Inhaltsverzeichnis

1. LOGGING

1.1. Ziele

☑ Modernes, flexibles Logging-, Protokollierungssystem basierend auf Design-pattern.

☑ Quellen:

- ☐ http://de.wikipedia.org/wiki/Singleton_%28Entwurfsmuster%29
- ☐ http://www.codeproject.com/Articles/288827/g2log-An-efficient-asynchronous-logger-using-Cplus

1.2. LOGGING: Grundlagen

Um **für die gesamte Applikation ein einheitliches Protokollierungs- bzw. Loggingsystem** zu erstellen, kann man sich eines sehr bekannten Entwurfs-Musters bedienen. Das Singleton-Muster.

Man möchte also für die gesamte Applikation ein einziges LOGGING-Objekt nutzen, das z.B. auf folgende Weise genutzt werden kann.

```
Logger logger= new Logger();
...
logger.log(INFO, "Data stored succesfully.");
logger.log(ERROR, "Wrong Input value.");
logger.log(FATAL, "Memory segmentation fault.");
...
```

Anforderungen:

1. Es sollte nur EIN logger-Objekt möglich sein (vgl. Druckerspooler)
2. Parallele Threads sollten gleichzeitig das logger-Objekt nutzen können
3. Das Logging sollte asynchron erfolgen, d.h. der aktuelle Thread/Prozess sollte nicht auf das Schreiben der Logging-Informationen (zb: auf die Festplatte) warten müssen.
4. Mögliche auftretende Signale sollten abgefangen werden:
    1. SIGSEGV ... illegal memory access

   2. SIGFPE … floating point error
   3. …
5. Ein konfigurierbares Benachrichtigungssystem sollte zur Verfügung stehen.
6. …

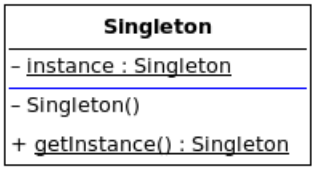
Wir wollen in der Folge uns dem Punkt 1 (NUR ein Logging-Objekt) widmen und das Singleton-Muster besprechen.

1.2.1. SINGLETON-MUSTER

http://de.wikipedia.org/wiki/Singleton_%28Entwurfsmuster%29

Das Singleton (auch Einzelstück genannt) ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster (engl. Creational Patterns).

Es verhindert, dass von einer Klasse mehr als ein Objekt erzeugt werden kann. Dieses Einzelstück ist darüber hinaus üblicherweise global verfügbar.



☑ Verwendung wenn

- ☐ nur ein Objekt zu einer Klasse existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird oder
- ☐ das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

☑ Anwendungsbeispiele sind:

- ☐ ein zentrales Protokoll-Objekt (Logging-Objekt), das Ausgaben in eine Datei schreibt.
- ☐ Druckaufträge, die zu einem Drucker gesendet werden, sollen nur in einen einzigen Puffer geschrieben werden.

☑ Vorteile: Das Muster bietet eine Verbesserung gegenüber globalen Variablen:

- ☐ Zugriffskontrolle kann realisiert werden.
- ☐ Das Einzelstück kann durch Unterklassenbildung spezialisiert werden.
- ☐ Welche Unterklasse verwendet werden soll, kann zur Laufzeit entschieden werden.
- ☐ Die Einzelinstanz muss nur erzeugt werden, wenn sie benötigt wird.
- ☐ Sollten später mehrere Objekte benötigt werden, ist eine Änderung leichter möglich als bei globalen Variablen.

1.2.2. Beispiel: muster-singleton.cpp

Bringen Sie das folgende Programm zum Laufen:

private sind:
- **Default-Konstruktor**
- **Kopierkonstruktor**
- **Zuweisungsoperator**

```
/*
muster-singleton.cpp
-------------------------------------------------------
private sind:
1. Default-Konstruktor
2. Kopierkonstruktor
3. Zuweisungsoperator
-------------------------------------------------------
*/

#include <ctime>
#include <iostream>
using namespace std;

class Singleton {
private:
     // you cannot create an object
     Singleton() {}

     // you cannot make a copy of an object
     Singleton(const Singleton&) {}

     // you cannot make a copy by assign-operator
     Singleton& operator=(const Singleton&) { return *this; }

     ~Singleton() {}

public:
     static Singleton& getInstance(){
          static Singleton instance;
          return instance;
     }

     // as an example: a Logger Object as a Singleton
     void log(int level, string msg){
          time_t second;
          struct tm *atime;
          char sTime[80];

          time(&second);
          atime= localtime(&second);
          strftime(sTime, 80, "%c", atime);

          cout << sTime << ":" <<level << ":" << msg<<endl;
     }
```

```
};

// LOGGING LEVEL
#define INFO 0
#define MESSAGE 1
#define WARNING 2
#define ERROR 3
#define FATAL 4

int main() {
     // create the one and only one singleton object.
     // its created within getInstance(), that returns
     // reference to the singleton object

     Singleton& logger= Singleton::getInstance();


     // SINGLETON:
     // Addresses are all the same, because of there is
     // only one and only one singleton object

     cout << "\ndemonstration of singleton pattern: " << endl;
     cout << "   3 addresses should have the same value:" <<endl;
     cout << "   "<< hex << & logger << endl;
     cout << "   "<< hex << & Singleton::getInstance() << endl;
     cout << "   "<< hex << & Singleton::getInstance() << endl;


     // LOGGING  and SINGLETON:
     // you can use/reference the singleton object
     Singleton::getInstance().log(INFO, "this is my first log entry");

     logger.log(MESSAGE, "here is my second log entry");


     return 0;
}
```

1.3. LOGGING: Fortgeschrittenes

Wir wollen uns im Internet nach einer besseren Lösung umsehen und finden dabei folgendes:

http://www.codeproject.com/Articles/288827/g2log-An-efficient-asynchronous-logger-using-Cplus

bzw.:
https://sites.google.com/site/kjellhedstrom2/g2log-efficient-background-io-processign-with-c11

aus: https://bitbucket.org/KjellKod/g2log/src

Hier ein kurzer Überblick zur Verwendung:

```
EXAMPLE USAGE OF G2LOG
======================
Optional to use either streaming or printf-like syntax
 LOG(INFO) << "As Easy as ABC or " << 123;
    or
 LOGF(WARNING, "Printf-style syntax is also %s", "available");


Conditional logging
 int less = 1; int more = 2
 LOG_IF(INFO,(less<more))<<"If[true],then this text will be logged";

 // or with printf-like syntax
 LOGF_IF(INFO,(less<more),"if %d<%d then this text will be logged",
less,more);


Design-by-Contract
CHECK(false) will trigger a "fatal" message.
It will be logged, and then the application will exit.

 CHECK(less != more); // not FATAL
 CHECK(less > more) << "CHECK(false) triggers a FATAL message");
```

Zusammenfassung (gekürzt): https://bitbucket.org/KjellKod/g2log/src

```
G2log is an
1. asynchronous,
2. "crash safe" logger

What this means in practice is that

1. All the slow I/O disk access is done in a background thread.

2. g2log provides flush of log to file at shutdown.

3. It is thread safe.

4. It is CRASH SAFE.
     If certain fatal signals occour (segmentation fault, SIGSEGV,...)
     it will log and save to file all previously buffered log entries
     before exiting.

5. It is cross platform. For now, tested on Windows7 and Ubuntu

7. G2log is used in commercial products as well as hobby projects since
     early 2011. The code is given for free as public domain.
     This gives the option to change, use, and do whatever with it.
```

```
8. The stable version of g2log is available at
     https://bitbucket.org/KjellKod/g2log
   The in-development g2log where new features are tried out is
     available at https://bitbucket.org/KjellKod/g2log-dev ongoing
     and released features can be seen at
     https://bitbucket.org/KjellKod/g2log-dev/wiki/Home
```

1.4. LOGGING: Übungen

Siehe 02-ueben