

Port-Designs und ihre Implementierungsansätze

Struktur, Definition, Realisierung, Verbindung und Zugriff

Thomas Batt, MicroConsult GmbH

0. Vorwort

Hier knüpfe ich an meinen ESE-Beitrag von 2020 „Interface-Designs und ihre Implementierungen“ [2] an. Die konsequente Erweiterung des Interface-Konzepts ist die Ergänzung um Ports. Insbesondere bei immer komplexer werdenden Embedded-Software-Architekturen bieten sich Ports zur logischen Gruppierung von bereitgestellten (provided) und erwarteten (required) Software-Interfaces an.

Welche Varianten der Architekt beim Port-Design kennen sollte und wie das Port-Konzept mit der Programmiersprache C++ implementierbar ist, zeigt dieser Beitrag [1]. Um diesen Beitrag vollständig zu verstehen, sind Kenntnisse der UML [3] sowie der objektorientierten Programmierung mit der Programmiersprachen C++ vorausgesetzt.

1. Vom Interface-Konzept zu Ports

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.

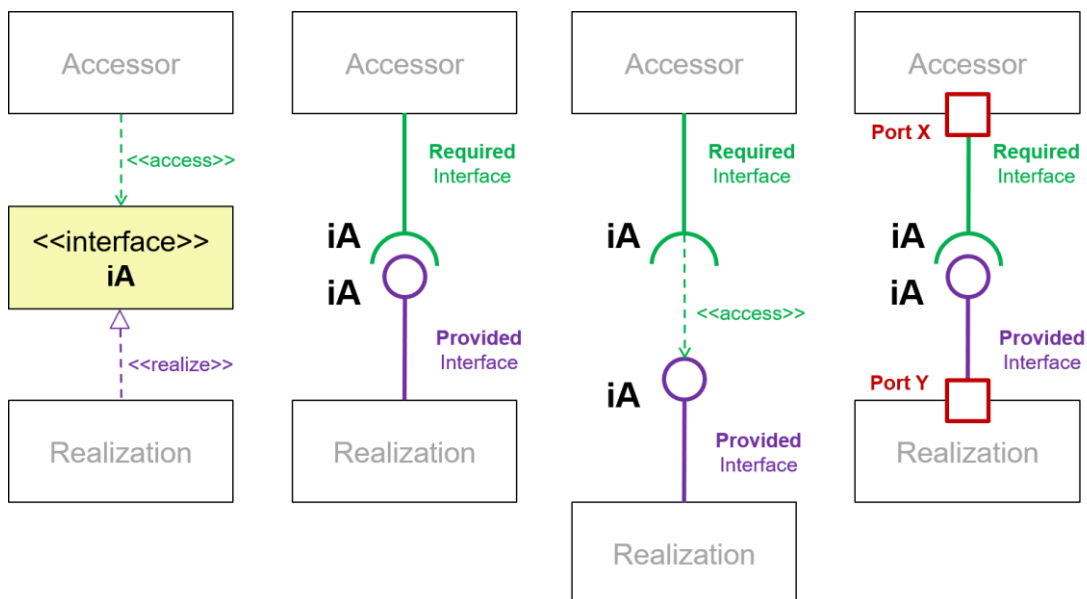


Bild 1: Verschiedene Darstellungsformen von Interfaces ohne und mit Port

Interfaces gruppieren Funktionen logisch. Eine Ebene darüber **gruppieren Ports Interfaces logisch**. Dabei umfasst ein **Port** null bis theoretisch unendlich viele **bereitgestellte** und/oder null bis unendlich viele **erwartete Interfaces**. Sind die bereitgestellten und erwarteten Interfaces zweier **Ports** identisch, können diese miteinander **verbunden** werden. Eine hierarchische Organisation von Ports führt zu verschachtelten (nested) Ports.

Basierend auf der UML [3] lassen sich **Klassen** und **Software-Komponenten** mit null bis theoretisch unendlich vielen Ports ausstatten, jedoch nicht Pakete.

2. Designvarianten von Ports

Der folgende Teil zeigt verschiedene Designvarianten bei der Anwendung von Ports. Dabei repräsentieren Accessor bzw. Realisation entweder Klassen oder Software-Komponenten.



Bild 2: Unidirektionaler Port mit einem und mehreren Interfaces

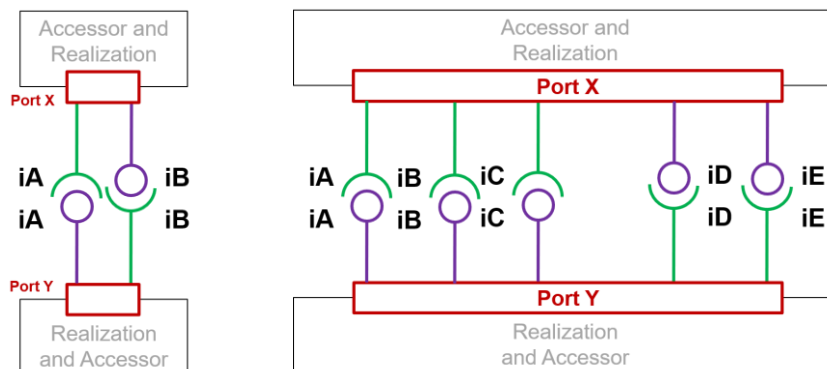


Bild 3: Bidirektionaler Port mit mehreren Interfaces

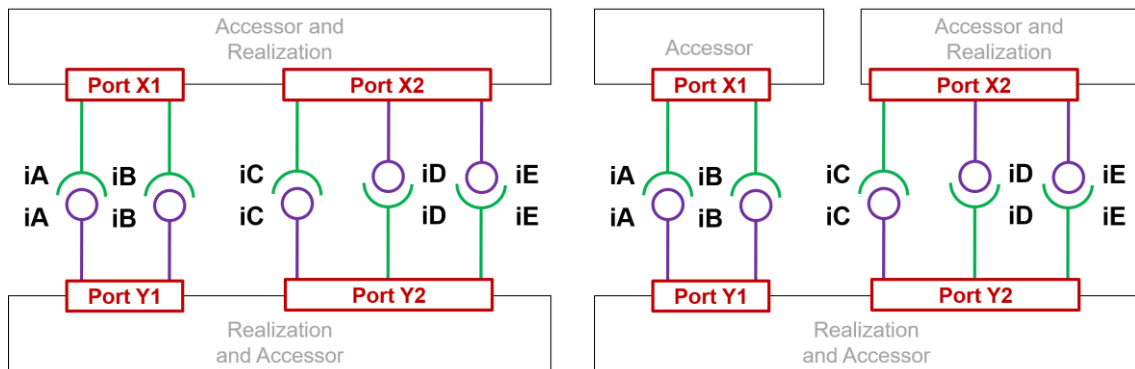


Bild 4: Mehrere Ports, einfach und mehrfach endend

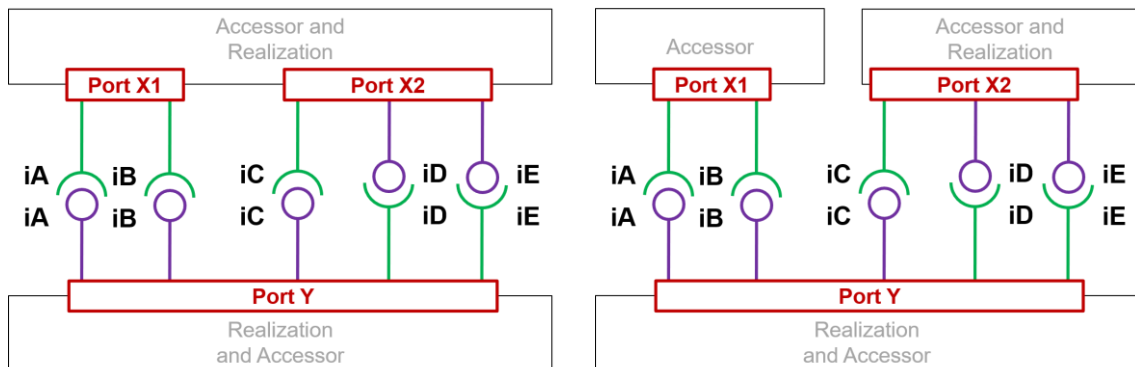


Bild 5: Gesplitteter Port, einfach und mehrfach endend

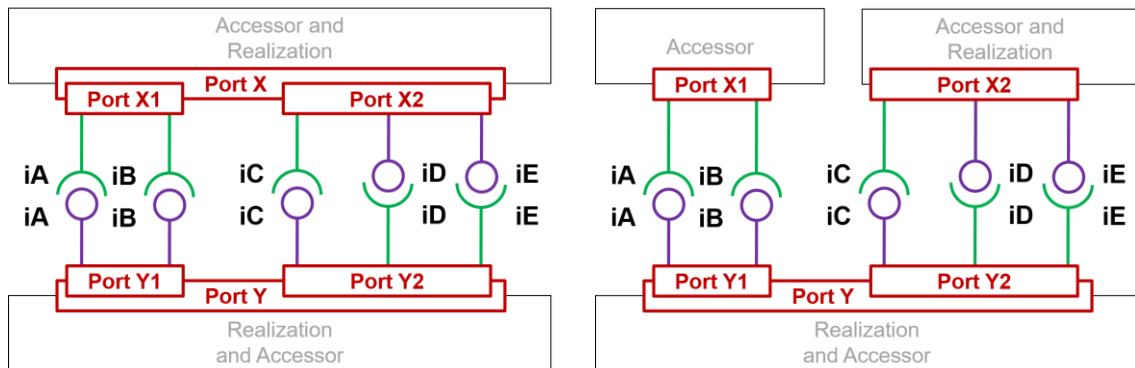


Bild 6: Verschachtelte Ports, einfach und mehrfach endend

Zwei oder mehr Objekte der entsprechenden Applikationsklassen werden über einen oder mehrere Ports miteinander verbunden.

3. Implementierungsansätze mit C++

Provided und required Interfaces bilden sich in nur einer Interface-Klasse ab. Die Interface-Klasse enthält alle Interface-Funktionen mit deren kompletter Semantik als rein virtuelle Funktionen. Alle provided Interfaces eines Ports werden in der InBound-Klasse und alle required Interfaces in der OutBound-Klasse zur Realisierung und für den Zugriff zusammengeführt. Somit enthält jede Portklasse eine Instanz von mindestens einer Bound-Klasse. Die Zeigerinitialisierung zwischen Ports bzw. derer Bound-Klassen verbindet zwei Instanzen von Applikationsklassen (Accessor / Realization). Diese können nun über die Interface-Funktionen interagieren.

Element	Implementierungsansätze mit C++
Interface	Interface-Klasse mit rein virtuellen Funktionen
Provided Interface(es)	Realisierung und Zugriff über InBound-Klasse
Required Interface(es)	Realisierung und Zugriff über OutBound-Klasse
Port	Portklasse mit eingebetteten Objekten der InBound- und/oder OutBound-Klasse
Nested Ports	Instanz der nested Port-Klasse in der Port Klasse
Accessor und Realization	Applikationsklassen , die eine Instanz ihrer Port-Klasse(n) und damit auch derer Bound-Klassen enthalten
Portverbindung	Zeiger , die die Instanzen der Bound-Klassen miteinander verbinden

Tabelle 1: Übersicht der Implementierungsansätze

Diese Implementierungsansätze lassen sich am besten an einem konkreten Implementierungsbeispiel erläutern.

4. Implementierungsbeispiel 1: Klasse A und B zur Basiskonzeptvorstellung

Die Funktion `execute()` der Applikationsklasse `cA` soll die `function()` der Applikationsklassen `cB` aufrufen. Dazu stellt die Klasse `cB` über ihren Port `pcB` das Interface `icX` bereit (provided Interface). Auf der anderen Seite erwartet die Klasse `cA` über ihren Port `pcA` das gleiche Interface `icX` (required Interface).

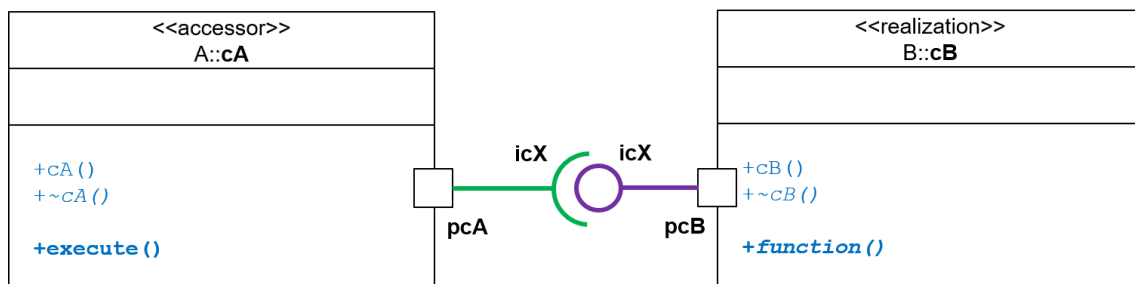


Bild 7: Implementierungsbeispiel 1

Auf Basis der in Tabelle 1 vorgestellten Implementierungsansätze ergibt sich für den Implementierungspfad der Klasse cA die folgende detaillierte Designsicht:

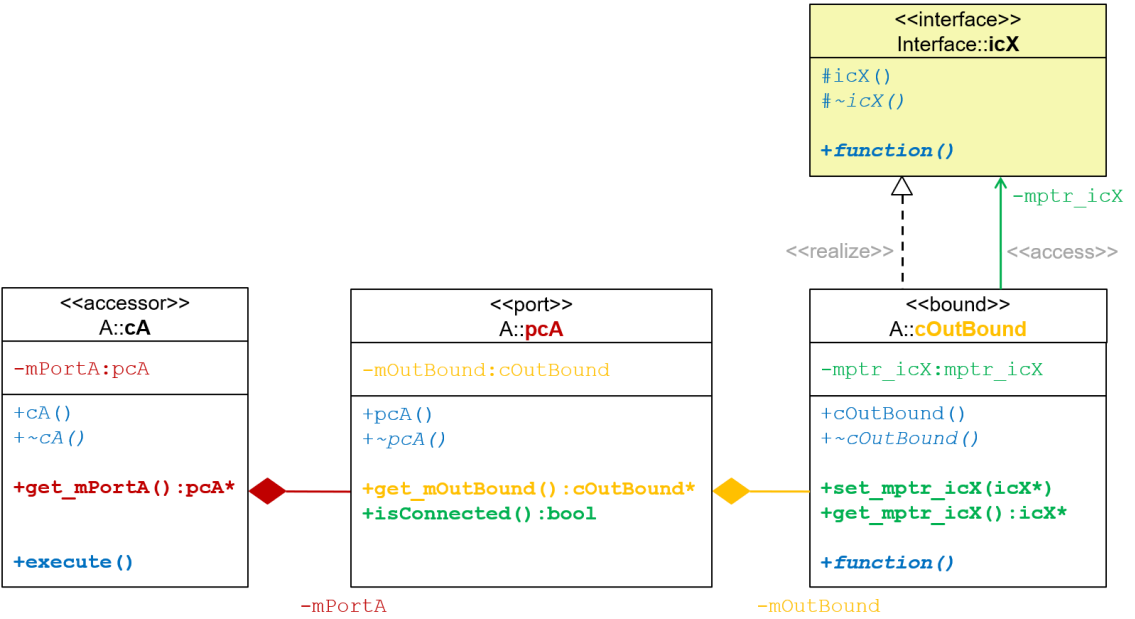


Bild 8: Implementierungspfad Klasse cA

Die `cA::execute()` Funktion ruft über das in der Klasse cA eingebettete Objekt `mPortA` der Klasse `pcA` über dessen eingebettetes Objekt `mOutBound` der Klasse `cOutBound` die Funktion `cOutBound::function()` auf. Die Implementierung von `cOutBound::function()` ruft über den Interfacezeiger `mptr_icX` die `function()` des dahinterliegenden Objektes (Objekt der Klasse cB) auf.

Der folgende Abschnitt zeigt die detaillierte Beschreibung der Daten und Funktionen der einzelnen Klassen aus **Pfad A**.

Interface-Klasse: icX	
Funktionen	Beschreibung
icX()	C++ Default-Implementierung
~icX()	C++ Default-Implementierung
function()	Rein virtuelle Interface-Funktion (hier noch ohne Implementierung)

Tabelle 2: Interface-Klasse icX: Beschreibung

Bound-Klasse: cOutBound	
Daten	Beschreibung
<code>mPtr_icX</code>	Zeigt auf ein Objekt einer Klasse, die die Interface-Funktion <code>icX::function()</code> realisiert
Funktionen	Beschreibung
<code>cOutBound()</code>	Initialisiert <code>mPtr_icX</code> mit <code>nullptr</code>
<code>~cOutBound()</code>	C++ Default-Implementierung
<code>set_mPtr_icX()</code>	Setzt <code>mPtr_icX</code> gemäß dem Parameter
<code>get_mPtr_icX()</code>	Liest die Objektadresse hinter <code>mPtr_icX</code>
<code>function()</code>	Implementierung von <code>icX::function()</code> , die über <code>mPtr_icX</code> wiederum <code>function()</code> aufruft

Tabelle 3: Bound-Klasse `cOutBound`: Beschreibung

Portklasse: pcA	
Daten	Beschreibung
<code>mOutBound</code>	Instanz der Bound-Klasse <code>cOutBound</code> , um auf die Interface-Implementierung <code>cOutBound::function()</code> zuzugreifen
Funktionen	Beschreibung
<code>pcA()</code>	C++ Default-Implementierung
<code>~pcA()</code>	C++ Default-Implementierung
<code>get_mOutBound()</code>	Liest die Objektadresse von <code>mOutBound</code>
<code>isConnected()</code>	Ermittelt, ob dieser Port mit einem anderen verbunden ist

Tabelle 4: Portklasse `pcA`: Beschreibung

Applikationsklasse: cA	
Daten	Beschreibung
<code>mPortA</code>	Instanz der Portklasse <code>pcA</code> , um über deren <code>cOutBound</code> Instanz final auf die Implementierung der Interface-Funktion <code>function()</code> in <code>cB</code> zuzugreifen
Funktionen	Beschreibung
<code>cA()</code>	C++ Default-Implementierung
<code>~cA()</code>	C++ Default-Implementierung
<code>get_mPortA()</code>	Liest die Objektadresse von <code>mPortA</code>
<code>execute()</code>	Ruft bei vorhandener Portverbindung <code>cOutBound::function()</code> und damit die finale Interface-Implementierung <code>cB::function()</code> auf

Tabelle 5: Applikationsklasse `cA`: Beschreibung

Auf Basis der in Tabelle 1 vorgestellten Implementierungsansätze ergibt sich für den Implementierungspfad der Klasse `cB` die folgende detaillierte Designsicht:

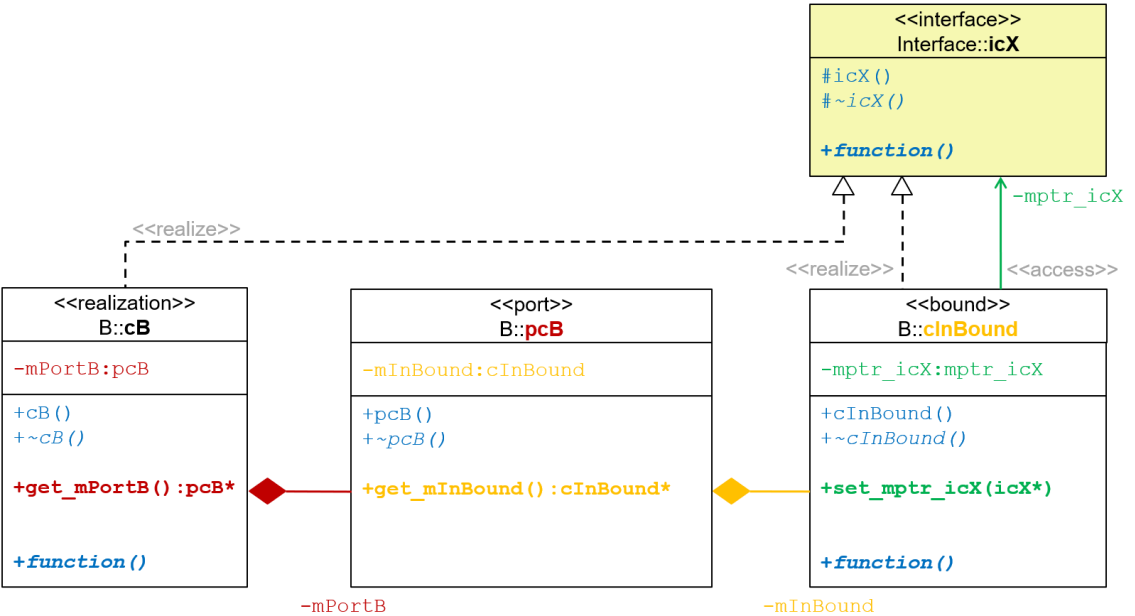


Bild 9: Implementierungspfad Klasse `cB`

Die Klasse `cInBound` realisiert die Interface-Funktion `icX::function()`. Diese Realisierung ruft über den in der Klasse `cInBound` enthaltenen Zeiger `mptr_icX` die dahinter über die Objektadresse initialisierte `cB::function()` auf. Daher ergeben sich hier im Pfad B zwei Interface-Realisierungen – eine in der Klasse `cInBound` und die andere in der Klasse `cB`.

Der folgende Abschnitt zeigt die detaillierte Beschreibung der Daten und Funktionen der einzelnen Klassen aus **Pfad B**.

Bound-Klasse: <code>cInBound</code>	
Daten	Beschreibung
<code>mptr_icX</code>	Zeigt auf ein Objekt der Klasse <code>cB</code> , die die Interface-Funktion <code>icX::function()</code> realisiert
Funktionen	Beschreibung
<code>cInBound()</code>	Initialisiert <code>mptr_icX</code> mit <code>nullptr</code>
<code>~cInBound()</code>	C++ Default-Implementierung
<code>set_mptr_icX()</code>	Setzt <code>mptr_icX</code> gemäß dem Parameter auf ein Objekt der Klasse <code>cB</code>
<code>function()</code>	Implementierung von <code>icX::function()</code> , die über <code>mptr_icX</code> wiederum <code>function()</code> aufruft

Tabelle 6: Bound-Klasse `cInBound`: Beschreibung

Portklasse: pcB	
Daten	Beschreibung
<code>mInBound</code>	Instanz der Bound-Klasse <code>cInBound</code> , um auf die Interface-Implementierung <code>cInBound::function()</code> zuzugreifen
Funktionen	Beschreibung
<code>pcA()</code>	C++ Default-Implementierung
<code>~pcA()</code>	C++ Default-Implementierung
<code>get_mInBound()</code>	Liest die Objektadresse von <code>mInBound</code>

Tabelle 7: Portklasse `pcB`: Beschreibung

Applikationsklasse: cB	
Daten	Beschreibung
<code>mPortB</code>	Instanz der Portklasse <code>pcB</code> , um über deren <code>cInBound</code> Instanz final auf die Implementierung der Interface-Funktion <code>function()</code> in <code>cB</code> zuzugreifen
Funktionen	Beschreibung
<code>cB</code>	Setzt den Zeiger <code>mPtr_icX</code> auf ein Objekt (<code>this</code>) der Interface-implementierenden Klasse <code>cB</code>
<code>~cB</code>	C++ Default-Implementierung
<code>get_mPortB</code>	Liest die Objektadresse von <code>mPortB</code>
<code>function</code>	Finale Implementierung der Interface-Funktion <code>function()</code> , die ihren Aufruf signalisiert

Tabelle 8: Applikationsklasse `cB`: Beschreibung

Zur Demonstration der Applikation sind die folgenden weiteren Schritte notwendig:

1. Die **Klassen** `cA` und `cB` zu Objekten `locA` und `locB` **instanciieren**:

cA `A::locA;`

cB `B::locB;`

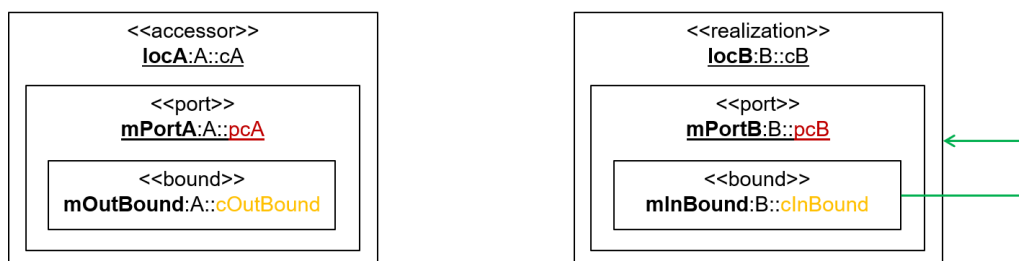


Bild 10: Objekte der Klassen `cA` und `cB` und deren eingebettete Objekte

Die Zeigerinitialisierung von `mInBound` nach `locB` übernimmt die Implementierung des Default-Konstruktors der Klasse `cB`.

2. Die beiden **Portobjekte** `mPortA` und `mPortB` der beiden Objekte `locA` und `locB` miteinander **verbinden**:

```
locA.get_mPortA()->get_mOutBound()->set_mptr_icX(
    locB.get_mPortB()->get_mInBound());
```

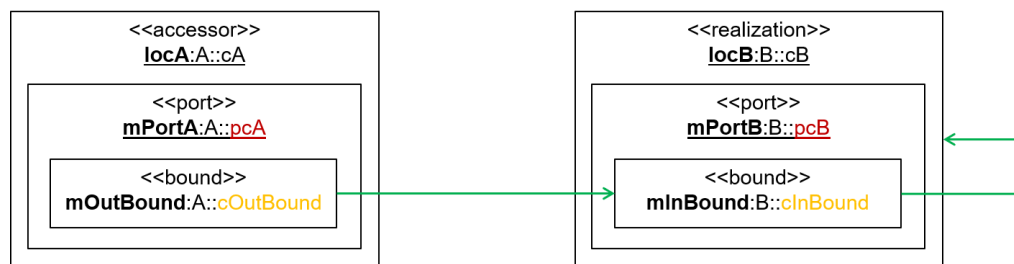


Bild 11: Portverbindung

3. Die **Funktion** `execute()` der Klasse `cA` für das Objekt `locA` **aufrufen**:

```
locA.execute();
```

Dieser Aufruf führt über die eingebetteten Objekte zu der folgenden Funktionssequenz:

```
cOutBound::function()
cInBound::function()
cB::function()
```

Der komplette, lauffähige C++ Programmcode zu diesem Implementierungsbeispiel 1 (Port_Basis_Concept) ist im Download [1] enthalten.

5. Implementierungsbeispiel 2: Controller und Counter

Im Gegensatz zu dem vorangegangenen abstrakten Beispiel ist es mit den Applikationsklassen `cController` und `cCounter` hier konkreter. Die Implementierungsansätze bleiben die Gleichen. Es ist immer noch eine unidirektionale Portverbindung mit einem `provided` bzw. `required` Interface.

Als Erweiterung zum vorangegangenen abstrakten Beispiel enthält die Interface-Klasse mehrere rein virtuelle Funktionen. Ein weiterer Unterschied sind die Implementierungsvarianten der virtuellen Interface-Funktionen `count()` in den Klassen `cUpCounter` und `cDownCounter`. Dazu enthält die Klasse `cController` zwei Instanzen der Portklasse `pcController`.

Die `control()` Funktion der Klasse `cController` ruft ihre privaten Funktionen `controlOutput()` und `controlInput()` in einer Endlosschleife auf. Die `controlOutput()` Funktion zeigt die Daten der Objekte aus den Klassen `cUpCounter` und `cDownCounter` an. Die Funktion `controlInput()` bietet die Möglichkeit, verschiedene Funktionen auf die Objekte anzuwenden.

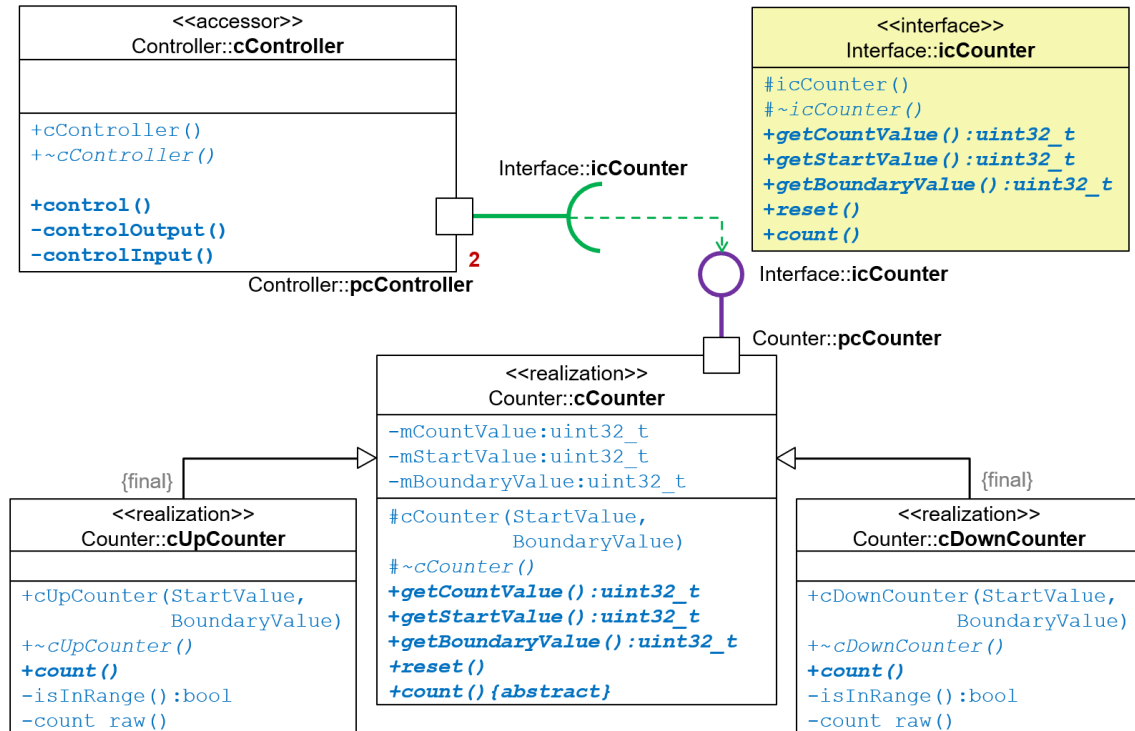


Bild 12: Implementierungsbeispiel 2

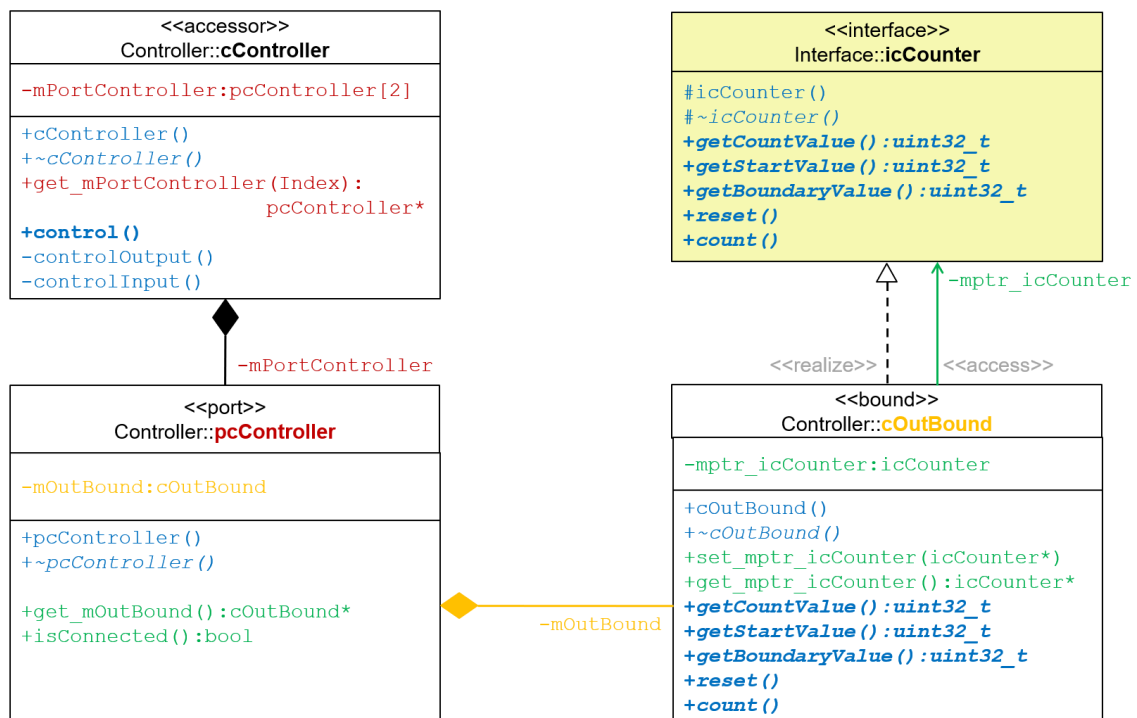


Bild 13: Implementierungspfad cController

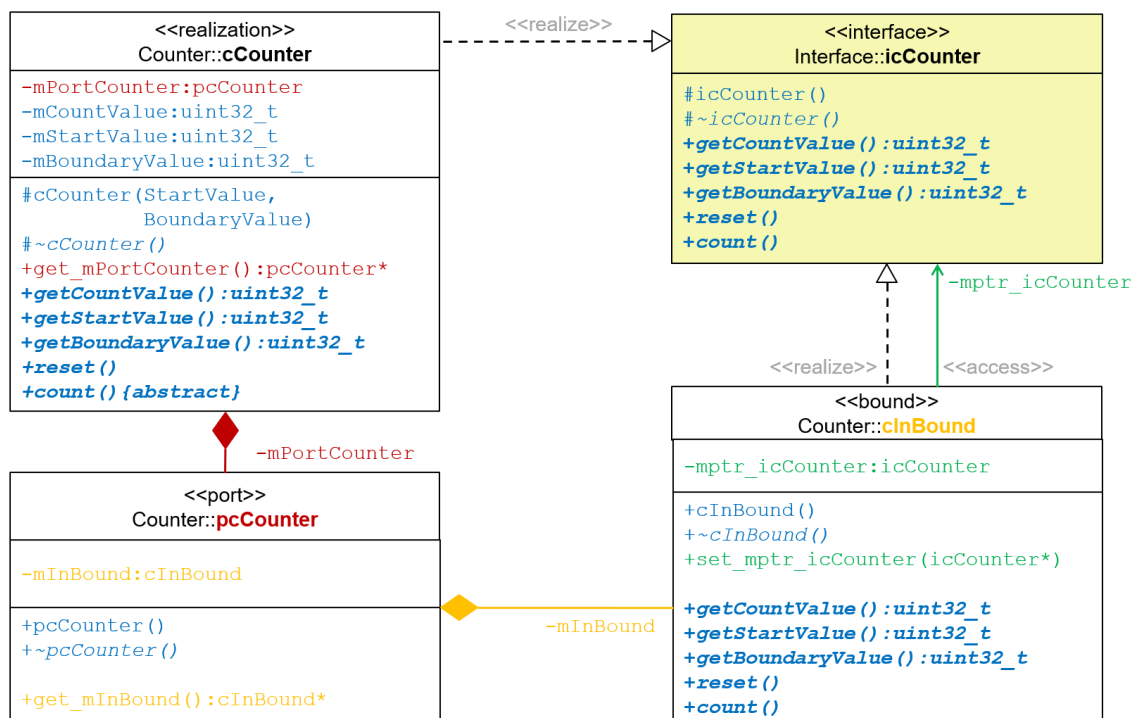


Bild 14: Implementierungspfad cCounter

Der komplette, lauffähige C++ Programmcode zu diesem Implementierungsbeispiel 2 (Port_Interface_Virtual) ist im Download [1] enthalten.

6. Implementierungsaspekte

Implementierungsaspekte bei Embedded-Software sind meist die Einhaltung von Qualitätsmerkmalen, wie Verständlichkeit, Einfachheit, (funktionale) Sicherheit, Zuverlässigkeit, Performance, Verbrauchsverhalten (Daten- und Programmspeicher).

Hierzu ein paar Gedanken zur diesbezüglichen Modifikation der vorgestellten Port-Implementierungsansätze:

- Singulär vorhandene eingebettete Objekte sind durch eingebettete Klassen ersetzbar.
- Provided und required Interfaces sind ohne Bound-Klasse(n) direkt in der Port-Klasse implementierbar.
- Anwendung von Template-Klassen anstatt virtueller Funktionen – aus der dynamischen wird eine statische Polymorphie [2].
- Die Portverbindung ist in dem Konstruktor / der Funktion einer anderen Klasse (`cBuilder` / `cManager`) initialisierbar.
- Sind Portverbindungen statisch, können diese einmalig zu Beginn verifiziert werden, ohne sie bei jedem Portaufruf erneut verifizieren zu müssen.
- Weitere Aspekte sind in [5] enthalten.

7. Resümee

Für einen Großteil heutiger Embedded-Software sind klassische Interfacekonzepte ohne Verwendung von Ports völlig ausreichend. Jedoch steigt die Software-Komplexität in vielen Produktbereichen rasant an. Themen wie beispielsweise Cloud-Anbindung, Cyber Physical Systems (CPS), echte Künstliche Intelligenz oder verteilte und vernetzte Systeme führen zu dieser steigenden Software-Komplexität. Aber auch die Innovationen der Mikrocontroller und Prozessoren (gestiegene Performance, Multi- und Manycore) verlangen nach veränderten Software-Architekturkonzepten. Mit wachsenden Software-Anforderungen steigen Umfang und Komplexität der Software-Architektur und der darin enthaltenen der Softwareschnittstellen. Genau an dieser Stelle kommt das hier vorgestellte Portkonzept zum Einsatz.

8. Referenzierte und weiterführende Links

[1] MicroConsult-Download für diesen Beitrag - komplett und aktuell

<http://download.microconsult.net/ese2021/port-designs.zip>

[2] MicroConsult-Download: Interface-Designs und ihre Implementierungsansätze

<http://download.microconsult.net/ese2020/interface-designs.zip>

[3] Object Management Group (OMG) - Unified Modeling Language (UML) Standard

www.uml.org

[4] IBM Engineering Systems Design Rhapsody

<https://www.ibm.com/products/uml-tools>

[5] Lightweight Realization of UML Ports for Safety-Critical Real-Time Embedded Software

<https://www.scitepress.org/Papers/2016/56896/>

[6] **MicroConsult-Trainings – Live-Online, Präsenz, Onsite:**

[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](#)

[RTOS-Grundlagen und Anwendung](#)

www.microconsult.de

9. 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

www.microconsult.de