metaSingleton

Technical Concept Paper

Reto Carrara
Gyrenstrasse 17
CH-8967 Widen
+4179/328’17’05
www.carrara.ch
info@carrara.ch

Inhalt

Revision

Datum	Version	Autor	Änderungen
21.01.02	1.0	rc	Basisversion
15.02.02	1.1	rc	Layout

Einführung

Der Singleton ist wohl jenes Design Pattern, welches dank seiner Einfachheit zuerst verstanden wird[1]. Trotzdem ist der Klassiker[2] noch nicht ausgereift für die Anwendung in komplexen Projekten.

Der vorgestellte Ansatz bietet eine wieder verwendbare Singleton-Mechanik basierend auf C++-Templates. Nebst einer einfachen und intuitiven Anwendung unterstützt das vorgeschlagene Konzept auch die Spezialisierung von Singletons. Gerade in komplexeren Software-Systemen ist es immer wieder nötig, die Funktionalität eines Singletons durch Vererbung anzupassen.

[1] Und aus demselben Grund auch immer wieder exzessiv eingesetzt wird.
[2] siehe dazu Design Patterns: E. Gamma et al.

Der Klassiker

Der Klassiker, wie er in Design Patterns von Erich Gamma et al beschrieben wird, legt eine Referenz auf das per Definition einzig existierende Objekt in einer statischen Klassenvariablen ab.

```
class UserSingleton
{
  public:

    static UserSingleton& instance()
    {
      if (!sm_paSingleton)
      {
        sm_paSingleton = new UserSingleton();
      }
      return *sm_paSingleton;
    }

    const char* getData() const
    {
      return "Singleton's Data";
    }

  private:

    UserSingleton()
    : m_pcData ("Singleton's data...")
    {}

    const char* m_pcData;
    static UserSingleton* sm_paSingleton;
};
```

Die Klassenvariable existiert, wie der Name bereits impliziert, nur einmal pro Klasse. Dies ermöglicht es, mit dem Klassennamen das Singleton-Objekt zu dieser Klasse zu finden:

```
UserSingleton& raUserSingleton = UserSingleton::instance();
cout << raUserSingleton.getData() << endl;
```

Wichtig zu bemerken ist, dass das Singleton-Objekt beim ersten Aufruf von instance() kreiert wird. Ein unerwünschtes Kreieren des Objektes ist nicht möglich, da der Konstruktor als privat deklariert wurde. Nur eine Methode der Klasse UserSingleton selbst kann somit ein Objekt instanzieren. Diese Vorgehensweise garantiert, dass nur genau ein Objekt von UserSingleton existieren kann.

Was ist nun aber, wenn die Klasse einer fremden Bibliothek als Singleton dienen sollte? Oder wie wird das Problem gelöst, falls ein Objekt wohl als Singleton angesprochen werden soll, zahlreiche andere Objekte aber ebenfalls Existenzberechtigung haben, aber keine Singletons sein sollen?

Diese Probleme lassen sich lösen, indem der Singleton die entsprechenden Objekte aggregiert. Trotzdem fällt unangenehm auf, dass die komplette Mechanik (die Methode instance() und die statische Variable) immer wieder aufs neue programmiert werden muss.

Hinzu kommt ein weiteres Problem, dass sich vor allem in komplexeren Software-Systemen immer wieder findet. Um das Problem zu erläutern, sei als Beispiel eine Kommunikationslibrary angeführt[3].

Gegeben sei ein Subsystem A, welches prozess-interne Meldungen verschicken kann. Um die Meldungen an einen Adressat zu schicken, wird ein Dispatcher verwendet, welcher die logische auf die physikalische Adresse übersetzt. Die physikalische Adresse ist in diesem Fall eine Referenz auf eine Meldungs-Queue.

Zu einem späteren Zeitpunkt wird Subsystem B entwickelt, welches Meldungen auch über Netzwerk zu senden vermag. Natürlich sollte Transparenz herrschen, wenn es darum geht, prozess-interne Meldungen oder solche über Netzwerk zu verschicken.

Nun muss aber der Netzwerk-Dispatcher ganz andere physikalische Adressen verwalten. Nicht das Senden an eine Message-Queue, sondern das Schreiben von Daten auf einen Netzwerktreiber steht an. Ein Dispatcher für System B muss also sowohl die neuen Anforderungen von B, aber auch die von A erfüllen.

Hier bietet es sich an, den Singleton zu spezialisieren. Dies ist aber mit dem „Klassiker“ nicht möglich, da die instance()-Methode die Instanzierung selbst vornimmt. Per Definition darf die Basisklasse natürlich auch nicht über die abgeleiteten Klassen bescheid wissen, so dass die Instazierung einer Sub-Klasse in instance() nicht möglich ist.

Die Lösung des Problems liegt in einem der wichtigsten Paradigmen der Objekt-Orientierten Technologie:

> Def: Für einen optimalen Design muss die Mechanik von den Daten getrennt werden.

Unter Daten verstehen wir die User-Daten inklusive aller dazu gehörenden Methoden. Die Mechanik umfasst den Zugriff auf die einzig existierende Instanz mit der instance()-Methode und die statische Referenz.

[3] Das angeführte Beispiel ist nur zu Erklärungszwecken gedacht.

Die Trennung von Daten und Mechanik mittels Templates

Mit Hilfe von C++-Templates ist es möglich, typen-bezogenen Code zu generieren. Interessant ist dabei, dass ein Template, welches mit unterschiedlichen Typen instanziert wird, komplett verschiedene Klassen ergibt.

Def.: Man spricht von einer Template-Instanz, wenn ein Template korrekt parametrisiert wurde und somit als Klasse verwendet werden kann.

list<A> und list<B> sind beides Instanzen vom Template list. aAList ist eine Instanz der Klasse list<A>. list<A> und list<B> haben keine Verwandtschaftsbeziehung.

```
list<A> aAList;
list<B> aBList;
```

Eine in einem Template deklarierte statische Variable muss daher für jede Template-Instanz neu angelegt werden.

```
template <class T>
class Singleton
{
  public:
    Singleton(T& i_raSingleton)
    {
      sm_paSingleton=&i_raSingleton;
    }

    static T& instance()
    {
      if (!sm_paSingleton)
      {
        cout << "Error: Singleton has not been initialized!" << endl;
      }
      return *sm_paSingleton;
    }

  private:
    static T* sm_paSingleton;
};
```

Obiger Code bietet nur die Möglichkeit respektive die Mechanik, um eine beliebige Klasse T zum Singleton zu erklären.

Die Instanzierung eines Objektes der Klasse T liegt dabei nicht in der Verantwortung der Singleton-Mechanik. Im Konstruktor kann aber eine Referenz auf ein bereits bestehendes Objekt übergeben werden, das als Singleton verfügbar sein soll.

Soll also das Objekt einer Klasse UserData zum Singleton gemacht werden, so muss dieses zuerst auf herkömmlichem Wege instanziert und dann als Singleton deklariert werden.

```
UserData aUserData("Singleton's data...");
```

```
Singleton<UserData> aUserDataSingleton(aUserData);
```

Auch die Anwendung ist denkbar einfach:

```
void f()
{
  cout << Singleton<UserData>::instance().getData() << endl;

}

void Test()
{
  UserData aUserData("Singleton's data...");
  Singleton<UserData> aUserDataSingleton(aUserData);

  f();
}
```

Wichtig zu bemerken ist, dass ein Programm nun natürlich abstürzt, falls ein Singleton nicht vorgängig geladen wurde, da in der Methode instance() keine automatische Instanzierung mehr stattfindet. Falls diese Funktionalität gewünscht wird, kann sie natürlich weiterhin verwendet werden.

```
static T& instance()
{
  if (!sm_paSingleton)
  {
    sm_paSingleton = new T();
  }
  return *sm_paSingleton;
}
```

Programmierer von Echtzeit-Systemen wissen aber, dass obige Vorgehensweise zu nicht deterministischem Verhalten führt. Vielmehr muss darauf geachtet werden, dass alle Ressourcen bereits während der Initialisierungs-Phase eines Programmes angefordert werden. So auch sämtliche Singleton-Objekte. Die gezeigte Methode unterstützt dieses Paradigma, da die Instanzierung eines Objektes komplett von der instance()-Methode für den Singleton-Zugriff getrennt wurde.

Dank dem C++-Keyword friend ist es auch weiterhin möglich, den Konstruktor der User-Klasse privat zu deklarieren, so dass der Applikationsprogrammierer nicht aus Unwissenheit ein als Singleton gedachtes Objekt instanziert.

Werden zum Beispiel alle Singletons in einer Klasse Init erzeugt, so könnte diese als friend deklariert werden:

```
class UserData
{
  public:

    const char* getData() const
    {
      return m_pcData;
```

```
    }

  private:

    friend Init;

    UserData(const char* i_pcData)
    : m_pcData (i_pcData)
    {}

    const char* m_pcData;
};
```

Zu bemerken bei der gezeigten Lösung ist auch, dass es nun auch möglich ist, vererbte Singletons zu verwenden. Nachfolgend sei dies erläutert.

Angenommen die Klasse UserDataDrv ist von UserData abgeleitet und somit referenzkompatibel. Instanzieren wir das Singleton-Template mit dem Typ UserData, so kann im Konstruktor natürlich auch die spezialisierte Klasse UserDataDrv mitgegeben werden:

```
void f()
{
  cout << Singleton<UserData>::instance().getData() << endl;

}

void Test()
{
  UserDataDrv aUserData("Singleton's data...");
  Singleton<UserData> aUserDataSingleton(aUserData);

  f();
}
```

Die Applikation selbst greift aber nach wie vor über dasselbe Statement auf den Singleton (und zwar immer noch über den Typ der Basis-Klasse) zu. Ist die getData Methode virtuell, so wird nun diejenige von UserDataDrv aufgerufen.

Natürlich lässt sich das Austauschen von Funktionalität auch sehr flexibel mit einem Bridge-Pattern[4] bewerkstelligen. Die zusätzlich eingeführte Indirektion kann aber gerade bei sehr zeitkritischen Systemen vermieden werden, indem die oben gezeigte Variante mit dem direkten Austauschen der Objekte angewendet wird.

[4] Auch Handle-Body-Idiom

Ein Singleton mit wohldefiniertem Lebenszyklus

Betrachtet man den Lebenszyklus eines Standard-Singletons, so stellt man eine Asymmetrie bezüglich Kreieren uns Destruieren fest. Das Kreieren geschieht nämlich beim ersten Aufruf der instance()-Methode. Das Destruieren hingegen fehlt komplett. Da das Singleton-Objekt lediglich durch einen statischen Pointer referenziert wird, hat der Compiler auch keinen Grund, zu irgendeinem Zeitpunkt den Destructor aufzurufen.

Dieser Umstand kann sehr wohl zu Problemen führen, wenn es sich beim Singleton um einen Hardware-Treiber, eine Datenbankschnittstelle oder ähnliches handelt, welche zum korrekten Funktionieren einen Close-Befehl benötigen.

Gerade im Bereich der Embedded-Systems sind folgende Punkte unerlässlich:

- ?? Alle Ressourcen werden zu Beginn eines Programmes akquiriert
- ?? Alle Ressourcen werden am Ende eines Programmes wieder zurückgegeben

Nur ein genaues Einhalten dieser Regeln lässt am Schluss eine Aussage darüber machen, in welchem Status sich ein System befindet (Memory Leaks ✍ siehe dazu auch Artikel „leakHunter“, Kommunikationsschwierigkeiten, korrupte Datenbestände etc.). Ein System muss also genauso sicher heruntergefahren werden können, wie es hochgefahren wurde.

> Def: Im korrekten und kontrollierten Herunterfahren widerspiegelt sich die Qualität einer Software.

Derselben Logik unterliegen die Singletons. Ein Singleton bietet lediglich einen Zugriffspunkt auf das einzige Objekt seiner Art. Dass dieses Objekt daher nie mehr gelöscht werden darf, ist ein Trugschluss.

Betrachten wir dazu den Startup einer Software. In C/C++ wird immer als erstes die Methode main aufgerufen. In der main()-Methode werden nun andere Sub-Routinen oder Objekt-Methoden aufgerufen.

```
main()
{

  // Call Subroutine
  f();

  return 0;
};
```

Ist die Methode f vollständig ausgeführt, so kehrt das Programm in die main()-Methode zurück und terminiert.

Dieser Aussage gilt nicht, falls in f() parallele Entitäten, sogenannte Threads, abgespaltet werden und deren Terminierung nicht korrekt synchronisiert wurde. Falls dies geschieht, handelt es sich um Code von schlechter Qualität, da das Hauptprogramm nicht weiss, ob noch Rechenvorgänge am laufen sind.

Geht man davon aus, dass mit dem Terminieren der Funktion f() alle anderen (auch parallelen) Aktivitäten korrekt beendet wurden, so lassen sich danach auch sämtliche existierenden Singletons sicher abräumen.

```
main()
{
  UserData aUserData("Singleton's data...");
  Singleton<UserData> aUserDataSingleton(aUserData);

  // Call Subroutine
  f();

  return 0;
};
```

Mit dem return-Statement geht sowohl die Variable aUserDataSingleton als auch aUserData „out of scope“ und zwar in der genannten Reihenfolge. Das heisst, dass die Destruktoren der Objekte aufgerufen werden.

Der Destruktor von Singleton setzt in erster Linie den statischen Pointer auf 0. Der Destruktor von UserData ist benutzerdefiniert.

Nebst der gezeigten Variante wäre es auch möglich, einen Pointer auf ein Objekt zu übergeben.

```
main()
{
  Singleton<UserData> aUserDataSingleton(new UserData(“....”);

  // Call Subroutine
  f();

  return 0;
};
```

In diesem Singleton würde natürlich ein Memory-Leak hinterlassen, wenn man die Problematik nicht gesondert behandelt. Man kann sich einen grossen Teil an Memory-Leaks ersparen, wenn folgende Regel eingehalten wird.

> Def: Übernimmt eine Methode einen nicht konstanten Pointer auf ein Objekt, so ist sie für dessen Löschen durch den Operator delete zuständig.

Der Code wurde angepasst, um obiger Regel Rechnung zu tragen:

```
template <class T>
class Singleton
{
  public:

    Singleton(T& i_raSingleton)
    : m_bDelete (false)
    {
      sm_paSingleton=&i_raSingleton;
```

```
    }

    Singleton(T* i_paSingleton)
    : m_bDelete (true)
    {
      sm_paSingleton=i_paSingleton;
    }

    ~Singleton()
    {
      if (m_bDelete) delete sm_paSingleton;
      sm_paSingleton=0;
    }

    static T& instance()
    {
      if (!sm_paSingleton)
      {
        cout << "Error: Singleton has not been initialized!" << endl;
      }
      return *sm_paSingleton;
    }

  private:

    bool m_bDelete;

    static T* sm_paSingleton;

};
```

Mit Hilfe eines booleschen Flags wird determiniert, ob ein Objekt beim Aufruf des Destruktors gelöscht werden soll oder nicht. Das Flag wird nur dann gesetzt, falls im Konstruktor ein Pointer anstelle einer Referenz übergeben wurde.

Reassignment für Singletons

Starten mehrere Software-Subsysteme nacheinander auf, so kann es Sinn machen, die Spezialisierung eines Singletons erst nach geglückter Initialisierung zu verwenden.

Man stelle sich ein Trace-System in einem grossen Netzwerk-Verbund vor.

Sobald die Software aufstartet, können erste Meldungen generiert werden. Diese sehr wichtigen Informationen könnten bei Embedded-Systems bei Bedarf auf eine Flashdisk geschrieben werden.

Sobald aber die Netzwerk-Kommunikation aufgebaut ist, werden alle Traces über Netzwerk an einen Host verschickt.

In diesem Beispiel wäre es möglich, den Trace –Service, realisiert als Singleton, zur Laufzeit durch eine Spezialisierung auszutauschen, sobald die Netzwerk-Kommunikation steht[5]. Genau gleich verhält es sich beim System-Shutdown. Auch wenn das Netzwerk heruntergefahren wird, können immer noch Trace-Meldungen generiert werden, welche nun wiederum auf die Flash-Disk geschrieben werden sollen.

Um obige Funktionalität sicherzustellen kann der Singleton mit einer Reassigment-Funktionalität ausgestattet werden, welche es erlaubt, ein neues Objekt zuzuweisen. Dabei wird die alte Referenz als Membervariable abgelegt und beim Aufruf des Destruktors wieder hergestellt,

```
template <class T>
class Singleton
{
  public:

    Singleton(T& i_raSingleton)
    : m_bDelete (false)
    {
      m_paOldSingleton=sm_paSingleton;
      sm_paSingleton=&i_raSingleton;
    }

    Singleton(T* i_paSingleton)
    : m_bDelete (true)
    {
      m_paOldSingleton=sm_paSingleton;
      sm_paSingleton=i_paSingleton;
    }

    ~Singleton()
    {
      if (m_bDelete) delete sm_paSingleton;
      sm_paSingleton=m_paOldSingleton;
    }

    static T& instance()
```

[5] Auch hier handelt es sich nur um ein Beispiel einer möglichen Anwendung.

```
    {
      if (!sm_paSingleton)
      {
        cout << "Error: Singleton has not been initialized!" << endl;
      }
      return *sm_paSingleton;
    }

  private:

    bool m_bDelete;
    T*   m_paOldSingleton;

    static T* sm_paSingleton;

};
```

Die Anwendung des vorgestellten Systems ist immer noch denkbar einfach:

```
void fSub()
{
  cout << Singleton<UserData>::instance().getData() << endl;

}

void f()
{
  cout << Singleton<UserData>::instance().getData() << endl;

  {
    // Reassign
    UserDataDrv aUserDataDrv("Singleton's NEW data...");
    Singleton<UserData> aUserData1Singleton(aUserDataDrv);

    fSub();
  }

  cout << Singleton<UserData>::instance().getData() << endl;

}

void Test()
{
  UserData aUserData("Singleton's data...");
  Singleton<UserData> aUserData1Singleton(aUserData);

  f();

}
```

Schlussüberlegungen

Das vorliegende Paper zeigt, wie die Mechanik für ein Singleton-Pattern mit Hilfe von C++-Templates extrahiert werden kann.

Durch Externalisierung des Konstruktions-Prozesses wird es möglich, die Lebenszeit von Singleton-Objekten genau zu kontrollieren. Diese Kontrolle ist unerlässlich, um eine gute Software-Qualität gewährleisten zu können.

Dank einer kleinen Modifikation wird es möglich, die Singleton-Funktionalität zur Laufzeit auszutauschen, indem ein neues Objekt geladen wird. Dank einer stack-ähnlichen Mechanik kann der alte Zustand wieder hergestellt werden, wenn das momentane Singleton-Objekt gelöscht wird.

Literatur

Gamma, Erich et al.: Design Patterns. Addison Wesley, 1995. ISBN 0-201-63361-2.
Ein Standardwerk, das in keiner Bibliothek fehlen sollte. Vor allem als Nachschlagewerk und Lebensgefährte gedacht. Das beste Buch auf seinem Gebiet! Sehr empfehlenswert für jedermann.

Stroustrup, Bjarne: The C++ Programming Language, Third Edition. Addison Wesley, 1997. ISBN 0-201-88954-4.
Die beste C++ Referenz. Achtung: Nur die dritte Edition kaufen. Exzellente Einführung in STL. Sehr empfehlenswert für jedermann.

Lippmann, Stanley B., Lajoie, Josée: C++ Primer, Third Edition. Addison Wesley, 1998. ISBN 0-201-82470-1.
Ebenfalls ein sehr gutes C++ Buch. Einfacher zu verstehen als das Buch Stroustrup. Auch hier: Nur die dritte Edition kaufen. Gute STL Referenz

Lippmann, Stanley B: Inside the C++ Object Model. Addison Wesley, 1996. ISBN 0-201-83454-5.
Ein geniales und einfach geschriebenes Buch, für alle die wissen wollen, wie C++ intern funktioniert. Vom "virtual call" bis hin zu "multiple inheritance RTTI".

Coplien, James O.: Advanced C++ Programming Styles and Idioms. . Addison Wesley, 1992. ISBN 0-201-54855-0.
Ein Buch mit weiterführenden Konzepten. Zum Teil hochtechnische und sehr schwierig zu verstehende Lösungsansätze. Die Perle unter den C++ Büchern.

Meyers, Scott: Effective C++, Second Edition. Addison Wesley, 1998. ISBN 0-201-92488-9.
Meyers, Scott: More Effective C++. Addison Wesley, 1998. ISBN 0-201-63371-X.
Zwei Bücher, die sich mit dem korrekten Schreiben von C++ Programmen befassen. So zum Beispiel, wie man Assignment Operators, Copy Constructors und deren mehr korrekt implementiert. Sehr empfehlenswert für jedermann.

Stroustrup, Bjarne: The Annotated C++ Reference Manual. Addison Wesley, 1995. ISBN 0-201-51459-1.
Die C++ Referenz für Compilerbauer. Geht sehr tief und ist mit vielen Bemerkungen versehen.