

Weiterführende Themen in C++

Schulung für Otto Bock Healthcare Products

C++ Core Guidelines

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>



- Richtlinien für die Verwendung von modernem C++ (aktuell C++20 und C++17)
- Die Richtlinien sollen Entwickler unterstützen um:
 - einfacheren,
 - effizienteren,
 - und besser wartbarenCode zu schreiben.
- Verwaltet von Bjarne Stroustrup und Herb Sutter
- In diesem Kurs wird bei den behandelten Themen auf diese Guidelines verwiesen

Weiterführung Einführungskurs



- Implementierung eines Stack Datentyps fixer Größe für die Speicherung von Integerwerten
- Die Daten (pData) sollen dynamisch allokiert werden
- Fehlerbehandlung in pop() und push(int) aktuell vernachlässigen

<i>Stack</i>
<ul style="list-style-type: none">- pData: int*- tos: int- size: int
<ul style="list-style-type: none"><<Create>> Stack(int)+ push(int) : bool+ pop() : int+ isFull() : bool+ isEmpty() : bool

```

1  #include <iostream>
2  #include "stack.hpp"
3
4  Stack::Stack(int size): _size(size), _tos(0), _pData(new int[size]) {}
5
6  Stack::~Stack() {
7      if (_pData != nullptr) {
8          std::cout << "freeing: " << _pData << std::endl;
9          delete[] _pData;
10     }
11 }
12
13 bool Stack::isEmpty() const {
14     return _tos == 0;
15 }
16
17 bool Stack::isFull() const {
18     return _tos == _size;
19 }
20
21 bool Stack::push(int value) {
22     if (isFull()) {
23         return false;
24     }
25     _pData[_tos++] = value;
26     return true;
27 }
28
29 int Stack::pop() {
30     if (isEmpty()) {
31         std::cout << "empty - we will fix this" << std::endl;
32         return 42;
33     }
34     return _pData[--_tos];
35 }

```

- Was passiert bei folgender Verwendung?



```

1  Stack createAndFill(int size) {
2      Stack s {size};
3      for (int i = 0; i < size; i++) {
4          s.push(i);
5      }
6      return s;
7  }
8
9  int main(int argc, char const *argv[]) {
10     Stack s = createAndFill(3);
11     cout << s.pop() << endl;
12     return 0;
13 }

```

Output

```

freeing: 0x664450
...
freeing: 0x664450

```

- Copy Constructor wird verwendet!
- **Zugriff auf freigegebenen Speicher**
- **Shallow Copy → Destruktor gibt Speicher doppelt frei**

Verhindern von Kopien - Copy Elision

https://en.cppreference.com/w/cpp/language/copy_elision

- Compiler **kann** unnötige Kopie von Objekten verhindern
- Seit C++17 gibt es für bestimmte Fälle eine “garantierte Copy Elision”
- **In diesem Codespace für Demonstrationszwecke deaktiviert**
 - Compiler-Flag: “**-fno-elide-constructors**” in der Datei `tasks.json`

```
1 Foo bar() {  
2     return Foo();  
3 }  
4  
5 int main(int argc, char const *argv[]){  
6     Foo f = bar;  
7 };
```

C.21: If you define or =delete any copy, ..., or destructor function, define or =delete them all

- Aufgrund der dynamischen Speicherallokation wird ein benutzerdefinierter Destruktor benötigt
- Es ist naheliegend, dass daher auch ein benutzerdefinierter Kopierkonstruktor und Zuweisungsoperator benötigt wird
- Container der Standard Library verwenden ebenfalls Kopierkonstruktoren
- Wird als “**Rule of three**” bezeichnet

Kopierkonstruktor/Zuweisungsoperator

```
1 Stack::Stack(const Stack& s): _size(s._size), _tos(s._tos), _pData(new int[s._size]) {
2     std::cout << "copy constructor called" << std::endl;
3     std::memcpy(_pData, s._pData, s._size * sizeof(int));
4 }
5
6 Stack& Stack::operator=(const Stack& rhs) {
7     std::cout << "copy assignment called" << std::endl;
8     if (this != &rhs) {
9         delete[] this->_pData;
10        this->_pData = new int[rhs._size];
11        this->_size = rhs._size;
12        this->_tos = rhs._tos;
13        std::memcpy(_pData, rhs._pData, rhs._size * sizeof(int));
14    }
15    return *this;
16 }
```

```
1 Stack createAndFill(int size) {
2     Stack s {size};
3     for (int i = 0; i < size; i++) {
4         s.push(i);
5     }
6     return s;
7 }
8
9 int main(int argc, char const *argv[]) {
10     Stack s = createAndFill(3);
11     while (!s.isEmpty()) {
12         cout << s.pop() << endl;
13     }
14     return 0;
15 }
```

Output

```
copy constructor called
freeing: 0xf04500
2
1
0
freeing: 0xf04540
```

Analyse des Kopierverhalten

```
1  Stack createAndFill(int size) {  
2      Stack s {size};  
3      for (int i = 0; i < size; i++) {  
4          s.push(i);  
5      }  
6      return s;  
7  }  
8  
9  int main(int argc, char const *argv[]) {  
10     Stack s = createAndFill(3);  
11     while (!s.isEmpty()) {  
12         cout << s.pop() << endl;  
13     }  
14     return 0;  
15 }
```

- Benötigt R-Value-Referenz
- Was ist eine R-Value-Referenz bzw. ein R-Value?

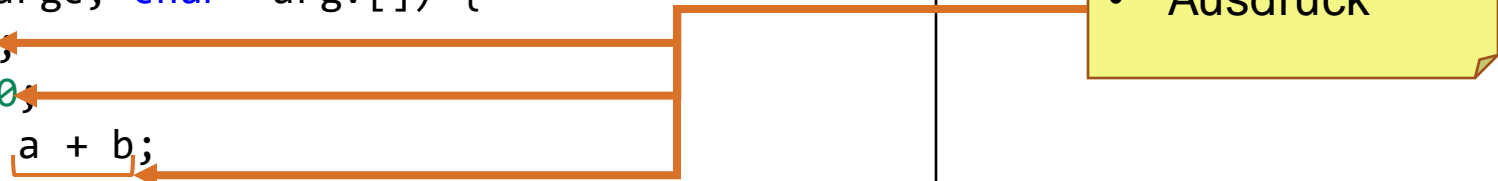
- Kein Speicherfehler
- Das Erstellen einer Kopie ist in diesem Fall unnötig
 - s läuft nach der Kopie aus dem Gültigkeitsbereich
 - s könnte weiterverwendet werden
- Es kann keine Referenz (Stack&) zurückgegeben werden → **Dangling Pointer**
- Statt einer Kopie (Speicherallokation auf dem Heap) soll der existierende Stack zurückgegeben werden?
- **Move Semantik**

Wertekategorien

https://en.cppreference.com/w/cpp/language/value_category

- Eigenschaft eines Ausdrucks (Expression)

```
1 int main(int argc, char* argv[]) {  
2     int a = 5;  
3     int b = 10;  
4     int sum = a + b;  
5 }
```



• Ausdruck

- Beschreibt ob das Resultat der Auswertung eines Ausdrucks
 - ein Wert
 - eine Funktion
 - ein Objekt darstellt.
- Basis für die Entscheidung bzgl. kopieren, moven und erstellen von Objekten
- Zwei Kategorien: L-Value und R-Value

L-Value



- Die Auswertung eines L-Value Ausdrucks resultiert in adressierbaren Objekten/Funktionen
- Das Objekt/Die Funktion kann eindeutig identifiziert werden (hat Identität)
- Lebensdauer in der Regel länger als ein einzelner Ausdruck/Anweisung

L-Value Beispiele

```
1 int main(int argc, char* argv[]) {  
2     int value {5};  
3     int value2 {value};  
4 }
```

- Name einer Variable

```
1 int main(int argc, char* argv[]) {  
2     int value {5};  
3     value += value;  
4 }
```

- Zuweisung

```
1 int main(int argc, char* argv[]) {  
2     std::string str = "hello world";  
3     str.insert(5, 1, ',');  
4 }
```

- Funktionsaufruf mit Referenz-Rückgabetyp

R-Value



- Die Auswertung eines R-Value Ausdrucks
 - resultiert in einem konkreten Wert
 - Oder initialisiert ein Objekt
- Sind nicht identifizierbar
- Lebensdauer ist auf den Gültigkeitsbereich des Ausdrucks begrenzt

R-Value Beispiele

```
1 int main(int argc, char* argv[]) {  
2     int value {5};  
3 }
```

- Literal

```
1 int main(int argc, char* argv[]) {  
2     int value {5};  
3     int value2 {value + 5};  
4 }
```

- Arithmetischer Ausdruck

```
1 int main(int argc, char* argv[]) {  
2     std::string str = "hello world";  
3     int n = str.find("e");  
4 }
```

- Funktionsaufruf mit **nicht** Referenz-Rückgabetyp

R-Value Beispiele

```
1  class Foo {  
2  };  
3  
4  Foo bar(Foo f) {  
5      return f;  
6  }  
7  
8  int main(int argc, char* argv[]) {  
9      Foo f;  
10     &bar(f);  
11 }
```

- Kein Referenz-Rückgabetyt

- R-Value – Zugriff auf Adresse ungültig

Folgendes ist gültig:

```
Foo f2 = bar(f);  
&f2;
```

Output

```
/workspaces/cpp20_devcontainer/example/startup.cpp: In function 'int main(int, char**)':  
/workspaces/cpp20_devcontainer/example/startup.cpp:10:9: error: taking address of rvalue [-fpermissive]  
  10 |     &bar(f);  
    |     ~~~^~~
```

Build finished with error(s).

L-Value/R-Value Referenzen

- Referenz muss initialisiert sein
- Eine L-Value-Referenz (Referenz) wird mit einem L-Value initialisiert

```
1 int main(int argc, char* argv[]) {  
2     int value = 10;  
3     int& v_ref = value;  
4 }
```

```
1 int main(int argc, char* argv[]) {  
2     const int& v_ref = 5;  
3 }
```

- R-Value-Referenz wird mit R-Value initialisiert

```
1 int main(int argc, char* argv[]) {  
2     int&& v_ref = 5;  
3 }
```

Achtung: Typ der Variable ist R-Value Referenz. Wird der Name in einer Expression verwendet ist dies ein L-Value.

- Die Lebensdauer des R-Value wird auf die Lebensdauer der Referenz erweitert
R-Value Referenzen sind veränderbar

Funktionen mit Referenzparametern

```
1 void foo(const std::string& str) {  
2     std::cout << "const lvalue ref: " << str << "\n";  
3 }  
4  
5 int main(int argc, char* argv[]) {  
6     std::string str {"hello"};  
7     foo(str);  
8     foo("world");  
9 }
```

- foo erwartet konstante L-Value Referenz als Parameter
- L- und R-Values sind Zuweisungskompatibel

Output

```
const lvalue ref: hello  
const lvalue ref: world
```


Funktionen mit Referenzparametern

```
1 void foo(const std::string& str) {  
2     std::cout << "const lvalue ref: " << str << "\n";  
3 }  
4  
5 void foo(std::string&& str) {  
6     std::cout << "rvalue ref: " << str << "\n";  
7 }  
8  
9 int main(int argc, char* argv[]) {  
10     std::string str {"hello"};  
11     foo(str);  
12     foo("world");  
13 }
```

- foo ist überladen
- Zwei Varianten:
 - Konstante L-Value Referenz
 - R-Value Referenz

- Welche Funktion wird in Zeile 12 aufgerufen?

Output

```
const lvalue ref: hello  
rvalue ref: world
```

Funktionen mit Referenzparametern

```
1 void foo(const std::string& str) {  
2     std::cout << "const lvalue ref: " << str << "\n";  
3 }  
4  
5 void foo(std::string&& str) {  
6     std::cout << "rvalue ref: " << str << "\n";  
7 }  
8  
9 int main(int argc, char* argv[]) {  
10     std::string&& str {"hello, world"};  
11     foo(str);  
12 }
```

- Welche Funktion wird in Zeile 11 aufgerufen?

- Die Wertekategorie und der Typ eines Objektes sind unabhängig voneinander

- str ist eine R-Value Referenz, in der Verwendung im Ausdruck ist es ein L-Value

Output

```
const lvalue ref: hello, world
```

C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

- Seit C++11 existieren R-Value Referenzen
- R-Value Referenzen ermöglichen u.a. die Verwendung von Move-Konstruktoren und Move-Zuweisung

- Schnittstellen:

```
1 class Foo {  
2     public:  
3         Foo(Foo&& f);  
4         Foo& operator=(Foo&& rhs);  
5 };
```

- Wird als **“Rule of five”** bezeichnet

Move-Konstruktor/Move-Zuweisung

```
1 Stack::Stack(Stack&& s): _size(s._size), _tos(s._tos), _pData(s._pData) {
2     std::cout << "move constructor called" << std::endl;
3     s._pData = nullptr;
4     s._size = 0;
5     s._tos = 0;
6 }
7
8 Stack& Stack::operator=(Stack&& rhs) {
9     std::cout << "move assignment called" << std::endl;
10    if (this != &rhs) {
11        delete[] this->_pData;
12        this->_pData = rhs._pData;
13        this->_size = rhs._size;
14        this->_tos = rhs._tos;
15        rhs._pData = nullptr;
16        rhs._size = 0;
17        rhs._tos = 0;
18    }
19    return *this;
20 }
```

```
1 Stack createAndFill(int size) {
2     Stack s {size};
3     for (int i = 0; i < size; i++) {
4         s.push(i);
5     }
6     return s;
7 }
8
9 int main(int argc, char const *argv[]) {
10    Stack s = createAndFill(3);
11    while (!s.isEmpty()) {
12        cout << s.pop() << endl;
13    }
14    return 0;
15 }
```

Output

```
move constructor called
2
1
0
freeing: 0x784450
```

Implizites “moven”

- Rückgabe von Funktionen (wenn Compiler nicht optimiert - “-fno-elide-constructors”)
- Container der Standard Library unterstützen Move-Semantik:

```
1 int main(int argc, char const *argv[]){  
2     vector<Stack> stacks {};  
3     stacks.push_back(Stack{10});  
4 };
```

Output

```
move constructor called  
freeing: 0x644450
```

- Allgemein werden R-Values implizit “gemoved” wenn der Datentyp die entsprechenden Funktionen anbietet

Explizites “moven”

- Die Funktion `std::move(value)` drückt aus: “Der Wert wird nicht mehr benötigt”
- **ACHTUNG:** `std::move(value)` ist eine Typumwandlung zu einer R-Value-Referenz:

```
1 int main(int argc, char const *argv[]) {  
2     Stack s {3};  
3     s.push(0);  
4     Stack s2 = std::move(s);  
5 }
```

Output

```
move constructor called  
freeing: 0xff4450
```

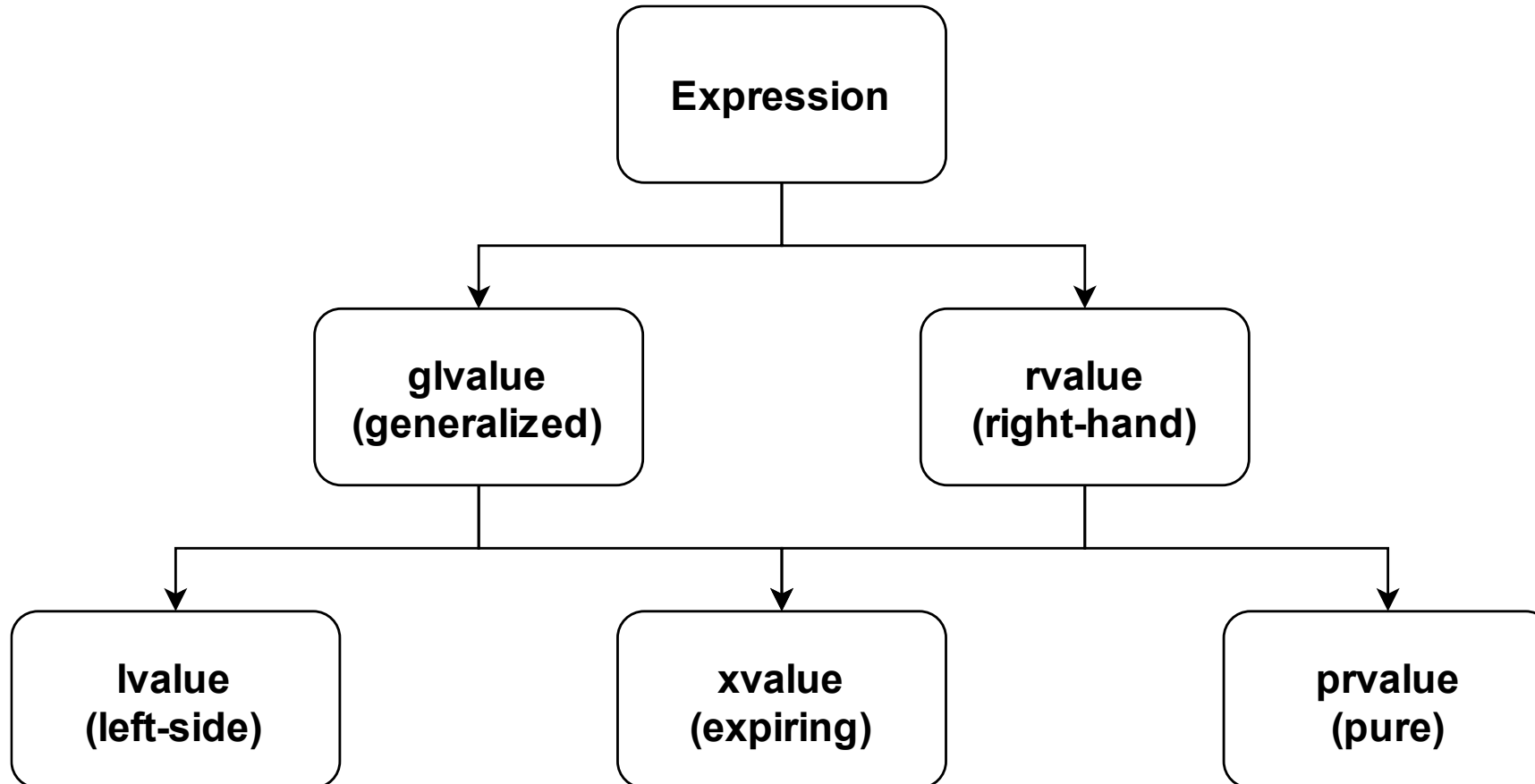
- **ACHTUNG:** `s` ist noch gültig, aber der Speicher wurde von `s2` übernommen.

compiler implicitly declares

user declares		default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
	Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
	Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
	default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
	destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
	copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
	copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
	move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
	move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Figure 1: Generated Member Functions Overview (Howard Hinnants ACCU 2014)

Wertekategorien seit C++11



Analyse der aktuellen Stack Implementation



- Warum wurde der Stack um Kopier- und Move-Konstruktoren erweitert?
 - Dynamischer Speicher benötigt Destruktor (RAII) → Kopierkonstruktor/Zuweisung
 - Benutzerdefinierter Kopierkonstruktor → kein Move-Konstruktor
- Ursache für die Anpassung ist die Verwendung von dynamischen Speicher
 - Verwendung eines Pointer (Raw Pointer)
 - Ein Pointer hat kein Konzept des “besitzen” (“owning”) einer Ressource
- Fehlerhandling für `pop()` nicht vorhanden

Probleme mit Raw Pointers

```
1  int* bar(int* a, int size, int* c) {
2      int* b = new int[size];
3      for (int i = 0; i < size; i++) {
4          b[i] = a[i];
5      }
6      foobar(a, c);
7      return b;
8  }
9
10 int* foo(int* x, int size) {
11     int* a = new int[size];
12     int* b = bar(a, size);
13     int* d = bar(a, size);
14     delete[] a;
15     delete[] x;
16     return d;
17 }
```

- **Probleme:**

- Speicherloch – b
- Speicherloch – a, wenn Allokation in bar fehlschlägt
- Was macht foobar mit a und c?
- Darf x deallokiert werden?
- Aufrufer muss d deallokieren!
- Wer ist für Ressource zuständig?

Smart Pointers



- Verwenden um Besitz einer Ressource zu verdeutlichen
- Kann wie ein regulärer (raw) Pointer verwendet werden
- Dynamic dispatch wird unterstützt
- Smart → Zusätzliches Verhalten für spezifische Anwendungsfälle
 - Beispiel: Ressourcen freigeben
- Smart Pointer in der Standard Library (<memory>):
 - `unique_ptr<T>`
 - `shared_ptr<T>`
 - `weak_ptr<T>`

Unique Pointer

https://en.cppreference.com/w/cpp/memory/unique_ptr



- Exklusiver Besitzer der verwendeten Ressource
- Automatische Freigabe von allokiertem Speicher
 - Benutzerdefinierter “Deleter”
- Hilfsfunktion `std::make_unique<T>(args)` verwenden
 - Leitet args an Konstruktor weiter
- Besitz kann “freigegeben” werden → `release()`
- Move-Only Typ → kann nicht kopiert werden

Shared Pointer

https://en.cppreference.com/w/cpp/memory/shared_ptr

- Geteilter Besitzer der verwendeten Ressource (Array-Typen ab C++17)
- Automatische Freigabe von allokiertem Speicher, wenn letzter Besitzer Gültigkeitsbereich verlässt
 - Benutzerdefinierter “Deleter”
- Hilfsfunktion `std::make_shared<T>(args)` für Initialisierung verwenden
 - Leitet args an Konstruktor weiter
 - Unterstützt Array-Typen ab C++20
- Kann kopiert werden
- `unique_ptr` kann zu `shared_ptr` konvertiert werden

```
1 int main(int argc, char const *argv[]) {  
2     int* values = new int[10];  
3     shared_ptr<int[]> p(values);  
4     shared_ptr<int[]> p2(values);  
5 }
```

• **ACHTUNG: Double Free**

Shared Pointer

https://en.cppreference.com/w/cpp/memory/shared_ptr

- Geteilter Besitzer der verwendeten Ressource (Array-Typen ab C++17)
- Automatische Freigabe von allokiertem Speicher, wenn letzter Besitzer Gültigkeitsbereich verlässt
 - Benutzerdefinierter “Deleter”
- Hilfsfunktion `std::make_shared<T>(args)` für Initialisierung verwenden
 - Leitet args an Konstruktor weiter
 - Unterstützt Array-Typen ab C++20
- Kann kopiert werden
- `unique_ptr` kann zu `shared_ptr` konvertiert werden

```
1 int main(int argc, char const *argv[]) {  
2     int* values = new int[10];  
3     shared_ptr<int[]> p(values);  
4     shared_ptr<int[]> p2(p);  
5 }
```

Shared Pointer - Mehraufwand

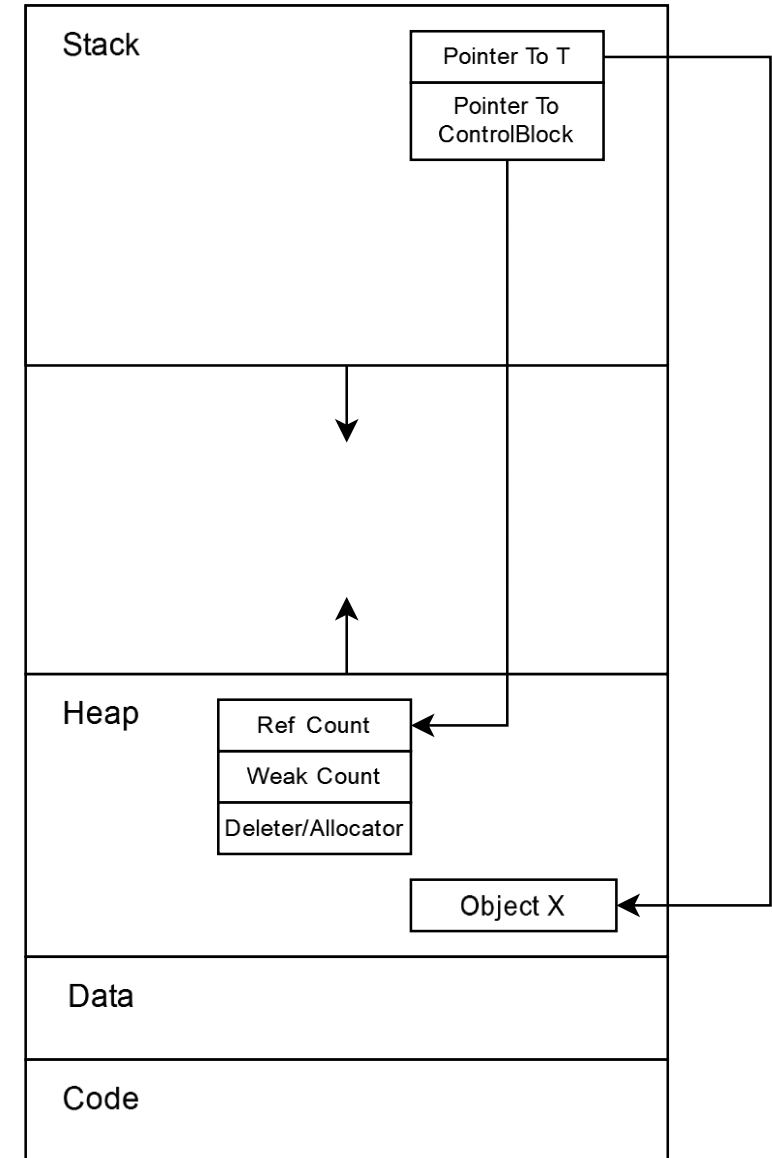
https://en.cppreference.com/w/cpp/memory/shared_ptr

- Zusätzlicher Kontrollblock nötig

```
1 int main(int argc, char const *argv[]) {  
2     std::shared_ptr<int> p = std::make_shared<int>(42);  
3 }
```

- Speicherlayout: Shared Pointer

0xFF



Weak Pointer



- Referenz zu einem `shared_ptr`, die **keine** “Besitz” darstellt
- Erkennt wann die referenzierte Ressource nicht mehr gültig ist
- Keine Dereferenzierung
 - `lock()` - ermöglicht Zugriff auf referenzierten `shared_ptr`
- Vorteile zu Raw Pointer bzgl. Dangling Pointer

Speicherverwaltung mit Smart Pointer



```
1 #include <iostream>
2 #include <memory>
3 #include "stack.hpp"
4
5 Stack::Stack(int size) : _size(size), _tos(0), _pData(std::make_unique<int[]>(size)) {}
6
7 bool Stack::isEmpty() const {
8     return _tos == 0;
9 }
10
11 bool Stack::isFull() const {
12     return _tos == _size;
13 }
14
15 bool Stack::push(int value) {
16     if (isFull()) {
17         return false;
18     }
19     _pData[_tos++] = value;
20     return true;
21 }
22
23 int Stack::pop() {
24     if (isEmpty()) {
25         std::cout << "empty - we will fix this" << std::endl;
26         return 42;
27     }
28     return _pData[--_tos];
29 }
```

- Kein Destruktor benötigt
- Move automatisch vorhanden
- Move wird verwendet
- Kopie benötigt?

Guidelines bezüglich Resource Management



- **R.11: Avoid calling new and delete explicitly**
- **R.20: Use unique_ptr or shared_ptr to represent ownership**
- **R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership**
- **R.22: Use make_shared() to make shared_ptrs**
- **R.23: Use make_unique() to make unique_ptrs**

Beispiel: Bestimmung des Maximum

- Funktion zur Bestimmung des Maximum (`int`):

```
1 int max_int(int a, int b) {  
2     return a > b ? a : b;  
3 }
```

- Es unterscheiden sich lediglich die Datentypen

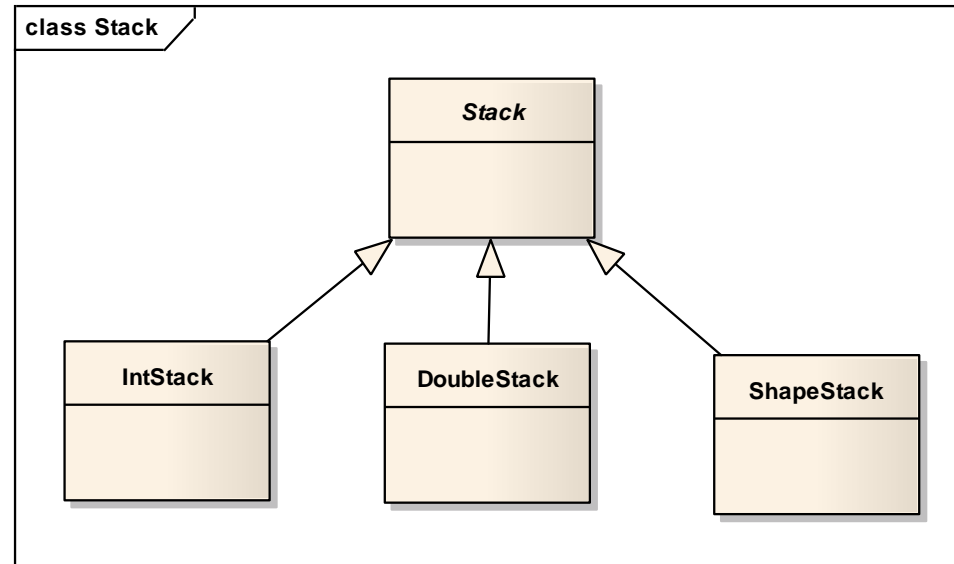
- Funktion zur Bestimmung des Maximum (`std::string`):

```
1 std::string max_string(std::string s1, std::string s2) {  
2     return s1 > s2 ? s1 : s2;  
3 }
```

Wiederverwendung von Funktionalität



- Problem: Die Implementierung einer Funktion/Klasse unterscheidet sich lediglich durch den Typ der Parameter



- Lösungsmöglichkeiten:
 - Verwenden eines void-Pointer (explizite Typumwandlung notwendig)
 - Verwendung von Templates (Gruppe von Funktionen/Klassen) → Generics

Template



- Parametrisierte Beschreibung einer Klasse/Funktion
- Ermöglichen generische Programmierung
 - Type safety
 - Code reuse
 - Instanziierung zur **Compile-Zeit**
- Basis für Standard Template Library → `algorithms`, `containers`, usw.
- Deklaration und Definition in einem Header-File

Syntax:

```
1 template<typename T>
```

Funktionstemplate: Bestimmung des Maximum

```
1 template<typename T>
2 T maximum(T a, T b) {
3     return a > b ? a : b;
4 }
```

- Verwendung des Template mit `int`

```
1 int x = 47;
2 int y = 11;
3 int max = maximum<int>(x, y);
```

- Verwendung des Template mit `string`

```
1 string s1 {"hello"};
2 string s2 {"world"};
3 string max = maximum(s1, s2);
```

- Anstatt `typename` ist auch `class` gültig

- Verkürzte Syntax seit C++20
("T" steht nicht direkt zur Verfügung)

```
1 auto maximum(auto a, auto b) {
2     return a > b ? a : b;
3 }
```

Funktionstemplate: Instanziierung

```
1 int x = 47;  
2 int y = 11;  
3 int max = maximum<int>(x, y);  
4 std::string s1 {"hello"};  
5 std::string s2 {"world"};  
6 std::string max = maximum(s1, s2);
```

- Instanziert eine konkrete Funktion für int

```
1 int maximum(int a, int b) {  
2     return a > b ? a : b;  
3 }
```

- Instanziert eine konkrete Funktion für string

```
1 std::string maximum(std::string a, std::string b) {  
2     return a > b ? a : b;  
3 }
```

Funktionstemplate: Mehrere Template Parameter



```
1 int main(int argc, char const *argv[]){  
2     maximum(47, 1.1);  
3 }
```

Output

```
error: no matching function for call to 'maximum(int, double)'  
15 |     maximum(47, 1.1);  
   |             ~~~~~^~~~~~  
note: candidate: 'template<class T> T maximum(T, T)'  
11 | T maximum(T a, T b) {  
   |   ^~~~~~  
note: template argument deduction/substitution failed:  
note: deduced conflicting types for parameter 'T' ('int' and 'double')  
15 |     maximum(47, 1.1);
```

```
1 template<typename T, typename U>  
2 auto maximum(T a, U b) {  
3     return a > b ? a : b;  
4 }
```


Klassentemplates: Stack

- Stack soll nicht nur für `int` ausgelegt sein, sondern auch andere Datentypen unterstützen

```
1 template<typename T>
2 class Stack {
3     private:
4         ...
5         std::unique_ptr<T[]> _pData;
6     public:
7         T pop() {...}
8         bool push(T value) {...}
9 }
```

- ACHTUNG:** Implementierung im Header-File

```
1 Stack<double> createAndFill(int size) {
2     Stack<double> s {size};
3     for (int i = 0; i < size; i++) {
4         s.push(i);
5     }
6     return s;
7 }
8
9 int main(int argc, char const *argv[]) {
10     Stack<double> s = createAndFill(3);
11     while (!s.isEmpty()) {
12         cout << s.pop() << endl;
13     }
14
15     return 0;
16 }
```

Class Template Argument Deduction (CDAT) seit C++17

https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

- Der Typ eines Klassen-Template-Argumentes kann automatisch ermittelt werden

```
1 int main(int argc, char const *argv[]) {  
2     array a {1, 2, 3, 4, 5};  
3     for (const auto& value: a) {  
4         cout << value << endl;  
5     }  
6 }
```

- array anstatt array<int, 5>

- Nur verwenden wenn das Ergebnis eindeutig ist

```
1 int main(int argc, char const *argv[]) {  
2     vector {"hello", "world"};  
3     vector<std::string> v {"hello", "world"};  
4 }
```

- vector<const char*>

- vector<string>

Class Template Argument Deduction (CDAT) seit C++17

https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

- Der Typ eines Klassen-Template-Argumentes kann automatisch ermittelt werden

```
1 template <typename T, typename U>
2 class MyTuple {
3     public:
4         T v1{};
5         U v2{};
6 };
```

```
1 int main(int argc, char const *argv[]) {
2     MyTuple t {1, 3.2};
3 }
```

Output

```
error: class template argument deduction failed:
  2 |     MyTuple t {1, 3.2};
    |                   ^
```

- Deduction Guide für eigene Klassentemplates

Class Template Argument Deduction (CDAT) seit C++17

https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

- Der Typ eines Klassen-Template-Argumentes kann automatisch ermittelt werden

```
1 template <typename T, typename U>
2 class MyTuple {
3     public:
4         T v1{};
5         U v2{};
6 };
7
8 template <typename T, typename U>
9 MyTuple(T, U) -> MyTuple<T, U>;
```

```
1 int main(int argc, char const *argv[]) {
2     MyTuple t {1, 3.2};
3     cout << t.v1 << " " << t.v2 << endl;
4 }
```

Output

1 3.2

- Nicht mehr nötig ab C++20

- Deduction Guide für eigene Klassentemplates

Non-Type Template Parameter

- Der Typ eines NTTP ist konkret definiert (hier `size_t`)
- Gültig NTTP Typen:
 - Integral Types (bool, char, int)
 - Enumerationen
 - `std::nullptr`
 - Pointer/Referenz auf Objekt
 - Pointer/Referenz auf Funktion
 - Gleitkommazahlen (seit C++20)

```
1  template<typename T, size_t SIZE>
2  class Stack {
3      private:
4          T _data[SIZE];
5          int _tos {0};
6      public:
7
8          bool isEmpty() const {
9              return _tos == 0;
10         }
11
12         bool isFull() const {
13             return _tos == SIZE;
14         }
15
16         void push(T value) {
17             if (isFull()) {
18                 return false;
19             }
20             _data[_tos++] = value;
21             return true;
22         }
23
24         T pop() {
25             if (isEmpty()) {
26                 return 42;
27             }
28             return _data[--_tos];
29         }
30         size_t size() { return SIZE; }
31     };
```



Parameter Pack – Variadic Templates

- Klassen- und Funktionstemplates können über eine variable Anzahl von Parametern verfügen
- Ellipsis (...)
 - Links eines Parameters == Parameter Pack
 - Rechts eines Parameters == Parameter werden “entpackt”

```
1 template <typename Initial, typename... Args>
2 void myPrintf(const Initial& initial, const Args&... args) {
3     std::cout << initial << " ";
4     myPrintf(args...);
5 }
```

- Variable Anzahl von Typen/Parametern

```
1 int main() {
2     myPrintf("hello", 1337, "world", 1.234);
3 }
```

Output

```
error: no matching function for call to 'myPrintf()'
note: template argument deduction/substitution failed:
```

Parameter Pack – Variadic Templates

- Klassen- und Funktionstemplates können über eine variable Anzahl von Parametern verfügen
- Ellipsis (...)
 - Links eines Parameters == Parameter Pack
 - Rechts eines Parameters == Parameter werden “entpackt”

```
1 void myPrintf() {  
2     std::cout << std::endl;  
3 }  
4  
5 template <typename Initial, typename... Args>  
6 void myPrintf(const Initial& initial, const Args&... args) {  
7     std::cout << initial << " ";  
8     myPrintf(args...);  
9 }
```

- Variable Anzahl von Typen/Parametern

```
1 int main() {  
2     myPrintf("hello", 1337, "world", 1.234);  
3 }
```

Parameter Pack – Variadic Templates

- Klassen- und Funktionstemplates können über eine variable Anzahl von Parametern verfügen
- Ellipsis (...)
 - Links eines Parameters == Parameter Pack
 - Rechts eines Parameters == Parameter werden “entpackt”
 - Anzahl der Parameter kann ermittelt werden: **sizeof...(args)**

```
1 template <typename Initial, typename... Args>
2 void myPrintf(const Initial& initial, const Args&... args) {
3     std::cout << initial << " ";
4     if (sizeof...(args) > 0) {
5         myPrintf(args...);
6     }
7 }
```

- **Besonderheit Template:**
Code muss zur Compile-Zeit gültig sein
auch wenn dieser zur Laufzeit nicht
aufgerufen wird

Output

```
error: no matching function for call to 'myPrintf()'
note: template argument deduction/substitution failed:
```


Parameter Pack – Variadic Templates

- Klassen- und Funktionstemplates können über eine variable Anzahl von Parametern verfügen
- Ellipsis (...)
 - Links eines Parameters == Parameter Pack
 - Rechts eines Parameters == Parameter werden “entpackt”
 - Anzahl der Parameter kann ermittelt werden: **sizeof...(args)**

```
1 template <typename Initial, typename... Args>
2 void myPrintf(const Initial& initial, const Args&... args) {
3     std::cout << initial << " ";
4     if constexpr(sizeof...(args) > 0) {
5         myPrintf(args...);
6     }
7 }
```

- **if constexpr**
zur Compile-Zeit ausgewertet
seit C++17

```
1 int main() {
2     myPrintf("hello", 1337, "world", 1.234);
3 }
```

Output

hello 1337 world 1.234

constexpr seit C++11/14

<https://en.cppreference.com/w/cpp/language/constexpr>

- Eine Constant Expression ist ein Ausdruck der zur Compile-Zeit evaluiert werden **kann**
- Der Bezeichner constexpr kann mit Variablen und Funktionen verwendet werden
- Verwendung bei Funktionen oder statischen Attributen sind implizit inline.
- **Kann nicht** mit virtuellen Funktionen verwendet werden (bis C++20)

```
1 constexpr uint64_t square(uint32_t n) {  
2     return n * n;  
3 }
```

```
1 int main(int argc, char const *argv[]) {  
2     constexpr uint64_t result = square(234);  
3     cout << result << endl;  
4 }
```

Constexpr seit C++11/14

<https://en.cppreference.com/w/cpp/language/constexpr>

- Eine Constant Expression ist ein Ausdruck der zur Compile-Zeit evaluiert werden **kann**
- Kann auch mit Templates verwendet werden

```
1 template<uint32_t base, uint32_t n>
2 constexpr uint64_t powN() {
3     uint32_t exp = n;
4     uint64_t result = base;
5     while (--exp > 0) {
6         result *= base;
7     }
8     return result;
9 }
```

```
1 int main(int argc, char const *argv[]) {
2     constexpr uint64_t result = powN<73, 10>();
3     cout << result << endl;
4 }
```

- Kann geprüft werden ob Funktion zur Compile-Zeit ausgeführt wird?

- **std::is_constant_evaluated()**
seit C++20

- **constexpr** Bezeichner seit C++20

Template Type Deduction

https://en.cppreference.com/w/cpp/language/template_argument_deduction

```
1  template<typename T>
2  void foo(T& x);
```

```
1  int main() {
2      const int a = 47;
3      int b = 11;
4      foo(a); foo(b);
5  }
```

```
1  template<typename T>
2  void foo(T x);
```

```
1  int main() {
2      const int a = 47;
3      int b = 11;
4      foo(a); foo(b);
5  }
```

- Typename = T
- “Parametertyp” = T&

- T = **int**
- foo(a) → x = **const int&**
- foo(b) → x = **int&**
- Selbiges gilt für Pointer

- Typename = T
- “Parametertyp” = T

- T = **int**
- foo(a) → x = **int**
- foo(b) → x = **int**

Template Type Deduction

https://en.cppreference.com/w/cpp/language/template_argument_deduction

```
1  template<typename T>
2  void foo(T&& x);
```

```
1  int main() {
2      const int a = 47;
3      int b = 11;
4      const int& refB = b;
5      foo(a); foo(b);
6      foo(refB); foo(1337);
7  }
```

- foo(1337)
T = **int**
x = **int&&**

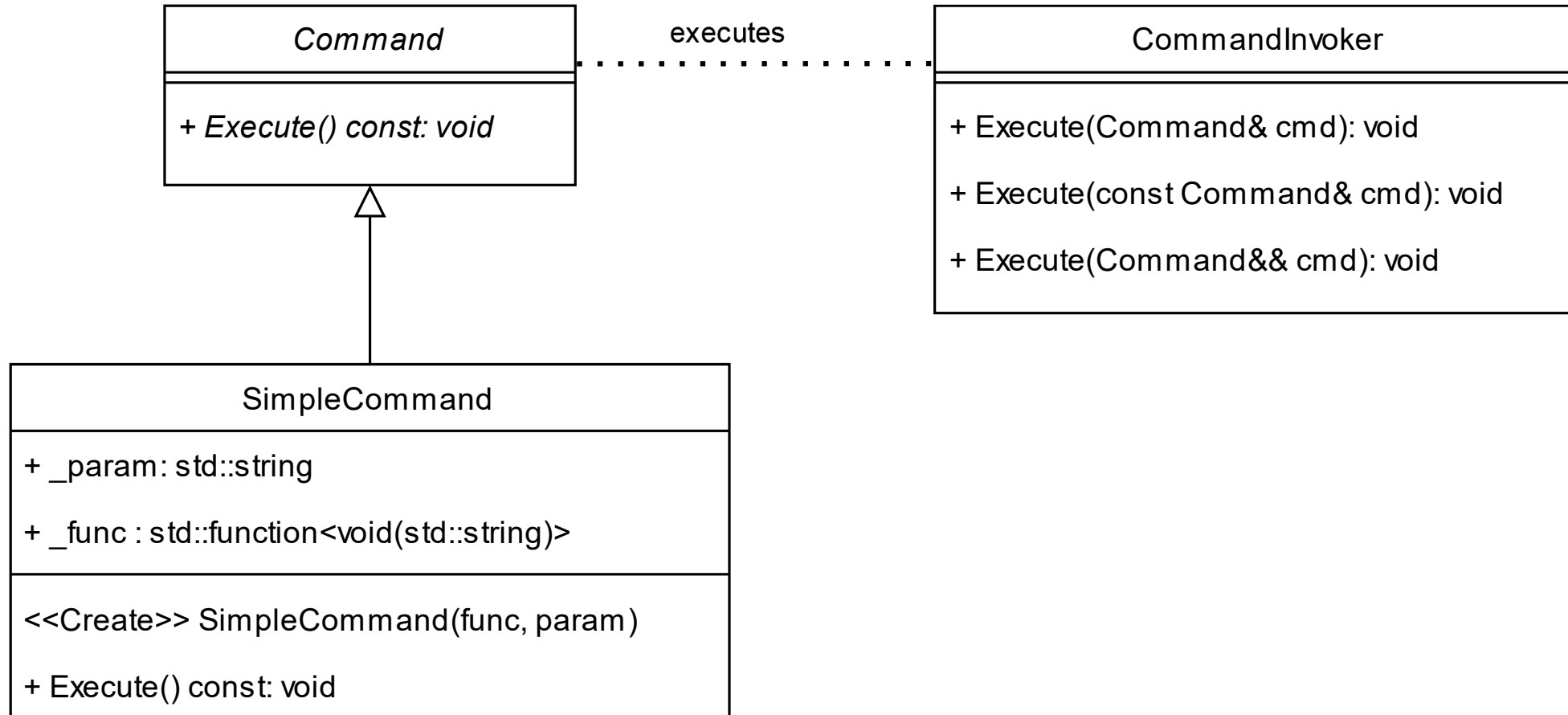
- Typename = T
- “Parametertyp” = T&&
Universal/Forwarding Referenz

- foo(a)
T = **const int&**
x = **const int&**

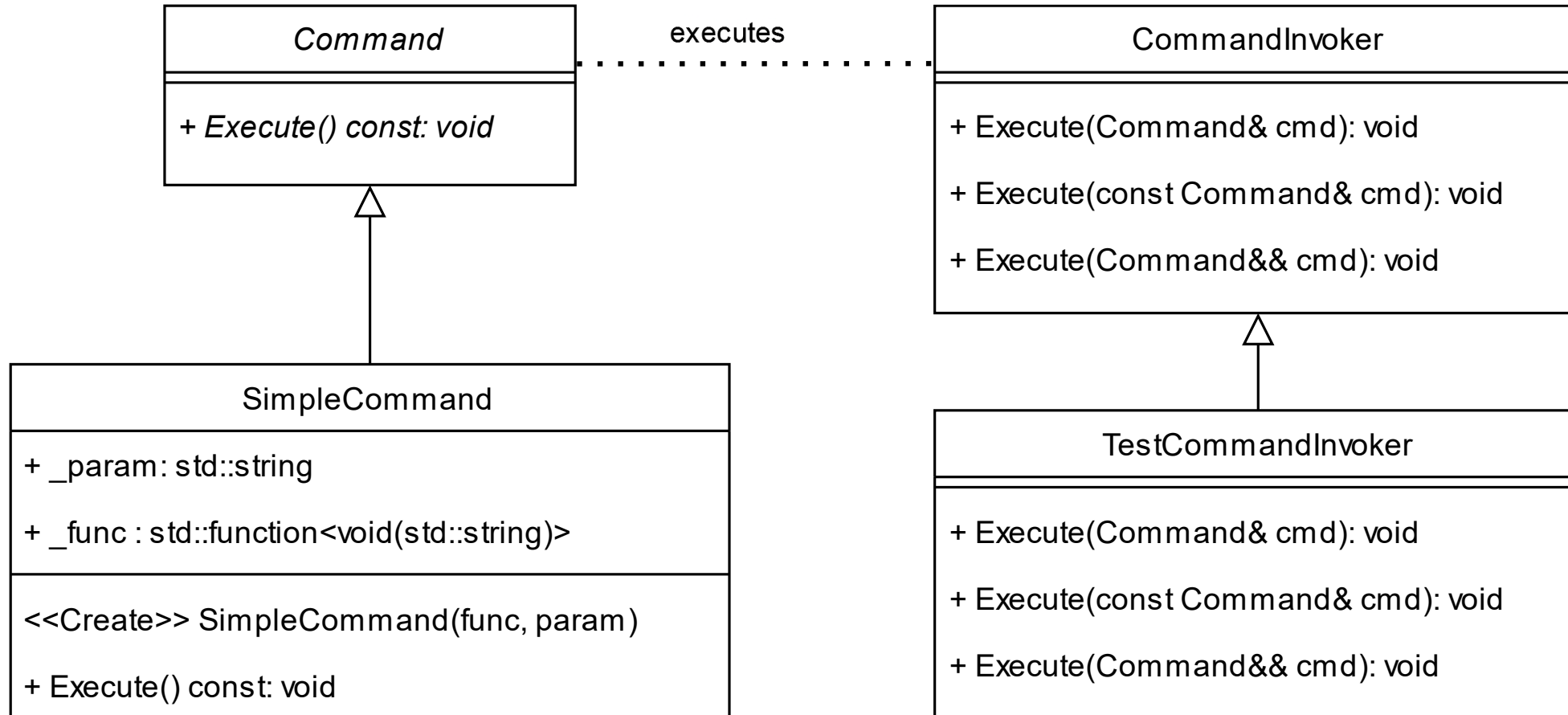
- foo(b)
T = **int&**
x = **int&**

- foo(refB)
T = **const int&**
x = **const int&**

Beispiel: Command Invoker



Beispiel: Command Invoker



Perfect Forwarding

<https://en.cppreference.com/w/cpp/utility/forward>

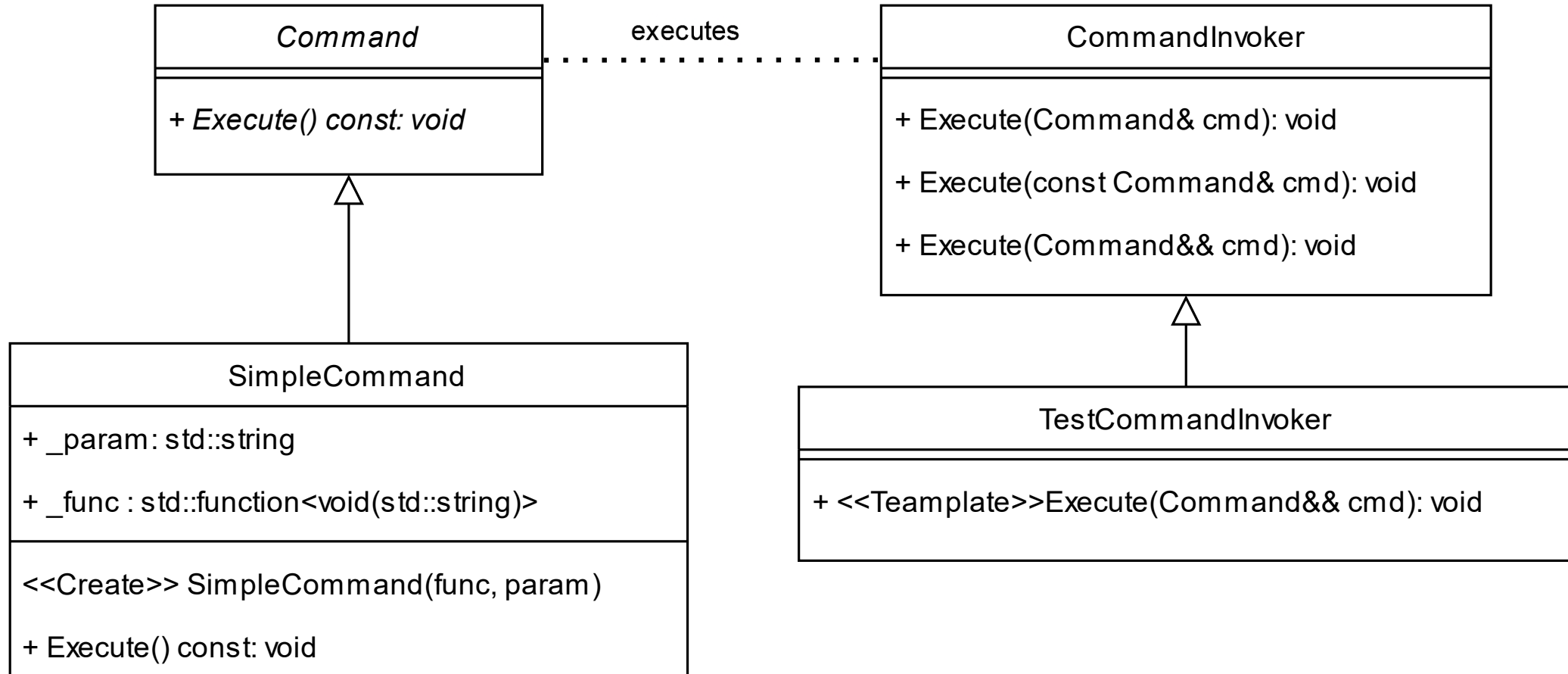
- Eine **&&** Referenz als **Template-Parameter** ist eine Forwarding/Universal-Referenz

```
1 template<typename T>
2     void foo(T&& param) {
3         bar(std::forward<T>(param))
4     };
```

- Eine Forwarding-Referenz kann beim Weiterleiten **die Wertekategorie erhalten**
- **std::forward<T>** verwenden um Parameter weiterzuleiten
- **Funktioniert nur im Kontext eines Template-Parameters**

Beispiel: Command Invoker – Perfect Forwarding

<https://en.cppreference.com/w/cpp/utility/forward>





```
1 class CommandInvoker {
2     public:
3         virtual void Execute(Command& command) {
4             std::cout << "CommandInvoker::Execute(Command&)" << std::endl;
5             command.Execute();
6         }
7
8         virtual void Execute(const Command& command) {
9             std::cout << "CommandInvoker::Execute(const Command&)" << std::endl;
10            command.Execute();
11        }
12
13        virtual void Execute(Command&& command) {
14            std::cout << "CommandInvoker::Execute(Command&&)" << std::endl;
15            command.Execute();
16        }
17 };
```

```
1 class TestCommandInvoker : public CommandInvoker{
2     public:
3         template<typename T>
4         void Execute(T&& command) {
5             std::cout << "template TestCommandInvoker(T&&) with forward reference" << std::endl;
6             CommandInvoker::Execute(std::forward<T>(command));
7         }
8 };
```

Perfect Forwarding

<https://en.cppreference.com/w/cpp/utility/forward>

- Forwarding Referenz ohne Template existieren ebenfalls:

```
1  std::string generate() {  
2      return "test";  
3  }  
4  
5  void handle(std::string&& s) {  
6      std::cout << "handle(std::string&&)" << std::endl;  
7  }  
8  
9  int main() {  
10     auto&& result = test();  
11     handle(result);  
12 }
```

Output

```
error: cannot bind rvalue reference of type 'std::string&&' to lvalue of type ...  
note: initializing argument 1 of 'void handle(std::string&&)'
```

Perfect Forwarding

<https://en.cppreference.com/w/cpp/utility/forward>

- Forwarding Referenz ohne Template existieren ebenfalls:

```
1  std::string generate() {  
2      return "test";  
3  }  
4  
5  void handle(std::string&& s) {  
6      std::cout << "handle(std::string&&)" << std::endl;  
7  }  
8  
9  int main() {  
10     auto&& result = test();  
11     handle(std::forward<decltype(result)>(result));  
12 }
```

- Forwarding muss verwendet werden

- Kein Template-Parameter, daher muss Typ mit `decltype` ermittelt werden

Analyse der aktuellen Stack Implementation

- Stack verwendet Smart-Pointer → automatische Ressourcenverwaltung (Rule of Zero)
(C.20: If you can avoid defining default operations, do)
- Stack unterstützt unterschiedliche Datentypen (Klassentemplate)
(T.1: Use templates to raise the level of abstraction of code)
(T.3: Use templates to express containers and ranges)
- Stack kommt ggf. ohne dynamische Speicher aus (NTTP)
- Fehlerhandling für pop() nicht vorhanden

Fehlerbehandlung



- Laufzeitfehler treten unter Umständen an einer Stelle auf an der die nötigen Informationen für die Behebung nicht vorhanden sind
- Möglichkeiten das Auftreten eines Fehlers “weiterzuleiten”:
 - Beenden des Programmes (Reset)
 - Fehlercodes (viel verwendet in C)
 - Exceptions

Fehlerbehandlung - Fehlercodes

- Vorteile

- Oft verwendet in C/C++
- Keine zusätzlichen Fehlerparameter nötig
- Direkte Fehlerprüfung bei bsplw. Funktionsaufruf

- Nachteile

- Kann einfach ignoriert werden
- Ist der Rückgabewert wirklich ein Fehler? }

- Seit C++17 können diese Probleme mit `std::variant` bzw. `std::optional` gelöst werden

- Wenig Information bzgl. des Fehlers
- Weiterleitung der Fehler ggf. nötig
- Resource Leaks

Fehlerbehandlung – std::variant/std::optional



```
1 int divide(int a, int b) {  
2     return a/b;  
3 }
```

```
1 int main(...){  
2     int result = divide(10, 0);  
3 }
```

```
1 bool divide(int a, int b, int& result) {  
2     if (b == 0) {  
3         return false;  
4     }  
5     result = a / b;  
6     return true;  
7 }
```

```
1 int main(...){  
2     int result;  
3     if (divide(10, 0, result)) {  
4         cout << result << endl;  
5     }  
6 }
```

```
1 optional<int> divide(int a, int b) {  
2     if (b == 0) {  
3         return nullopt;  
4     }  
5     return a / b;  
6 }
```

```
1 int main(...){  
2     auto result = divide(10, 0);  
3     if (result.has_value()) {  
4         cout << result.value() << endl;  
5     }  
6 }
```


Fehlerbehandlung – `std::variant/std::optional`

```
1 variant<int, string> divide(int a, int b) {  
2     if (b == 0) {  
3         return "Error: Division by zero";  
4     }  
5     return a / b;  
6 }
```

- Wenig Information bzgl. des Fehlers
- Weiterleitung der Fehler ggf. nötig
- Resource Leaks

```
1 int main(...){  
2     auto result = divide(10, 5);  
3     if (holds_alternative<int>(result)) {  
4         cout << get<int>(result) << endl;  
5     } else {  
6         cout << get<string>(result) << endl;  
7     }  
8 }
```

Fehlerbehandlung – Exceptions



- Vorteile
 - Trennung von Logik und Fehlerbehandlung
 - Weiterleitung von Fehlern
 - Detaillierte Fehlerinformationen
- Nachteile
 - Performanz
 - Real-Time Anforderungen
 - Speicherbedarf

Fehlerbehandlung – Exceptions

- **throw** keyword um Exceptions zu “werfen”
- **try/catch** Keywords um Exceptions zu behandeln

```
1 int divide(int a, int b) {  
2     if (b == 0) {  
3         throw invalid_argument("b cannot be zero");  
4     }  
5     return a/b;  
6 }
```

```
1 int main(int argc, char const *argv[]) {  
2     try {  
3         int result = divide(10, 2);  
4         cout << result << endl;  
5     } catch(const std::invalid_argument& e) {  
6         cerr << e.what() << endl;  
7     }  
8 }
```

- Exception aus
#include <stdexcept>

- Üblicherweise:
throw by value
catch by const reference

Fehlerbehandlung – Exceptions (Stack unwinding)



```
1 class Foo {
2     public:
3         ~Foo() { cout << "Foo destructor" << endl;}
4 };
5
6 unique_ptr<Foo> bar(int x) {
7     unique_ptr<Foo> f = make_unique<Foo>();
8     if (x == 0) {
9         throw invalid_argument("x cannot be zero");
10    }
11    return f;
12 }
```

- **Stack unwinding**
Ressourcen werden
automatisch freigegeben

- **ACHTUNG: Raw pointer**
wird nicht automatisch
freigegeben

Output

```
Foo destructor
x cannot be zero
```

```
1 int main(int argc, char const *argv[]) {
2     try {
3         unique_ptr<Foo> f = bar(0);
4     } catch(const std::invalid_argument& e) {
5         cerr << e.what() << endl;
6     }
7 }
```

Fehlerbehandlung – Exceptions (catch all)

```
1 class Foo {
2     public:
3         ~Foo() { cout << "Foo destructor" << endl;}
4 };
5
6 unique_ptr<Foo> bar(int x) {
7     unique_ptr<Foo> f = make_unique<Foo>();
8     if (x == 0) {
9         throw invalid_argument("x cannot be zero");
10    }
11    return f;
12 }
```

- ... Catch-All-Handler behandelt alle Exceptions

```
1 int main(int argc, char const *argv[]) {
2     try {
3         unique_ptr<Foo> f = bar(0);
4     } catch(...) {
5         cerr << "exception occurred" << endl;
6     }
7 }
```

Output

Foo destructor
exection occurred

Fehlerbehandlung – Exceptions (Klassen)

- Exceptions können auch in Member-Funktionen verwendet werden
- Was passiert mit Ressourcen bei Exceptions im Konstruktor?

```
1 class Bar {  
2     public:  
3         ~Bar() { cout << "Bar destructor" << endl;}  
4 };  
5  
6 class Foo {  
7     Bar* _pData;  
8     public:  
9         Foo() : _pData(new Bar()) {  
10             throw invalid_argument("Foo constructor failed");  
11         }  
12         ~Foo() { cout << "Foo destructor" << endl;}  
13     };
```

- Destruktor wird nicht aufgerufen
→ Memory Leak

Output

Foo constructor failed

Fehlerbehandlung – Exceptions (Klassen)

- Exceptions können auch in Member-Funktionen verwendet werden
- Was passiert mit Ressourcen bei Exceptions im Konstruktor?

```
1 class Bar {  
2     public:  
3         ~Bar() { cout << "Bar destructor" << endl;}  
4 };  
5  
6 class Foo {  
7     unique_ptr<Bar> _pData;  
8     public:  
9         Foo() : _pData(make_unique<Bar>()) {  
10             throw invalid_argument("Foo constructor failed");  
11         }  
12         ~Foo() { cout << "Foo destructor" << endl;}  
13 };
```

- Mit RAII Objekt wird alles korrekt wieder freigegeben

Output

```
Bar destructor  
Foo constructor failed
```

Fehlerbehandlung – Exceptions

- Exceptions können “weitergereicht” werden
- Benutzerdefinierte Exceptions können erstellt werden

```
1 void foo() {  
2     try {  
3         Foo f;  
4     } catch (const FooBarException& e) {  
5         cerr << "before rethrow" << endl;  
6         throw; ←  
7     }  
8 }  
9  
10 int main(int argc, char const *argv[]) {  
11     try {  
12         foo();  
13     } catch (const FooBarException& e) {  
14         cerr << e.what() << endl;  
15     }  
16 }
```

- **throw;**
gibt Exception weiter

Output

```
Bar destructor  
before rethrow  
Foo constructor failed
```


Fehlerbehandlung – Exceptions

- Exceptions können “weitergereicht” werden
- **Benutzerdefinierte Exceptions können erstellt werden**

```
1 class FooBarException: public std::exception {  
2     private:  
3         std::string _message;  
4     public:  
5         FooBarException(std::string message) : _message(message) {};  
6         const char* what() const noexcept override {  
7             return _message.c_str();  
8         }  
9 };
```

Fehlerbehandlung mit Exceptions

```
1  template<typename T, size_t SIZE>
2  class Stack {
3      private:
4          T _data[SIZE];
5          int _tos {0};
6      public:
7
8          bool isEmpty() const {
9              return _tos == 0;
10         }
11
12         bool isFull() const {
13             return _tos == SIZE;
14         }
15
16         void push(T value) {
17             if (isFull()) {
18                 throw std::out_of_range{"Stack is full"};
19             }
20             _data[_tos++] = value;
21         }
22
23         T pop() {
24             if (isEmpty()) {
25                 throw std::out_of_range{"Stack is empty"};
26             }
27             return _data[--_tos];
28         };
29
30
```

```
1  /**
2   * Pushes a value onto the stack.
3   *
4   * @param value The value to be pushed onto the stack.
5   * @throws std::out_of_range if the stack is full.
6   */
7  void push(T value) { ... }
8  /**
9   * Pops the top value from the stack.
10  *
11  * @return The top value of the stack.
12  * @throws std::out_of_range if the stack is empty.
13  */
14  T pop() { ... }
```

- Keine Möglichkeit eine Methode zu markieren, dass eine Exception geworfen wird
→ Dokumentation

Typumwandlung



- Umwandlung des Datentyp eines Ausdrucks zu einem anderen Datentyp
- Beispiele Typumwandlung in C:

```
1 float f = 3.14;  
2 int a = (int)f
```

- Änderung eines Zuweisungskompatiblen Typs: a=3

Typumwandlung



- Umwandlung des Datentyp eines Ausdrucks zu einem anderen Datentyp
- Beispiele Typumwandlung in C:

```
1 float f = 3.14;  
2 int a = (int)f  
3  
4 int size = 10;  
5 int* p = (int*)malloc(sizeof(int) * size);
```

- Änderung der Interpretation des Speichers

Typumwandlung



- Umwandlung des Datentyp eines Ausdrucks zu einem anderen Datentyp
- Beispiele Typumwandlung in C:

```
1 float f = 3.14;  
2 int a = (int)f  
3  
4 int size = 10;  
5 int* p = (int*)malloc(sizeof(int) * size);  
6  
7 const int a = 10;  
8 int* b = (int*)&a;
```

- Entfernen von const
ACHTUNG: Undefiniertes Verhalten

Typumwandlung in C++



- Es existieren vier Arten von Type-Casts:
 - `static_cast<T>(x)`
 - `const_cast<T>(x)`
 - `dynamic_cast<T>(x)`
 - `reinterpret_cast<T>(x)`

Typumwandlung in C++ - static_cast

https://en.cppreference.com/w/cpp/language/static_cast

- Ein `static_cast` führt eine Umwandlung zwischen Typen durch für die eine implizite oder eine benutzerdefinierte Konvertierung existiert

```
1 class Foo {  
2 public:  
3     operator int() const { return 42; }  
4 };  
5  
6 float f = 3.14;  
7 int a = static_cast<int>(f);  
8 int b = static_cast<int>(Foo{});
```

Typumwandlung in C++ - dynamic_cast



https://en.cppreference.com/w/cpp/language/dynamic_cast

- Ein `dynamic_cast` führt eine Umwandlung zwischen Typen innerhalb einer Vererbungshierarchie durch
 - Umwandlung kann aufwärts, abwärts und seitwärts in der Hierarchie erfolgen

```
1 class Base {  
2     public: virtual int foo() = 0;  
3 };  
4  
5 class Derived1 : public Base {  
6     public: int foo() override { return 1; }  
7 };  
8  
9 class Derived2 : public Base {  
10     public: int foo() override { return 2; }  
11 };
```

```
1 Base* b1 = new Derived1();  
2 Derived1* d = dynamic_cast<Derived1*>(b1);  
3 if (d) {  
4     d->foo;  
5 }  
6 Derived2* e = dynamic_cast<Derived2*>(b1);  
7 if (e) {  
8     e->foo;  
9 }  
10 Base* b2 = dynamic_cast<Base*>(d);  
11 if (b2) {  
12     b2->foo;  
13 }
```


Typumwandlung in C++ - const_cast

https://en.cppreference.com/w/cpp/language/const_cast

- Ein const_cast führt eine Umwandlung zwischen Typen mit unterschiedlichen const/volatile Eigenschaften durch

```
1 int x = 42;  
2 const int& crefX = x;  
3 int& refX = const_cast<int&>(crefX);  
4 refX = 10;  
5 std::cout << x << std::endl;
```

- x ist nicht konstant

Output

10

```
1 const int y = 10;  
2 int& yRef = const_cast<int&>(y);  
3 yRef = 20;  
4 std::cout << yRef << std::endl;
```

- ACHTUNG: x ist konstant**
Undefiniertes Verhalten laut Standard

Typumwandlung in C++ - reinterpret_cast

https://en.cppreference.com/w/cpp/language/reinterpret_cast

- Ein `reinterpret_cast` führt eine Umwandlung zwischen Typen durch indem die Interpretierung des zugrundeliegende Bitmuster geändert wird

```
1 struct Packet {  
2     uint16_t crc;  
3     uint8_t version;  
4 };  
5  
6 int main() {  
7     uint8_t data[1024];  
8     receiveData(buffer, 1024);  
9     Packet* packet = reinterpret_cast<Packet*>(buffer);  
10 }
```

Statischer Polymorphismus

- Die aus dem Einführungskurs bekannte Form des Polymorphismus verwendet:
 - Vererbung
 - und Dynamic Dispatch
- Dynamic Dispatch wird zur Laufzeit durchgeführt
 - Overhead durch V-Table
- Statischer (Compile-Zeit) Polymorphismus kann in C++ u.a. durch das CRTP (Curiously Recurring Template Pattern) realisiert werden

Curiously Recurring Template Pattern (CRTP)

<https://en.cppreference.com/w/cpp/language/crtp>

- Template-Parameter ermöglicht Zugriff auf Subklassen

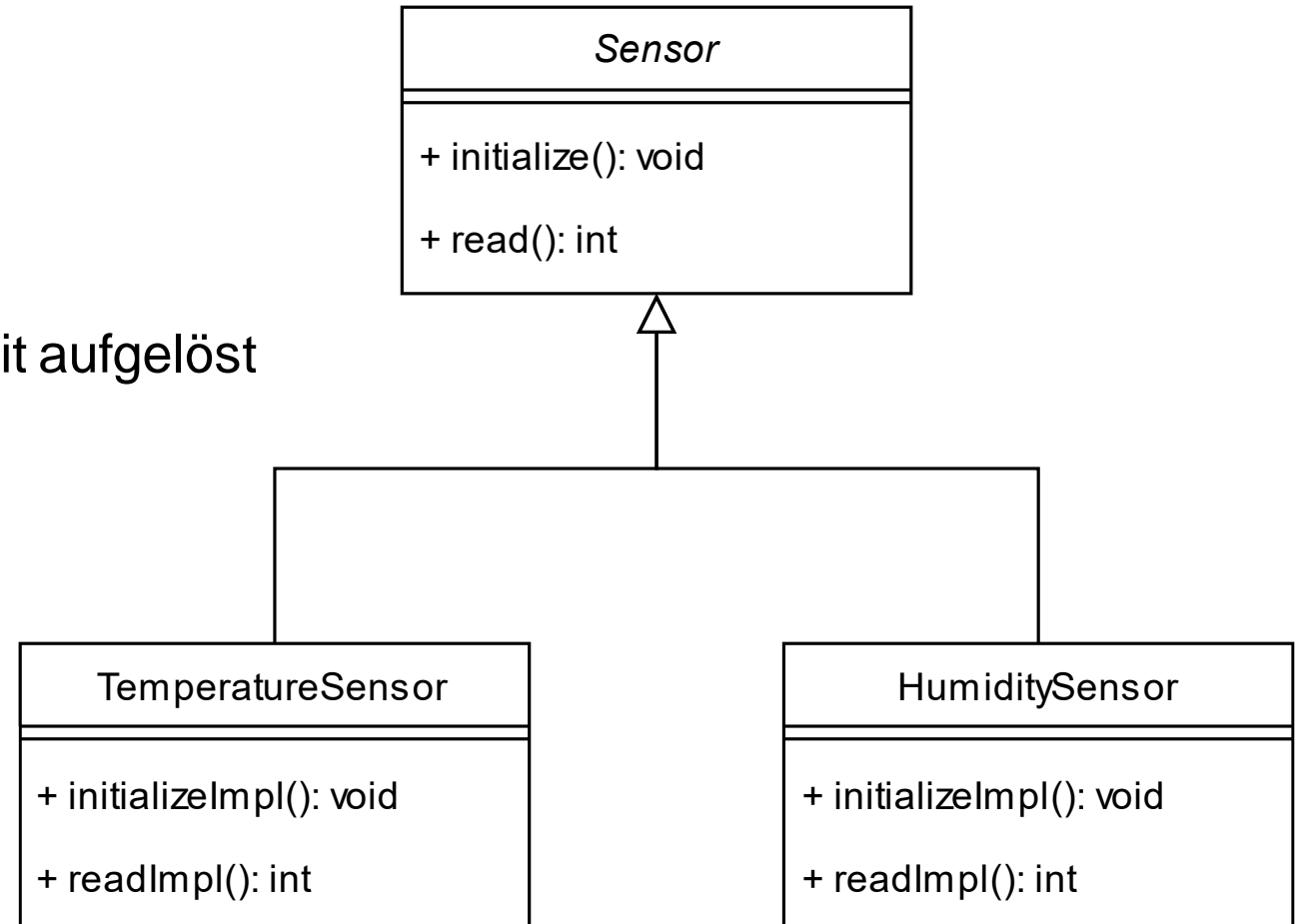
```
1  template<typename Derived>
2  class Base {
3      public:
4          void foo() {
5              static_cast<Derived*>(this)->fooImpl();
6          }
7  };
8
9
10 class Derived1: public Base<Derived1> {
11     public:
12         void fooImpl() {
13             std::cout << "Derived1" << std::endl;
14         }
15 };
```

- Base leitet Funktionsaufruf an Implementierung weiter

- **Derived1** erbt von Base
Derived1 ist Template-Parameter

Curiously Recurring Template Pattern (CRTTP)

- Vorteile:
 - Kein Overhead zur Laufzeit
 - Funktionsaufrufe werden zur Compile-Zeit aufgelöst
→ eventuelle Steigerung der Performanz
- Nachteile
 - Geringere Flexibilität
 - Code-Bloat bei vielen Template-Instanziierungen



Curiously Recurring Template Pattern (CRTP)



```
1 #include <iostream>
2
3 template <typename Derived>
4 class SensorBase {
5     public:
6         void initialize() {
7             static_cast<Derived*>(this)->initImpl();
8         }
9
10        double read() {
11            return static_cast<Derived*>(this)->readImpl();
12        }
13 };
14
15 class TempSensor : public SensorBase<TempSensor> {
16     friend class SensorBase<TempSensor>;
17     private:
18         void initImpl() {
19             std::cout << "TempSensor init" << std::endl;
20         }
21
22         double readImpl() {
23             return 25.3;
24         }
25 };
```

```
1 class HumSensor : public SensorBase<HumSensor> {
2     friend class SensorBase<HumSensor>;
3     private:
4         void initImpl() {
5             cout << "HumSensor init" << endl;
6         }
7
8         double readImpl() {
9             return 55.5;
10        }
11 };
12
13 TempSensor s;
14 HumSensor h;
15 std::cout << s.read() << std::endl;
16 std::cout << h.read() << std::endl;
```

- Keine Super-Klasse
→ kein Interface “über” den Sensoren

Curiously Recurring Template Pattern (CRTP)

```
1 class AbstractSensor {
2 public:
3     virtual void initialize() = 0;
4     virtual double read() = 0;
5     virtual ~AbstractSensor() = default;
6 };
7
8 template <typename Derived>
9 class SensorBase : public AbstractSensor {
10 public:
11     void initialize() override {
12         static_cast<Derived*>(this)->initImpl();
13     }
14
15     double read() override {
16         return static_cast<Derived*>(this)->readImpl();
17     }
18 protected:
19     SensorBase() = default;
20     SensorBase(const SensorBase &) = default;
21     SensorBase(SensorBase &&) = default;
22 };
```

- Problem: Kein Interface/Überklasse für Sensoren
- Lösung: Abstrakte Klasse (AbstractSensor) über CRTP einführen

```
1 void run(AbstractSensor& sensor) {
2     sensor.initialize();
3     std::cout << sensor.read() << std::endl;
4 }
```

Lambdas

<https://en.cppreference.com/w/cpp/language/lambda>

- Anonyme Funktionsobjekte (Functor)
- Generische Lambdas existieren seit C++14
- Syntax:

```
1 [](int i) { return i < 0; }
```

Capture Clause

Parameter list

Body

- **T.40: Use function objects to pass operations to algorithms**

Lambdas



```
1 int main(int argc, char const *argv[]) {  
2     auto square = [](int x) {  
3         return x * x;  
4     };  
5     cout << square(3) << endl;  
6 }
```

- Verwendung wie eine Funktion
- Typ ist nicht bekannt, daher **auto**

Output

9

Lambdas



- Vorteile

- Lokalität
- Immer definiert
- Optimierungspotenzial für Compiler
- Funktionen mit Status

```
1 int main(int argc, char const *argv[]) {  
2     auto square = [](int x) {  
3         return x * x;  
4     };  
5     cout << square(3) << endl;  
6 }
```

- Nachteile

- Debugging wird ggf. erschwert
- Mit Capture Clause nicht kompatibel zu Raw Function Pointer

Lambdas und Funktionen

- Jede Funktion kann als Lambda ausgedrückt werden

```
1 bool biggerThan3(int i) {  
2     return i > 3;  
3 }  
4  
5 int main(int argc, char const *argv[]) {  
6     vector<int> v {-11, 37, 3, 0, 7, 13, 6, 8, -3, -5, 4};  
7     count = count_if(v.begin(), v.end(), biggerThan3);  
8     cout << count << endl;  
9 }
```

- `count_if` erwartet eine Funktion mit Schnittstelle `bool foo(T value)`
- Weitere Funktion für Werte größer als bsplw. 5?

Lambdas und Funktionen

- Jede Funktion kann als Lambda ausgedrückt werden
- Durch die Captures Clause sind Lambdas mächtiger als Funktionen → Lambdas sind Funktionsobjekte

```
1 int main(int argc, char const *argv[]) {  
2     vector<int> v {-11, 37, 3, 0, 7, 13, 6, 8, -3, -5, 4};  
3     int threshold = -1;  
4     int count = count_if(v.begin(), v.end(), [threshold](int i) {  
5         return i > threshold;  
6     });  
7     cout << count << endl;  
8 }  
9
```

Lambdas – Capture Clause

```
1 [](int i) { ... }
```

```
1 int x = 10;  
2 [=](int i) { x+=1; }
```

```
1 int x = 10;  
2 [&](int i) { x+=1; }
```

```
1 int x = 10;  
2 int y = 11;  
3 [=, &y](int i) { x+=1; y+=1; }
```

```
1 int x = 10;  
2 [x](int i) { x+=1; }
```

```
1 [x = 10](int i) { x+=1; }
```

- Kein Capturing
- Alle verwendeten Variablen werden kopiert
- Alle verwendeten Variablen sind Referenzen
- Alle verwendeten Variablen werden kopiert, nur y ist eine Referenz
- Nur x wird kopiert
- Captures können direct initialisiert werden

• Value-Captures (kopierte Variablen) sind per Default read-only

Generische Lambdas

```
1 int main(int argc, char const *argv[]) {  
2     auto min = [] (auto x, auto y) {  
3         return x < y ? x : y;  
4     };  
5     cout << min(string{"hello"}, string{"world"}) << endl;  
6     cout << min(3.1, 4) << endl;  
7     const char* s1 = "hello";  
8     const char* s2 = "world";  
9     cout << min(s1, s2) << endl;  
10 }
```

Output

```
hello  
3.1  
world
```

Lambdas - Zustand

```
1 int main(int argc, char const *argv[]) {  
2     auto foo = [x = 0] () {  
3         while (x < 10) {  
4             cout << x << endl;  
5             x++;  
6         }  
7     };  
8     foo();  
9     foo();  
10 }
```

Output

```
error: increment of read-only  
variable 'x' x++
```

- Value-Captures (kopierte Variablen) sind per Default read-only

Lambdas - Zustand

```
1 int main(int argc, char const *argv[]) {  
2     auto foo = [x = 0] () mutable {  
3         while (x < 10) {  
4             cout << x << endl;  
5             x++;  
6         }  
7     };  
8     foo();  
9     foo();  
10 }
```

Output

```
0  
1  
2  
3  
4
```

- **ACHTUNG:** Mehrfache Ausführung eines Lambda
→ unterschiedliches Ergebnis/Verhalten

Lambdas - Funktionsobjekte

- Syntaktischer Zucker für ein Funktionsobjekt (Functor)

```
1 auto min = [](int x, int y) {  
2     return x < y ? x : y;  
3 };
```

```
1 class LambdaABC {  
2     public:  
3         LambdaABC() {}  
4         int operator() (int x, int y) const {  
5             return x < y ? x : y;  
6         }  
7 };
```

```
1 int x = min(3, 4);
```

```
1 int x = min.operator()(3, 4);
```

- Compiler generiert eine Klasse für jedes Lambda

- Identische Aufrufe

Lambdas - Funktionsobjekte

- Syntaktischer Zucker für ein Funktionsobjekt (Functor)

```
1 double measurment = getMeasurment();  
2 auto scale = [measurment] (double x) {  
3     return x * measurment;  
4 };
```

- Compiler generiert eine Klasse für jedes Lambda

```
1 class LambdaXYZ {  
2     double _measurement;  
3     public:  
4         LambdaXYZ(double measurement)  
5             : _measurement {measurement} {}  
6         double operator() (double x) const {  
7             return x * _measurement;  
8         }  
9 };
```

```
1 double x = scale(3.0);
```

```
1 double x = scale.operator()(3.0);
```

Generische Lambdas - Funktionsobjekte

- Syntaktischer Zucker für ein Funktionsobjekt (Functor)

```
1 auto min = [] (auto x, auto y) {  
2     return x < y ? x : y;  
3 };
```

```
1 class LambdaXYZ {  
2     public:  
3         LambdaXYZ() {}  
4         template<typename T, typename U>  
5         auto operator() (T x, U y) const {  
6             return x * _measurement;  
7         }  
8 };  
9
```

- Compiler generiert eine Klasse für jedes Lambda

```
1 auto x = min(3.1, 4);
```

```
1 auto x = min.operator()<double, int>(3.1, 4);
```

Lambdas – Rückgabewert

- Kann automatisch ermittelt werden
- Unter Umständen ist die Ermittlung nicht eindeutig

```
1 auto halve = [](int x) {  
2     if (x % 2 == 0) {  
3         return x / 2;  
4     } else {  
5         return x / 2.0;  
6     }  
7 };
```

• Rückgabebetyp: **int**

• Rückgabebetyp: **double**

Output

```
error: inconsistent types 'int' and 'double' deduced for lambda return type  
return x / 2.0;
```

Lambdas – Rückgabewert

- Kann automatisch ermittelt werden
- Hinweis für Compiler welcher Rückgabebetyp verwendet werden soll

```
1 auto halve = [](int x) -> double {  
2     if (x % 2 == 0) {  
3         return x / 2;  
4     } else {  
5         return x / 2.0;  
6     }  
7 };
```

```
1 int main(int argc, char const *argv[]) {  
2     cout << halve(10.0) << endl;  
3 }
```

Output

5

SOLID

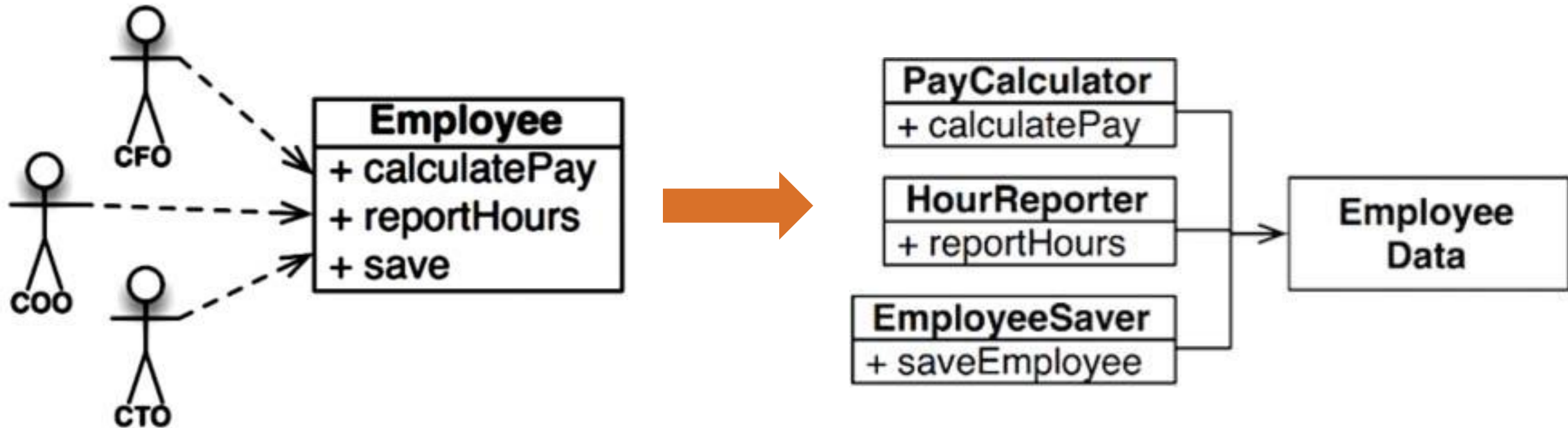


- Single-responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single-responsibility Principle (SRP)

- *“A module should have one, and only one, reason to change”*
- *“A module should be responsible to one, and only one, user or stakeholder.”*
- *“A module should be responsible to one, and only one, actor”*

Single-responsibility Principle (SRP)



Open-Closed Principle (OCP)



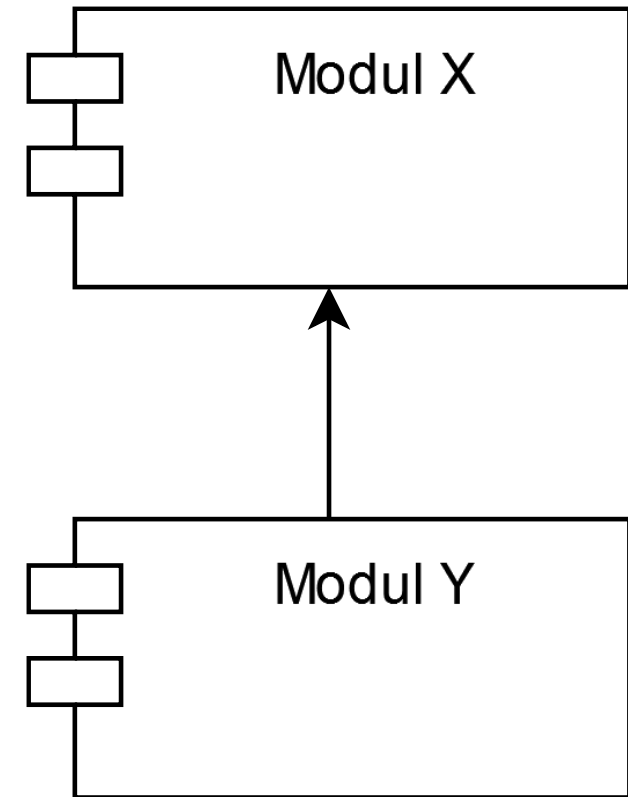
- „A software should be open for extension but closed for modification“
 - Erweiterungen sollen möglich sein, ohne vorhandene Funktionalität anpassen zu müssen!

Open-Closed Principle (OCP)



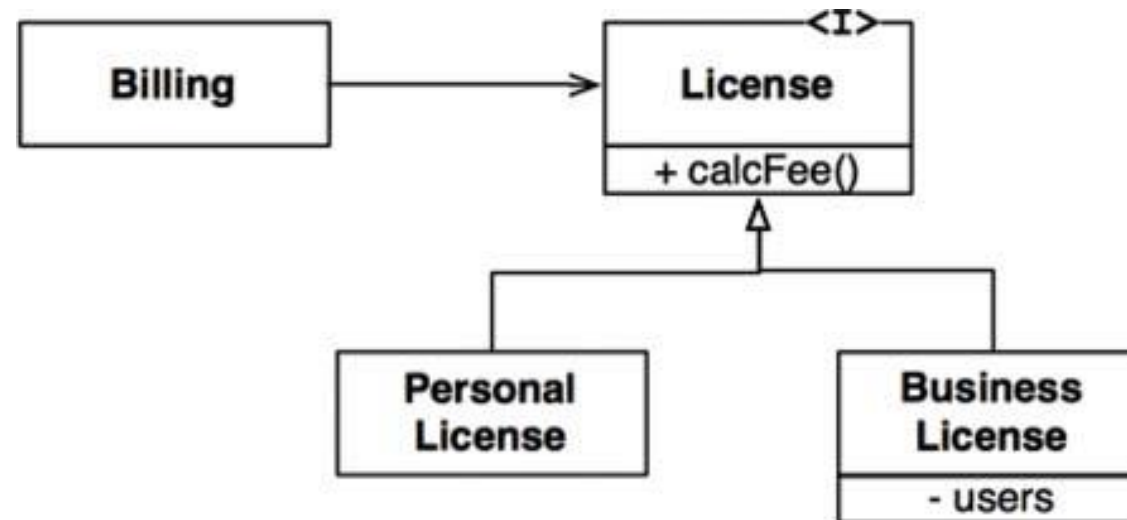
- Vererbungshierarchie

```
1  class Sensor {  
2      public:  
3          virtual double read() = 0;  
4  };  
5  
6  class TemperatureSensor : public Sensor {  
7      public:  
8          double read() override {  
9              return 3.14;  
10         }  
11  };
```

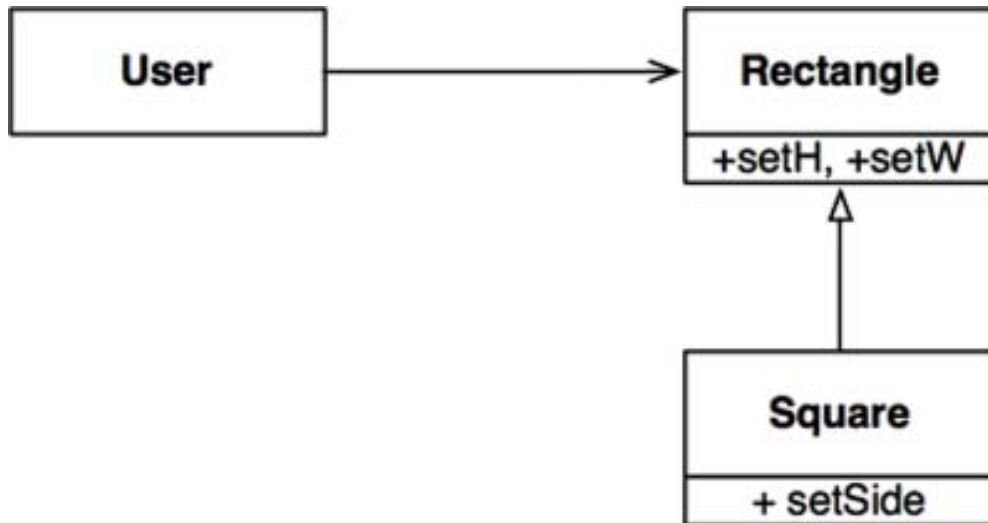


Liskov Substitution Principle (LSP)

- Ein „Objekt“ einer Überklasse muss immer durch ein Objekt einer der Unterklassen ersetzt werden können, ohne die korrekte Funktionsweise der Applikation zu verhindern.



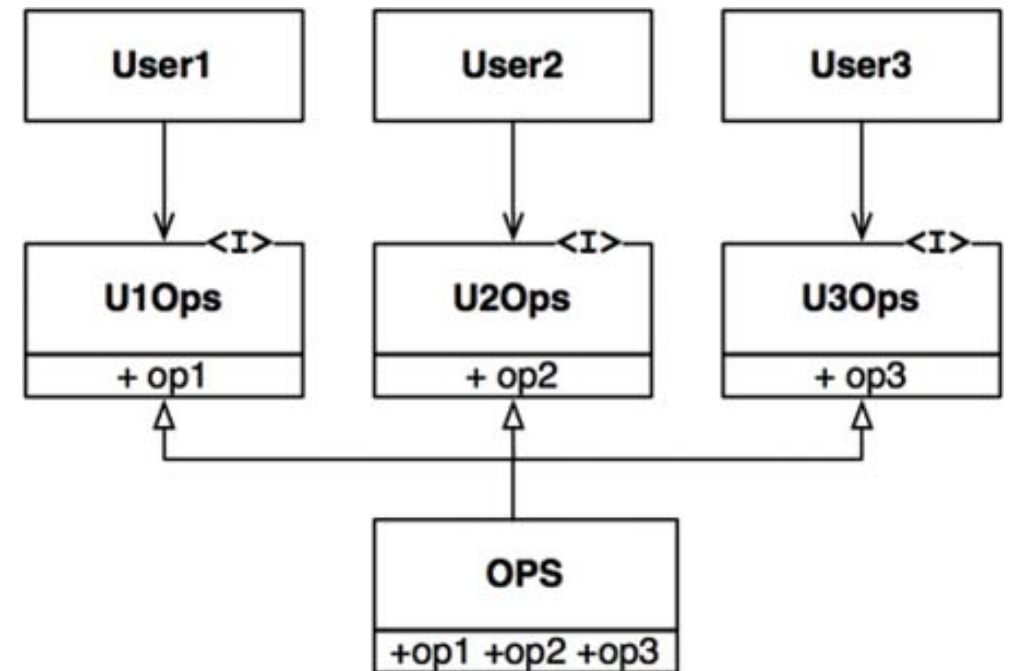
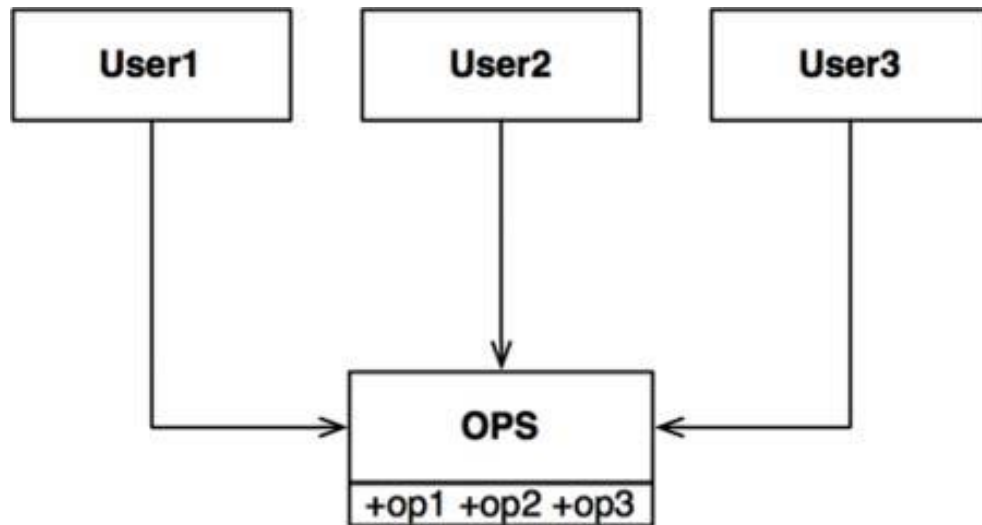
Liskov Substitution Principle (LSP)



```
1 class Sensor {
2 public:
3     virtual double readData() const = 0;
4 };
5
6 class HumiditySensor : public Sensor {
7 public:
8     double readData() const override {
9         return 20.0;
10    }
11 };
12
13 class Negative HumiditySensor : public Sensor {
14 public:
15     double readData() const override {
16         return -20.0;
17    }
18 };
```

Interface Segregation Principle (ISP)

- „No code should be forced to depend on methods it does not use“
 - Minimierung von Abhängigkeiten

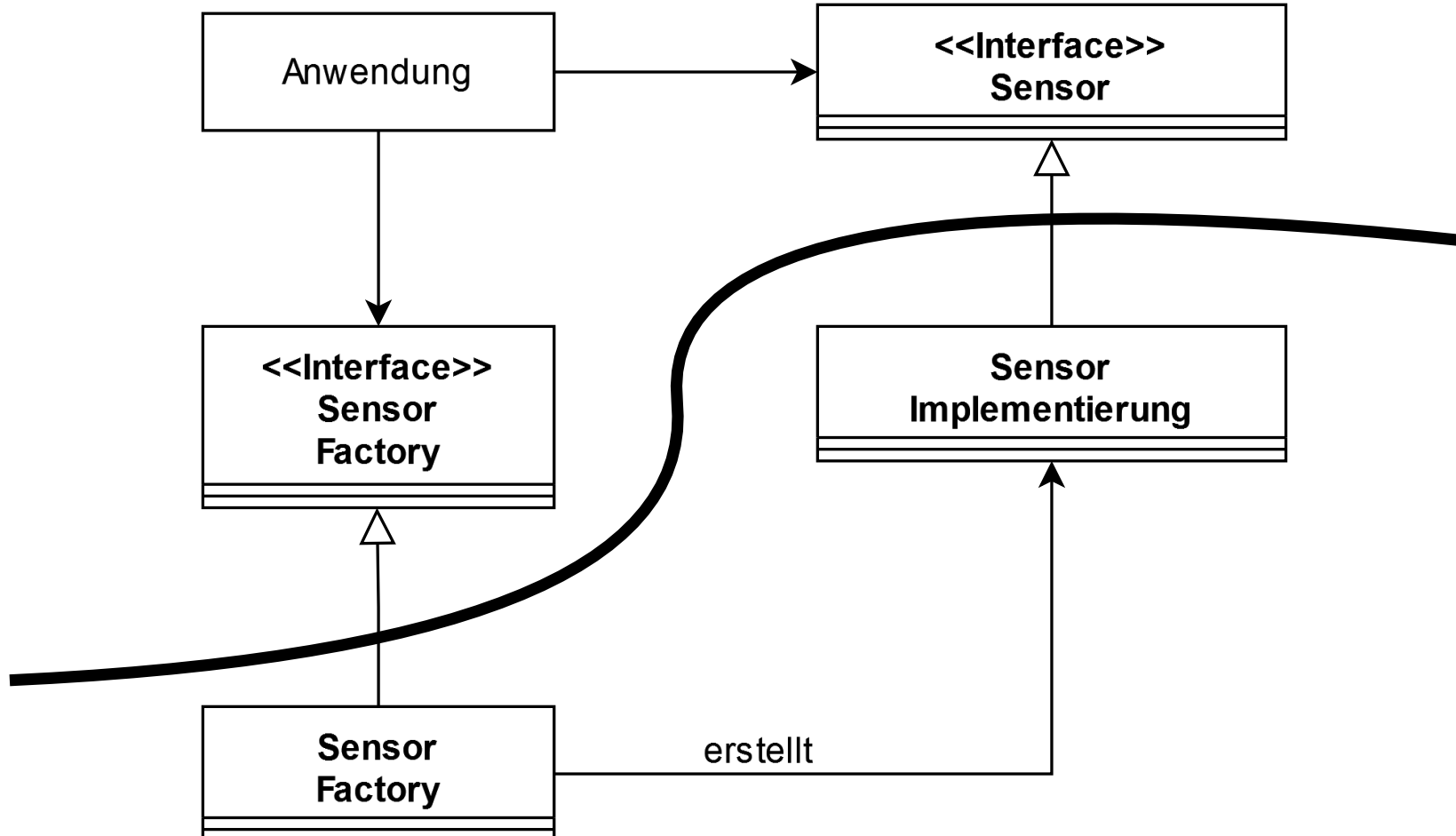


Dependency Inversion Principle



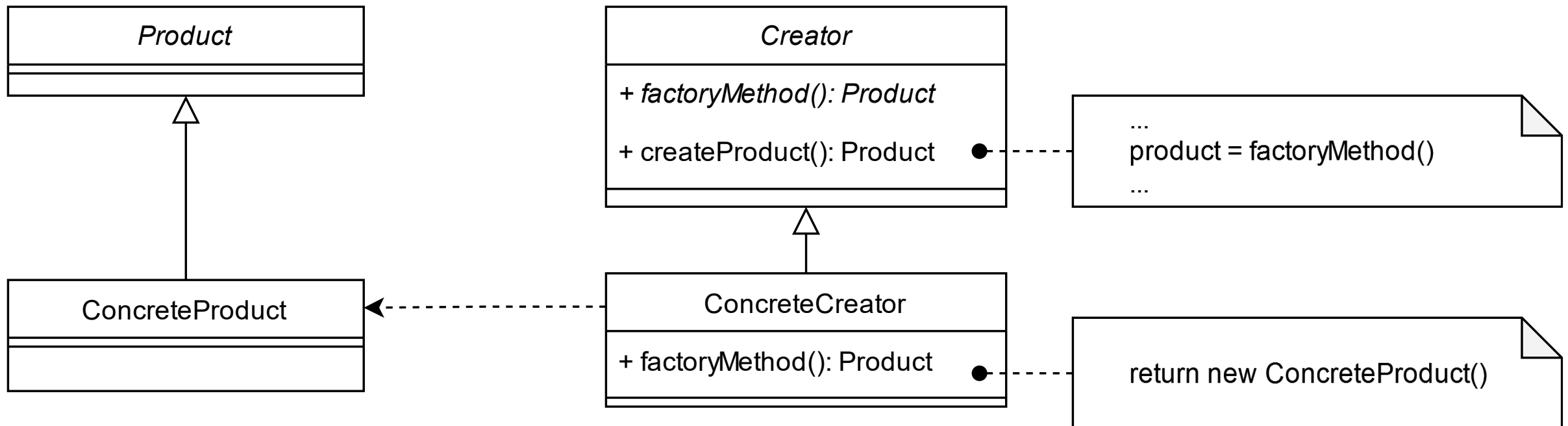
- Ein „high-level“ Modul soll nie etwas von einem „low-level“ Modul verwenden/importieren
- Abhängigkeiten sollen immer über Abstraktionen und keine konkrete Ausprägungen abgebildet werden

Dependency Inversion Principle



Patterns – Factory Method

- Idee:
 - Die Erstellung von Objekten soll von der Verwendung entkoppelt werden (**S**ingle Responsibility Prinzip)
 - Die Einführung neuer Objekte soll bestehenden Code nicht beeinflussen (**O**pen-Closed Prinzip)
 - Abhängigkeit von Interfaces (**D**ependency-Inversion Prinzip)



Nachbildung des Factory Method Pattern (Gamma et al. 1994, S. 108)

Patterns – Factory Method – Beispiel: Sensoren

```
1 class Sensor {
2     public:
3         virtual double read() = 0;
4         virtual ~Sensor() = default;
5 };
6
7 class TemperatureSensor : public Sensor {
8     public:
9         double read() override {
10             std::cout << "Reading temperature sensor..." << std::endl;
11             return 3.14;
12         }
13 };
```

```
1 std::unique_ptr<Sensor> sensor = std::make_unique<TemperatureSensor>();
2 sensor->read();
```

Patterns – Factory Method – Beispiel: Sensoren

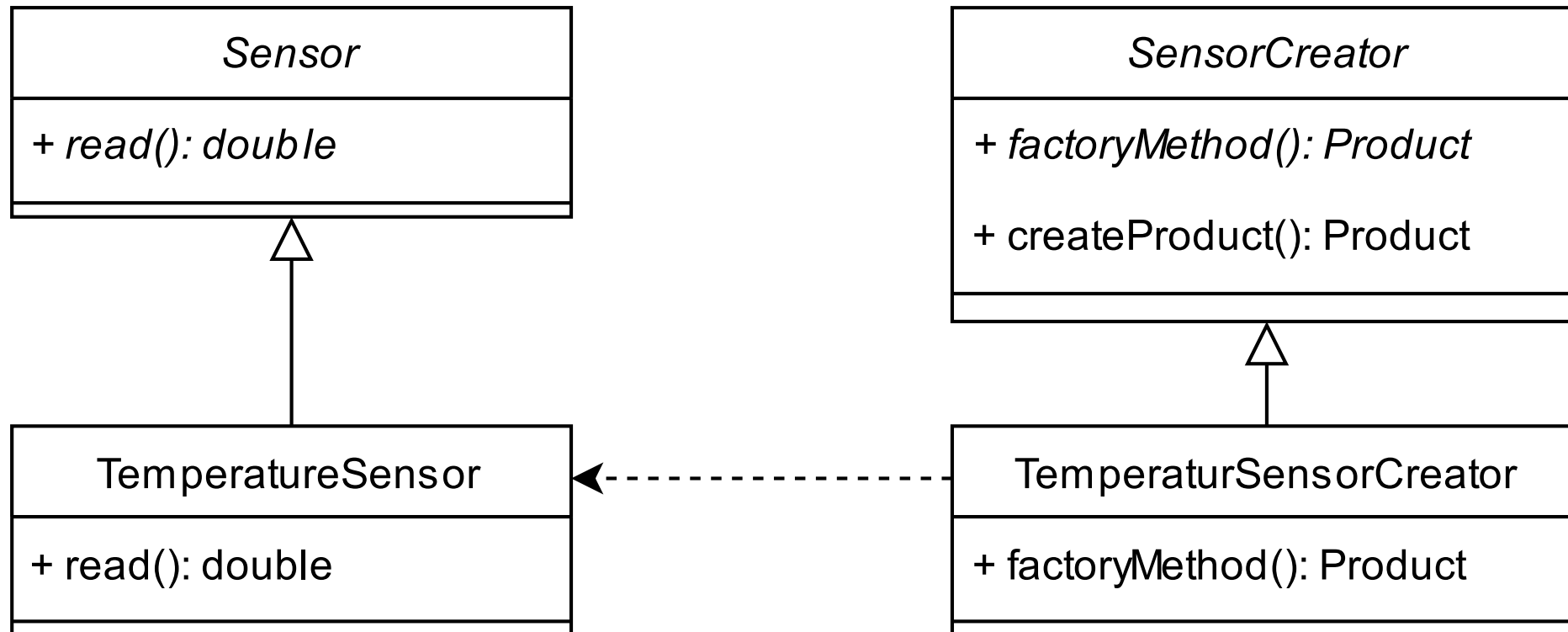
```
1 class Sensor {
2     public:
3         virtual double read() = 0;
4         virtual ~Sensor() = default;
5 };
6
7 class TemperatureSensor : public Sensor {
8     public:
9         double read() override {
10             std::cout << "Reading temperature sensor..." << std::endl;
11             return 3.14;
12         }
13 };
14
15 class PressureSensor : public Sensor {
16     public:
17         double read() override {
18             std::cout << "Reading pressure sensor..." << std::endl;
19             return 2.71;
20         }
21 };
```

```
1 std::unique_ptr<Sensor> sensor = nullptr;
2 std::string sensorType = "Temperature";
3 if (sensorType == "Temperature") {
4     sensor = std::make_unique<TemperatureSensor>();
5 } else if (sensorType == "Pressure") {
6     sensor = std::make_unique<PressureSensor>();
7 }
8 sensor->read();
```

Patterns – Factory Method – Beispiel: Sensoren



- Anwenden des Factory Method Pattern auf Sensoren



Patterns – Factory Method – Beispiel: Sensoren

- Erstellen der “Creators” zusätzlich zur Sensorhierarchie

```
1 class SensorCreator {
2     public:
3         virtual std::unique_ptr<Sensor> createSensor() = 0;
4         virtual ~SensorCreator() = default;
5         std::unique_ptr<Sensor> create() {
6             return createSensor();
7         }
8 };
9 class TemperatureSensorCreator : public SensorCreator {
10     public:
11         std::unique_ptr<Sensor> createSensor() override {
12             return std::make_unique<TemperatureSensor>();
13         }
14 };
15 class PressureSensorCreator : public SensorCreator {
16     public:
17         std::unique_ptr<Sensor> createSensor() override {
18             return std::make_unique<PressureSensor>();
19         }
20 };
```

```
1 void run(SensorCreator& creator) {
2     auto sensor = creator.create();
3     sensor->read();
4 }
5
6 TemperatureSensorCreator c1;
7 PressureSensorCreator c2;
8 run(c1);
9 run(c2);
```

- Der “Client”/Anwender muss keine Kenntnis von spezifischen Sensoren haben
- Neue Sensoren/Creator erfordern keine Anpassung des Client

Patterns – Factory Method – Beispiel: Sensoren

- Parameterized Factory Method:

```
1 enum class SensorType {  
2     Temperature,  
3     Pressure,  
4 };  
5 class SensorCreator {  
6     public:  
7         virtual ~SensorCreator() = default;  
8         virtual std::unique_ptr<Sensor> create(SensorType type) {  
9             switch (type) {  
10                case SensorType::Temperature:  
11                    return std::make_unique<TemperatureSensor>();  
12                case SensorType::Pressure:  
13                    return std::make_unique<PressureSensor>();  
14                default:  
15                    return nullptr;  
16            }  
17        }  
18 };
```

```
1 SensorCreator creator;  
2 auto sensor = creator.create(SensorType::Temperature);  
3 sensor->read();
```

- Steuerung durch Parameter
- Kann überschrieben werden

Patterns – Factory Method – Beispiel: Sensoren



- Verwendung von Templates:

```
1 template<typename T>
2 class DefaultCreator: public SensorCreator {
3     public:
4         std::unique_ptr<Sensor> createSensor() override {
5             return std::make_unique<T>();
6         }
7 };
```

```
1 void run(SensorCreator& creator) {
2     auto sensor = creator.create();
3     sensor->read();
4 }
5
6 DefaultCreator<TemperatureSensor> c1;
7 DefaultCreator<PressureSensor> c2;
8 run(c1);
9 run(c2);
```

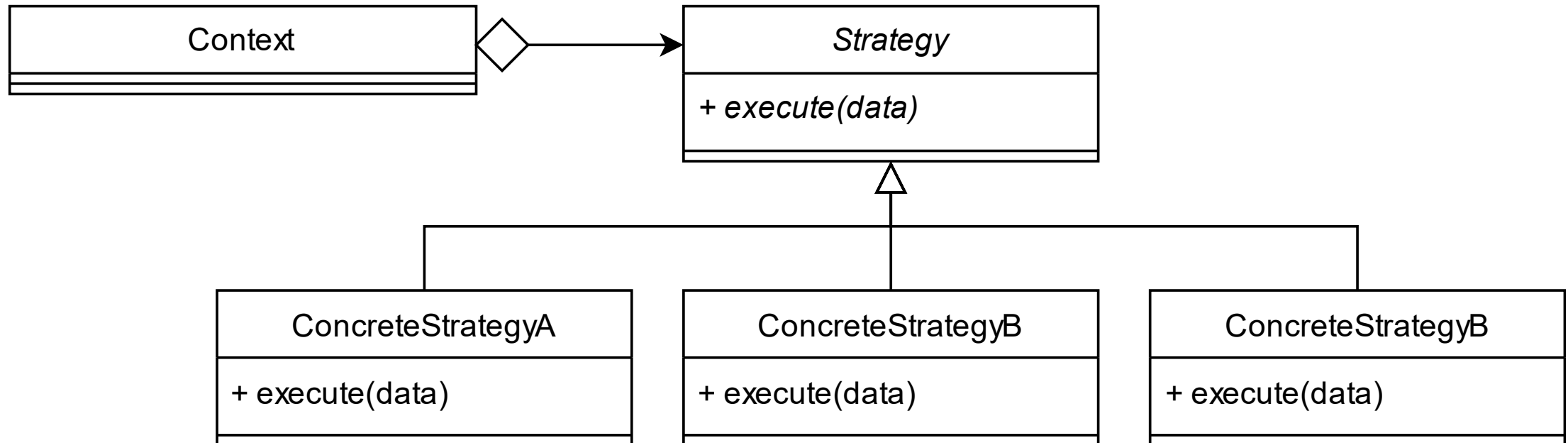
```
1 template<typename T>
2 class DefaultCreator: public SensorCreator {
3     public:
4         std::unique_ptr<Sensor> createSensor() override {
5             return std::make_unique<T>();
6         }
7 };
```

- Keine Vererbungshierarchie für “Creator” nötig

Patterns - Strategy



- Idee:
 - Entkopplung einer Vorgehensweise/Verhalten/Algorithmus aus einem Kontext (**S**ingle Responsibility)
 - Die Einführung neuen Verhaltens soll bestehenden Code nicht beeinflussen (**O**pen-Closed Prinzip)

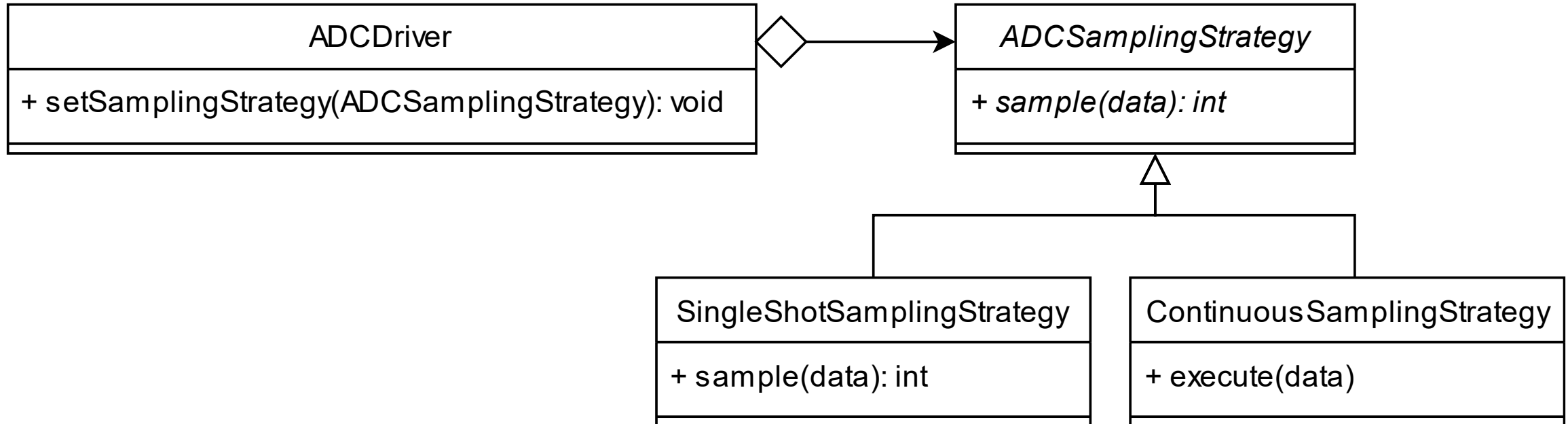


Nachbildung des Strategy Pattern (Gamma et al. 1994, S. 316)

Patterns – Strategy – Beispiel: ADC Sampling



- Anwenden des Strategy Patterns auf ADC Sampling



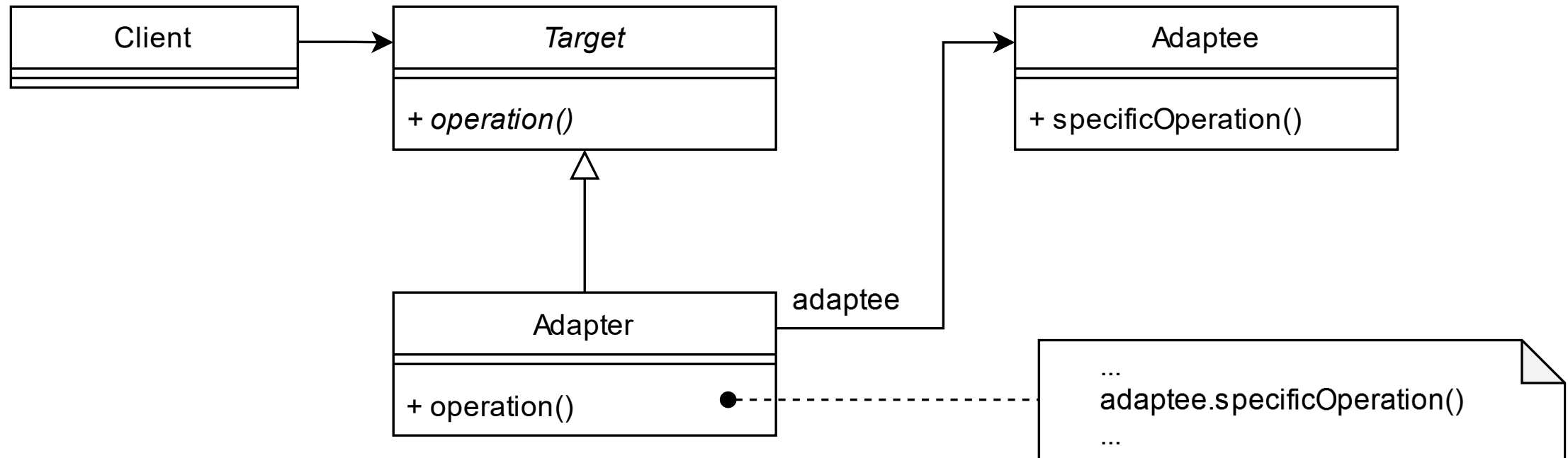
Patterns – Strategy – Beispiel: ADC Sampling

```
1 class ADCSamplingStrategy {
2     public:
3         virtual ~ADCSamplingStrategy() = default;
4         virtual int sample() = 0;
5 };
6
7 class SingleShotSamplingStrategy : public ADCSamplingStrategy {
8     public:
9         int sample() override {
10             std::cout << "Single-shot sampling..." << std::endl;
11             return 0;
12         }
13 };
14
15 class ContinuousSamplingStrategy : public ADCSamplingStrategy {
16     public:
17         int sample() override {
18             std::cout << "Continuous sampling..." << std::endl;
19             return 0;
20         }
21 };
```

```
1 ADCDriver adc(std::make_unique<SingleShotSamplingStrategy>());
2 int value = adc.sample();
3 adc.setSamplingStrategy(std::make_unique<ContinuousSamplingStrategy>());
4 value = adc.sample();
```

Patterns - Adapter

- Idee:
 - Gemeinsame Verwendung von Objekten mit nicht kompatiblen Schnittstellen (Interfaces) (**O**pen-**C**losed Prinzip)



Nachbildung des Adapter Pattern (Gamma et al. 1994, S. 141)

Patterns – Adapter – Beispiel: Protokoll

- Es existiert folgende Code-Base:

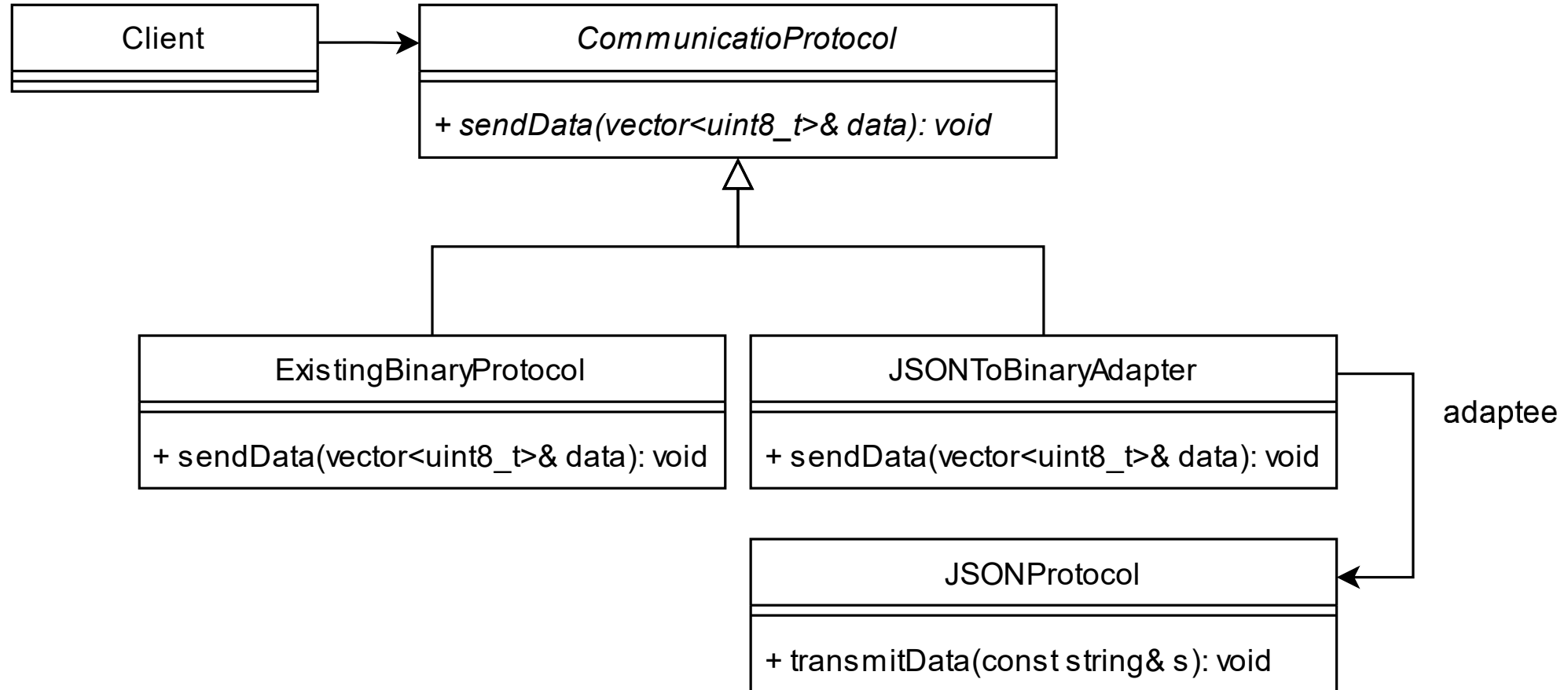
```
1 class CommunicationProtocol {
2     public:
3         virtual void sendData(const std::vector<uint8_t>& data) = 0;
4         virtual ~CommunicationProtocol() = default;
5 };
6
7 class ExistingBinaryProtocol : public CommunicationProtocol {
8     public:
9         void sendData(const std::vector<uint8_t>& data) override {
10             std::cout << "Transmitting binary data..." << std::endl;
11         }
12 };
13
14 void run(CommunicationProtocol& protocol) {
15     std::vector<uint8_t> data = {0x01, 0x02, 0x03, 0x04};
16     protocol.sendData(data);
17 }
```

- Eine 3rd Party Library verwendet ein JSON-Protokoll und soll integriert werden:

transmitData(string& d)

- Der Client-Code (run) soll nicht angepasst werden müssen

Patterns – Adapter – Beispiel: Protokoll



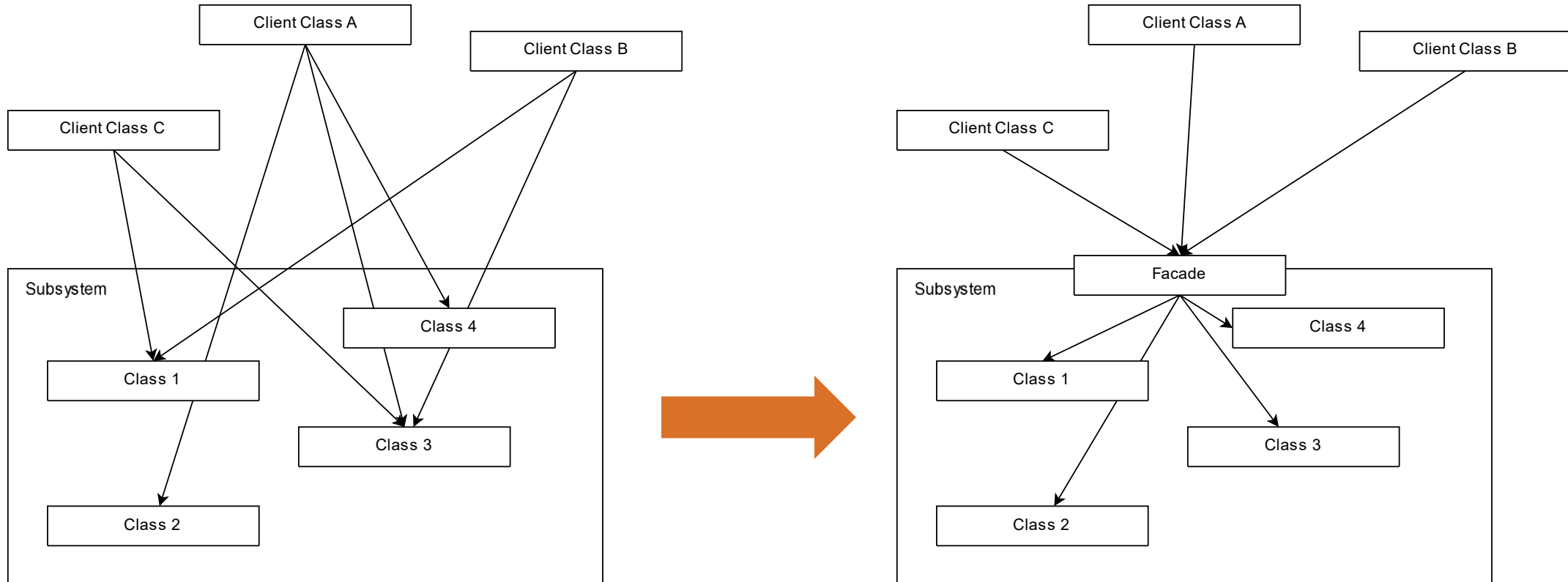
Patterns – Adapter – Beispiel: Protokoll

```
1 class JSONProtocol {
2     public:
3         void transmitData(const std::string& data) {
4             std::cout << "Transmitting JSON data: " << data << std::endl;
5         }
6 };
7 class JSONToBinaryAdapter : public CommunicationProtocol {
8     private:
9         std::unique_ptr<JSONProtocol> protocol;
10    public:
11        JSONToBinaryAdapter() : protocol(std::make_unique<JSONProtocol>()) {}
12
13        void sendData(const std::vector<uint8_t>& data) override {
14            std::string jsonData = convertBinaryToJson(data);
15            protocol->transmitData(jsonData);
16        }
17
18        std::string convertBinaryToJson(const std::vector<uint8_t>& data) {
19            return "0xDEADBEEF";
20        }
21 };
22
```

```
1 ExistingBinaryProtocol bp;
2 run(bp);
3 JSONToBinaryAdapter adapter;
4 run(adapter);
```

Patterns - Facade

- Idee:
 - Die Kopplung unterschiedlicher Module/Subsystemen soll gering gehalten werden
 - Client-Code hängt von einer Facade ab und nicht von details eines Subsystems (**D**ependency **I**nversion)



Nachbildung des Facade Pattern (Gamma et al. 1994, S. 185)

Patterns – Facade – Beispiel Cellular Subsystem



```
1 class ModuleA : public CellularModule {
2     bool _isPoweredOn = false;
3     bool _registeredInNetwork = false;
4 public:
5     void powerOn() override {
6         std::cout << "Powering on module A..." << std::endl;
7         _isPoweredOn = true;
8     }
9     bool isPoweredOn() override {
10         return _isPoweredOn;
11     }
12     void searchForNetworks() override {
13         std::cout << "Searching for networks with module A..." << std::endl;
14         _registeredInNetwork = true;
15     }
16     bool isRegisteredInNetwork() override {
17         return _registeredInNetwork;
18     }
19     bool createPDPCContext() override {
20         std::cout << "Creating PDP context with module A..." << std::endl;
21         return true;
22     }
23 };
```

```
1 class CellularModule {
2 public:
3     virtual void powerOn() = 0;
4     virtual bool isPoweredOn() = 0;
5     virtual void searchForNetworks() = 0;
6     virtual bool isRegisteredInNetwork() = 0;
7     virtual bool createPDPCContext() = 0;
8 };
```

Patterns – Facade – Beispiel Cellular Subsystem

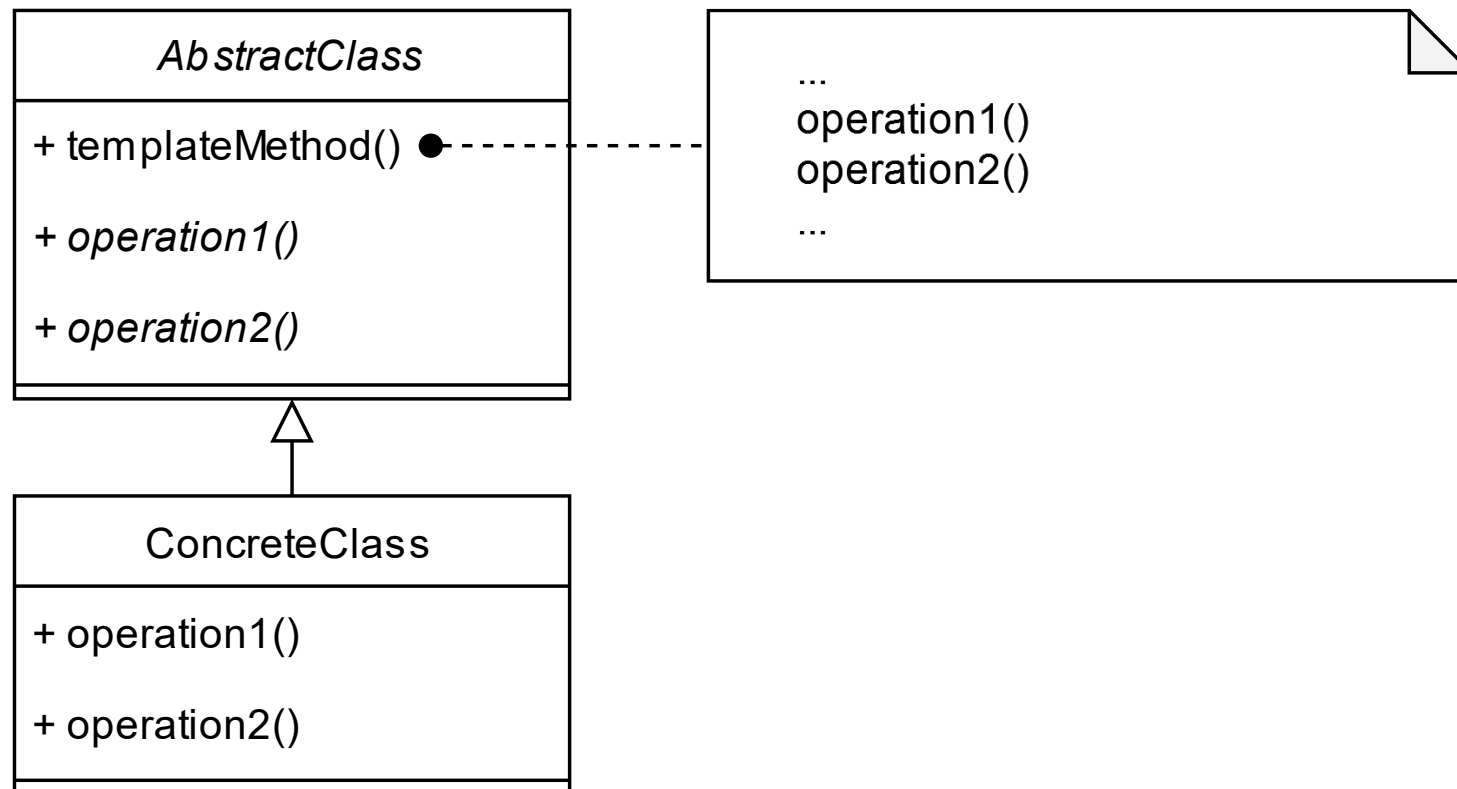
```
1 class CellularFacade {
2     private:
3         std::unique_ptr<CellularModule> _module;
4     public:
5         CellularFacade(CellularModule* module) : _module(module) {}
6
7         void powerOn() {
8             if (!_module->isPoweredOn()) {
9                 _module->powerOn();
10            }
11        }
12
13        bool connect() {
14            if (!_module->isPoweredOn()) {
15                _module->powerOn();
16            }
17            if (!_module->isRegisteredInNetwork()) {
18                _module->searchForNetworks();
19            }
20            return _module->createPDPCContext();
21        }
22 };
```

- Keine Abhängigkeit von Interface/Typen aus Subsystem
- **connect** entkoppelt Details des Subsystems

Patterns – Template Method



- Idee:
 - Aufteilung eines Algorithmus in einzelnen Schritte – Schritte können von Subklassen überschrieben werden



Patterns – Template Method

```

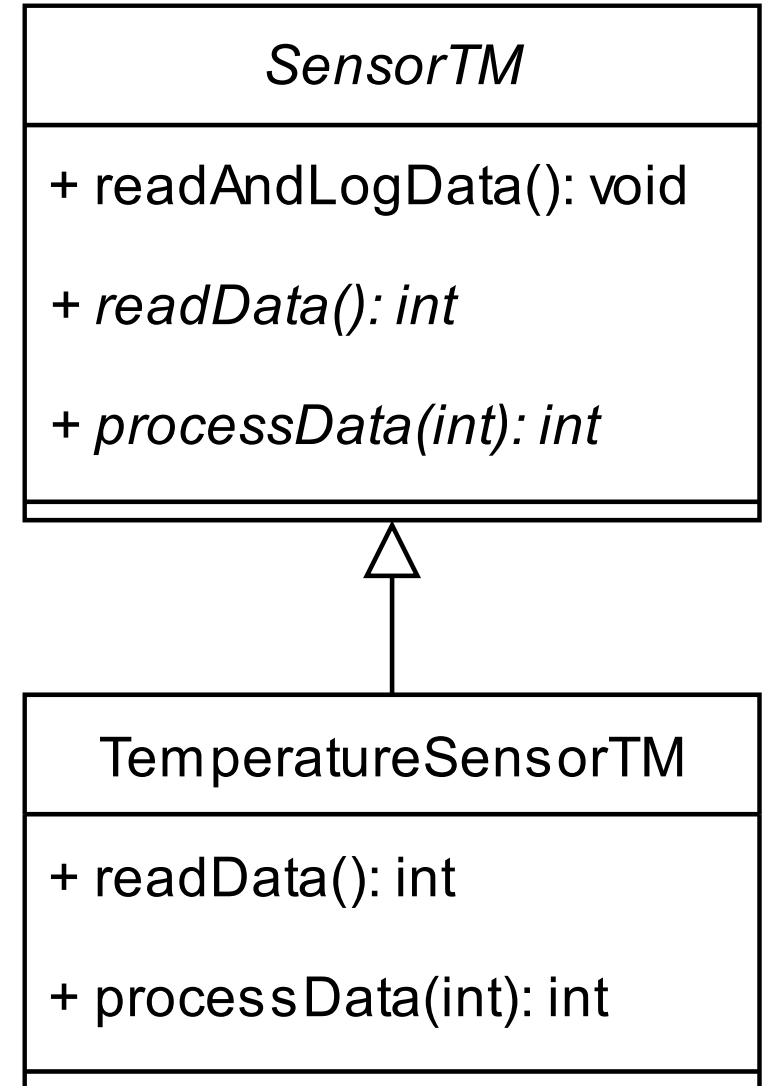
1 class SensorTM {
2     public:
3         void readAndLogData() {
4             int data = readData();
5             data = processData(data);
6             logData(data);
7         }
8         virtual ~SensorTM() = default;
9
10        protected:
11            virtual int readData() = 0;
12            virtual int processData(int data) = 0;
13
14        void logData(int data) {
15            std::cout << "Sensor data: " << data << std::endl;
16        }
17    };
18    class TemperatureSensorTM : public SensorTM {
19        protected:
20            int readData() override {
21                return 22;
22            }
23            int processData(int data) override {
24                return data;
25            }
26    };

```

```

1 void run(SensorTM& sensor) {
2     sensor.readAndLogData();
3 }
4
5 TemperatureSensorTM tempSensor;
6 run(tempSensor);

```



Nachrichten-Verteiler



- Entwickeln Sie ein System, bei dem von einem Nachrichter-Verteiler Nachrichten an alle registrierten Interessenten weitergeleitet werden
- Der Nachrichten-Verteiler bietet zumindest folgende Funktionalität an:
 - Registrierung von Interessenten
 - Deregistrierung von Interessenten
 - Aufforderung zur Weiterleitung einer Nachricht an alle registrierten Interessenten

Konzepte seit C++20

<https://en.cppreference.com/w/cpp/language/constraints>

- Ermöglichen es Anforderungen an Platzhalter des Templates konkret zu spezifizieren

```
1 template <typename T>
2 requires std::copyable<T>
3 T maximum(T a, T b) {
4     return a > b ? a : b;
5 }
```

```
1 int main(int argc, char const *argv[]) {
2     unique_ptr<int> pi1 = make_unique<int>(47);
3     unique_ptr<int> pi2 = make_unique<int>(11);
4     cout << maximum(pi1, pi2) << endl;
5 }
```

Output

```
error: no matching function for call to 'maximum(std::unique_ptr<int, std::default_delete<int> >&,
std::unique_ptr<int, std::default_delete<int> >&)'cout << maximum(pi1, pi2) << endl;
```

...

```
note: the expression 'is_constructible_v<_Tp, _Args ...> [with _Tp = std::unique_ptr<int,
std::default_delete<int> >; _Args = {std::unique_ptr<int, std::default_delete<int> >&}]' evaluated to
'false'
```

```
1 auto maximum(std::copyable auto a, std::copyable auto b) {
2     return a > b ? a : b;
3 }
```

Konzepte seit C++20

- Ermöglichen es Anforderungen an Platzhalter des Templates konkret zu spezifizieren
- Benutzerdefiniert Konzepte können erstellt werden:

```
1 template <typename T>
2 concept FooBarConcept = requires(T t) {
3     { t.bar() };
4 };
```

- Funktion bar wird vorausgesetzt

```
1 template <typename T>
2 concept FooBarConcept2 = requires(T t, int v) {
3     { t.baz() };
4     { t.f(v) };
5 };
```

- Funktion f mit int Parameter wird vorausgesetzt

```
1 template <typename T>
2 concept FooBarConcept3 = requires(T t, int v) {
3     { t.baz() } -> std::integral;
4     { t.f(v) };
5 };
```

- Funktion baz hat als Rückgabedatentyp ein integralen Wert (int, char, bool)

Konzepte seit C++20

```
1 template <typename T>
2 concept FooBarConcept3 = requires(T t, int v) {
3     { t.baz() } -> std::integral;
4     { t.f(v) };
5 };
6
7 template<FooBarConcept3 T>
8 void handle(T t) {
9     t.bar();
10    t.baz();
11    t.f(42);
12 }
```

```
1 class Foo {
2 public:
3     void bar() {
4         std::cout << "Foo bar" << std::endl;
5     }
6     double baz() {
7         return 42.3;
8     }
9     void f(int x) {
10        std::cout << "Foo f " << x << std::endl;
11    }
12 };
13
14 Foo f;
15 handle(f);
```

Output

```
required for the satisfaction of 'FooBarConcept3<T>'
note: 't.baz()' does not satisfy
return-type-requirement { t.baz() } -> std::integral;
```

Template Metaprogramming

https://en.cppreference.com/w/cpp/types/enable_if

- Konzepte sind erst ab C++20 verfügbar
- **T.48: If your compiler does not support concepts, fake them with enable_if**

```
1  template<typename T>
2  enable_if_t<is_function_v<T>> foo(T v);
3
4  template <typename T>
5  void foo(T x) {
6      x();
7  }
8
9  void foo(const char* v) {
10     cout << "is string " << v << endl;
11 }
12
13 int main(int argc, char const *argv[]) {
14     foo("test");
15     foo([]() { cout << "is lambda" << endl; });
16     return 0;
17 }
```

- Durch **enable_if/enable_if_t** und **type traits** wird Verhalten von Konzepten nachgestellt werden

Output

```
is string test
is lambda
```

Modules seit C++20

<https://en.cppreference.com/w/cpp/language/modules>

```
1 export module CustomModule;
2 import <iostream>;
3
4 constexpr int answer = 42;
5
6 export void greet() {
7     ↑ std::cout << answer << std::endl;
8 };
```

• Neues Modul wird erstellt

• `iostream` Modul wird verwendet

• Funktion von außen sichtbar

```
1 import <iostream>;
2 import CustomModule;
3
4 int main(int argc, char const *argv[]) {
5     greet();
6     return 0;
7 }
```

• Standard-Modul importieren

• Benutzerdefiniertes Modul importieren

• Benutzerdefiniertes Modul verwenden

Modules seit C++20



```
1 module;
2 //Präprozessor Anweisungen
3 export module foo;
4 module: private;
5
6 int f() {
7     return 42;
8 }
```

- Globales Modul Fragment

- Privates Modul Fragment

Modules seit C++20



```
1 export module Hal.Uart;
```

```
1 export module Hal.Spi;
```

```
1 export module Hal;  
2 export import Hal.Uart;  
3 export import Hal.Spi;
```

- Submodule

Modules seit C++20



```
1 module Hal:Uart;
```

```
1 module Hal:Internal;
```

```
1 export module Hal;  
2 export import :Uart;  
3 import :Internal
```

- Partitionen

Modules seit C++20

- Compile Support/IDE-Integration unter Umständen noch nicht ausgereift
- Verwendung von Modulen benötigt zusätzliches Compiler-Flag (**-fmodules-ts**)
- Standard-Header zu Module konvertieren
g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
- Abhängigkeiten eines Moduls werden nicht weitergegeben

Template Syntax für Lambdas seit C++20



```
1 auto f = []<typename T>(T v) {  
2     ...  
3 };
```

- typename kann in Lambdas verwendet werden

```
1 auto f = []<FooBarConcept T>(T v) {  
2     v.bar();  
3 };
```

- Direkte Verwendung von Konzepten möglich

std::span seit C++20

- Repräsentiert eine zusammenhängende Sequenz von Objekten (vgl. Array)
- Kennt die Anzahl der Element in der Sequenz
- Nicht-Besitzendes Konstrukt
→ führt keine Allokationen/Deallokationen durch
- **P.5: Prefer compile-time checking to run-time checking**

```
1  #include <span>
2  #include <iostream>
3
4  void f(int* data, size_t size) {
5      for (size_t i = 0; i < size; i++) {
6          std::cout << data[i] << std::endl;
7      }
8  }
9  void f(std::span<int> data) {
10     for (size_t i = 0; i < data.size(); i++) {
11         std::cout << data[i] << std::endl;
12     }
13 }
14 int main(int argc, char const *argv[]) {
15     int data[] = {1,2,3,4,5};
16     f(data, 5);
17     f(data);
18     return 0;
19 }
```

Koroutinen seit C++20



- Eine Koroutine ist eine Funktion die ihren Zustand über Funktionsaufrufe hinweg behält
- Platzsparend → Thread benötigt Speicher im Megabyte-Bereich, Koroutine ~100Byte
- Schnelleres Kontext-Switching als bsplw. Threads
- Aktuell fehlen die “Komfort”-Funktionalität in der Standard Library um eine einfache Nutzung zu ermöglichen
- Es existieren Implementierung: <https://github.com/lewissbaker/cppcoro>

Quellen



- B. Stroustrup: *The C++ Programming Language*, Addison-Wesley, 2013
- B. Stroustrup: *A Tour of C++ (3rd Edition)*, Addison-Wesley, 2022
- S. Meyers: *Effective Modern C++*, O'Reilly, 2014
- Erich Gamma et al: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
- Robert C. Martin: *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 2017