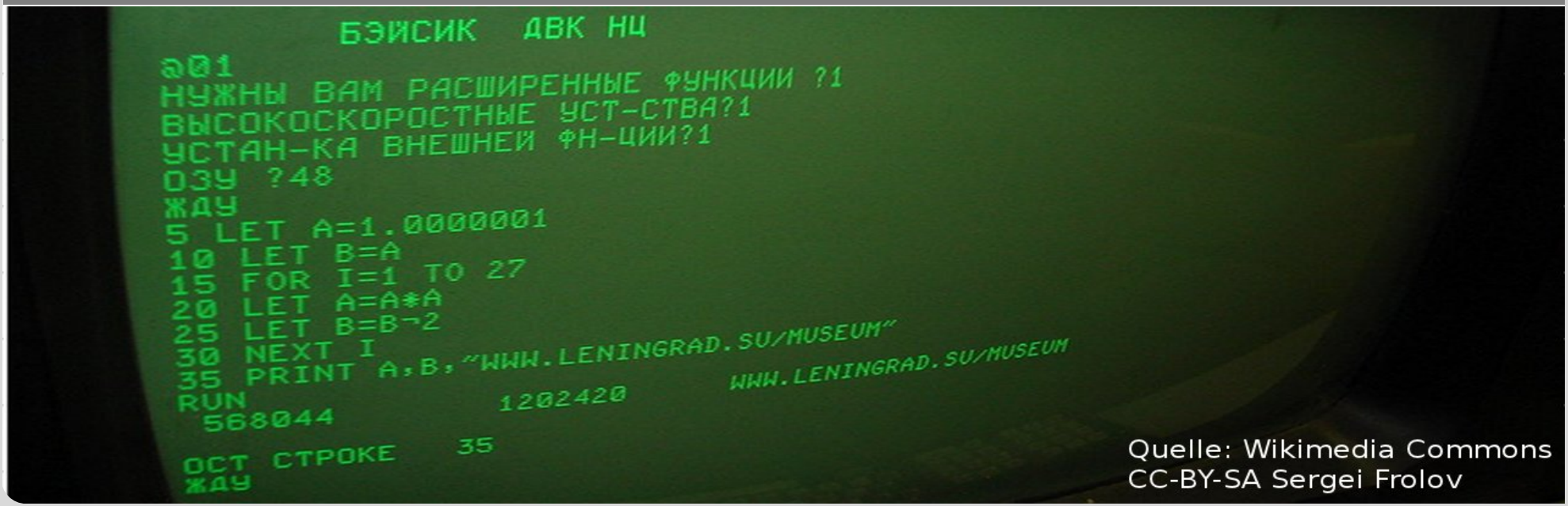
Karlsruher Institut für Technologie

C++ Workshop

11. Block, 13.07.2012

Markus Jung, Robert Schneider | 13. Juli 2012



Gliederung

1. Mehr zu exceptions
   - Exception Guarantees

Exception Guarantees

Sind nötig bei:

- Funktionen die Objekte verändern
- Memberfunktionen von Klassen
- Konstruktoren
- Destruktoren

Mögliche Stufen:

1. „exception unsafe“: ungültige pointer, memleaks, unlogische (inkonsistente) zustände von objekten
2. „basic/weak guarantee“: keine memleaks, objekt kann noch benutzt werden, zustand sicher und wohldefiniert aber (dem Nutzer zunächst) unbekannt
3. „strong guarantee“: wenn eine exception geworfen wurde, wurde nichts verändert
4. „no-throw guarantee“: es wird keine exception geworfen

Stack Unwinding

```
1  void blub()
2  {
3          MyClass obj;
4          MyClass2 obj2;
5          MyClass3 * obj3 = new MyClass3();
6          throw std::runtime_error("kat cilled");
7          MyClass4 obj4;
8  }
9
10 try{
11     MyClass1 obj1;
12     blub();
13 } catch(...) {
14 }
15
16
17     MyClass1::MyClass1()
18     MyClass::MyClass()
19     MyClass2::MyClass2()
```

strong guarantee

```
1  class MyClass{
2      int * arr;
3  public:
4      MyClass(int size)
5      {
6          arr = nullptr;
7          resize(size);
8      }
9      void resize(int size)
10     {
11         if(size <= 0) {
12             throw std::runtime_error("invalid_size");
13         }
14         delete[] arr;
15         arr = new int[size];
16     }
17 };
```

Destruktor

Der Destruktor soll keine Exception werfen (no-throw guarantee).
Theoretisch kann ein Destruktor ohne Probleme eine Exception werfen.
Wenn jedoch beim Bearbeiten einer Exception ein Objekt gelöscht wird, und dessen Destruktor eine Exception wirft, dann terminiert das progamm.
Im Destruktor dürfen Funktionen, die Exceptions werfen, aufgerufen werden, solange die Exceptions abgefangen werden.

Konstruktor

Der Konstruktor sollte dann eine Exception werfen, wenn das Objekt nicht in einen gültigen Zustand versetzt werden kann.
Allokierter Speicher muss vorher freigegeben werden.
Veränderte andere Objekte müssen vorher in den Ausgangszustand zurück versetzt werden.
strong guarantee

```
1
2  Werfer * test = nullptr;
3  try{
4      test = new Werfer();
5      assert(test != nullptr);
6  } catch(...) {
7      assert(test == nullptr);
8  }
```

2 Workflow

Softwareentwicklungsprozesse

- Strukturierung der Softwareentwicklung
- Definition von Abläufen und Artefakten
- Notwendiges Übel?
  - Implizit schon bei relativ kleinen Projekten
  - Unbedingt für größere Projekte
- Wasserfall-Modell
- Agile Prozesse
  - Extreme Programming
  - Scrum
  - ...

Wasserfall-Modell

- Relativ alt und weit verbreitet
- Fest durchstrukturierter Ablauf
  - Planung (Anforderungsanalyse, Lastenheft)
  - Definition (Festlegung der Projektziele, Pflichtenheft)
  - Entwurf (Diverse UML-Artefakte)
  - Implementierung (Programmcode)
  - Test (gemäß Pflichtenheft)
  - Einsatz und Wartung

Probleme

- Grundannahme: Anforderungen stehen nach Beginn fest
- Grundannahme: „Kunde“ kennt (alle) Anforderungen

  . . . und kommuniziert seine Vorraussetzungen/Annahmen
- Starrer Prozess vs. Kreativität

Agile Softwareentwicklungsprozesse

Das Agile Manifest

- Menschen und Interaktionen sind wichtiger als Prozesse und Werkzeuge
- Funktionierende Software ist wichtiger als umfassende Dokumentation
- Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen
- Eingehen auf Veränderungen ist wichtiger als Festhalten an einem Plan

Quelle: Wikipedia

- Prozesse sind klar strukturiert
- Iterativ mit geringem Planungs-/Entwurfsanteil
- Teamorientiert
- Starke Einbeziehung des „Kunden“
- Bekannte Vertreter
  - Extreme Programming
  - Scrum

3 Defensive Programming

Paranoia!

- Murphys Gesetz: „Whatever can go wrong, will go wrong.“
- „Programming today is a race between software engineers striving to build bigger and better idiot-proof programs and the universe, trying to produce bigger and better idiots. So far, the universe wins.“ [Rich Cook]

Defensive Programmierung?!

Ziele

Verbesserung von Software im Bezug auf:

- Sicherheit
- Stabilität
- Robustheit

- Guter Code ist gut lesbar (und damit überprüfbar)
- Vermeiden von Fehlerquellen
- (Frühzeitige) Erkennung/Behandlung möglicher Fehlerfälle
  - Dazu gehört auch: Fehler nicht vertuschen!
- Schutz von Invarianten
- „never trust the client“

Design by Contract

- „Vertrag“ zwischen Schnittstellen/Klassen/Methoden und Nutzern über deren Verhalten und Verwendung
- Schlüsselelemente: Invarianten
  - Vor- und Nachbedingungen
  - Invarianten von Klassen/Algorithmen
- Weitere Aussagen/Zusicherungen:
  - Zulässige Eingaben, mögliche Ausgaben
  - Sonstige Seiteneffekte
  - Mögliche Fehlerzustände/Exceptions
  - Leistungsgarantien

Design by Contract bedeutet . . .

. . . für den Klassenentwickler

- Seine Vorraussetzungen/Bedingungen zu dokumentieren
  - Und im Sinne defensiver Programmierung auch durchzusetzen!
- Eigenen Code anhand ausgearbeiteter Invarianten zu verifizieren
- Implementierungsdetails zu verbergen (information hiding)
  - Alles von außen Sichtbare gehört zum Vertrag
  - Implementierungen können sich ändern - Die Schnittstelle nicht

. . . für den Nutzer

- Einhaltung der dokumentierten Bedingungen
- Ausschließliche (!) Nutzung auf Basis des Vertrages
  - Nach außen verborgene Implementierungsdetails gehören nicht dazu!

Assertions

- Aussagen die bei einem korrekten Programm immer wahr sein sollen
- Fehlschlagen einer Assertion → Sofortiges Programmende!
- Dokumentieren und überprüfen Invarianten

Verwendung

- Einbindung: `#include <cassert>`
  - Standardmäßig sind Assertions aktiv
  - Deaktivierung: `#define NDEBUG` vor dem `include`

    Besser/Bequemer: Über Compiler-Flag (`-DNDEBUG`)
- Einsatz: `assert(23 == 42);`

Assertions vs. Exceptions

- Exceptions behandeln Fehlerfälle und Vertragsverletzungen
- Assertions überwachen Korrektheit von Annahmen/Vorraussetzungen

4 Dokumentation

Warum Dokumentation wichtig ist

Beispiel: `std::map::insert`

```
pair<iterator,bool> insert(const pair<const Key, T> &x)
```

- Vorbedingungen?
- Mögliche Fehlerzustände?
- Rückgabewerte?
- Seiteneffekte?
- Sonstiges Verhalten?
- Laufzeit?

Doxygen

- Ein Werkzeug zur Dokumentation
- Dokumentation erfolgt mit speziellen Kommentaren im Quellcode
  - Vorteil: Lokalität, Wartbarkeit
  - Verschiedene Stil wählbar (Javadoc, Qt)
- `doxygen` durchsucht Quelltext nach Dokumentation
- Steuerung über ein „doxyfile“ (Suchpfade, Formatierung etc.)
- Ausgabeformate: PDF, HTML, RTF, Manpage . . .

`http://www.stack.nl/~dimitri/doxygen/`

Beispiel

```
1  /**
2   * Returns the color value of the pixel at coordinate (p_x, p_y).
3   *
4   * @param p_x the X coordinate component,
5   *            with 0 <= p_x <= width - 1
6   * @param p_y the Y coordinate component,
7   *            with 0 <= p_y <= height - 1
8   * @return a reference to the color value at pixel (p_x, p_y)
9   * @throws std::out_of_range if the passed coordinates exceed
10  *                           the valid value range
11  */
12 Color24& getPixel(unsigned int p_x, unsigned int p_y);
```

(Javadoc-Style, `JAVADOC_AUTOBRIEF = YES`)

5 Testing

Software-Testing

- Ziel: Fehler finden!
- Fehlerfreiheit kann damit nicht garantiert werden!

White/Glass Box Testing

- Test bezieht Wissen über Funktionsweise der zu testenden Funktionen ein
- Ziel: Möglichst gründliche Prüfung aller Codepfade

Black Box Testing

- Kein Wissen über die Funktionsweise/Implementierung
- Prüfung der dokumentierten Verhaltensweisen (Vertrag!)

Test-Frameworks

- Tests von Hand: Fehleranfällig, langsam und machen keinen Spaß
- Test-Frameworks ermöglichen die Automation von Tests
- Einbindung von Tests in den Workflow!

Grundlagen

- Testcase: Prüft einen spezifischen Aspekt einer Klasse/Methode
- Fixture: Schafft Rahmenbedingungen für Tests
- Testsuite: Eine Menge von Tests die die gleiche Fixture nutzen
- Ablauf (die Schritte 2-4 werden für jeden Test wiederholt):
  1. Initialisierung der Testsuite
  2. Vorbereitung der Fixture
  3. Ausführung Text X
  4. Aufräumen der Fixture
  5. Aufräumen der Testsuite

Test-Driven Development

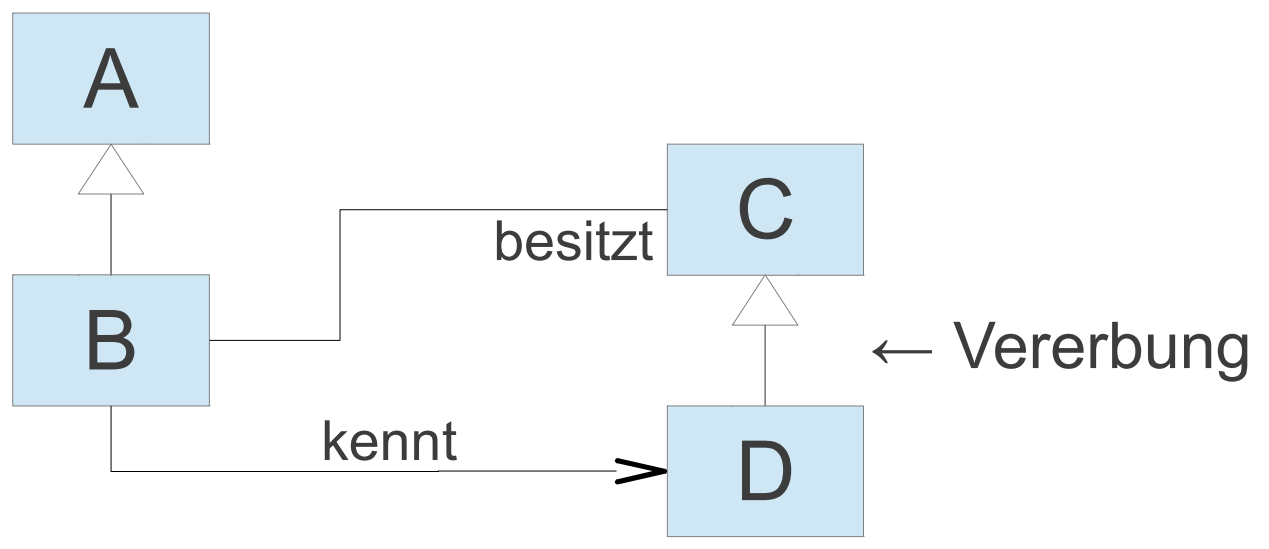
- Klassischer Ablauf: Erst Implementierung, dann Testen
- Test-Driven: Zuerst Testfälle auf Basis der Spezifikation
- Implementierung mit dem Ziel, Testfälle zu bestehen
- Grey Box Testing
- Schnelle Rückkoppelung schon während der Implementierung
  - Wichtige Komponente agiler Entwicklungsprozesse
- Fördert Modularisierung, Flexibilität und Erweiterbarkeit

6 Entwurfsmuster

Das Rad muss nicht neu erfunden werden!

- Manche Probleme tauchen mit schöner Regelmäßigkeit auf
- Wichtigste Kategorien:
  - Entkoppelung von Komponenten
  - Strukturierung von Abstraktionsmöglichkeiten
  - Vereinfachung der Komponentennutzung

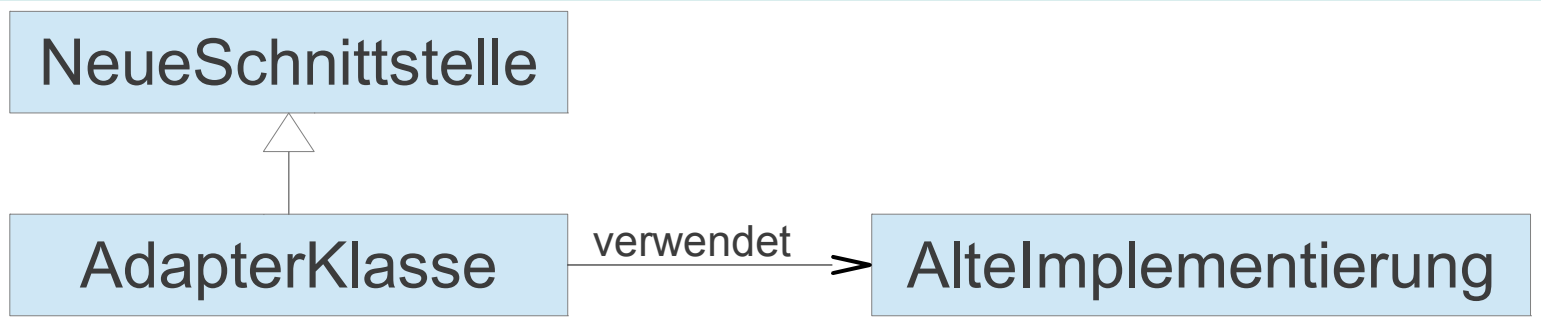
Notation



Wichtige Entwurfsmuster I

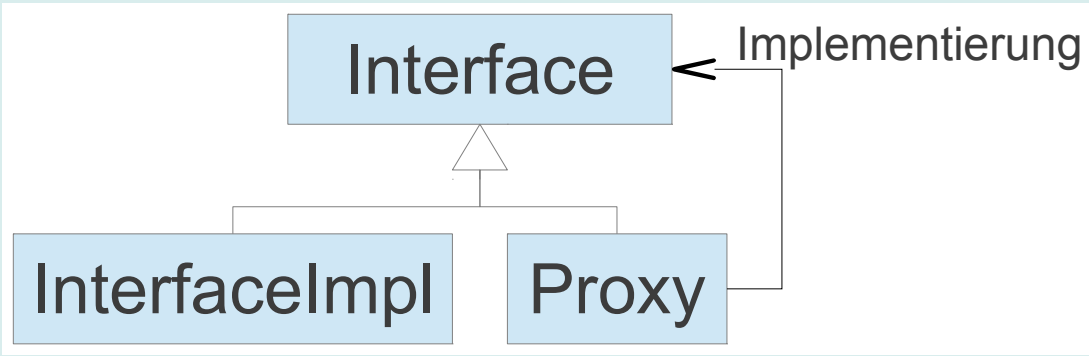
Adapter

Übersetzt inkompatible Schnittstellen



Proxy/Decorator/Wrapper

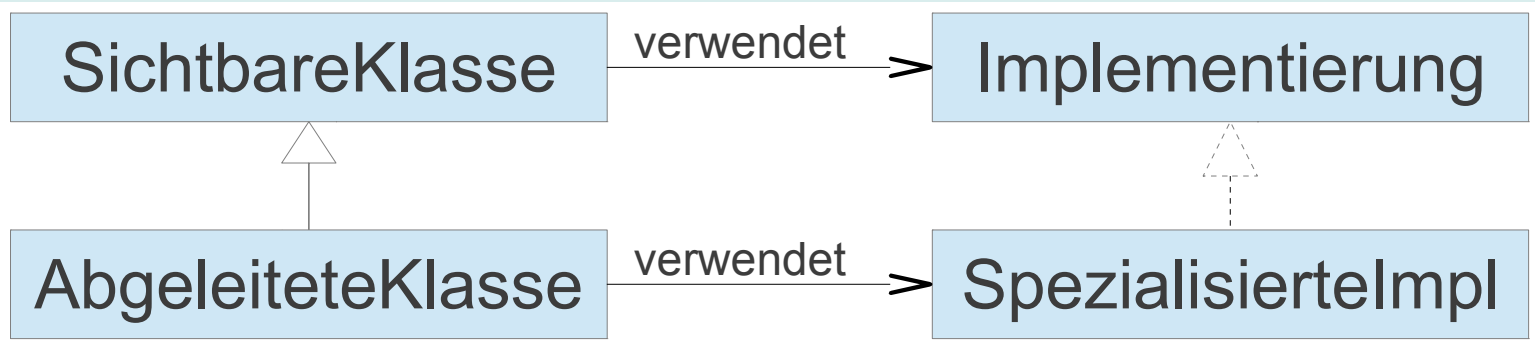
Wird einer anderen Instanz vorgeschaltet und kann diese so kontrollieren oder ihre Funktionalität erweitern



Wichtige Entwurfsmuster II

Bridge/Pointer to Implementation (Pimpl)

Ermöglicht das Entkoppeln von Schnittstelle und Implementierung und erlaubt es, letztere zu verbergen (Information hiding!)



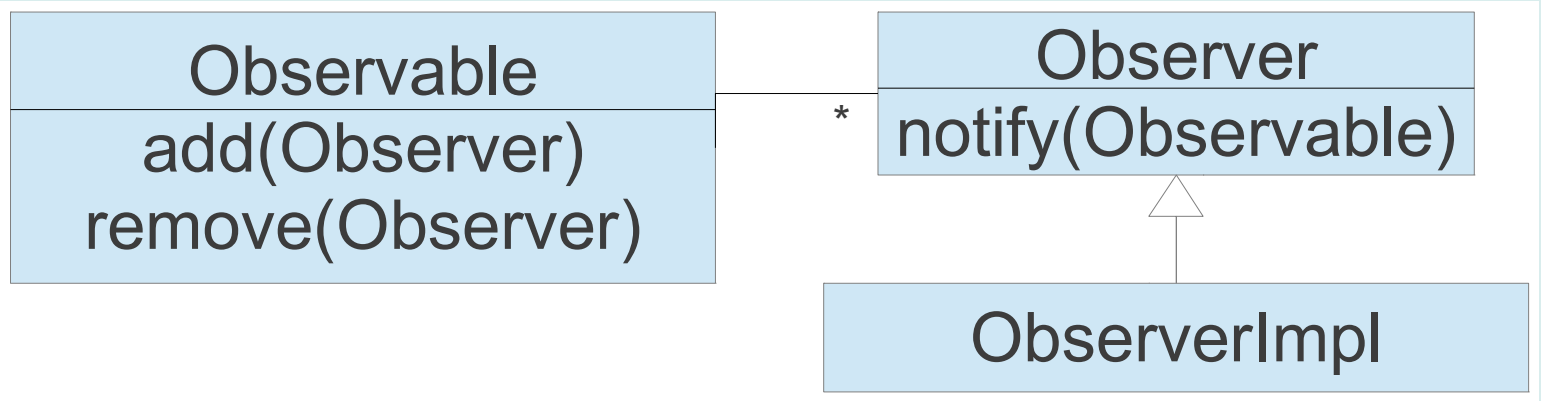
Implementierungsdetails (`private`-Bereich!) werden unsichtbar, es bleibt nur ein Zeiger auf eine anonyme Klasse.

```
1   class fooImpl; // forward declaration
2   class foo {
3       public:
4           // imagine constructor and destructor here ...
5           void bar() {
6               impl->bar();
7           }
8       private:
9           fooImpl *impl; // pointer to implementation
10  }
```

Wichtige Entwurfsmuster III

Observer

- Benachrichtigung anderer Komponenten über Ereignisse
- Die konkreten Implementierungen müssen/sollen sich dabei nicht kennen

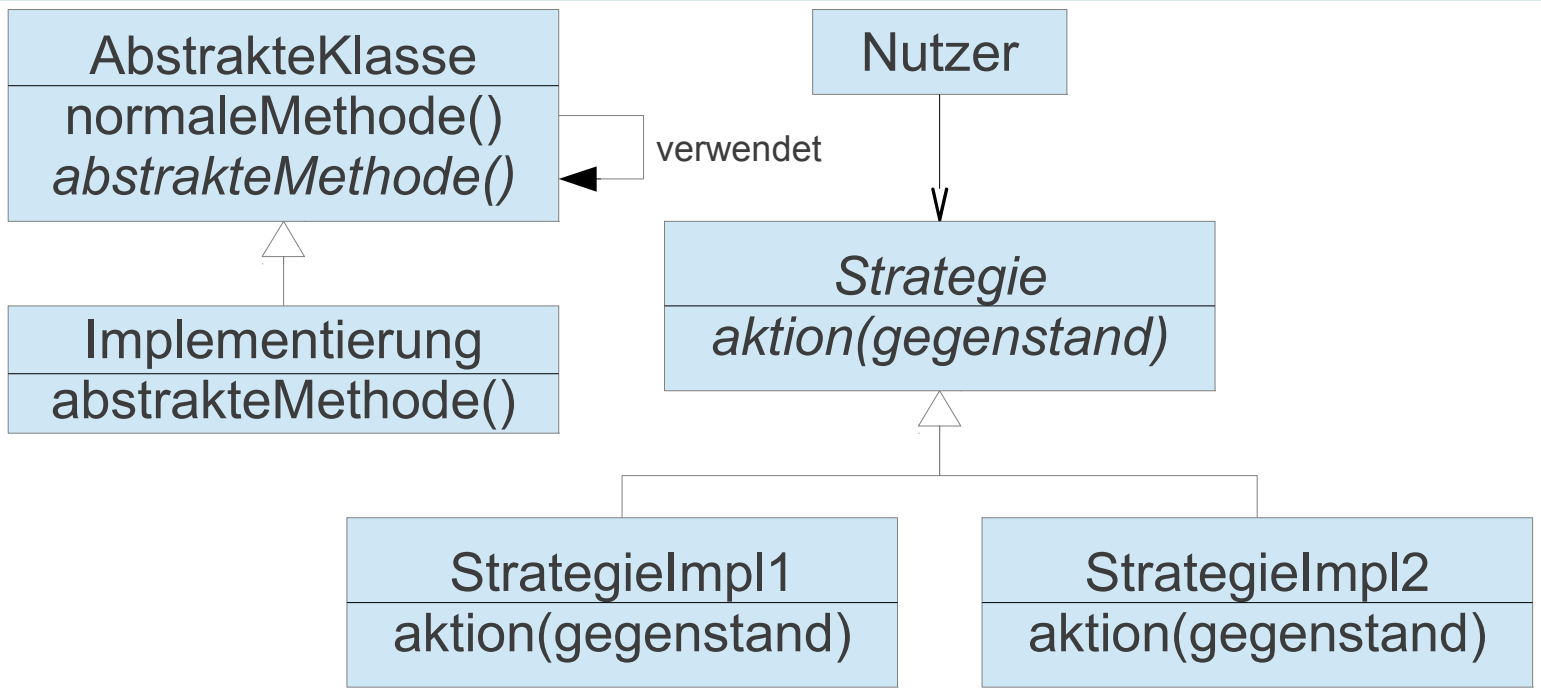


Sonderfall: Vermittler (Mediator) ähnelt dem Observer

Wichtige Entwurfsmuster IV

Template (Schablonenmethode) und Strategie

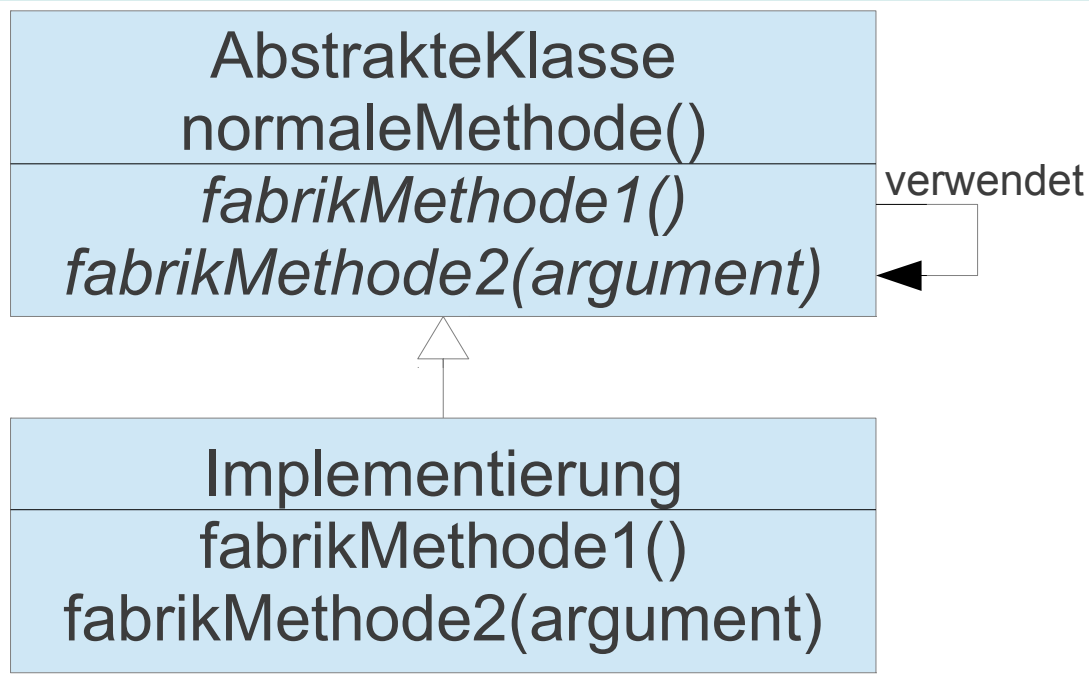
- Schablonenmethoden delegieren die Implementierung einzelner Aspekte an Kindklassen
- Das Strategie-Pattern macht komplette Klassen als Gegenstand einer Implementierung austauschbar



Wichtige Entwurfsmuster V

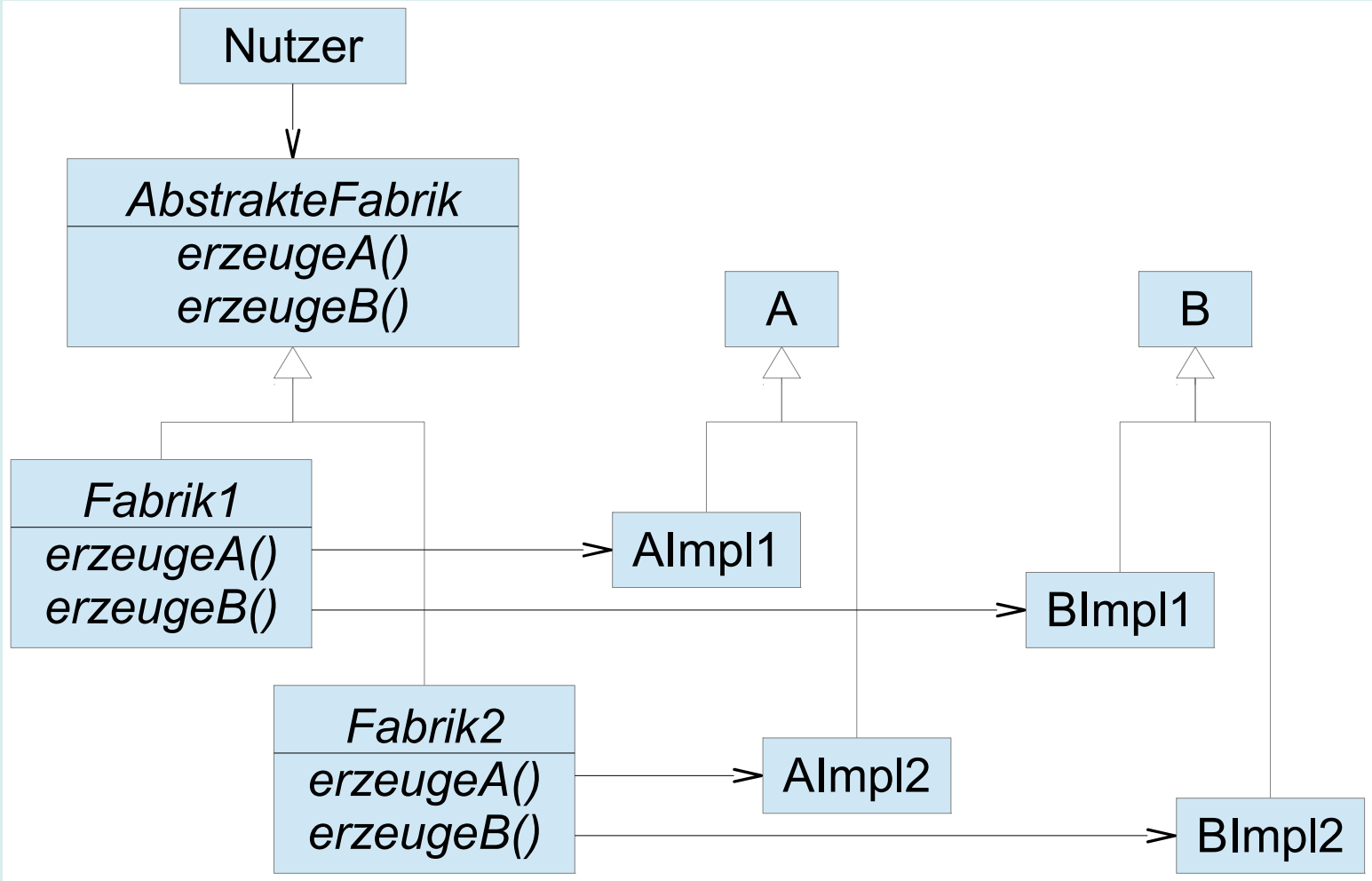
Factory

- Klassen oder Methoden, die neue Objekte erzeugen
- Interpretierbar als Sonderfall von Template und Strategie



Wichtige Entwurfsmuster V

Abstrakte Factory



Wichtige Entwurfsmuster VI

Fassade

Eine Klasse die das (komplizierte) Zusammenspiel innerhalb eines ganzen Subsystems hinter einer einfachen, einheitlichen Schnittstelle verbirgt.

Dummy/Null-Objekt

Eine Klasse die einfach „nichts“ tut. Solche Objekte können aufwändige Sonderfälle ersetzen.

7 Praxis

Praxis!

- Bearbeitung der Aufgaben aus den letzten Workshops

`https://github.com/kit-cpp-workshop/workshop-ss12-11`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`