MicroConsult

# Interface-Designs und ihre Implementierungen – Teil 1: Struktur und Definition

**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 Sie beim Interface-Design kennen sollten und wie diese in den Programmiersprachen C und C++ implementierbar sind, verrät dieser Beitrag. Der erste Teil stellt Interface-Konzepte und unterschiedliche Interface-Typen vor.

## Das Interface-Konzept und Designvarianten

Ein Software-Interface stellt eine Summe von **Funktionen** mit der kompletten Semantik (Name, Parameter, Parametertypen, Rückgabetypen, spezielle Modifizierer) für den Zugriff und die Realisierung bereit.

Mindestens ein Element (Accessor) greift auf das Interface zu. Der **Zugreifer** erwartet das Interface (**Required** Interface).

Mindestens ein Element (Realization) muss das Interface implementieren. Die **Realisierung** stellt das Interface bereit (**Provided** Interface).

Das Interface dient zur Entkopplung zwischen dem Zugreifer und der Realisierung, weiter übergeordnet zur Entkopplung zwischen Architekturelementen.

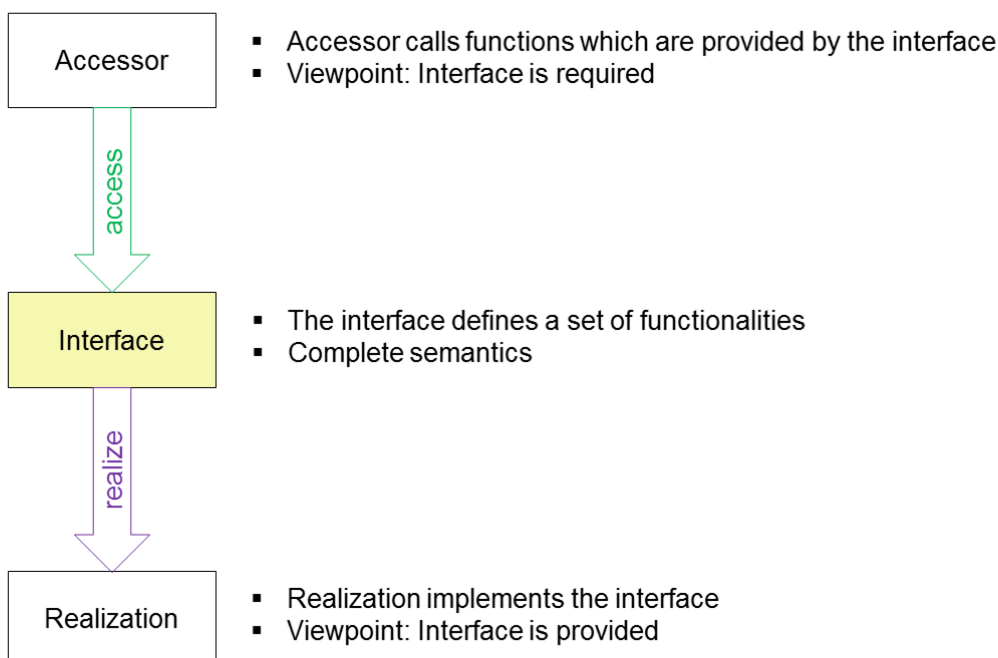


*Bild 1: Interface-Konzept*

©MicroConsult 01-21

Bei verschiedene Designvarianten zwischen Zugreifer und Interface(es)

- greift ein Zugreifer auf ein oder mehrere Interfaces zu
- greifen mehrere Zugreifer auf ein oder *jeweils* ein Interface zu
- greifen mehrere Zugreifer auf ein oder mehrere Interfaces zu
- greifen mehrere Zugreifer auf unterschiedliche Interface-Ebenen zu oder
- greifen mehrere Zugreifer auf *jeweils* unterschiedliche Interface-Ebenen zu

Bei den Designvarianten der Interface-Realisierung sind möglich:

- ein Interface mit einer Realisierung
- ein Interface mit mehreren oder partiellen Realisierungen
- mehrere Interfaces in einer oder mehreren Realisierungsvarianten

**Interface-Typisierung**

Interfaces lassen sich (abhängig von verschiedenen Zugreifern) thematisch aufteilen, beispielsweise pro Architekturelement jeweils eines für die Konfiguration, die Diagnose und den Normalbetrieb. Generell lassen sich Interfaces in drei unterschiedliche Typen kategorisieren:

- 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.
- 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.
- 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.

Zur Umsetzung der in Bild 2 dargestellten Struktur lässt sich das **Observer-Pattern** 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.

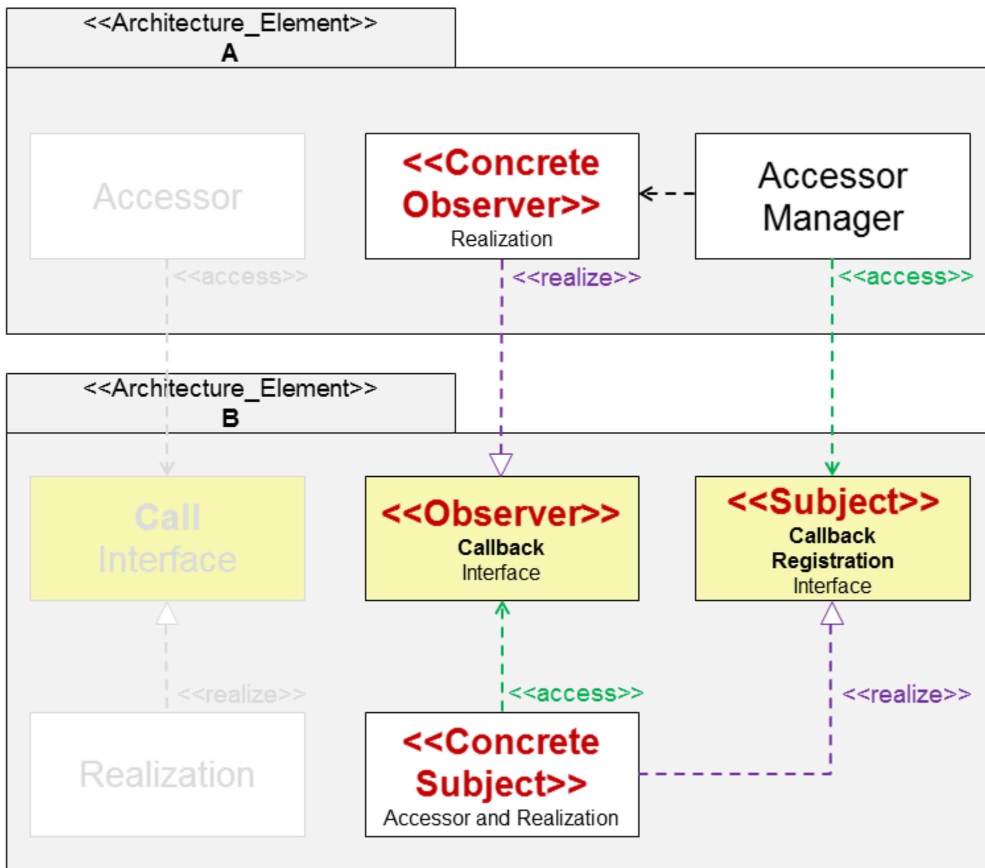


*Bild 2: Callback-Struktur mit Registrierungsinterface*

**Das Semantik-Interface und die Funktionen**

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.

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

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

Als **Funktionsnamen** eigenen 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-registriere (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 übergeben kann (sofern noch Register frei sind).

©MicroConsult 01-21

Sobald ein komplexer Datentyp, wie beispielsweise eine Struktur, per Wert übergeben wird, reicht der Compiler diesen und alle folgenden Parameter per Stack in die Funktion.

Als **Parametername** eignen 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 eine Übergabe als Zeiger / Referenz (keine Kopie!) gegenüber einer Übergabe als Wert (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, *,  &,[], … .

## Ansätze der Interface-Implementierung

Im einfachsten Fall ist ein Interface ein Header-File mit einer Summe deklarierter 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 |
|---|---|---|---|
| Implementation Approach | ▪ One realization<br>▪ Non-virtual functions | ▪ More than one realization<br>▪ Virtual functions | ▪ More than one realization<br>▪ Non-virtual functions |
| Accessor | ▪ (Template-) Class(es) | ▪ Class(es) | ▪ (Template-) class(es) |
| Access | ▪ Pointer(s)<br>▪ Reference(s)<br>▪ Container<br>▪ Embedded instance(s)<br>▪ Template parameter | ▪ Pointer(s)<br>▪ Reference(s)<br>▪ Container | ▪ Pointers<br>▪ References<br>▪ Container<br>▪ Embedded instances<br>▪ Template parameters |
| Interface | ▪ Class<br>▪ Facade Pattern<br>▪ Template Parameter | ▪ Virtual interface class<br>▪ Non-virtual interface class | ▪ Class<br>▪ Template parameter |
| Realize | ▪ Inheritance(s)<br>▪ Pointer(s)<br>▪ Reference(s)<br>▪ Embedded object(s) | ▪ Inheritances<br>▪ Multiple inheritance (multiple interfaces realized in one class) | ▪ (Multiple-) Inheritances<br>▪ Pointers<br>▪ References<br>▪ Embedded instances |
| Realization | ▪ One class | ▪ Classes<br>▪ Partly inside the interface class | ▪ (Template) classes<br>▪ MixedIn with Curiously Recurring Template Pattern CRTP |

*Bild 3: 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 |
|---|---|---|---|
| **Advantages** | ▪ Simple<br>▪ Closest coupling | ▪ Flexible during run-time<br>▪ Expandable<br>▪ Loos coupling | ▪ Resource consumption decreases<br>▪ Performance increases<br>▪ Run-time risk decreases<br>▪ Implementation dependent, no pointer or reference is required<br>➔ Safety and reliability increases<br>▪ Expandable<br>▪ Closer coupling |
| **Disadvantage** | ▪ Un-expandable | ▪ Resource consumption increases<br>▪ Performance decreases<br>▪ Run-time risk increases<br>▪ Pointer or reference is required<br>➔ Safety and reliability decreases | ▪ Flexibility only during compilation time |
| **Interface Examples** (introduced next) | ▪ Association<br>▪ Aggregation<br>▪ Composition<br><br>▪ Facade Pattern | ▪ Virtual Interfaces<br>▪ Non-Virtual Interfaces | ▪ Template Parameter<br>▪ Curiously Recurring Template Pattern (CRTP) |

*Bild 4: 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.

Der zweite Teil des Beitrags zeigt Implementierungsansätze durch Assoziation, Komposition, Fassade, virtuelle Interfaces, nicht-virtuelle Interfaces, C++ Templates und CRTP-Pattern auf.

**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.**

©MicroConsult 01-21

## 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.

©MicroConsult 01-21