

# Interface-Designs und ihre Implementierungen

## Struktur, Definition, Realisierung und Zugriff

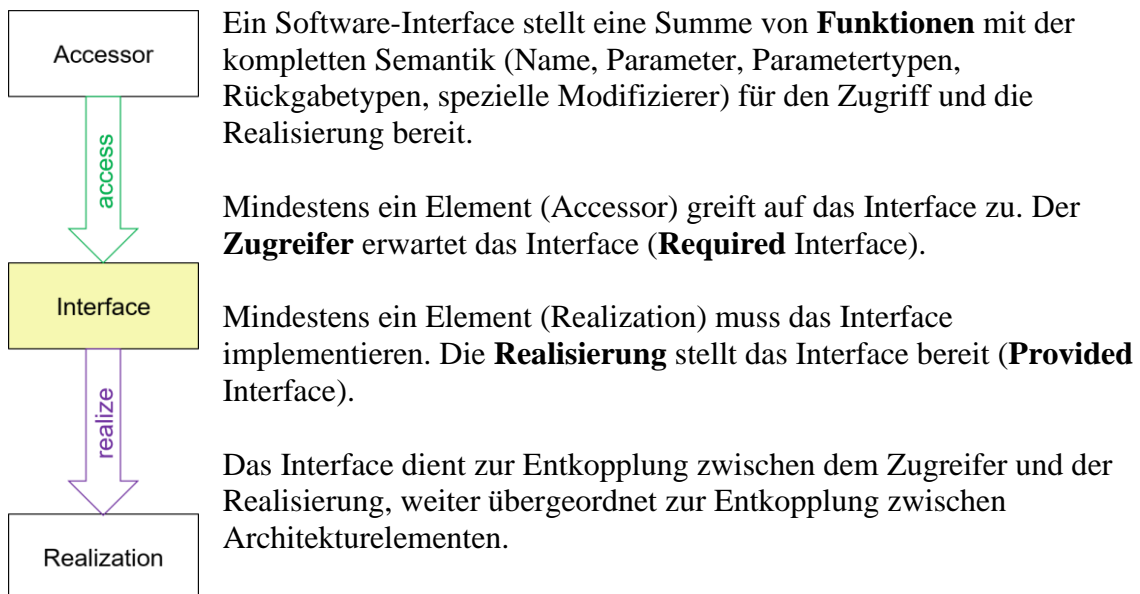
Thomas Batt, MicroConsult GmbH

### 0. Vorwort

Die konsequente Nutzung von Software-Interfaces ist ein elementares Mittel zur Entwicklung von langlebigen und tragfähigen Software-Architekturen. Ziel des Software-Architekten muss es sein, Interfaces so früh wie möglich in der Architektur zu etablieren und diese zu stabilisieren. Nur so ist ohne weitere „Reibungsverluste“ eine schnelle Aufgabenverteilung auf unabhängige Personen, Teams oder Standorte durchführbar. Später Interfaceänderungen sind nur mit der Zustimmung des Software-Architekten erlaubt.

Welche Varianten der Architekt beim Interface-Design kennen sollte und wie diese in den Programmiersprachen C und C++ implementierbar sind, zeigt dieser Beitrag [1]. Um diesen Beitrag vollständig zu verstehen, sind Kenntnisse der UML [2], der objektorientierten Programmierung und der Programmiersprachen C und C++ vorausgesetzt.

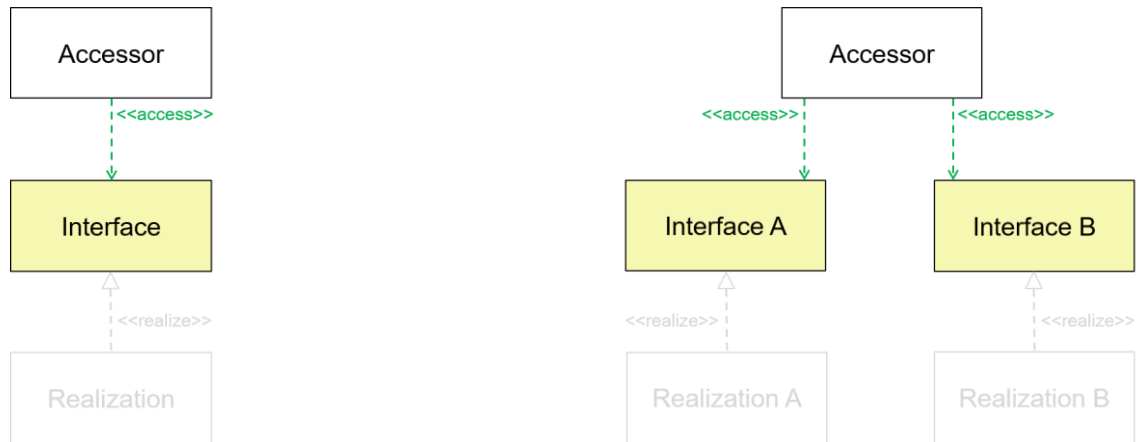
### 1. Interface-Konzept



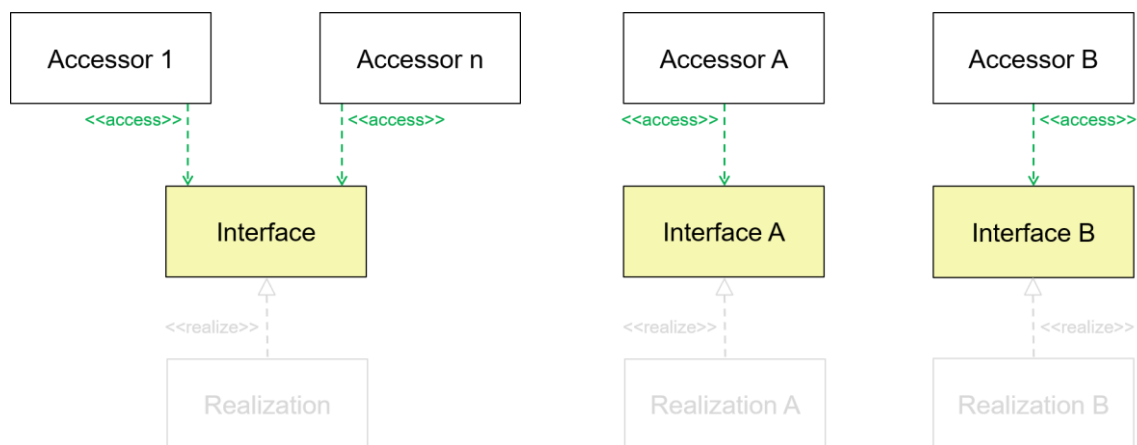
**Bild 1:** Interface-Konzept

## 2. Designvarianten des Interface-Zugriffs

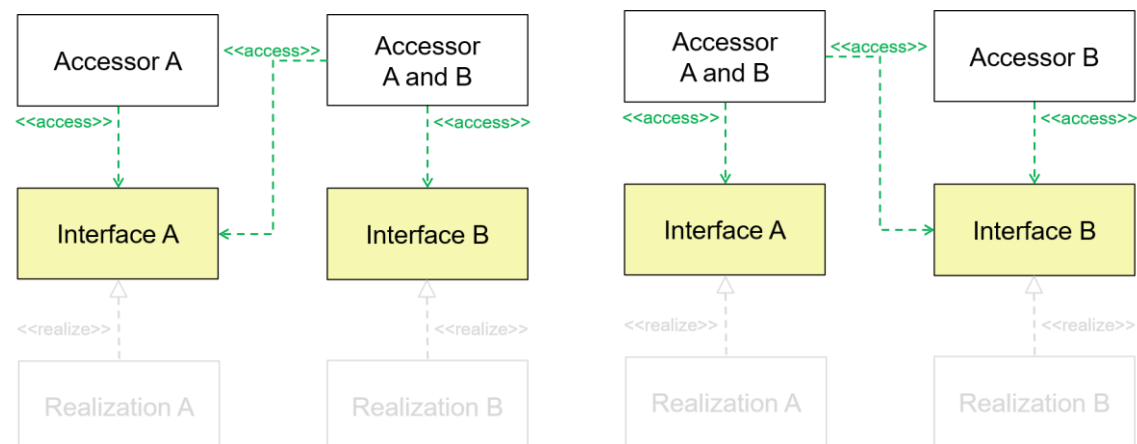
Der folgende Teil zeigt verschiedene Designvarianten zwischen Zugreifer und Interface(es).



**Bild 2:** Ein Zugreifer greift auf ein oder mehrere Interfaces zu

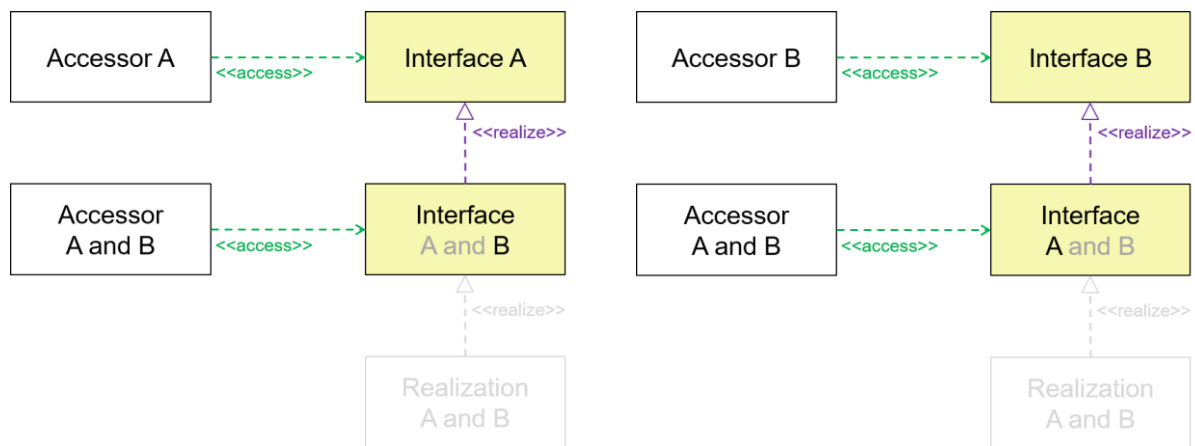


**Bild 3:** Mehrere Zugreifer greifen auf ein oder jeweils ein Interface zu

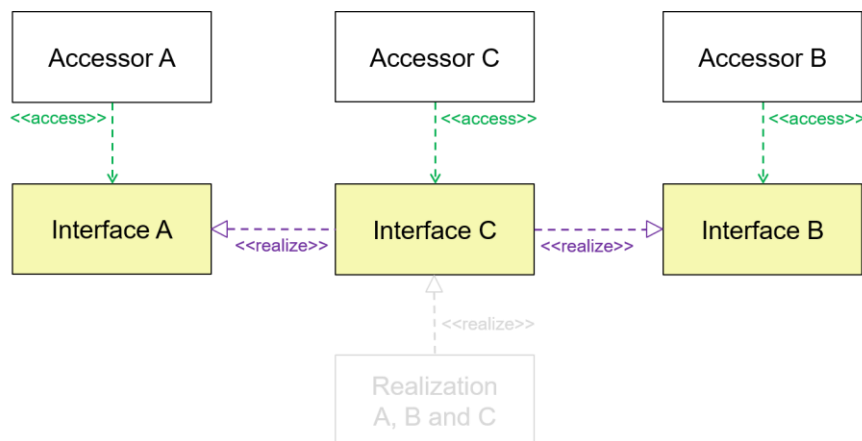


**Bild 4:** Mehrere Zugreifer greifen auf ein oder mehrere Interfaces zu

Die in Bild 4 dargestellten Varianten sind in die folgenden überführbar. Dabei ergeben sich unterschiedliche Interface-Ebenen:

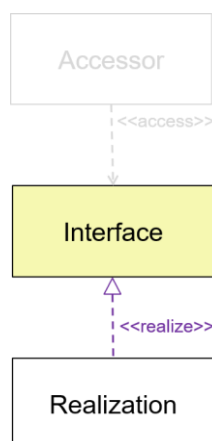


**Bild 5:** Mehrere Zugreifer auf unterschiedliche Interface-Ebenen



**Bild 6:** Mehrere Zugreifer auf jeweils unterschiedliche Interface-Ebenen

### 3. Designvarianten der Interface-Realisierung



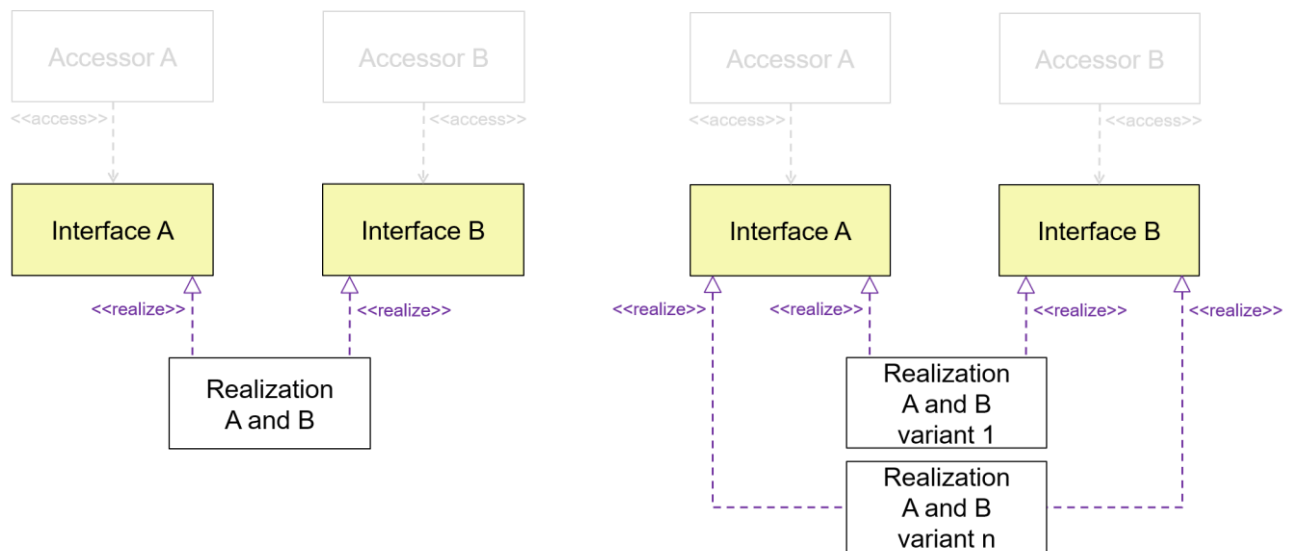
Der vorherige Abschnitt zeigte Interface-Designs mit dem Schwerpunkt Interface-Zugriff.

In diesem Teil verlagert sich der Schwerpunkt in die Interface-Realisierung.

**Bild 7:** Ein Interface mit einer Realisierung



**Bild 8:** Ein Interface mit mehreren oder partiellen Realisierungen



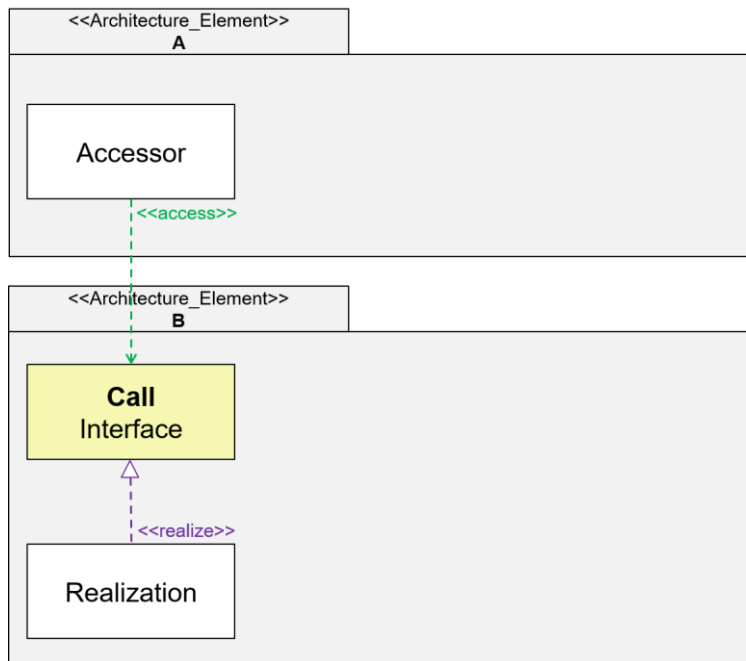
**Bild 9:** Mehrere Interfaces in einer oder mehreren Realisierungsvarianten

#### 4. Interface-Typisierung

Interfaces lassen sich (abhängig von verschiedenen Zugreifern) thematisch aufteilen, beispielsweise pro Architekturelement jeweils eines für die Konfiguration, die Diagnose und den Normalbetrieb. Dieser Ansatz ist hier nicht weiter thematisiert.

Generell lassen sich Interfaces in drei unterschiedliche Typen kategorisieren:

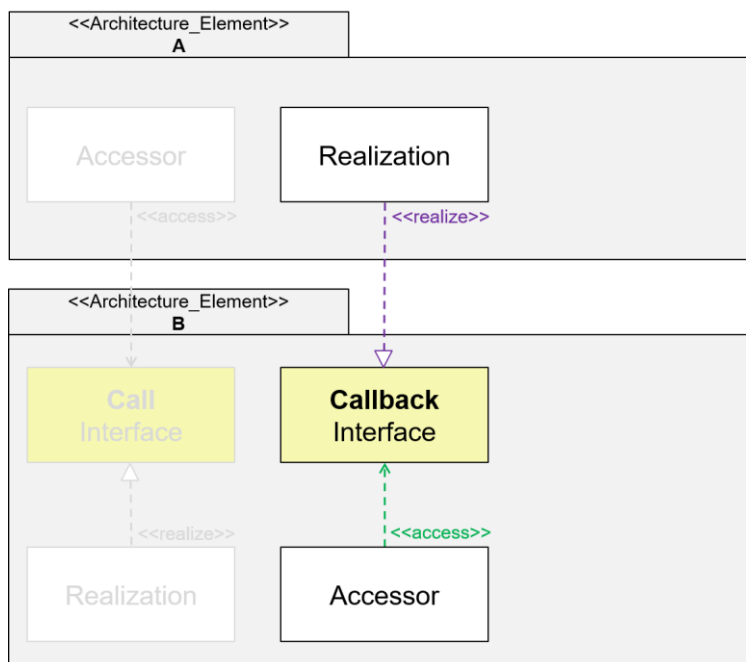
- Call-Interface
- Callback-Interface
- Callback-Registration-Interface



Das **Call-Interface** bietet dem Zugreifer aus dem Architekturelement A beispielsweise Funktionen an, um Werte aus dem Architekturelement B zu lesen, Werte hineinzuschreiben oder dort Algorithmen auszulösen.

**Bild 10:** Call-Interface

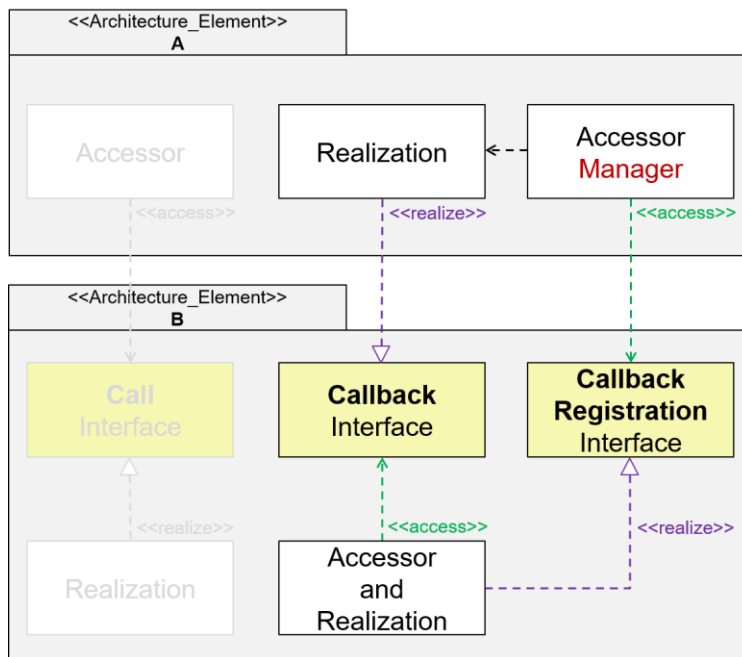
Das **Callback-Interface** meldet aktiv neue Werte oder Ereignisse vom Architekturelement B an A. Bei der Struktur ist zu beachten, dass die Realisierung des



Callback-Interfaces über Architekturelement-Grenzen hinweg geht und sich hier in Architekturelement A befindet. Im Zusammenspiel mit dem Call-Interface ist so in der Summe immer noch eine **unidirektionale Abhängigkeit** zwischen den beiden Architekturelementen A und B erreichbar.

**Bild 11:** Callback-Interface

Falls die Registrierung des Callbacks dynamisch zur Laufzeit und nicht statisch zur Compilezeit durchgeführt wird, ist dafür ein spezielles Element (Manager) in der Software-Architektur verantwortlich. Dieser Manager kann mittels eines speziellen **Callback-Registrierungsinterfaces** ein oder mehrere Callback-implementierende Elemente registrieren und de-registrieren.

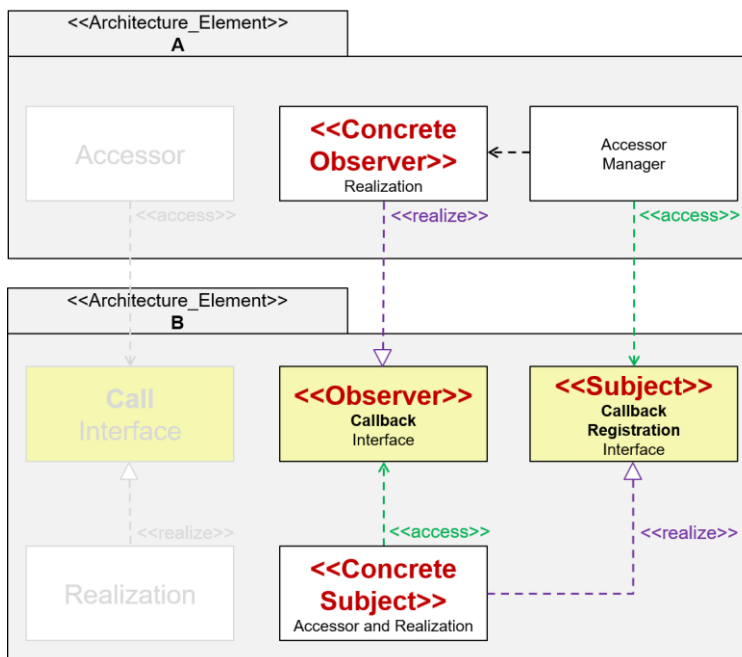


**Bild 12:** Callback-Struktur mit Registrierungsinterface

Zur Umsetzung der in Bild 12 dargestellten Struktur lässt sich das **Observer-Pattern** [6] anwenden.

Das konkrete Subjekt erfährt einen neuen Wert. Durch den Aufruf einer entsprechenden Funktion aus dem Observer meldet das konkrete Subjekt allen zuvor registrierten konkreten Observern den neuen Wert.

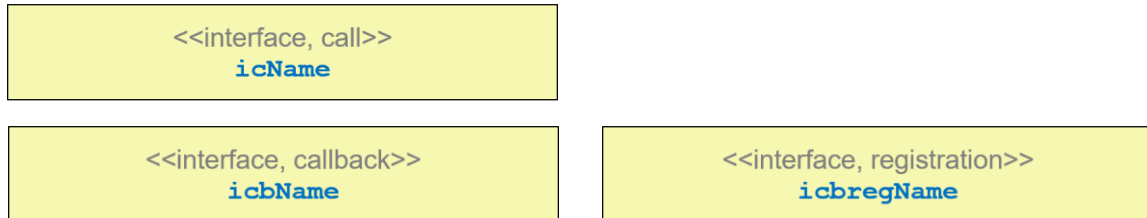
Das Observer-Pattern hat sich inzwischen zu einem sehr populären Pattern für Embedded-Software entwickelt.



**Bild 13:** Callback-Struktur mit Observer-Pattern

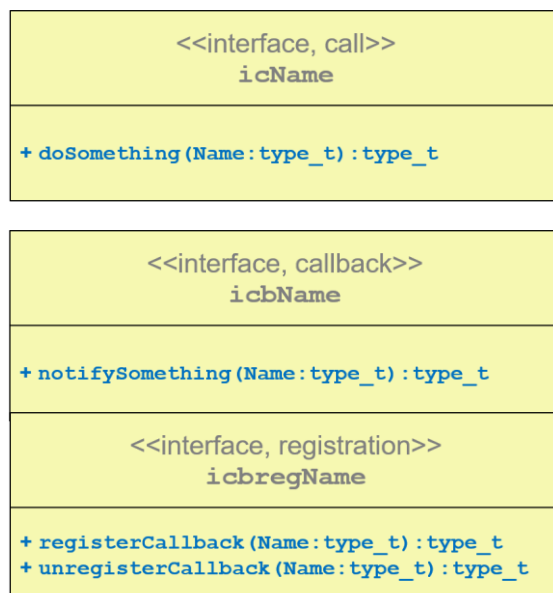
## 5. Semantik Interface

Die Interface-Typisierung spiegelt sich in den vergebenen **<<Stereotypen>>** und in den Interfacenamen als **Prefix** **ic** (interface), **icb** (interface callback) und **icbreg** (interface callback registration) wider. Als **Interfacename** eignet sich ein aussagekräftiges und bedeutungsvolles Substantiv.



**Bild 14:** Semantik Interface

## 6. Semantik Interface-Funktionen



**Bild 15:** Semantik Interface-Funktionen

In allen Interface-Typen sind die Funktionen **öffentlich** (public).

Bei Funktionen sind programmiersprachenabhängige **Modifizierer** wie **virtual**, **const**, **static**, **inline**, ... zu berücksichtigen.

Als **Funktionsnamen** eignen sich Verb-/Substantiv-Kombinationen. Bei Callback-Interface-Funktionen bietet sich als Verb **melde** (notify) an.

Bei Callback-Registrierungsinterface-Funktionen bieten sich die Verben **registriere** (register) und **de-registrierte** (unregister) an.

**Funktionsparameter** sind optional und sollten eine maximale Anzahl von sieben bis zwölf nicht überschreiten. Die Richtung ist durch **in** | **out** | **inout** vor dem Parameternamen visualisierbar. Unterstützt die Programmiersprache (z.B. C++) einen Default-Parameterwert, ist dieser spezifizierbar. Die Reihenfolge der Funktionsparameter kann die Performance beeinflussen. Es sollte mit den Standard-Datentypen begonnen werden, da der Compiler diese per CPU-Register in die Funktion passen kann (sofern noch Register frei sind). Sobald ein komplexer Datentyp, wie beispielsweise eine Struktur, per Wert übergeben wird, passt der Compiler diesen und alle folgenden Parameter per Stack.

Als **Parametername** eignet sich ein Substantiv oder eine Kombination aus mehreren aussagekräftigen und bedeutungsvollen Substantiven.

**Parameter-** und **Returntypen** sind optional `void`. Als Datentypen können entweder Standard-Datentypen aus der Programmiersprache oder bereits definierte eigene Datentypen zum Einsatz kommen; jedoch keine, die erst in der Zukunft definiert werden.

Aus Gründen der Performance ist immer ein `pass by Pointer / Referenz` (keine Kopie!) im Vergleich zu einem `pass by Value` (Kopie!) zu bevorzugen.

Daten-Typenamen sollten durch ein Postfix `_t` gekennzeichnet sein.

Wie bei Funktionen gibt es auch bei Typen programmiersprachenabhängige Modifizierer wie `const`, `static`, `*`, `&`, `[]`, ... .

## 7. Ansätze der Interface-Implementierung

Im einfachsten Fall ist ein Interface ein einfaches Header-File mit einer Summe deklartierter Funktionen. Der Zugreifer inkludiert das Interface-Header-File für den Aufruf der Interface-Funktionen. Ein oder mehrere realisierende Module inkludieren das Header-File zur Implementierung der Interface-Funktionen. Dieser einfachste Implementierungsansatz wird hier nicht weiter betrachtet. Vielmehr soll es um fortschrittlichere Implementierungsoptionen in C++ und, wo dies sinnvoll möglich ist, auch in C gehen.

Die zu Beginn vorgestellten Interface-Designs sind mit den folgenden Implementierungsansätzen umsetzbar:

	Non-Polymorphic Structure	Polymorphic Structure with Dynamic / Late Binding	Polymorphic Structure with Static / Early Binding
<b>Implementation Approach</b>	<ul style="list-style-type: none"> <li>One realization</li> <li>Non-virtual functions</li> </ul>	<ul style="list-style-type: none"> <li>More than one realization</li> <li>Virtual functions</li> </ul>	<ul style="list-style-type: none"> <li>More than one realization</li> <li>Non-virtual functions</li> </ul>
<b>Accessor</b>	<ul style="list-style-type: none"> <li>(Template-) Class(es)</li> </ul>	<ul style="list-style-type: none"> <li>Class(es)</li> </ul>	<ul style="list-style-type: none"> <li>(Template-) class(es)</li> </ul>
<b>Access</b>	<ul style="list-style-type: none"> <li>Pointer(s)</li> <li>Reference(s)</li> <li>Container</li> <li>Embedded instance(s)</li> <li>Template parameter</li> </ul>	<ul style="list-style-type: none"> <li>Pointer(s)</li> <li>Reference(s)</li> <li>Container</li> </ul>	<ul style="list-style-type: none"> <li>Pointers</li> <li>References</li> <li>Container</li> <li>Embedded instances</li> <li>Template parameters</li> </ul>
<b>Interface</b>	<ul style="list-style-type: none"> <li>Class</li> <li>Facade Pattern</li> <li>Template Parameter</li> </ul>	<ul style="list-style-type: none"> <li>Virtual interface class</li> <li>Non-virtual interface class</li> </ul>	<ul style="list-style-type: none"> <li>Class</li> <li>Template parameter</li> </ul>
<b>Realize</b>	<ul style="list-style-type: none"> <li>Inheritance(s)</li> <li>Pointer(s)</li> <li>Reference(s)</li> <li>Embedded object(s)</li> </ul>	<ul style="list-style-type: none"> <li>Inheritances</li> <li>Multiple inheritance (multiple interfaces realized in one class)</li> </ul>	<ul style="list-style-type: none"> <li>(Multiple-) Inheritances</li> <li>Pointers</li> <li>References</li> <li>Embedded instances</li> </ul>
<b>Realization</b>	<ul style="list-style-type: none"> <li>One class</li> </ul>	<ul style="list-style-type: none"> <li>Classes</li> <li>Partly inside the interface class</li> </ul>	<ul style="list-style-type: none"> <li>(Template) classes</li> <li>MixedIn with Curiously Recurring Template Pattern CRTP</li> </ul>

**Bild 16:** Implementierungsansätze

Nicht-**polymorphe Struktur** bedeutet, dass das Interface genau eine Implementierung besitzt, während polymorphe Struktur das mehrfache Vorhandensein von Implementierung meint. Bei polymorphen Strukturen lässt sich die Art der **Bindung** zwischen Objekt / Funktionszeiger und Funktion unterscheiden. **Dynamische** Bindung (Bindung zur Laufzeit) ermöglicht einen vom Kontext abhängigen Aufruf verschiedener



Interface-Funktionsimplementierungen. Bei der **statischen** Bindung (Bindung zur Compilezeit) bindet der Compiler bereits eine zur Laufzeit nicht veränderbare Interface-Funktionsimplementierung.

Die unterschiedlichen Implementierungsansätze lassen sich wie folgt bewerten:

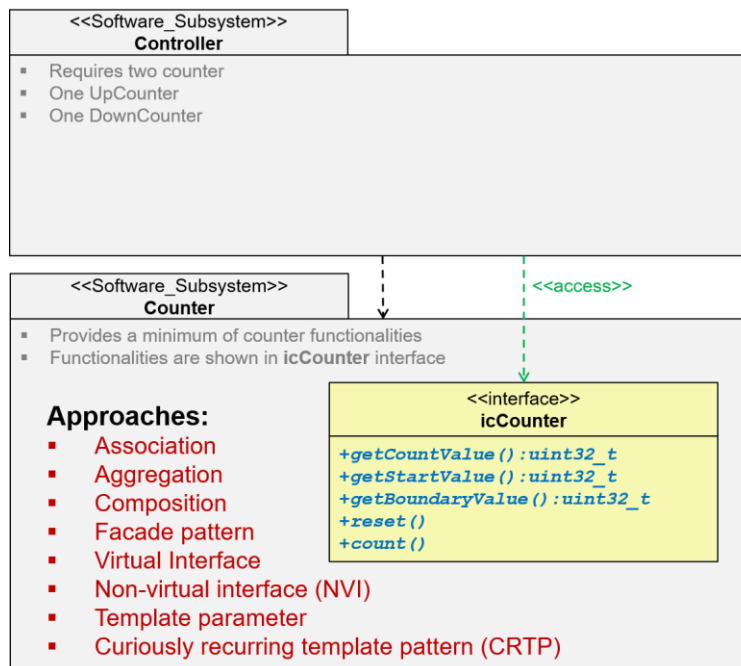
	Non-Polymorphic Structure	Polymorphic Structure with Dynamic / Late Binding	Polymorphic Structure with Static / Early Binding
<b>Advantages</b>	<ul style="list-style-type: none"> <li>Simple</li> <li>Closest coupling</li> </ul>	<ul style="list-style-type: none"> <li>Flexible during run-time</li> <li>Expandable</li> <li>Loos coupling</li> </ul>	<ul style="list-style-type: none"> <li>Resource consumption decreases</li> <li>Performance increases</li> <li>Run-time risk decreases</li> <li>Implementation dependent, no pointer or reference is required → Safety and reliability increases</li> <li>Expandable</li> <li>Closer coupling</li> </ul>
<b>Disadvantage</b>	<ul style="list-style-type: none"> <li>Un-expandable</li> </ul>	<ul style="list-style-type: none"> <li>Resource consumption increases</li> <li>Performance decreases</li> <li>Run-time risk increases</li> <li>Pointer or reference is required → Safety and reliability decreases</li> </ul>	<ul style="list-style-type: none"> <li>Flexibility only during compilation time</li> </ul>
<b>Interface Examples</b> (introduced next)	<ul style="list-style-type: none"> <li>Association</li> <li>Aggregation</li> <li>Composition</li> <li>Facade Pattern</li> </ul>	<ul style="list-style-type: none"> <li>Virtual Interfaces</li> <li>Non-Virtual Interfaces</li> </ul>	<ul style="list-style-type: none"> <li>Template Parameter</li> <li>Curiously Recurring Template Pattern (CRTP)</li> </ul>

**Bild 17:** Bewertung der Implementierungsansätze

Die Auswahl des richtigen Ansatzes hängt sehr stark von den zu erfüllenden Software-Qualitätsanforderungen ab. Ist eine ausgeprägte Flexibilität zur Laufzeit gefordert, so sind polymorphe Strukturen mit dynamischer Bindung die richtige Wahl. Dominiert die funktionale Sicherheit die Flexibilität, so sind nicht-polymorphe Strukturen oder polymorphe Strukturen mit statischer Bindung zu bevorzugen.

## 8. Konkrete Interface-Implementierungsbeispiele

Bezogen auf die im Bild 16 und 17 erläuterten Interface-Implementierungsansätze zeigen die folgenden Ausführungen dazu ein konkretes Interfacebeispiele.



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.

**Bild 18:** Grundlage für konkrete Implementierungsbeispiele

Implementierungsansätze	Programmcode downloadbar in
Assoziation ohne Interface-Klasse	<a href="#">C und C++</a>
Aggregation ohne Interface-Klasse	<a href="#">C und C++</a>
Komposition ohne Interface-Klasse	<a href="#">C und C++</a>
Fassade-Pattern	<a href="#">C und C++</a>
Interfaceklasse mit rein virtuellen Funktionen	<a href="#">C und C++</a>
Interfaceklasse mit rein virtuellen und implementierten Funktionen	<a href="#">C und C++</a>
Interface als Template-Parameter	<a href="#">C++</a>
CRT-Pattern	<a href="#">C++</a>

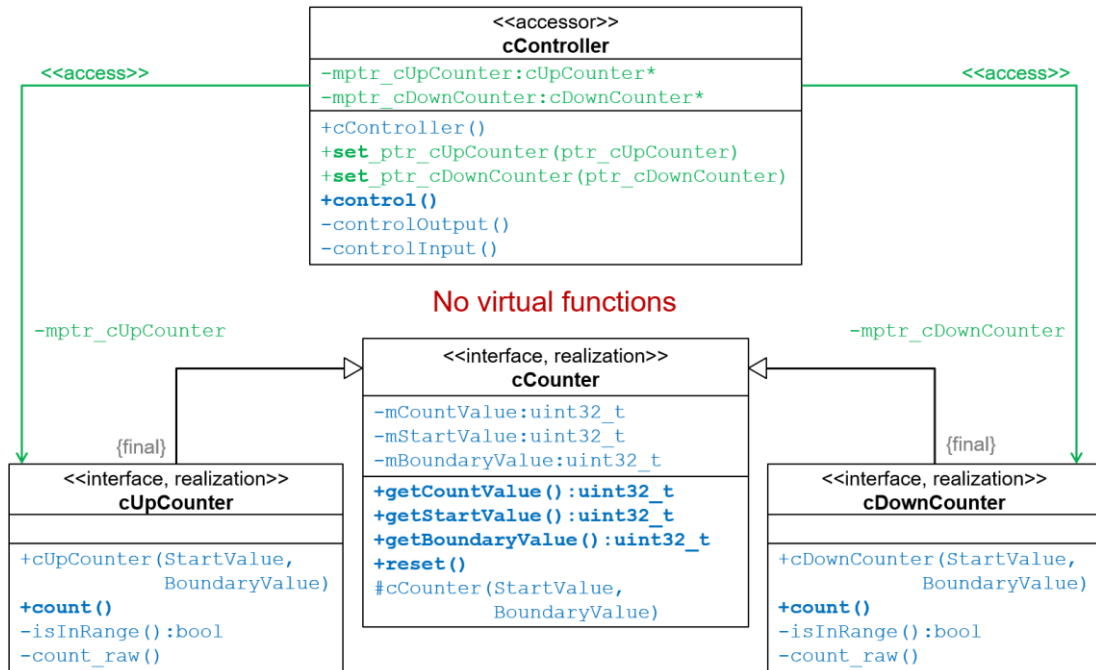
**Bild 19:** Übersicht Implementierungsansätze

Der den Implementierungsbeispielen entsprechende Programmcode ist unter dem Downloadlink [1] 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.

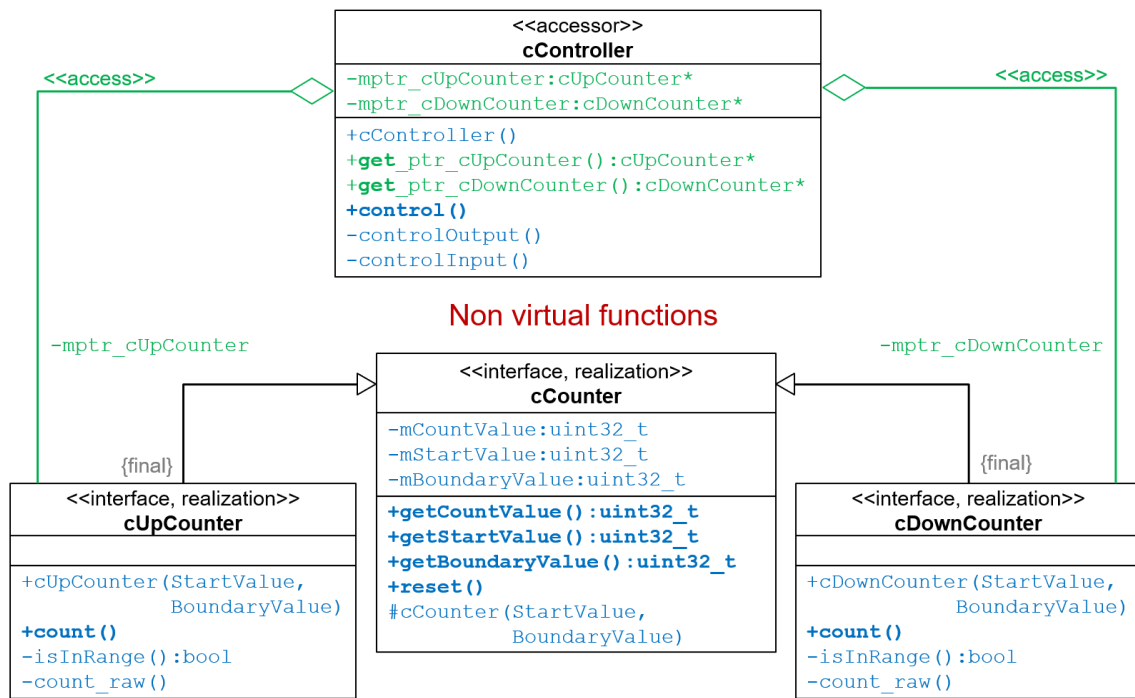


**Bild 20:** Assoziation

- **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.

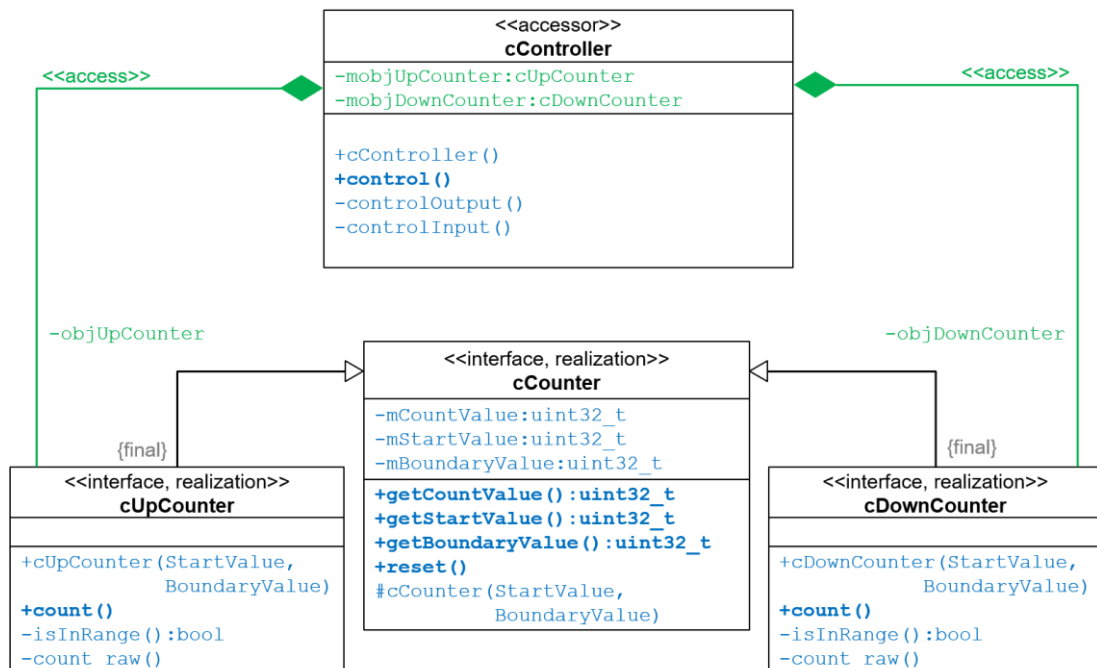
Wie bei der Anwendung der Assoziation ergeben sich auch bei der Aggregation für dieses Beispiel zwei Abhängigkeiten. Ebenfalls wurde gezielt auf die Anwendung von virtuellen Funktionen / Funktionszeiger verzichtet.



**Bild 21:** Aggregation

## ▪ Komposition ohne Interfaceklasse

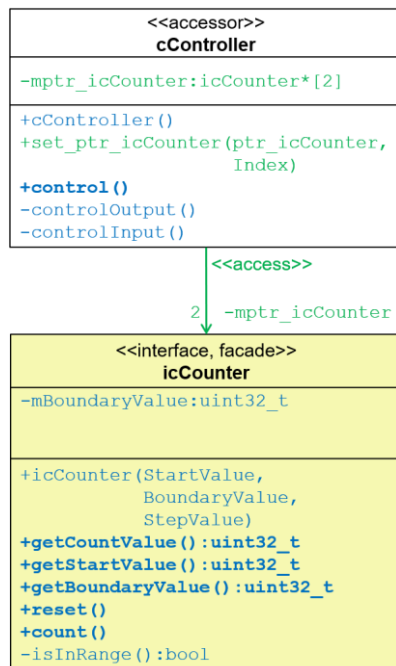
Bei den Varianten mit Assoziation und Aggregation erfolgen die Objektzugriffe jeweils mit Zeigern (optional mit Referenzen). Der komplette Verzicht auf Zeiger führt zur Anwendung der Komposition.



**Bild 22:** 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

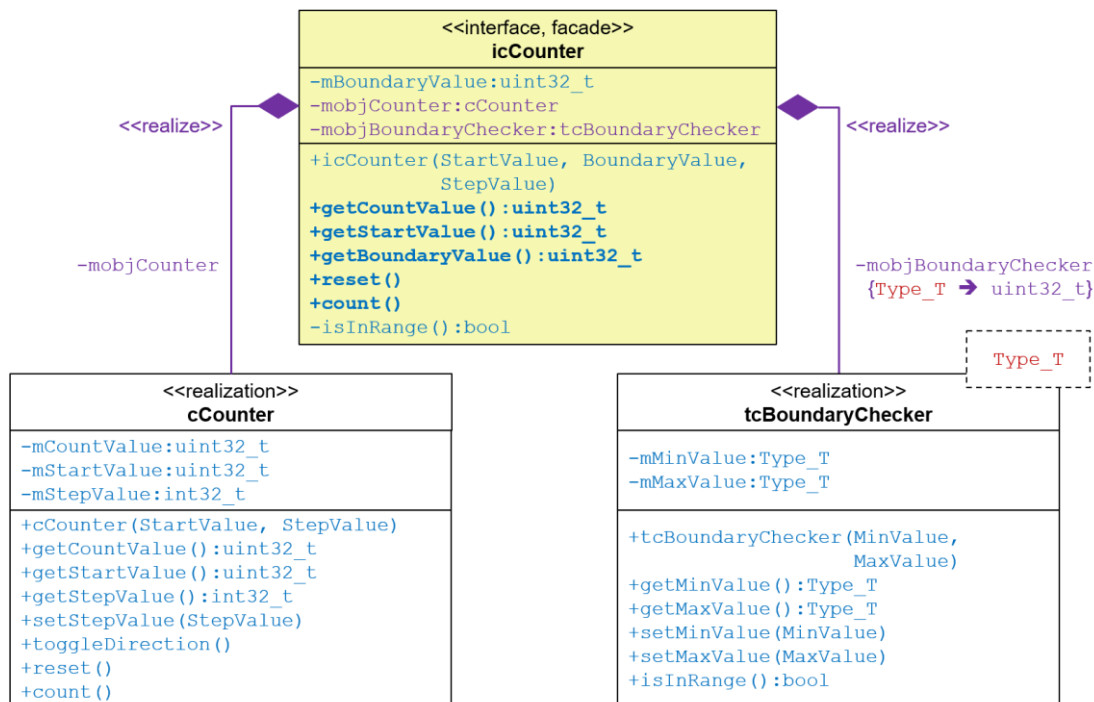


Das Fassade-Pattern [6] bietet dem Zugreifer `cController` das Interface `icCounter` an, welches bereits Daten und Funktionsimplementierungen enthält. Was sich hinter der Fassade `icCounter` verbirgt, ist für den Zugreifer nicht wissenswert und auch nicht sichtbar.

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.

Wie es hinter der Fassade aussieht, zeigt das Bild 24. Das Interface (Fassade) enthält ein Zähler- und ein Grenzwert-Prüfobjekt, die gemeinsam die komplette Funktionalität der Fassade realisieren.

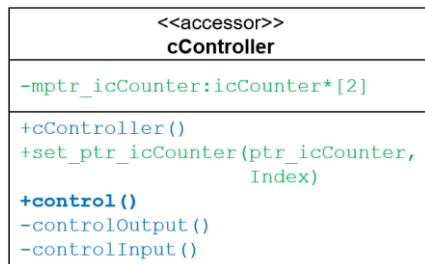
**Bild 23:** Fassade-Pattern – Interfacezugriff



**Bild 24:** Fassade-Pattern – Interfacerealisierung

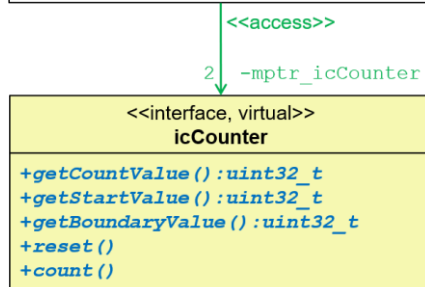
## ▪ Interfaceklasse mit rein virtuellen Funktionen

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.

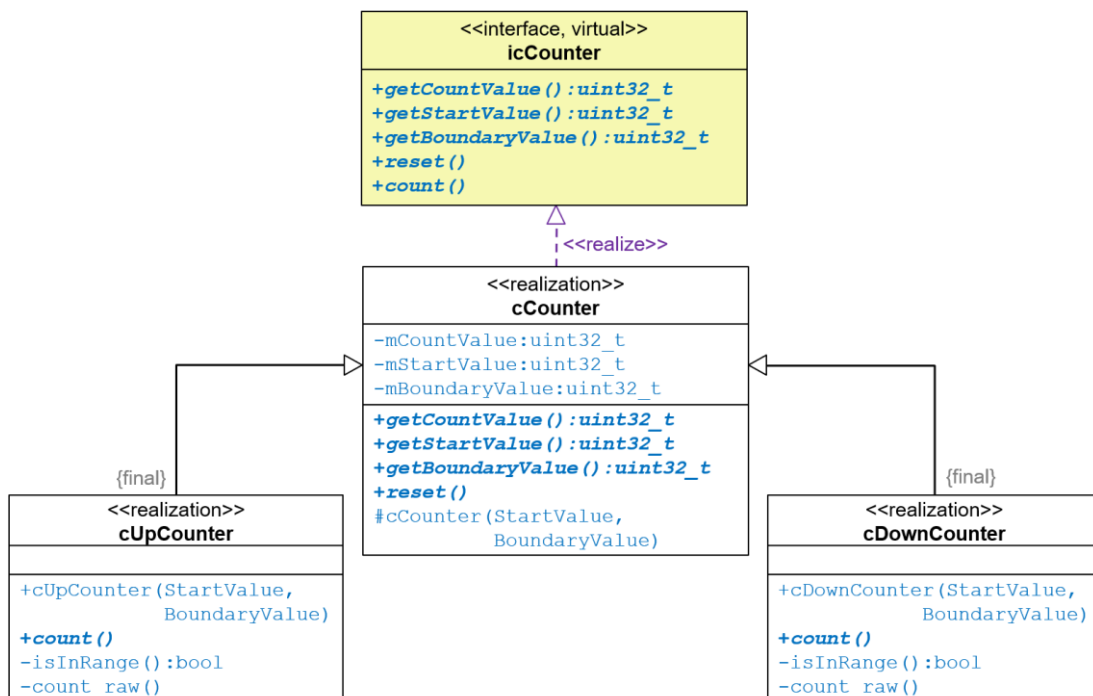
Der Zugriff auf das Interface erfolgt im cController durch einen Zeiger / eine Referenz vom Typ des Interfaces. Dieser Zeiger / diese Referenz muss später auf ein Objekt der Interface-realisierten Klassen zeigen.



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

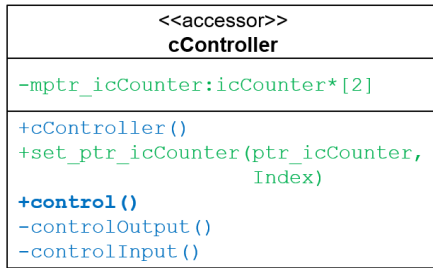
**Bild 25:** Virtual Interface – Interfacezugriff

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.



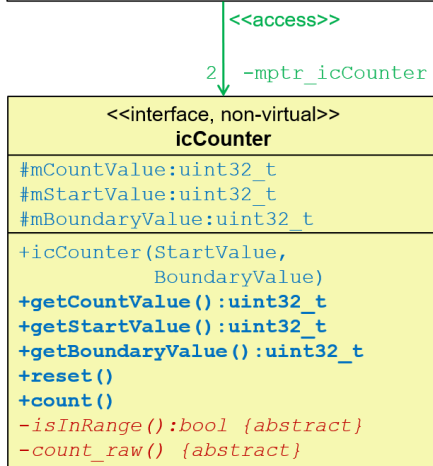
**Bild 26:** Virtual Interface – Interfacerealisierung

- **Interfaceklasse mit rein virtuellen und implementierten Funktionen**



Diese Implementierungsvariante basiert auf dem C++ Idiom Non-Virtual Interface (NVI) [4].

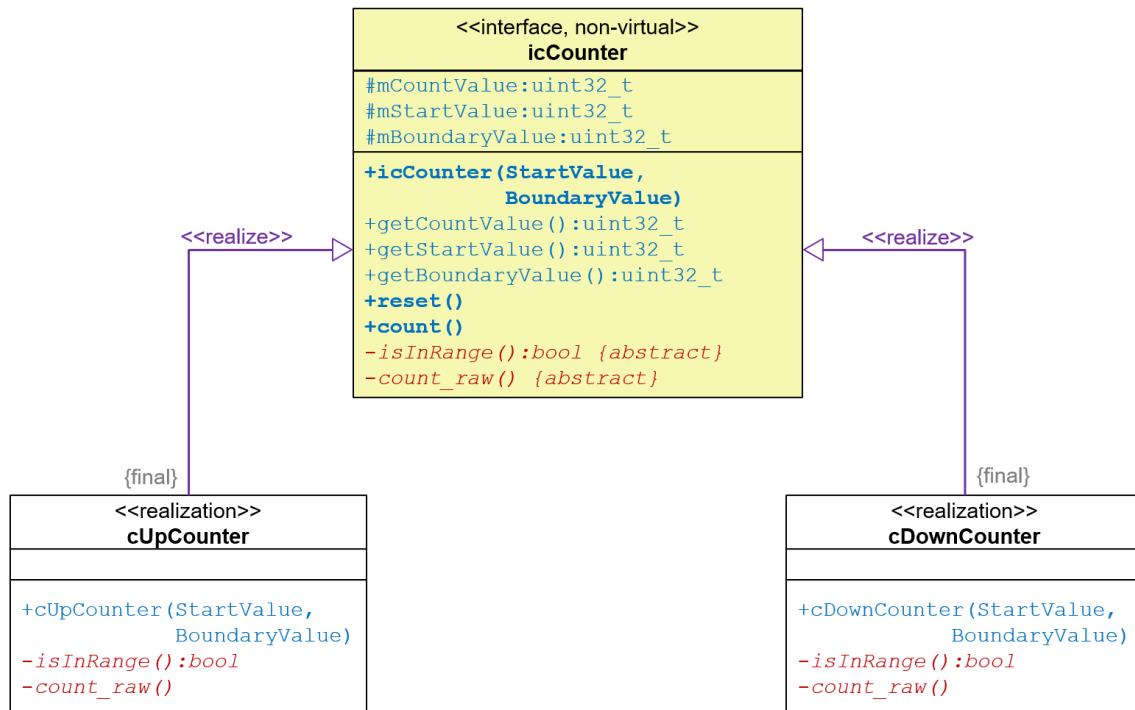
Bei der Implementierung von rein virtuellen Interfaces ergeben sich im Falle mehrerer Implementierungen typischerweise redundante Programmcode-Anteile.



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.

**Bild 27:** Non-Virtual Interface – Interface Zugriff

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



**Bild 28:** Non-Virtual Interface – Interface Realisierung

In den Beispielen mit virtuellen Funktionen in Interfaces oder / und Klassen führt das Ergebnis auf dem Target zu einer dynamischen Bindung. Dadurch entstehen verschieden

Aufwände, die aber im Vergleich zum Nutzen in dem meisten Fällen vernachlässigbar sind:

a) **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.

b) **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.

c) **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).

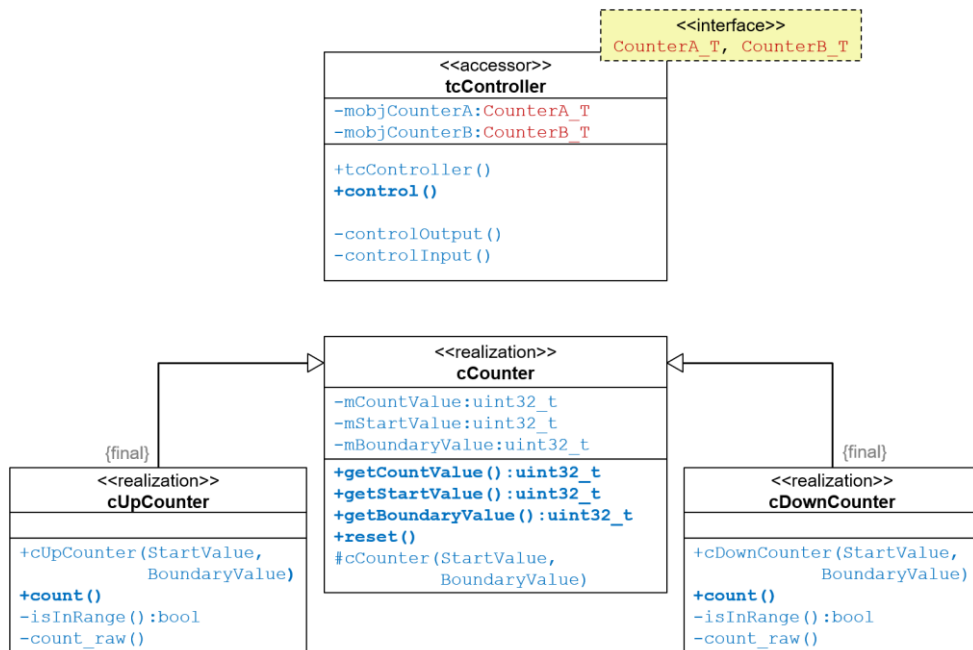
Dies ist die Funktionalität der dynamischen Bindung. Sie erlaubt so die Programmierung der **dynamischen Polymorphie**. Diesen Mechanismus führt die C++ Toolkette automatisch aus. In C ist die dynamische Bindung mit etwas mehr Aufwand manuell nachbildbar.

▪ **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!).

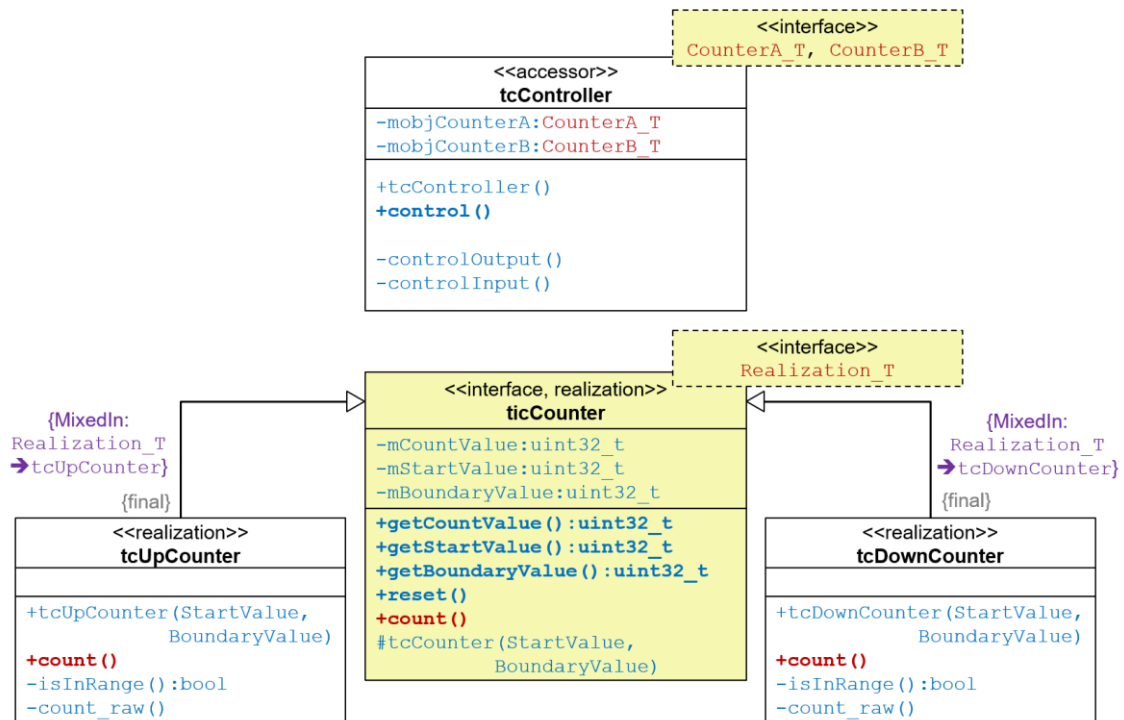




**Bild 29:** Interface als Template-Parameter

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 30:** 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.

## 9. 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, wie beispielsweise 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.

## 10. Referenzierte und weiterführende Links

- [1] MicroConsult-Download für diesen Beitrag - komplett und aktuell  
<http://download.microconsult.net/ese2020/interface-designs.zip>
- [2] Object Management Group (OMG) - Unified Modeling Language (UML) Standard  
[www.uml.org](http://www.uml.org)
- [3] Wikipedia: Curiously recurring template pattern (CRTP)  
[https://en.wikipedia.org/wiki/Curiously\\_recurring\\_template\\_pattern](https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern)

- [4] Wikibooks: More C++ Idioms – Non-Virtual Interface (NVI)  
[https://en.wikibooks.org/wiki/More\\_C%2B%2B\\_Idioms/Non-Virtual\\_Interface](https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface)
- [5] **MicroConsult-Trainings – auch Live-Online:**  
  
[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](#)  
  
[www.microconsult.de](http://www.microconsult.de)
- [6] Fachliteratur:  
**Design Patterns**  
Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides  
Prentice Hall Verlag  
ISBN-10: 0201633612  
ISBN-13: 978-0201633610  
<http://www.prenticehall.com>

## 11. Autor:



Dipl.-Ing. (FH) Thomas Batt ist gebürtiger Freiburger. Nach seiner Ausbildung als Radio- und Fernsehtechniker studierte er Nachrichtentechnik in Offenburg. Seit 1994 arbeitet er kontinuierlich in verschiedenen Branchen und Rollen im Bereich Embedded-/Real-Time Systementwicklung. 1999 wechselte Thomas Batt zur MicroConsult GmbH. Dort verantwortet er heute als zertifizierter Trainer und Coach die Themenbereiche Systems- /Software Engineering für Embedded-/Real-Time-Systeme sowie Entwicklungsprozess-Beratung.

## Kontakt:

[t.batt@microconsult.de](mailto:t.batt@microconsult.de)  
[www.microconsult.de](http://www.microconsult.de)