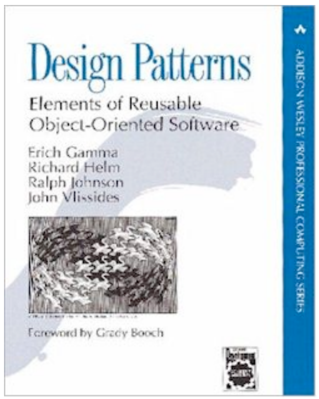
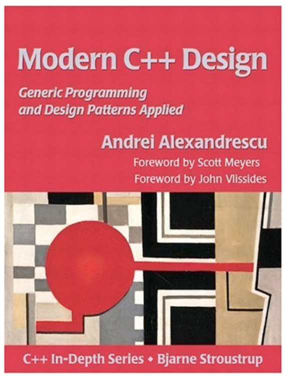
Software Design Tidbits

Bálint Joó,
Scientific Computing Group
Jefferson Lab

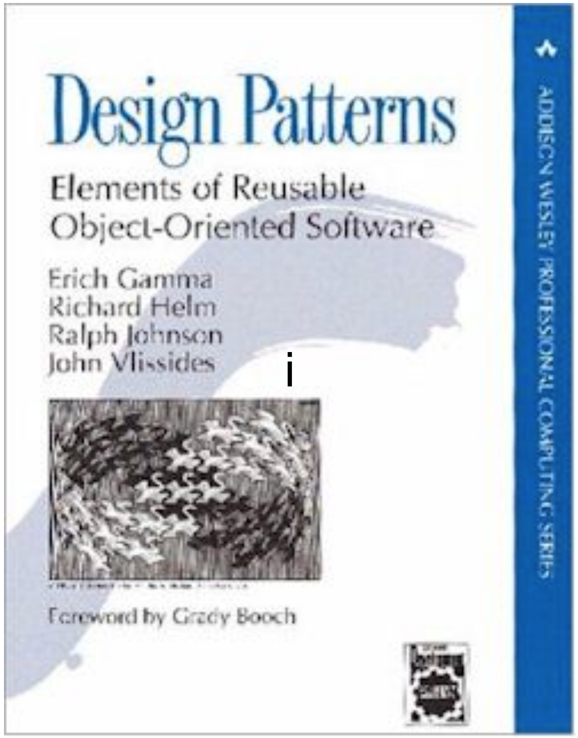
Design Patterns

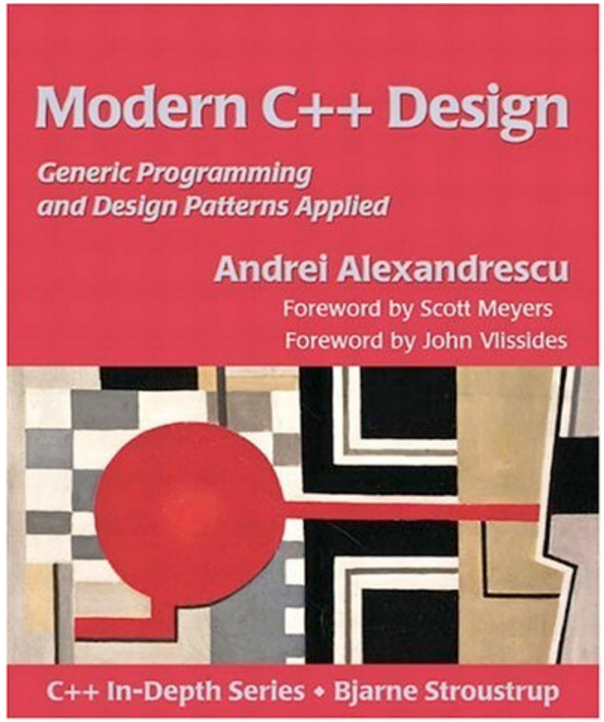
- Tried and tested object oriented techniques to solve commonly occurring problems
- Classic Software Design Book: “Design Patterns: Elements of Reusable Object Oriented Software”, E. Gamma, R. Heim, R. Johnson & J. Vlissides (aka The Gang Of Four)
- Our implementations of design patterns come from the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

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

}
```

Initialize with a raw pointer

Dereference like a normal pointer

f goes out of scope here.
Since there is only 1 reference to the pointer in it (from f itself) it is decreased, and f will be freed

Design Patterns II: Singleton

- An entity of which there is only one in 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)

Policy templates: how to create, lifetime, static or not

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

Library implementation of Singleton (LOKI)

Name to refer to the Singleton

- Use as

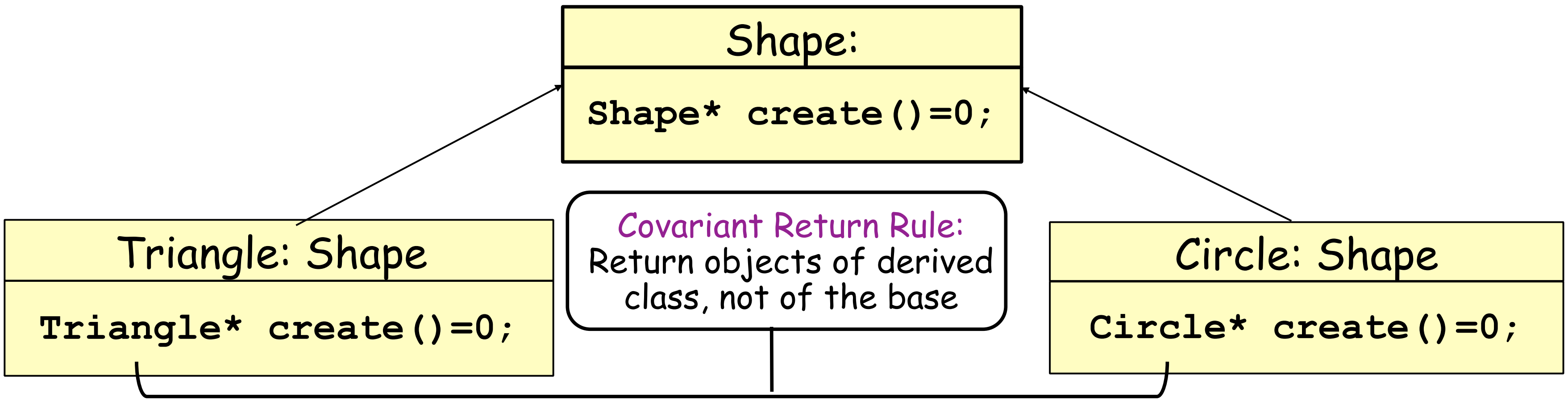
```
#include "my_singleton.h"

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

Returns Reference to the MyClass within the singleton

Member of MyClass

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

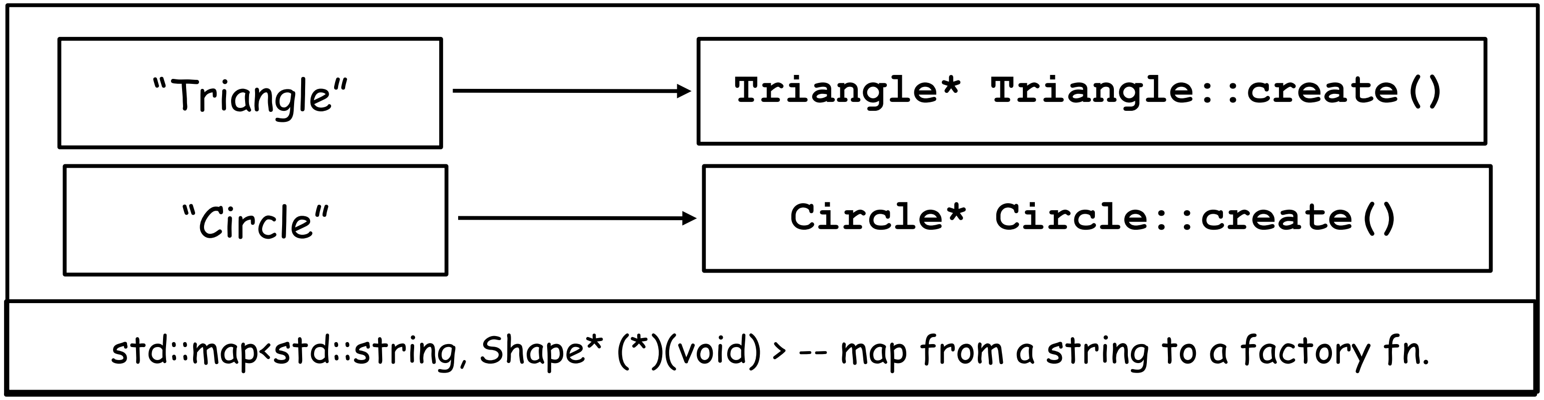
- A new instance of an object is created
  - Memory is allocated
  - Drop result into a Handle

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

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

```
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 ])() );
```

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()));
```

Typical Chroma Scenario

```
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::registerAll();
     foo &= TheMonomialFactory::Instance().registerObject(name,
                                                            createMonomial);

   }

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

Creation Function

Name

Register self and dependencies

call registerAll();

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::registerAll();
   success &= EvenOddPrecConstDetTwoFlavorWilsonTypeFermMonomialEnv::registerAll();
   success &= EvenOddPrecLogDetTwoFlavorWilsonTypeFermMonomialEnv::registeAll();

   // and more ...

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

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

Aggregate various Wilson like monomials

Linkage Hack and Aggregation

- Using the aggregation our linkageHack function is simplified

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

- Not ideal, since we loose some fine grained control
- Its an ‘in principle’ annoyance
  – kind of like a big switch statement that we were trying to avoid

Design Pattern 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

C++ Templates in General

- Templates allow you to perform 'substitutions' in code at ***compile time***
- Typical Use: Containers for several types

```
template< typename T>
class Bag {
public:
   insert( const T& item ) { // Code goes here … }
};

int main(int argc, char *argv[])
{
  Bag<float> BagOfFloats;
  Bag<int>   BagOfInts;

  int i=5; float f=6.0;
  BagOfFloats.insert( f );
  BagOfInts.insert( i );
  BagOfInts.insert( f );    // NONONO! Unless automatic conversion
}
```

'Value' Templates

- Bag<T> was templated only on types. Can also template on values
- This is useful e.g. if I want to pick a container size at compile time, but it will stay fixed after that

```
template< typename T, int N>
class FixedSizedBag {
public:
   insert( const T& item, int position ) { // Code goes here … }
};

int main(int argc, char *argv[])
{
  FixedSizedBag<float,2> BagOfTwoFloats;

  float f1=6.0;
  float f2=3.0;
  BagOfTwoFloats.insert(f1, 0); // Insert at position 0
  BagOfTwoFloats.insert(f2, 1); // Insert at position 1
}
```

Templating Classes, Functions

- We can template functions as well as classes

```
#include<iostream>

template< typename T >
void doSomethingWithT( const T& input )
{
   std::cout << “T is “ << input << endl;
}
```

- NB: a definition of **operator<<** must be available that can print a type T – so called 'duck typing'

```
class MyFunnyType {} ; // Empty class
int main(int argc, char *argv[])
{
   int i=6 ; doSomethingWithT(i);     // OK: << handles int
   float f=5.0 ; doSomethingWithT(f); // OK: << handles float
   MyFunnyType mf; doSomethingWithT(mf); // BARF!!!!!!
}
```

Class Specialization

- One can 'customize' the code of a template for specific template values – this is called specialization

```
template<class T>
class FancyPrinter {
public:
  FancyPrinter(const T& input) {
    std::cout << "Fancy Printer: " << input << endl;
  }
};

// SPECIALIZE The Entire Class
template<>
class FancyPrinter< MyFunnyType > {
   FancyPrinter(const MyFunnyType& input) {
     std::cout << "Fancy Printer: trying to print FunnyType"
              << endl;
   }
};
```

Class Member Specialization

- I can specialize individual member functions of a class template e.g.:

```
template<class T>
class FancyPrinter {
public:
  FancyPrinter(const T& input) {
    std::cout << "Fancy Printer: " << input << endl;
  }
};

// SPECIALIZE The constructor
template<>
FancyPrinter< MyFunnyType >::FancyPrinter(const MyFunnyType& input)
{
  std::cout << "Fancy Printer: trying to print FunnyType"
            << endl;
};
```

Traits...

- Classes can 'export' internal types:

```
class SomeClass {
public:
   typedef float  MyFloatType;
};

// This declares a float really
SomeClass::MyFloatType  t=5.6;
```

- MyFloatType is a 'type trait' of SomeClass
- A Traits Class can hold several traits:

```
class SomeClass {
public:
   typedef float  MyFloatType;
   static const int FloatSize = sizeof(MyFloatSize);
};

std::cout << “The Size of SomeClass::MyFloatType is :”
       << SomeClass::FloatSize << endl;
```

More useful traits

- Traits become powerful when combined with templating:

```
template <typename T>
class MyTraits {
public:
   typedef T  MyType;
   static const int MyTypeSize = sizeof(T);
};
```

- Can now write generic code in terms of 'MyType' and control say memory copies using 'MyTypeSize'
- Traits are heavily used in the STL and standard libraries.

Type Computing Traits

- One can construct a set of templates to 'compute' about types:

```
template< typename T>          // Catchall case... Empty...
struct DoublePrecType {        // Will cause compiler error if One tries
};                             // to access its nonexistent members

template<>
struct DoublePrecType<float> { // Specialization for floats
  typedef double Type_t;       // this is the trait...
};

template<>
struct DoublePrecType<double> { // Specialization for doubles
  typedef double Type_t;        // This is the trait
};

std::cout << “Sizeof DP(float)”
          << sizeof( DoublePrecType<float>::Type_t ) << endl;

std::cout << “But not for ints” << sizeof( DoublePrecType<int>::Type_t )
       << endl;  // This'll barf, matches struct with no Type_t
```

Recursive Traits

- Type computations can be recursive:

```
#include <vector>
using namespace std;

template< typename T>          // Catchall case. Hopefully never
struct DoublePrecType {        // Instantiate this
};

template<>                     // Base case for floats
struct DoublePrecType<float> {
  typedef double Type_t;
};

template<typename T>           // Recursive case for vectors
struct DoublePrecType< vector<T> > {
  typedef vector< typename DoublePrecType< T >::Type_t > Type_t;
};

DoublePrecType< vector< float > >::Type_t  doublevec(5);
cout << "doublevec[0] has size=" << sizeof( doublevec[0] ) << endl;
```

Some QDP++ Traits

- Basic QDP++ Traits:
  - **WordType<T>::Type_t** - the innermost word (int, float)
  - **SinglePrecType<T>::Type_t** single precision version of T
  - **DoublePrecType<T>::Type_t** double precision versions of T
  - **QIOStringTraits<T>** - holds strings needed by QIO functions
    - **QIOStringTraits<T>::tname** = “Lattice” or “Scalar”
    - **QIOStringTraits<T>::tprec** = “U” “I” “F” etc...
- More advanced traits that arise in Expression Templates
  - **UnaryReturn< T, Op>::Type_t** - the return type produced by a Unary function Op acting on T
  - **BinaryReturn<T1, T2, Op>::Type_t** - the return type produced by a Binary Operator Op acting on inputs with type T1 and T2 respectively
- In all the cases, if we try and use a trait, for which there is no specialization case, or a 'catchall' case. We'll get horrible compiler errors...