Nutzen Sie die Macht der Sprache
Programmieren einmal anders

Andreas Fertig, Philips Medizin Systeme Böblingen

Programmieren ist heute ein leichtes. Es gibt dutzende Programmiersprachen und viele Möglichkeiten sie zu erlernen. Ein wichtiger Teil, der dabei oft in Vergessenheit gerät, ist der zweite Teil des Wortes Programmiersprache: die Sprache. Meist konzentrieren wir uns darauf den Teil "Programmieren" zu erledigen. Im Embedded Umfeld haben Programme eine Lebensdauer von vielen Jahren. Das fehlerfreie Programmieren ist deshalb eine wichtige Aufgabe. Oft vernachlässigt wird, dass es beim Programmieren noch ein weiteres wichtiges Ziel gibt und zwar die Kommunikation mit den anderen Entwicklern. Mithilfe der Sprache können wir unsere Intention über Jahre hinweg kommunizieren. Durch Sprache wird die Fehlersuche unterstützt und das bestehende Programm lässt sich leichter erweitern. Mit der Weiterentwicklung des C++ Standards zu C++11 und 14 ergeben sich neue Möglichkeiten unsere Ausdrucksweise weiter zu verbessern.

Programmieren und Programmiersprachen werden vorrangig zum Lösen von Aufgaben eingesetzt. Mithilfe des Programmierens soll etwas Neues geschaffen werden oder etwas Existierendes verbessert bzw. repariert werden. Nach Springer Gabler wird Programmiersprache, wie folgt, definiert:
„*Eine Programmiersprache ist eine künstliche Sprache zur Verständigung zwischen Mensch und Computer. Sie ist durch ihre Syntax und Semantik definiert. In einer Programmiersprache stellt man Verfahren zur Problemlösung in einer für den Computer „verständlichen" Form dar.* [1]**„**

In dieser Definition steht die Kommunikation mit dem Computer im Vordergrund, der Fokus liegt erneut auf dem Lösen einer Aufgabenstellung. Streichen wir den ersten Teil des Worts, bleibt Sprache übrig. Gemäß Pons ist Sprache definiert als
„*System von Zeichen (das der Kommunikation o.Ä. dient)* [2]**"**

Eine Programmiersprache dient also nicht allein der Lösung von Problemen und der Kommunikation mit dem Compiler. Durch die Programmiersprache kommunizieren wir ebenfalls mit anderen Menschen, die mit unserem Code arbeiten. Sie übernehmen den Code von uns oder führen Änderungen an ihm durch. Code ist somit neben der Spezifikation eine weitere Form der Dokumentation, wobei nur der Code die tatsächliche Implementierung widerspiegelt. Deshalb sollte der Code auch für den Leser geschrieben sein.

Google hat das erkannt und in den Coding-Regeln entsprechend berücksichtigt. In seinem Vortrag auf der cppcon 2014 hat Titus Winter die Philosophie der Coding-Regeln von Google, wie folgt, erklärt: **„***Code is going to live a long time, and be read many times. We choose explicitly to optimize for the reader, not the writer.***"**

Wie in Abb. 1 gezeigt wird, erfüllt der Code den Aspekt mit dem Computer zu kommunizieren, während er gleichzeitig für die Kommunikation unter Menschen sowie für Dokumentationszwecke verwendet wird.

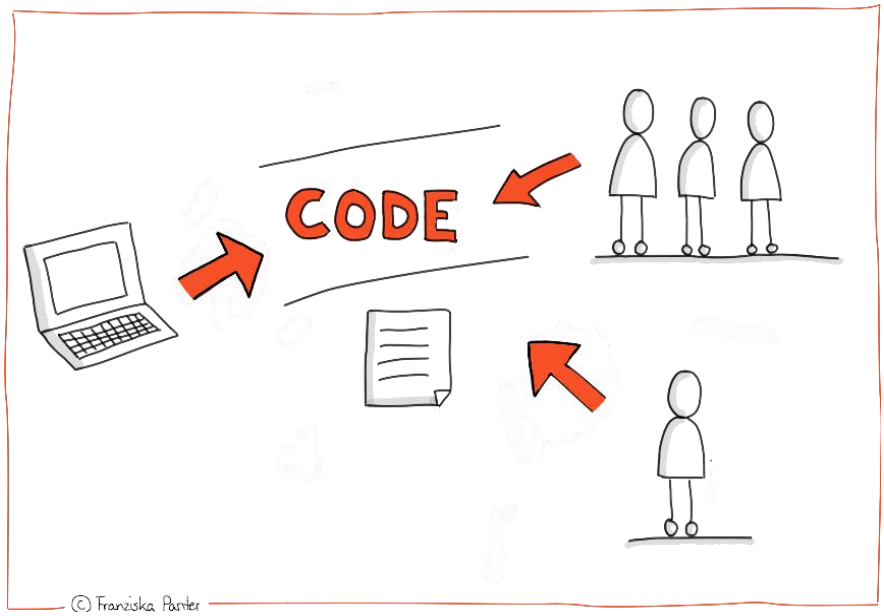

Abb. 1**:** Code wird auch von anderen Menschen gelesen oder

Der erste Aspekt ist soweit klar. Die anderen beiden werden jedoch oft übersehen. Jede Sprache hat die Eigenschaft, dass sich Sätze unterschiedlich formulieren lassen. Beispiele dafür sind Kommasetzung oder Rechtschreibung. Menschen erkennen den Sinn eines Satzes auch, wenn dieser leicht falsch geschrieben ist oder ein Komma fehlt. Unser bester Freund, der Compiler, versucht Menschen nachzueifern. Er versucht, solange semantisch korrekt, jeglichen Code in eine ausführbare Form zu übersetzen. Dieser Code ist dann in den allermeisten Fälle lauffähig. Das folgende Beispiel ist übersetzbar und lauffähig:

```
void PrintString(char* str) {
   for( float x=0.0; nullptr != str[x]; x+=1.0 ) {
       printf( „%c\n“, str[x] );
   }
}
```

Wahrscheinlich würde kaum ein Programmierer an dieser Stelle einen `float` verwenden. Ungeachtet dessen gilt: es compiliert und läuft!

Was nicht erkennbar ist, ist die Intention des Autors. Gab es einen Grund den Datentyp `float` zu verwenden oder war es schlicht Unkenntnis? Sofern der Einsatz von `float` Absicht war, sollte der Grund direkt an der eingesetzten Stelle kommentiert werden.

Weniger auffällig sind Beispiele mit Referenzen, Zeigern oder `const`. Speziell C-Entwickler neigen dazu aus Gewohnheit Zeiger zu verwenden. Zeiger funktionieren semantisch fast genauso wie Referenzen. Dabei haben Referenzen zwei entscheidende Vorteile: sie können nie `NULL` sein und sie müssen immer direkt initialisiert werden.

Wenn wir einen Zeiger als optional interpretieren und Referenzen als erforderlich, haben wir ein gutes Stück Ausdrucksmöglichkeit gewonnen. Eine Funktion, die eine Referenz als Parameter entgegen nimmt, erfordert demnach einen Wert. Nimmt die gleiche Funktion einen Zeiger entgegen, ist der Wert optional und kann durch `NULL` weggelassen werden. Jetzt ist es die Aufgabe der gerufenen Funktion mit dem Umstand eines `NULL`-Zeigers zurecht zu kommen.

Im Falle der Referenz als Parameter spart sich der Autor nebenbei auch die oft lästige `NULL`-Zeigerprüfung im Funktionsrumpf. Ein positiver Nebeneffekt ist, dass der resultierende Code kleiner ist und die Prüfung nicht vergessen werden kann. Weiter signalisiert eine Referenz bereits nach außen, dass die Funktion ist nur mit validen Werten aufrufbar ist.

Ein anderes Beispiel ist `const`. Bei einem Parameter kann der Aufrufer sich darauf verlassen, dass die übergebenen Daten unverändert bleiben. Eine komplette Funktion kann als Konstant markiert werden. Ein Aufruf verändert dann die Klasse nicht. Bauen wir die `const`-Hierarchie weiter aus, erhalten wir ein vollständiges konstantes Objekt. Im Bereich eingebetteter Systeme ist das ein hohes Gut, da das Objekt nie kopiert oder initialisiert werden muss. Es kann direkt im ROM, Read-only Memory, abgelegt werden. In diesem Fall wird die Kommunikation mit dem Compiler effektiv genutzt. Natürlich läuft der gleiche Code auch ohne `const`. Er braucht dann jedoch mehr Ressourcen.

Mit neuen Sprachelementen wie `override` können wir ausdrücken, dass eine geerbte Funktion überschrieben werden soll. Leider ist dieses Schlüsselwort erst spät zu C++ hinzugekommen. Diesem Umstand ist die gewöhnungsbedürftige Semantik geschuldet. Die Verwendung von `override` ist optional. Wir können bisher Funktionen, die `virtual` sind, einfach überschreiben. Der Gewinn liegt in der neuen Ausdrucksmöglichkeit. Programmierer können nun ausdrücken, dass genau diese Funktion überschrieben werden soll. Sie erwarten, dass die Funktion in der Vererbungshierarchie existiert. Jetzt sehen Kollegen, dass die Funktion aus einer Basisklasse kommt, sogar kommen muss. Als weiteres Plus unterstützt der Compiler und weist darauf hin, wenn keine solche Funktion in der Hierarchie existiert. Ohne `override` würde er die Funktion, als neue eigenständige Funktion, anlegen. Erst bei einer Fehlersuche wäre mühsam herauszufinden, warum sich die Funktion ganz anders verhält, als gedacht. Vielleicht wird sie sogar nie aufgerufen.

Aussagekräftiger und klarer Code ist ein hohes Gut. Abgesehen vom Compiler und Wartung, gibt es Kollegen, die mit dem Code arbeiten. Sie verwenden Funktionen oder Klassen, die andere bereitgestellt haben. Bereits 1993 schrieb Robert Murray in C++ Strategies and Tactics: **„***If you use object-oriented technology, you can take any class someone else wrote, and by using it as a base class, refine it to do a similar task.* [3]**“**

Behalten Sie diesen Satz immer im Hinterkopf. Die Natur von C++ ermöglicht es auf einfache Art und Weise Funktionen oder Klassen weiter zu nutzen. Mittels `override` und `final` können wir sehr gezielt entscheiden, wann die Vererbungsmöglichkeit enden soll. Diese Entscheidung können wir sowohl auf Funktions- als auch Klassenebene treffen.

Eine Programmiersprache dient zu mehr als nur der Lösung eines Problems. Sie ist zugleich Dokumentation und Kommunikation. Neben dem Compiler wird sie auch von Menschen gelesen. Darum gilt: das Auge liest mit. Nutzen Sie die Sprachmittel, um ihrem Code mehr Klarheit zu geben. Schützen Sie ihn gegen ungewollten Einsatz.

Literatur- und Quellenverzeichnis

[1] Spinger Gabler, Programmiersprache,
[2] Duden, Sprache,
[3] Murray, Robert B., C++ Strategies and Tactics, 1993

Autor

Andreas Fertig studierte Informatik in Karlsruhe. Seit 2010 ist er für die Philips Medizin Systeme als Softwareentwickler mit Schwerpunkt eingebettete Systeme tätig. Er verfügt über fundierte praktische und theoretische Kenntnisse von C++ auf verschiedenen Betriebssystemen. Nebenbei arbeitet er als Dozent und entwickelt verschiedene Mac OS X Anwendungen.

.

Kontakt

Internet: https://www.andreasfertig.info
Email: contact@andreasfertig.info