

SWE Zusammenfassung

Elias Leonhardsberger

11. Juli 2025, Hagenberg

Inhaltsverzeichnis

1	C++	3
1.1	Standard Library	3
1.1.1	Strings	3
1.1.2	Ein-/Ausgabe	3
1.2	STL	5
1.2.1	Iteratoren	5
1.2.2	Behälter	7
1.2.3	Funktionsobjekte	8
1.2.4	Algorithmen	9
2	Java	10
2.1	Eigenschaften	10
2.2	Datentypen	10
2.2.1	Klassen und Objekte	12
2.2.2	Interfaces und Abstrakte Klassen	12
2.2.3	Generizität	12
2.2.4	Lamda-Ausdrücke und Streams	13
2.3	JCF	13
3	Softwaremuster	14
3.1	Idioms	14
3.2	MVC - Model-View-Controller	14
3.3	Liskovsches Substituionsprinzip (LSP)	14
3.4	OOP Prinzipien	14
3.5	Gang of Four	15
3.6	Erzeugungsmuster	16
3.6.1	Abstract Factory oder Kit	16
3.6.2	Builder	16
3.6.3	Factory Method oder Virtual Constructor	16
3.6.4	Prototype	17
3.6.5	Singleton	17
3.7	Strukturmuster	17
3.7.1	Adapter oder Wrapper	17

3.7.2	Bridge	18
3.7.3	Composite	18
3.7.4	Decorator oder Wrapper	19
3.7.5	Facade	19
3.7.6	Flyweight oder Shared Object	19
3.7.7	Proxy	19
3.8	Verhaltensmuster	19
3.8.1	Chain of Responsibility	19
3.8.2	Command	19
3.8.3	Interpreter	19
3.8.4	Iterator	19
3.8.5	Mediator	19
3.8.6	Memento	20
3.8.7	Observer oder Publisher/Subscriber	20
3.8.8	State	20
3.8.9	Strategy	20
3.8.10	Template Method	20
3.8.11	Visitor	20
3.9	Weitere Arten von Mustern	20
3.10	Softwarearchitektur	21
3.10.1	Sichten in der Softwarearchitektur	21
4	Syntaxvergleich	22
4.1	C++	22
4.1.1	Interface	22
4.1.2	Basisklasse	22
4.1.3	Abgeleitete Klasse	23
4.1.4	Main	27
4.2	Java	29
4.2.1	Interface	29
4.2.2	Basisklasse	29
4.2.3	Abgeleitete Klasse	29
4.2.4	Main	31
4.3	C# kommt nicht zur Klausur - hab ich zu spät gesehen	32
4.3.1	Interface	32
4.3.2	Basisklasse	32
4.3.3	Abgeleitete Klasse	32
4.3.4	Main	34

1 C++

1.1 Standard Library

Der Namensraum *std* ist für die Standardbibliothek von C++ und C reserviert. Alle C Header-Dateien sind in C++ verfügbar, ihr Name ist immer mit einem *c* vorangestellt, z.B. *cstdio* für *stdio.h* in C.

1.1.1 Strings

Im gegensatz zu C, wo Strings als char-Arrays implementiert sind, gibt es in C++ die Klasse `std::string`, die es ermöglicht mit Templates Strings aus `char`, `wchar_t` oder einem beliebigen eigenen Typ zu erstellen. Die ersten beiden Varianten sind bereits vordefiniert.

```
1 typedef basic_string< char >    string;
2 typedef basic_string<wchar_t> wstring;
```

1.1.2 Ein-/Ausgabe

In C und C++ gibt es keine in die Sprache eingebauten Ein-/Ausgabefunktionen. In C werden die Funktionen aus *stdio.h* verwendet, in C++ die Klassen aus *iostream*.

Der Unterschied ist, dass C++ mit Streams arbeitet, diese sind

- typsicher
- implementierbar für eigene Klassen
- effizienter da man nicht auf interpretierte Formatzeichenketten angewiesen ist (z.B. `%d` für `int`)
- auf Zeichenebene Thread-sicher

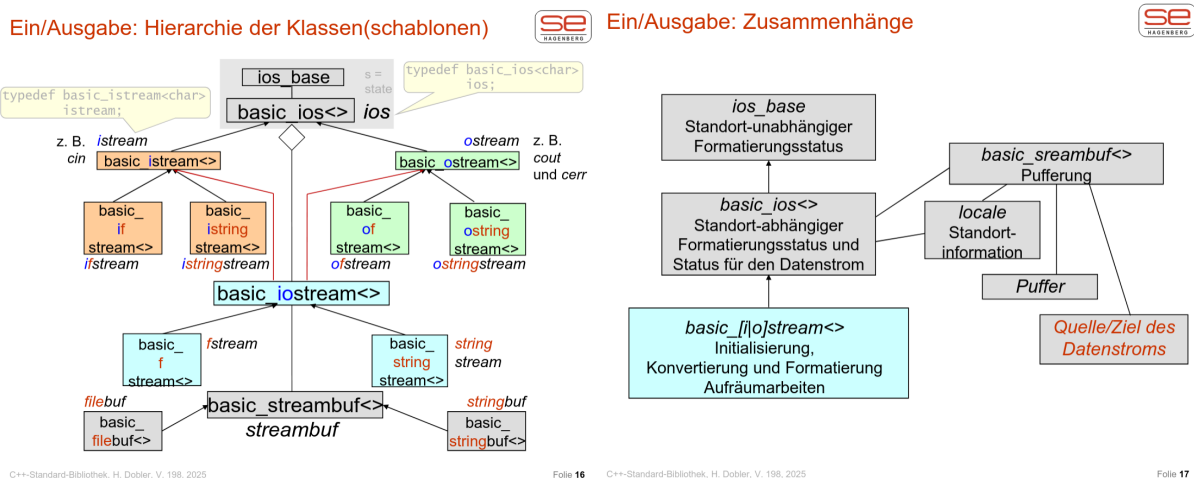


Abbildung 1: Klassenhierarchie der Ein-/Ausgabeklassen in C++

Diese Streams sind in mehreren Header-Dateien aufgeteilt:

- **iosfwd** für Vorwärtsdeklarationen(Sehr klein)
- **streambuf** für die Pufferung von Ein-/Ausgabe
- **istream** für die Eingabe
- **ostream** für die Ausgabe
- **iostream** für die Standard-Ein-/Ausgabe
- **fstream** für Dateiein-/ausgabe
- **sstream** für String-Ein-/Ausgabe
- **strstream** für für Char*-Ein-/Ausgabe
- **omanip** für Ein-/Ausgabeformatierung

Flush kann auch mit Stream Aufrufen verkettet werden, z.B. `cout << std::flush; std::endl` flusht den Stream auch nachdem er einen Zeilenumbruch ausgegeben hat. Achtung kann Performanceprobleme verursachen, da es den Puffer jedes Mal leert! Wenn mehrere Zeilenumbrüche ausgegeben werden eher den folgenden Ausschnitt verwenden.

```
1 stream << "\n"
```

Es gibt auch diverse Manipulatoren, die das Verhalten von Streams beeinflussen.

- `std::dec` für Dezimalzahlen
- `std::hex` für Hexadezimalzahlen
- `std::oct` für Oktalzahlen
- `std::setbase(n)` für die Basis der Zahlendarstellung
- `std::setfill(c)` für das Füllzeichen
- `std::setprecision(n)` für die Anzahl der Nachkommastellen
- `std::setw(n)` für die Breite des Feldes

1.2 STL

!!Diese Schablone hat keine Laufzeitprüfungen!!

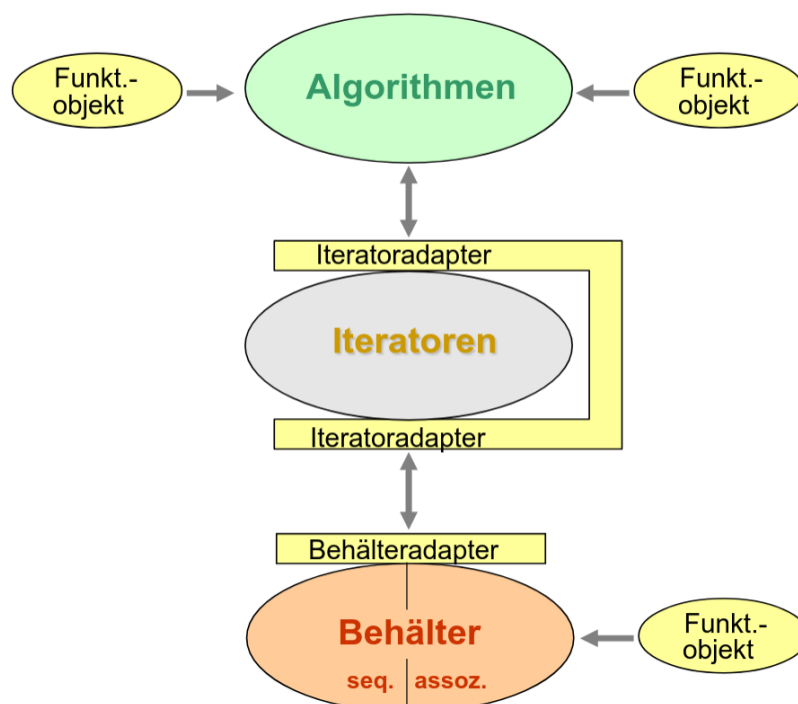
Doblers Foliensatz

Generische Programmierung statt OOP um den Code wiederverwendbar und effizient zu machen.

Die STL ist in mehrere Teile aufgeteilt:

- **Behälter** für die Speicherung von Daten
- **Iteratoren** für den Zugriff auf die Daten
- **Algorithmen** für die Verarbeitung von Daten
- **Funktionsobjekte** für die Kapselung von Funktionen für andere Komponenten
- **Adapter** für die Anpassung von Behälter und Iteratoren

Konzept des Zusammenwirkens



STL, H. Dobler, V. 19, 2025

Folie 6

Abbildung 2: Verhältnisse der STL Komponenten

1.2.1 Iteratoren

Iteratoren sind eine Verallgemeinerung von Zeigern, die eine abstrakte Schnittstelle zwischen Behältern und Algorithmen darstellen.

In der STL sind Iteratoren mithilfe von Ducktyping umgesetzt, das heißt, wenn die Schnittstelle erfüllt ist, also die Methoden richtig implementiert sind, dann kann der Iterator verwendet werden.

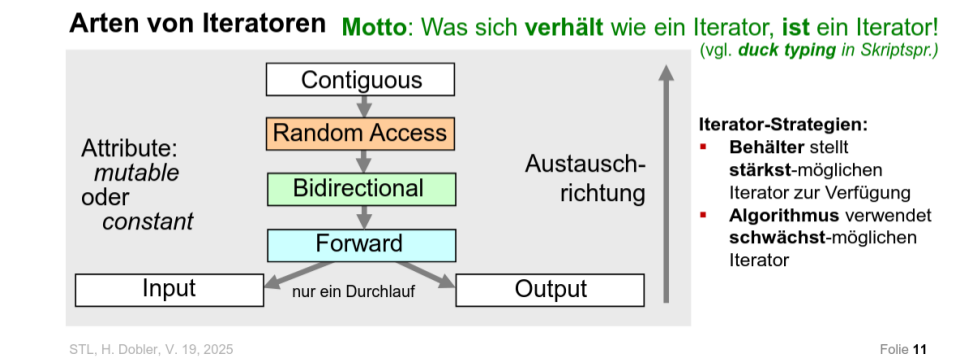


Abbildung 3: Iteratoren in der STL

Allgemein müssen Iteratoren die Vergleichsoperatoren `==` und `!=` implementieren, um zu überprüfen, ob zwei Iteratoren auf das gleiche Element zeigen, und die Dereferenzierungsoperatoren `*`, um auf das Element zuzugreifen.

Der **Vorwärtsiterator** implementiert zusätzlich den Inkrementoperator `++`, um zum nächsten Element zu gelangen.

Der **bidirektionale Iterator** implementiert zusätzlich den Dekrementoperator `-`, um zum vorherigen Element zu gelangen.

Der **Random-Access-Iterator** implementiert zusätzlich die Operatoren `+`, `-`, `+=` und `-=`, um auf ein beliebiges Element im Behälter zuzugreifen, und die Vergleichsoperatoren `<`, `>`, `<=` und `>=`, um die Position der Iteratoren zu vergleichen. Dabei wird auch ein Differenztyp benötigt (z.B. `ptrdiff_t`).

Der **Input-Iterator** ist ein Vorwärtsiterator, der nur gelesen werden kann.

Der **Output-Iterator** ist ein Vorwärtsiterator, auf den nur geschrieben werden kann.

Streams und damit auch Dateien können mit dem `istream_iterator` und `ostream_iterator` als Iteratoren verwendet werden.

Es gibt auch Adapter für Iteratoren, wie

- **reverse_iterator** für die Umkehrung der Iteration (`v.rbegin()` und `v.rend()`)
- **back_insert_iterator** für das Einfügen von Elementen am Ende eines Behälters
- **front_insert_iterator** für das Einfügen von Elementen am Anfang eines Behälters
- **insert_iterator** für das Einfügen von Elementen an einer bestimmten Position in einem Behälter

Die Einfügeiteratoren sind da um die Einfügeoperationen in Algorithmen zu abstrahieren.

Iteratoreigenschaften werden verwendet um verschiedene Implementierungen für verschiedene Iteratoren zu ermöglichen. Die Eigenschaften sind. Außerdem wird der Wertetyp und der Distanztyp damit gespeichert.

1.2.2 Behälter

Behälter sind Klassen, die Daten speichern und verwalten. Es gibt verschiedene Arten von Behältern, die sich in ihrer Implementierung und ihrem Verhalten unterscheiden. Die wichtigsten Behältertypen sind:

Behältertyp	Beschreibung	Verwendung
vector	Dynamisches Array, das hinten offen ist und schnellen direkten Zugriff ermöglicht	Standard für sequentielle Daten
list	Doppelt verkettete Liste, die schnelle Einfüge- und Löschoperationen ermöglicht, aber keine schnelle Zugriffsfunktion	Wenn häufige Einfüge-/Löschungenoperationen erforderlich sind
deque	beidseitig offenes dynamisches Array, das schnellen Zugriff auf beide Enden ermöglicht	Wenn häufige Einfüge-/Löschungenoperationen an beiden Enden erforderlich sind, aber direkter Zugriff auch wichtig ist
array	Schnittstelle für statisches Array	Wenn die Größe des Arrays zur Compilezeit bekannt ist und STL Algorithmen verwendet werden
forward_list	Minimale einfach verkettete Liste	Wenn Speicherverbrauch wichtig ist und nur Vorwärtsiteration benötigt wird

Behältertyp	Beschreibung	Verwendung
stack	LIFO-Adapter für einen Behälter. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>deque</i> oder <i>vector</i>	Wenn nur die zuletzt hinzugefügten Elemente benötigt werden.
queue	FIFO-Adapter für einen Behälter. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>deque</i> oder <i>list</i>	Wenn nur die zuerst hinzugefügten Elemente benötigt werden.
priority_queue	Adapter für einen Behälter, der die Elemente nach Priorität sortiert. Der genaue Behälter ist nicht spezifiziert, aber meist ein <i>vector</i> oder <i>deque</i>	Wenn die Elemente nach Priorität sortiert werden müssen.
set	Assoziativer Behälter, in dem die Elemente der Zugriffsschlüssel sind. (Zugriff $O(\log n)$)	Wenn die Elemente als Zugriffsschlüssel dienen.
multiset	Set in dem Duplikate erlaubt sind	Wenn die Elemente als Zugriffsschlüssel dienen aber Duplikate erlaubt sind
map	Assoziativer Behälter, der zu jedem Schlüssel ein Element zuordnet.	Wenn auf die Elemente nach einem Schlüssel zugegriffen wird.
multimap	Map in der Schlüsselduplikate erlaubt sind	Wenn auf die Elemente nach einem Schlüssel zugegriffen wird, aber Duplikate erlaubt sind.

Assoziative Behälter sind entweder mit einem Rot-Schwarz-Baum oder einem Hash-Table implementiert.

1.2.3 Funktionsobjekte

Funktionsobjekte sind Objekte einer Klasse mit öffentlichem `operator()`. Sie können also wie Funktionspointer aufgerufen werden. Da es sich aber um Objekte handelt, können sie auch Zustände speichern und so z.B. Zähler implementieren.

In den Folien wird auf 4 besondere Typen eingegangen:

- 1. Basisklassen:** Zur einfachen Definition und Verwendung von Funktionsobjekten mit einem oder zwei Argumenten:

```
template<class Arg, class Result>
struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
}; // unary_function<Arg, Result>

template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1    first_argument_type;
    typedef Arg2    second_argument_type;
    typedef Result  result_type;
}; // binary_function<Arg1, Arg2, Result>
```

- 2. Arithmetische Operationen:** Für alle arithmetischen Operationen (+, -, *, /, %) gibt es entsprechende Funktionsobjekte (*plus*, *minus*, *times*, *divide*, *modulus* und *negate*), z. B.:

```
template<class T>
struct plus: public binary_function<T, T, T> {
    T operator()(const T &x, const T &y) {
        return x + y;
    } // operator()
}; // plus<T>
```

- 3. Vergleichsoperationen:** Für alle Vergleichsoperationen (==, !=, <, <=, >, >=) gibt es entsprechende Funktionsobjekte (*equal_to*, *not_equal_to*, *less*, *less_equal*, *greater* und *greater_equal*), z. B.:

```
template<class T>
struct equal_to: public binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) {
        return x == y;
    } // operator()
}; // equal_to<T>
```

- 4. Logische Operationen:** Für alle log. Operationen (&&, || und !) gibt es entsprechende Funktionsobjekte (*logical_and*, *logical_or* und *logical_not*), z. B.:

```
template<class T>
struct logical_and: public binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) {
        return x && y;
    } // operator()
}; // logical_and<T>
```

Abbildung 4: Funktionsobjekte in der STL

1.2.4 Algorithmen

Die Algorithmen in der STL sind durch Iteratoren unabhängig von den Behältern implementiert. Parametrisiert werden sie mit Iteratortyp, Wertetyp oder Funktionsobjekten (z.B. Prädikat mit return Typ bool). Davon können beliebig viele verwendet werden.

Algorithmus	Beschreibung
ohne Änderungen	
for_each	Führt eine Funktion auf jedes Element eines Behälters aus
find	Sucht ein Element in einem Behälter (mit ==)
find_if	Sucht ein Element in einem Behälter, das ein Prädikat erfüllt
count	Zählt die Anzahl der Vorkommen eines Elements in einem Behälter
count_if	Zählt die Anzahl der Vorkommen eines Elements in einem Behälter, das ein Prädikat erfüllt
equal	Vergleicht zwei Behälter auf Gleichheit (mit ==, Reihenfolge ist relevant)
mismatch	Vergleicht zwei Behälter auf Ungleichheit
mit Änderungen	
copy	Kopiert die Elemente eines Behälters in einen anderen Behälter
copy_if	Kopiert die Elemente eines Behälters in einen anderen Behälter, die ein Prädikat erfüllen
remove	Entfernt Elemente aus einem Behälter (mit ==)
remove_if	Entfernt Elemente aus einem Behälter, die ein Prädikat erfüllen
unique	Entfernt aufeinanderfolgende Duplikate aus einem Behälter
reverse	Kehrt die Reihenfolge der Elemente in einem Behälter um
sort	Sortiert die Elemente eines Behälters in aufsteigender Reihenfolge
stable_sort	Sortiert die Elemente eines Behälters in aufsteigender Reihenfolge, aber behält die Reihenfolge von gleichen Elementen bei

2 Java

2.1 Eigenschaften

- **Objektorientiert** - Objekt und Klassenbasiert mit Strukturierung in Paketen
- **Interpretierte Ausführung** - Java wird in Bytecode kompiliert, der von der Java Virtual Machine(JVM) interpretiert wird.
- **Architekturunabhängig** - Java-Programme können auf jeder Plattform, die eine JVM hat, ausgeführt werden.
- **Dynamisch und verteilt** - Klassen werden zur Laufzeit vom Interpreter geladen, es gibt keine statische Bindung.
- **Einfach** - Nur unbedingt notwendige Features. Zeiger, Mehrfachvererbung und Operatorüberladung sind nicht erlaubt.
- **Robust** - Starke Typisierung, keine Zeigerarithmetik, Laufzeitprüfungen bei Feldern, Ausnahmenbehandlung und automatische Speicherverwaltung.
- **Sicher** - Java nutzt Sandboxing, das den Zugriff auf Systemressourcen einschränkt.
- **Hohe Effizienz** - Nur 10 mal langsamer als C ;). Mit JIT-Compiler kommt oft ausgeführter Code näher an C.
- **Leichtgewichtige Prozesse** - Threads sind leichtgewichtig durch einfache Synchronisation und Kommunikation.

Ein Programm besteht aus beliebig vielen Klassen. Jede public Klasse muss in einer eigenen Datei liegen, die den gleichen Namen wie die Klasse hat. Nur Klassen, Interfaces und Pakete haben globale Namen, es gibt keine globalen Variablen oder Funktionen. Ein Paket ist eine Sammlung von logisch zusammengehörigen Klassen, die in einem Verzeichnis liegen. Die package-Deklaration muss am Anfang der Datei stehen. Hierarchische Paketnamen werden 1 zu 1 auf Verzeichnisse abgebildet, z.B. *java.lang* ist im Verzeichnis *java/lang*. Pakete der Java Standardbibliothek werden über die Property *java.home* gefunden, Third-Party-Bibliotheken müssen über den Classpath angegeben werden.

Es gibt keinen Präprozessor, dadurch sind keine Makros, *#define* oder *#include* möglich. Stattdessen gibt es die *import*-Anweisung, die Klassen aus anderen Paketen importiert. Für Konstanten gibt es *static final* Membervariablen.

2.2 Datentypen

boolean ist kein Zahlenwert, also kann man nicht mehr damit rechnen. Es gibt keine unsigned Datentypen, also sind alle Ganzzahligen Datentypen vorzeichenbehaftet.

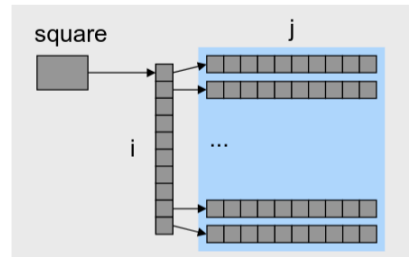
Statt Pointer gibt es Referenzen, diese werden automatisch dereferenziert und man kann nicht mit ihnen rechnen. Es gibt ein *null*-Referenz Literal, die auf kein Objekt zeigt.

Objekte werden nicht bei der Definition statisch angelegt, sondern mit dem *new*-Operator dynamisch im Heap angelegt. Es gibt kein delete, da der Speicher automatisch vom Garbage Collector freigegeben wird, wenn es keine Referenz auf ein Objekt mehr gibt.

Felder werden in Java mit mit eckigen Klammern angelegt. Es ist auch möglich, mehrdimensionale Felder nicht rechteckig zu machen, also dass die Zeilen unterschiedlich lang sind.

1. "Rechteckig" (*rectangular*), z. B.

```
double[][] square =
    new double[10][10];
square[i][j] = ...;
```



2. "Ausgefranst" (*ragged/jagged*), z. B.

```
double[][] triangle =
    new double[10][];
// triangle.length == 10, see next slide
for (int i = 0; i <= 9; i++) {
    triangle[i] =
        new double[i + 1];
    // triangle[i].length == i + 1
} // for
triangle[i][j] = ...;
```

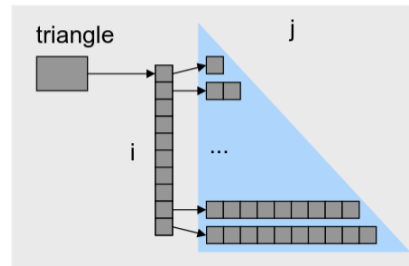


Abbildung 5: Felder in Java

Strings sind Objekte der Klasse *java.lang.String*, die immutable sind, also nicht verändert werden können. Alle Stringoperationen erzeugen ein neues String-Objekt. Für String-literale werden automatisch String-Objekte erzeugt. Wenn man Strings bearbeiten möchte ohne neue Objekte zu erzeugen, muss man *StringBuilder* oder *StringBuffer* verwenden.

"Normale" Exceptions (Checked Exceptions) müssen entweder behandelt werden oder in der Methodendeklaration mit *throws* angegeben werden. Unchecked Exceptions sind *RuntimeExceptions*, die nicht behandelt werden müssen.

2.2.1 Klassen und Objekte

¹ Initializer und Konstruktoren existieren.



Komponenten sind zugreifbar in wenn für Komponente gilt:			
	<i>public</i>	<i>protected</i>	<i>(package)</i>	<i>private</i>
... selber Klasse	ja	 ja	ja	ja
... anderer Klasse in gleichem Paket	ja	 ja	ja	nein
... abgeleit. Klasse in anderem Paket	ja	ja	nein	nein
... anderer Klasse in anderem Paket	ja	nein	nein	nein

Abbildung 6: Sichtbarkeiten in Java

Finalizer werden aufgerufen, wenn ein Objekt vom Garbage Collector freigegeben wird. Sie sind nicht garantiert, dass sie aufgerufen werden, also sollten sie nicht für wichtige Logik verwendet werden.²

Für Vererbung siehe Kapitel 4.

Mit *Super* kann auf die Basisklasse zugegriffen werden, wenn der Konstruktor der Basisklasse aufgerufen werden soll, muss *super()* im Konstruktor der abgeleiteten Klasse aufgerufen werden.

Membervariablen können versteckt werden, indem man eine Variable mit dem gleichen Namen in der abgeleiteten Klasse deklariert. Mit *this* und *super* kann auf die jeweilige Variable zugegriffen werden.

Fürs Überschreiben von Methoden muss vor der Methode *@Override* stehen. Methoden mit dem Schlüsselwort *final* können nicht mehr überschrieben werden.

2.2.2 Interfaces und Abstrakte Klassen

Klassen können als *abstract* deklariert werden, wenn sie nicht instanziiert werden sollen. Abstrakte Klassen können abstrakte Methoden enthalten, die in abgeleiteten Klassen implementiert werden muss. Abstrakte Klassen können auch normale oder *final* Methoden enthalten.

Interfaces sind pur abstrakte Klassen, die nur abstrakte Methoden enthalten dürfen.

Mehrfachvererbung ist in Java von Klassen nicht erlaubt, aber eine Klasse kann mehrere Interfaces implementieren.

2.2.3 Generizität

Bis Java 5.0 gab es in Java keine Generizität, also mussten alle Collections mit Polymorphismus implementiert werden (nur Object Collections).

In Java werden Generics mit dem *<>*-Operator implementiert. Dabei wird der Typ in spitzen Klammern angegeben, z.B. *List<String>* für eine Liste von Strings. Anders als in C# werden Generics in Java mit Type-Erasure implementiert, das heißt, dass die Implementierung weiterhin mit Objekten arbeitet und die Typinformationen nur zur Compilezeit geprüft werden können³.

¹ Ab hier wurde weniger Aufwand in die Zusammenfassung hineingesteckt. lol

² Schmutz

³ Ebenso Schmutz

2.2.4 Lamda-Ausdrücke und Streams

Lamda-Ausdrücke wurden in Java 8 eingeführt und werden mit dem `->`-Operator geschrieben. Mit ihnen kann man anonyme Funktionen erstellen, die zum Beispiel als Argumente an Methoden übergeben werden können.

Ein wichtiger Anwendungsfall sind Streams, die das Abarbeiten von Elementen in einer Collection vereinfachen. Man kann zum Beispiel mit einem Prädikat und der *filter*-Methode nur die Elemente einer Collection auswählen, die das Prädikat erfüllen. Mit der *map*-Methode kann man jedes Element einer Collection durch eine Funktion transformieren. Die *reduce*-Methode kann verwendet werden, um eine Collection auf einen einzigen Wert zu reduzieren, z.B. die Summe aller Elemente. *forEach* kann verwendet werden, um eine Funktion auf jedes Element einer Collection anzuwenden, ohne eine Schleife zu verwenden.⁴

2.3 JCF

Behältertyp	Beschreibung
Vererbung von <i>AbstractCollection</i> (Interface Collection)	
Vererbung von <i>AbstractList</i> (Interface List)	
ArrayList	Dynamisches Array, das hinten offen ist und schnellen direkten Zugriff ermöglicht
Vector	Dynamisches Array, das hinten offen ist und schnellen direkten Zugriff ermöglicht, aber synchronisiert ist
LinkedList	Einfach verkettete Liste, die schnelle Einfüge- und Löschoperationen ermöglicht, aber keinen schnellen Zugriff auf Elemente bietet
Vererbung von <i>AbstractSet</i> (Interface Set)	
HashSet	Implementierung einer Menge mit schneller Suche, Einfügen und Löschen
TreeSet	Implementierung einer Menge, die Elemente in sortierter Reihenfolge speichert
Vererbung von <i>AbstractMap</i> (Interface Map)	
HashMap	Implementierung einer Map, die Schlüssel-Wert-Paare speichert und schnellen Zugriff auf die Werte ermöglicht
TreeMap	Implementierung einer Map, die Schlüssel-Wert-Paare speichert und die Einträge in sortierter Reihenfolge hält

⁴Worse Linq

Algorithmus	Beschreibung
min und max	Finden des minimalen oder maximalen Wertes in einer Collection, die Comparable implementiert oder ein Comparator angegeben ist
sort	Sortiert die Elemente einer Collection in aufsteigender Reihenfolge, wenn Comparable implementiert ist oder ein Comparator angegeben ist
binarySearch	Sucht ein Element in einer sortierten Collection, die Comparable implementiert oder ein Comparator angegeben ist
reverse	Kehrt die Reihenfolge der Elemente in einer Collection um

3 Softwaremuster

3.1 Idioms

Ein Idiom ist ein sehr einfaches Muster das häufig für spezifische Sprachen und Probleme verwendet wird. Beispiele sind **Smart Pointer** in C++ oder **Properties** in C++, Java und C#.

3.2 MVC - Model-View-Controller

Graphische Oberflächen bestehen aus drei Teilen:

- **Model** - Das Model enthält die Daten und die Logik
- **View** - Die View ist die grafische Darstellung der Daten
- **Controller** - Der Controller ist die Schnittstelle zwischen Model und View.

Dadurch sind View und Model voneinander getrennt, was die Wartbarkeit und Testbarkeit erhöht. Der Controller kann auch mehrere Views bedienen, was die Wiederverwendbarkeit erhöht und die Views können unabhängig vom Model entwickelt werden.

3.3 Liskovsches Substituionsprinzip (LSP)

Das Liskovsches Substituionsprinzip besagt, dass eine abgeleitete Klasse überall dort verwendet werden kann, wo die Basisklasse verwendet wird, ohne dass das Verhalten des Programms verändert wird. Das bedeutet, dass die abgeleitete Klasse alle Methoden der Basisklasse implementieren muss und das Verhalten der Methoden nicht verändert werden darf. Es dürfen nur neue Methoden hinzugefügt werden, die das Verhalten der Basisklasse erweitern.

3.4 OOP Prinzipien

- **Program to an interface, not an implementation** - Interfaces sind wichtiger als die Implementierung, da man diese austauschen kann. Algorithmen sollten nicht auf konkrete Implementierungen angewiesen sein, sondern auf Interfaces.

- **Favor composition over inheritance** - Vererbung ist nicht immer die beste Lösung, da sie zu einer engen Kopplung zwischen Klassen führt. Stattdessen sollte man Komposition verwenden, um flexiblere und wiederverwendbare Komponenten zu erstellen.
- **Delegate wherever possible** - Delegation ist eine Form der Komposition, bei der eine Klasse die Verantwortung für bestimmte Aufgaben an eine andere Klasse delegiert. Dadurch wird die Verantwortung auf mehrere Klassen verteilt und die Wartbarkeit erhöht.

Vererbung	Komposition
<p>Neue Funktionalität durch Vererbung von Eigenschaften an abgeleitete Klasse, haben i. d. R. Zugriff auf Internas der Basisklasse (white-box reuse)</p> <p>Vorteile:</p> <ul style="list-style-type: none"> ▪ Statisch definiert ▪ Ermöglicht statische Typprüfung ▪ Direkt durch oo Programmiersprachen unterstützt ▪ Wiederverwendung durch Ableitung weiterer Klassen möglich <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Keine Anpassungen zur Laufzeit mehr möglich ▪ Basisklasse gibt abgeleiteter Klasse Implementierung vor 	<p>Neue Funktionalität durch geschickten Zusammenbau von passenden Objekten bereits vorhandener "kleiner" Klassen (black-box reuse)</p> <p>Vorteile:</p> <ul style="list-style-type: none"> ▪ Nur wenige Implementierungsabhängigkeiten ▪ Austauschbarkeit zur Laufzeit (dynamische Anpassung) ▪ Nutzung von Polymorphismus und dynamischer Bindung <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Explizite Objektkonstruktion muß zur Laufzeit erfolgen ▪ Erfordert sorgfältig entworfene Schnittstellen und hohe Granularität

Abbildung 7: Vererbung vs Komposition

3.5 Gang of Four

Die Gang of Four(GOF) ist eine Gruppe von 4 Autoren, die 1994 ein Buch über Design Patterns veröffentlicht haben. (Richard Helm, Ralph Johnson, John Vlissides und Erich Gamma) Sie unterteilen Muster in 4 Teile:

- **Name** - Ein oder zwei Worte, die das Muster beschreiben
- **Problem** - Eine kurze Beschreibung des Problems, das das Muster löst
- **Lösung** - Eine kurze Beschreibung der Lösung, die das Muster bietet
- **Konsequenzen** - Eine kurze Beschreibung der Konsequenzen, die das Muster hat(pros und cons)

Die GOF unterteilen Muster in 3 Kategorien:

- **Erzeugungsmuster** - Muster, die die Erzeugung von Objekten betreffen

- **Strukturmuster** - Muster, die die Struktur von Klassen und Objekten betreffen. Es wird zwischen Klassen- und Objektstrukturmustern unterschieden.
- **Verhaltensmuster** - Muster, die das Verhalten von Klassen und Objekten betreffen. Beschreiben die Dynamik des Zusammenwirkens.

3.6 Erzeugungsmuster

3.6.1 Abstract Factory oder Kit

Eine abstrakte Schnittstelle, die eine Familie von verwandten oder abhängigen Objekten erstellt, ohne Angabe der konkreten Klassen.

Dadurch kann man zum Beispiel verschiedene GUI-Elemente für verschiedene Plattformen erstellen, ohne dass der Code für die GUI-Elemente geändert werden muss. Eine konkrete Factory liefert dann die konkreten Objekte die verwendet werden sollen.

Ist anwendbar, wenn ein System unabhängig von der Art der Objekte sein soll, es aber Konfigurationen gibt, die unterschiedliche Arten von Objekten benötigen.

Konsequenzen sind die Isolation von konkreten Klassen von Anwendungen, die Austauschbarkeit von Produktfamilien, die Förderung von Konsistenz und eine erschwerte Erweiterbarkeit von Produktfamilien, da die Factory angepasst werden muss.

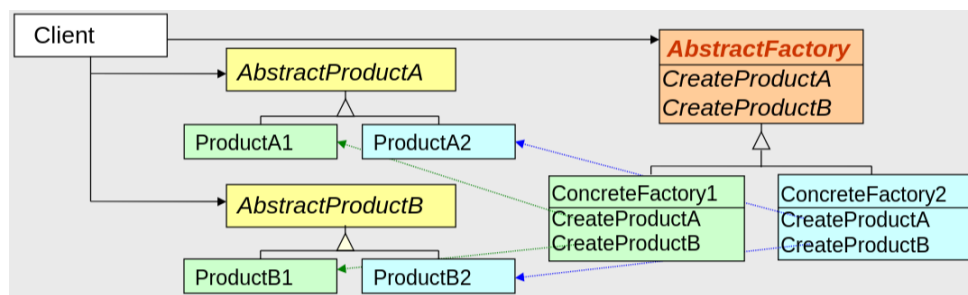


Abbildung 8: Abstract Factory

3.6.2 Builder

Trennung der Erzeugung eines komplexen Objekts von seiner Darstellung, sodass der gleiche Erzeugungsprozess verschiedene Darstellungen erzeugen kann.

3.6.3 Factory Method oder Virtual Constructor

Eine abstrakte Factory Method ist eine Methode, die ein Objekt erstellt, aber nicht angibt, welches Objekt erstellt wird. Dadurch kann ein abgeleitetes Objekt erst das konkrete Objekt erstellen.

Beispielsweise werden Iteratoren in vielen Programmiersprachen mit einer Factory Method erstellt. In der Basisklasse wird ein abstrakter Iterator deklariert und in der abgeleiteten Klasse wird die Factory Method implementiert, die den konkreten Iterator(z.B. ListIterator) erstellt.

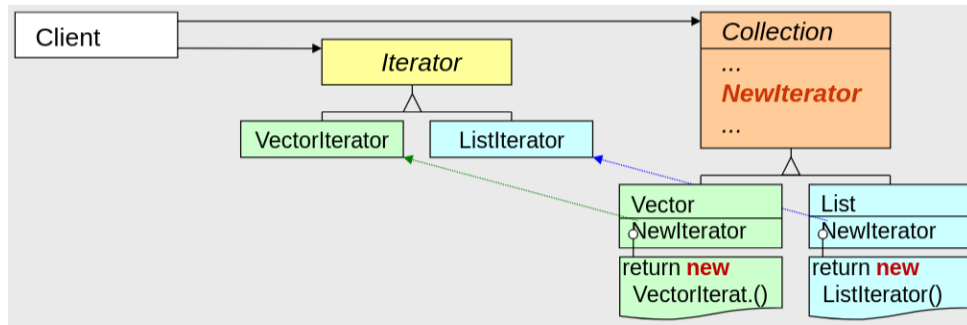


Abbildung 9: Factory Method

3.6.4 Prototype

Ein Prototyp ist ein Objekt, das als Vorlage für die Erstellung neuer Objekte dient. Neue Objekte werden durch Kopieren des Prototyps erstellt, anstatt sie von Grund auf neu zu erstellen.

3.6.5 Singleton

Klasse von der es nur eine Instanz gibt, die über eine statische Methode abgerufen werden kann.

Probleme mit Singletons sind die unübliche Erzeugung und dass sie nie freigegeben werden dürfen, da sie sonst nicht mehr verfügbar sind. Daher gibt es die Alternative eine **Monostate** Klasse zu verwenden, die mehrere Instanzen erlaubt, aber alle Instanzen den gleichen Zustand haben.

3.7 Strukturmuster

3.7.1 Adapter oder Wrapper

Konvertierung einer Schnittstelle in eine andere, die der Client erwartet. Dadurch kann eine Klasse mit einer inkompatiblen Schnittstelle verwendet werden.

Beispielweise kann eine Textbox als Shape dargestellt werden um die Kompatibilität mit einer Grafikbibliothek zu gewährleisten.

Verwendet wird das Muster, wenn eine Klasse mit einer inkompatiblen Schnittstelle verwendet werden soll, ohne die Klasse zu ändern.

In Java ist Integer ein Adapter für int, da es eine Klasse ist, die einen int-Wert kapselt und man dadurch int werte als Objekte verwenden kann.

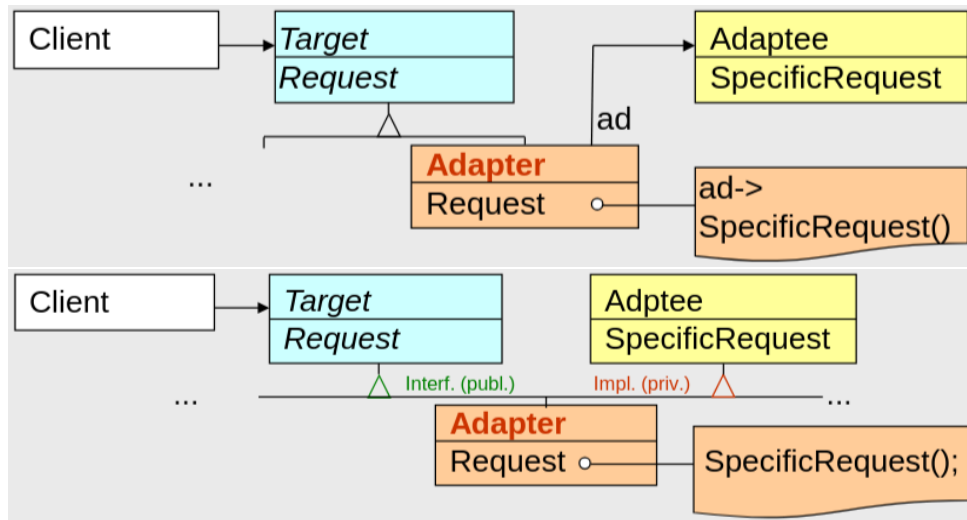


Abbildung 10: Adapter

3.7.2 Bridge

Trennung der Abstraktion von der Implementierung, sodass beide unabhängig voneinander variiert werden können.

3.7.3 Composite

Ein Composite wird genutzt um einen Baumstruktur zu modellieren, in der Knoten sowohl Blätter als auch andere Knoten sein können. Es wird eine gemeinsame Schnittstelle für Blätter und Knoten definiert, sodass der Client nicht zwischen Blättern und Knoten unterscheiden muss.

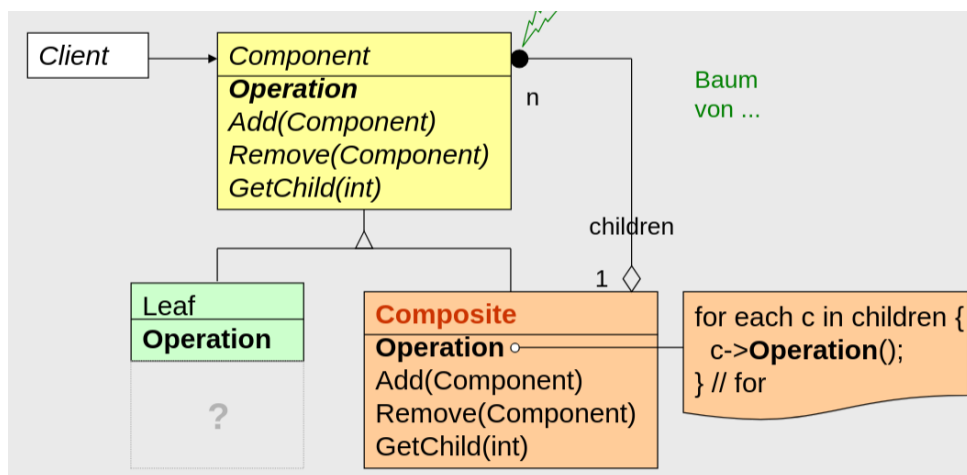


Abbildung 11: Composite

Ich weiß nicht warum darauf so viel eingegangen wird, da es ein recht einfaches Muster ist.

3.7.4 Decorator oder Wrapper

Zusätzliche Funktionalität zu einem Objekt hinzufügen, ohne die Klasse des Objekts zu ändern.

3.7.5 Facade

Einheitliche Schnittstelle für eine Gruppe von Klassen, die eine vereinfachte Schnittstelle für komplexe Systeme bereitstellt.

3.7.6 Flyweight oder Shared Object

Ein Flyweight ist ein Objekt, das gemeinsam genutzt wird, um den Speicherverbrauch zu reduzieren. Es wird verwendet, wenn viele Objekte mit ähnlichen Eigenschaften erstellt werden müssen.

3.7.7 Proxy

Ein Platzhalter für ein anderes Objekt, das den Zugriff auf das andere Objekt kontrolliert.

3.8 Verhaltensmuster

3.8.1 Chain of Responsibility

Trennung zwischen Sender und Empfänger, sodass der Sender nicht weiß, wer der Empfänger ist. Der Sender sendet eine Anfrage an einen Handler, der die Anfrage entweder verarbeitet oder an den nächsten Handler in der Kette weiterleitet. Dadurch kann die Kette von Handlern dynamisch geändert werden, ohne dass der Sender geändert werden muss.

3.8.2 Command

Kapselung einer Anfrage als Objekt, sodass Parameter an Methoden übergeben werden können, die die Anfrage ausführen. Dadurch kann die Anfrage in einer Warteschlange gespeichert, rückgängig gemacht oder protokolliert werden.

3.8.3 Interpreter

Ein Interpreter ist ein Muster, das eine Sprache mit einer Grammatik definiert und einen Interpreter implementiert, der die Sprache ausführt.

3.8.4 Iterator

Auf Iteratoren wurde bereits im STL Kapitel eingegangen.

3.8.5 Mediator

Ein Mediator ist ein Objekt, das die Kommunikation zwischen mehreren Objekten koordiniert, ohne dass die Objekte direkt miteinander kommunizieren. Dadurch wird die Kopplung zwischen den Objekten reduziert und die Wartbarkeit erhöht.

3.8.6 Memento

Ein Memento ist ein Objekt, das den Zustand eines Objekts zu einem bestimmten Zeitpunkt speichert, sodass der Zustand später wiederhergestellt werden kann. Das Memento-Muster wird verwendet, um den Zustand eines Objekts zu speichern, ohne dass das Objekt selbst den Zustand speichern muss.

3.8.7 Observer oder Publisher/Subscriber

Ein Observer ist ein Objekt, das über Änderungen an einem anderen Objekt informiert wird. Das Observer-Muster wird verwendet, um eine Eins-zu-viele-Abhängigkeit zwischen Objekten zu definieren, sodass bei einer Änderung des einen Objekts alle abhängigen Objekte benachrichtigt werden. Die Observer bekommen beim Update nicht gleich den neuen Zustand, sondern nur eine Benachrichtigung, dass sich etwas geändert hat. Sie rufen dann selbst den Zustand des Objekts ab, um die Änderungen zu erhalten.

3.8.8 State

Ein State-Muster ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sich sein interner Zustand ändert. Das Objekt wird so behandelt, als hätte es eine andere Klasse. So kann das Verhalten zur Laufzeit geändert werden, ohne dass die Klasse geändert werden muss.

3.8.9 Strategy

Ein Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie und macht sie untereinander austauschbar. Wenn Algorithmen gleiche Schnittstellen haben, können sie zur Laufzeit ausgetauscht werden, ohne dass der Client-Code geändert werden muss.

3.8.10 Template Method

Ein Template Method-Muster definiert das Gerüst eines Algorithmus in einer Methode, während die Unterklassen einige Schritte des Algorithmus implementieren können. Das Template Method-Muster ermöglicht es, die Struktur eines Algorithmus zu definieren, während die spezifischen Implementierungen in den Unterklassen erfolgen.

3.8.11 Visitor

Operation, die auf die Elemente einer Sammlung anzuwenden ist.

3.9 Weitere Arten von Mustern

- Metamuster
- Analysemuster
- Organisationsmuster
- Pädagogische Muster
- Anti-Muster

3.10 Softwarearchitektur

Softwarearchitektur beschäftigt sich mit Abstraktion, Zerlegung, Zusammenbau, Stil und Ästhetik von Software.

Das Software Engineering Institut(SEI) an der Carnegie Mellon University führt über 200 weitere Definitionen, bei denen die Gemeinsamkeit ist, dass Softwarearchitektur immer nur aus verschiedenen Sichten dargestellt werden kann.

3.10.1 Sichten in der Softwarearchitektur

- **Konzeptuelle-Sicht** - Wie funktioniert das System? Use Cases
- **Infrastruktur-Sicht** - Wie ist das System aufgebaut? Hardware, Netzwerke, Betriebssysteme
- **Implementierungs-Sicht** - Wie ist das System implementiert? Programmiersprachen, Frameworks, Bibliotheken
- **Laufzeit-Sicht** - Wie ist der Ablauf des Systems? Welche Objekte gibt es zur Laufzeit? Wie interagieren sie?
- **Daten-Sicht** - Welche Daten bearbeitet das System? Von wo kommen sie, welches Format haben sie, wie werden sie gespeichert?
- **4+1-Sichtenmodell** - Fasst 5 Sichten zusammen um einen Überblick über das System zu geben.
 - **Logische-Sicht** - Funktion
 - **Entwicklungs-Sicht** - Statik
 - **Prozess-Sicht** - Dynamik
 - **Physische-Sicht** - Hardware
 - **Szenarien-Sicht** - Use Cases
- **C4-Modell** - Basiert auf dem 4+1-Sichtenmodell und ist im Wesentlichen eine Notationstechnik zur Visualisierung von Softwarearchitektur.
 - **Context Diagram** - Zeigt das System im Kontext seiner Umgebung, z.B. externe Systeme, Benutzer und Schnittstellen.
 - **Container Diagram** - Zeigt die Container des Systems, z.B. Webanwendungen, Datenbanken und Microservices.
 - **Component Diagram** - Zeigt die Komponenten eines Containers, z.B. Klassen, Module und Bibliotheken.
 - **Code Diagram** - Zeigt die Klassen und deren Beziehungen innerhalb einer Komponente.

4 Syntaxvergleich

Um den Syntax von C++ und Java zu vergleichen, hab ich ein kleines Beispielprogramm geschrieben mit:

- einem Interface
- einer Basisklasse, die das Interface implementiert
- einer abgeleiteten Klasse, die von der Basisklasse erbt
- einer Iteratorimplementierung in der abgeleiteten Klasse
- einer Main-Methode, die ein Objekt der abgeleiteten Klasse verwendet

4.1 C++

4.1.1 Interface

```
1  template <typename T> // Generic programming with templates
2  class interface // in C++ there is no special keyword for interfaces,
   ↪ so we use a class
3  {
4  public:
5      virtual ~interface() = default; // Virtual destructor for proper
   ↪ cleanup of derived classes
6
7      virtual T getValue(int index) const = 0; // Pure virtual function,
   ↪ must be implemented by derived classes
8      virtual T operator[](int index) const = 0; // operator overload for
   ↪ indexing, must be implemented by derived classes
9      virtual void addValue(T value) = 0; // Pure virtual function, must
   ↪ be implemented by derived classes
10     virtual void operator+=(T value) = 0; // operator overload for
   ↪ adding a value, must be implemented by derived classes
11     virtual int getSize() const = 0; // Pure virtual function, must be
   ↪ implemented by derived classes
12 };
```

4.1.2 Basisklasse

```
1  #include "interface.h"
2
3  template <typename T>
4  class baseclass : public interface<T>
5  {
6  private:
7      int _size;
8
9  public:
```

```

10     baseclass(int size = 10) : _size(size) {} // Constructor with default
    ↪ parameter
11
12     virtual int getSize() const override
13     {
14         return _size;
15     }
16 };

```

4.1.3 Abgeleitete Klasse

```

1  #include "baseclass.h"
2  #include <iostream>
3
4  template <typename T>
5  class derivedclass : public baseclass<T>
6  {
7      friend std::ostream& operator<<(std::ostream& os, const
    ↪ derivedclass<T>& obj)
8      {
9          for (int i = 0; i < obj._currentIndex; ++i)
10         {
11             os << obj._data[i] << " "; // Access _data directly since
    ↪ it's a friend
12         }
13
14         return os;
15     }
16
17 private:
18     T* _data;
19     int _currentIndex;
20
21 public:
22     // Constructor calls base constructor with size parameter and
    ↪ initializes _currentIndex
23     derivedclass(int size = 10) : baseclass<T>(size), _currentIndex(0)
24     {
25         // Allocate memory for _data based on the size provided
26         _data = new T[size];
27     }
28
29     // Destructor to clean up allocated memory
30     virtual ~derivedclass()
31     {
32         delete[] _data; // Free the allocated memory for _data
33     }
34

```

```

35 // copy constructor to create a deep copy of the derivedclass
36 derivedclass(const derivedclass& other) :
    ↳ baseclass<T>(other.getSize()),
    ↳ _currentIndex(other._currentIndex)
37 {
38     // Allocate memory for _data and copy the values from the other
    ↳ instance
39     _data = new T[other.getSize()];
40     for (int i = 0; i < other.getSize(); ++i)
41     {
42         _data[i] = other._data[i];
43     }
44 }
45
46 // move constructor to transfer ownership of resources
47 derivedclass(derivedclass&& other) noexcept :
    ↳ baseclass<T>(other.getSize())
48 {
49     // Transfer ownership of the data pointer and current index
    ↳ from the other instance
50     _data = other._data;
51     _currentIndex = other._currentIndex;
52
53     // Set the other instance's data pointer to nullptr to avoid
    ↳ double deletion
54     other._data = nullptr;
55     other._currentIndex = 0; // Reset the index of the moved-from
    ↳ instance
56 }
57
58 // copy assignment operator to create a deep copy of the
    ↳ derivedclass
59 derivedclass& operator=(const derivedclass& other)
60 {
61     if (this != &other) // Check for self-assignment
62     {
63         // Clean up existing resources
64         delete[] _data;
65
66         // Allocate new memory and copy the values from the other
        ↳ instance
67         _data = new T[other.getSize()];
68         for (int i = 0; i < other.getSize(); ++i)
69         {
70             _data[i] = other._data[i];
71         }
72
73         _currentIndex = other._currentIndex;

```



```

74     }
75
76     return *this;
77 }
78
79 // move assignment operator to transfer ownership of resources
80 derivedclass& operator=(derivedclass&& other) noexcept
81 {
82     if (this != &other) // Check for self-assignment
83     {
84         // Clean up existing resources
85         delete[] _data;
86
87         // Transfer ownership of the data pointer and current index
88         ↪ from the other instance
89         _data = other._data;
90         _currentIndex = other._currentIndex;
91
92         // Set the other instance's data pointer to nullptr to
93         ↪ avoid double deletion
94         other._data = nullptr;
95         other._currentIndex = 0; // Reset the index of the
96         ↪ moved-from instance
97     }
98
99     return *this;
100 }
101
102 // Override getValue to return the value at the current index
103 virtual T getValue(int index) const override
104 {
105     if (index < 0 || index >= this->getSize())
106     {
107         throw std::out_of_range("Index out of range");
108     }
109
110     return _data[index];
111 }
112
113 // Override operator[] to provide access to the value at the
114 ↪ current index
115 virtual T operator[](int index) const override
116 {
117     return getValue(index); // Use getValue to access the value at
118     ↪ the index
119 }

```

```

116 // Override addValue to add a value at the current index and
    ↪ increment the index
117 virtual void addValue(T value) override
118 {
119     if (_currentIndex < 0 || _currentIndex >= this->getSize())
120     {
121         throw std::out_of_range("Index out of range");
122     }
123
124     _data[_currentIndex] = value; // Store the value at the current
    ↪ index
125     _currentIndex++; // Increment the index for the next value
126 }
127
128 // Override operator+= to add a value at the current index and
    ↪ increment the index
129 virtual void operator+=(T value) override
130 {
131     addValue(value); // Use addValue to handle the addition and
    ↪ index increment
132 }
133
134 class iterator
135 {
136 private:
137     T* _ptr; // Pointer to the data
138     int _index; // Current index in the data array
139
140 public:
141     // Constructor to initialize the iterator with a pointer
142     iterator(T* ptr, int index) : _ptr(ptr), _index(index) {}
143
144     // Overload the dereference operator to return the value at the
    ↪ current index
145     T& operator*() const
146     {
147         return _ptr[_index];
148     }
149
150     // Overload the increment operator to move to the next index
151     iterator& operator++()
152     {
153         _index++;
154         return *this;
155     }
156
157     // Overload the equality operator to compare two iterators
158     bool operator==(const iterator& other) const

```

```

159     {
160         return _index == other._index && _ptr == other._ptr;
161     }
162
163     // Overload the inequality operator to compare two iterators
164     bool operator!=(const iterator& other) const
165     {
166         return !(*this == other);
167     }
168
169 };
170
171 iterator begin()
172 {
173     return iterator(_data, 0); // Return an iterator pointing to
174     ↪ the start of the data
175 }
176
177 iterator end()
178 {
179     return iterator(_data, _currentIndex); // Return an iterator
180     ↪ pointing to the end of the data
181 }
182 };

```

4.1.4 Main

```

1  #include "derivedclass.h"
2  #include <iostream>
3
4  int main()
5  {
6      try
7      {
8          // Create an instance of derivedclass with a size of 5
9          derivedclass<int> myDerived(5);
10
11         // Add some values to the derived class
12         myDerived.addValue(10);
13         myDerived.addValue(20);
14         myDerived += 30;
15
16         // Access values using operator[]
17         std::cout << "Value at index 0: " << myDerived[0] << std::endl;
18         std::cout << "Value at index 1: " << myDerived[1] << std::endl;
19         std::cout << "Value at index 2: " << myDerived.getValue(2) <<
20         ↪ std::endl;

```

```

21     // Demonstrate copy constructor
22     derivedclass<int> copiedDerived(myDerived);
23     std::cout << "Copied: " << copiedDerived << std::endl;
24
25     // Demonstrate move constructor
26     derivedclass<int> movedDerived(std::move(myDerived));
27     std::cout << "Moved: " << movedDerived << std::endl;
28
29     // Iterate through the values using the iterator
30     std::cout << "Iterating through values: ";
31     for (auto i : movedDerived)
32     {
33         std::cout << i << " ";
34     }
35     std::cout << std::endl;
36 }
37 catch (const std::exception& e)
38 {
39     std::cerr << "Exception: " << e.what() << std::endl;
40 }
41
42 return 0;
43 }

```

4.2 Java

4.2.1 Interface

```
1 package javademo;
2
3 // Java supports interfaces and generics but not operator overloading
4 public interface Interface<T> {
5     T getValue(int index);
6
7     void addValue(T value);
8
9     int getSize();
10    // No operator[] or operator+= in Java
11 }
```

4.2.2 Basisklasse

```
1 package javademo;
2
3 public abstract class BaseClass<T> implements Interface<T> {
4     private int size;
5
6     BaseClass(int size) {
7         this.size = size;
8     }
9
10    @Override
11    public int getSize() {
12        return size;
13    }
14
15    // Java supports not implementing interfaces in abstract classes
16 }
```

4.2.3 Abgeleitete Klasse

```
1 package javademo;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5 import java.util.NoSuchElementException;
6
7 public class DerivedClass<T> extends BaseClass<T> implements
8     ⇨ Iterable<T> {
9     private ArrayList<T> data;
10    private int currentIndex;
```

```

11 public DerivedClass(int size) {
12     super(size);
13     data = new ArrayList<>(size); // generic arrays in Java are
    ↪ semi-supported
14     currentIndex = 0;
15 }
16
17 @Override
18 public T getValue(int index) {
19     if (index < 0 || index >= getSize()) {
20         throw new IndexOutOfBoundsException();
21     }
22
23     return data.get(index);
24 }
25
26 @Override
27 public void addValue(T value) {
28     if (currentIndex < 0 || currentIndex >= getSize()) {
29         throw new IndexOutOfBoundsException();
30     }
31
32     data.add(value);
33     currentIndex++;
34 }
35
36 @Override
37 public String toString() {
38     StringBuilder sb = new StringBuilder();
39
40     for (int i = 0; i < currentIndex; i++) {
41         sb.append(data.get(i));
42         sb.append(" ");
43     }
44
45     return sb.toString().trim();
46 }
47
48 // Iterable implementation for for-each loop
49 @Override
50 public Iterator<T> iterator() {
51     return new Iterator<T>() {
52         private int index = 0;
53
54         @Override
55         public boolean hasNext() {
56             return index < currentIndex;
57         }

```

```

58
59         @Override
60         public T next() {
61             if (!hasNext()) {
62                 throw new NoSuchElementException();
63             }
64
65             return data.get(index++);
66         }
67     };
68 }
69
70 // No operator overloading in Java, so no operator[] or operator+=
71 // No need for destructors or manual memory management
72 }

```

4.2.4 Main

```

1  package javademo;
2
3  public class Demo {
4      public static void main(String[] args) {
5          DerivedClass<Integer> myDerived = new DerivedClass<>(5);
6          myDerived.addValue(10);
7          myDerived.addValue(20);
8          myDerived.addValue(30);
9
10         System.out.println("Value at index 0: " +
11             ↳ myDerived.getValue(0));
12         System.out.println("Value at index 1: " +
13             ↳ myDerived.getValue(1));
14         System.out.println("Value at index 2: " +
15             ↳ myDerived.getValue(2));
16
17         System.out.println("toString: " + myDerived.toString());
18
19         System.out.print("Iterating through values: ");
20         for (int value : myDerived) {
21             System.out.print(value + " ");
22         }
23     }
24 }

```

4.3 C# kommt nicht zur Klausur - hab ich zu spät gesehen

4.3.1 Interface

```
1 namespace demo;
2
3 // C# supports interfaces and generics and operator overloading(only
4   ↳ indexers are supported in interfaces)
5 public interface IInterface<T>
6 {
7     T GetValue(int index);
8
9     void AddValue(T value);
10
11     T this[int index] { get; }
12 }
```

4.3.2 Basisklasse

```
1 namespace demo;
2
3 public abstract class BaseClass<T>(int size) : IInterface<T>
4 {
5     private readonly int _size = size;
6
7     public int Size => _size; // readonly property for size
8
9     // the interface methods need to be mentioned in the abstract
10    ↳ class
11    public abstract T GetValue(int index);
12
13    public abstract void AddValue(T value);
14
15    public abstract T this[int index] { get; }
16 }
```

4.3.3 Abgeleitete Klasse

```
1 using System.Collections;
2 using System.Text;
3
4 namespace demo;
5
6 // IEnumerable is commonly used for iteration in C#, but it is not
7   ↳ required because foreach can be used with any class that implements
8   ↳ GetEnumerator(Duck Typing).
9 public class DerivedClass<T> : BaseClass<T>, IEnumerable<T>
10 {
```



```

9     private readonly T[] data;
10    private int currentIndex;
11
12    // could also be a primary constructor like in the base class
13    public DerivedClass(int size) : base(size)
14    {
15        data = new T[size];
16        currentIndex = 0;
17    }
18
19    public override T GetValue(int index)
20    {
21        if (index < 0 || index >= Size)
22        {
23            throw new IndexOutOfRangeException();
24        }
25
26        return data[index];
27    }
28
29    public override void AddValue(T value)
30    {
31        if (currentIndex < 0 || currentIndex >= Size)
32        {
33            throw new IndexOutOfRangeException();
34        }
35
36        data[currentIndex++] = value;
37    }
38
39    // C# supports indexers
40    public override T this[int index]
41    {
42        get { return GetValue(index); }
43    }
44
45    public static DerivedClass<T> operator +(DerivedClass<T> obj, T
    ↪ value)
46    {
47        obj.AddValue(value);
48        return obj;
49    }
50
51    public override string ToString()
52    {
53        var stringBuilder = new StringBuilder();
54
55        for (int i = 0; i < currentIndex; i++)

```

```

56     {
57         stringBuilder.Append($"{data[i]} ");
58     }
59
60     return stringBuilder.ToString().Trim();
61 }
62
63 public IEnumerator<T> GetEnumerator()
64 {
65     for (int i = 0; i < currentIndex; i++)
66     {
67         yield return data[i];
68     }
69 }
70
71 // The non-generic version of GetEnumerator needs to be implemented
72 ↪ explicitly
73 IEnumerator IEnumerable.GetEnumerator()
74 {
75     return GetEnumerator();
76 }
77
78 // No need for destructors or manual memory management
79 }

```

4.3.4 Main

```

1  namespace demo;
2
3  static class Program
4  {
5      static void Main()
6      {
7          var myDerived = new DerivedClass<int>(5);
8          myDerived.AddValue(10);
9          myDerived.AddValue(20);
10         myDerived += 30; // Using operator overloading
11
12         Console.WriteLine($"Value at index 0: {myDerived[0]}");
13         Console.WriteLine($"Value at index 1: {myDerived[1]}");
14         Console.WriteLine($"Value at index 2: {myDerived.GetValue(2)}");
15
16         Console.WriteLine($"ToString: {myDerived}"); // ToString is
17         ↪ called implicitly
18
19         // Iteration using foreach
20         Console.Write("Iterating through values: ");
21         foreach (var value in myDerived)

```

```
21     {
22         Console.Write($"{value} ");
23     }
24     Console.WriteLine();
25 }
26 }
```