



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Programmierpraktikum Technische Informatik (C++)



19.5.2022



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Vererbung

Vererbung

- Häufig teilen sich verschiedene Klassen identische Funktionalität
 - Ansprechen über gemeinsames Interface wünschenswert
 - Wiederverwendung von identischen Codepfaden
- In objektorientierter Programmierung über Vererbung gelöst
- Definiert ein **is-a**-(ist-ein)-Verhältnis
 - Abgeleitete Klassen übernehmen alle Member (Daten und Funktionen) ihrer Basisklasse
 - Instanz der abgeleiteten Klasse als Instanz der Basisklasse verwendbar
- Bereits bekannt in Form von IOStreams
 - `fstream` ist ein `iostream` ist ein `ostream`

Vererbung

- Häufig teilen sich verschiedene Klassen identische Funktionalität
 - Ansprechen über gemeinsames Interface wünschenswert
 - Wiederverwendung von identischen Codepfaden

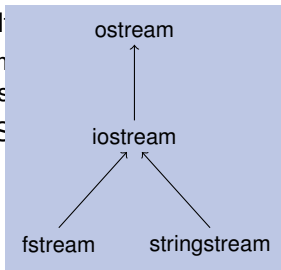
- In objektorientierter Programmierung über Vererbung gelöst

- Definiert ein **is-a**-(ist-ein)-Verhältnis

- Abgeleitete Klassen übernehmen (Attribute und Funktionen) ihrer Basisklasse
 - Instanz der abgeleiteten Klasse kann als Instanz der Basisklasse verwendbar

- Bereits bekannt in Form von IOS

- `fstream` ist ein `iostream` ist



Vererbung und Sichtbarkeit

- Abgeleitete Klassen erben alle Member der Basisklasse
- Bedeutet nicht, dass sie auch auf alle zugreifen können
- Methoden aus abgeleiteten Klassen gehören **nicht** zur Basisklasse
- Kapselung
 - Kein Zugriff auf `private`-Daten der Basisklasse
 - Kapselung von Basisklasseninterna gegenüber der abgeleiteten Klassen
- Zugriff auf Interna einer Basisklasse aus abgeleiteter Klasse manchmal notwendig
 - Kann über `friend` gelöst werden, ist aber keine skalierende Lösung
- Die Sichtbarkeit `protected` ist für diesen Zweck vorgesehen

protected

- `protected` wie `private` mit einer Ausnahme:
Memberfunktionen abgeleiteter Klassen dürfen auf `protected` Member der Basisklasse zugreifen
- Nützlich, um abgeleiteten Klassen die Anpassung von Interna der Basisklasse zu erlauben

Sichtbarkeit	Zugriff aus			
	der Klasse selbst	Friends	Abgeleiteter Klasse	Sonstigem Code
<code>private</code>	ja	ja	nein	nein
<code>protected</code>	ja	ja	ja	nein
<code>public</code>	ja	ja	ja	ja

Beispiel für Vererbung

```
class Expression {
private:
    std::string type;
public:
    using Ptr = std::unique_ptr<Expression>;
    const std::string& getType() const { return this->type; }
    Expression(std::string type): setType{std::move(type)} {}
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};

class Addition: public Expression {
private:
    Expression::Ptr left;
    Expression::Ptr right;
public:
    Addition(Expression::Ptr l, Expression::Ptr r)
        : Expression{"Add"}, left{std::move(l)}, right{std::move(r)}
    {}
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};
```

Beispiel für Vererbung

```
class Constant: public Expression {
private:
    int value;
public:
    Constant(int val): Expression{"Constant"}, value{val} {}
    int evaluate() const override { return this->value; }
};

void printType(std::ostream& os, const Expression& expr)
{ os<<expr.getType(); }
//...

Constant c{5};
printType(std::cout, c);
std::cout<<" "<<c.getType()<<std::endl;
```

- Basisklassen werden in der Klassendefinition mit
: *Sichtbarkeit Basisklassenname* angegeben
- Entspricht Angabe als extends in Java
- Im Gegensatz zu Java wird in C++ eine Sichtbarkeit für die Basisklasse angegeben

Zugriffskontrolle bei Vererbung

- Angegebene Sichtbarkeit bestimmt, mit welcher Sichtbarkeit geerbte Member übernommen werden

```
class A {...};
```

```
class B: public A  
{...};
```

```
class C: protected A  
{...};
```

```
class D: private A  
{...};
```

Ist Member in A	public	protected	private
so wird es in B	public	protected	private
so wird es in C	protected	protected	private
so wird es in D	private	private	private

- Achtung:** Nur **public**-Vererbung definiert eine is-a-Beziehung
- protected** und **private** Vererbung nur sinnvoll, wenn Zugriff auf **protected**-Member der Basisklasse notwendig, ohne dass eine is-a-Beziehung vorliegt
 - Für änderbare Basisklassen ist möglicherweise **friend** sinnvoller

Üblicherweise **public**-Vererbung verwenden, andere nur in Ausnahmefällen!

Beispiel für Vererbung

```
class Constant: public Expression {
private:
    int value;
public:
    Constant(int val): Expression{"Constant"}, value{val} {}
    int evaluate() const override { return this->value; }
};

void printType(std::ostream& os, const Expression& expr)
{ os<<expr.getType(); }
//...

Constant c{5};
printType(std::cout, c);
std::cout<<" "<<c.getType()<<std::endl;
```

- Aufruf des Konstruktors der Basisklasse wie für Member in der Initialisierungsliste
 - Ohne expliziten Basiskonstruktoraufruf wird Defaultkonstruktor (falls vorhanden) aufgerufen
- Basis ist vor allen anderen Membern definiert, daher erstes Element der Initialisierungsliste

Beispiel für Vererbung

```
class Constant: public Expression {
private:
    int value;
public:
    Constant(int val): Expression{"Constant"}, value{val} {}
    int evaluate() const override { return this->value; }
};
void printType(std::ostream& os, const Expression& expr)
{ os<<expr.getType(); }
//...
Constant c{5};
printType(std::cout, c);
std::cout<<" "<<c.getType()<<std::endl;
```

- Objekt vom Typ Constant ist auch Instanz von Expression
- Kann als Expression verwendet werden
 - Übergabe an Funktion, die ein Objekt der Basisklasse erwartet

Beispiel für Vererbung

```
class Constant: public Expression {
private:
    int value;
public:
    Constant(int val): Expression{"Constant"}, value{val} {}
    int evaluate() const override { return this->value; }
};
void printType(std::ostream& os, const Expression& expr)
{ os<<expr.getType(); }
//...
Constant c{5};
printType(std::cout, c);
std::cout<<" "<<c.getType()<<std::endl;
```

- Objekt vom Typ Constant ist auch Instanz von Expression
- Kann als Expression verwendet werden
 - Übergabe an Funktion, die ein Objekt der Basisklasse erwartet
 - Direkter Zugriff auf Member der Basisklasse über eine abgeleitete Klasse

Polymorphie

```
class Expression {
    //...
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};
class Addition: public Expression {
    //...
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};
class Constant: public Expression {
    //...
    int evaluate() const { return this->Value; }
};
```

- Manchmal ist in abgeleiteten Klassen eine Änderung des Verhaltens von geerbten Methoden wünschenswert
 - Wird als Polymorphie bezeichnet
- In abgeleiteter Klasse Methode mit identischem Namen und gleicher Signatur erstellen
 - Erreicht nicht ganz das gewünschte Verhalten

Virtuelle Methoden

- Methoden normalerweise statisch an den jeweiligen Typ gebunden
 - Aufruf über Pointer auf Basisklasse ruft Methode der Basisklasse und nicht die Methode der abgeleiteten Klasse auf
- Als **virtual** deklarierte Methoden sind dynamisch gebunden
 - Methode wird immer aus der abgeleiteten Klasse genommen
 - **Ausnahme:** In Konstruktoren
(In Konstruktoren keine virtuellen Methoden aufrufen!)
- In Java sind alle Methoden **virtual**
- Warum nicht auch in C++?
 - Performancegründe
 - **virtual** erzeugt zusätzliche Kosten pro Aufruf, verhindert inlining
 - Dynamische Typinformationen kosten zusätzlichen Speicher
 - Eiserner Grundsatz von C++: Don't pay for what you don't need

Polymorphie

```
class Expression {  
    //...  
    virtual int evaluate() const = 0;  
    virtual ~Expression(){}  
};  
class Addition: public Expression {  
    //...  
    int evaluate() const override  
    { return this->left->evaluate() + this->right->evaluate(); }  
};  
class Constant: public Expression {  
    //...  
    int evaluate() const { return this->Value; }  
};
```

- Mit **virtual** wird eine Methode in der Basisklasse als virtuell markiert
- In abgeleiteter Klasse wird **virtual** nicht explizit angegeben
 - Methode mit gleicher Signatur und gleichem Namen überschreibt automatisch die virtuelle Basismethode
 - Signatur muss exakt übereinstimmen

Polymorphie

```
class Expression {
    //...
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};
class Addition: public Expression {
    //...
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};
class Constant: public Expression {
    //...
    int evaluate() const { return this->Value; }
};
```

- Bei Änderungen können sich Signaturen unbeabsichtigt unterscheiden
 - Kompiliert meistens, führt aber zu unerwartetem Verhalten zur Laufzeit
- Dafür override: Signalisiert, dass virtuelle Basisklassenmethode überschrieben wird
 - Kompilierfehler, falls nicht erfüllt
 - Verwendung von override ist optional, aber empfohlen

Polymorphie

```
class Expression {
    //...
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};

class Addition: public Expression {
    //...
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};

class Constant: public Expression {
    //...
    int evaluate() const { return this->Value; }
};
```

- Normalerweise müssen auch virtuelle Methoden definiert sein (sonst Linkerfehler)
- Häufig ist in der Basisklasse aber keine sinnvolle Implementation möglich
 - Mit `= 0`; wird eine **virtuelle** Methode als abstrakt gekennzeichnet
 - Bedeutet, dass in der Basisklasse keine Implementation vorliegt
 - Entspricht dem `abstract`-Keyword in Java
 - Basisklasse wird automatisch abstrakt und kann nicht instantiiert werden

Polymorphie

```
class Expression {
    //...
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};
class Addition: public Expression {
    //...
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};
class Constant: public Expression {
    //...
    int evaluate() const { return this->Value; }
};
void printEval(const Expression& expr) { std::cout<<expr.evaluate()<<"\n"; }
//...
printEval(Addition{...});
printEval(Constant{...});
```

- Expression ist Basisklasse und soll polymorph verwendet werden
- Es wird deshalb ein virtueller Destruktor benötigt

Destruktoren

- Aus der letzten Vorlesung bekannt:
Objekte können am Ende ihrer Lebenszeit hinter sich aufräumen
- Dazu wird der sog. Destruktor aufgerufen
- Membermethode mit speziellem Namen: `~ClassName()`
 - Nichtinline-Definition: `ClassName::~~ClassName()`
 - Muss für die Zerstörung sichtbar sein, also nicht `private` deklarieren
- Wird automatisch aufgerufen, sobald die Lebenszeit eines Objektes endet
 - Manueller Aufruf möglich, aber nur in Ausnahmefällen korrekt
- Destruktoren dürfen keine Parameter haben
 - Könnten sonst nicht automatisch aufgerufen werden
- Destruktor wird direkt vor der Zerstörung eines Objektes aufgerufen
 - Zugriff auf Member und Methoden ist im Destruktor noch möglich

Destruktoren II

- Aufruf der Destruktoren von Memberobjekten nach Ausführung des Destruktors der Klasse
 - Automatisch in umgekehrter Reihenfolge der Deklaration am Ende des Destruktoraufrufs
- Destruktoren werden bei automatischen Objekten immer in umgekehrter Reihenfolge der Konstruktoren aufgerufen
- Ist kein Destruktor deklariert, wird er automatisch vom Compiler generiert
 - Compilergenerierter Destruktor führt keine Operationen aus
 - Destruktoren von Members werden normal aufgerufen
- Manuelle Implementation selten notwendig
 - Nur wenn die Klasse manuell Ressourcen managed
 - Und für Klassen, deren Destruktoren Seiteneffekte haben

Destruktoren und Exceptions

- Aufruf von Destruktoren immer beim Verlassen des jeweiligen Scopes
 - Auch wenn der Scope über Exceptions verlassen wurde
- Können somit während der Exceptionbehandlung ausgeführt werden
- Was passiert wenn ein Destruktor während der Behandlung einer Exception eine weitere wirft?
- In C++ über einfache Regel gelöst:
Destruktoren dürfen keine Exceptions werfen
 - Bei Missachtung wird das Programm abrupt beendet
 - Aufruf von werfenden Funktionen ist erlaubt, Exceptions müssen aber im Destruktor gefangen werden
 - Einhaltung ist Aufgabe des Entwicklers

Virtuelle Destruktoren

- Destruktoren sind wie normale Methoden statisch gebunden
 - Problematisch bei polymorpher Verwendung:
`delete` bei einem Pointer auf die Basisklasse zerstört nur den Basisklassenanteil
- Destruktoren können auch als `virtual` deklariert werden
 - Muss in dem Fall auch manuell definiert werden
 - In abgeleiteten Klassen wird der Destruktor automatisch `virtual`, falls dies in der Basisklasse gilt
 - Manuelle Implementation in abgeleiteten Klassen in der Regel nicht notwendig
- Erlaubt sicheres Löschen polymorpher Objekte

Destruktoren von Basisklassen

- ~~Vermutung: Basisklassen sollten immer virtuelle Destruktoren haben~~
 - Aber: Viele Basisklassen sehen keine polymorphe Verwendung vor
 - Und: Objekte werden durch dynamische Typinformationen größer
- Nachteil von nicht-virtuellen Destruktoren:
Mögliche direkte Aufrufe des Basisklassendestruktors
 - Gilt nur für sichtbare Destruktoren
 - `private`-Destruktoren verbieten den (notwendigen) Aufruf aus abgeleiteten Klassen
 - Als `protected` deklarierter Destruktor umgeht diese Probleme
- Regel:
Eine Basisklasse sollte entweder einen virtuellen Destruktor oder einen als `protected` deklarierten Destruktor besitzen
 - Virtueller Destruktor bei polymorph zu verwendenden Klassen
 - Sonst `protected` (insbesondere für Mixin-Klassen)

Polymorphie

```
class Expression {
    //...
    virtual int evaluate() const = 0;
    virtual ~Expression(){}
};

class Addition: public Expression {
    //...
    int evaluate() const override
    { return this->left->evaluate() + this->right->evaluate(); }
};

class Constant: public Expression {
    //...
    int evaluate() const { return this->Value; }
};

void printEval(const Expression& expr) { std::cout<<expr.evaluate()<<"\n"; }
//...
printEval(Addition{...});
printEval(Constant{...});
```

- Expression ist Basisklasse und soll polymorph verwendet werden
- Es wird deshalb ein virtueller Destruktor benötigt

Das Non Virtual Interface Pattern (NVI)

- Manchmal sollen in einer virtuellen Methode bestimmte Operationen für alle Implementationen gleich ausgeführt werden
 - Beispiel: Eintrag in eine Logdatei zu Debugzwecken
 - Überprüfung der Argumente auf Gültigkeit
- Virtuelle Methoden können nur komplett überschrieben werden
 - Wiederholung des Codes in allen abgeleiteten Klassen ist fehleranfällig
- Elegantere Lösung: Das Non Virtual Interface Pattern
 - Die virtuelle Methode ist `private` oder `protected`
 - Wird von nicht virtueller `public`-Methode der Basisklasse aufgerufen
 - Basisklasse kann beliebige Operationen vor und nach dem virtuellen Aufruf einfügen
- Leichtes Erstellen von Methoden mit gleicher Basisimplementierung
 - Beispiel: Laden aus einer Datei, ein Overload akzeptiert einen Stream, der andere einen Dateinamen

Beispiel für NVI

```
class ContainerBase {
private:
    virtual bool tryGetItemImpl(size_t id, Item& result) const = 0;
public:
    std::tuple<Item, bool> tryGetItem(size_t id) const {
        Item result;
        bool success = this->tryGetItemImpl(id, result);
        return std::make_tuple(result, success);
    }
    Item getItem(size_t id) const {
        Item result;
        if(!this->tryGetItemImpl(id, result))
            throw std::runtime_error("illegal id detected in ContainerBase::GetItem");
        return result;
    }
    virtual ~ContainerBase(){}
};

class ActualContainer: public ContainerBase {
private:
    bool tryGetItemImpl(size_t id, Item& result) const override {
        //... implementation
    }
};
```

Beispiel für NVI

```
class ContainerBase {  
private:  
    virtual bool tryGetItemImpl(size_t id, Item& result) const = 0;  
public:  
    std::tuple<Item, bool> tryGetItem(size_t id) const {  
        Item result;  
        bool success = this->tryGetItemImpl(id, result);  
        return std::make_tuple(result, success);  
    }  
    Item getItem(size_t id) const {  
        Item result;  
        if(!this->tryGetItemImpl(id, result))  
            throw std::runtime_error("Item not found");  
        return result;  
    }  
    virtual ~ContainerBase(){}  
};  
  
class ActualContainer: public ContainerBase {  
private:  
    bool tryGetItemImpl(size_t id, Item& result) const override {  
        //... implementation  
    }  
};
```

Als **private** deklarierte virtuelle Methoden können in abgeleiteten Klassen überschrieben, aber nicht (direkt) aufgerufen werden.

Optimal für NVI.

Beispiel für NVI

```
class ContainerBase {
private:
    virtual bool tryGetItemImpl(size_t id, Item& result) const = 0;
public:
    std::tuple<Item, bool> tryGetItem(size_t id) const {
        Item result;
        bool success = this->tryGetItemImpl(id, result);
        return std::make_tuple(
    }
    Item getItem(size_t id) const {
        Item result;
        if(!this->tryGetItemImpl(id, result))
            throw std::runtime_error("Item not found");
        return result;
    }
    virtual ~ContainerBase(){}
};

class ActualContainer: public ContainerBase {
private:
    bool tryGetItemImpl(size_t id, Item& result) const override {
        //... implementation
    }
};
```

Als **private** deklarierte virtuelle Methoden können in abgeleiteten Klassen überschrieben, aber nicht (direkt) aufgerufen werden.

Optimal für NVI.

Deklaration des Destruktors als virtuell nicht vergessen!

Slicing

```
class Base {  
    virtual void print()  
    { std::cout<<"Base Print\n"; }  
    virtual ~Base(){}  
};  
class Derived: public Base {  
    void print() override  
    { std::cout<<"Derived Print\n"; }  
};  
void printObj(Base b)  
{ b.print(); }  
  
int main() {  
    Derived d;  
    printObj(d);  
}
```

- Was ist die Ausgabe des Programmes?
 - Antwort: Base Print

Slicing II

- In dem gezeigten Beispiel tritt als Slicing bezeichnetes Verhalten auf
- `printObj` wird eine **Kopie** von `d` übergeben
 - Genauer: Eine Kopie des Base-Teils von `d`
 - `Derived`-Teil wird weggesliced
- Es wird nur der dem Zieltyp entsprechende Teil einer polymorphen Klasse kopiert
 - Bei Kopie der Basisklasse gehen Informationen verloren
 - Für abstrakte Basisklassen nicht möglich, da Erzeugung einer Instanz der Klasse illegal ist

Polymorphe Objekte immer als Reference oder als Pointer (eventuell als Smartpointer) übergeben!

dynamic_cast

- Zum Casten von Pointern auf polymorphe Basisklassen in Pointer auf abgeleitete Klassen
 - `Expression* x;`
`Constant* c = dynamic_cast<Constant*>(x);`
 - Nur, wenn die Klasse polymorph ist, also virtuelle Methoden besitzt
- Überprüft, ob das Objekt auch wirklich Instanz der Zielklasse
 - Rückgabewert ist ein Nullpointer, falls dies nicht gilt
- Auch Cast von Referenz auf Basisklasse
 - `Constant& c = dynamic_cast<Constant&>(*x);`
 - Gleicher Check wie für Pointer, wirft aber `std::bad_cast` im Fehlerfall
- Verhält sich abgesehen von der Überprüfung wie ein `static_cast`

Multiple Vererbung

- In einigen Fällen ist es wünschenswert, von verschiedenen Basisklassen zu erben
 - Insbesondere zur Implementierung verschiedener Interfaces
 - In vielen anderen Sprachen nicht direkt möglich
 - In Java nur für Interfaces erlaubt
- C++ unterstützt echte Mehrfachvererbung, Klassen können von beliebig vielen anderen Klassen erben
 - Kann bei unbedachtem Einsatz aber leicht zu Problemen führen
 - C++ vertraut darauf, dass der Programmierer alles richtig macht

Multiple Vererbung II

```
class foo;
class bar;
class foobar: public foo, public bar
{
    foobar(const foo& f, const bar& b):foo(f), bar(b) {}
};
```

- Basisklassen stehen in kommaseparierter Auflistung
- Konstruktor wird für jede Basisklasse aufgerufen
 - Aufrufreihenfolge wie für andere Member entsprechend der Deklarationsreihenfolge
- Für mehr Details:

http://www.cprogramming.com/tutorial/multiple_inheritance.html

Das Diamantproblem

```
class ancestor;  
class son: public ancestor {};  
class daughter: public ancestor {};  
class grandchild: public son, public daughter {};
```

- Bedeutendes Problem in multipler Vererbung:
Abgeleitete Klasse erbt auf mehreren Wegen von einer einzelnen Basisklasse
 - Das sog. Diamant-Problem (“Inzest”)
- Standardverhalten in C++: Die Basisklasse wird mehrfach beerbt
 - ancestor wird “geklont”
 - Datenmember von ancestor sind in grandchild mehrfach vorhanden
 - Aufruf von Methoden von ancestor aus grandchild aufgrund von Mehrdeutigkeit nicht ohne weiteres möglich
- Häufig nicht gewünschtes Verhalten

Visualisierung des Problems nur in den Präsentationsfolien

Virtuelle Vererbung

```
class ancestor;
class son: public ancestor {};
class daughter: public ancestor {};
class grandchild: public son, public daughter {};
```

■ Lösung: virtuelle Vererbung

- `class son: public virtual ancestor ...;`
- `class daughter: public virtual ancestor ...;`
- grandchild bestimmt, wo die Datenmember liegen
- Stellt sicher, dass sich alle Vererbungspfade eine Instanz von ancestor teilen
- Auch hier einige Fallstricke
- http://www.cprogramming.com/tutorial/virtual_inheritance.html

Multiple (und virtuelle) Vererbung hat viele Fallstricke. Nur wenn notwendig und nach Möglichkeit nur mit einfachen Basisklassen verwenden!

Verwendung von Vererbung und Polymorphie

- Vererbung und Polymorphie sind nützliche Werkzeuge
 - Aber auch nur Werkzeuge
- Häufiger Fehler: übermäßige Verwendung von Vererbung
 - Vererbung (**is-a**) weicht die Kapselung auf
 - Virtuelle Methoden kosten Performance
- Laufzeitpolymorphie häufig nicht notwendig
 - Templates bieten Polymorphie zur Compilezeit
 - Unterscheidet C++ von anderen Sprachen wie Java
 - Laufzeitpolymorphie wird durch Value-Semantik von Objekten verkompliziert
- Abwägen, ob Vererbung notwendig ist, oder ob Komposition reicht
 - Im Zweifelsfall Komposition bevorzugen
- Überprüfen, ob Compilezeitpolymorphie über Templates ausreicht
 - Implementation eigener Templates aber erst später



Allgemeines zu den Aufgaben

Weitere Regeln für die C++-Programmierung

- Für Basisklassen Destruktor als `protected` oder als `virtual` deklarieren
 - Virtuelle Destrukturen verwenden, wenn polymorphe Verwendung gewünscht
 - `protected` verwenden, wenn keine direkte Verwendung der Basisklasse vorgesehen ist
- Überschreiben von virtuellen Funktionen mit `override` markieren
- Keine Exceptions aus Destrukturen werfen

Nichtbeachtung ohne **überzeugende** Begründung kann zu Punktabzug führen!