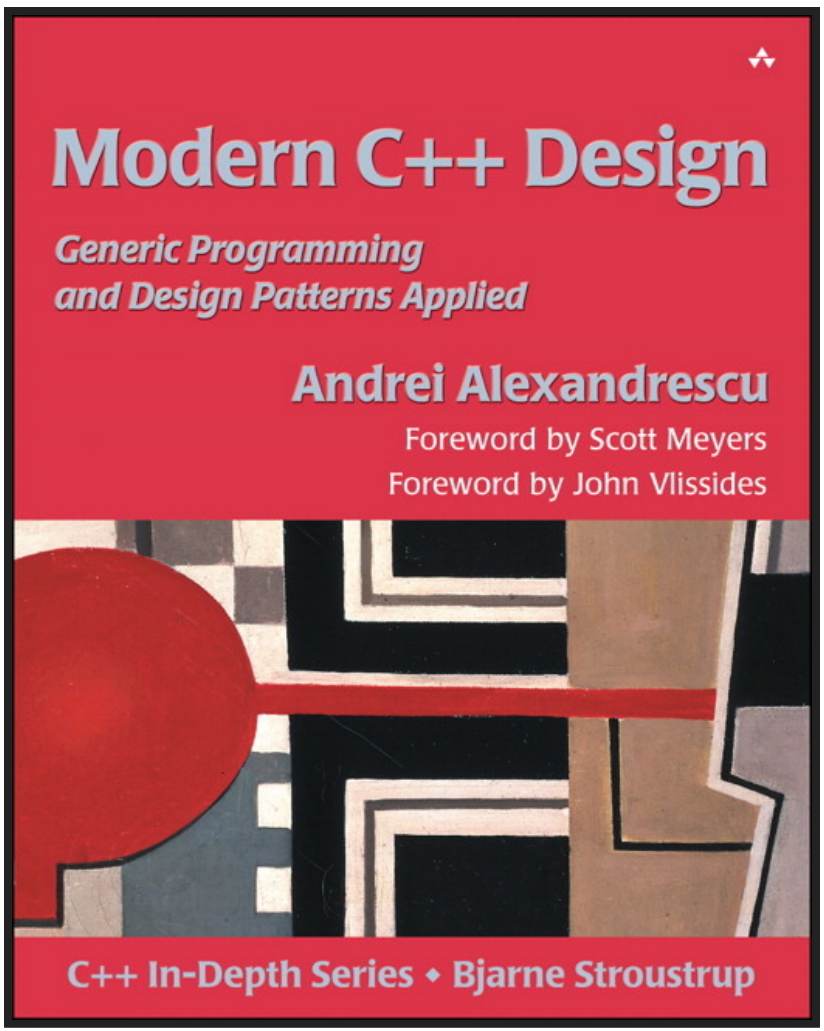
DYNAMIC FACTORY PITFALLS

WHAT THEY ARE AND HOW TO AVOID THEM

Author Owen Arnold

WHAT IS THE DYNAMICFACTORY?

A “Factory” type encapuslating a map where values are of type *AbstractFactory* and keys are of type *std::string*



WHAT IS THE DYNAMICFACTORY?

This is what the map looks like

```
std::map<std::string, AbstractFactory*, Comparator>;
```

The AbstractFactory is declared like this

```
typedef AbstractInstantiator<Base> AbstractFactory;
```

WHAT IS THE DYNAMICFACTORY?

This is how new types are registerd

```
template <class C>
void subscribe(const std::string &className) {
    subscribe(className, new Instantiator<C, Base>);
}
```

WHAT IS THE DYNAMICFACTORY?

`AbstractInstantiator` provides a factory method

```
template <typename Base>
class  AbstractInstantiator  {
  public:
  virtual boost::shared_ptr<Base> createInstance() const = 0;
};
```

A concrete `Instantiator` is used everywhere. Providing `C` "is a" `Base`

```
template <typename C, typename Base>
class Instantiator : public AbstractInstantiator<Base> {
  public:
  boost::shared_ptr<Base> createInstance() const {
    boost::shared_ptr<Base> ptr(new C);
    return ptr;
  }
};
```

WHY ARE THEY SO USEFUL?

- Easily accessible global map
- Stores light weight proxies to our Product type
- Very easy to register new Products
- Gives the illusion of compile time subscription
- Products from User Dynamically Loaded Libraries can be registered into the DynamicFactory via global map.

PROBLEM #1

DESIGNING AROUND DYNAMICFACTORIES REQUIRES SIDE-EFFECTS

Direct access to singletons prevents IOC.

PROBLEM #2

DYNAMICFACTORIES HAVE THEIR OWN LIFETIME

Most implementations such as *AlgorithmFactory* encloses the *DynamicFactory* in a *SingletonHoder*.

- Violates the “Single Responsibility” principle
- Ties lifetime of DynamicFactory to the lifetime of the application
- Doesn't seem necessary to have one instance in lots of cases

PROBLEM #3

ALL FACTORY PRODUCTS ARE CORRUPTABLE (PART 1)

All factory products created via the Instatiators are corruptable by design.

```
virtual Base *createUnwrappedInstance() const {
  return static_cast<base *="">(new C);
}
```

- Client has to know about “Temporal Coupling”
- DynamicFactory products are corrupt until you fix them up
- Weakens enapsulation
- Ties API of “Things we want to make” to mechanism for creating them

  Not strictly true that Products must have a default constructor, but Never done otherwise in the codebase.

PROBLEM #3

ALL FACTORY PRODUCTS ARE CORRUPTABLE (PART 2)

Actual code from a `Product` of a DynamicFactory in master mantid code base

```
// WARNING!!!! THESE METHODS ARE USED BEFORE INITIALIZE IS EXECUTED
// CAN NOT RELY ON THE CONTENTS OF THE CLASS (THEY ARE VIRTUAL STAT
/** return the number of dimensions, calculated by the transformati
   workspace.
   Depending on EMode, this numebr here is either 3 or 4 and do not
   input workspace*/
unsigned int
getNMatrixDimensions(Kernel::DeltaEMode::Type mode,
                     API::MatrixWorkspace_const_sptr Sptr =
                         API::MatrixWorkspace_const_sptr()) const;
```

PROBLEM #4

ALL FACTORY COMPARISONS MUST BE STRING BASED

No support for complex comparison. It's just a map.

ALTERNATIVE #1

VOTE BASED CONSTRUCTION

Factories and Products are fully separated concepts. Allows discovery of best factory match. Does not involve Temporal Coupling. Complex matching is possible considering all possible alternatives.

IFileLoader mechanism loosely based upon this.

ALTERNATIVE #2

CHAIN OF RESPONSIBILITY

ChainableFactory uses the GOF “Chain of Repsonsibility” to daisy-chain successive possible Factories. Factories are fully separated from Products. Complex matching is possible. It does not introduce Temporal Coupling, and does not involve a Singleton.

ALTERNATIVE #3

GOF CREATIONAL PATTERNS

No template or macro magic. Standard OO patterns that have stood the test of time. Are you familar with them?

- Factory Method
- Abstract Factory
- Builder
- Virtual Constructor

https://en.wikipedia.org/wiki/Design_Patterns#Creational