# Design Patterns in Chroma

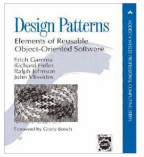
Bálint Joó (bjoo@jlab.org)
Jefferson Lab, Newport News, VA

given at

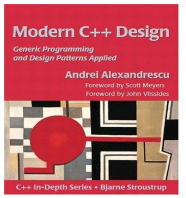
HackLatt'06
NeSC, Edinburgh
March 29, 2006

# Design Patterns

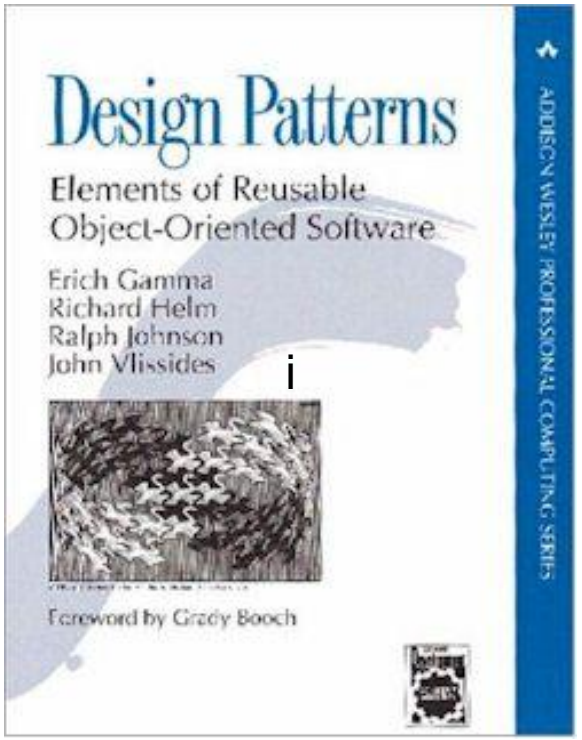
- Tried and tested object oriented techniques to solve commonly occurring problems
- Classic Software Design Book: "Design Patterns: Elements of Reusable Object Oriented Software", E. Gamma, R. Heim, R. Johnson & J. Vlissides (aka The Gang Of Four)

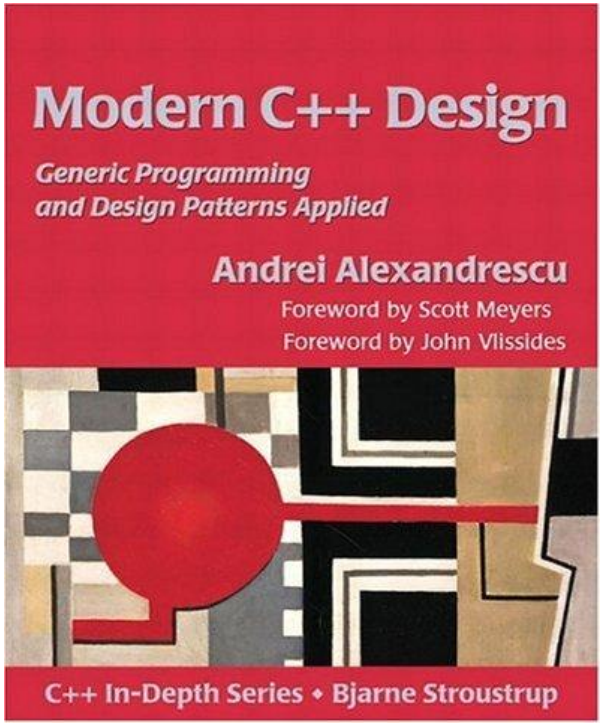



- Our implementations of design patterns come from the LOKI library described in "Modern C++ Design, Generic Programming and Design Patterns Applied", by Andrei Alexandrescu



# Read (at least bits of) these books!!!!!!




You can find them in your local library!
(gratuitous plug for librarians everywhere)

# Design Patterns I: Smart Pointer (Handle)

- Reference counting "smart pointer"
- Assignment / copy of handle increases ref. count
- Destruction of handle reduces reference count
- When ref. count reaches zero destructor is called.

Construct with newly allocated pointer. Reference count is set to 1

```
#include <handle.h>
{
  Handle<Foo> f( new Foo() );
  Foo& f_ref = (*f);
  f_ref.method();
  f->method();
}
```

Dereference like normal pointer

Handle goes out of scope, reference count is decreased, reaches 0, so delete is called and memory is freed

# Design Patterns II: Singleton

- An entity of which there is only one within a program
- Kind of a "virtuous global object"
- Static class + static methods != singleton
- Destruction/Life-time/Co-dependency issues
- Used for eg:
  - Factories (see later)
  - Shared XML Log file
  - QDP++ Memory Allocator
  - Staggered Fermion Phases

# Design Patterns II: Singletons

- Define as (eg: in my_singleton.h)

LOKI Singleton implementation template

Policy Templates (eg: staticity, lifetime)

```
typedef SingletonHolder< MyClass,... > TheMySingleton;
```

Class of which there will be only one instance

(Type)Name to refer to singleton. Our convention: singleton names start with "The" or "the"

- Use as

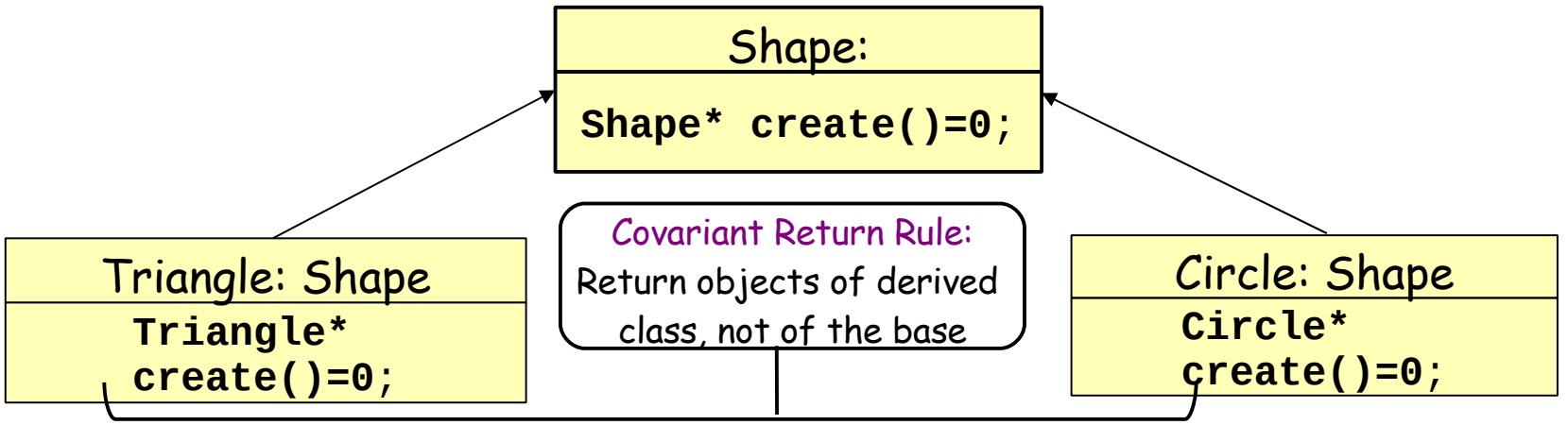
```
#include "my_singleton.h"

TheMySingleton::Instance().memberFunction();
```

Member function of instance object

Returns Reference to singleton Instance

# Design Patterns III: Factory Function

- A function to create objects of a given kind.
- Abstracts away details involved in creation
- Can create Derived Classes of a given Base Class
  - ie: allows selection of particular implementation for an abstract interface
- Useful as a Virtual Function in a class



# Design Patterns III: Factory Function

- A new instance of an object is created
  - Memory is allocated
  - Drop result into a Handle

```
Handle< Shape > my_shape( Circle::create() );
```

- Sometimes a concept needs several objects
  - Fermions: link state with BCs, Fermion Matrix, a propagator solver for the kind of fermion.
- Group together (virtual) factory functions in a (base) class => Factory Class

(Warning: Not every virtual func. is a factory func.)

# Design Patterns IV: Factory

- Suppose you want a choice of creating shapes at run time
- What is the best pattern?
- Naively:

```
int t; read(xml, "/Shape/Type", t);
Shape *my_shape;
switch(t) {
 case CIRCLE:
      my_shape = Circle::create();
      break;
 case TRIANGLE:
      my_shape = Triangle::create();
    break;
 default:
      QDPIO::cerr << "Unknown shape" <<
endl;
    QDP_abort(1);
};
Handle<Shape> shape_handle(my_shape);
```

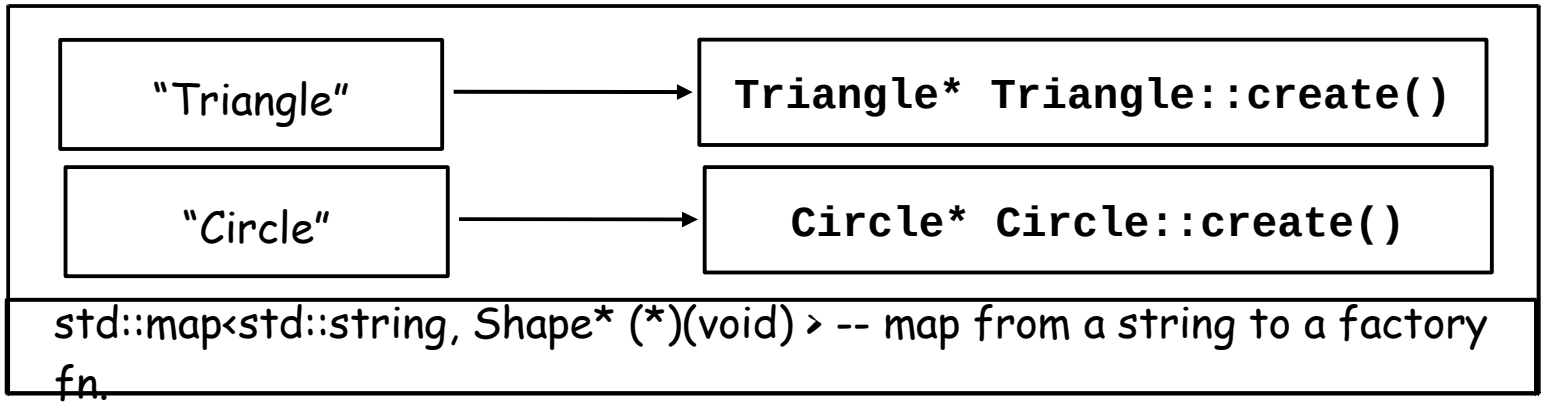
# Design Patterns IV: Factory

- Criticism
  - For every new shape I create I need to edit
    - the source files for the shape
    - The switch statement in

      EVERY SINGLE PLACE WHERE I CREATE A SHAPE
  - Having to edit seemingly unrelated files gets error prone
  - As I have more shapes, my switch statement becomes unmanageably long
- Is there a better way?
  - Yes! Use a map!

# Design Patterns IV: Factory

- A Map is an associative array (indices don't have to be numbers)




- Can now create shapes by querying the map

Insert factory function and name pairs

```
std::map<std::string, Shape* (*)(void)> shape_factory_map;
shape_factory_map.insert( make_pair("Triangle", Triangle::create() ) );
shape_factory_map.insert( make_pair("Circle", Circle::create() ) );

std::string shape_name;
read(xml, "/Shape/Name", shape_name);

Handle<Shape> my_shape(  (shape_factory_map[ shape_name ])() );
```

Look up name in map, invoke returned function

# Design Pattern IV: Factory

- Details of creation localized in the map.
- Individual creations simplified.
- BUT Name,Function pairs need to be added to map
  - If there was a global map, each Shape could call the insert function in own source file
  - Implement map as a Singleton

triangle.cc:

```
class Triangle : public Shape {
public:
  Triangle* create() { ... };
};
static bool registered =
              theShapeMap::Instance().insert(make_pair("Triangle",
                                                       &(Triangle::create())));
```

Singleton access

# Design Patterns IV: Factory

- This pattern is the Factory pattern
- The essence is a map from ProductID to Product Creation Function
- We use the LOKI implementation from Alexandrescu's book (ObjectFactory<> template)
  - Provides registerObject function for map insertion.
  - Provides createObject function for map look-up
  - Allows control of parameters to createObject
  - Allows us to customize policies (eg create using new, create using malloc, etc etc)

# Our Typical Scenario in Chroma

Define Factory in xxx_factory.h – specialise SingletonHolder and Object Factory templates
(eg: chroma/lib/update/molecdyn/monomial/monomial_factory.h)

```
typedef SingletonHolder<
 ObjectFactory<
   Monomial< multi1d<LatticeColorMatrix>,
             multi1d<LatticeColorMatrix> >,
   std::string,
   TYPELIST_2(XMLReader&, const std::string&),
   Monomial< multi1d<LatticeColorMatrix>,
             multi1d<LatticeColorMatrix> >* (*)(XMLReader&,
                                                const std::string&),
   StringFactoryError> > TheMonomialFactory;
```

Singleton Style

Factory Template

Product Type

Product ID (key) type

Params of Creation Func

Creation Function Type

Map Lookup ErrorType

# Our Typical Scenario in Chroma

In xxx_product.h - define the product and a product specific namespace (eg: chroma/lib/update/molecdyn/monomial/unprec_two_flavor_monomial_w.h)

```
namespace UnprecTwoFlavorWilsonTypeFermMonomialEnv
{
  extern const std::string name;
  extern const bool registered;
};

class UnprecTwoFlavorWilsonTypeFermMonomial :
  public  TwoFlavorExactUnprecWilsonTypeFermMonomial<
  multi1d<LatticeColorMatrix>,
  multi1d<LatticeColorMatrix>,
  LatticeFermion>
  {
    ...
};
```

Namespace for product so we can reuse the "name" and "registered" elsewhere

key in map (defined in .cc)

is product registered/linkage

The actual class declaration

# Our Typical Scenario in Chroma

In xxx_product.cc – almost everything else:

(eg: chroma/lib/update/molecdyn/monomial/unprec_two_flavor_monomial_w.cc)

```
namespace UnprecTwoFlavorWilsonTypeFermMonomialEnv
{
   Monomial< multi1d<LatticeColorMatrix>, multi1d<LatticeColorMatrix> >*
   createMonomial(XMLReader& xml, const string& path)
   {
     return new UnprecTwoFlavorWilsonTypeFermMonomial(
       TwoFlavorWilsonTypeFermMonomialParams(xml, path));
   }

   const std::string name("TWO_FLAVOR_UNPREC_FERM_MONOMIAL");

   bool registerAll()
   {
     bool foo = true;
     foo &= WilsonTypeFermActs4DEnv::registered;
     foo &= TheMonomialFectory::Instance().registerObject(name,
                                                          createMonomial);

   }

     const bool registered = registerAll();
}
```

Code for creation fn

The name, declared as extern in .h

Ensure dependency is registered (see later)

Call to Registration

called at start up

# Fly in Ointment - Linkage

- If the registered symbol is not referenced in our program then the compiler may not link xxx_product.o. No linkage means:
  - registerAll() is not called at startup
  - our Monomial does not get registered
  - our temple collapses around our heads
- A solution (aka hack) to this program is to make sure we reference the symbol.
  - linkageHack() function in chroma.cc and hmc.cc

# linkageHack and Aggregation

- in linkageHack() we explicitly reference every registered product we need.
- too many products – we want to aggregate
  - xxx_aggregate.h and xxx_aggregate.cc files

```
namespace WilsonTypeFermMonomialAggregrateEnv
{
 bool registerAll()
 {
   bool success = true;
   success &= UnprecTwoFlavorWilsonTypeFermMonomialEnv::registered;
   success &= EvenOddPrecConstDetTwoFlavorWilsonTypeFermMonomialEnv::registered;
   success &= EvenOddPrecLogDetTwoFlavorWilsonTypeFermMonomialEnv::registered;

   // and more ...

   return success;
 }
 const bool registered = registerAll();
}
```

Namespace for Aggregate

Reference individual **registered**-s

Referencing this will pull in ***all*** the individual ones

(chroma/lib/update/molecdyn/monomial/monomial_aggregate_w.cc)

# Comments on Linkage Hack and Aggregation

- Using the aggregation our linkageHack function is simplified

```
bool linkageHack(void)
{
  bool foo = true;
  foo &= GaugeMonomialEnv::registered;
  foo &= WilsonTypeFermMonomialAggregrateEnv::registered;
  foo &= LCMMDIntegratorAggregateEnv::registered;
  foo &= ChronoPredictorAggregrateEnv::registered;
  foo &= InlineAggregateEnv::registered;
  return foo;
}
```

- Still not ideal solution – since now we lose fine control
  - eg: on QCDOC want to omit some individual unused products or we run out of space (.text segment)
- In principle annoyance: Aggregates and Linkage Hack
  - equivalents of big switch statement we didn't want

# Summary

- In Chroma, we make use of several design patterns
  - Smart Pointer, Factory Function, Singleton, Factory
  - We use these patterns EVERYWHERE
- We make great use of the LOKI library
- I have shown how these patterns 'look' in the code
- Using patterns allowed us great flexibility and solved many problems
  - eg: using many kinds of fermion without recompilation
- BUT: We are still annoyed by the linkage issue and are looking for a portable solution