MICRO CONSULT

Interface-Designs und ihre Implementierungen – Teil 2: Realisierung und Zugriff

Der Einsatz von Software-Interfaces ist ein elementares Mittel zur Entwicklung von langlebigen und tragfähigen Software-Architekturen. Deshalb sollten sie so früh wie möglich in der Architektur etabliert werden, um diese zu stabilisieren. Der Software-Architekt kann so eine schnelle Aufgabenverteilung auf unabhängige Personen, Teams oder Standorte ohne weitere „Reibungsverluste“ sicherstellen.

Welche Varianten der Architekt beim Interface-Design kennen sollte und wie diese in den Programmiersprachen C und C++ implementierbar sind, verrät dieser Beitrag in seinem zweiten Teil – er zeigt Implementierungsansätze durch Assoziation, Komposition, Fassade, virtuelle Interfaces, nicht-virtuelle Interfaces, C++ Templates und CRTP-Pattern auf.

Konkrete Interface-Implementierungsbeispiele

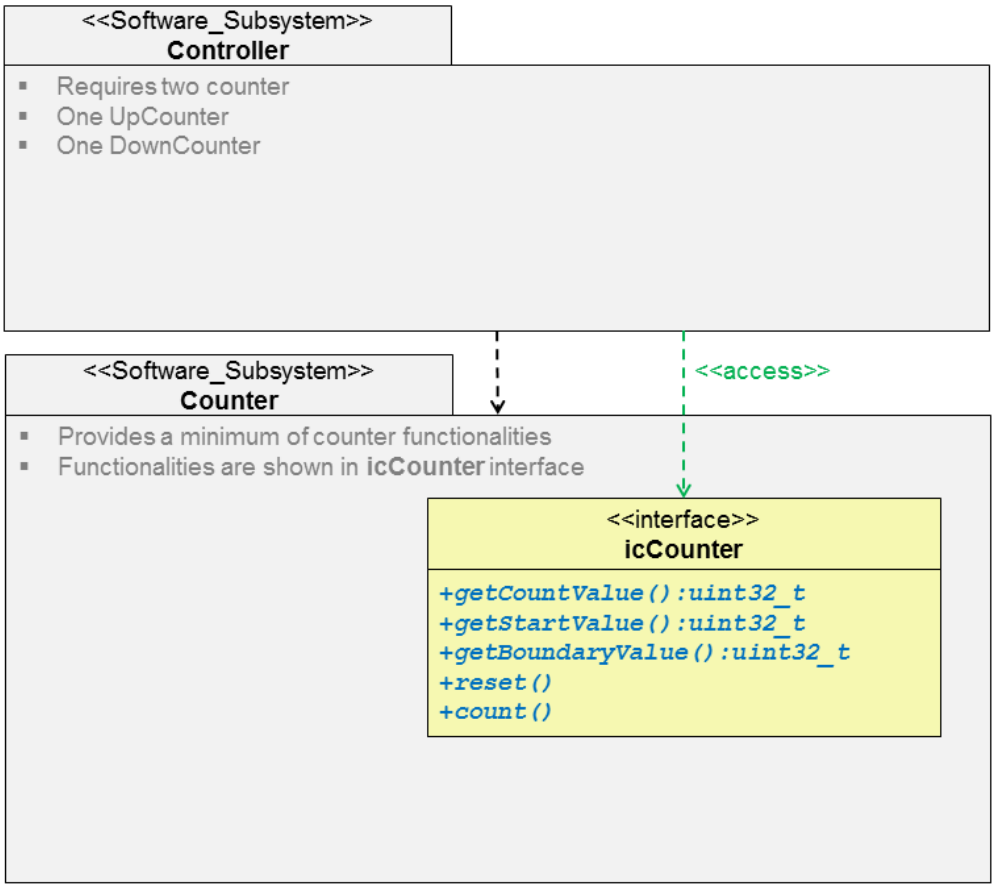

Bild 1: Grundlage für konkrete Implementierungsbeispiele

Das Software-Subsystem Controller enthält eine Klasse cController, die zwei Counter – einen UpCounter und einen DownCounter – benötigt. Hierfür bietet das Software-Subsystem Counter dem Controller das Interface icCounter an, um mit den Countern zu arbeiten.

Implementierungsansätze	Programmcode downloadbar in
Assoziation ohne Interface-Klasse	C und C++
Aggregation ohne Interface-Klasse	C und C++
Komposition ohne Interface-Klasse	C und C++
Fassade-Pattern	C und C++
Interfaceklasse mit rein virtuellen Funktionen	C und C++
Interfaceklasse mit rein virtuellen und implementierten Funktionen	C und C++
Interface als Template-Parameter	C++
CRT-Pattern	C++

Bild 2: Übersicht Implementierungsansätze

Der den Implementierungsbeispielen entsprechende Programmcode ist hier bereitgestellt.

Assoziation ohne Interfaceklasse

Aus dem Software-Subsystem Controller greift die Klasse cController über zwei Zeiger direkt auf je ein Objekt vom Typ cUpCounter und cDownCounter zu. Dabei ergeben sich zwischen den beiden Software-Subsystemen Controller und Counter zwei Abhängigkeiten (Include-Pfade).

Hier wurde mit Absicht nicht die Assoziation von cController auf cCounter gezogen, damit keine virtuellen Funktionen bzw. Funktionszeiger notwendig sind, aber zum Preis der stärkeren Kopplung.

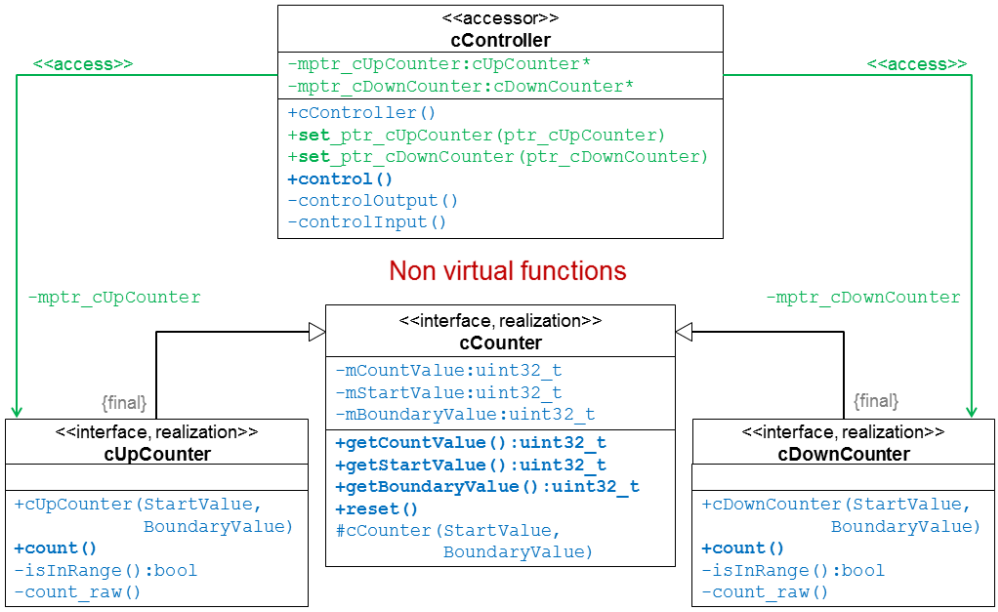

Bild 3: Assoziation

Aggregation ohne Interfaceklasse

Im Vergleich zu dem vorherigen Assoziationsbeispiel übernimmt hier die Klasse cController direkt die Instanziierung der benötigten Objekte vom Typ cUpCounter und cDownCounter. Die Instanziierung erfolgt hier dynamisch auf dem Heap mittels malloc() in C und new() in C++.
Damit erreichen wir die Grundidee der Weitergabe (des „Ausbaus“) der erzeugten Counter-Objekte bei der Aggregation. Der Einsatz des Heaps in der Embedded-Softwareentwicklung ist in vielen Projekten verboten, da er u.a. mit Risiken der Fragmentierung verbunden und damit nicht vorhersagbar bzw. nicht echtzeitfähig ist.

Wie bei der Anwendung der Assoziation ergeben sich auch bei der Aggregation für dieses Beispiel zwei Abhängigkeiten. Ebenfalls wurde gezielt auf die Anwendung von virtuellen Funktionen / Funktionszeiger verzichtet.

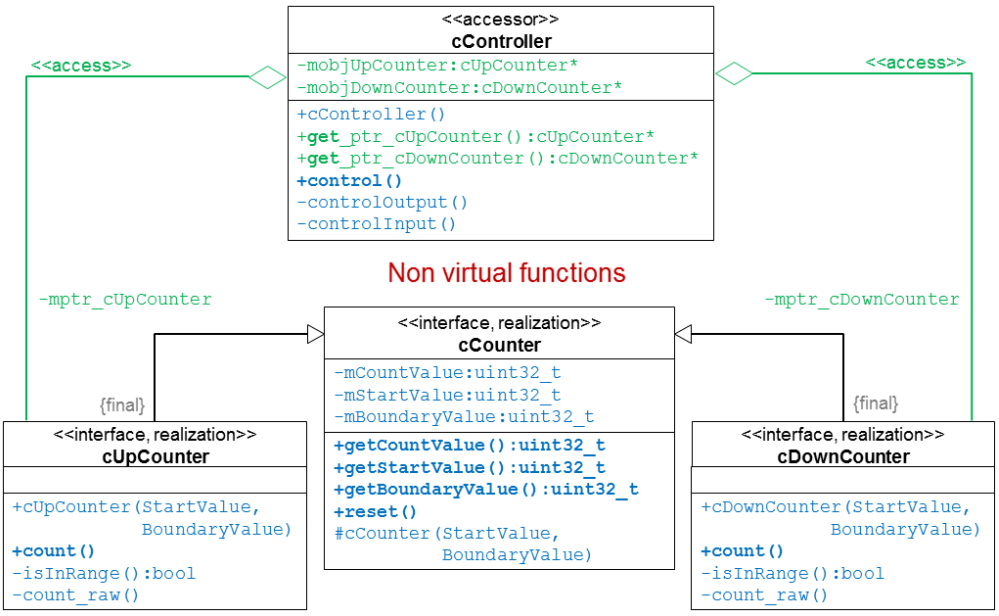

Bild 4: Aggregation

Komposition ohne Interfaceklasse

Bei den Varianten mit Assoziation und Aggregation erfolgen die Objektzugriffe jeweils mit Zeigern (optional mit Referenzen). Der komplette Verzicht auf Zeiger führ zur Anwendung der Komposition.

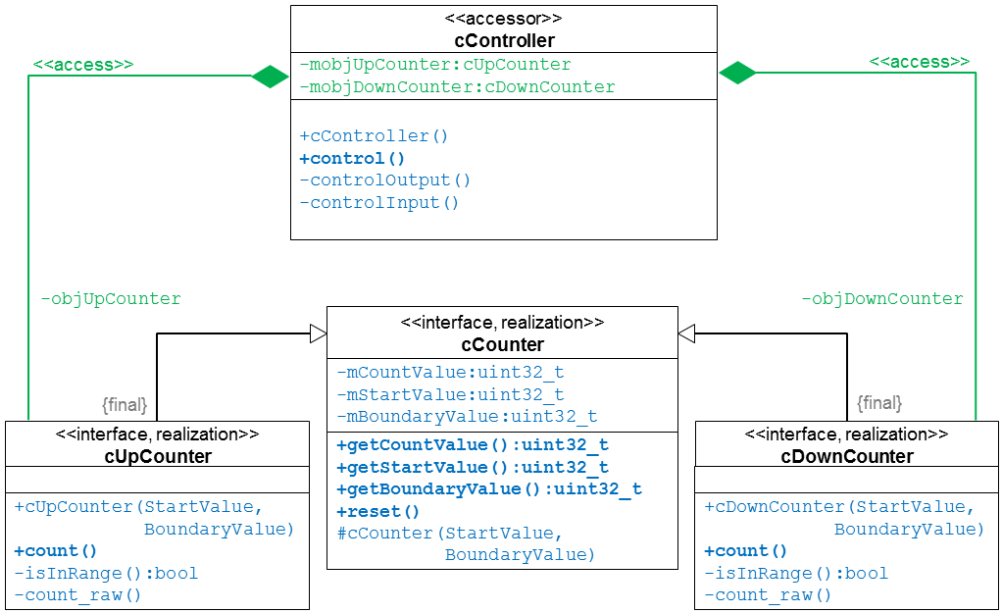

Bild 5: Komposition

Hierbei enthält die Klasse cController als Members Objekte der Klassen cUpCounter und cDownCounter. Die Anzahl der Abhängigkeiten bleibt bei zwei, wobei die Kopplung durch die eingebetteten Objekte bei der Komposition gegenüber der Assoziation und Aggregation verstärkt wird.

Fassade-Pattern

Das Fassade-Pattern bietet dem Zugreifer cController das Interface icCounter an, welches bereits Daten und Funktionsimplementierungen enthält.

Was sich hinter der Fassade icCounter verbirgt, ist für den Zugreifer nicht wissenswert und auch nicht sichtbar.

Wie bei den vorherigen Beispielen enthält diese Implementierung keine virtuellen Funktionen oder Funktionszeiger. Es ergibt sich nur eine Abhängigkeit (Include-Pfad) und damit eine geringe / lose Kopplung. Das Interface (Fassade) enthält ein Zähler- und ein Grenzwert-Prüfobjekt, die gemeinsam die komplette Funktionalität der Fassade realisieren.

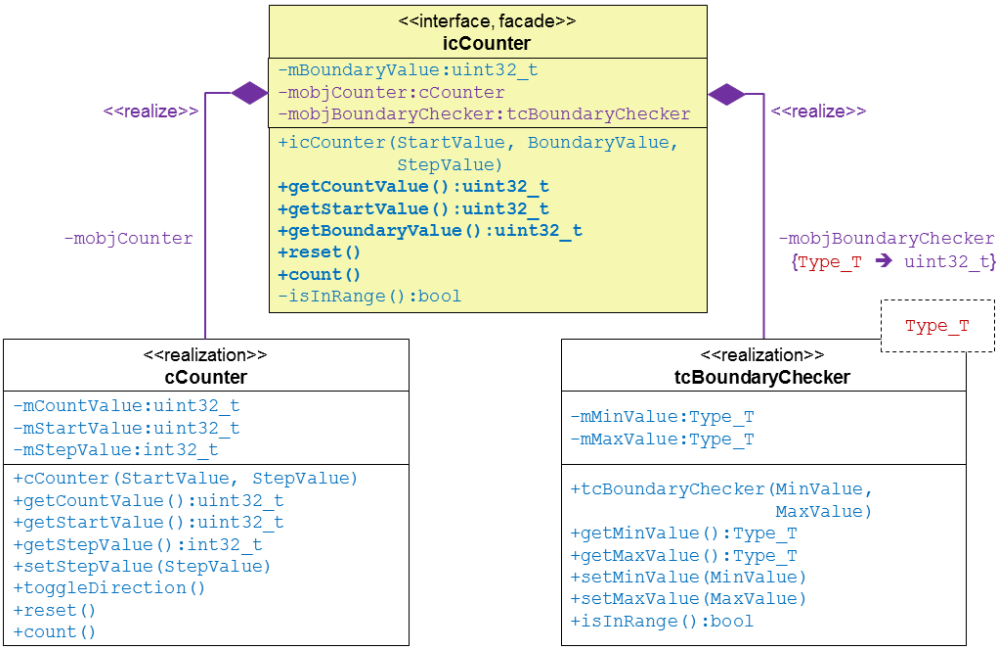

Bild 6: Fassade-Pattern – Interfacerealisierung

Interfaceklasse mit rein virtuellen Funktionen

Ein klassischer, aus der objektorientierten Welt stammender Interfaceansatz ist die Verwendung von rein virtuellen Funktionen (nur Deklarationen ohne Implementierungen) im Interface. Des Weiteren enthält das Interface icCounter keine Daten.

Der Zugriff auf das Interface erfolgt im cController durch einen Zeiger / eine Referenz vom Typ des Interfaces. Dieser Zeiger / diese Referenz muss später auf ein Objekt der Interface-realisierenden Klassen zeigen.

In C gibt es keine virtuellen Funktionen, daher müssen dort die Mechanismen des C++ Compilers manuell mit Hilfe von Funktionszeigertabellen nachgebildet werden.

Was sich hinter dem Interface verbirgt, ist für den Zugreifer cController zunächst nicht wissenswert und auch nicht sichtbar. Erst beim Initialisieren der Zeiger / Referenzen müssen konkrete Objekte von cUpCounter und cDownCounter vorhanden sein.

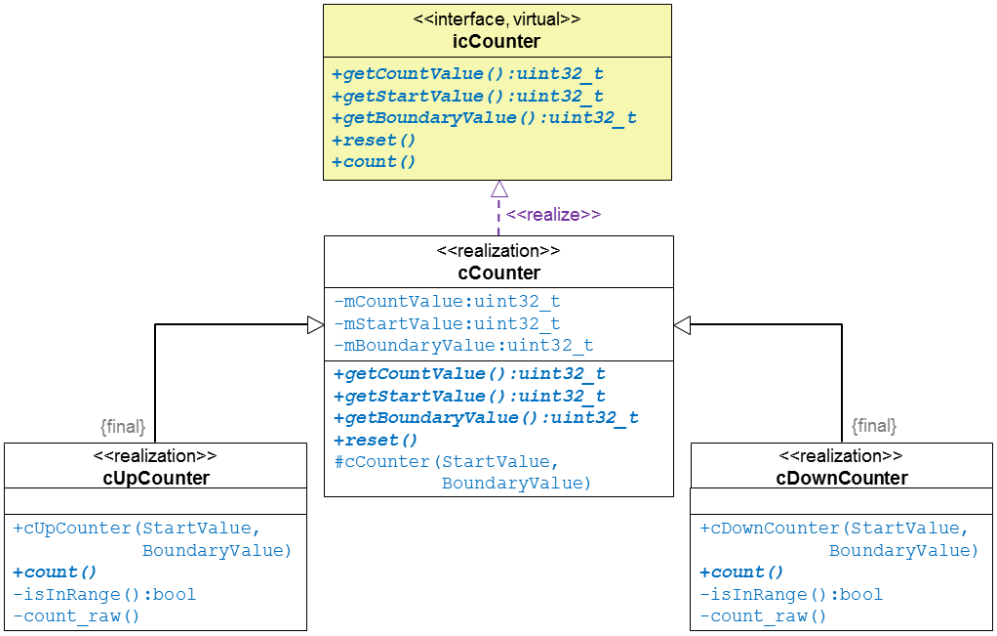

Bild 7: Virtual Interface – Interfacerealisierung

Interfaceklasse mit nicht nur rein virtuellen Funktionen

Diese Implementierungsvariante basiert auf dem C++ Idiom Non-Virtual Interface (NVI). Bei der Implementierung von rein virtuellen Interfaces ergeben sich im Falle mehrerer Implementierungen typischerweise redundante Programmcode-Anteile.

Das Idiom Non-Virtual Interface implementiert diesen gemeinsamen Code bereits in der Interface-Funktion. Nur die kleinen varianten Anteile der typspezifischen Implementierung sind im Interface als rein virtuelle Funktionen deklariert (isInRange()) und (count_raw()) und bereits in anderen implementierten Funktionen (count()) aufgerufen.

Nur die beiden typspezifischen virtuellen Funktionen isInRange() und count_raw() sind jeweils in den Klassen cUpCpounter und cDownCounter individuell implementiert.

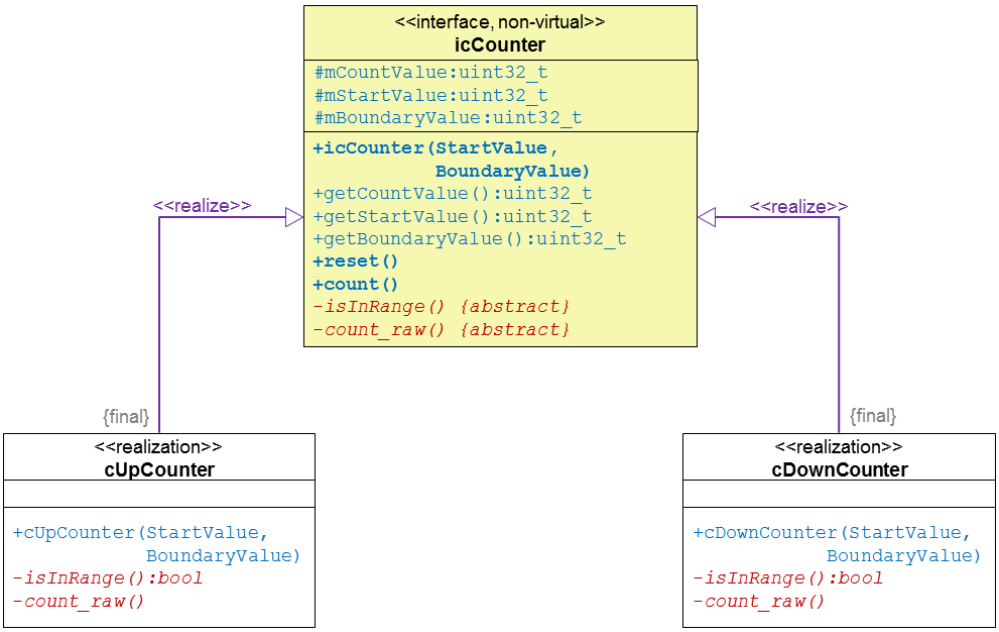

Bild 8: Non-Virtual Interface

In den Beispielen mit virtuellen Funktionen in Interfaces oder/und Klassen führt das Ergebnis auf dem Taget zu einer dynamischen Bindung. Dadurch entstehen verschieden Aufwände, die aber im Vergleich zum Nutzen in dem meisten Fällen vernachlässigbar sind:

1. **Programmspeicher** (und Compilezeit)
   Zur Compilezeit erzeugt die Toolkette zu jeder Klasse, die eine oder mehrere virtuelle Funktionen deklariert oder/und implementiert, eine VMT (Virtual Method Table). Diese legt der Linker / Lokator üblicherweise in den Programmspeicher. Die VMT enthält die Funktionseinsprung-Adressen zu den klassenspezifischen Funktionsimplementierungen.
2. **Datenspeicher** und **Laufzeit**
   Objekte, instanziiert aus einer Klasse mit virtuellen Funktionen, enthalten als zusätzliches, erstes Attribut die Einsprung-Adresse in dessen Klassen-VMT. Dieses Attribut fügt der Compiler automatisch hinzu, und der Konstruktor initialisiert es ebenfalls automatisch.

3. **Laufzeit**
   Beim Aufruf einer virtuellen Funktion für ein bestimmtes Objekt über einen Zeiger oder eine Referenz wird über dessen VMT-Einsprung-Adresse die Funktionsadresse aus der VMT gelesen und anschließend zu dieser Funktionsadresse gesprungen (-> eine In-Direktion mehr als beim Aufruf nicht-virtueller Funktionen). Dies ist die Funktionalität der dynamischen Bindung. Sie erlaubt so die Programmierung der **dynamischen Polymorphie**. Diesen Mechanismus führt die C++ Toolkette automatisch aus. In C ist die dynamische Bindung mit etwas mehr Programmieraufwand manuell nachbildbar.

Interface als Template-Parameter

Die in den Beispielen mit rein virtuellen Interfaces und nicht rein virtuellen Interfaces beschriebenen Aufwände und die damit verbundenen Risiken lassen sich durch die Anwendung von Template-Klassen eliminieren. Der Preis dafür ist der Verzicht auf die dynamische Polymorphie, die in vielen Embedded-Softwaresystemen nicht zwingend erforderlich ist. Es ergibt sich nur eine **statische Polymorphie**.

Der Interface-Zugreifer tcController bekommt als Template-Parameter die Objekt- / Interfacetypen CounterA_T und CounterB_T, deren Realisierungen cUpCounter und cDownCounter er adressieren möchte. Eine direkte Abhängigkeit im Programmcode zwischen den Software-Subsystemen Controller und Counter gibt es nicht mehr. Die indirekte Abhängigkeit entsteht durch die Template-Typisierung bei der Instanziierung. Die spezifizierten Typen cUpCounter und cDownCounter müssen alles bereitstellen, was der Zugreifer tcController aufruft. Ist das nicht der Fall, meldet dies bereits der Compiler als Fehler (kein Laufzeitfehler!).

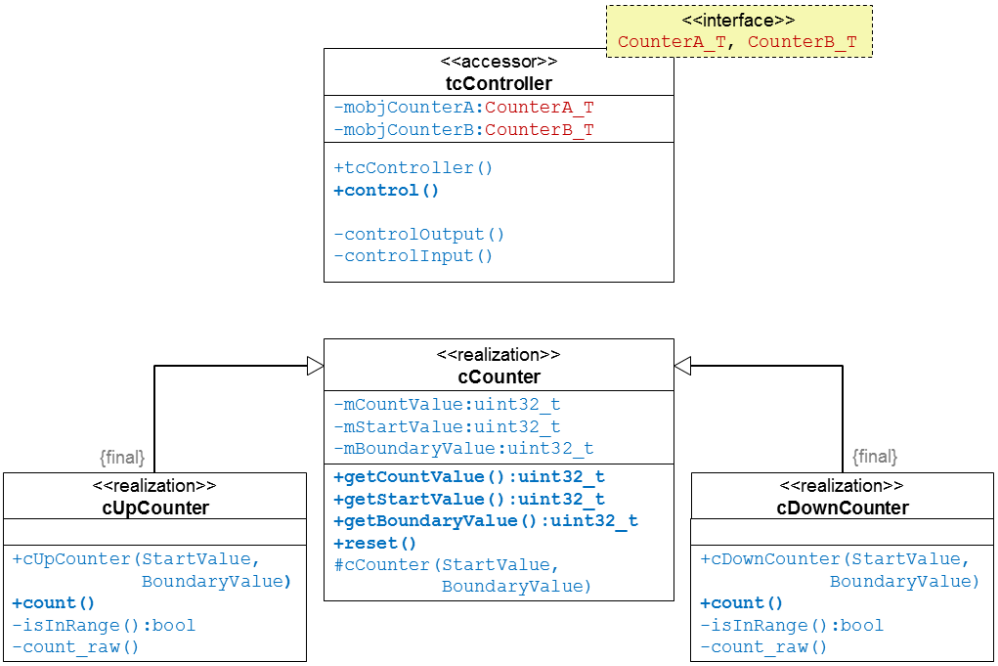

Bild 9: Interface als Template-Parameter

Das Interface selbst ist bei diesem Interface-Design nicht eindeutig sichtbar und nur indirekt im Zugreifer durch dessen Aufrufe spezifiziert. Diese Problematik verbessert sich durch die Anwendung des Curiously Recurring Template Pattern (CRTP) [3].

Curiously Recurring Template Pattern (CRTP)

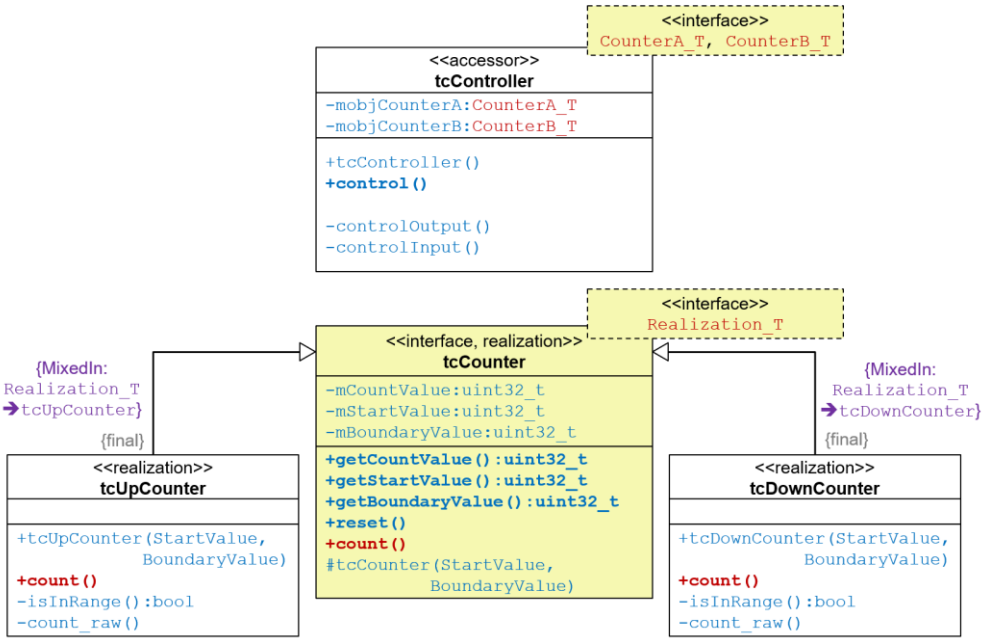

Bild 10: CRT-Pattern

Die Template-Klasse tcCounter bzw. deren Funktionen repräsentieren das komplette Interface und sind vom Zugreifer tcController aufrufbar:

mobjCounterA.count();
mobjCounterB.count();

Die Interface-Funktion count() der Interface-Template-Klasse tcCounter ruft als Delegate über den Template-Parameter Realization_T die count() Funktion der realisierenden Klasse tcUpCounter bzw. tcDownCounter auf:

```
template <typename Realization_T>

void tcCounter<Realization_T>::count()

{

static_cast<Realization_T*>(this)->count();

}
```

Der Template-Parameter Realization_T wird bereits direkt bei der Vererbung von tcCounter in tcUpCounter und tcDownCounter gesetzt:

class **cUpCounter** final : public **tcCounter<cUpCounter>**
class **cDownCounter** final : public **tcCounter<cDownCounter>**

Das Setzen eines Template-Parameters mit sich selbst als Klassentyp wird als **MixedIn** bezeichnet.

Resümee

Die Entscheidung für das „richtige“ Interface-Design ist immer von den geltenden Software-Anforderungen abhängig.

Interfaces unterstützen positiv die Umsetzung von Software-Qualitätsmerkmalen, wie beispielsweise Wiederverwendbarkeit, Portabilität, Austauschbarkeit und Erweiterbarkeit. Interface-Konzepte sind ein geeignetes Mittel zur Erfüllung von Software-Entwurfsprinzipien, z.B. lose Kopplung, Externalisierung von Abhängigkeiten, Modularisierung und Erreichen einer hohen Kohäsion.

Ein weiterführendes Konzept zu und mit Interfaces sind **Ports**. Ein Port vereint thematisch null bis unendlich viele bereitgestellte Interfaces und null bis unendlich viele erwartete Interfaces und lässt sich mit anderen kompatiblen Ports verbinden.

Erfahren Sie in Teil 1 des Beitrags alles über Interface-Konzepte und unterschiedliche Interface-Typen.

Holen Sie sich das richtige Wissen darüber, welche Varianten Sie beim Interface-Design kennen sollten und wie diese in den Programmiersprachen C und C++ implementierbar sind.

MicroConsult bietet Ihnen professionelle Trainings und Coachings rund um die Themen Analyse, Design und Architektur uvm. an – auch im Live-Online-Format.

Weiterführende Informationen

Trainings zum Thema – auch im Live-Online-Format:

- Requirements Engineering und Management für Embedded-Systeme
- Software-Architektur-Schulung für Embedded- und Echtzeitsysteme
- Embedded C++ für Fortgeschrittene: Objektorientierte Programmierung für Mikrocontroller mit C++/EC++
- Embedded-Software-Design und Patterns mit C
- Interfacedesign – Analyse, Design und Architektur

Alle Trainings & Termine auf einen Blick

MicroConsult Fachwissen zum Thema Embedded SW-Entwicklung

Autor

Thomas Batt studierte nach seiner Ausbildung zum Radio- und Fernsehtechniker Nachrichtentechnik. Seit 1994 arbeitet er kontinuierlich in verschiedenen Branchen und Rollen im Bereich Embedded-/Realtime-Systementwicklung. 1999 wechselte Thomas Batt zur MicroConsult GmbH. Dort verantwortet er heute als zertifizierter Trainer und Coach die Themenbereiche Systems/ Software Engineering für Embedded-/Realtime-Systeme sowie Entwicklungsprozess-Beratung.