# Embedded Software Manager Pattern

## Zentrale Aufgaben skalierbar in der Software etablieren

Thomas Batt, MicroConsult GmbH

**(Embedded-) Software muss verschiedene zentrale Aufgaben softwareweit koordinieren. Das klassische Beispiel dafür ist die Initialisierung, die auf allen Ebenen der Software stattfinden muss. Bei genauerer Betrachtung lassen sich produktabhängig viele weitere dieser softwareweiten Aktionen identifizieren.**

Dieser Beitrag [1] stellt für das Management dieser Aufgaben in der Software das Manager Pattern vor. Es ist hochgradig skalierbar und somit anwendbar für sehr einfache Software, aber auch für komplexeste Architekturen. Das Pattern ist unabhängig von der Programmiersprache und unterstützt sowohl die prozedurale als auch objektorientierte Softwareentwicklung. Um diesen Beitrag vollständig zu verstehen, sind Kenntnisse der UML [2], der prozeduralen und objektorientierten Programmierung in C und C++ [8] von Vorteil.

## 1. Zentrale Aufgabe in der Software

Zur Instanziierung und Initialisierung von Objekten in der Software sind bereits Design Patterns wie „Abstract Factory“ und „Builder“ als „Creational Patterns“ von Erich Gama et.al. [3] publiziert. Des Weiteren ist im Internet [4] das „Manager Design Pattern“, auch bekannt als „Manager-Managed Design Pattern“ oder „Managed Object Design Pattern“ veröffentlicht.

Das hier vorgestellte Embedded Software Manager Pattern verantwortet alle in der Software verteilten und zentral zu koordinierenden Aufgaben. Die Anzahl und Vielfalt der Aufgaben ist abhängig von den System-/ Softwareanforderungen, die das Produkt erfüllen muss. Dieser Beitrag betrachtet die folgenden Aufgaben:

| Zentral zu koordinierende verteilte Aufgabe in der Software | Funktionsaufruf |
|---|---|
| Instanziierung | `create()` |
| Initialisierung | `init()` |
| Initialisierung der Relationen zwischen Objekten | `buildRelations()` |
| Konfiguration, Parametrierung mittels einer (Mini-) Datenbank | `config()` |
| Allokation von Ressourcen | `allocateResources()` |
| Bootvorgang | `start()`, `taskInit()` |
| Applikationsausführung | `run()`, `taskOperation()` |
| Diagnose, Selbsttest und Debugging | `executeDiagnostics()` |
| Neustart | `restart()` |
| Herunterfahren | `shutdown()` |
| Fehlermanagement | `notifyError()`, `handleError()`, `recoverError()`, `enterFailSafe()` |

## 2. Prinzip des Embedded Software Manager Patterns

In einer sehr einfachen Software besteht das Manager Pattern lediglich aus einer `SoftwareManager` Klasse oder einem `SoftwareManager` Modul, welches die zuvor genannten Aufgaben übernimmt. Mit wachsender Softwarearchitektur erhalten die Architekturelemente (`AE`) auf jeder Architekturebene (`architectureLevel = x`) ihre eigenen `AEnManager`, die die Aufgaben dezentral für ihr Architekturelement übernehmen.

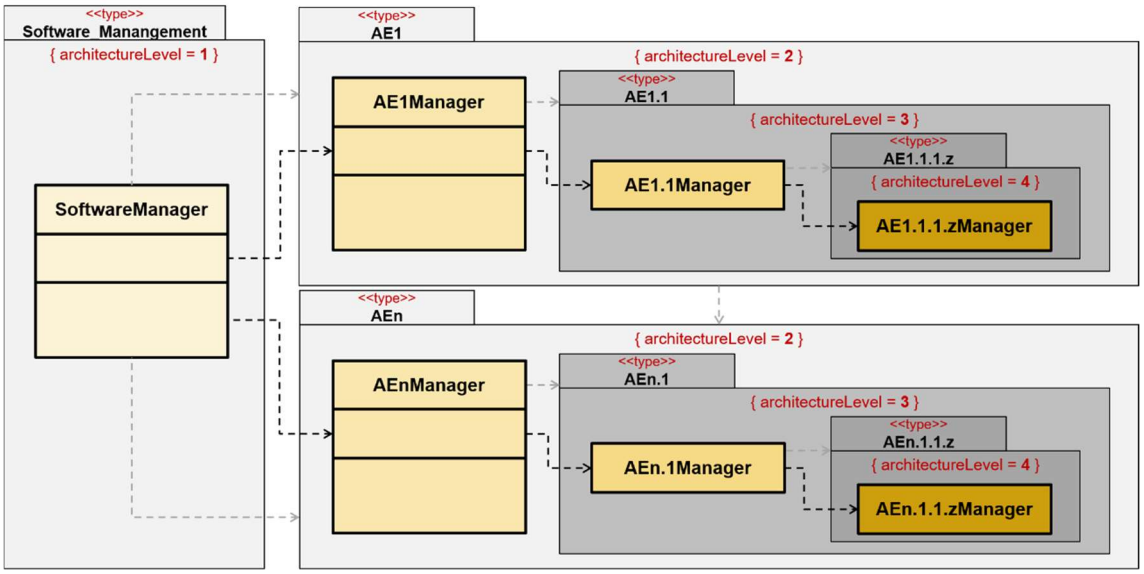


Bild 1: Prinzip des Manager Patterns

Die verschiedenen Architekturelemente sind durch Namen `<<type>>` typisierbar (z.B. Software-Layer, Software-Subsystem). Bei tieferen Architekturhierarchien sind häufig keine individuellen Typennamen mehr identifizierbar. Aus diesem Grund sind die Architekturhierarchien hier nicht mit Typennamen versehen, sondern der `architectureLevel` ist angegeben.

Somit ist das Manager Pattern für beliebig hierarchisch und beliebig flach wachsende Architekturen anwendbar. Die Manager sind fähig, untereinander zu kommunizieren.

## 3. Skalierbarkeit des Embedded Software Manager Patterns

Ein Architekturelement `AE1.1` kann theoretisch eine unbegrenzte Anzahl 0…z weiterer Architekturelemente auf der geleichen Ebene enthalten, die wiederum mit Managern ausgestattet sind.

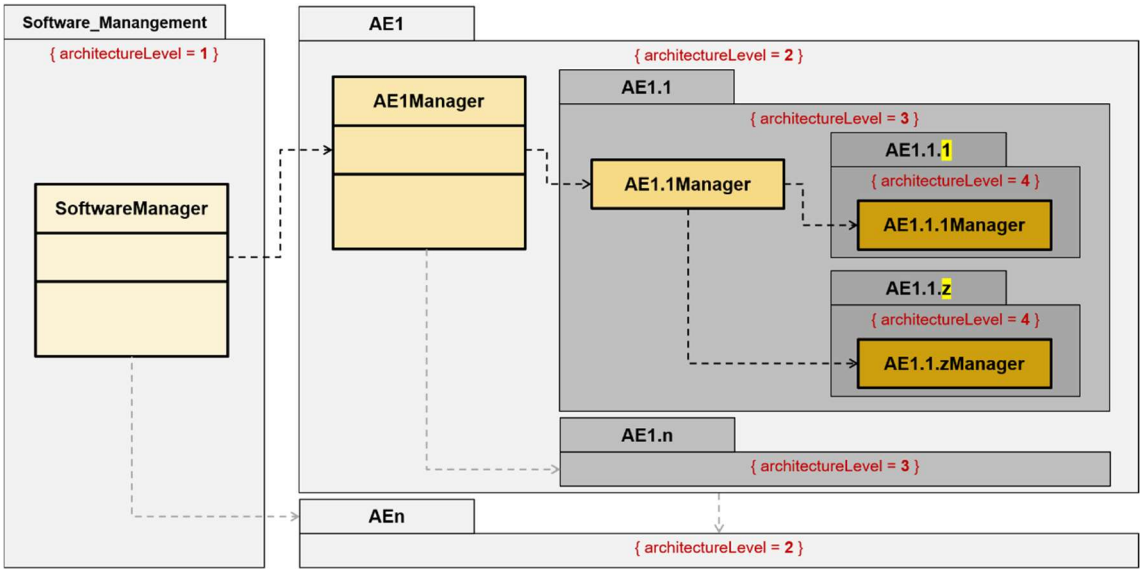


Bild 2: Manager Pattern mit mehreren Architekturelementen auf gleicher Ebene

Die Hierarchiestufen-Anzahl der Architekturelement ist theoretisch ebenfalls unbegrenzt von 1…x oder 1…y. Je nach Architekturzweig sind unterschiedliche Hierarchiestufen-Anzahlen möglich.

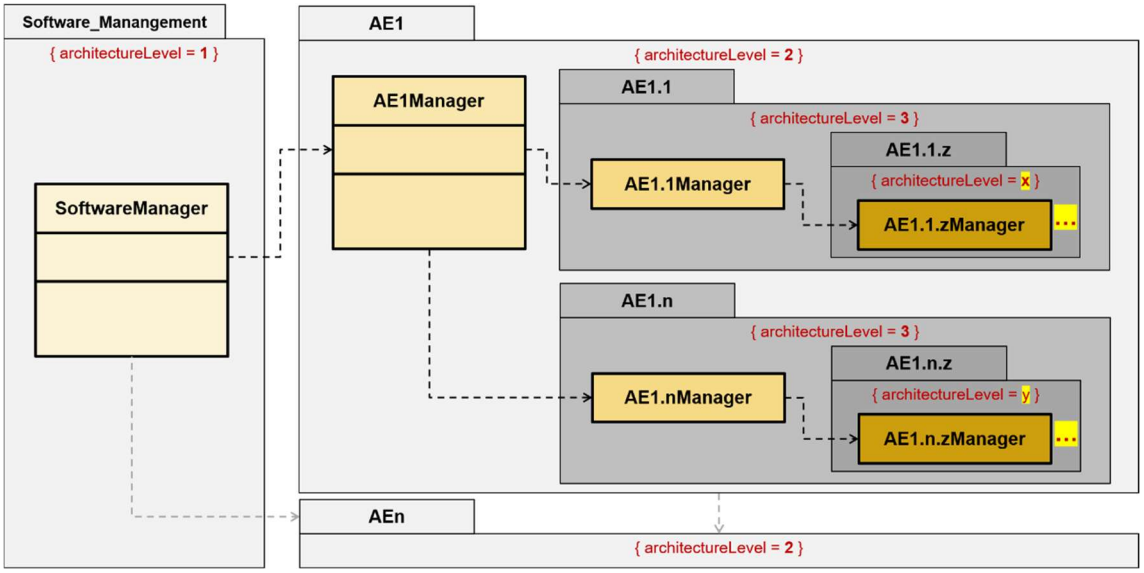


Bild 3: Manager Pattern mit unterschiedlichen Architekturelement-Hierarchiestufen

Unabhängig von der UML-Notation lassen sich die Architekturebenen und Anzahlen der Architekturelemente in den einzelnen Eben sehr gut in einem Baumdiagramm darstellen. Die Astknoten bilden die Managerklassen/ -module ab.

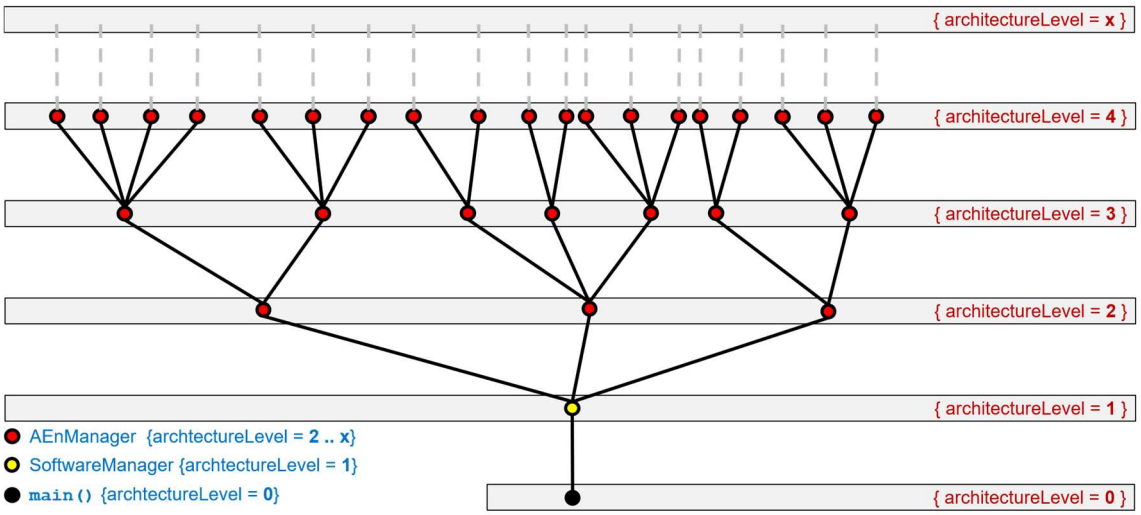


Bild 4: Baumdiagramm der Architekturebenen

Die unterste Architektureben 0 ist `main()` zugeordnet, während die Ebene 1 für den übergeordneten `SoftwareManager` reserviert ist. Danach beginnen die eigentlichen Architekturebenen 2…x mit ihren `AEnManagern`.

## 4. Initialisierung (prozeduraler Ansatz)

Im prozeduralen Umfeld kommen Managermodule zum Einsatz, symbolisiert durch das „m“ vor den Namen. Im Architekturelement `AEn` ist der `mAEnManager` für seine Module `m1`...`mN` verantwortlich.

`main()` ruft `init()` vom `mSoftwareManager` auf. Diese Funktion ruft wiederum von allen `mAEnManagern` die `initAEn()` Funktionen in der erforderlichen Reihenfolge auf (Delegationen). Diese Funktionen initialisieren die Module aus deren Verantwortlichkeit. **Das Prinzip der Delegation ist für die meisten weiteren Managerfunktionen anwendbar.**

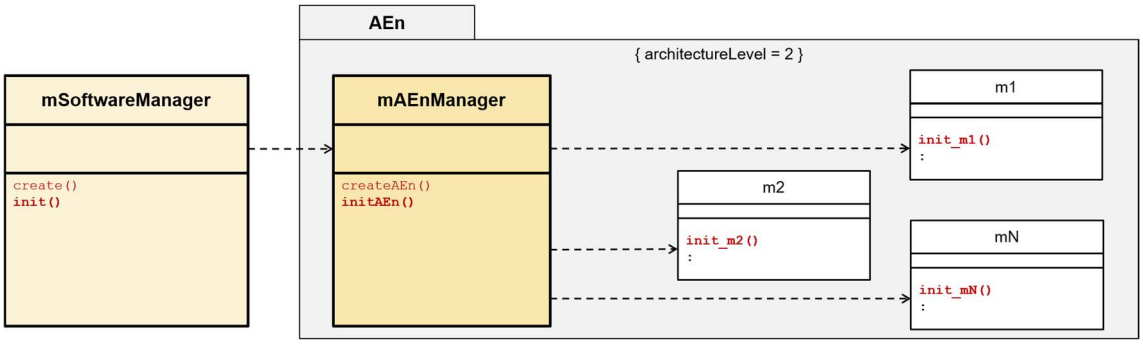


Bild 5: Manager Pattern für prozeduralen Ansatz

Im Vergleich zu einem objektorientierten Ansatz ist beim prozeduralen häufig keine Objektinstanziierung erforderlich. Somit könnten ggf. die `create()` Funktionen

entfallen. Die aus der objektorientierten Programmierung bekannten Konstruktoren und Destruktoren von Klassen lassen sich in der prozeduralen Programmierung durch manuell aufzurufende Funktionen `construct()` und `destruct()` ersetzen.

## 5. Objektinstanziierung und Initialisierung (objektorientierter Ansatz)

`main()` instanziiert ein `cSoftwareManager` Ur-Objekt. Dieses instanziiert alle `cAEnManager` Objekte, die wiederum alle Objekte der Klassen `c1...cN` aus deren Verantwortung instanziieren. Abhängig von den angewandten Relationen (Assoziation, Aggregation) erzeugen die `createX()` Funktionen deren Objekte dynamisch und implementierungsabhängig auf dem Heap. Bei der durchgängig angewandten Komposition werden alle Objekte automatisch auf dem Stack angelegt, wobei die `createX()` Funktionen entfallen.

Die `initX()` Funktion erlaubt zusätzlich zum Konstruktor-Aufruf weitere Initialisierungen. Als Alternative ruft der Konstruktor die jeweilige `initX()` Funktion auf.

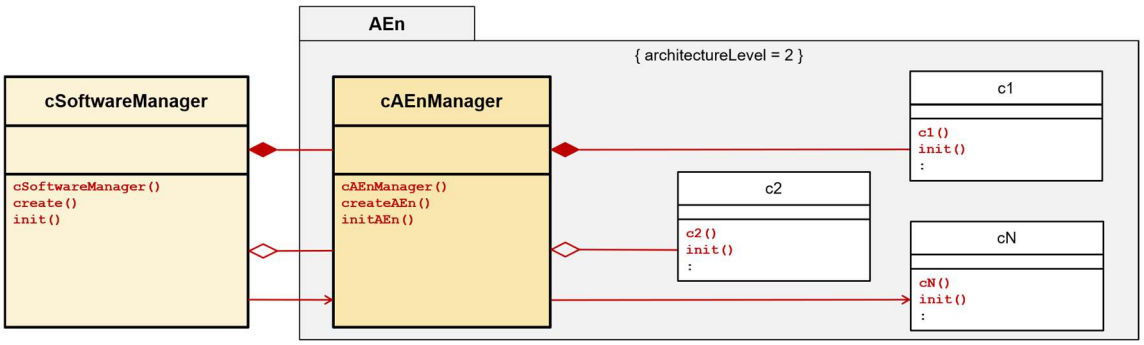


Bild 6: Manager Pattern für objektorientierten Ansatz

Ist die Architekturhierarchie tiefer als die hier dargestellte, erstreckt sich der oben beschriebene Ablauf über die gesamte Hierarchie.

Als Ergebnis dieses Schrittes sind alle Objekte instanziiert und grundinitialisiert.

## 6. Initialisierung der Relationen

Die Funktion `buildXRelations()` initialisiert die Relationen (Assoziation, Aggregation) zwischen den Objekten intern, aber auch über Architekturelement-Grenzen hinweg. Dazu gehören auch die Callback-Registrierungen. Typischerweise sind Zeiger mit entsprechender Objektaderesse initialisiert.

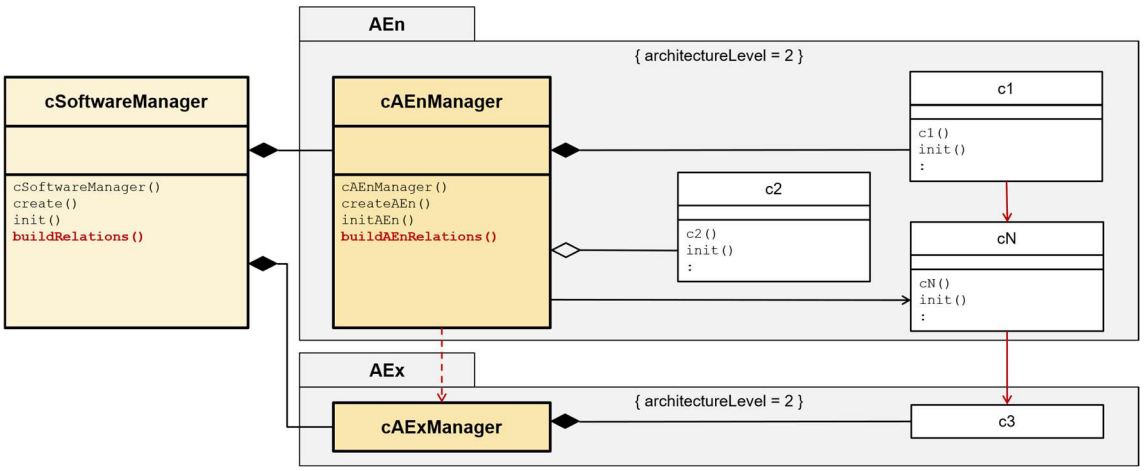


Bild 7: Manager Pattern um Relationsinitialisierung ergänzt

Häufig sind dazu Objektadressen aus nebenliegenden Architekturelementen erforderlich. Kennen sich diese `AEnManager` untereinander, so können sie Objektadressen übermitteln.

## 7. Konfiguration, Parametrierung, Datenbankzugriff

Nach der komplett abgeschlossenen Initialisierung detektiert die Software z.B., auf welcher Hardware die Ausführung stattfindet. Ergebnisabhängig liest der `SoftwareManager` die korrespondierenden Parameter aus einer lokalen oder in der Cloud befindlichen Datenbank.

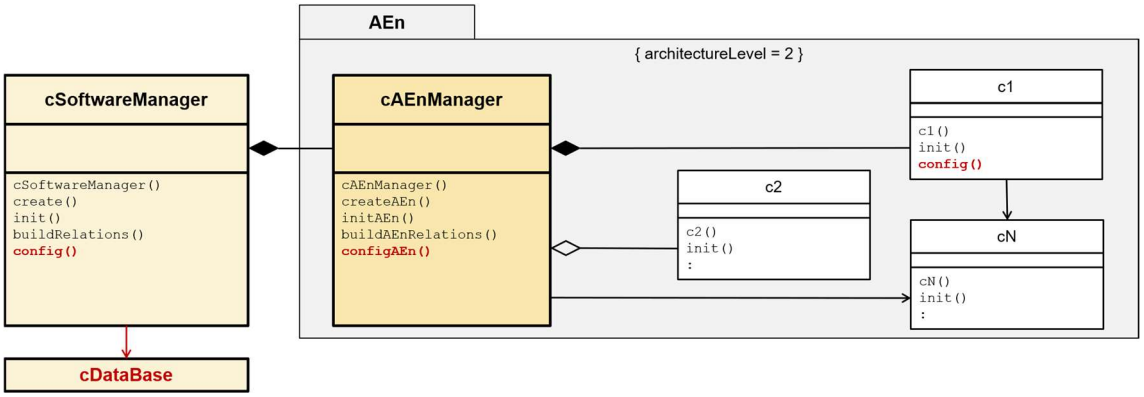


Bild 8: Manager Pattern um Konfiguration ergänzt

Die Funktion `configX()` der `AEnManager` gibt die Parameter zu den zu konfigurierenden Klassen/ Modulen weiter. Eventuell ist `configX()` bereits vor `initX()` ausführbar und gibt die Parameter über die `initX()` Funktion weiter.

## 8. Allokation von Ressourcen

Die einzelnen Architekturelemente benötigen ggf. für den weiteren Betrieb zusätzliche individuelle Ressourcen. Besonders in einer funktional-sicherheitskritischen Embedded-Software ist es empfehlenswert, die erforderlichen Ressourcen bereits zu Beginn der Softwareausführung komplett anzulegen. Damit ist zum einen sichergestellt, dass alle Ressourcen zur Laufzeit verfügbar sind, zum anderen erfolgt dadurch gleichzeitig eine Laufzeit-Performanceverbesserung.

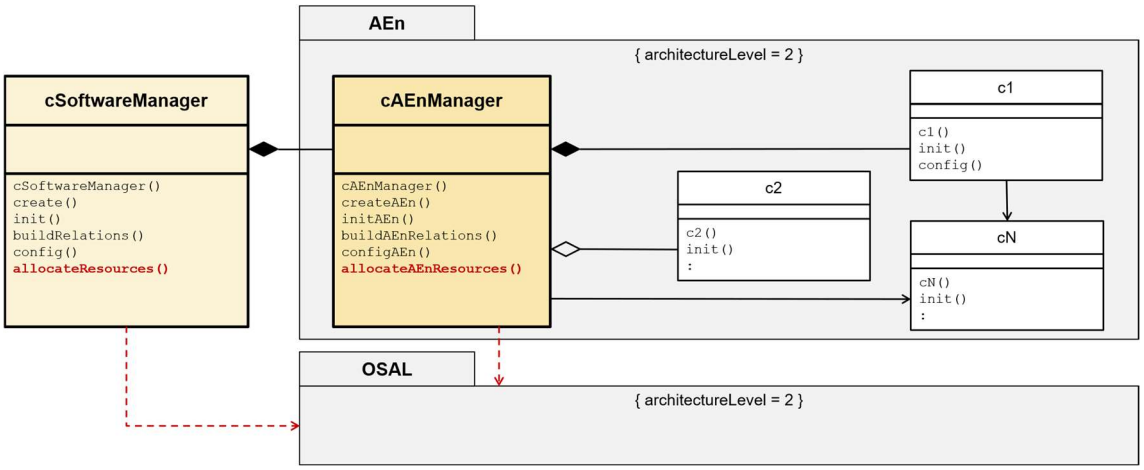


Bild 9: Manager Pattern um Ressourcen-Allokation ergänzt

Ressourcen sind z.B. Speicher oder bei Anwendung eines Betriebssystems bereits Betriebsmittel wie Mailboxen, Eventgruppen, Semaphore, Mutexe und Timer. Bedingung wäre, dass das Betriebssystem vor dem Bootvorgang dies bereits erlaubt. Falls nein, lässt sich die Reihenfolge der Funktionsaufrufe ändern oder der Funktionsaufruf `allocateXResources()` in die Architekturelement-spezifischen `taskInit()` verschieben.

## 9. Betriebsphase: Ohne Betriebssystem

Ein Bare-Metal-Applikation führt eine oder mehrere Funktionen kontinuierlich aus. Hier enthält die `run()` Funktion diese zentrale `while(true)` loop und führt in vorgegebener Reihenfolge kontinuierlich die `runAEn()` Funktionen aus.

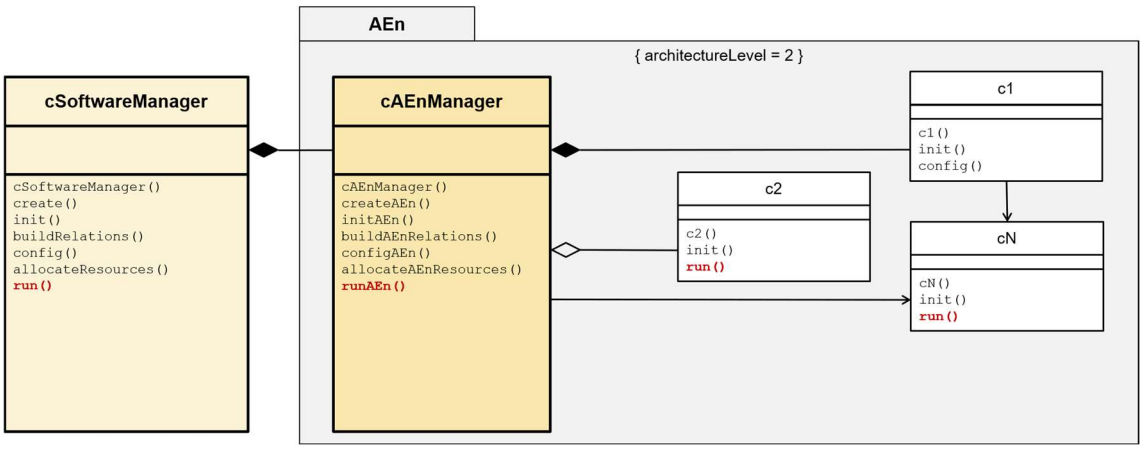

Bild 10: Manager Pattern um Betriebsphase ergänzt

## 10. Boot-, Startup- und Betriebsphase: Mit Betriebssystem

Die `start()` Funktion des `SoftwareMangers` bootet das Betriebssystem. Anschließend schedult das Betriebssystem als erste Task die `taskInit()`. Diese kreiert `taskOperation()` und alle `taskAEnInit()` Tasks der `AEnManager`. Anschließend beendet sie sich selbst. Die `taskAEnInit()` kreiert `taskARnOperation()` und ggf. weitere Architekturelement-spezifische Tasks. Anschließend beendet sie sich ebenfalls. Nun beginnt die eigentliche Betriebsphase mit dem Scheduling der verbleibenden Tasks.

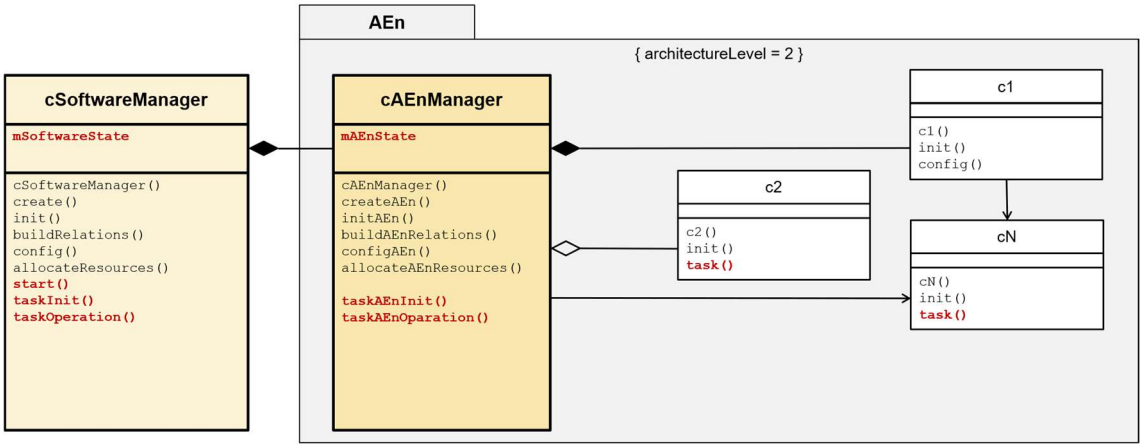


Bild 11: Manager Pattern für Startup und Betriebsphase

Die Manager-Taskfunktionen `taskXOperation()` enthalten einen Zustandsfolgeautomaten, der den aktuellen Zustand der Software bzw. der einzelnen Architekturelemente repräsentiert. Dieser Zustandsautomat ist im weiteren Verlauf des Kapitels vorgestellt.

## 11. Diagnose

Die Funktion `executeDiagnostics()` des `SoftwareManagers` startet die Diagnoseausführung und sammelt alle Diagnosedaten über die Aufrufe der in den `AEnManagern` enthaltenen `executeAEnDiagnostics()` Funktionen.

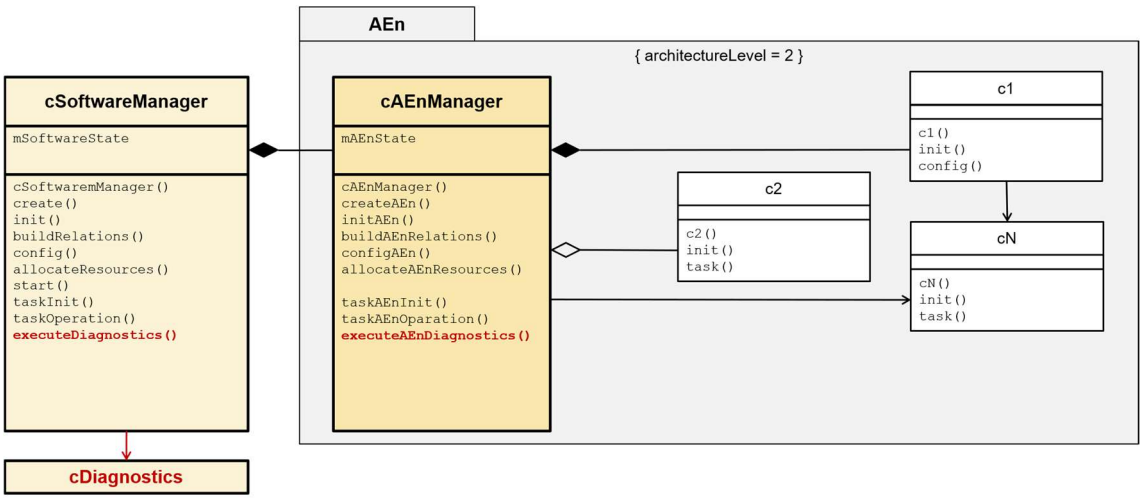


Bild 12: Manager Pattern um Diagnose ergänzt

Das Element `Diagnostics` speichert bzw. ermöglicht die Ausgabe der gespeicherten Daten.

## 12. Neustart

Die Funktion `restart()` des `SoftwareManagers` löst den Software-Neustart aus, indem die Aufrufe der in den `AEnManagern` enthaltenen `restartAEn()` Funktionen die einzelnen Architekturelemente neu aufstarten.

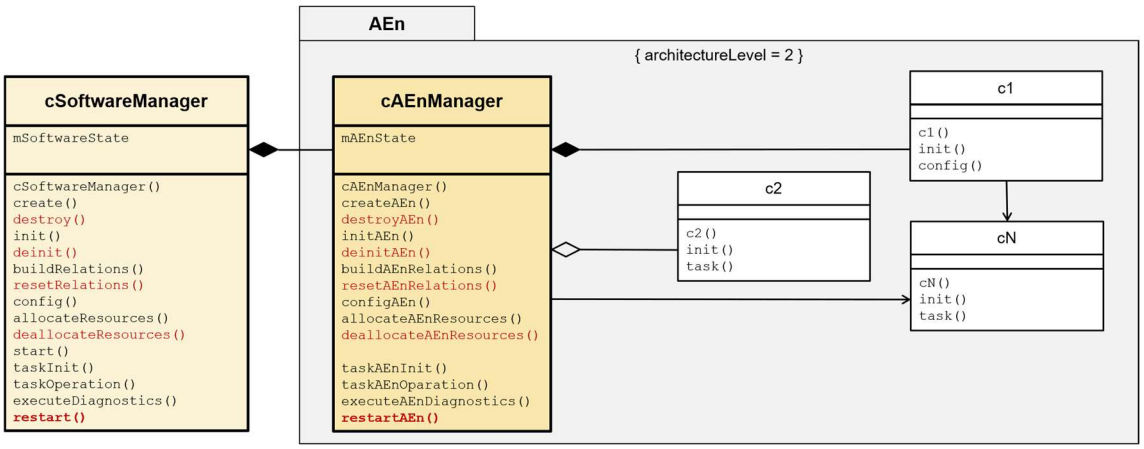


Bild 13: Manager Pattern um Neustart ergänzt

Zur feineren Unterteilung des Neustarts, aber auch des nachfolgend erklärten Herunterfahrens, dienen die gegenläufigen Funktionen zu denen, die die Manager während des Hochlaufs ausführen.

| Funktionen des Hochlaufs | Funktion des Herunterfahrens |
|---|---|
| Konstruktor<br>`construct()` | Destruktor<br>`destruct()` |

| | |
|---|---|
| `create()` | `destroy()` |
| `init()` | `deinit()()` |
| `buildRelations()` | `resetRelations()` |
| `allocateResources()` | `deallocateResources()` |

## 13. Herunterfahren

Die Funktion `shutdown()` des `SoftwareManagers` löst das Herunterfahren der Software aus, indem die Aufrufe der in den `AEnManagern` enthaltenen `shutdownAEn()` Funktionen die einzelnen Architekturelemente herunterfahren.

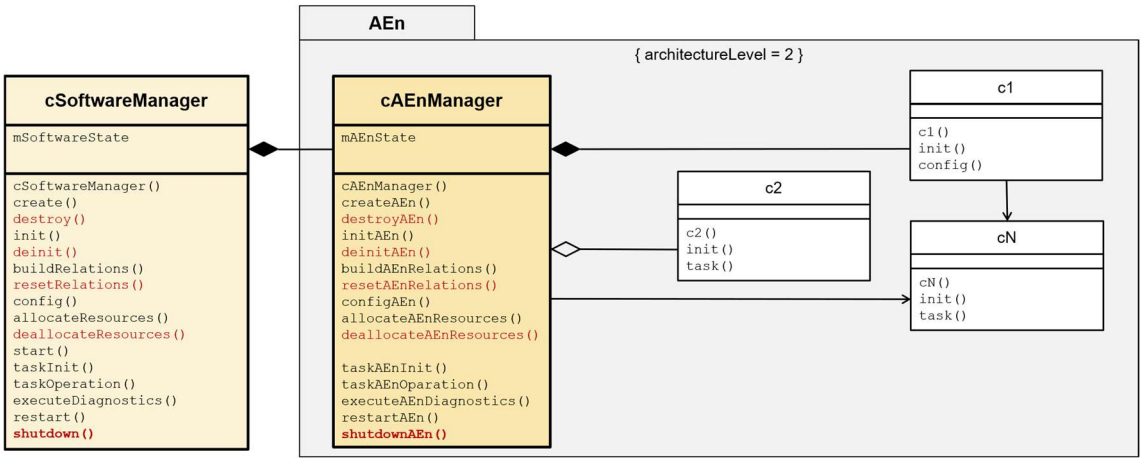


Bild 14: Manager Pattern um Herunterfahren ergänzt

## 14. Zentrale Fehlerbehandlung

Die Klassen `c1…cN` bzw. Module `m1...mN` erkennen ihre Fehler und melden diese ihrem `AEnManager`. Dieser leitet die Fehler direkt an den übergeordneten `SoftwareManager` weiter. Der `SoftwareManager` übernimmt zentral die Fehlerbehandlung, auch für potentiell auftretende Fehler in den `AEnManagern`.

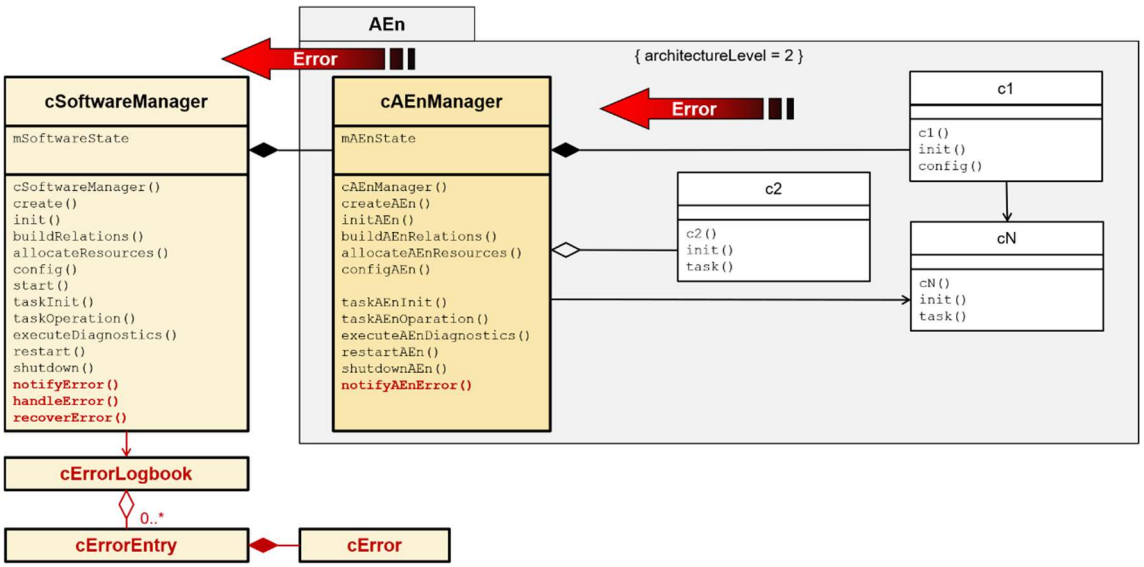


Bild 15: Manager Pattern um zentrale Fehlerbehandlung ergänzt

Neben der richtigen Fehlerreaktion, z.B. mittels `handleError()` und `recoverError()`, übernimmt der `SoftwareManager` auch den Fehlereintrag in sein `ErrorLogbook`.

## 15. Dezentrale Fehlerbehandlung

Die Klassen `c1…cN` bzw. Module `m1...mN` erkennen ihre Fehler und melden diese ihrem `AEnManager`. Kann der `AEnManager` den Fehler bearbeiten, tut er dies und trägt den Fehler in sein lokales `ErrorLogbook` ein.

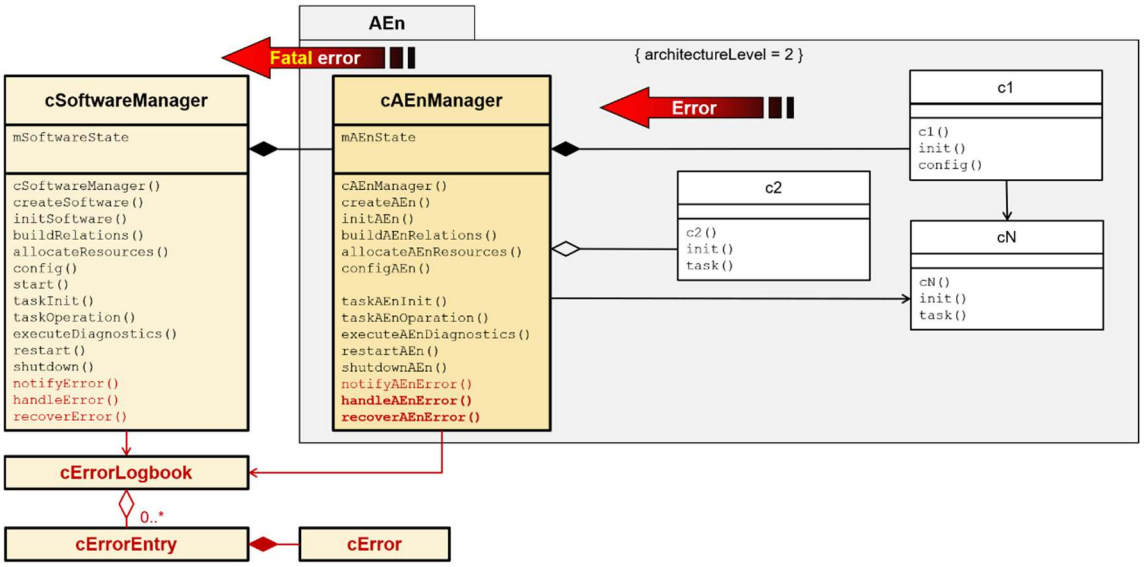


Bild 16: Manager Pattern um dezentrale Fehlerbehandlung ergänzt

Fatale Fehler, die der `AEnManager` nicht selbst bearbeiten kann, leitet er an den übergeordneten `SoftwareManager` weiter. Dieser übernimmt die Fatal-Fehlerbehandlung und trägt diese in sein `ErrorLogbook` ein.

## 16. Manager Zustandsfolge-Automat

Der `SoftwareManager` Zustandsfolge-Automat repräsentiert die möglichen Zustände der gesamten Software bzw. des Systems. Der `AEnManager` Zustandsfolge-Automat repräsentiert die möglichen Zustände des einzelnen Architekturelements.

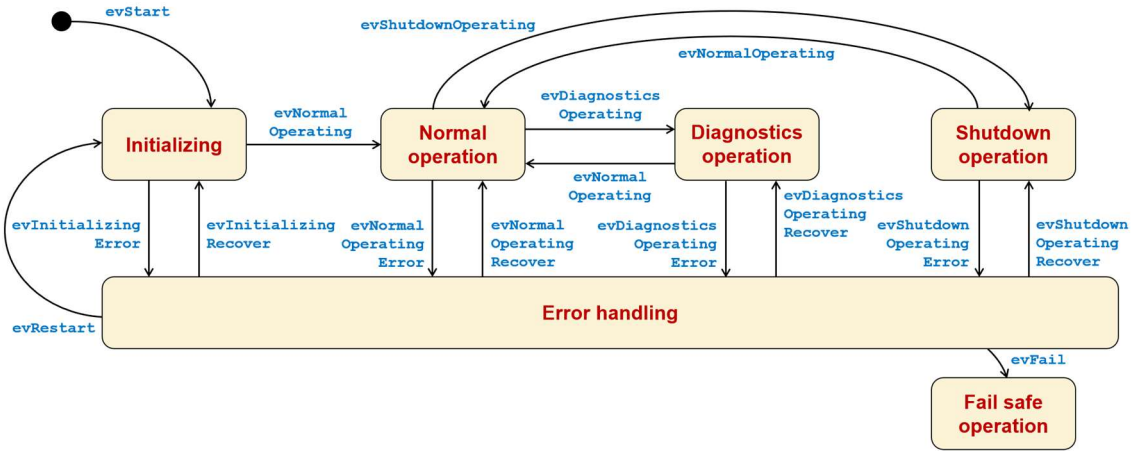


Bild 17: Manager Zustandsfolge-Automat

## 17. Manager Ausführungsfluss

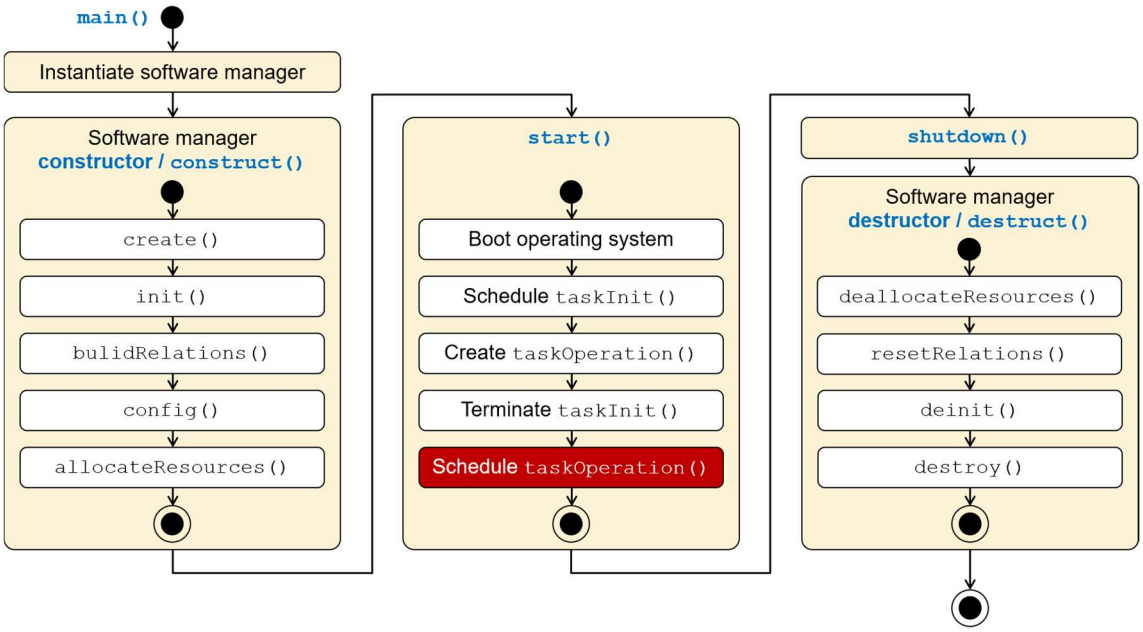


Bild 18: Manager Ausführungsfluss auf oberster Ebene

Als Zwischenzusammenfassung ist hier der Ausführungsablauf der einzelnen Funktionen auf Ebene des `SoftwareManagers` dargestellt. Dahinter verbergen sich die delegierenden Aufrufe der `AEnManager`. Dies ist nur eine Ablaufvariante von vielen möglichen.

## 18. Diskussionspunkte und Verbesserungen

Das bisher vorgestellte Manager Pattern lässt sich durch weitere Design- und Implementierungsdetails, aber auch Varianten verändern und ergänzen. Im Folgenden werden hierzu ein detaillierteres Grundkonzept und ein detaillierteres erweitertes Konzept vorgestellt.

## 19. Detailliertes Grundkonzept mit Fehlernotifikation

Im `Common::Management` sind das Fehler-Notifikationsinterface `icbcErrorHandler` (icbc == **I**nterface **C**all**b**ack **C**lass) und weitere Klassen für Diagnose und Fehlerbehandlung enthalten. Diese Elemente werden gleichermaßen in allen Managern verwendet.

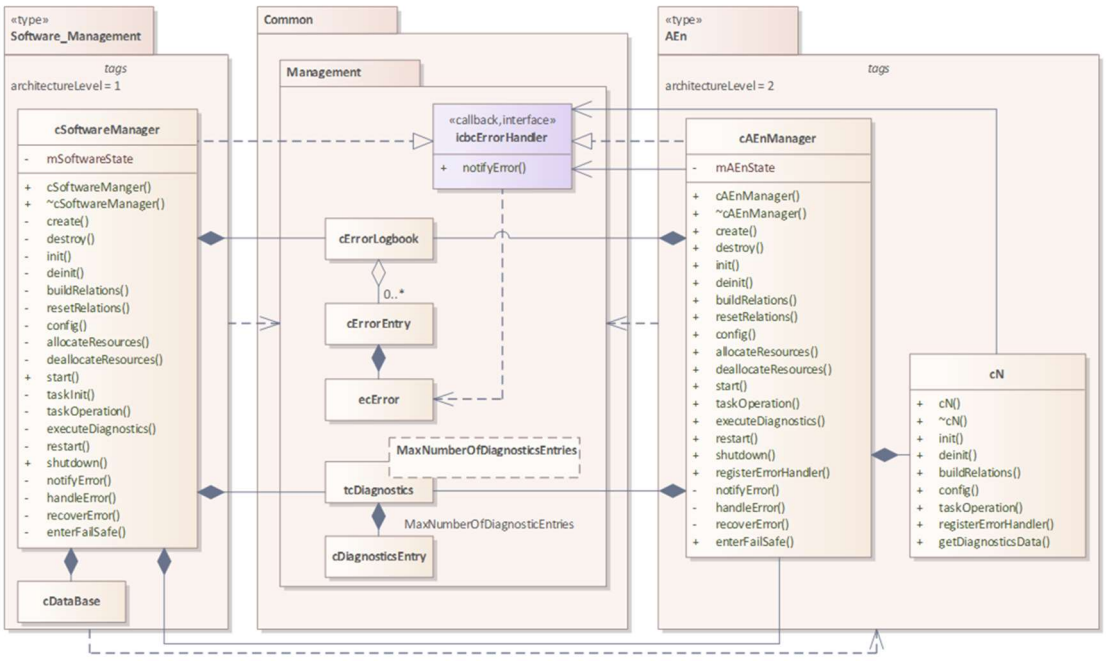


Bild 19: Detailliertes Grundkonzept mit Fehlernotifikation

Das Applikationselement `cN` notifiziert über die `notifyError()` Funktion des Interfaces `icbcErrorHandler` die Fehler an den `cAEnManager`. Der `cAEnManager` wiederum notifiziert seine Fehler über das gleiche Interface an den `cSoftwareManager`.

Bei Anwendung der Komposition erfolgt die Objektinstanziierung als eingebettete Elemente/ Attribute:

```
class cSoftwareManager : public Common::Management::icbcErrorHandler
{
  //…
  private:
    //…
    tcDiagnostics<mMaxNumberOfDiagnosticsData> mDiagnostics;
    cErrorLogbook mErrorLogbook;
    cAEnManager mAEnManager;
};
```

Die Aufrufe der delegierten Funktionen müssen hier einzeln erfolgen, da keine gemeinsamen Typen definiert sind:

```
void cSoftwareManager::init(void)
{
  mAEnManager.init();
}
```

Der exemplarische C++ Programmcode des detaillierten Grundkonzepts ist im Download [1] enthalten.

## 20. Detailliertes erweitertes Konzept mit Fehlernotifikation und Manager-Interface

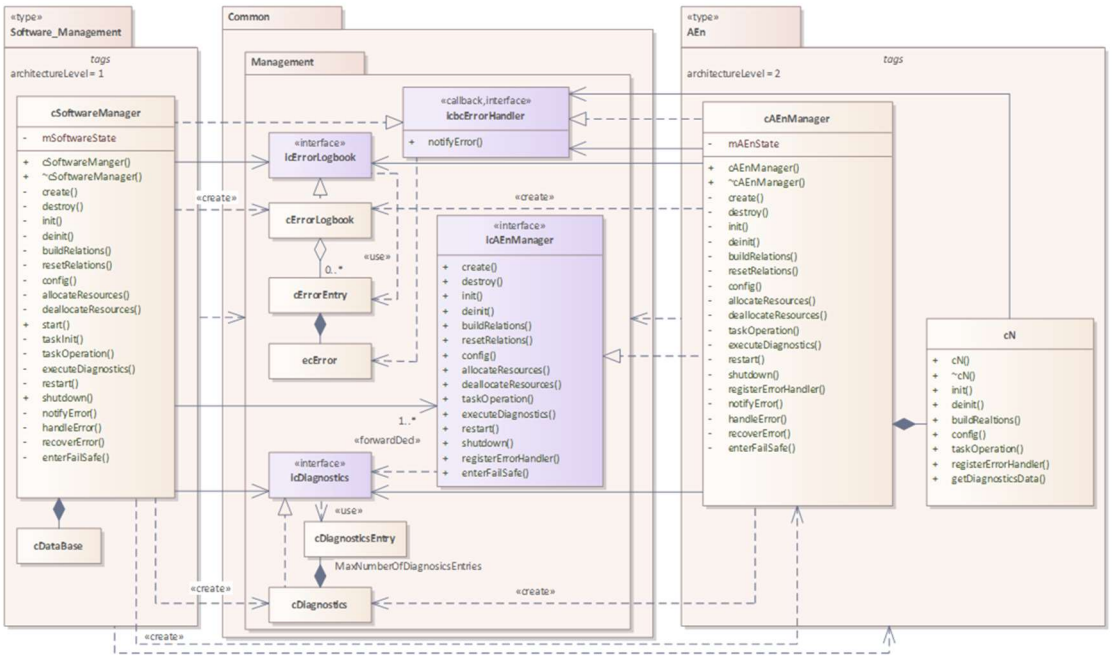


Bild 20: Detailliertes erweitertes Konzept mit Fehlernotifikation und Manager-Interface

Bei Anwendung der Assoziation/ Aggregation erfolgt die Objektinstanziierung dynamisch und der Zugriff über Zeiger:

```
class cSoftwareManager : public Common::Management::icbcErrorHandler
{
  //…
  private:
    //…
    icDiagnostics* mPtrDiagnostics;
    icErrorLogbook* mPtrErrorLogbook;
    std::array<icAEnManager*, mMaxNumberOfAEnManager>
                                                    mAEnManagerContainer;
};
```

Die Aufrufe der delegierten Funktionen erfolgen in einer Schleife (einfach erweiterbar), da alle `cAEnManager` den gemeinsame Interfacetype `icAEnManager` (ic == Interface Class) implementieren und der `cSoftwareManager` über ein Zeigerarray darauf zugreift [5-7]:

```
void cSoftwareManager::create(void)
{
  mPtrDiagnostics  = new cDiagnostics{ mObjectID_Diagnostics, this };
  mPtrErrorLogbook = new cErrorLogbook{ mObjectID_ErrorLogbook, this };

  mAEnManagerContainer[mAEnManagerID] =
                              new AEn::cAEnManager{mObjectID_AEnManager};
  // add other AEnManager objects here

  // for all AEnManager objects call ...
  for (auto&& refElement : mAEnManagerContainer)
  {
    if (refElement != nullptr)
    {
      refElement->create();
    }
  }
}
```

Der exemplarische C++ Programmcode des detaillierten erweiterten Konzepts ist im Download [1] enthalten.

## 21. Resümee

In der Praxis muss jede Embedded-Software die hier vorgestellten verteilten und zentral zu koordinierenden Aufgaben ausführen. Anforderungsabhängig sind weniger oder mehr dieser Aufgabenart auszuführen. Um die konzeptionelle Integrität auch über mehrere Projekte zu wahren, eignet sich der Einsatz von Softwarepatterns. Wie bei jedem Pattern muss der Softwarearchitekt, der Softwareentwickler bzw. das Softwareteam die hier exemplarisch vorgestellten Strukturen und Abläufe auf die individuellen Gegebenheiten in der Software anpassen.

## 22. Referenzierte und weiterführende Links


[1] MicroConsult-Download für diesen Beitrag, komplett und aktuell
http://download.microconsult.net/ese2023/manager-pattern.zip

[2] Object Management Group (OMG) - Unified Modeling Language (UML) Standard
www.uml.org

[3] Literaturhinweis: Design Patterns
Autoren: Erich Gama, Richard Helm, Ralph Jonson und John Vlissides
ISBN-13: 978-0201633610

[4] Manager Design Pattern
https://www.eventhelix.com/design-patterns/manager

[5] MicroConsult-Download: Dynamisch versus statische Polymorphie mit C++
http://download.microconsult.net/ese2022/polymorphism.zip

[6] MicroConsult-Download: Port-Designs und ihre Implementierungsansätze
http://download.microconsult.net/ese2021/port-designs.zip

[7] MicroConsult-Download: Interface-Designs und ihre Implementierungsansätze
http://download.microconsult.net/ese2020/interface-designs.zip

[8] **MicroConsult-Trainings – auch Live-Online**:

Requirements Engineering und Management für Embedded-Systeme

Software-Architekturen für Embedded- und Echtzeitsysteme

Embedded C++: Objektorientierte Programmierung für Mikrocontroller mit C++/EC++

Embedded C++ für Fortgeschrittene: Objektorientierte Programmierung für Mikrocontroller mit C++/EC++

Embedded-Software-Design und Patterns mit C

www.microconsult.de


## 23. 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