

Objektorientiertes Programmieren Teil 1

Vererbung, Polymorphie, Gruppenprojekt mit Gosu

Vererbung, Polymorphie, Gruppenprojekt mit Gosu

Dr.-Ing. Jörg Matthes
Dipl.-Inf. Oliver Scherer

5 Objektorientierung

5.1 Private Felder, Methoden und Konstruktoren

5.1.1 Private Felder, Methoden und Konstruktoren

```
struct Foo {  
    int32_t normales_feld;  
private:  
    int32_t privates_feld;  
    // hier auch Methoden oder Konstruktoren  
};  
  
Foo foo;  
cout << foo.normales_feld; // OK  
cout << foo.privates_feld; // Fehler!
```

5.2 Klassen

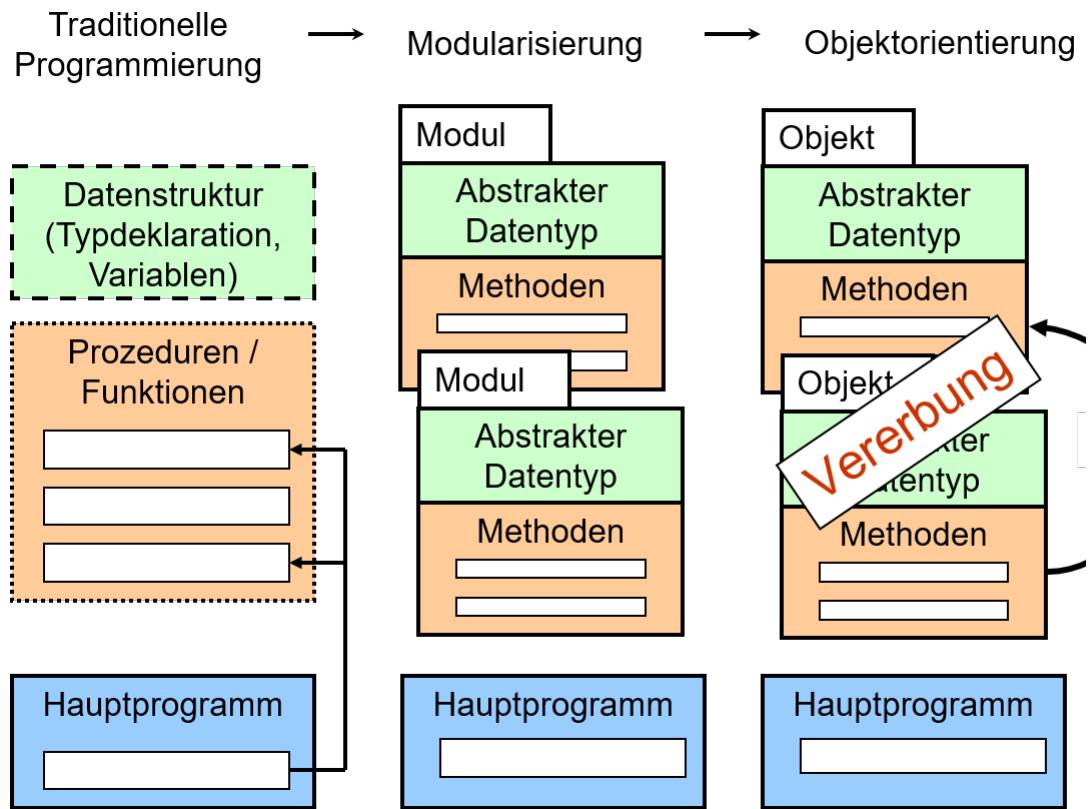
5.2.1 Klassen

Klassen sind `structs`, bei denen alle Felder standardmäßig privat sind. Es muss das `public` keyword verwendet werden, um Konstruktoren und Methoden öffentlich zu machen.

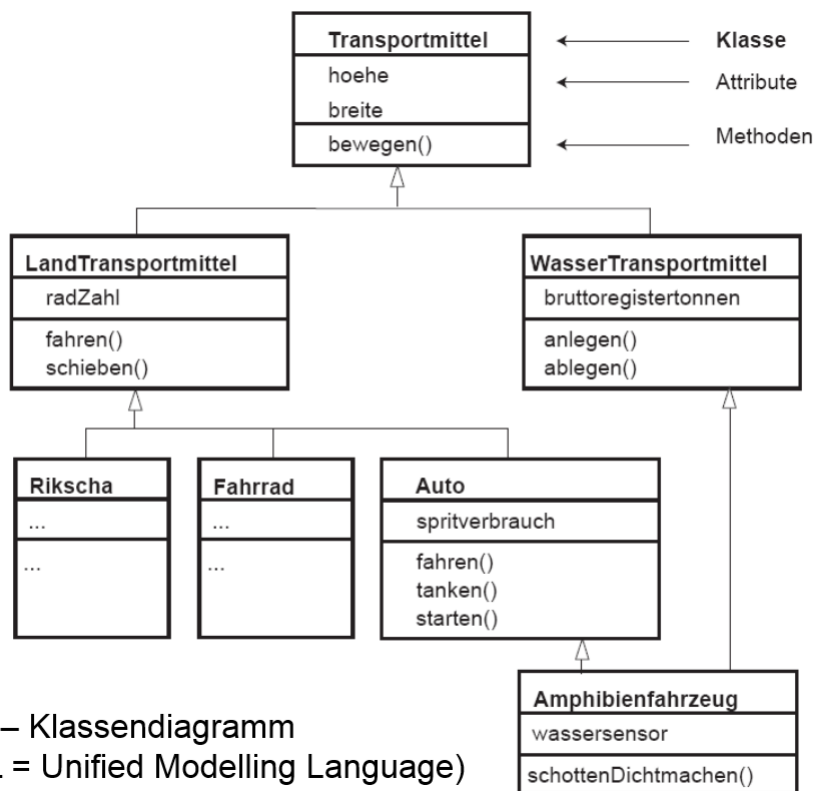
```
class Foo {  
    int32_t privates_feld;  
public:  
    Foo(int32_t wert);  
};  
  
Foo::Foo(int32_t wert) {  
    this->privates_feld = wert;  
}
```

5.3 Vererbung

5.3.1 Grundlagen Vererbung



5.3.2 Vererbung dargestellt im Klassendiagramm



5.3.3 Vererbung von Attributen

Kindklasse `Mitarbeiter` ist eine spezielle Form von Elternklasse `Person` und erbt alle Attribute.

Kindklasse `Mitarbeiter` kann durch neue Attribute erweitert werden.

```

class Person {
public:
    string name;
};

class Mitarbeiter : public Person { //Vererbung
public:
    // Attribut 'name' wurde von Person geerbt
    int32_t personalnummer;
};

int main() {
    Person p1;
    p1.name = "Müller";

    Mitarbeiter m1;
    m1.name = "Mayer";
    m1.personalnummer = 12345;
}
  
```

5.3.4 Vererbung von Attributen - Achtung!

Zuweisung ist erfolgreich, "vergisst" aber alle Felder die `Mitarbeiter`-spezifisch sind.

```
Person p2 = m1;
```

5.3.5 Vererbung ist transitiv

Kindklasse `Leitender_Mitarbeiter` erbt alle Attribute von `Person` und `Mitarbeiter` und kann durch neue Attribute erweitert werden.

```
class Leitender_Mitarbeiter : public Mitarbeiter {
public:
    // Attribut name wurde über Mitarbeiter von Person geerbt
    // Attribut personalnummer wurde von Mitarbeiter geerbt
    int32_t anzahl_untergebene;
};

int main() {
    Leitender_Mitarbeiter lml;
    lml.name = "Schulze";
    lml.personalnummer = 98765;
    lml.anzahl_untergebene = 17;
}
```

5.3.6 Vererbung von Methoden

Kindklasse `Mitarbeiter` ist eine spezielle Form von Elternklasse `Person` und erbt alle Methoden.

Kindklasse `Mitarbeiter` kann durch neue Methoden erweitert werden.

```

class Person {
    string name; // private
public:
    void set_name(const string& n) {
        this->name = n;
    }
    string get_name() {
        return this->name;
    }
};

class Mitarbeiter : public Person { // Vererbung
    int32_t personalnummer; // private
public:
    void set_personalnummer(const int32_t& pn) {
        this->personalnummer = pn;
    }
    int32_t get_personalnummer()
        return this->personalnummer;
}
};

```

5.3.7 Vererbung von Methoden

```

int main() {
    Mitarbeiter m1;
    m1.set_name("Mayer");
    m1.set_personalnummer(12345);
    cout << m1.get_name() << ", " << m1.get_personalnummer() << endl;
}

```

- Beim Aufruf `m1.get_name()` wird geprüft, ob die Klasse `Mitarbeiter` eine eigene Implementierung für diese Methode besitzt.
- Wenn nicht (wie hier) wird in der Elternklasse `Person` geschaut und die `get_name()`-Methode von `Person` ausgeführt.
- Gäbe es die Methode in `Person` auch nicht, würde in die Elternklasse (wenn vorhanden) von `Person` geschaut usw., ansonsten Compiler-Fehler

Die Vererbung von Methoden ist ebenfalls transitiv!

5.3.8 Vererbung und Konstruktoren

Jedes Objekt der Kindklasse `Mitarbeiter` besitzt ein anonymes Objekt der Elternklasse `Person`.

Beim Erzeugen eines Objekts der Kindklasse wird also immer ein Konstruktor der Elternklasse aufgerufen

Durch

```
Mitarbeiter m1;
```

wird der vom System generierte Standardkonstruktor für `Mitarbeiter` aufgerufen.

Dieser ruft automatisch den vom System generierten Standardkonstruktor von `Person` auf.

5.3.9 Vererbung und Konstruktoren

Besitzen die Klassen einen allgemeinen Konstruktor (der vom System generierte Standardkonstruktor existiert dann nicht mehr), dann muss auch für das Erzeugen des anonymen `Person`-Objekts dessen allgemeiner Konstruktor aufgerufen werden.

```
class Person {
    string name; //private
public:
    Person(string n);
};

class Mitarbeiter : public Person { //Vererbung
    int32_t personalnummer; //private
public:
    Mitarbeiter(string n, int32_t pn);
};

Person::Person(string n)
    : name(n) { }

Mitarbeiter::Mitarbeiter(string n, int32_t pn)
    : Person(n), personalnummer(pn) {}

int main() {
    Mitarbeiter m1("Meyer", 12345);
}
```

5.3.10 Vererbung und Zeiger/Referenzen

Referenzen und Zeiger der Elternklasse können auch für Objekte der Kindklasse genutzt werden.

```
Mitarbeiter m1("Meyer", 12345);
cout << m1.get_name(); // ruft Person::get_name auf

// Referenz vom Typ Person auf Objekt vom Typ Mitarbeiter
Person& p1 = m1;
cout << p1.get_name();

// Zeiger vom Typ Zeiger-auf-Person auf Objekt vom Typ Mitarbeiter
Person* p2_ptr = &m1;
cout << p2_ptr->get_name();
```

5.3.11 Vererbung und Zugriffsschutz

- `public`-Attribute und -Methoden unterliegen keiner Zugriffsbeschränkung
- `protected`-Attribute und -Methoden sind in der eigenen und in den abgeleiteten Kindklassen zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse
- `private`-Attribute und -Methoden sind ausschließlich innerhalb der Klasse zugreifbar. (Ausnahme: sog. `friend`-Klassen)

5.4 Polymorphismus

5.4.1 Überschreiben geerbter Methoden

In einer Kindklasse können geerbte Methoden überschrieben werden:

- es wird eine Methode mit identischem Namen und identischer Übergabeschnittstelle neu definiert
- ACHTUNG! Wenn sich die Übergabeschnittstelle unterscheidet, findet kein Überschreiben statt, sondern der Klasse wird eine neue Methode hinzugefügt.
- die Methode sollte die gleiche Bedeutung wie in der Elternklasse haben
- die Implementierung der überschriebenen Methode ist spezifisch für die Klasse
- Methoden mit gleichem Namen und gleicher Schnittstelle aber unterschiedlicher Implementierung nennt man "polymorph" (vielgestaltig).

5.4.2 Überschreiben geerbter Methoden

Die von `Geom_Figur` geerbte Methode `get_flaeche` wird in den Kindklassen `Quadrat` und `Kreis` überschrieben:

```

class Geom_Figur {
public:
    float get_flaeche() {return 0;}
}

class Quadrat : public Geom_Figur {
    float breite;
public:
    void set_breite(float b) {this->breite = b;}
    float get_flaeche() {return breite*breite;}
}

class Kreis : public Geom_Figur {
    float radius;
public:
    void set_radius(float r) {this->radius = b;}
    float get_flaeche() {return 3.1415 * radius * radius;}
}

```

5.4.3 Überschreiben geerbter Methoden

Die von Geom_Figur geerbte Methode get_flaeche wird in den Kindklassen Quadrat und Kreis überschrieben.

Vorteil: Der Aufruf zur Flächenberechnung ist immer gleich, egal, um welche geometrische Figur es sich handelt.

```

Quadrat q1;
q1.set_breite(4);
cout << "Fläche:" << q1.get_flaeche() << endl; //16

Kreis k1;
q1.set_radius(5);
cout << "Fläche:" << k1.get_flaeche() << endl; //78.54

```

5.4.4 Virtuelle Methoden

Wenn wir nun die Flächeninhalte einer ganzen Liste von geometrischen Figuren ausgeben lassen wollen, dann müsste das doch eigentlich so funktionieren:

```

// Liste von Zeigern auf geometrische Objekte
vector<Geom_Figur*> Liste = {&q1, &k1};

for (size_t i = 0; i < Liste.size(); i++)
{
    cout << "Fläche:" << Liste.at(i)->get_flaeche() << endl;
}

```

Liefert 0 0 anstelle 16 und 78.54

Problem: Zeiger in der Liste ist vom Typ `Geom_Figur*`. Deshalb wird `get_flaeche` von `Geom_Figur` aufgerufen und nicht von `Quadrat` bzw. `Kreis`.


5.4.5 Virtuelle Methoden

Damit nicht der Zeigertyp, sondern der Objekttyp entscheidet, welche Variante der polymorphen Methode `get_flaeche` aufgerufen wird, muss die Methode als virtuelle Methode deklariert werden:

```
class Geom_Figur {
public:
    virtual float get_flaeche() {return 0;}
}

class Quadrat : public Geom_Figur {
    float breite;
public:
    void set_breite(float b) {this->breite = b;}
    virtual float get_flaeche() override {return breite*breite;}
}

class Kreis : public Geom_Figur {
    float radius;
public:
    void set_radius(float r) {this->radius = b;}
    virtual float get_flaeche() override {return 3.1415 * radius * r;}
}
```



5.4.6 Virtuelle Methoden

- Virtuelle Methoden dienen zum Überschreiben bei gleicher Signatur und gleichem Rückgabetyt.
- Der Aufruf einer nicht-virtuellen Methode hängt *vom Typ des Zeigers* ab, über den die Methode aufgerufen wird, während der Aufruf einer virtuellen Methode *vom Typ des Objekts* abhängt, auf das der Zeiger verweist.
- Eine als virtuell deklarierte Methode definiert eine *Schnittstelle* für alle abgeleiteten Klassen, auch wenn diese zum Zeitpunkt der Festlegung der Elterklasse (Basisklasse) noch unbekannt sind.

5.4.7 Virtueller Destruktor

```

class Geom_Figur {
public:
    Geom_Figur() { cout << "Konstruktor Geom_Figur" << endl; }
    ~Geom_Figur() { cout << "Destruktor Geom_Figur" << endl; }
};

class Kreis : public Geom_Figur {
    float radius;
public:
    Kreis() { cout << "Konstruktor Kreis" << endl; }
    ~Kreis() { cout << "Destruktor Kreis" << endl; }
};

int main() {
    {
        // Aufruf des Konstruktors für Kreis und damit auch für Geom_Fig
        unique_ptr <Geom_Figur> gf_ptr = make_unique <Kreis> ();
    }
    // Nur Aufruf des Destruktors für Geom_Figur,
    // da Zeiger vom Typ unique_ptr< Geom_Figur >
}

```

Bei Aufruf `malloc` im Konstruktor von `Kreis` und passendem `free` im Destruktor würde ein Memory-Leak entstehen, da `free` nicht aufgerufen wird.

5.4.8 Virtueller Destruktor

Deshalb: Destruktoren immer als virtuelle Methoden deklarieren!

```

class Geom_Figur {
public:
    Geom_Figur() { cout << "Konstruktor Geom_Figur" << endl; }
    virtual ~Geom_Figur() { cout << "Destruktor Geom_Figur" << endl; }
};

class Kreis : public Geom_Figur {
    float radius;
public:
    Kreis() { cout << "Konstruktor Kreis" << endl; }
    virtual ~Kreis() override { cout << "Destruktor Kreis" << endl; }
};

```

Damit wird auch der Destruktor von `Kreis` korrekt aufgerufen.

5.5 Abstrakte Klassen

5.5.1 Abstrakte Klassen

- Bei bestimmten Elternklassen macht es keinen Sinn, dass Objekte von ihnen erzeugt werden.

- Z.B. gibt es entweder Objekte der Klassen `Quadrat` bzw. `Kreis`, aber nicht Objekte der Klasse `Geom_Figur`.
- Solche Klassen dienen nur der Abstraktion und werden als *abstrakte Klassen* bezeichnet.

In C++ sind Klassen abstrakt, wenn sie mindestens eine *rein virtuelle Methode* besitzen:

```
class Geom_Figur {
public:
    virtual float get_flaeche() = 0;
}
```

5.5.2 Abstrakte Klassen

```
int main() {
    Kreis k;
    Geom_Figur gf; // Compiler-Fehler, da Geom_Figur abstrakt

    Geom_Figur* gf_ptr = &k; // funktioniert weiterhin

    unique_ptr<Kreis> k_uptr = make_unique<Kreis>();
    unique_ptr<Geom_Figur> gf_uptr = move(k_uptr);

    cout << gf_uptr -> get_flaeche();
}
```

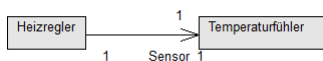
5.6 Assoziation zwischen Klassen

5.6.1 Reine Assoziation

Neben der Vererbung können Klassen auch durch Assoziationsbeziehungen miteinander verknüpft sein.

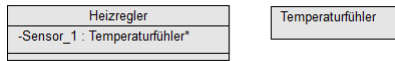
Die *reine Assoziation* (Benutzt-/Kennt-Beziehung, Lose Kopplung) zwischen zwei Klassen bedeutet, dass ein Objekt der assoziierenden Klasse (Heizregler) ein Objekt der assoziierten Klasse (Temperaturfühler) benutzt/kennt.

In der UML wird dies durch einen einfachen Pfeil mit Angabe der Multiplizität (z.B. 1,1) angegeben. Der Pfeil kennzeichnet dabei eine gerichtete Beziehung (Heizregler kennt Temperaturfühler, aber nicht umgekehrt.)



5.6.2 Reine Assoziation

In der Umsetzung bedeutet das, dass das assoziierende Objekt einen `shared_ptr` für das assoziierte Objekt besitzt:

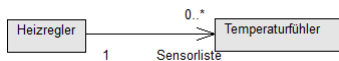


```
class Temperaturfühler {
...
}

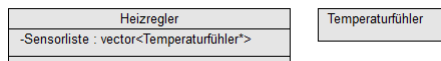
class Heizregler {
    shared_ptr<Temperaturfühler> Sensor_1;
}
```

5.6.3 Reine Assoziation

Bei einer Multiplizität 1,n benutzt/kennt 1 Objekt der assoziierenden Klasse n Objekte der assoziierten Klasse:



Umsetzung:



```
class Heizregler {
    vector<shared_ptr<Temperaturfühler>> Sensorliste;
}
```

5.6.4 Aggregation

Die *Aggregation* ist eine spezielle Form der Assoziation. Durch sie soll eine engere Verbindung zwischen den beteiligten Objekten ausgedrückt werden:



In der Umsetzung unterscheidet sie sich jedoch nicht von der Assoziation.

```
class Heizregler {
    vector<shared_ptr<Mitarbeiter>> Belegschaft;
}
```

5.6.5 Komposition

Die *Komposition* ist eine spezielle Form der Aggregation und damit auch der Assoziation. Durch sie soll eine sehr enge Verbindung mit Existenzabhängigkeit zwischen den beteiligten Objekten ausgedrückt werden:



Bei der Komposition existieren die kompositionierten Objekte (Kapitel) nur solange, solange auch das kompositionierende Objekt (Buch) existiert.

(Wenn das Buch zerstört wird, existieren auch die Kapitel nicht mehr.)

5.6.6 Komposition

Bei der Umsetzung besitzt das kompositionierende Objekt (Buch) nun nicht mehr nur Zeiger auf, sondern die kompositionierten Objekte (Kapitel) selbst:



```
class Kapitel {
    ...
}

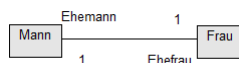
class Buch {
    vector<Kapitel> Inhalt;
}
```

Wenn der Typ des kompositionierten Objektes eine Basisklasse ist, so kann mit einem `unique_ptr` der Besitz umgesetzt werden.

5.6.7 Ungerichtete Assoziation

Bei einer ungerichteten Assoziation kennt nicht nur ein Objekt das andere, sondern die Objekte kennen sich gegenseitig (keine Pfeile im Klassendiagramm).

(Eine gegenseitige Aggregation also eine engere Bindung ist in UML nicht möglich.)



Umsetzung:



```
class Frau; // nur Deklaration wegen gegenseitiger Sichtbarkeit

class Mann {
    shared_ptr<Frau> Ehefrau;
}

class Frau {
    shared_ptr<Mann> Ehemann;
}
```

6 Grafikengine

6.0.1 Git installieren

(auf DHBW PCs normalerweise nicht nötig)

- TortoiseGit
- Git for Windows

6.0.2 Neues Projekt

1. Neuen Ordner erstellen
2. Rechtsklick -> Git clone
3. "https://github.com/oli-obk/dhbw-objektorientierung.git" in Url Feld eingeben
4. Ok klicken
5. Warten
6. Beispielprojekt.sln (Microsoft Visual Studio Solution) öffnen

6.0.3 Neues Projekt

Notwendige Includes:

```
#include <Gosu/Gosu.hpp>
#include <Gosu/AutoLink.hpp>
```

6.0.4 Neues Projekt

```

class GameWindow : public Gosu::Window {
public:
    GameWindow()
        : Window(640, 480)
    {
        set_caption("Gosu Tutorial Game");
    }

    void update() override {
        // ...
    }

    void draw() override {
        // ...
    }
};

int main() {
    GameWindow window;
    window.show(); //blockierender Aufruf bis window geschlossen wird
}

```

6.0.5 Window Konstruktor

```

GameWindow()
    : Window(640, 480)
{
    set_caption("Gosu Tutorial Game");
}

```

Angabe von Höhe und Breite im Konstruktor.

Aufrufen von `set_caption` erlaubt ersetzen des Titeltexes des Fensters

6.0.6 Formen zeichnen

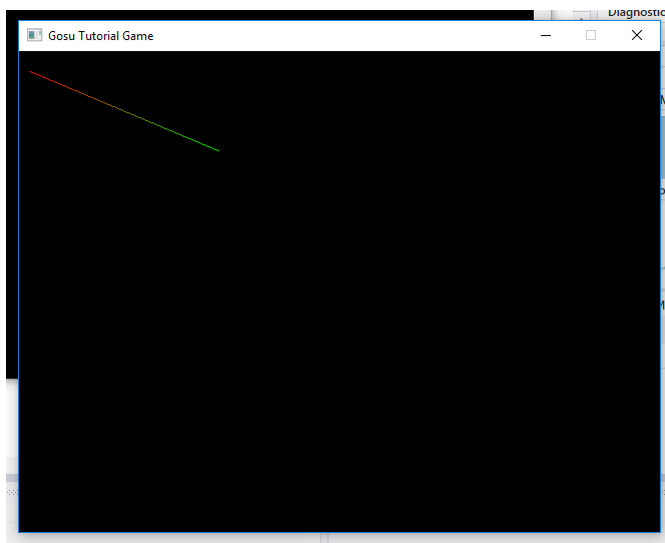
In der `draw` Methode kann mittels des `Graphics`-Objektes gezeichnet werden.

Zugriff auf das `Graphics`-Objekt erhält man über die `graphics()`-Methode.

```

graphics().draw_line(
    10, 20, Gosu::Color::RED,
    200, 100, Gosu::Color::GREEN,
    0.0
);

```



6.0.7 Dokumentation

Alle Funktionen und Typen sind als Webseite dokumentiert

https://www.libgosu.org/cpp/namespace_gosu.html

6.0.7.1 Aufgabe

- Aufsuchen der `draw_triangle` Funktion
- Ein buntes Dreieck zeichnen mit der Funktion zeichnen

6.0.7.2 ERRINNERUNG

Funktionsdeklaration in der Doku: `int foo(int bar) const` Funktionsaufruf in eurem Code: `int x = objekt.foo(y);`

Wer einen Funktionsaufruf der Form `int foo(int bar = y) const;` abliefert, bringt nächstes Vorlesung einen Kuchen mit.

6.0.8 Animieren

Die `draw` Methode kann auf Variablen des eigenen `GameWindow` Objektes zugreifen. Mittels der `update` Methode können diese regelmäßig verändert werden.

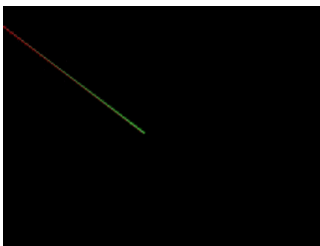

```

int x = 0;

void update() override {
    x = (x + 1) % 300;
}

void draw() override {
    Gosu::Graphics::draw_line(
        x, 20, Gosu::Color::RED,
        200, 100, Gosu::Color::GREEN,
        0.0
    );
}

```



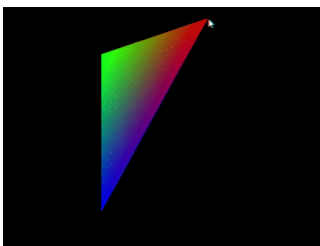
6.0.9 Benutzereingaben

Benutzereingaben können via `input()` abgefragt werden.

6.0.9.1 Aufgabe

- Finden von Funktionen zum Abfragen der Mausposition
- Auslesen der Mausposition in `update`
- Zeichnen eines Dreieckes bei dem eine Ecke an der Mausposition ist

6.0.10 Dreieck



6.0.11 Dreieck

```

double x = 0;
double y = 0;

void update() override
{
    x = input().mouse_x();
    y = input().mouse_y();
}

void draw() override
{
    Gosu::Graphics::draw_triangle(
        x, y, Gosu::Color::RED,
        200, 100, Gosu::Color::GREEN,
        200, 400, Gosu::Color::BLUE,
        0.0
    );
}

```

6.0.12 Bilder

Gosu macht das Bilder laden sehr einfach. Es gibt einen `Image` Typ, dessen Konstruktor einen Dateinamen als einziges Argument nimmt.

Eine Variable vom Typ `Image` kann mit den Funktionen `draw` und `draw_rot` gezeichnet werden.

Damit das Bild nicht 60 Mal pro Sekunde geladen wird, muss das Bild ein Feld der `GameWindow` Klasse sein.

Die zu ladenden Bilder müssen sich im Projektordner befinden (nicht im Solutionordner!)

6.0.13 Bilder laden

```

Gosu::Image bild;
GameWindow()
    : Window(640, 480)
    , bild("rakete.png")
{
    set_caption("Gosu Tutorial Game mit Git");
}

void draw() override
{
    bild.draw_rot(x, y, 0.0,
        0.0, // Rotationswinkel in Grad
        0.5, 0.5 // Position der "Mitte" relativ zu x, y
    );
}

```

6.0.14 Bilder - Aufgabe

Diverse Werte für den 5. und 6. Parameter auswählen und die Effekte beobachten.

- 0.0, 0.0 (Linke obere Ecke vom Bild ist an x, y)
- 1.0, 1.0 (Rechte untere Ecke vom Bild ist an x, y)
- Werte kleiner 0 oder größer 1

6.0.15 Bilder drehen - Aufgabe

Tastatur- und Mauseingaben können mit der `down` funktion des `input()` Objektes abgefragt werden.

Als Argument wird ein Wert des Enums `ButtonName` erwartet. Finden Sie die Werte für die rechte und die linke Maustaste.

Drehen Sie ihr Bild nach rechts, wenn die rechte Maustaste gedrückt ist, und nach links, wenn die linke Maustaste gedrückt ist.

Es sollte möglich sein, durch gedrückt halten einer Maustaste das Bild komplett im Kreis zu drehen.

Errinnerung: Nur in der `update` Funktion dürfen Felder von `GameWindow` verändert werden.

6.0.16 Bilder drehen - Lösung

```
void draw() override
{
    bild.draw_rot(x, y, 0.0,
        rot, // Rotationswinkel in Grad
        0.5, 0.5 // Position der "Mitte"
    );
}

double rot = 0.0;
double x = 0;
double y = 0;

void update() override
{
    x = input().mouse_x();
    y = input().mouse_y();
    if (input().down(Gosu::MS_LEFT)) {
        rot += 10;
    }
    else if (input().down(Gosu::MS_RIGHT)) {
        rot -= 10;
    }
}
```

6.1 Versionsverwaltung

6.1.1 Was ist Versionsverwaltung?

- Backup
 - Kopie auf einem oder mehreren Servern
- Archiv
 - Jede Änderung wird aufgezeichnet und kann in Zukunft angezeigt werden
 - "Zeile X wurde am Y.Z.A von B zusammen mit Zeilen D, E, F erstellt/verändert"
- Teamwork
 - Zusammen an einem Dokument arbeiten - ohne Chaos
 - Experimentieren ohne dass die Anderen gestört werden
 - Zusammenführen von eigenem experimentellem Code und gemeinsamen Dokumenten, wenn Experiment erfolgreich beendet
- Automatisches Testen bei jeder Änderung
 - Email an alle wenn einer den Code kaputt macht

6.1.2 Austausch und Backup Server

Auf <https://github.com> registrieren und einloggen

Built for developers

GitHub is a development platform inspired by the way you work. From **open source** to **business**, you can host and review code, manage projects, and build software alongside millions of other developers.

Username
Pick a username

Email
you@example.com

Password
Create a password
Use at least one letter, one numeral, and seven characters.

Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We'll occasionally send you account related emails.

6.1.3 Neues Git Repository anlegen

Geht auf


<https://github.com/oli-obk/dhbw-objektorientierung>

Und klickt auf `Fork` rechts oben in der Ecke

Nach erfolgreichem klonen, rechts auf "Clone or download" klicken und http Adresse kopieren.

6.1.4 Projekt zu git hinzufügen

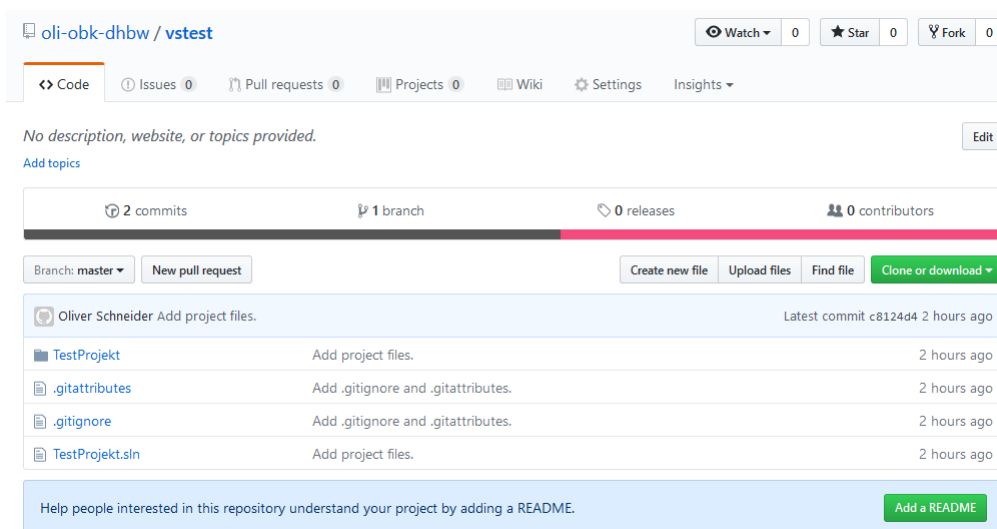
Im Visual Studio kann nun das eigene Projekt mit Github verbunden werden.

- Ansicht
- Team Explorer
- Einstellungen 
- Repository Einstellungen
- Remotes
 - Hinzufügen
 - "Name" Feld muss "origin" sein
 - "Fetch" Feld ist http-Adresse von Github

6.1.5 Lokalen Zustand auf Server laden

- Haussymbol im Team-Explorer klicken
- Sync
- Outgoing Commits
 - Veröffentlichen
 - Benutzername + Passwort angeben

6.1.6 Zustand auf Server



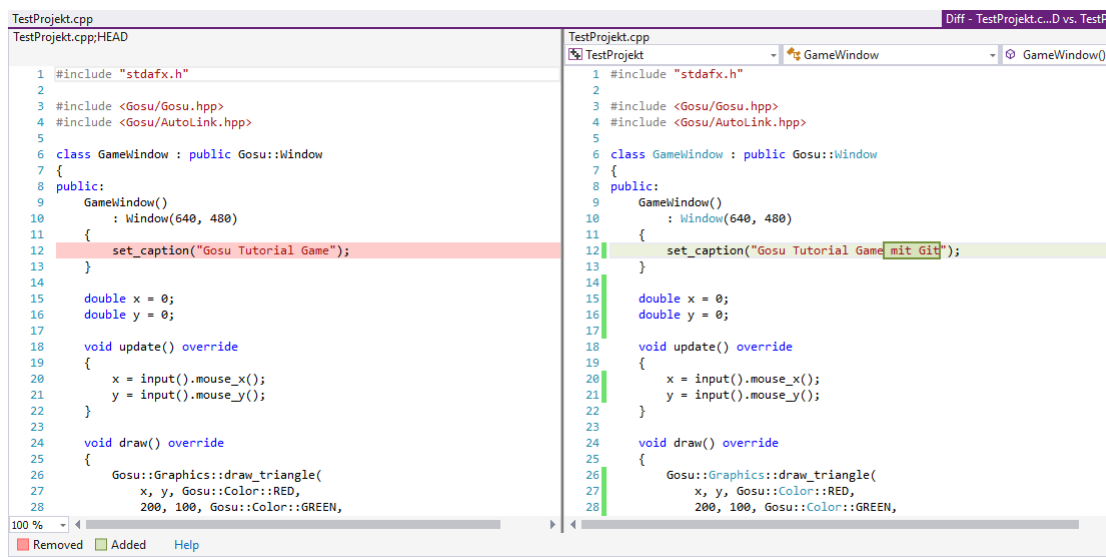
6.1.7 Änderungen

Zum Test eine Änderung am Programm durchführen. Zum Beispiel den Titel (set_caption) des Fensters ändern.

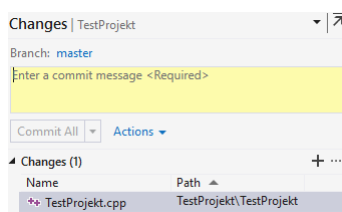
Im Team-Explorer kann nun unter "Änderungen" eingesehen werden, was genau sich verändert hat.

Mit Doppelklick auf eine Datei im Team-Explorer erscheint ein Vergleichsfenster.

6.1.8 Vergleichen



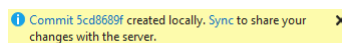
6.1.9 Änderungen einchecken



Kurze Beschreibung der Änderungen eingeben, und "Alles einchecken" klicken

6.1.10 Änderungen hochladen

Visual Studio bietet direkt an, mit dem Server zu synchronisieren.



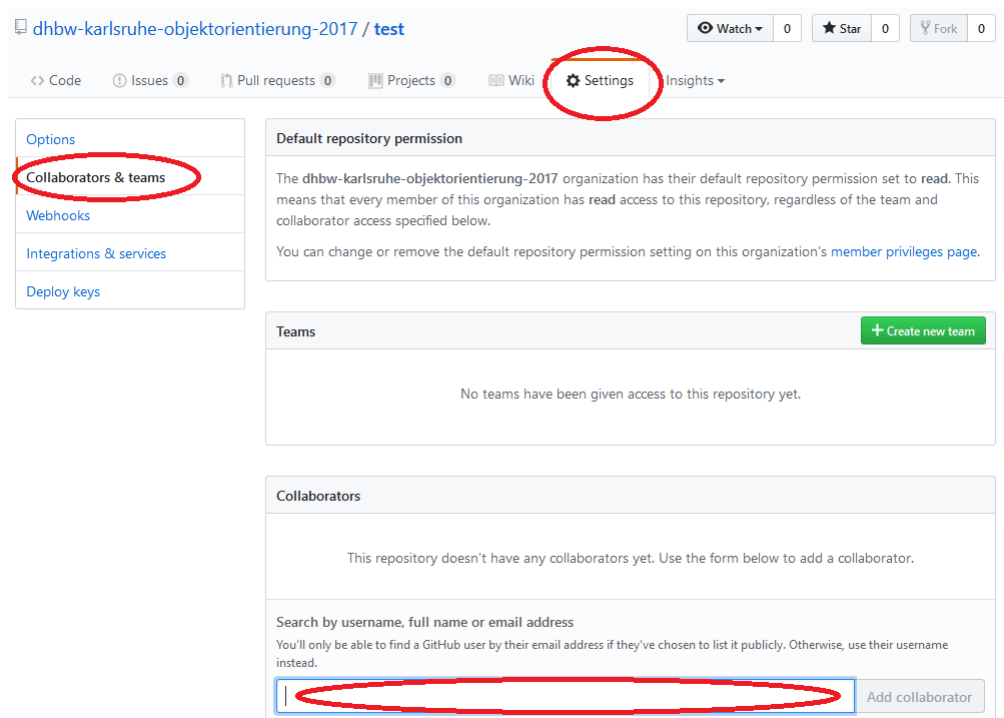
Auf "Sync" klicken

Im neuen Fenster auf "Push" klicken.

6.1.11 Änderungen auf Server einsehen

Auf der github Webseite des Repositorys ist nun unter "Commits" der neue Commit zu sehen.

6.1.12 Allen Gruppenmitgliedern Schreibzugriff geben



6.1.13 Einrichten

Ein Gruppenmitglied lädt sein Projekt in das neue Repository wie bereits vorgestellt.

6.1.13.1 Alle anderen führen die nun folgenden Anweisungen durch.

- Visual Studio ohne Projekt öffnen
- Team -> Verbindungen Verwalten
- Rechts unten auf "Clone" klicken
- Repositoryadresse und lokalen Speicherort angeben
 - Speicherort darf kein existierender Ordner sein
- "Clone" klicken