Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 13 | More About Object Oriented Programming

Reference counting

To avoid memory leaks it is essential to keep track of how many pointer references there are to a given class instance on the heap. In the case where many different functions are accessing the same instance, it becomes unclear where the deletion should be done. Consider the following example.

```
//complex_data definition
class complex_data {
   public:
     complex_data(complex const& c);
     ~complex_data();
     //return the data
     complex* data(){return data_;}
   private:
     complex* data_;
}

//complex_data implementation
complex_data::complex_data(complex const& c) {
  data_ = new complex(c);
}
```

The class **complex_data** is responsible for storing a complex number on the heap, for use by many different functions.

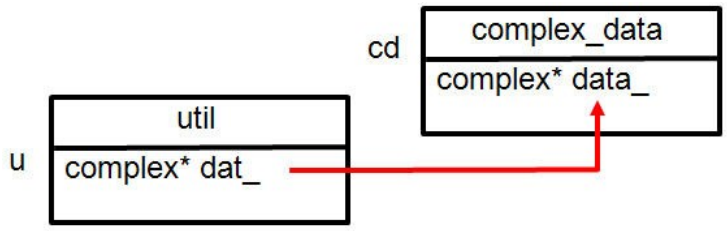
Next suppose that a function **util** is defined that stores a pointer to one element of the data in the dataset.

```
// a class using a complex pointer
class util {
   public:
     ~util();
     void set_data(complex* dp){dat_=dp;}
   private:
     complex* dat_;
}

...
```

The interaction between these two classes might be as follows.

```
polar_impl pi(3,1.57);
complex_data cd(complex(pi));

//assign the complex numbers specific values
…
util u;
u.set_data(cd.data());
…
```

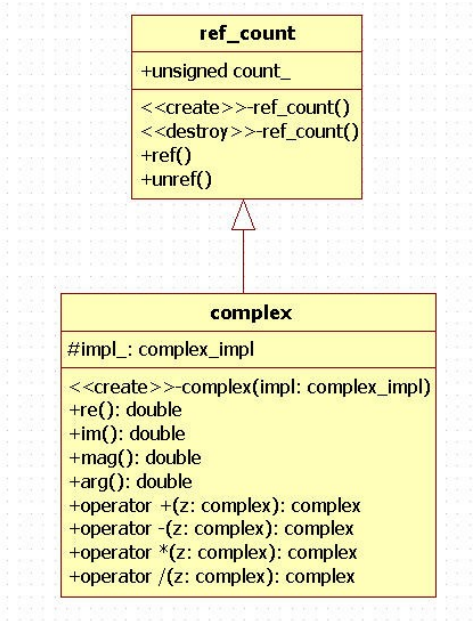
Here the **complex** element of **cd** is being attached to a pointer inside the **util** class, as shown below.



The question is when can this heap memory be deallocated? One possibility is that it can be done in the destructor of **complex_data**, i.e.,

```
complex_data::~complex_data() {
  delete data_;
}
```

However, it is not known whether **util** instance **u** is still around or not at the time the destructor is called. It isn't nice deleting the **dat_** member of **util** behind its back! What about delteting the **dat_** element in the destructor of **util**? That move is also bad, since other classes, other than **util** may be holding a pointer to the data element as well. One way out of this trap is reference counting. We can augment the complex class as follows.



A new class is implemented that keeps a reference count that is incremented by **ref()** and decremented by **unref()**. The code for these two methods, defined in the class specification, is as follows.

```
class ref_count {
 public:
  ref_count() : count_(0){};
  virtual ~ref_count(){};
  void ref() { ++count_; }
  void unref() { if (--count_ <= 0) delete this; }
  private:
  int count_;
};
```

Note that **unref()** does one additional operation. When the **count_** value is pre-decremented to less than or equal to zero the **ref_count** class deletes itself.

The class specification for **complex** is adapted as follows.

```
class complex : public ref_count {
…
```

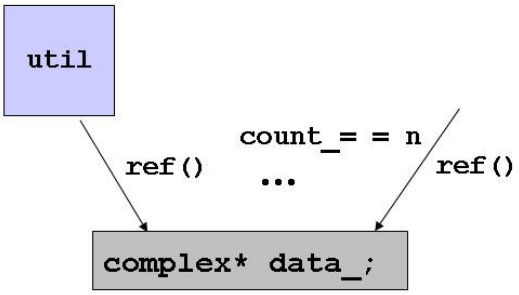
Note that the **~ref_count** destructor is made a virtual method so that the destructor of subclasses can take appropriate actions on being eliminated. The destructor of **ref_count** itself doesn't have to do anything.

The set method on **util** is modified as follows.

```
void set_data(complex* dp){dat_=dp; dat_->ref();}
```

The count of users of the **complex** data item is increased by one when it is assigned to the **dat_** pointer inside **util**. The change to the methods of **complex_data** are as follows.

```
complex_data::complex_data(complex const& c) {
  data_ = new complex(c);
  data_->ref();
}
```



Thus each user of **data_** increases the value of **count_** as shown above.
As each class that holds a pointer to **data_** is deleted it reduces the value of **count_** e.g.,

```
complex_data::~complex_data() {
  data_->unref();
}
```

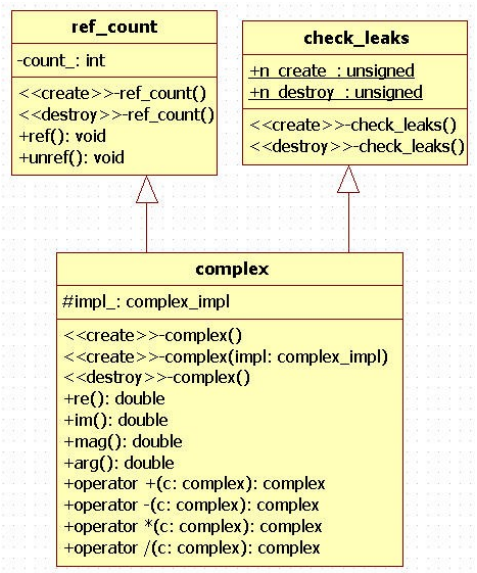
Now the memory is being managed properly. The last user of **data_** will actually trigger the deletion even though instance **cd** may go out of scope and lower **count_** by 1. If **count_** isn't less than or equal to zero, the memory isn't deleted.

Multiple Inheritance

The use of the inheritance relationship between classes was introduced in previous Lectures. It is sometimes necessary to derive more than one interface in the child class. A good example would be a mechanism to keep track of creations and deletions of class instances. This capability is very useful for investigating memory leaks and should be available for use by other classes. A simple class can be defined that carries out the matching of creations and deletions.

```
class check_leaks {
public:
  check_leaks() { ++n_create_; }
  virtual ~check_leaks(){++n_destroy_;}
private:
  static unsigned n_create_;
  static unsigned n_destroy_;
};
```

The specification of class **complex** can be altered to inherit from both reference counting and leak checking capability as follows. The UML diagram is,



The synatx of this *multiple inheritance* is a comma-separated list after the `':'` operator. Each item has the privacy specified separately as follows. If **`public`** were not repeated then **`check_leaks`** would be assumed to have **`private`** inheritance.

```
# include "check_leaks.h"
# include "ref_count.h"

class complex : public ref_count, public check_leaks {
...
```

The values of static member variables, **`n_create_`** and **`n_destroy_`**, are initialized to 0 outside of the class scope, say in the main program, as follows.

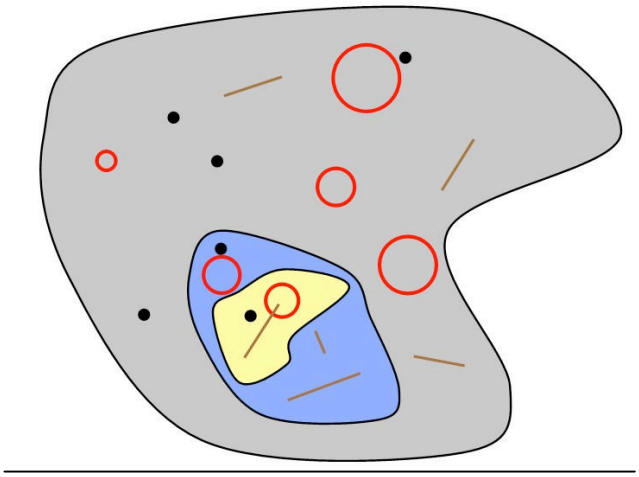
```
# include <iostream>
# include "complex.h" using namespace std;
// typically initialized in check_leaks.cxx
// unsigned check_leaks::n_create_ = 0;
// unsigned check_leaks::n_destroy_ = 0;
int main() {
  cartesian_impl ctimpl(-1.0, 0);
  polar_impl plimpl(1.0, 3.1415926535897932);
  {
    //note cpx1 and cpx2 are local to this scope
    complex cpx1(ctimpl);
    complex cpx2(plimpl);
    // n_create_==2 at this point,n_destroy_==0
  } // leaving scope, 2 destructor calls
  complex cpx3(ctimpl);
  // at this point n_create_==3 n_destroy_==2
  if(check_leaks::n_create_!=check_leaks::n_destroy_)
    cout << "Memory leaks detected!" << endl;
}
```

Since **cpx3** was constructed but not yet deleted, the output of this program will be,

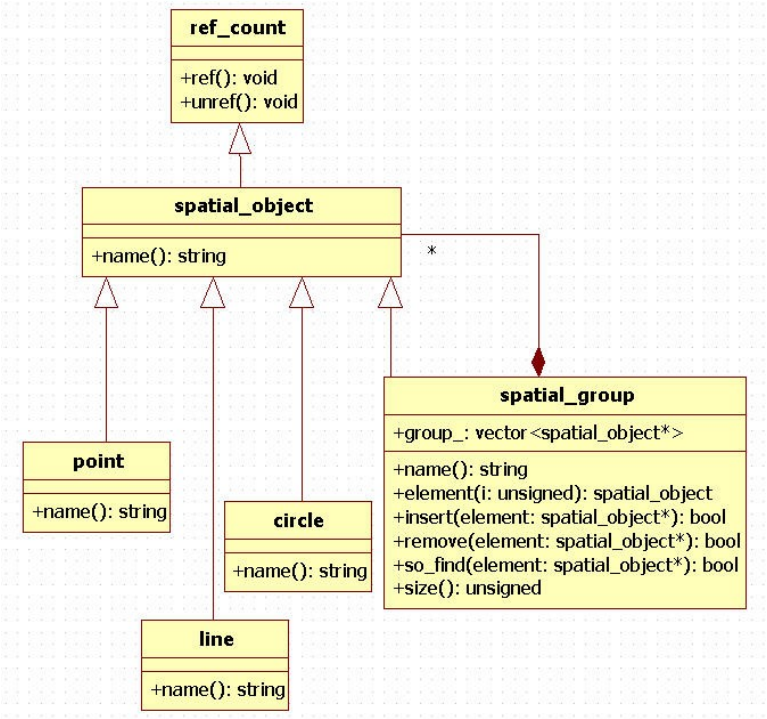
```
Memory leaks detected!
```

Normally, all function scopes except the main program will have exited by the time the last statement in the main program is reached and so the true status of memory leaks will be reported.

Spatial groups



Consider the grouping of spatial objects in the figure above. There is an overall collection of points, circles and lines, indicated in grey. The grey collection also includes other collections, … and so on.



This recursive group structure can arise in any situation where there is an inherent part-whole relationship. For example, the components of a vehicle, such as the engine, also have multi-element components.

An implementation of a hierarchical collection of spatial objects is shown in the UML diagram above.
There is a generic base class, the **spatial_object**, with subclasses being particular kinds of spatial objects such as **point**, **line** and **circle** classes. There is a virtual method, **name()**, which returns a string name of the **spatial_object** type.
So far, this aspect of the design is similar to that for **curve**, introduced in Lecture 1.

The spatial object hierarchy inherits a **ref_count** interface so that pointers to **spatial_object** class instances can be safely stored in the group and cast to appropriate subclasses.

The new aspect of the design is the **spatial_group** , which is a subclass of **spatial_object**. The **spatial_group** class has a member, **group_**, which contains a set of **spatial_object** pointers.

Thus, a group of spatial objects can itself be a **spatial_object**.

An implementation of **spatial_group** is as follows. The `spatial_group.h` file is shown below.

```
#ifndef spatial_group_h_
#define spatial_group_h_

#include <vector>
#include <string>
#include "spatial_object.h"  //Trivial implementation, not shown
using namespace std;

class spatial_group {
 public:
  spatial_group(){}
  virtual ~spatial_group();
  virtual string name(){return "spatial_group";}
  bool insert(spatial_object* so);
  bool remove(spatial_object* so);
  bool so_find(spatial_object* so);
  unsigned size(){return group_.size();}
  spatial_object* element(unsigned i);
 private:
  vector<spatial_object*> group_;
};

#endif  // spatial_group_h_
```

The contents of the `spatial_group.cxx` file follows below.

```
#include "spatial_group.h"
#include <algorithm>

// The destructor. Note that unref() is called on elements in
group_
spatial_group::~spatial_group() {
  vector<spatial_object*>::iterator vit;
  for(vit=group_.begin();vit!=group_.end();++vit)
    (*vit)->unref();
  group_.clear();// empty out the potentially dead objects
}

// insert a group element. Return false if it already exists
bool spatial_group::insert(spatial_object* so) {
  if(!so) return false; // a null object
  vector<spatial_object*>::iterator vit =
    find(group_.begin(), group_.end(),so);
  if(vit == group_.end()) {
    // up the refcount since group_ has ownership
    group_.insert(so);
    so->ref();
    return true;
  }
  else return false;
}

// remove an element. If it doesn't exist return false bool
spatial_group::remove(spatial_object* so) {
  if(!so) return false; // a null object
  vector<spatial_object*>::iterator vit =
    find(group_.begin(), group_.end(),so);
  if(vit == group_.end())
    return false;
  // so is no longer owned by the group
  group.erase(vit);//remove from group_
  so->unref();
  return true;
}

// Find if an element is in the group
bool spatial_group::so_find(spatial_object* so) {
  if(!so) return false; // a null object
  vector<spatial_object*>::iterator vit =
    find(group_.begin(), group_.end(),so);
  if(vit == group_.end())
    return false;
  return true;
}
```

```
// Return a specific element of the group
spatial_object* spatial_group::element(unsigned i) {
  if( i >= group_.size() ) return 0;
  return group_[i];
}
```

The application of this design is shown below.

```
#include "spatial_group.h"

...
spatial_group sg;
string sgn = sg.name();// sgn == "spatial_group"

spatial_object* soa = new spatial_object();
spatial_object* sob = new spatial_object();
sg.insert(soa);
sg.insert(sob);

bool fa = sg.so_find(soa);// fa == true
bool rm = sg.remove(sob);// rm == true, destructor of sob called
rm = sg.remove(sob); // rm == false

spatial_object* soc = sg.element(0); // actually is soa
soc->ref(); //protect against deletion by someone else
```

The implementation just described is an example of a *design pattern* called the *composite* pattern. This design is applicable whenever a structure contains itself as a component.
The implementation would be almost identical for electrical components as for spatial objects or any other hierarchical structures.

The formalization of such reusable patterns was first popularized by the book,

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

The so-called "gang of four," GOF, are shown below.

The concept of a design pattern is specified as follows:

1) The purpose of the pattern:
   creational – defines the process of object creation;
   structural – defines the composition of classes in terms of other classes;
   behavioral – defines the process interaction between classes.

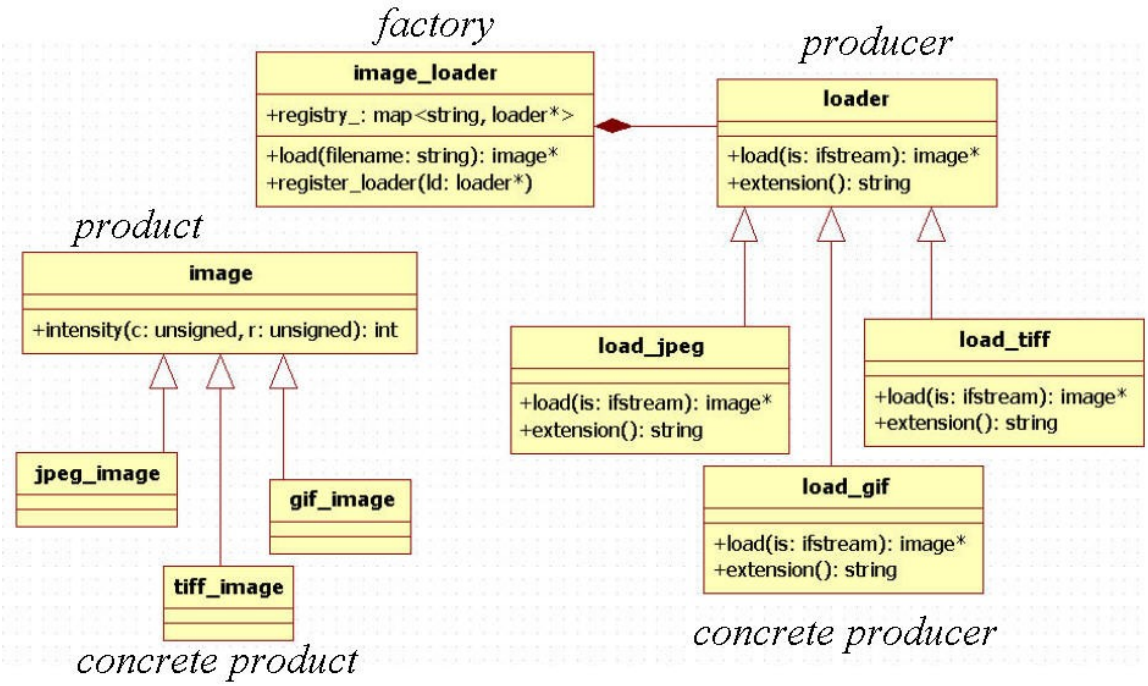
2) The representation of a pattern:
   name – a descriptive name that helps connect the pattern to the problem;
   problem – a specification of the problem that the design is meant to solve;
   solution – a generic description of the solution that suppresses details of any specific situation;
   pros and cons – an analysis of the strengths and weaknesses of the design.

The book outlines 23 patterns of creational, structural and behavioral types. This lecture has so far described the *composite* pattern, which is in the structural category. The composite pattern is described in the book starting on page 163.

It will be impossible to cover all of these patterns in this course. However, a few more examples are presented to illustrate the design pattern philosophy and its power. Note that the patterns are not restricted to C++ but could be implemented in any language. However, many of the pattern concepts are naturally expressed as classes.

Factory patterns

It can often be the case that resources available to a process are unknown until runtime and actually may be augmented on the fly. A typical example is the availability of a viewer for a particular file format, e.g. `vrml`, `pdf`, `ps`, etc. If the process attempts to load a file, it needs to determine if a reader is available. At the same time, a client of the program should have the facility to add a new reader without any recoding of the main application.



The *factory* design pattern is shown in the figure above. The factory is a *creational* type of pattern. In this example, the problem has been specialized to loading images of various image file formats. The image sub-classes (*products*) account for accessing pixel intensity values according to the layout of the particular image file format.

The image loaders have specialized knowledge about the header information in the image file. For example, the jpeg image format is compressed and the jpeg file header gives information needed to uncompress the image. In addition, each **`loader`** can return its unique extension string via the **`extension()`** method.

The *factory*, **`image_loader`**, provides the interface to other parts of the program (clients) that have need to load an image. The specific type of loader (*producer*) is dispatched based on the filename extension, e.g. `file.tif`, `file.jpg`, etc.

The **image_loader** maintains a registry of available loaders. This registry is implemented using a **map** as shown in the image_loader.h file.

```
#include <map>
#include <string>
#include "loader.h" using namespace std;

//forward declare image
class image;
class image_loader {
 public:
  image_loader() {}
  ~image_loader() {}
  image* load(string const& filename);
  void register_loader(loader* ld);
 private:
  // map associates file extension to loader
  map<string, loader* > registry_;
};
```

The load and register_loader methods are implemented as follows.

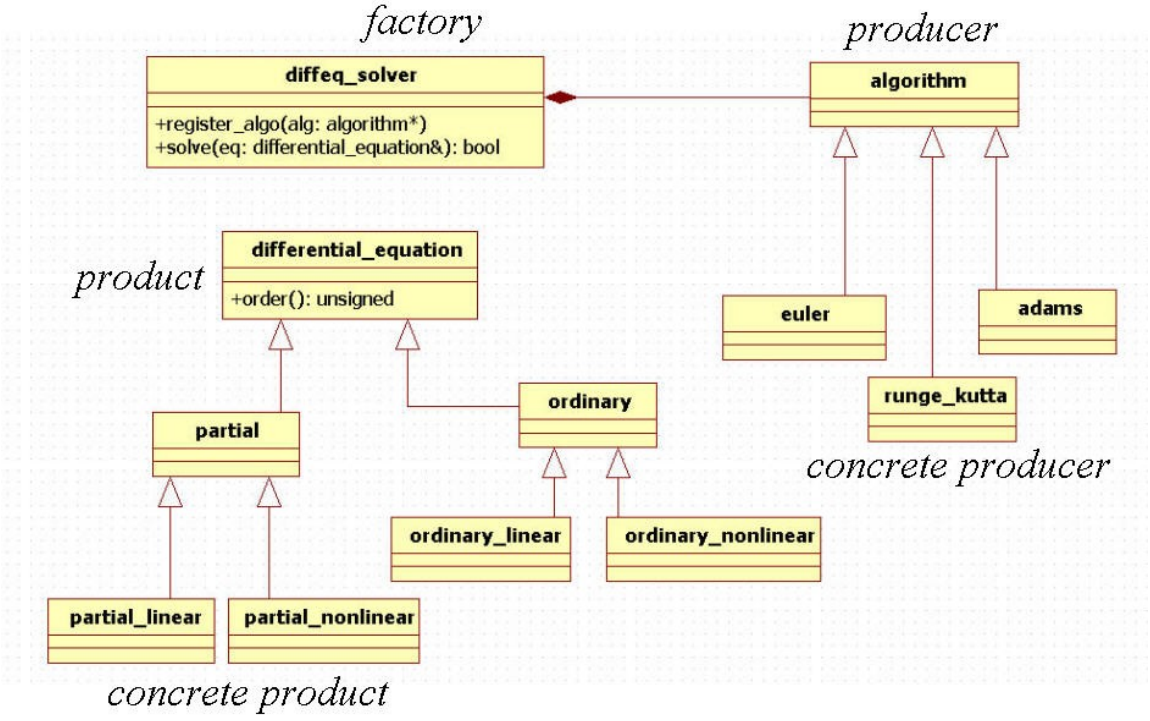
```
#include "image_loader.h"
#include <fstream>

void image_loader::register_loader(loader* ld) {
  string ext = ld->extension(); // the defined extension
  // associate file extension with the loader
  // e.g. tif, jpg, gif ...
  pair<string, loader*> p(ext, ld);
  registry_.insert(p);
}

image* image_loader::load(string const& filename) {
  // construct the stream
  ifstream is(filename.c_str());
  //the file may not exist
  if(!is) return 0;

  //search for extension
  unsigned i = filename.find('.');
  if(i == string::npos ) return 0;  // not found

  // extensions are 3 characters
  string ext = filename.substr(i+1, 3);
  //get the loader
  loader* ld = registry_[ext];
  if(!ld) return 0; // may not be in the map
  return ld->load(is);
}
```

The client of the **image_loader** factory will access the factory as shown in the code fragment below[1].

```
//during setup part of execution
image_loader il;
il.register_loader(new load_jpeg);
il.register_loader(new load_special);

...

//during routine operation, the suitable loader is retrieved
image* img = il.load("myfile.ext");
```

The factory pattern is applicable to any situation where there are heterogeneous sets of producers and products. The factory itself is freed from having to account for the details of how the products are made and the details of the product itself.

To show the wide applicability of the factory pattern, consider its application to an object- oriented design of differential equation processing.



This design is just a sketch, but it illustrates how a real system might be implemented. There is a *factory* component that solves differential equations supplied by a client. The

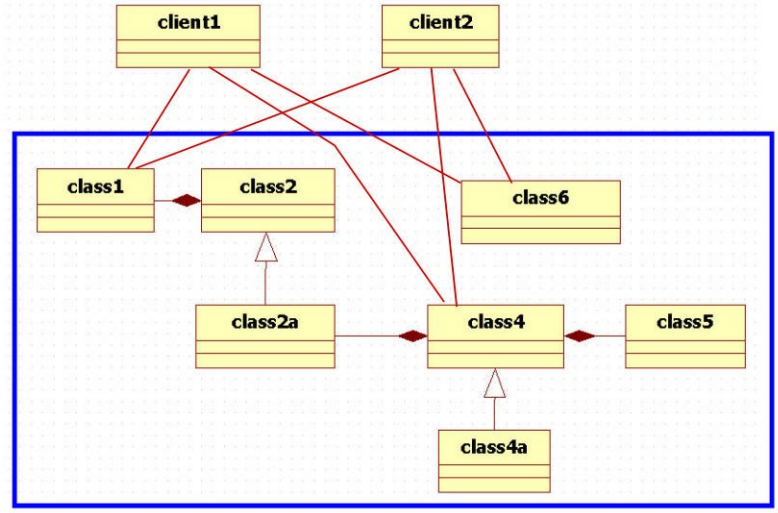
[1] It would be desirable to have only one instance of the **image_loader** factory that can be accessed everywhere in the application. This result can be achieved using the *singleton* design pattern. See http://msdn.microsoft.com/en-us/library/Ee817670(pandp.10).aspx for details.

factory uses its registry to retrieve an appropriate algorithm for the particular type of differential equation.

In this sketch, it is assumed that the differential equation instances (*products*) contain structures to represent the solution. There are a set of registered algorithms (producers) that can solve differential equation of various types.

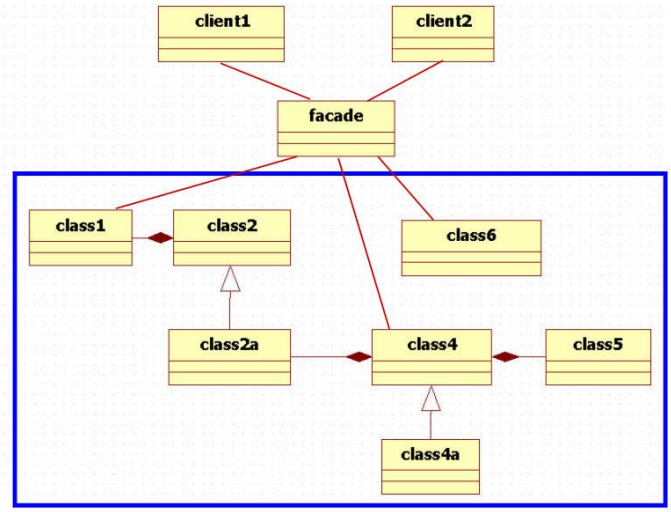
The façade pattern

An important type of *behavioral* pattern is the *façade*. Suppose there is a relatively complex set of interactions that a client has to perform to compute a result as shown below.



Each user (*client*) of the class library inside the blue box has to repeat these complex interactions to achieve a given task. The *façade* class encapsulates these diverse operations and interactions into a simple interface that hides the hard work behind the scenes.

The façade implementation for the same situation is shown below.



Now the interaction is neat and tidy, with the façade taking care of the details.

A concrete realization of the façade pattern will be taken up a future assignment.