::Singleton

::Index

::Zweck

- Sicherstellung, dass nur eine Instanz einer bestimmten Klasse existiert
- Klasse mit globalen Zugriffspunkt, damit beliebig viele Klienten zugreifen können

::Motivation

- Es kann für bestimmte Anwendungen wichtig sein, dass es nur eine Instanz gibt
- Wie könnte man dies sicherstellen?
  -> z.B. durch globale Variable oder modifizierte Klasse
- Warum keine Globale Variable?
  -> Zugriff auf Objekt, aber keine Kontrolle über Erzeugung
  -> nicht genügend Informationen bei statischer Erzeugung
  -> globales Singleton wird erzeugt, auch wenn sie nicht benutzt werden

::Motivation

- Daher besser, wenn die Klasse die Verwaltung selbst übernimmt
- Damit kann Klasse die Erzeugung neuer Instanzen abfangen & Zugriffe gewähren

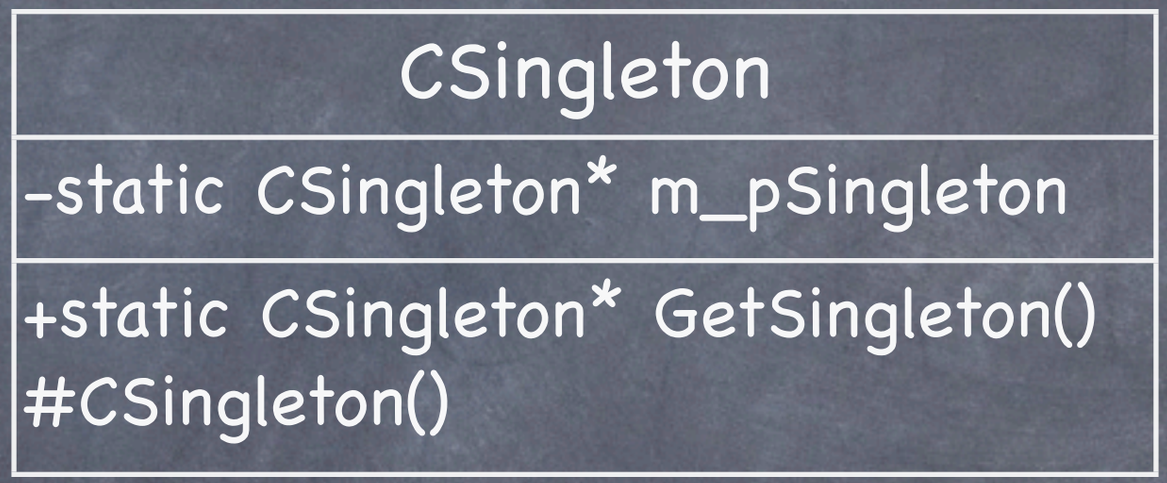
::Anwendbarkeit

- Verwendung des Singleton-Musters, wenn ...

1. es nur eine Instanz einer Klasse geben und es für Klienten an einem definierten Punkt zugreifbar sein muss

2. die einzige Instanz durch Bildung von Unterklassen erweiterbar sein soll und Klienten in der Lage sein sollen, die erweiterte Instanz ohne Veränderungen ihres Codes verwenden zu können

::Struktur


CSingleton
-static CSingleton* m_pSingleton
+static CSingleton* GetSingleton()
#CSingleton()

::Teilnehmer & Interaktionen

- Teilnehmer sind Singleton & Klienten
- Singleton definiert Methode (z.B. GetSingleton()), die es Klienten ermöglicht, auf die einzige Instanz zuzugreifen
- Singleton ist potentiell für Erzeugung & Zugriffsteuerung der einzigen Instanz zuständig
- Klienten erhalten nur Zugriff auf die Instanz über die "Singleton-Methode"

::Konsequenzen

Vorteile:

- Zugriffskontrolle auf das Singleton-Objekt
- eingeschränkter Namensraum
- Verfeinerung von Operationen & Repräsentationen
- variable Anzahl von Instanzen
- flexibler als Klassenoperationen

::Implementierung

1. Garantie einer einzigen Instanz
   (C++ & Java-Code)

2. Vererbung der Singletonklasse
   (C++-Code)

::Implementierung

1. Variante - Singleton via Konstruktor

   -> nicht möglich, da der Konstruktor erst nach der Erzeugung greift

::Implementierung

2. Variante - Singleton via Überladung von "new"

```
class CSingleton
{
   public:
      void *operator new(size_t stAllocateBlock);

   private:
      static CSingleton *m_pSingleton;
};

CSingleton* CSingleton::m_pSingleton = NULL;
```

::Implementierung

2. Variante - Singleton via Überladung von "new"

```
void *CSingleton::operator new (size_t stAllocateBlock)
{
   if(!m_pSingleton)
   {
      m_pSingleton = (CSingleton*)malloc (stAllocateBlock);
   }

   return m_pSingleton;
}
```

::Implementierung

2. Variante - Singleton via Überladung von "new"

```
int main()
{
   CSingleton *s = new CSingleton();  // Heap-Variable
   CSingleton *t = new CSingleton();
   CSingleton u;                      // Stack-Variable
   CSingleton v;

   printf("s: %p\nt: %p\n\n", s, t);  // Heap-Singleton
   printf("u: %p\nv: %p\n\n", &u, &v);// kein Singleton

   return 0;
}
```

::Implementierung

2. Variante - Singleton via Überladung von "new"

Ausgabe

```
s: 00320FE0       <- Heap-Variablen
t: 00320FE0

u: 0012FF08       <- Stack-Variablen
v: 0012FF04
```

::Implementierung

3. Variante - Singleton via Memberfunktion (Standard)

```
class CSingleton
{
   public:
            ~CSingleton() {}
      static CSingleton *GetSingleton();

   protected:
            CSingleton() {}

   private:
      static CSingleton *m_pSingleton;
};
```

::Implementierung

3. Variante - Singleton via Memberfunktion (Standard)

```
CSingleton* CSingleton::m_pSingleton = NULL;

CSingleton *CSingleton::GetSingleton()
{
   if(!m_pSingleton)
   {
      m_pSingleton = new CSingleton();
   }

   return m_pSingleton;
}
```

::Implementierung

3. Variante - Singleton via Memberfunktion (Standard)

```
int main()
{
   // CSingleton a;                          // Stack
   // CSingleton *b = new CSingleton();  // Heap
   //=>Compiler-Fehler: Kein Zugriff auf protected Element

   CSingleton *x = CSingleton::GetSingleton();
   CSingleton *y = y->GetSingleton();
   CSingleton *z = x->GetSingleton();

   printf("x: %p\ny: %p\nz: %p\n\n", x, y, z);
...
```

::Implementierung

3. Variante – Singleton via Memberfunktion (Standard)

Ausgabe

```
x: 00320FE0
y: 00320FE0
z: 00320FE0
```

::Implementierung

Probleme: Singleton (Standard)

```
class CSingleton
{
   public:
            ~CSingleton() {}
      static CSingleton *GetSingleton() {... = new ...}

   protected:
            CSingleton() {}

   private:
      static CSingleton *m_pSingleton;
};
```

::Implementierung

Problem: Konstruktor nur protected

```
class CSingleton2 : public CSingleton
{
   //...
};

int main()
{
   CSingleton2 *a = new CSingleton2();
   CSingleton2 *b = new CSingleton2();
   CSingleton2 c, d;
   printf("a: %p\nb: %p\nc: %p\nd: %p\n\n", a, b, &c, &d);
...
```

::Implementierung

Problem: Konstruktor nur protected

Ausgabe

```
a: 00321018
b: 00323190
c: 0012FF08
d: 0012FF04
```

Lösung

Konstruktor als private definieren
(Vorsicht beim Vererben!!)

::Implementierung

Problem: Kopiekonstruktor vergessen

```
int main()
{
   CSingleton *x = CSingleton::GetSingleton();

   CSingleton *a = new CSingleton(*x);
   CSingleton *b = new CSingleton(*x);

   printf("x: %p\n\n", x);
   printf("a: %p\nb: %p\n\n", a, b);

   return 0;
}
```

::Implementierung

Problem: Kopiekonstruktor vergessen

Ausgabe

```
        x: 00320FE0    <- Singleton-Objekt

        a: 00321018    <- Objekte durch Kopiekonstruktor
        b: 00323190
```

Lösung

Kopiekonstruktor als private definieren

::Implementierung

Problem: Speicherbereinigung (1. Lösung: Stack benutzen)

```
...
   public:
      static SingletonStatic* GetSingleton();
   private:
      SingletonStatic(const SingletonStatic&);
...

SingletonStatic* SingletonStatic::GetSingleton()
{
   static SingletonStatic singleton;
   return &singleton;
}
```

::Implementierung

Problem: Speicherbereinigung (2. Lösung: Wächterobjekt)

```
...
   private:
      ~CSingleton();

      class CGuard
      {
         public:   ~CGuard()
                   { if(CSingleton::m_pSingleton)
                      { delete CSingleton::m_pSingleton; } }
      };
      friend class CGuard;
...
```

::Implementierung

Problem: Speicherbereinigung (2. Lösung: Wächterobjekt)

```
CSingleton *CSingleton::GetSingleton()
{
     static CGuard guard;

     if(!m_pSingleton)
     {
        m_pSingleton = new CSingleton();
     }

     return m_pSingleton;
}
```

::Implementierung

Singleton in Java

```
public class CSingleton{
   private CSingleton(){
      //...
   }

   private static CSingleton m_Singleton=new CSingleton();

   public static CSingleton GetSingleton(){
      return m_Singleton;
   }
   ...
}
```

::Implementierung

Singleton in Java

```
public class CTestSingleton{

   public static void main(String args[]){
        // CSingleton a = new CSingleton();
        //=>The constructor CSingleton() is not visibles

        CSingleton.GetSingleton().DoIt();
        CSingleton.GetSingleton().x;
   }
}
```

::Implementierung

Vererbung der Singletonklasse

- statische Variante (mittels Umgebungsvariable)
- Template
- ((Macro))

::Implementierung

statische Variante (mittels Umgebungsvariable)

```
class CSingleton
{
   protected:
                CSingleton() {}
...

class CSingletonModified1 : public CSingleton
{
   friend class CSingleton;
   private:
                CSingletonModified2() {}
...
```

::Implementierung

statische Variante (mittels Umgebungsvariable)

```
CSingleton* CSingleton::GetSingleton()
{
   if(!m_pSingleton)
   {
      char *strg = getenv("SINGLETON");

      if(strcmp(strg, "Mod1")==0)
      { m_pSingleton = new CSingletonModified1(); }
      else
      { m_pSingleton = new CSingleton(); }
   }
...
```

::Implementierung

statische Variante (mittels Umgebungsvariable)

```
int main()
{
   // SET SINGLETON = "Sing" oder "Mod1"

   CSingleton *a = CSingleton::GetSingleton();

   CSingletonModified1 *b =
      (CSingletonModified1*) CSingleton::GetSingleton()

   printf("a: %p\nb: %p\n\n", a, b);
...
```

::Implementierung

statische Variante (mittels Umgebungsvariable)

Ausgabe

```
a: 00427E43
b: 00427E43
```

::Implementierung

Template (Stack-Singleton)

```
template <class Derived>
class CSingleton
{
   public:
      static Derived*   GetSingleton();

   protected:
                        CSingleton() {};

   private:
                        CSingleton(const CSingleton&);
};
```

::Implementierung

Template (Stack-Singleton)

```
template <class Derived>
Derived* CSingleton<Derived>::GetSingleton()
{
   static Derived singleton;

   return &singleton;
}
```

::Implementierung

Template (Stack-Singleton)

```
class ChildA : public CSingleton<ChildA>
{
   friend class CSingleton<ChildA>
   private:
           ChildA() {};
           ChildA(const ChildA&);
};

class ChildB : public CSingleton<ChildB>
{
   friend class CSingleton<ChildB>
...
```

::Implementierung

Template (Stack-Singleton)

```
int main()
{
   ChildA *a = ChildA::GetSingleton();
   ChildA *b = ChildA::GetSingleton();

   ChildB *c = ChildB::GetSingleton();
   ChildB *d = ChildB::GetSingleton();

   printf("a: %p\nb: %p\n\n", a, b);
   printf("c: %p\nd: %p\n\n", c, d);

...
```

::Implementierung

Template (Stack-Singleton)

Ausgabe

```
        a: 00427E43    <- ChildA-Singleton
        b: 00427E43

        c: 00427E41    <- ChildB-Singleton
        d: 00427E41
```

::Implementierung

Macro (keine Vererbung)

```
#define DEC_SINGLETON(name)                    \
   public:                                     \
      static  name*   GetSingleton();          \
private:                                       \
   static  name*   m_pSingleton;               \
                   name() {}                   \
                   name(const name&);          \
```

::Implementierung

Macro (keine Vererbung)

```
#define DEF_SINGLETON(name)          \
name* name::m_pSingleton = NULL;     \
name *name::GetSingleton()           \
{                                    \
   if(!m_pSingleton)                 \
   {                                 \
      m_pSingleton = new name();     \
   }                                 \
   return m_pSingleton;              \
}                                    \
```

::Implementierung

Macro (keine Vererbung)

```
class CSingleton
{
   DEC_SINGLETON(CSingleton)

   public:
      void Print() { printf("CSingleton\n"); }
   ...
};

DEF_SINGLETON(CSingleton)
...
```

::Implementierung

Macro (keine Vererbung)

```
void task5()
{
   CSingleton *a = CSingleton::GetSingleton();
   a->Print();
}
```

::Verwendung

- Druckerspooler
  (alle Ausdrucke werden über eine Instanz einer Klasse gesteuert)
- Audioausgabe
  (begrenzte Zahl von DACs, meist genau einer)
- Dateizugriff
  (z.B. Zugriff mehrerer Objekte auf eine INI-Datei)
- Datenbankzugriff
  (Zugriff mehrerer Objekte auf eine Datenbank)

::Verwandte Muster

- mit Hilfe vom Singleton-Muster können andere Muster implementiert werden
- z.B. Abstrakte Fabrik, Erbauer, Prototyp

::Erzeugungsmuster

- zwei Möglichkeiten, ein System mit den Klassen von ihm erzeugter Objekte zu parametrisieren

::Erzeugungsmuster

1. Möglichkeit:
   - Unterklasse der Klasse erstellen, welche die Objekte erzeugt (Fabrikmuster)
   - Nachteil:
     -> man muss neue Unterklasse erzeugen, um Klasse des Produkts zu ändern
     -> dies kann sich kaskadierend fortpflanzen

2. Möglichkeit:
   - Anwendung von Objektkompositionen
   - Objekt definieren, welches für Wissen zu ständig ist
   - Objekt zu Parametersystem machen

::Erzeugungsmuster

- zentraler Aspekt bei Abstrakte Fabrik, Erbauer & Prototyp
  -> alle 3 führen zur Erzeugung eines neuen
  Fabrikobjekts, welche Produktobjekte erzeugt

-> es hängt von verschiedene Faktoren ab, welches
Muster geeigneter ist

::Glossar

Attribut (engl. member data, member variable)

- Variable als Datenbestandteil einer Struktur oder Klasse

Methode (engl. member function)

- kann nur für ein Objekt der Klasse ausgerufen werden
- Methoden legen das Verhalten der Objekte fest
- die Methode hat Zugriff auf die Attribute des Objektes und auf Klassenvariablen
- nichtkonstante Methoden können den Objektzustand ändern

::Glossar

Klasse (engl. class)

- zusammengesetzter Typ aus Attributen und Methoden zu deren Bearbeitung
- durch Einschränkung der Zugriffsrechte kann eine Kapselung erreicht werden
- Variablen eines Klassen-Typs werden als Objekte bzw. Instanzen bezeichnet

Klassenmethode (engl. static member function)

- Methode kann auch ohne Objekt aufgerufen werden
- statische Membermethoden können nicht "virtual" sein und nur auf statische Membervariablen zugreifen

::Glossar

Klassenvariable (engl. static member data)

- Variable im Namensraum der Klasse
- ist nur einmal für die gesamte Klasse, unabhängig von der Existenz von Objekten, vorhanden
- alle Objekte greifen auf dieselbe Variable zu

Konstruktor (engl. constructor)

- zur Klasse gehörende Erzeugerfunktion für ein Objekt
- legt Objekt an & belegt dessen Attribute mit Werten
- eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Parameterlisten besitzen
- der ohne Parameter wird Standardkonstruktor genannt

::Glossar

Objekt (engl. object, instance variable)

- Variable eines Klassentyps / Instanz einer Klasse
- ein Objekt belegt alle in der Klasse definierten Attribute mit individuellen Werten (Identität)
- kann die in Klasse deklarierten Methoden ausführen
- Attributwerte repräsentieren Zustand des Objektes

Zeiger (engl. pointer)

- Typ, dessen Objektwerte Speicheradressen von (anderen) Objekten sind
- über Zeiger sind Verweise & Zugriffe auf Daten möglich
- Zeiger sind an Typ gebunden, der zu Daten passen muss

::Glossar

Heap

- Teil des Speichers, der längerfristig zur Verfügung steht
- man hat Einfluss auf Lebensdauer (new/delete)
- Heap kann wachsen (Auslagerung auf Festplatte)
- verzögert Ausführung von Programmen stark
- z.B. in C++: int *i = new int;

Stack

- Stapelspeicher mit LIFO-Eigenschaft (last in, first out)
- Stack ist schneller als Heap
- man hat keinen Einfluss auf Lebensdauer (kurzlebig)
- ist normalerweise in der Größe limitiert
- z.B. in C++: int i;