MICRO CONSULT

C++ Programmierung: Dynamische oder statische Polymorphie?

Mit steigender Komplexität von Embedded-Software erlangt die Erfüllung von Qualitätsmerkmalen, wie Änderbarkeit, Erweiterbarkeit, Anpassbarkeit und Wiederverwendbarkeit, eine immer größere Bedeutung. Ein wichtiges Mittel, um diese Software-Qualitätsanforderungen zu erfüllen, ist die Anwendung von polymorphen Strukturen in der Architektur, im Design und in der Implementierung. Die Softwareentwicklung unterscheidet dynamische und statische Polymorphie.

Dieser Beitrag erklärt die dynamische und statische Polymorphie und zeigt deren Anwendung an einem einfachen Fallbeispiel. Ein Ergebnisvergleich erfolgt anhand von drei verschiedenen Design- und Implementierungsansätzen. Um diesen Beitrag vollständig zu verstehen, sind Kenntnisse der UML, der objektorientierten Programmierung und der Programmiersprache C++ vorausgesetzt.

Polymorphismus

Im Kontext der Software-Entwicklung bedeutet Polymorphie die Vielgestaltigkeit von Funktionen, also Funktionen mit gleicher Semantik und Aufruf bei unterschiedlichen Implementierungen.

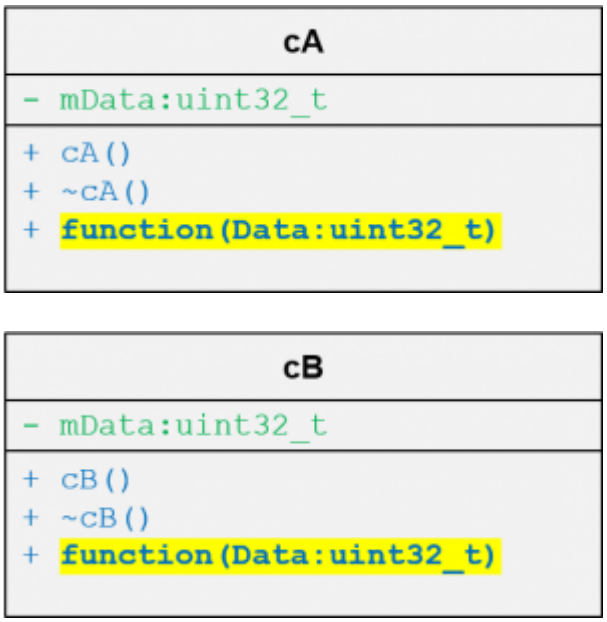

Bild 1: Polymorphie bei Klassen-Funktionen

Im Kontext der objektorientierten Software-Entwicklung bezieht sich Polymorphie auf Klassen-Funktionen. Mehrere Klassen enthalten eine oder mehrere semantisch identische, aber klassenspezifisch unterschiedlich implementierte Funktionen.

Der Aufruf der unterschiedlich implementierten Funktionen erfolgt dennoch gleichartig. Somit ergeben sich bei einem gleichartigen Aufruf zwei oder mehr Varianten der Funktionsausführung, auch Polymorphie genannt (Vielgestaltigkeit).

Dynamische versus statische Polymorphie

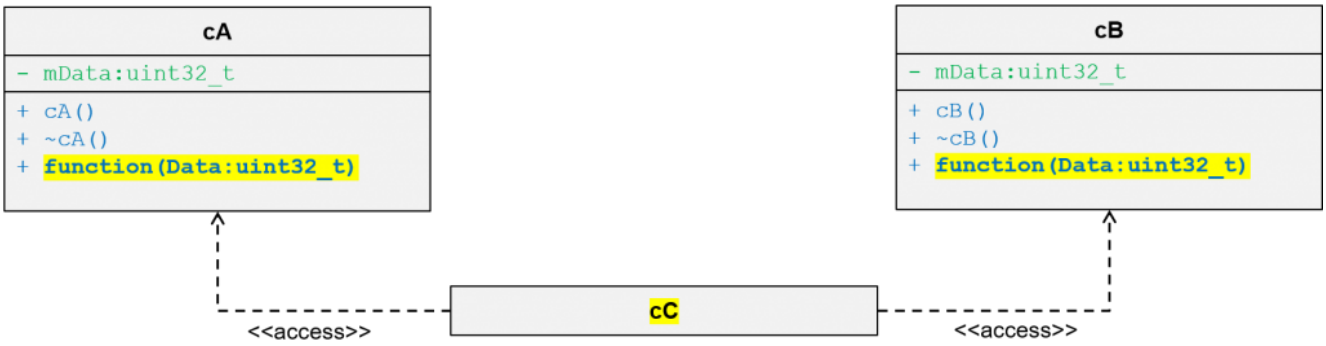

Bild 2: Polymorphie bei Klassen-Funktionen mit Zugreifer

Bei der Anwendung der **dynamischen Polymorphie** ist es eine Laufzeit-Entscheidung, welche der möglichen Funktionen aufgerufen wird (dynamische / späte Bindung), und die aufrufbare **Funktion** ist gegen eine andere ebenfalls **zur Laufzeit austauschbar**.

Bei der Anwendung der **statischen Polymorphie** ist es eine Compilezeit-Entscheidung, welche der möglichen Funktionen aufgerufen wird (statisch / frühe Bindung), und die aufrufbare **Funktion** ist gegen eine andere nur **zur Compilezeit austauschbar**.

Beispiel: Ressourcen-Zugriffsschutz

Die Klasse cCounter ist nicht thread-safe, da sie innerhalb verschiedener Funktionen auf die statische Member-Variable mStartValue zugreift.

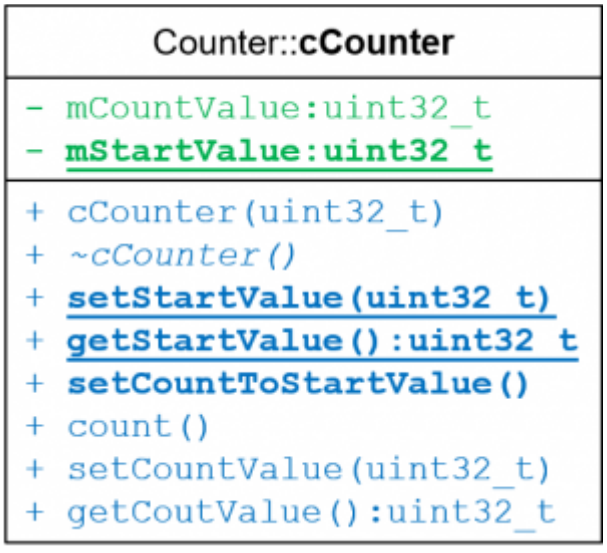

Bild 3: Klasse cCounter

In der von cCounter spezialisierten (vererbten) Klasse cCounter_ThreadSafe soll durch verschiedenartige Betriebssystem-Mechanismen (Critical Section, Mutex, später optional Semaphore) der Ressourcenschutz für mStartValue flexibel austauschbar gewährleistet werden.

Für die gleichartigen Betriebssystem-Mechanismen Klassen bietet sich die Anwendung der Polymorphie an: Die Klassen cCriticalSection, cMutex und cSemaphore enthalten semantisch identische Funktionen zum Anfordern / Sperren lock()und zum Freigeben der Ressource unlock(). Die weiteren Detailunterscheidungen dieser Ressource-Schutzmechanismen stehen nicht im Mittelpunkt dieses Beitrags und werden hier nicht weiter erläutert.

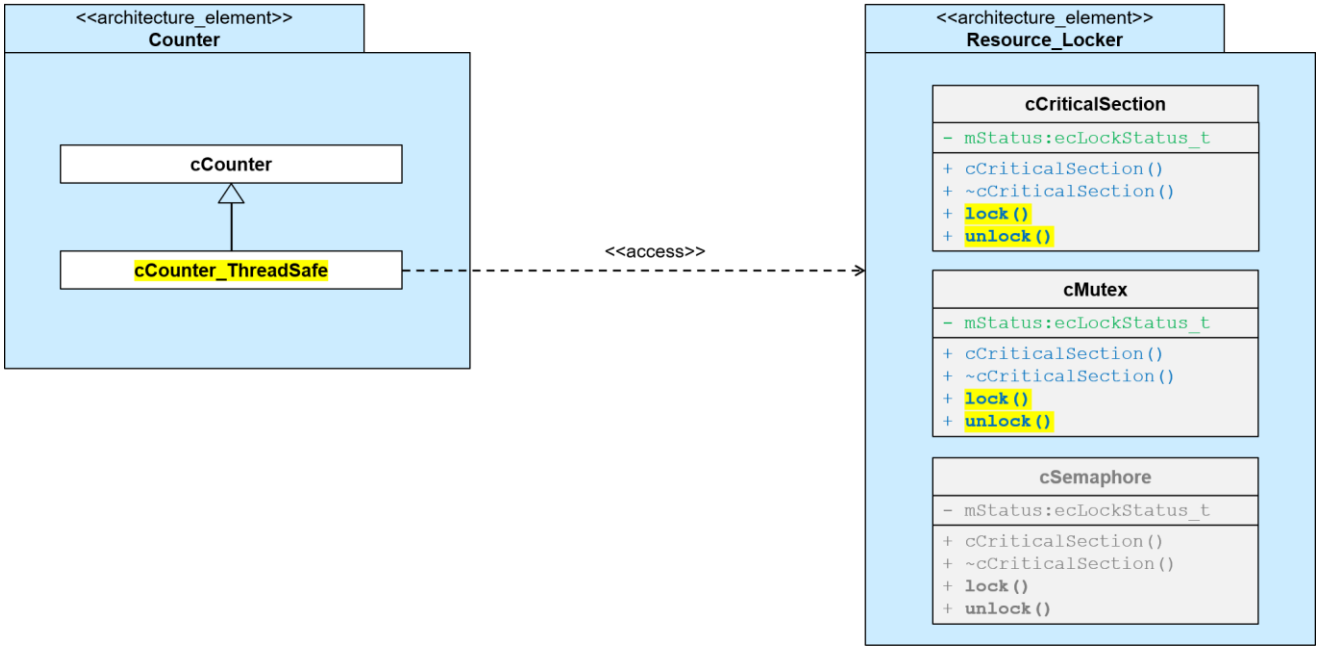

Bild 4: Beispiel Ressourcen-Zugriffsschutz: Architektur und Design

Der folgende Abschnitt stellt drei konkrete **Design- und Implementierungsansätze** vor:

- **Interface und Assoziation**
- **Template Parameter**
- **CRTP (Curiously Recurring Template Pattern)**

Neben der Unterscheidung zwischen dynamischer und statischer Polymorphie erfolgt ebenfalls ein Effizienzvergleich.

Interface und Assoziation

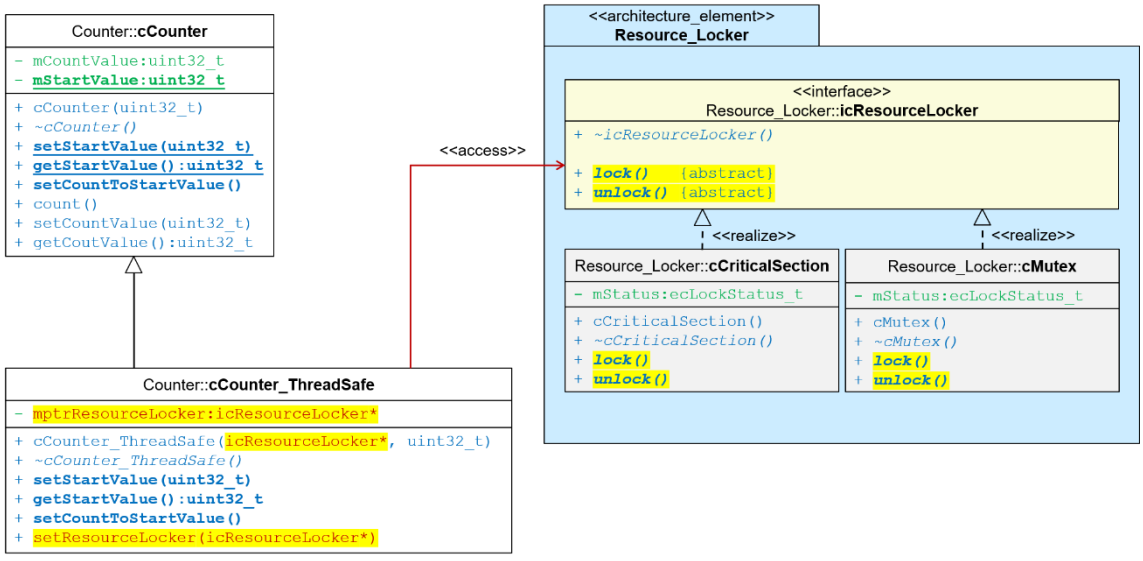

Bild 5: Interface und Assoziation: Architektur und Design

Die gemeinsame Interface-Klasse icResourceLocker abstrahiert die konkreten Ressource-Locker-Funktionen lock() und unlock(). In C++ sind dies rein virtuelle Funktionen.

```
class icResourceLocker
{
  public:
    virtual ~icResourceLocker() =default;
    virtual void lock(void)     =0;
    virtual void unlock(void)   =0;
};
```

Die beiden Ressource-Locker-Klassen cCriticalSection und cMutex implementieren das Interface durch eine Vererbung und überschreiben die Interface-Funktionen.

```
class cMutex : public icResourceLocker
{
  public:
    cMutex(void)       =default;
    ~cMutex() override =default;
    void lock(void)   override;
    void unlock(void) override;

  private:
    ecLockStatus_t mStatus = ecLockStatus_t::Unlocked;
};
```

Auf die Anbindung an ein konkretes Betriebssystem wurde hier verzichtet, da es nicht weiter zum Verständnis der Polymorphie erforderlich ist. Zur Symbolisierung des Ressourcen-Zustands ist die Klasse ecLockStatus_t als enum class enthalten.

```
void cMutex::lock(void)
{
  // ... operating system call
  mStatus = ecLockStatus_t::Locked;
  showValue("\nMutex status =   locked");
}
void cMutex::unlock(void)
{
  // ... operating system call
  mStatus = ecLockStatus_t::Unlocked;
  showValue("\nMutex status = unlocked");
}
```

Die showValue()Ausgabefunktion ist Teil der in den Beispielen enthaltenen kleinen Plattform zur Portierung auf beliebige Targets. Die Implementierungen von cCriticalSection und cMutex sind wesentlich identisch.

Die von cCounter abgeleitete Klasse cCounter__ThreadSafe definiert die Zugriffsfunktionen auf das statische Attribut mStartValue der Basisklasse mit Ressourcen-Schutz neu.

```
class cCounter_ThreadSafe : public cCounter
{
  public:
    cCounter_ThreadSafe(
            icResourceLocker* const ptrResourceLocker = nullptr,
            uint32_t const CountValue = 0);
    ~cCounter_ThreadSafe() override =default;
    void setStartValue(uint32_t const StartValue);
    uint32_t getStartValue(void);
    void setCountToStartValue(void);
    void setResourceLocker(icResourceLocker* const ptrResourceLocker);

  private:
      icResourceLocker* mptrResourceLocker;
};
```

Der Zugriff auf den Resource-Locker erfolgt über den Zeiger mptrResourceLocker vom Typ der Interface-Klasse icResourceLocker. Somit kann der Zeiger auf ein Objekt vom Typ cCrtiticalSection oder cMutex zeigen (siehe LSP [Liskov Substitution Principle]). Über den Konstruktor und / oder die cCounter_ThreadSafe::setResourceLocker() Funktion erfolgt die Auswahl des gewünschten Resource-Lockers.

```
void cCounter_ThreadSafe::setCountToStartValue(void)
{
  if (mptrResourceLocker != nullptr)
  {
    mptrResourceLocker->lock();
  }
  cCounter::setCountToStartValue();

  if (mptrResourceLocker != nullptr)
  {
    mptrResourceLocker->unlock();
  }
}
```

Das Beispiel der cCounter_ThreadSafe::setCountToStartValue() Funktion zeigt exemplarisch die Anwendung des Resource-Schutzmechanismus: sperren, zugreifen und freigeben. Der Aufruf der Funktionen lock() und unlock() erfolgt jeweils über den Zeiger. Somit ist der konkrete Aufruf / Bindung vom Typ des Objektes abhängig, auf welches der Zeiger initialisiert ist. Dabei greift der Mechanismus der dynamischen Bindung mittels VMT (Virtual Method Tables). Hier zeigt sich die Anwendung der Polymorphie.

Die Anwendung instanziiert Resource-Locker-Objekte und initialisiert damit Thread-Safe Counter-Objekte.

```
Resource_Locker::cCriticalSection locCriticalSection{ };
Resource_Locker::cMutex locMutex{ };

Counter::cCounter_ThreadSafe locCounter_A{ &locCriticalSection };
Counter::cCounter_ThreadSafe locCounter_B{ &locMutex };
```

Unter anderem sind Funktionen, die den Ressourcen-Schutz enthalten, auf die Zählerobjekte anwendbar.

```
locCounter_A.setCountToStartValue();
locCounter_B.setCountToStartValue();
locCounter_A.count();
locCounter_B.count();
```

Mittels der Funktion cCounter_ThreadSafe::setResourceLocker() lassen sich die Resource-Locker-Objekte zur **Laufzeit** tauschen.

```
locCounter_A.setResourceLocker(&locMutex);
locCounter_B.setResourceLocker(&locCriticalSection);
```

Somit entspricht diese Variante einem Beispiel für die **dynamische Polymorphie**.

Template Parameter

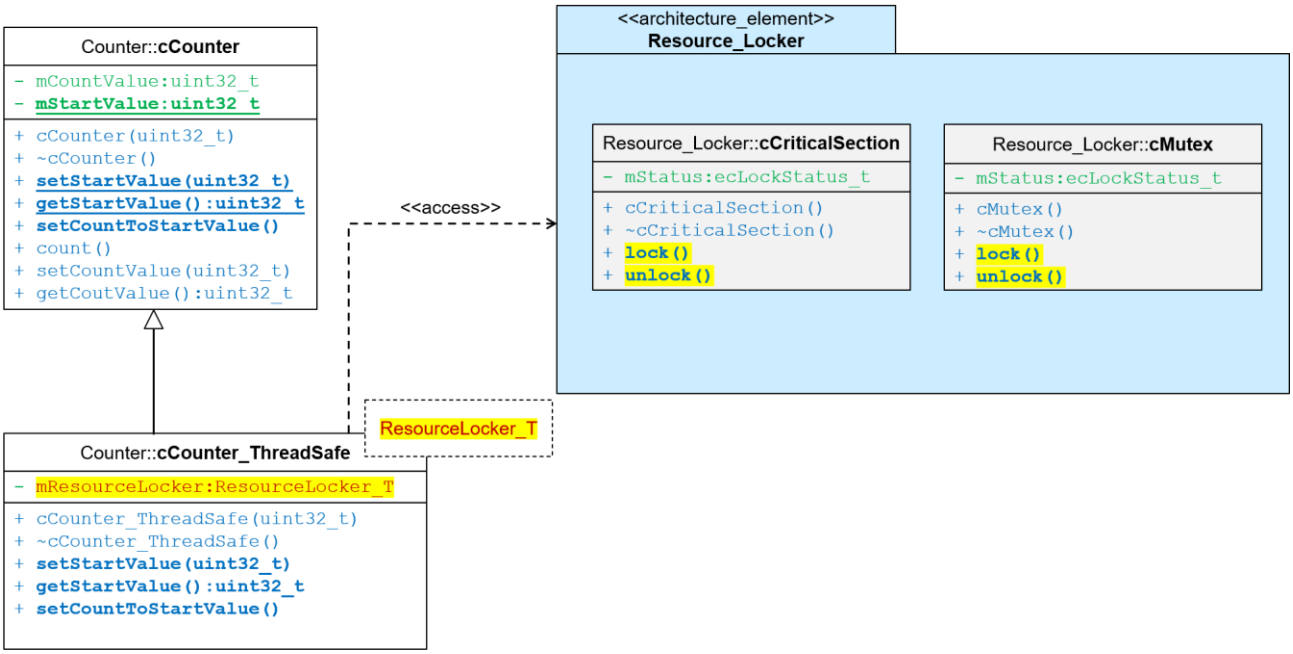

Bild 6: Template Parameter: Architektur und Design

Im Vergleich zu der vorherigen Variante enthält der Resource-Locker kein Interface und keine virtuellen Funktionen mehr.

```
class cMutex
{
  public:
    cMutex(void) =default;
    ~cMutex()    =default;
    void lock(void);
    void unlock(void);

  private:
    ecLockStatus_t mStatus = ecLockStatus_t::Unlocked;
};
```

Die weitere Implementierung von cMutex und cCriticalSection bleibt unverändert.

Diese Variante ersetzt den vorherigen Interface-Zeiger in der Klasse cCounter_ThreadSafe der Template Parameter ResourceLocker_T. Die spätere Typisierung dieses Parameters selektiert den zur Anwendung kommenden Resource-Locker, was wiederum einer Polymorphie entspricht.

```
template<typename ResourceLocker_T>
class tcCounter_ThreadSafe : public cCounter
{
  public:
    tcCounter_ThreadSafe(uint32_t const CountValue = 0);
    ~tcCounter_ThreadSafe() =default;
    void setStartValue(uint32_t const StartValue);
    uint32_t getStartValue(void);
    void setCountToStartValue(void);
  private:
    ResourceLocker_T mResourceLocker;
};
```

Anstatt des Interface-Zeigers kommt ein konkretes Objekt vom Typ des Template Parameters zur Anwendung; somit entfallen die sonst benötigten Zeigerabfragen.

```
template<typename ResourceLocker_T>
void tcCounter_ThreadSafe<ResourceLocker_T>::setCountToStartValue(void)
{
  mResourceLocker.lock();
  cCounter::setCountToStartValue();
  mResourceLocker.unlock();
}
```

Die Bindung zwischen Objekt und Funktion erfolgt in diesem Fall statisch / früh, also zur Compilezeit.

Die Anwendung instanziiert Objekte der Template-Klasse cCounter_ThreadSafe und initialisiert damit den Resource-Locker über den Template Parameter.

```
tcCounter_ThreadSafe<cCriticalSection> locCounter_CriticalSection{ };
tcCounter_ThreadSafe<cMutex>           locCounter_Mutex{ };
```

Im Vergleich zur vorherigen Variante lassen sich hier die Resource-Locker nur zur Codier- / Compilezeit festlegen, aber nicht mehr zur Laufzeit tauschen.

Somit entspricht diese Variante einem Beispiel für die **statische Polymorphie**.

CRTP

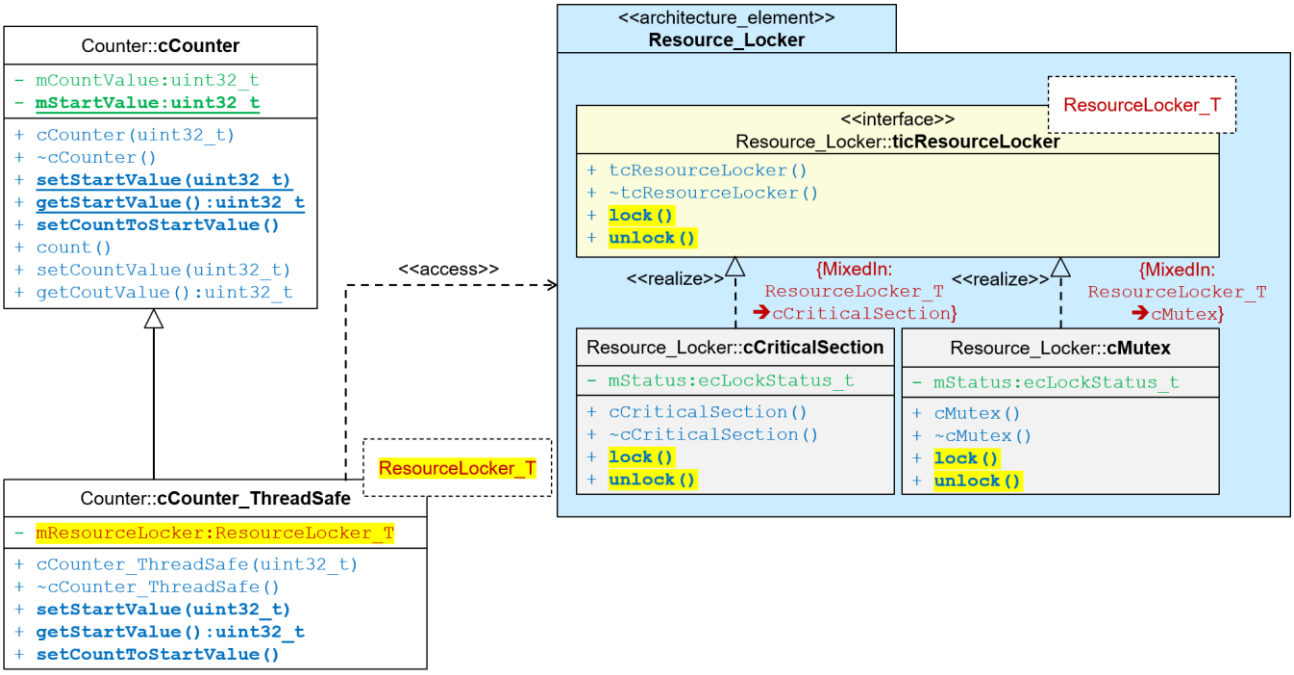

Bild 7: CRTP: Architektur und Design

Ein Kritikpunkt an der Template-Parameter-Variante könnte der sein, dass eine gemeinsame Interface-Vereinbarung für die Resource-Locker-Klassen fehlt. Diesen Kritikpunkt eliminiert die Anwendung des Curiously Recurring Template Patterns (CRTP).

```
template<typename ResourceLocker_T>
class ticResourceLocker
{
  public:
    ticResourceLocker(void) =default;
    ~ticResourceLocker()    =default;
    void lock(void);
    void unlock(void);
};
```

Auch diese Implementierungsvariante bleibt frei von virtuellen Funktionen. Die Interface-Klasse ist nun ebenfalls eine Template-Klasse, die als Template-Parameter den konkreten Resource-Locker erwartet.

```
} template<typename ResourceLocker_T>
void ticResourceLocker<ResourceLocker_T>::lock(void)
{
  static_cast<ResourceLocker_T*>(this)->lock();
}
template<typename ResourceLocker_T>
void ticResourceLocker<ResourceLocker_T>::unlock(void)
{
  static_cast<ResourceLocker_T*>(this)->unlock();
}
```

Die Funktionen ticResourceLocker::lock() und ticResourceLocker::unlock() rufen jeweils über den Template Parameter die entsprechend in den abgeleiteten Klassen spezialisierten Funktionen auf.

```
class cMutex : public ticResourceLocker<cMutex>
{
  public:
    cMutex(void) =default;
    ~cMutex()    =default;
    void lock(void);
    void unlock(void);

  private:
    ecLockStatus_t mStatus = ecLockStatus_t::Unlocked;
};
```

Bei der Vererbungsimplementierung wird bereits der Template Parameter der Basisklasse mit dem Typ der erbenden Klasse spezifiziert (MixedIn). Dies ist am Beispiel der Klasse cMutex gezeigt.

```
void cMutex::lock(void)
{
  // ... operating system call
  mStatus = ecLockStatus_t::Locked;
  showValue("\nMutex status =   locked");
}

void cMutex::unlock(void)
{
  // ... operating system call
  mStatus = ecLockStatus_t::Unlocked;
  showValue("\nMutex status = unlocked");
}
```

Die Funktionsimplementierungen von lock() und unlock() bleiben unverändert. Ebenfalls unverändert im Vergleich mit der Variante Template Parameter bleibt die cCounter_ThreadSafe Klassenimplementierung.

```
tcCounter_ThreadSafe<ticResourceLocker<cCriticalSection>>
                        locCounter_CriticalSection{ };

tcCounter_ThreadSafe<ticResourceLocker<cMutex>>
                        locCounter_Mutex{ };
```

Der Resource-Locker wird nur zur Codier- / Compilezeit festgelegt und ist nicht zur Laufzeit tauschbar.

Somit entspricht diese Variante einem zweiten, erweiterten Beispiel für die **statische Polymorphie**.

Vergleich

Dieser Abschnitt vergleicht alle drei Varianten (Interface und Assoziation, Template Parameter und CRTP). Die Verbrauchsangaben basieren auf dem folgenden Setup:

- STM32F407 Mikrocontroller mit SYSCLK = 168MHz
- MCBSTM32F400 Evaluation Board
- MDK-ARM v5.35.0.2 Tool Chain
- ARM Clang Compiler v6.16
- Compiler Optimierung: Default
- C++14

Aspekte	Interface & Assoziation	Template Parameter	CRTP
Programmcode im Programm-speicher der C++ Library [Bytes]	812	518	518
Programmcode im Programm-speicher der Applikation [Bytes]	9404	9010	9058
Konstante Daten im Programm-speicher Applikation [Bytes]	616	512	512
VMTs als konstante Daten im Programmspeicher	Ja	Nein	Nein
VMT-Zeiger als zusätzliches Attribut im Datenspeicher	Ja	Nein	Nein
Anzahl ausgeführter Assembler-Instruktionen Konstruktor cCounter_ThreadSafe mit cCriticalSection	35	33	22
Anzahl ausgeführter Assembler-Instruktionen cCounter_ThreadSafe:: setCountToStartValue()	53	39	53
Anzahl ausgeführter Assembler-Instruktionen cCounter::count()	9	9	9
Erweiterung um die Klasse CSemaphore	• Neue Klasse • Realisierung der Interfacefunktionen • Instanziierung	• Neue Klasse • Instanziierung • Template Parameter setzen	• Neue Klasse • Realisierung der Interfacefunktionen • Instanziierung • Template Parameter setzen
Zusatzaufwände durch VMT	Ja	Nein	Nein
Zusatzaufwände durch Interface-Zeiger	Ja	Nein	Nein
Objekt zu Funktionsbindung	Zur Laufzeit (dynamisch / spät)	Zur Compilezeit (statisch / früh)	Zur Compilezeit (statisch / früh)
Resource-Locker Austauschbarkeit	Zur Laufzeit	Nur zur Codier- / Compilezeit	Nur zur Codier- / Compilezeit
Polymorphismus	Dynamisch	Statisch	Statisch

Resümee

Die Implementierung der statischen Polymorphie ist aus der Perspektive der Software-Qualitätsmerkmale funktionale Sicherheit, Angriffssicherheit, Zuverlässigkeit und Verbrauchsverhalten immer die bessere Wahl. Der Preis, der dafür zu zahlen ist, ist der Verlust der positiven Unterstützung von Software-Qualitätsmerkmalen wie Änderbarkeit, Erweiterbarkeit oder Anpassbarkeit durch die dynamische Polymorphie.

Die Entscheidung „dynamische versus statische Polymorphie“ muss nicht digital für die gesamte Software-Architektur getroffen werden – Mischformen sind denkbar. Doch wie immer hängen die konkreten Entscheidungen von den konkreten Software-Anforderungen ab.

Weiterführende Informationen

MicroConsult Fachwissen zum Thema Embedded SW-Entwicklung

MicroConsult Training & Coaching zu Analyse, Design, Architektur

Alle Trainings & Termine auf einen Blick

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.