The Capsule Pattern

A design pattern that reduces coupling between application layers.

Robert C. Martin

Introduction

Sometimes you need the lowest levels of your application to communicate with the highest levels, without the intervening levels knowing much about that communications. For example, the functions at the lowest software levels may experience errors that they need to communicate to the topmost layers. If these errors require complex data structures to describe, and if these structures are passed as return values or reference arguments, then all intervening layers of the software will have to know about those data structures. Then, when the internals of those data structures change, all the intervening modules will have to be recompiled.

This is a traditional problem in software engineering that goes back to the days when we put a bunch of #defines in a file named error.h and returned those values from our functions. Everybody #included error.h, so adding new error types always meant that every module would need to be recompiled[1]. To avoid the spectre of long compiles we would often try to be inventive with the kind of error code we returned. After all, returning RT_MEMERR when you tried to pop an empty stack isnÕt really *that* misleading.

Nowadays, it is not just error codes that we want to pass back up the calling hierarchy; we also want to pass up complex data structures that describe the error. This allows our complex human interfaces to describe the low--level problem in precise detail.

1. We used to shoot people who tried to thwart the make system by touching all the ÒunaffacedÓ .o files.

This detail may be displayed to the user, or may simply be added to a log file as a debugging aid. Or, that extra detail may be used by higher level functions as a means to choose ~~corresponding~~ remedial action.

So, how do we solve the problem of the dependencies? How can we rig it so that the middle level modules do not have to #include the header files that describe the return types that the low level modules want to pass back to the high level modules?

Exceptions

One answer is to throw exceptions. Middle layer modules can declare the types of exceptions that they can handle. Other exceptions simply pass right by them. Thus we could throw our complex error types from the low levels and catch them in the high levels without the middle levels knowing anything about them.

This approach works well in some cases, but there are other cases where it presents significant problems. First of all, the cost of using exceptions is high. Just Òturning onÓ exceptions in some compilers causes *all* functions to take longer to execute.

Second~~ly~~, the act of throwing and then processing an exception can be costly in terms of CPU time. Not all errors, indeed perhaps only a very few kinds of errors, warrant the cost associated with throwing them as exceptions. The exceptions mechanism in C++ was not meant to be used to communicate simple logical errors. It was meant to handle gross errors like memory outages, or hardware failures.

Furthermore, many errors do not warrant the severity of immediately terminating the detecting scope. Such errors may be remembered while other processing is taken care of, and then carefully returned to the caller.

Finally, in many applications (e.g. Windows apps) the high-~~-~~level modules are separated from the low levels by a foreign stack that you cannot throw exceptions acrossacross~~accross~~. That is, the high level modules written in C++ invoke APIs or system calls that ~~which~~ were not written in C++. These, in turn, call back to the lower-level C++ modules. This means that there is a foreign stack between the high level layers and the low level layers. When an exception is thrown from the lower layers, it walks up the stack looking for an exception handler. If it walks into the foreign stack it will not recognize its structure, and will then call the terminate function.

So~~,~~ it appears that there are cases in which using exceptions to communicate complex errors is not a viable options, and that such errors must be passed back up the calling hierarchy as return values or reference arguments. How do we prevent the middle layers from having to #include the error types?

void*

Another solution might be to pass the error back as a void*. The higher layers could then cast the error back to its original type and access the data within it. The middle layers simply pass the error around as a void* and donÕt need to #include the error header file. This solution works well so long as the high level modules are *sure* of the type of the error. Such surety can be achieved, for example, by using only one kind of data structure for all the various types of errors. For example:

```
struct Error
{
 enum Code {e1,e2,e3} itsErrorCode;
 union
 {
  struct E1 itsE1;
  struct E2 itsE2;
  struct E3 itsE3;
 } itsErrorData;
};
```

This data structure contains a type code, plus data fields for all the data needed by any of the various errors. This scheme works well~~,~~; however there is a disadvantage. Whenever some new error type is needed, or whenever some low level module needs a new field added to one of the existing error types, all the old low level modules -- and all the old high level modules -- must be recompiled. Thus, all we have protected are the middle layer modules. The creators and users of the errors are still strongly coupled through the Error structure. Changes to the header file that describes this structure still cause extensive ~~large amounts of~~ recompiling.

Common Base Class

Yet another solution to this problem is to create something similar to the following base class:

```
class Error
{
 public:
  virtual ~Error() {}
};
```

We can then derive the different error types from this Error class. This allows the low level functions to create an error object, upcast it to an Error, and return it. The middle layers all #include Òerror.hÓ~~,~~; however, this is not harmful since this class is virtually empty. It will never change, and so the middle layers will never change. The high levels can then downcast the Error to the appropriate derivative by using RTTI.

```
Error* e = f(); // function returns Error
if ((MyError* me = dynamic_cast<MyError*>(e)))
{
  // process the MyError.
}
```

This method works quite well but has two disadvantages. Firstly, dynamic_cast has unknown timing characteristics. We donÕt know how long it will take to execute. Moreover, the time it takes may well be related to the width and depth of the inheritance hierarchy for Error.

The second disadvantage is ~~a bit~~ more severe. It presumes that there is only one client at the higher levels that needs to access the information supplied by the lower levels. However, in many systems such an assumption would be ~~a bit~~ naive. In such systems, error messages from the lower levels are passed along to many different clients~~;~~, each having ~~of whom have~~ different needs~~,~~ and plac~~e~~ing independent requirements on the data contained within the error message.

One could imagine, for example, that among the clients interested in the error information are a GUI that displays the error~~,~~ and a~~n intelligent~~ module that analyses the error and takes remedial action. The GUI is going to want to know the type, location, and severity of the error so that it can create messages, icons, and colors that reflect that information. The remedial function may not care about severity or location~~,~~ but may need an IO address and/or the address of some object X that it can use to take remedial action.

If we use the above solution, the error class must be written so ~~such~~ that all the data that any of its clients need is present in the error. Thus when any of those clients needs new data to be added, all the clients must be recompiled. In many systems, this kind of coupling between otherwise independent clients is unacceptable.

Cross Casting

The dynamic_cast feature of C++ affords another kind of solution -- cross casting. Consider the following code.

```
class A {public: virtual ~A();};
class B {public: virtual ~B();};
class C : public A, public B {};

A* ap = new C;
B* bp = dynamic_cast<B*>(ap);
```

Notice that classes A and B are completely unrelated. Now when we create an instance of C we can safely upcast it to an A*. However, we can now take that pointer to A and *cross cast* it to a pointer to a B. This works because the A pointer ÔapÕ really points at a C object; and C derives from B.

Thus, we have cast *across*~~*accross*~~ the inheritance hierarchy between completely unrelated classes. It should be noted that this will *not* work with regular casts since they will not be able to do the address arithmetic to get the pointer to B correct. For example:

```
B* bp = (B*)ap;
```

While this will compile without errors[2], it will not generate working code. The value of ÔbpÕ will not actually point to the B part of C. Rather it will still point at the A part of C. This will lead to undefined behavior[3].

The same argument holds true for reinterpret_cast~~;~~: undefined behavior will ensue. And static_cast will give you a compiler error since there is no implicit conversion from B* to A*.

So, of the various forms of casts, only dynamic_cast gives us the power to properly cast across~~accross~~ an inheritance hierarchy.

Now, how can we use this ability to solve the problem of hiding the error data structure from the middle layers of the application as well as keeping the clients of the errors isolated? We can employ the Interface Segregation Principle[4] (ISP). Consider the following code:

```
class Capsule
{
  public:
    virtual ~Capsule() {};
};

class GUIError
{
  virtual ~GUIError();
  virtual int GetErrorCode() = 0;
  virtual Severity GetSeverity() = 0;
  virtual Location GetLocation() = 0;
};

class RemedialError
{
  virtual ~RemedialError();
  virtual int GetErrorCode() = 0;
  virtual IOAddr GetIOAddr() = 0;
  virtual X* GetX() = 0;
};

class HiddenError : public Capsule
                  , public GUIError
                  , public RemedialError
{
  // implement all the pure virtuals to return
  // the error data needed.
};
```

Now the low level level functions that detect the error can build a HiddenError object and then pass it up the calling chain as a Capsule*. The higher levels can cross-cast the

2. A good compiler might give you a warning.

3. The true meaning of Òundefined behaviorÓ is: ÒWorks in the lab.Ó

4. See ÒThe Interface Segregation PrincipleÓ, C++ Report, August, 1996. Or http://www.oma.com/Publications/publications.html

Capsule* to the kind of error that they need to deal with. For example, the GUI would cross-cast the Capsule* to a GUIError*; while the remedial function would cross-cast the Capsule* to a RemedialError*.

```
Capsule* c = f(); // function returns an error capsule.

// in the GUI
if ((GUIError* ge=dynamic_cast<GUIError*>(c)))
{
  // process GUIError ge.
}


// in remedial function
if ((RemedialError* re=dynamic_cast<RemedialError*>(c)))
{
  // process RemedialError re.
}
```

Notice that this also allows a crude form of query that the clients can use to determine if they need to process an error. For example, those errors for which no remedial action can be taken can simply not inherit from RemedialError. This will prevent the remedial function from paying any attention to the error.

The high-level clients of the error simply #include those header files that describe the specific error types that they are interested in. The low-level creators of the errors simply #include those error types that they need to create. Intermediate modules #include nothing more than capsule.h. When new types of errors are created, or old types of errors are modified, only those modules that are truly affected need to be recompiled. Modules that donÕt care about the changes are left unaffected and do not need to be recompiled.

Deferred Function Invocation

Once the client module has determined that the Capsule can be downcast to an error type that it is interested in, it still has the problem of parsing the error. Consider the following code:

```
class GUIError
{
  public:
    enum code {deviceFailure,
              logicalError,
              communicationsError} itsCode;
    int moduleNumber;
    int systemState;
}

Capsule* c = f(); // function returns an error capsule.

// in the GUI
if ((GUIError* ge=dynamic_cast<GUIError*>(c)))
```

```
{
 switch (ge->itsCode)
 {
 case GUIError::deviceFailure:
   DeviceFailure(ge->moduleNumber,ge->systemState);
 break;

 case GUIError::logicalError:
   LogicalError(ge->moduleNumber, ge->systemState);
 break;


 case GUIError::communicationsError:
   CommunicationsError(ge->moduleNumber,
                 ge->systemState);
 break;
 }
}
```

Here the client has determined that the error conforms to the type GUIError, and then proceeds to parse the error and marshall its contents to the appropriate error handling functions. We can easily imagine that these functions are declared pure virtual in the high level client class, and then implemented in derivatives to display the various errors in different language, formats, or technologies.

This parsing code is problematic. It must change every time a new kind of error code is created. Moreover, if a new error type *is* added, there is no way to enforce that the parsing code selects for it. For example, if the code controllerError is added to the GUIError::Code enumeration, there is no way to enforce that the appropriate case statement is added to the parsing code.

We can address this problem with an interesting technique. Consider the following code:

```
class GUIErrorHandler
{
 public:
   virtual void DeviceFailure(int, int) = 0;
   virtual void LogicalError(int, int) = 0;
   virtual void CommunicationsError(int, int) = 0;
};

class GUI : private GUIErrorHandler
{
 public:
   void DoSomething();
 private:
   virtual void DeviceFailure(int, int);
   virtual void LogicalError(int, int);
   virtual void CommunicationsError(int, int);
}

class GUIError
{
 public:
```

```
    virtual ~GUIError() {}
    virtual void Handle(GUIErrorHandler&) = 0;
};

class GUIDevErr : public GUIError
{
  public:
    GUIDevErr(int module, int state)
    : itsModule(module), itsState(state) {}
    virtual void Handle(GUIErrorHandler& eh)
    {eh.DeviceFailure(itsModule, itsState);}
  private:
    int itsModule;
    int itsState;
};

class GUILogicErr : public GUIError
{
  public:
    GUILogicErr(int module, int state)
    : itsModule(module), itsState(state) {}
    virtual void Handle(GUIErrorHandler& eh)
    {eh.LogicError(itsModule, itsState);}
  private:
    int itsModule;
    int itsState;
};

class GUIComErr : public GUIError
{
  public:
    GUIComErr(int module, int state)
    : itsModule(module), itsState(state) {}
    virtual void Handle(GUIErrorHandler& eh)
    {eh.CommunicationsError(itsModule, itsState);}
  private:
    int itsModule;
    int itsState;
};

void GUI::DoSomething()
{
    //do a bunch of stuff and then
    //call a function that returns a Capsule:
    Capsule* c = f();
    if ((GUIError* ge = dynamic_cast<GUIError*>(c)))
    {
      ge->Handle(this);
    }
};
```

You probably recognize this scheme as a variation of dual dispatch. Actually this has certain similarities to the Visitor[5] pattern,~~;~~ and its cousin,~~also~~ the Acyclic Visitor[6] pattern.

5. Design Patterns, Gamma, et. sl., Addison Wesley, 1995

6. Pattern Language of Program Design 3, Martin,Rhiele,Buschmann, Addison Wesley, 1997

The client GUI is in the midst of executing GUI::DoSomething~~.~~, ~~In the course of this execution it~~ callings~~s~~ a function f that returns an error as a Capsule. As before, the GUI uses dynamic_cast to determine whether or not it can process the error as a GUIError. If it can, then it passes itself to the Handle function of the GUIError class. Note that this is done by upcasting the this pointer to a pointer to its private base GUIErrorHandler. We want GUIErrorHandler to be a private base so that only GUI can upcast itself to a GUIErrorHandler and thereby make its error handling functions available.

The three different kinds of errors are coded as derivatives of GUIError rather than as enumerators in an enumeration. Each derivative implements the Handle function such that it calls the appropriate GUIErrorHandler function~~;~~, passing it the appropriate data elements.

Note that this decouples the GUI class from the details of the GUIError derivatives. Lots of other elements could be placed in the GUIError derivatives without affecting the GUI class. Note also that the actual disposition of the error is now determined by the GUIError derivative rather than the GUI.

This is interesting. It means that we could break the one-to-one correspondence between the error types and the error handling functions. The error types could then implement the Handle function to invoke a sequence of calls to the error handler functions. Thus the Handle functions of each error type would contain a kind of script for disposing of the error.

Its not just for errors anymore.

~~Note that w~~We have created a mechanism whereby a low level element can, by virtue of a value that it returns, invoke a function in a high level element. Moreover, this invocation can be deferred at the discretion of the levels that are higher than the low level element that wants the function invoked. ~~This is interesting.~~ Thise scheme ~~that we have developed on these pages~~ is not just for ~~simply~~ ~~an~~ reporting errors ~~reporting scheme~~. *It is a deferred function calling scheme*. It could be used, for example, in a system where lower levels decide which operation that the high levels should perform. And yet the high levels are not at the mercy of the low levels. The higher levels can decide when (not what, but when) that function will be invoked.

~~Thus w~~We might envision a robotic system where some low-~~-~~level video controller sees a barrier in its path and decides to inform the higher levels that is should consider avoiding that barrier. The higher levels can defer the processing of this information until a convenient time~~.~~ (h~~H~~opefully before collision). We might envision a high-~~-~~level controller that collects a number of return values from various lower level components~~,~~ and then invokes them in the order of their priority; as determined by the higher level.

Moreover, the~~se little~~ capsule objects that are ~~being~~ passed up to the high levels from

---

also see http://www.oma.com/Publications/publicatoins.html the low levels can contain ~~be full of~~ multivariate data. A capsule that means one thing to one high-level element can mean something completely different to other high-level elements.

Reversing the Direction

If the Capsule pattern can be used to provide deferred communication from the lower levels of an application to its higher levels, it can also be used for the reverse. High-level modules can build capsules that are eventually processed by low-level modules. These capsules might be passed to the lower levels in one thread~~,~~ before they are ~~and then~~ invoked in a completely different thread.

One could envision the high-level modules creating a single Capsule to be used by many different low-level modules. The high levels would pass the capsule to each low-level module and then allow those lower levels to invoke their particular part of the capsule at their convenience~~conveinece~~. The capsule ~~then~~ resembles a folder of work orders that gets passed around from agent to agent. Each agent looks in the folder (using dynamic_cast) to see if it contains any work orders specifically for it. If so, it invokes those work orders (by calling Handle on the downcast class). It may not invoke those work orders immediately, it may wait for a different thread, or a special event.

The fact that all the work orders are actually contained in the same object means that object can mediate between the agents. When one agent invokes one of the work orders, the underlying capsule object will know it, and can make sure that the other agents are properly coordinated.

Conclusion

In this article we have discussed a design pattern named Capsule. This pattern is used to provide deferred invocation of functions between software elements that do not communicate directly with each other. The pattern ensures that the intermediary ~~intermedary~~ software elements are not dependent upon the contents and details of the messages that pass between the two indirectly communicating elements.

This pattern makes use of cross-casting. Cross casting is a feature of statically typed OO programming language such as C++, Java, and Eiffel. The Capsule pattern should be applicable to all such languages.

The runtime cost of cross-casting is difficult to quantify. Moreover it may change as the inheritance hierarchies it manipulates change and grow. Thus care must be taken when using this pattern in hard real time applications. We are used to casts taking very little time. Such may not be the case with cross-casts. Caveat Emptor!