14. Singletons

"DCLP works only if Step1 and 2 are completed before Step3 is performed, but there is NO way to express this in C or C++ ☹"

Zwar hat der C++-Standard den Begriff *sequence point:*

> ***At certain specified points in the excution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. ... A sequence point occurs at the end of each statement.***

Dieser hilft hier aber nicht weiter, weil er keinerlei Relation zu Threads steht
Der Begriff Thread kommt im ganzen ISO/IEC FDIS 14882 (C++) überhaupt nicht vor.

Auch `volatile` hilft hier nicht weiter (s. Dr. Dobbs Artikel Part II)

Ein künftiger Standard C++0x wird sicher Aussagen über das Verhalten von C++-Programmen mit (abstrakten) Threads machen.

14. Singletons

Bis dahin bleibt nur die Empfehlung:

"In conclusion you should check your compiler documentation before implementing the DCLP. (This makes it the Triple-Checked Lock Pattern.) Usually the platform offers alternative, non-portable concurrency-solving primitives, such as memory barriers, which ensure ordered access to memory."

oder eine der Alternativen (s. Dr. Dobbs Artikel Part II):

- ***lock always* Lösung, wenn diese gar nicht kritisch ist**
- **teure *lock always* Lösung mit lokalem Cache für den Zeiger**
- **Singleton-Initialisierung in einer (*single threaded*) startup Phase**

14. Singletons

Dekomposition eines `SingletonHolder` (Singleton aus loki):
Entkopplung dreier Zuständigkeiten (*policies*) (hier nur angedeutet)

1. ***Creation*** **: steuert die Art der Erzeugung**
2. ***Lifetime*** **: diverse (LIFO, Phoenix, longevity, infinite)**
3. ***ThreadingModel*** **: single, multi, non-portable**

```
template
<
   class T,
   template <class> class CreationPolicy = CreateUsingNew,
   template <class> class LifetimePolicy = DefaultLifetime,
   template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder
{
public:
   static T& Instance();
private:
   // Hilfsfunktionen
   static void DestroySingleton();
   // Schutz
   SingletonHolder();
   ...
   // Daten
   typedef ThreadingModel<T>::VolatileType InstanceType;
   static InstanceType* pInstance_;
   static bool destroyed_;
};
```

14. Singletons

Mit C++0x sind die Probleme lösbar:

```
#include <mutex>
#include <thread>
#include <iostream>

std::mutex m; // for synch. access to n

class Singleton {
   static int n; // sample singleton data
   static std::once_flag flag;
   static Singleton* it;
public:
   static Singleton& Instance() {
    std::call_once(Singleton::flag, &Singleton::init);
       return *it;                  }
```

14. Singletons

```
// Singleton.cpp cont.
   static up(){std::lock_guard<std::mutex> lock(m); ++n;}
   static void report() { std::cout<<n<<std::endl;}
private:
   static void init() { it = new Singleton(); }
   Singleton(){std::cout<<"Singleton created\n";}
   Singleton(const Singleton&) = delete;
   Singleton& operator=(const Singleton&) = delete;
   ~Singleton() = delete;
};
Singleton* Singleton::it = 0;
std::once_flag Singleton::flag;
int Singleton::n = 0;
```

14. Singletons

```
class UseIt {
public:
   void operator()() {
       for (int i=0; i< 1000000; ++i)
             Singleton::Instance().up();
   }
};
int main() {
    std::thread t1{UseIt()};
    std::thread t2{UseIt()};
    t1.join();
    t2.join();
    Singleton::report();
}
```

```
$ g++ -v
… gcc version 4.4.4 …
$ g++ -o s –std=c++0x Singleton.cpp –pthread
$ s
Singleton created
2000000
$
```

14. Singletons

Mehr über C++0x Threads z.B. unter

The draft standard itself:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf

The nice blog posted by Anthony Williams **[just::thread]**:

http://www.justsoftwaresolutions.co.uk/threading/

15. Object Factories

Objektorientierte Programme benutzen Vererbung und virtuelle Funktionen, um Abstraktion und Modularität zu erreichen: „By postponing until runtime the decision regarding which specific function will be called, polymorphism promotes binary code reuse and extensibility.“

```
class Base { public: virtual void foo(); };
class Derived : public Base
{public: virtual void foo(); };
class AnotherDerived : public Base
{public: virtual void foo(); };

Base* pB = /* woher auch immer */;
pB->foo(); // don‘t care for the type
```

15. Object Factories

Was mit existierenden Objekten aller Art gut funktioniert, scheitert bei der Objekterzeugung:

```
Base* pB = new Derived; // hard coded and to be known at
                        // compile time: a „magic number“
```

Manchmal kann / will man den Typ aber nicht fest kodieren:

- **Das Wissen über den Typ ist lokal nicht verfügbar, es kommt von woanders her‘, z.B. aus einer separaten (virtuellen) Funktion `Create`**
- **Der Typ ist schon bekannt, aber in einer dem Programm nicht direkt zugänglichen Form, z.B. als Zeichenkette `“Derived“`**

Man braucht gewissermaßen *virtuelle Konstruktoren.*

15. Object Factories

Wir brauchen dann eine Objekt-Factory, die den Typ eines zu erzeugenden Objektes erst zur Laufzeit erfährt, weil dieser

- **erst interaktiv eingegeben/ausgewählt wird,**
- **von einer Datei gelesen wird ...**

In einer Sprache, in der Klassen auch Objekte sind könnte man schreiben:

```
// not C++:
Class Read (const char* fileName);
Document*
DocumentManager::OpenDocument(const char* fileName) {
  Class theClass = Read(fileName);
  Document* pDoc = new theClass;
}
```

15. Object Factories

In vielen C++ Büchern wird ein Szenario a la:

```
class Shape {
public:
   virtual void Draw() const = 0;
   virtual void Rotate(double angle) = 0;
   virtual void Zoom(double zoomFactor) = 0;
   ...
};
```

verwendet um die Vorzüge virtueller Methoden zu illustrieren

```
class Drawing // holds a collection of Shape-Pointers
{
public:
   void Save(std::ofstream& outFile); // easy done with virtuality ☺
   void Load(std::ifstream& inFile);  // cannot be done with virtuality ☹
   ...
};


void Drawing::Save(std::ofstream& outFile)
{
   Drawing-Header
   for (jedes Element der Zeichnung)
   {
      (aktuelles Element)->Save(outFile);
   }
}
```

15. Object Factories

Eine denkbare Lösung besteht darin, die Varianten durchzunummerieren:

```
namespace DrawingType {
   const int LINE = 1, POLYGON = 2, CIRCLE = 3
};

void Drawing::Load(std::ifstream& inFile) {
   while (inFile) {
      int drawingType;
      inFile >> drawingType;
      Shape* pCurrentObject;
      switch (drawingType) {
          using namespace DrawingType;
       case LINE:
          pCurrentObject = new Line;
          break;
       case POLYGON:
          pCurrentObject = new Polygon;
          break;
       case CIRCLE:
          pCurrentObject = new Circle;
          break;
       default:
          // Fehler behandeln - unbekannter Objekttyp
       }
       pCurrentObject->Read(inFile);
       // add the Sahpe to the collection
   }
}
```

Ist wohl eine (ganz schlechte) Objekt-Factory:

1. Nutzt **`switch`** — the OO-NoGo ☹
2. Ist ein Flaschenhals bzgl. Quelltextabhängigkeit und Wartung ☹
3. Kann nicht (vernünftig) erweitert werden ☹

15. Object Factories

Stattdessen sollte eine Zerlegung so stattfinden, dass Operationen dort angeboten werden, wo sie hin gehören (Line-Erzeugung in Line.h , ...)

Idee: Mit Zeigern entkoppeln

```
class ShapeFactory {
public:
   typedef Shape* (*CreateShapeCallback)();
private:
   typedef std::map<int, CreateShapeCallback> CallbackMap;
public:
   // Bei erfolgreicher Registrierung wird 'true' zurückgegeben.
   bool RegisterShape(int shapeId, CreateShapeCallback createFn);
   // Gibt 'true' zurück, wenn shapeId zuvor registriert wurde.
   bool UnregisterShape(int shapeId);
   Shape* CreateShape(int shapeId);
private:
   CallbackMap callbacks_;
};
```

15. Object Factories

```
// Implementierungsmodul für die Klasse Line
// Einen anonymen Namespace anlegen, damit die Funktion
// für andere Module unsichtbar ist.
namespace
{
   Shape* CreateLine()
   {
      return new Line;
   }
   // Die ID der Klasse Line
   const int LINE = 1;
   // TheShapeFactory sei eine Singleton-Fabrik
   // (siehe Abschnitt 14).
   const bool registered =
      TheShapeFactory::Instance().RegisterShape(LINE, CreateLine);
}
```

15. Object Factories

```
bool ShapeFactory::RegisterShape
                (int shapeId, CreateShapeCallback createFn) {
   return callbacks_.insert
                (CallbackMap::value_type(shapeId, createFn)).second;
}

bool ShapeFactory::UnregisterShape(int shapeId) {
   return callbacks_.erase(shapeId) == 1;
}


Shape* ShapeFactory::CreateShape(int shapeId) {
   CallbackMap::const_iterator i = callbacks_.find(shapeId);
   if (i == callbacks_.end())
   {
      // not found
      throw std:: runtime_error("Unbekannte Shape-ID");
   }
   // Aufruf der Erzeugungsfunktion
   return (i->second)();
}
```

15. Object Factories

Problem bleibt:
konsistente & flexible Vergabe der ID‘s, z.B. Strings statt Zahlen (leider eignet sich `std::type_info::name` nicht ☹), UUIDs, ...

Lösung: Entkopplung von der Factory durch Generalisierung:

Was sind die Zutaten zu einer Objekt-Factory:

- ***ConcreteProduct*: das, was die Fabrik ausliefern kann**
- ***AbstractProduct*: die abstrakte Basis aller Produkte**
- ***ProductTypeIdentifier*: das Identifikationsschema**
- ***ProductCreator*: Abstraktion von der Creater-Funktion**

Bis auf die konkreten Produkte (die die Fabrik ja gar nicht kennen soll), gehen alle Zutaten als Parameter in die generische Fabrik ein.

15. Object Factories

```
template <
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator  >
class Factory {
public:
   bool Register(const IdentifierType& id, ProductCreator creator)
   {
      return associations_.insert(AssocMap::value_type(id, creator)).second;
   }
   bool Unregister(const IdentifierType& id){
      return associations_.erase(id) == 1;
   }
   AbstractProduct* CreateObject(const IdentifierType& id)
   {
      typename AssocMap::const_iterator i = associations_.find(id);
      if (i != associations_.end())
         return (i->second)(); // Creator as callable entity
      // Fehlerbehandlung
   }
private:
   typedef std::map<IdentifierType, ProductCreator> AssocMap;
   AssocMap associations_;
};
```

15. Object Factories

Wie sollte man Fehler behandeln?

- **return 0?**
- **Programmabbruch?**
- **Programmausnahme?**
- **Eine dynamische Bibliothek nachladen, einen passenden Creator registrieren und es nochmal versuchen?**

Am besten (wie immer): nicht festlegen, sondern generisch halten:

```
template <
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator,
   template<typename, class>
      class FactoryErrorPolicy >
class Factory : public FactoryErrorPolicy<IdentifierType, AbstractProduct> {
public:
   AbstractProduct* CreateObject(const IdentifierType& id) {
      typename AssocMap::const_iterator i = associations_.find(id);
      if (i != associations_.end()) return (i->second)();
      return OnUnknownType(id);
   }
private: // ... as above
};
```

15. Object Factories

```
// eine Variante:
template <class IdentifierType, class ProductType>
class DefaultFactoryError {
public:
   class Exception : public std::exception {
      public:
         Exception(const IdentifierType& unknownId): unknownId_(unknownId){}
         virtual const char* what() {
            return "Unbekanntes Objekt an Factory übergeben.";
         }
         const IdentifierType GetId(){
            return unknownId_;
         };
      private:
         IdentifierType unknownId_;
      };
protected:
   static ProductType* OnUnknownType(const IdentifierType& id){
      throw Exception(id);
   }
};
```

15. Object Factories

```
// Factory prototype with defaults:

template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;

// ProductCreator = AbstractProduct* (*)() is the simplest
// could be even Functor<AbstractProduct*> (Abschnitt 13):
//    any callable thing returning a AbstractProduct*

// loki uses instead of std::map a drop-in replacement AssocVector
// making the container another template template parameter
// is (per standard) not allowed WHY?
```

15. Object Factories

Eine Variante: Cloning-Factories – statt einer ID gibt es jeweils ein Prototyp-Objekt, alle konkreten Produkte müssen `Clone` anbieten (also im Quelltext verfügbar sein ☹)

```
class Shape {
public:
   virtual Shape* Clone() const = 0;
   ...
};
class Line : public Shape {
public:
   virtual Line* Clone() const { // covariant return types
      return new Line(*this);
   }
   ...
};
```

`Clone` darf dann aber auch in keinem der Produkte vergessen werden!

```
class dottedLine: public Line { /*without Clone */ } dotdot;
Shape* pShape = &dotdot;
Shape* pDuplicate = pShape->Clone(); // trouble ahead
```

15. Object Factories

Cloning-Factories

Die Prototypen selbst (bzw. ihre std::type_info, in loki ihre TypeInfo) kann als ID benutzt werden, der ProductCreator erhält den Prototyp als Parameter

```
template <
   class AbstractProduct,
   class ProductCreator = AbstractProduct* (*)(AbstractProduct*),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory {
public:
   AbstractProduct* CreateObject(const AbstractProduct* model);
   bool Register(const TypeInfo&, ProductCreator creator);
   bool Unregister(const TypeInfo&);
private:
   typedef AssocVector<TypeInfo, ProductCreator> IdToProductMap;
   IdToProductMap associations_;
};
```