5.2 Einführung und Klassen/Zugriffsschutz für Komponenten

5.2.1 Zugriffsschutz gesucht

Ab jetzt können wir also Daten und Funktionen unter dem Dach einer Klassen zusammenfassen. Mit den Funktionen greife ich dann auf meine Daten sachgemäß zu. So sorgt z. B. die Funktion `printpkw()` dafür, dass Kennzeichen und Kilometerstand richtig aus dem Objekt gelesen und auf den Bildschirm ausgegeben werden:

```
1    printf("%s␣%d\n", a.kennz, a.kmstand-5000);
```

Genauso kann man jetzt z. B. mit einer Funktion `setze_kilometerstand()` dafür sorgen, dass der Kilometerstand richtig in das Objekt geschrieben wird. Negative Kilometerstände sollen nicht erlaubt sein:

```
1    setze_kilometerstand(int neuerstand)
2    {
3        if(neuerstand>0)
4        {
5            kmstand=neuerstand;
6        }
7    }
```

Das sieht einfach und ist es auch. Was ist aber, wenn ein vorwitziger Anwender an der Funktion vorbei direkt mit den Attributen arbeitet?

```
1    struct pkwtyp a={"BI-XY␣123", 0};
2    a.kmstand=-42000; // so umgehe ich setze_kilometerstand()...
3    printf("%s␣%d\n", a.kennz, a.kmstand-5000); // und so umgehe ich
4                                                // printpkw()...
```

Deshalb brauchen wir einen Schutz, der den Zugriff auf bestimmte Teile von `struct pkwtyp` verhindert: An `kennz` und `kmstand` soll man von außen nicht mehr herankommen.

Andere Teile von `struct pkwtyp` sollen zugänglich bleiben; auf die Methoden `printpkw()` und `setze_kilometerstand()` soll man weiter zugreifen können.

Diesen Schutz nennt man *Zugriffsschutz* oder auch *Kapselung*.

5.2.2 Zugriffsschutz in C++

Mit den Labels `public:` und `private:` teilen wir die Klasse in Abschnitte:

`public:` Alles, was hinter `public:` steht, ist wie bisher überall sichtbar

`private:` Alles, was hinter `private:` steht, ist nur innerhalb der Klasse sichtbar

`public:` und `private:` sind sogenannte *Zugriffsspezifizierer*.

Im Beispiel `struct pkwtyp` sieht das so aus (`src/pkw2defekt.cpp`):

```
1  #include <stdio.h>
2  #include <string.h>
3  struct pkwtyp
4  {
5  private:
6      char kennz[20];
7      int kmstand;
8  public:
9      void printpkw(void)
10     {
```

```
11          printf("%s, %d\n", kennz, kmstand);
12      }
13  };
14  /*******************************************/
15  int main(void)
16  {
17      int x;
18      struct pkwtyp fw;
19      strcpy(fw.kennz,"BI-XY123");
20      fw.kmstand=20000;
21      fw.printpkw();
22      return 0;
23  }
```

Das Kompilieren zeigt jedoch ein Problem:

Terminal

```
schueler@debian964:~$ c++ src/pkw2defekt.cpp
pkw2defekt.cpp: In function 'int main()':
pkw2defekt.cpp:6: error: 'char pkwtyp::kennz [20]' is private
pkw2defekt.cpp:19: error: within this context
pkw2defekt.cpp:7: error: 'int pkwtyp::kmstand' is private
pkw2defekt.cpp:20: error: within this context
```

Damit, dass man auf die Attribute nicht mehr zugreifen kann, ist keine Initialisierung (Zeile 19) und keine Zuweisung (Zeile 20) mehr möglich. Selbst ein Lesezugriff auf ein Attribut ist ausgeschlossen.

Deshalb müssen wir jetzt die benötigten Methoden einrichten.

5.2.3 Typische Zugriffsmethoden

Nun ist genau das passiert, was wir wollten: Wir kommen nicht mehr direkt an die privaten Attribute (=Variablen) einer Klasse heran, sondern der Zugriff ist nur noch über die passenden Methoden (=Funktionen) möglich.

Aber welche Zugriffsmethoden baut man nun sinnvollerweise in eine solche Klasse ein? Am Beispiel der Klasse `struct pkwtyp` soll das hier gezeigt werden.

5.2.3.1 Niedrige Ebene: get- und set-Methoden Zunächst bietet es sich an, für jedes Attribut eine Methode zum Lesen zu erstellen. Das sind die so genannten get-Methoden:

```
1   struct pkwtyp
2   {
3   private:
4       char kennz[20];
5       int kmstand;
6   public:
7       const char* get_kennz(void)
8       {
9           return kennz;
10      }
11      int get_kmstand(void)
12      {
13          return kmstand;
14      }
15  };
```

Damit sieht eine Ausgabe des Kilometerstands auf den Bildschirm in `main()` so aus (oben alte Version, unten neue):

```
1    printf("Kilometerst.: %d\n", fw.kmstand);   // funkt. nicht wg. private
2    printf("Kilometerst.: %d\n", fw.get_kmstand()); // so geht's
```

Ebenso könnte man für jedes Attribut eine Methode zum Schreiben erstellen. Das sind die set-Methoden:

```
1  struct pkwtyp
2  {
3  private:
4     char kennz[20];
5     int kmstand;
6  public:
7     void set_kennz(const char *k)
8     {
9        kennz[0]='\0';
10       strncat(kennz, k, 19);
11    }
12    void set_kmstand(int neuer_kmstand)
13    {
14       kmstand=neuer_kmstand;
15    }
16 };
```

Damit sieht das Einlesen des Kilometerstands von der Tastatur jetzt so aus (erste Zeile die alte Version, unten neue):

```
1    scanf("%d", &fw.kmstand); // funktioniert nicht wg. private
2    int kilo;                 // so geht's Teil 1
3    scanf("%d", &kilo);       // so geht's Teil 2
4    fw.set_kmstand(kilo);     // so geht's Teil 3
```

Hier zeigt sich ein Vorteil der Kapselung: Die set-Methode kann ihre Parameter prüfen und dafür sorgen, dass kein Unsinn in die Attribute geschrieben wird. Im Beispiel sollte z. B. der Kilometerstand positiv sein:

```
1    void set_kmstand(int neuer_kmstand)
2    {
3       if(neuer_kmstand>0)
4          kmstand=neuer_kmstand;
5    }
```

Die Methode kann noch verbessert werden: Wenn man weiß, was ein Kilometerstand in der Praxis bedeutet, kann man dafür sorgen, dass er wie in der Praxis nur erhöht werden kann:

```
1    void set_kmstand(int neuer_kmstand)
2    {
3       if(neuer_kmstand>kmstand)
4          kmstand=neuer_kmstand;
5    }
```

5.2.3.2 Hohe Ebene: Objektzustand ändern und analysieren Damit kommen wir zu den Methoden, die bestimmte Attribute oder auch das ganze Objekt ansehen und nur bestimmte Änderungen des Objektzustands, also der Attributinhalte, zulassen[1].

Ein Beispiel ist die init-Methode, mit der ein Objekt auf einen definierten Anfangszustand gesetzt wird:

```
1  void init(const char *anfang_kennz, int anfang_kmstand)
2  {
3      kennz[0]='\0';
4      strncat(kennz, anfang_kennz, 19);
5      if(anfang_kmstand>0)
6          kmstand=anfang_kmstand;
7      else
8          kmstand=0;
9  }
```

Genauso kann man mit einer Methode bestimmte Attribute oder das ganze Objekt ansehen und bestimmte Sachverhalte analysieren: Ist das Objekt gültig? Hat es die Eigenschaft XYZ?

Diese Methoden heißen häufig is-Methoden. Ein Beispiel ist die Methode `is_ok()`, die man sinnvollerweise einmal in jede Klasse einbauen sollte. Sie findet heraus, ob die Attribute jeweils sinnvolle und zueinander passende Inhalte haben und gibt dann den Wert `falls` (0) oder `true` (1) heraus:

```
1  bool is_ok(void)
2  {
3      if(strlen(kennz)<4)
4          return false;
5      if(kmstand<0)
6          return false;
7      return true;
8  }
```

Wenn man die Methode `is_ok()` einmal erstellt hat, kann man sie in (fast) alle anderen Methoden der Klasse einbauen:

```
1  int get_kmstand(void)
2  {
3      if(is_ok()==false) // oder: if(!is_ok())
4          return 0;
5      else
6          return kmstand;
7  }
```

Prinzipiell sollte nämlich jede Methode zu Beginn ihrer Arbeit einmal mit `is_ok()` testen, ob der Zustand ihres Objekts in Ordnung ist oder nicht.

5.2.3.3 Zusammenfassung als Beispiel Hier ist noch einmal die Zusammenfassung als Quelltext (`pkw2alles.cpp`):

```
1 #include <stdio.h>
2 #include <string.h>
3 struct pkwtyp
```

[1] Welche Änderungen es sind, hängt vom betreffenden Sachgebiet ab: Im Bauwesen sollte keine Mauer dünner als null sein, im Steuerrecht keine Einkommensteuer größer als 100% usw. Wer programmiert, muss also vorher die entsprechenden Fachleute fragen.

```
4  {
5  private:
6     char kennz[20];
7     int kmstand;
8  public:
9     void init(const char *anfang_kennz, int anfang_kmstand)
10    {
11       kennz[0]='\0';
12       strncat(kennz, anfang_kennz, 19);
13       if(anfang_kmstand>0)
14          kmstand=anfang_kmstand;
15       else
16          kmstand=0;
17    }
18    bool is_ok(void)
19    {
20       if(strlen(kennz)<4)
21          return false;
22       if(kmstand<0)
23          return false;
24       return true;
25    }
26    const char* get_kennz(void)
27    {
28       return kennz;
29    }
30    int get_kmstand(void)
31    {
32       if(!is_ok())
33          return 0;
34       return kmstand;
35    }
36    void set_kennz(const char *k)
37    {
38       kennz[0]='\0';
39       strncat(kennz, k, 19);
40    }
41    void set_kmstand(int neuer_kmstand)
42    {
43       if(neuer_kmstand>kmstand)
44          kmstand=neuer_kmstand;
45    }
46    void printpkw(void)
47    {
48       printf("%s, %d\n", kennz, kmstand);
49    }
50 };
51 /*********************************************/
52 int main(void)
53 {
54    struct pkwtyp fw;
55    fw.init("BI-XY 123", 20000);
56    fw.set_kennz("GT-AB 321");
57    fw.set_kmstand(14000);
```

```
58      printf("Kilometerstand: %d\n", fw.get_kmstand());
59      fw.printpkw();
60      return 0;
61  }
```

5.2.4 Defaultwert bei `struct` und `class`

Wenn man in einem `struct` keines der Labels `public:` oder `private:` angibt, so sind alle Elemente dieses `struct` öffentlich zugänglich. Das gleiche gilt für alle Elemente, die in der Definition *vor* dem ersten `public:` oder `private:` liegen. Das ist gut so, denn dadurch sind die `struct`-Konstruktionen in C++ kompatibel zu C.

Neu in C++ ist der Datentyp `class`. `class` und `struct` sind miteinander fast identisch. Der einzige Unterschied: Bei `class` ist der Standardwert für den Zugriffsschutz anders. *Vor* dem ersten `public:` oder `private:` gelten hier alle Elemente als `private`.

Wir sollten ohnehin jeden Bereich eines `struct` oder einer `class` ausdrücklich mit `private:` oder `public:` kennzeichen. Derjenige, der die Klasse liest, freut sich darüber. Und dann ist es egal, ob man die Klasse als `struct` oder als `class` deklariert.

5.2.5 Zugriffsspezifizierer in UML

Den Zugriffsschutz kann man auch im UML-Klassendiagramm vermerken. `private`-Elemente werden am Beginn der Zeile mit einem Minuszeichen gekennzeichnet und `public`-Elemente mit einem Pluszeichen. Sonst ändert sich hier nichts (Abbildung 1).

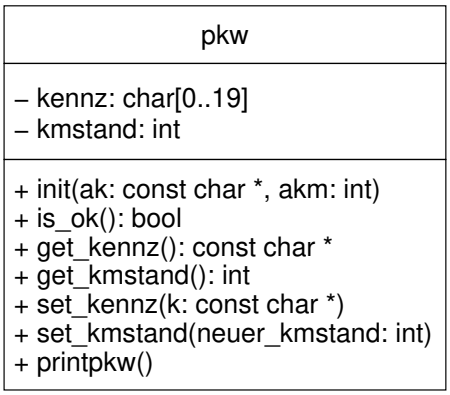

Abbildung 1: UML-Klassendiagramm zu `pkw2alles.c`