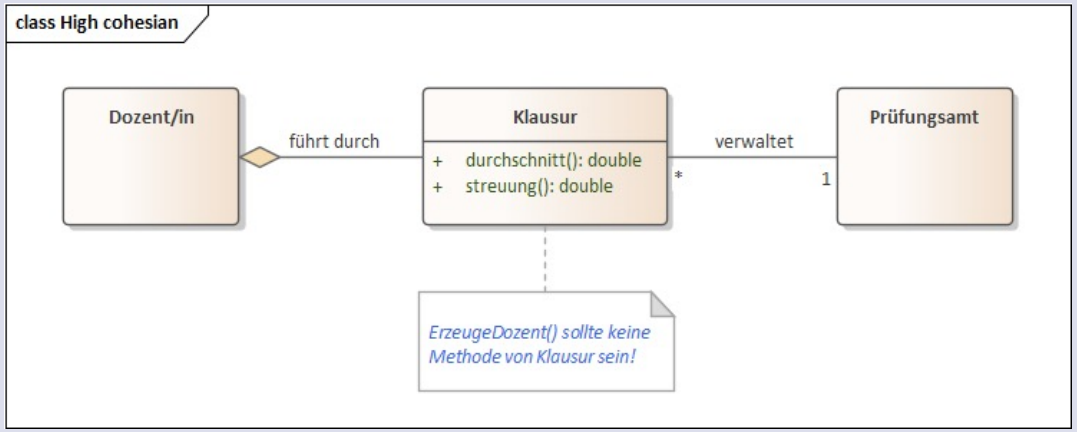


Objektorientierte Analyse und Design
Muster: Low Coupling

- Prinzip:
  - Aufgaben unter den Klassen so verteilen, dass die Abhängigkeiten unter den Klassen möglichst gering sind!
- Hintergrund:
  - Klassen sollten möglichst isoliert sein, denn dadurch werden Entwicklung, Test, Verständnis Wiederverwendbarkeit der Klassen erleichtert.

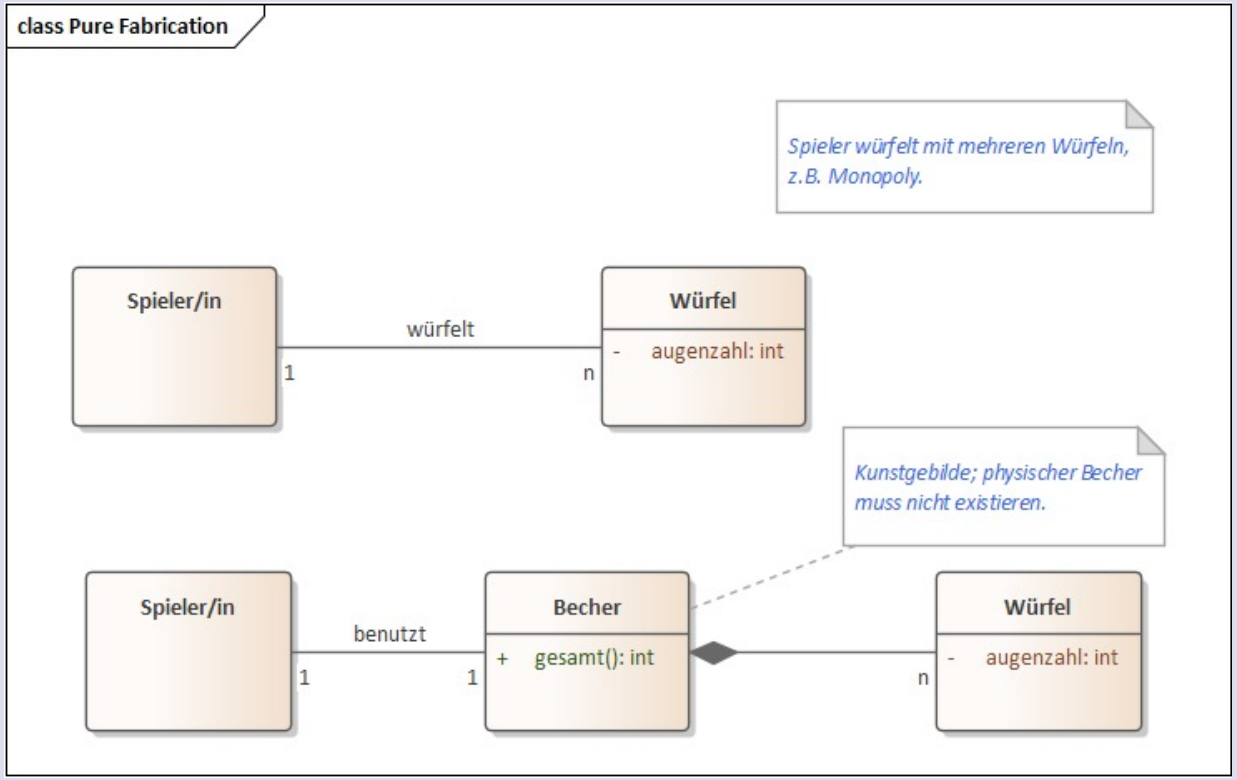
Objektorientierte Analyse und Design
Muster: High Cohesion

- Prinzip:
  - Die Verantwortlichkeiten, die einer Klasse übertragen werden, sollten ähnlich oder zueinander verwandt sein.
- Hintergrund:
  - Klassen, die unterschiedliche Aufgaben erfüllen, sind schwierig zu verstehen, zu warten und wiederzuverwenden.



Objektorientierte Analyse und Design
Muster: Pure Fabrication

- Prinzip:
  - Falls einer Klasse bestimmte Aufgaben übertragen wird und dadurch der hohe Zusammenhalt verletzt wird, so sind Aufgaben in eine Kunstklasse auszulagern.
- Hintergrund:
  - Dieses dient zur Auflösung eines Konfliktes zwischen natürlicher Modellierung und hohem Zusammenhalt.



Objektorientierte Analyse und Design
Muster: Don’t Talk to Strangers

- Prinzip:
  - Gibt es eine Kopplung von 3 Klassen A mit B und B mit C, dann sollte A nur indirekt, d.h. über die Klasse B, mit C kommunizieren.
- Hintergrund:
  - Änderungen/Erweiterungen an der Klasse C haben keine Auswirkungen auf die Klasse A.

- **Faustregeln**, zu welchen anderen Objekten Nachrichten geschickt sollten: zu
  - dem `this/self` Objekt,
  - einem Attribut von `this`
  - einem Objekt, das Parameter einer Methode ist,
  - einem Element eines Containers, welcher ein Attribut von `this` ist
  - einem Objekt, das innerhalb einer Methode erzeugt wurde.

Objektorientierte Analyse und Design
Muster: Don’t Talk to Strangers 2

- Ein Objekt sollte nicht eine Nachricht zu einem Objekt senden, dessen Adresse (Zeiger, Referenz) es als Rückgabewert eines Methodenaufruf erhalten hat.
- …auch bekannt als „**Law of Demeter**“
- https://www.youtube.com/watch?v=OMIZt2W_T_I.