Abstract Constructor[1]

Author: Manfred Lange
Hewlett-Packard GmbH
European Support Technology Lab (ESTL)
Herrenberger Str. 103
71034 Boeblingen
e-mail: Manfred_Lange@bbn.hp.com

Problem Statement

Sometimes you wish to create an object from a description of some kind, but you don't know which constructor to call.
In addition you need a registration mechanism for adding further classes without the need of changing existing code.

Problem Details

The 'real' problem is that in certain languages, e.g. C++, it is possible to declare member functions (synonyms: service, method) as virtual, which means, that late binding is used. In other words: it is during runtime (not compile time), when it is determined, which implementation of the virtual function is actually used.
On the other hand it is not possible to declare a constructor as virtual. There are situations where this might be appropriate, e.g.

1. an object within one hierarchy tree is given and an appropriate object in a different hierarchy tree should be created, e.g. attaching user interface elements to objects of the problem domain
2. a stream with serialized objects is given, and they need to be deserialized
3. Instead of a stream, any kind of information or description of 'something' is given, and an object of the most appropriate class should be created, e.g. a decimal number is given in string format and several classes with different precision for representing decimal numbers are available

In all these cases a mechanism is needed, represented as a function call, which takes the information or description as a parameter and creates the best-fitting class. This mechanism needs to 'know' all classes, that might be able to 'convert' the information or description into a valid object.
<u>Motivating Sample:</u>
One might have several classes representing integer numbers with different precision, e.g. 64-bit integer, 128-bit integer etc. Reading from a stream containing a lot of different integer numbers, you want to create the appropriate object for each number in that stream.
In addition you don't know whether one day the requirements for your code will change and you might be forced to support 256-bit integers, or you need to support floating point numbers or strings as well.
Also you don't want to change existing code when adding further classes.

Forces

1. We don't want base classes to change when we add derived classes.
2. We don't want clients of the base class to be affected.
3. Our language does not necessarily support RTTI.
4. We don't want the classes to know about their descriptions.

Solution

The „abstract constructor" pattern - described in this paper - is a solution of how to make a constructor virtual. Similar solutions can be found in [Coplien92] and in [Berczuk95]. The „abstract constructor" pattern extends Coplien's and Berczuk's solutions by adding a registration mechanism, which provides a way of adding further classes to the mechanism without the need of changing existing classes, which also might be tested already.

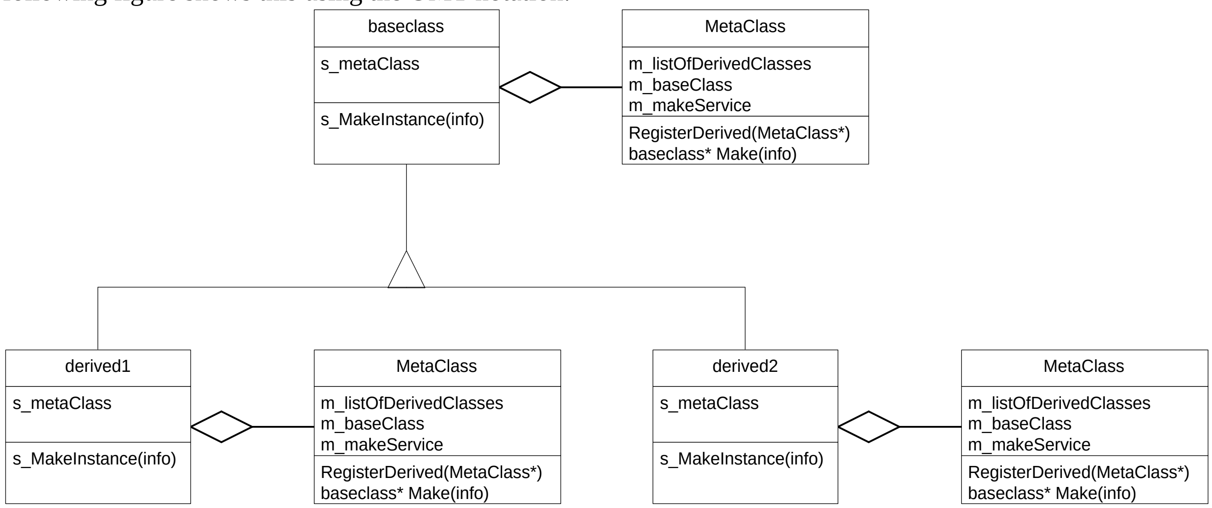
---

[1] The pattern is based upon ideas, that Desmond D'Souza presented on the OOP '94 in Munich.

In general the solution can be divided into two steps:

1. Register all classes that are part of the mechanism at a well known place.
2. Provide a uniform, abstract interface to the client object.

Step 1 can be done with a static member, which is added to each class that takes part of the mechanism. The following figure shows this using the OMT notation:



The classes that take part in the mechanism are 'baseclass', 'derived1' and 'derived2'. Each of them has a static member of type MetaClass. Here this member is named s_metaClass. It contains information about the baseclass, a list of derived classes and a pointer 'm_makeService', which points to a static function s_MakeInstance() of the associated class. s_MakeInstance() plays the role of a constructor.

The structure that represents the class hierarchy is built during startup of the application. Static members are initialized before the applications main() function is called. This means that the static members of type MetaClass are initialized as well.

Actually MetaClass is a class itself, so the static members of type MetaClass are initialized using the constructor of class MetaClass. Within this constructor, each of the members of type MetaClass registers its associated class as a derived class of the baseclass. This is done using the RegisterDerived() service. Actually RegisterDerived() takes a pointer to a MetaClass object. m_listOfDerivedClasses therefore only contains pointers to MetaClass objects, which are associated with derived classes.

Step 2: During runtime a client wants to use the abstract constructor by calling baseclass::s_metaClass::Make(info) and expects, that the return value is a pointer to a valid object of class baseclass or of one of the derived classes, e.g. of class derived1 or derived2.

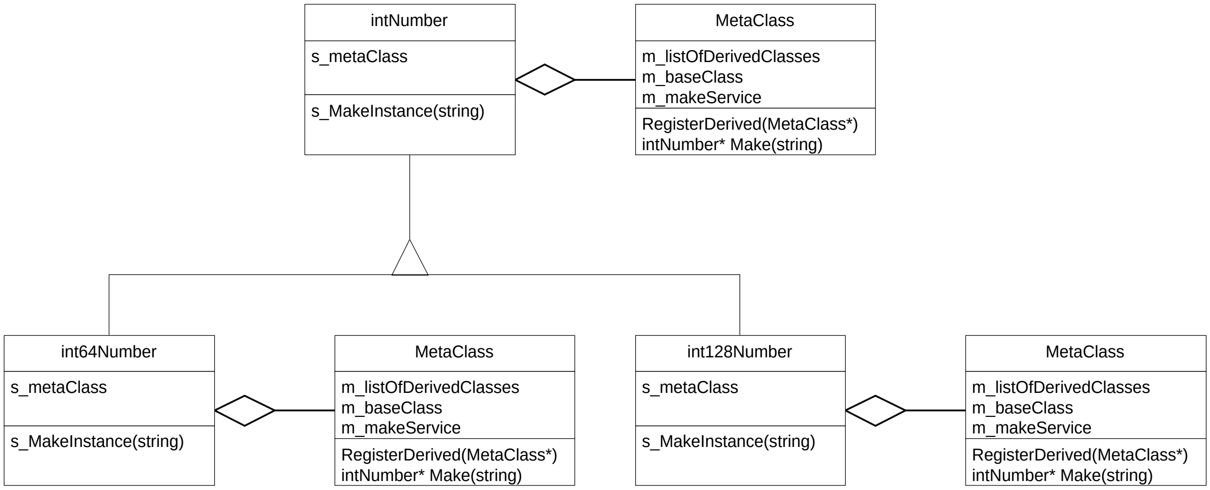
MetaClass::Make(info) is called recursively. Each MetaClass object first iterates through the list and calls Make() on each of the MetaClass objects in that list.

The iteration terminates when one of the derived classes is able to use 'info' for creating an object out of it. This is determined calling s_MakeInstance(). s_MakeInstance() checks what is given in 'info' and returns NULL, if it can't handle it.

If the list is empty or if none of the derived classes can create an object using 'info' then the MetaClass object checks, whether it has a pointer to a s_MakeInstance() function. If yes, this will be called in a last try. If this fails or if m_makeService is NULL, Make() returns NULL, indicating the creation has been failed.

Example 1 (integer numbers with different precision)

The following class hierarchy is given:



During startup the following lines of code are executed before main() is called:

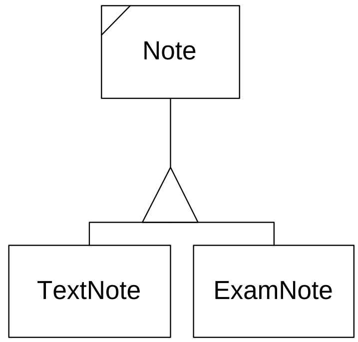
```
MetaClass intNumber::s_metaClass(     NULL,          // No baseclass.
                                        intNumber::s_MakeInstance );
MetaClass int64Number::s_metaClass(intNumber::s_metaClass,
                                        int64Number::s_MakeInstance);
MetaClass int128Number::s_metaClass(intNumber::s_metaClas,
                                        int128Number::s_makeInstance);
```

The MetaClass constructor takes as first argument a pointer to another MetaClass object which is associated with the baseclass. This argument is needed to register the derived class at the baseclass. As intNumber has no baseclass, this member is NULL and no registration is needed.
Assume a client calls intNumber::s_metaClass::Make(„12,345,678,901"). The client expects a pointer to an object appropriate to represent the number given in the string. The given number will not fit into a intNumber object as it is assumed that it takes only 32-bit values.
The following steps will then be executed:

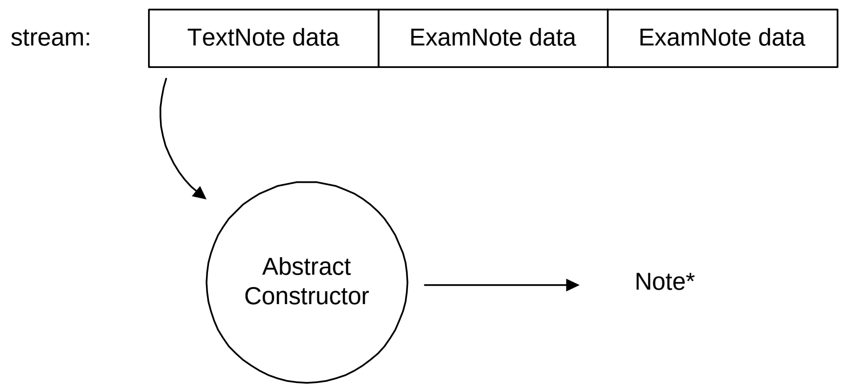
1. intNumber::s_MetaClass::Make() checks, whether it m_makeService is NULL.
2. intNumber::s_MetaClass::m_makeService is not NULL, but points to intNumber::s_MakeInstance() so intNumber::s_MakeInstance() will be called. The parameter is the string „12,345,678,901".
3. intNumber::s_MakeInstance() checks the given string and finds out that it this is not a number that can be represented using 32 bits. The function returns NULL, indicating it couldn't handle it.
4. intNumber::s_MetaClass::Make() checks, whether m_listOfDerivedClasses contains any elements and iterates through the list.
5. The first element called is: int64Number::s_metaClass::Make(). Again the parameter is the string „12,345,678,901".
6. int64Number::s_metaClass::Make() checks, whether it m_makeService is NULL.
7. int64Number::s_MetaClass::m_makeService is not NULL, but points to int64Number::s_MakeInstance() so int64Number::s_MakeInstance() will be called. The parameter is the string „12,345,678,901".
8. int64Number::s_MakeInstance() checks the given string and finds out that it is capable of handling the number.
9. int64Number::s_MakeInstance() calls an ordinary constructor of class int64Number, fills the new object and returns the pointer to the new object to int64Number::s_metaClass::Make().
10. int64Number::s_metaClass::Make() returns the pointer to intNumber::s_metaClass::Make()
11. intNumber::s_metaClass::Make() returns the pointer to the client.

Example 2 (A stream is given)

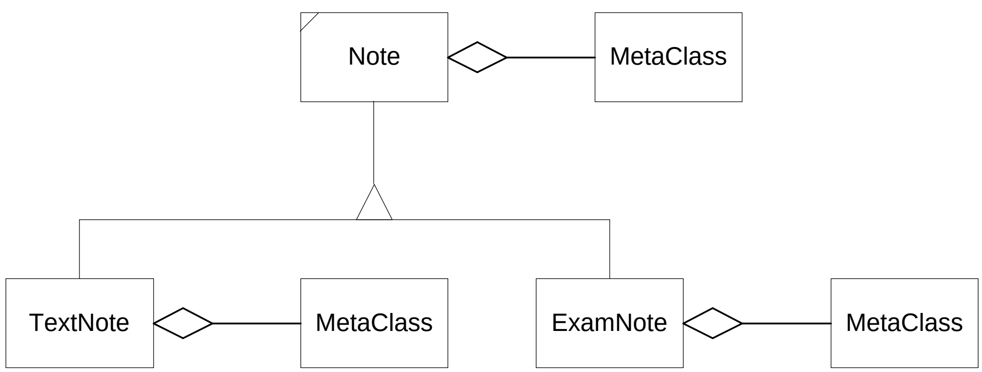
Consider a hierarchy of Note objects:



A set of TextNote objects and ExamNote objects are serialized in the stream. In addition to the note data, each note has a header in the stream, which identifies the type of the note. This type identifier is used as parameter for the Abstract Constructor. We want to read them from that stream:



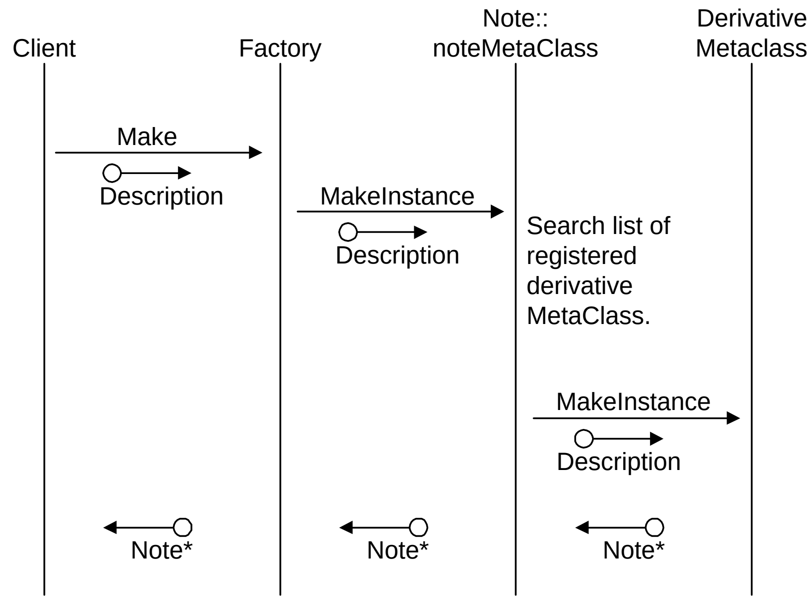
Each note derivation has a static member (see next figure):



Each MetaClass object is instantiated prior to the invocation of main() through static initialization. The following statements are examples for the instantiation of the objects of class MetaClass:

```
MetaClass      Note::s_metaClass(     0,                     // No baseclass
                                      Note::MakeInstance);
MetaClass      TextNote::s_metaClass( Note::noteMetaClass,
                                      TextNote::MakeInstance);
MetaClass      ExamNote::s_metaClass( Note::noteMetaClass,
                                      ExamNote::MakeInstance);
```

The MetaClass objects are constructed with the address of the parent MetaClass with which they register and the address of a function that takes the object description as an argument and returns the properly constructed object. When an object needs to be created from a description, the following scenario occurs:



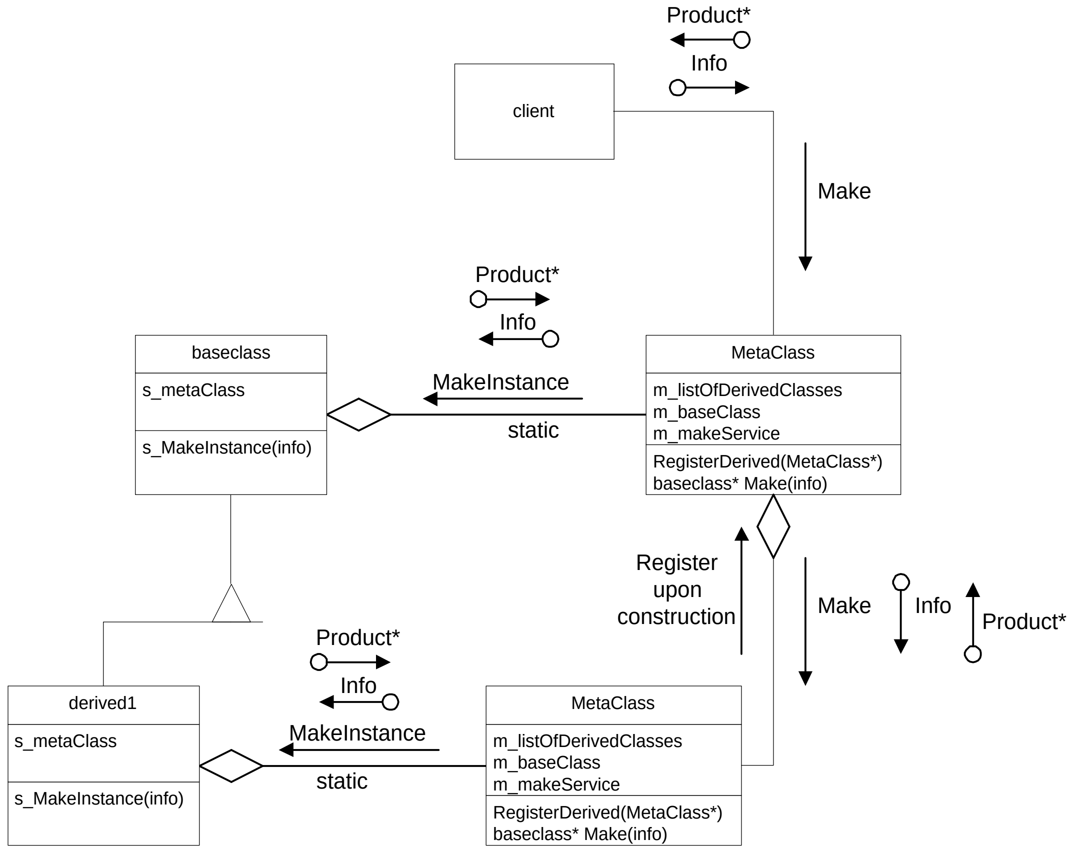
It is possible to use 'MakeInstance' to check the typeids and the formatting of the description.

Consequences

Note: + indicates positive consequences, **-** indicates negative consequences)

+ Allows flexible creation of objects from various sources and description formats, i.e. data files, IPC communication, pipes, internetworks, etc.
+ Allows creation of appropriate object from a dual hierarchy. The creating object passes the appropriate description to the factory.
+ It adds RTTI, or can use it if it is build into the programming language.
+ Allows for changing the type of an object by passing the description to the appropriate factory
+ Cross language/Cross platform portability of objects, it this pattern is used for storing objects.
– Specific primarily to statically typed languages.
– Unique Id's must be allocated as enums or is yucky. (Unique Id's can be used instead of strings containing the classnames when reading from a stream in order to identify which object should be created.)
– It is a pain to define and implement the static members in each derivative.
– Current solution does not support multiple inheritance.
– The order of static initialization is not specified in all languages. (Note: For C++ an implementation is available that is independent of the order of static initializations.)

Structure



Implementation Issues

- It is possible to implement this pattern using C++ templates.
- Some languages do not guarantee a certain order in which static members are initialized. Therefore an approach must be taken to make sure that the initialization of the different static members is done correctly, i.e. without memory leaks or access violations.
- The current implementation requires the client to call baseclass::s_metaClass::Make() (where baseclass needs to be replaced by the 'real' baseclass) in order to call the abstract constructor. A different way would be to add a static Make() member to baseclass which hides the static MetaClass member from the client[2].
- Sometimes it might be useful to make an object of a derived class, although a class which is located higher in the inheritance tree might be appropriate, too. One possible solution is that the T_CREATE_INFO also includes a description of some kind, so that a different object can be created depending of some other facts, e.g. precedences, rules, etc.
- If more than one class claim the same service to be their 'makeservice' this means that the specified 'makeservice' needs to distinguish between the different classes in order to create the correct object.[3]

The following section gives some more details on how some of these issues can be solved.

Implementation Of The Registration Mechanism

In some programming languages the initialization order of static variables is not determined. As the MetaClass member variables are static, this leads to the problem that if the MetaClass member of a derived class initializes and

[2] Thanks to Steve Berczuk for this enhancement.
[3] Initialization of the same static more than once leads to a linker error.

tries to register at the baseclasses MetaClass before the latter has been initialized, execution of the code on a non-initialized object will lead to unpredictable results.
One solution would be that the declaration of all static variables is put in one single source file and the declarations are put in the appropriate order. The first disadvantage is, that if you want to add classes, you need to change existing code which probably has already been tested. The second disadvantage is, that this approach might be error-prone, as you need to know the inheritance tree and that you need to 'describe' it manually by entering the MetaClass member variables into the source file.
The better solution is to let the development environment take care of properly initializing the static member variables. In the current implementation of the pattern this is done in two steps:
Step 1: Collect all MetaClass objects in a common list during startup time of the application.
Step 2: When the Make() service is called for the first time pick all objects from the list built in step 1 and register all MetaClass members at the MetaClass members of their base class.
In the following these two steps are explained in more detail:

Step 1

The first step is independent of the order in which the different static MetaClass objects are initialized during startup. Initialization of a MetaClass object means calling its constructor, e.g.

```
MetaClass intNumber::s_metaClass( NULL,              // No baseclass.
                                  intNumber::s_MakeInstance );
MetaClass int64Number::s_metaClass(   intNumber::s_metaClass,
                                      int64Number::s_MakeInstance);
MetaClass int128Number::s_metaClass(  intNumber::s_metaClas,
                                      int128Number::s_makeInstance);
```

Within the constructor the MetaClass object adds itself to the common list of unregistered classes:

```
template< class T, class T_CREATE_INFO >
MetaClass< T, T_CREATE_INFO >::MetaClass( const char* className,
                                          MetaClass* baseClass,
                                          T* (*makeService)(T_CRATE_INFO))
{
   // Initialize member variables.
   // ...

   this->RegisterDerived( this );
}

template< class T, class T_CREATE_INFO >
void MetaClass< T, T_CREATE_INFO >::RegisterDerived( MetaClass* derived )
{
   CObList*   unregisteredClasses = NULL;

   unregisteredClasses = this->GetUnregisteredClassesObList();
   if( NULL != unregisteredClasses )
   {
      unregisteredClasses->AddTail( derived );
   }// if
}

template< class T, class T_CREATE_INFO >
CObList* MetaClass< T, T_CREATE_INFO >::GetUnregisteredClassesObList()
{
   static CObList unregisteredClasses;
   return &unregisteredClasses;
}
```

The static variable within the GetUnregisteredClassesObList() service is initialized the first time the service executed only. This means that it is initialized once and then reused by all objects of class MetaClass.

Step 2

During step two, the different MetaClass objects will be connected to each other, when the Make() service is called for the first time:

```
template< class T, class T_CREATE_INFO >
T* MetaClass< T, T_CREATE_INFO >::Make(T_CREATE_INFO createInfo)
{
  CObList*   unregisteredClasses = NULL;
  // ...

  unregisteredClasses = this->GetUnregisteredClassesObList();
  if( NULL != unregisteredClasses )
  {
    if( FALSE == unregisteredClasses->IsEmpty() )
    {
      for( pos = unregisteredClasses->GetHeadPosition();
           NULL != pos; )
      {
        currentMetaClass = (MetaClass*) unregisteredClasses->GetNext(pos);
        if( NULL != currentMetaClass->m_baseClass )
        {  // Yes, it has a base class
          (currentMetaClass->m_baseClass->m_listOfDerivedClasses).AddTail(currentMetaClass);
        }// if
      }// for
      unregisteredClasses->RemoveAll();
    }// if
 }// if
}
```

When this service is called the second time, the list of unregistered classes is empty and therefore the for-loop is not executed again, which means the derived classes will not be registered for a second time.

Known Uses

1. HP Series 50 TraceVue is a medical system where this pattern has been used in four different places. One use is to retrieve serialized objects from a file. Another use is to attach elements of the graphical user interface to objects of the problem domain component.
2. Within the customer support organization of HP there are at least two projects using the Abstract Constructor pattern in several places. In addition to the above mentioned uses (de-serializing objects, attaching GUI elements), the Abstract Constructor pattern is used to bind objects within one hierarchy tree to objects in a second hierarchy tree.

Related Pattern

Interpreter, abstract factory, both found in [Gamma+94]
Parser/Builder by Steve Berczuk [Berczuk95]
Virtual Constructor by James O. Coplien [Coplien92]
The abstract constructor is similar to the „Parser/Builder" and to the „Virtual Constructor" patterns. The major differentiator is, that the „abstract constructor" described in this paper has a registration mechanism which allows adding classes without any changes to existing classes and without the need for having a certain initialization order of the used static variables.

Literature

[Berczuk95] Steve Berczuk: „Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams", Submission to the 1995 Pattern Languages of Programs conference
[Coplien92] James O. Coplien: „Advanced C++ Programming Styles and Idioms", Addison-Wesley, 1992
[Gamma+94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.