Database *as-a* (OO)P language late binding, delegation, inheritance, and the Fragile Base Class problem

Oleg Kiselyov
Computer Sciences Corporation
FNMOC, 7 Grace Hopper Ave, Stop 651   Monterey CA 93943-5501
oleg@pobox.com, oleg@acm.org, http://pobox.com/~oleg/ftp/

Database query and programming language are twins separated at birth: although they grew up in different communities and have rather different (inter)faces, they both accomplish the same task: instructing a computing system to locate a datum given its description (name, address), apply a certain transformation to it, and output/store the result. This common semantics makes it possible to "translate" between a database mindset and that of the "regular" programming. This paper aims to elucidate the rules of such interpretation.

This translation may be insightful for its own sake. It shows for example that all the variety of OO species has sprung from different structures of underlying databases-environments, which hold the data and their labels. Conversely, a way queries to these databases are formulated and executed has everything to do with how powerful and expressive the corresponding OO system is.

There is also a practical side to this seemingly abstract topic, relevant to the day-to-day programming. The unified perspective helps to evaluate tradeoffs in existing OO systems; it gives an insight into extending a particular framework, to be more resilient to changes in base classes, to support dynamic inheritance, virtual construction, etc. This extension does not necessarily mean giving up on the current programming language and switching over to a fad of the day. With a clear understanding that the devil is in lookups, and an idea which particular lookup would best suit the task at hand, one can often phrase the cooler OOP features in any programming language. Some languages may have had these frills already built-in, but even lowly C can easily bear rather intricate environments. This paper demonstrates one such system, which supports late binding and dynamic inheritance, and is implemented in C/C++. The paper also dwells on how the database viewpoint can illuminate the fragile data base problem; several solutions are discussed, including a technique of extending a base class of a compiled hierarchy without breaking/recompiling it.

Thus following a particular OO system is a question of style, that is, how one writes the code, not what language he uses.

Simple stuff

To see how database queries and programming languages are intertwined, let us consider the most trivial example:

```
x := x + 1
```

This innocuous statement clearly shows how and when a database query creeps in. For the most part, these are the issues for a compiler writer to deal with. Still, a user of the language may also want to know what is going on, to assess efficiency, expandability, correctness, or to add new functionality. Thus, the issues the simple statement above poses are:

- reference resolution

'x' is obviously a *reference* (handle, name, label) of something. The first problem is then to find what `x` stands for, and which operations it permits. Depending on the language (latent vs. manifest types) this involves searching for a declaration of `x` in the current "scope", or locating the value of `x` in the current *environment*.

In Programming Language Theory this function – which, given a name, finds the corresponding value – is usually denoted `rho(x)` and called "environment". It is a very general formulation indeed: the meaning of `rho(x)` spans from fetching a data from memory given its address or tag to `(eval 'x)` in a language like Scheme/Lisp, to translating a name into an "address" (done by the compiler) and following the address at run time. What matters for the current discussion is that the reference resolution is essentially a database query; capabilities of this database and the richness of the current "context" is what determines the expressive power of the resulting (OO) system.

- updating/mutating data

Modification of the current context (database) can be accomplished in several different ways: by mutating a data item in place, by updating an item out-of-place (`become` in Smalltalk), or rebinding the name to a new value (and garbage-collecting the old one).

- finding executable code that implements an operation

This is a particular case of the reference resolution: just as one has to find out the meaning of `x` in the expression above, one needs to know the meaning of '+', the operation itself. Some systems (Lisp/Scheme) do not draw any distinction at all: operation resolution is really just a particular case of name resolution[1] . Other frameworks consider the operation as sending of a message '+' to an object 'x'; the job of determining the meaning for '+' is thus relegated to the object. Often the query for '+' is done in a different database than the one consulted for 'x'. One reason for that is the database of code (operations) is persistent (as the body of an executable file), while the database of data (the run-time environment) is often not.

Pointers and databases

A hype surrounding OO databases may make an impression that the database mindset and vocabulary somehow lack such a fundamental notion as pointers (references). These terms must therefore be deliberately introduced into databases, if databases and OOP are to get engaged. Pointers however are not an alien term for databases: in fact, pointers are very essential to hierarchical and network data models. For example, in a hierarchical database, all pieces of information are linked to form a (directed) tree or forest. To access (read/delete/modify) a particular item you have to walk the tree to it, given a *path*, a sequence of keys in a strict order. Just as you need to specify a path (sequence of directory names) to access a file, or a Web resource. This is hardly a surprise: file system and URL system are typical examples of a hierarchical database. Getting hold of an AOCE object/property by providing its path also falls into the same category. Indeed, AOCE, file systems, the URL system – all implement a *containment* model, which is a hallmark of the hierarchical data model. Incidentally, this means that AOCE can be considered a *plug-in file system.*

In a network database (e.g., ADABAS), logical organization of information is represented by an arbitrary graph, not necessarily a tree. Locating data items entails traversing the graph's edges: *pointers*. This must be very familiar to any C/C++ programmer: creating data structures with pointers to other data structures is a fundamental C/C++ programming skill. This is also what an Operating System does all the time: For example, a process control block has pointers to memory control blocks, file control blocks, session control blocks; a file control block points back to the process(es) that own a file, to a device control block, to a cache with the read data blocks, etc. Most of the OS functionality is chasing and chaining all these pointers. The network logical model is the fastest as far as speed/efficiency of accessing data is concerned; it is little wonder OS kernel's databases are all of the network kind.

In a relational model, getting hold of necessary pieces of information may also involve tracing "links"/"pointers", from a row of one table to a row of another. However, these "pointers" are of a peculiar kind. While network databases typically store associations using regular pointers to physical records (blocks on disk), a relational database breaks these links in two halves (the `from` half and the `to` half: a key and a lock) and stores the two parts separately. The pointer itself is "computed"

[1] "There is no distinction between 'methods' and 'instance variables'; a method is simply an instance variable with a procedural value", MzScheme Reference Manual, http://www.cs.rice.edu/CS/PLT/packages/doc/mzscheme/index.htm

(reconstructed) on the fly: given a key, a database manager finds a lock that corresponds to it, or the other way around. Of course it is possible to have a key opening several locks, and a lock opened by several keys; many-to-many association is really a snap. Since the association (an "array of pointers" so to speak) is computed, it does not have to be explicitly stored: relational databases are rather space efficient. On the other hand, access requires a computation – *table join*, – which is usually done by sorting two tables (by key's and lock's values) and matching the records. This process takes time, and a lot of scratch space. Note that some database systems (for example, UniSQL) can "cache" thus computed association: `join`-ed pointers become real memory pointers between the corresponding objects in a `Workspace`. Therefore, repeated link chases are performed with utmost efficiency.

Object as a private context (namespace)

To simplify and speed up lookups of variables, make code more comprehensible, and following the universal rule "divide-and-conquer", the search space can be partitioned into a set of (relatively small, hopefully) contexts. Just as caching data for CPU, this technique relies on a locality principle: within a short interval of time or a small section of code, most of the queries occur within a narrow context.

Objects and classes are the way to erect partitions in the lookup space. Although objects are most commonly defined as local (encapsulated) state plus a reference to shared code which operates on this state, "An equally important way to look at a class is as a means of establishing a namespace. A namespace is essentially a context for names associated with entities. Within each class, names of member functions are mapped to the code that implements those functions, and names of member fields are mapped to the values within an object of that class. Different classes can contain the same name, but the value of that name will be different because the classes are different namespaces."[2] Note that there is a definition of objects that directly relates to a notion of a named, isolated context: "We define an object as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand [Rumbaugh 91]" [OOFAQ].

Implementations of objects are as varied as their definitions. In the most straightforward sense, an object is merely a "private" environment. A data item in this environment (collection) – an object's member – is represented by its tag, a name, which is used to look up and access the item. Examples abound: an object in Perl is merely a (blessed) hash. In JavaScript, objects are just associative arrays of JavaScript variables, "properties". A single property can be accessed either via dot notation, or an associative indexing. For example, if one has defined an object `bestCPU` with a property `maker`, the property can be referenced as either

```
bestCPU.maker
```

or

```
bestCPU["maker"]
```

This example very clearly exposes an object as a separated and named lexical scope, and accessing its members as a query.

This straightforward implementation, coupled with a delegation-like inheritance (see below) is the easiest way of turning a `foo` into a `OO-foo`. Other examples include MOPS (OO Forth), Delphi's and VB's property sheets, Python, Tcl/Tk, Bob, to name just very few. One also has to mention in this context NewtonScript frames, Scheme closures, Apple Event objects, BMessages and other inter-process communication "packets". Indeed, a message from one (Objective-C, BeOS, MacOS, etc) entity to another is a small database, a collection of tagged and typed data items; in case of Objective C, referencing items in this collection even looks similar to accessing a regular variable. Incidentally, a file system's directory is an object too.

Beyond a class: encapsulation

The previous section has shown how closely objects and database entities (dictionary, table, etc.) are related. It is interesting to trace this spiritual connection one generation further. Just as objects were introduced to cope with abundance of data by grouping together related items, there have emerged ways to manage proliferation of objects, by bundling them into classes and hierarchies. An embedding (*has-a*) hierarchy and deriving (subclassing, *is-a*) hierarchy are the two most conceptually important and most frequently occurring types of hierarchy.

[2] Jim Waldo, "Protected Classes", Java Advisor column, UNIX Review, Oct 1996, p. 97-101

The embedding hierarchy is inherent to hierarchical databases, which are, by one definition, collections of containers containing other containers. In a network database, embedding is represented by linking a parent record with all the records it "owns". Placing an object's body or reference inside another object are also the two ways of performing embedding in programming languages. As far as relational databases are concerned, tuples may not contain nor *point* to other tuples; still, embedding can be represented just as well. Containment of relations is expressed not by placement or reference, but by computation of a table join. "The specification of a class D as the domain of an attribute of another class C in an OODB is in essence a static representation of a join between the classes C and D" [UniSQL Whitepaper]. Granted, joins are expensive from the computational point of view; yet they are conducive to an efficient and flexible data organization. Furthermore, a late binding – *late pointing* – inherent to join queries makes them immune to a fragile base class problem (see below).

Inheritance as a fall-back

Another way of constructing a hierarchy of objects is by marking common and particular traits (members, methods) between a pair of objects. One object can then be chosen as a base, reference point, with the other – *derived* – object containing only new and overriding members, as compared to its *parent*. This inheritance hierarchy is a difference-type encoding of a collection of objects, similar to a delta-coding of sound samples, or LZ encoding of text strings. When a set of objects' data items is partitioned into a base and difference subsets, searching an object for an item has to be generalized. That is, a failure to locate a slot with a given name/tag in an object no longer implies that the whole query must have failed. The object's parent (and its parent, etc) must be searched in turn. Thus inheritance introduces a fallback, or an *ordered* sequence of lookups.

A particular kind of inheritance is grouping objects into classes, that is, "objects" that specify methods, data, and slots common to any object of that class. Individual class instances contain then only particular values for these slots: data or *virtually overridden* (in C++ parlance) methods. Note that in some OO systems, for example, Smalltalk, class "objects" are objects themselves: they are objects that construct other objects.

To represent lineage, a derived object must contain a reference to its parent (class), or even physically incorporate it. What distinguishes this from a regular containment hierarchy is the fact that this "embedding" of a parent affects the derived object's slot lookup procedure: the contained object/class acts as a default "clause". In some OO systems this embedding of a parent shows only in compiler tables describing an instance/class, with no corresponding run-time representation. Some other systems make a reference to a parent a part of an object itself, as a `parent/_proto` slot, or a "virtual" slot – an `"item not found"` exception procedure. Self, NewtonScript, Lua are a few examples of such – so-called 1-level, single-hierarchy, or delegation – systems. Note that in these OO systems, the distinction between an object and a class is blurred: all classes are objects, any object may act as a class, that is, provide object instantiation or act as a shared parent. These systems also make it possible for an object to change its parent at run time (which is called a *dynamic inheritance*).

Applications and examples

Smart defaults

This is the most trivial example of a fallback closure, the example that ought not even be mentioned if it were not for an amazing expressive power of this trick. It is used when a code runs into a problem, but not necessarily fatal condition: for example, a sought piece of information could not be located, or one of the parameters was left unspecified. In this situation, the code may assume some default value, crash, give a user the second chance to enter the parameter, or merely alert the user about the default value being used. However primitive this all seems, it does take a fair amount of code to handle all peculiarities, especially to enable a function to communicate its failure to its caller, so the latter can do something intelligent about it. This common part can easily be automated if, for example, a lookup function is written in a smart way: that is, it takes a default argument, in general, a default functor. Here are a few examples of the expressiveness of this approach (taken from `http://pobox.com/~oleg/ftp/c++advio.README.txt`)

```
                    // Default default: crashes the program if the file
                    // does not exist
const int file_size = get_file_size(file_name);
...
                    // A smarter, custom default
struct GFS_big_default : public GFS_Default
{
  enum { def_size = (size_t)(-2) };
  size_t operator () (const char * file_name)
    { cerr << file_name << " does not exist; default size "
            << def_size << " used" << endl;
       return def_size;
    }
  };
  if( get_file_size(file_name,GFS_big_default()) ==
       GFS_big_default::def_size )
  handle_file_does_not_exist();
...
struct Ask_user_default : public GFS_Default
{
  dialog_stream& ask_user;
  Ask_user_default(const dialog_stream& ask_user_dlg) :
            ask_user(ask_user_dlg) {}
  size_t operator () (const char * file_name)
    { ask_user << file_name << " does not exist; "
            << "do you want me to create it? " << endl;
      if( ask_user.cancel() )
        throw die_die();
       else
       {
         assert( close(creat(file_name,0777)) == 0 );
         return 0;
       }
    }
  };
const int file_size = get_file_size(file_name,Ask_user_default(Alert()));
```

These examples clearly show the power of the default functor, which easily accommodates various strategies of handling lookup failures (in our case, looking up an OS property 'size' for a given file). Of course one could also use exceptions. However, not all C++ compilers support exceptions well (the HP-PA platform is notorious in this respect); some compilers impose a heavy overhead or require you to recompile all code with exceptions enabled. The last example in the snippet above – when the default value is "computed", which involves a dialog with a user and a side-effect – is rather messy to implement with exceptions: once the stack is unwound, it can not be wound back up. The smart default approach provides both safety, flexibility, and yes, efficiency.

Twists and turns in falling back

The idea of a smart but single default can be extended to multi-level fallbacks, supported by a sequence of dictionaries searched in a (dynamically changeable) order. That is, an item is first sought in a top dictionary. If the lookup succeeds, the found value is returned. Otherwise, the next dictionary – whose name/reference is contained in a "dictionary path" – is consulted. If nothing was found, the next dictionary is searched in turn, etc. Only when all dictionaries mentioned in the path have been looked through (in vain), a default action (if any) is executed.

At first glance, this system looks and feels like a lexical or dynamic scoping: indeed, when code refers to a variable 'i', it causes the compiler/interpreter to consult the innermost scope (environment), falling back to the containing scope etc. all the way to the global environment of the task. However, the dynamic nature of the "search path" sets our hierarchical dictionaries apart. Dictionaries can be added or removed from the path at run time, which will affect the order fallbacks are executed. The search path can change even in the process of a lookup itself. For example, a dictionary, once entered, may sever the search path, thus cutting access to its "parents" and making its inheritance from them private. It has to be stressed again that because of the dynamic nature of the search path, no rigid parent-child relationships exist: hierarchies are built and torn down on the fly as circumstances warrant. The closest model of these dynamic hierarchies is a set of dictionaries in Forth/PostScript. As we are about to show, one does not need to switch to these languages to benefit from their dictionary framework: one can easily implement a similar thing in C++ (or even C), or any other language of choice.

As an example, I will demonstrate a client/server system to disseminate weather information (temperature and pressure grids, wind directions, etc.) to a number of clients. An earlier version of the system, called NODDS, has been in operation at a Fleet Numerical Oceanography Center (FNOC) at Monterey, CA for a number of years. I was to rewrite its server part, taking advantage of a different database of weather products. Thus I had to abide by the existing client-server protocol.

Requesting a weather product, for example, a temperature grid, requires quite a few parameters to be specified: grid's boundary, grid resolution, the name of a weather model used to process raw observations or make forecasts, the product code, desired units, scale factors, etc. The database required all these keys to retrieve the data; a user, however, does not need to specify every one of them: some values are to be inferred. The inference rules vary however depending on a client (for example, a user in Europe would like to see temperatures in degrees C, while degrees F may be preferred in other parts of the world). The defaults may also vary with a model (certain weather models support only certain grid resolutions), a product, etc. Thus we have a hierarchy of configuration information – units, model name, region, scale factors, etc. – where each piece can be overridden by client's profile, client's request, or by the results of evaluating the request.

The dictionary system with a dynamic search path proved to be very useful in implementing the weather products server. Here are a few details. When the server starts up, it loads the top-level vocabulary and makes it default: pushes it on the top of the search path. This top vocabulary is rather trivial, for example:

```
Voc(R): TopRefVoc= "Top Reference
Vocabulary" (
R,2D_grid= "2d_grid.voc"
)
```

(this is the vocabulary contents *exactly* as written in the file it is loaded from). The vocabulary is made of a single item, with a key "`2D_grid`" and the value which is another vocabulary, referenced by an external file name. This latter vocabulary is not loaded until required. When a request for a temperature, pressure or any other 2D grid is received, the `2D_grid` vocabulary is looked up and pushed to the top:

```
VocPath::push("2D_grid"); // Load
Vocabulary of products
```

This statement first looks up an item with a key "`2D_grid`" in the vocabulary path, which at the moment contains only the top vocabulary. The reference is resolved, and thus loaded `2D_grid` vocabulary is pushed to the top of the search path. Here is how this vocabulary looks like:

```
Voc(R): 2D_grid= "2D grid products" (
S,geom_name= "global_73x144"
S,model_name= "nogaps"
S,units_name= "*"
Voc(R): W= "flaps for NMC products" (
S,model_name= "NMC"
S,return_flaps_char= "W"
)
Voc(R): A01= "Surface Pressure" (
S,units_name= "mb"
)
Voc(R): A07= "Surface Air
Temperature" (
S,model_name= "otis"
S,units_name= "C"
)
```

```
Voc(R): C10= "1000Mb Temperature" (
S,units_name= "C"
)
Voc(R): E34= "700Mb Relative
Humidity" (
S,model_name= "uanva"
S,units_name= "fraction"
)
)
```

This vocabulary contains a few simple items (like "geom_name", "model_name", "units_name") and a number of embedded dictionaries for specific weather products. These slots tell the server that it should assume a default value of grid_73x144 for the grid geometry, default value of "nogaps" for the model name, etc. A client's profile, loaded by the server after parsing a request, may include specializations for that particular client: an area the client is interested in, defaults for units (say, always "ft") and for the model (say, some particular client is interested only in NORAPS data). The client profile vocabulary is placed in the search path ahead of the product group dictionary, giving defaults of a client precedence over the server defaults. The request for products (made of one or several request lines) provides further details. Each product has its own set of defaults (e.g., preferred units), which override the client or server defaults but can in turn be overridden by request line options.

For example, if a user has requested surface air temperatures, the corresponding subvocabulary (with a key "A07") is looked up and pushed at the top:

```
VocPath::push(cat_no);
```

Therefore, model_name and units_name slots from this vocabulary override the (generic) ones in the "parent" 2D_grid vocabulary. If the request contained a "flaps" character W, the corresponding vocabulary "W" is loaded on the top, which changes the default for the model_name to "NMC". When the time comes to query the database, the necessary keys are looked up in the resulting environment:

```
xstrncpy(szModelName,VocPath::find_st
r("model_name"),sizeof(szModelName)-
1);

xstrncpy(szGeomName,VocPath::find_str
("geom_name"),sizeof(szGeomName)-1);

xstrncpy(szParamUnits,VocPath::find_s
tr("units_name"),sizeof(szParamUnits)
-1);
```

In the present example, the "model_name" would be "NMC" (as found in the topmost "W" vocabulary, which is searched first), "units_name" would be "C" (the top vocabulary does not have this item, but the next vocabulary in the path, "A07", does), and "geom_name" would be "global_73x144" (found in the second vocabulary from the top). After the request is processed, all the relevant vocabularies are removed from the vocabulary search path:

```
voc_path_mark.back_off();
```

The server is now ready for a new request. It has to be stressed that the server configuration is entirely controlled by the vocabularies, which are loaded on demand from self-explanatory text files. This makes it easy to change the name of a weather model, units names, etc. by editing the files in any text editor.

The vocabulary system described above is a part of the c++advio library available from http://pobox.com/~oleg/ftp/c++advio.READ ME.txt . The vocabularies are clearly polymorphic: a vocabulary slot may contain a simple data type (int, double, or string, that is, const char *), a reference to another vocabulary, or a (sub)vocabulary. Homogeneous vocabularies are those whose slots have exactly the same structure, which is reinforced when new entries are added.

A vocabulary of vocabularies contains all full-fledged vocabularies: that is, every dictionary is always contained in some other (parent) vocabulary, which in many cases is the Vocabulary of vocabularies. Note, a vocabulary can be referenced from many different dictionaries. A vocabulary path is a list of dictionaries to search for a named slot in, in order of appearance in the path. Vocabularies can be referred to from within other dictionaries by special items, vocabulary references. A reference either points to a vocabulary (when the reference is completely "resolved") or merely holds the name of the referred vocabulary, which is to be looked up when needed (a *lazy* reference). The reference may also contain a file name, from which the vocabulary would be loaded when it fails to be located otherwise.

Vocabularies support all the basic functions: lookup, insert, delete, traverse. Looking up a slot (or a slot value) is the main function, and exists in many varieties. There is a function to look up a slot in the current dictionary, in all vocabularies in the path (the top dictionary in the path is searched first, the vocabulary underneath is search only if the lookup in the top dictionary failed, etc).

Note, the vocabulary objects and their slots are somewhat unusual C++ objects. They do not offer any public constructor or destructor. A vocabulary can only be created by a special friend function, which automatically inserts the newly made dictionary into the vocabulary of vocabularies, and puts it on the top of the vocabulary path (which makes the new vocabulary current). By the same token, the only way to make a new vocabulary item object is to call a special friend function. This function allocates object's memory off the heap, fills it in, and inserts the new item into the current vocabulary. Therefore, an item or a vocabulary cannot exist by themselves, not being a part of some dictionary. A user can get hold of only references to a vocabulary or a slot, but never of the objects themselves.

I would also like to briefly mention a more primitive system (implemented earlier in plain C), which provides a uniform way to specify a variety of parameters and values for a complex project. The most common way of passing parameters to a module – via command-line options – is probably the worst solution, as anyone who ever took a look at `gcc` command-line arguments could testify to. A much better approach is to use a global database of project options (something like a pool dictionary in Smalltalk), available to all project's functions/modules via a simple lookup interface. The database is loaded at start-up from a simple configuration file, whose name is the only parameter passed to the root module. The following are two examples of the configuration files, from two real projects:

```
Header = H2O
Write_to =real_water1.dat;No
Molar_mass = 18.02
;
P_min = 1                ; Lower bound
Pressures =
1,100,500,700,1000,1500,2000,3000,700
0,10000 ; pressure range
;
T_min = 303.15 ; Min temperature
value
T_max = 353.15 ; Max temperature
value
;
Delta_p = 10;0.5 ; Grid mesh
Delta_t = 0.01
;
Mixing_factor = 0.96  ; Mixing factor
```

And

```
; Configuration file for CLEAN
;
Input_spectrum = /tmp/aaa
;
Cleaned_spectrum =
/tmp/cleaned_re.dat
Remainder_spectrum = /tmp/rem_re.dat
;
Dead_time   = 4  ; Dead time in ms
Freq_range  = 1  ; Freq. range for
the spectrum, kHz
Noise_level = 6  ; in abs units (in
units of the spectrum)
;
Window_half = 3  ; Window half-width
for the deconvolution algorithm
Gamma = 0.5      ; Scale factor for
the deconvolution algorithm
Large_step = 10  ; Size for the large
iteration step
Max_no_steps = 1000 ; Maximal no. of
steps to perform
Print_level = C     ; D-detailed, C-
concise, N-nothing
```

The configuration files have a very straightforward structure: merely a list of parameter names and their values. Everything after a semicolon is a comment. Note, that parameters may have an *array* of values: as in

```
Pressures =
1,100,500,700,1000,1500,2000,3000,
7000,10000
```

For one thing, the configuration files are more informative than command-line options. The root module is also spared the trouble of validating every option passed to it: this is the job of the immediate consumer of a particular parameter. Adding, deleting, or changing the meaning of parameters will not therefore require the root module be recompiled. Another important advantage of this global configuration database is that the program, having finished calculations, can dump its configuration at the beginning of its output files. When the results are to be processed further by some other code, this code can easily find out the value of any parameter of the originating calculation. Putting the configuration along with the results also makes the output files self-documented.

Fragile Base-Class problem

In a nutshell, the "fragile base-class (FBC) problem" is what requires client programs to be recompiled whenever the class library on which they depend is modified. The FBC problem is a lookup problem; more precisely, a lookup cache-coherence problem. This problem arises whenever a reference resolution, identifier lookup, or any other query is performed non-atomically, while the database is being concurrently modified. Especially problem-prone are the multi-stage lookups that "cache" the results of their intermediate steps, providing no mechanisms to flush or update this cache should the underlying environment gets altered. Given below are a number of classical as well as lesser known examples of the FBC problems. These examples hopefully show that the problem is more pervasive than it appears to be.

Let us consider the following C code fragment that uses the standard `stdio` facilities:

```
FILE * fp = fopen("/dev/tty","r");
char buffer[120];
if( fgets(buffer,
          sizeof(buffer)-1,fp) == 0 )
  if( ferror(fp) )
    handle_error();
```

In most common `stdio` implementations, `ferror()` is a macro or an inline function, for example,

```
#define ferror(file) ((file)
          ->state.error)
```

as in Metrowerks' CW 11 MSL. Thus the second `if()` statement in the code above would be compiled as

```
check the byte located at fp+13
```

(or, in PowerPC assembly)

```
lbz      r0,13(r31)
cmpwi    r0,0
beq      *+12
bl       .handle_error__Fv
```

Now imagine that the vendor of the standard C library has decided to enhance his product. For example, he now wants to support very large files, bigger than 4Gb; or he resolved to pack the `error` flag in one bit, rather than devoting a whole byte to it. Thus the vendor has changed the `__file_state` structure from

```
typedef struct {
   unsigned int  io_state  : 3;
   unsigned int  free_buffer     : 1;
   unsigned char eof;
   unsigned char error;
} __file_state;
```

to

```
typedef struct {
   unsigned int io_state   : 3;
   unsigned int free_buffer      : 1;
   unsigned int reserved   : 2;
   unsigned int eof        : 1;
   unsigned int error            : 1;
} __file_state;
```

Linking this new library with the *compiled* code fragment above creates a problem: the `error` field is now a bit rather than a byte, and is located at `fp` +11. The old version of the `ferror()` function inlined in the compiled code would now check a wrong field, which has nothing to do with the file status. In contrast, `fgets()` continues to work just fine. The compiled fragment contains a reference to `fgets()` rather than the function itself, and the code passes it a *pointer* to the `FILE` structure. Both references will be resolved late, using the new version of the library: the function's body would be located during linking, and the `FILE` structure will be "looked up" only at the run time, by `fopen()`. Function `ferror()` on the other hand has been located early, at the compile time. This eager binding saves time when linking and running the code, but causes a problem when the underlying data structure changes.

The Fragile Base Class problem is most evident in C++. To illustrate it, let us consider the following `transaction_ofstream` class, which adds *commit*/*backoff* functionality to the regular C++

file stream:

```
class transaction_ofstream: public
strstream
{
  filebuf * output_buffer;
public:
  transaction_ofstream(const char *
file_name)
   : output_buffer(new
      filebuf(file_name,ios::out)) {}
  void backoff(void)
   {seekoff(0,ios::beg);}
  void commit(void)
   { output_buffer->xsputn(str(),
                          pcount());
     output_buffer->sync();
     backoff(); freeze(0); }
};
```

This `transaction_ofstream` redirects all incoming information into a memory buffer; it is when the user `commits()` that all the accumulated data are written into a destination file. Suppose now that the vendor of the C++ stream library has decided to make the library thread-safe, and added a mutex lock to the `strstream` class. This obviously compels recompilation of the entire code that implements and uses `transaction_ofstream`. Indeed, the `commit()` method above gets hold of the `output_buffer` data item as the contents of a word of memory located at `&(this->output_buffer)`, that is, at `this+sizeof(strstream)`: derived class' data members are appended to base class' ones in any instance of the class. Adding new data members (the mutex, in our example) to the base `strstream` class changes the `sizeof(strstream)`, hence altering the offset to the `output_buffer` in the `transaction_ofstream` object's layout. Thus every method that depends on the derived class' layout must be recompiled to take into account the modified offsets. Note, the `commit()` method above does not use the mutex lock, the field added to the base class; nevertheless this method must be recompiled, because the new field changed offsets to those data members that the `commit()` method uses.

This problem is not limited to C++, it lurks everywhere a (name) lookup is performed "early". For example, consider an application that dynamically links to a (shared) standard C library, `libc.so`. Suppose that the application at some point calls a library function `printf()`. Since the compiler cannot know where this function's code will be located in memory when the application runs, the compiler generates a dummy instruction, for example,

```
call 0
```

with 0 in place of the actual call address. In the object module's external dictionary, the compiler leaves a note for the linker to replace this placeholder 0 with the correct address of the `printf()` function. When building the executable file, the linker may be tempted to perform the following optimization, especially if prodded by the user, for example, by an option '`ld -b`' on Solaris 2.5. The linker can scan the dictionary of the shared library `libc.so` and find out that `printf()`'s code starts at an offset, say, `0x7000` from the beginning of the library. The linker then instructs a run-time loader to map the shared library file into the process virtual space starting at, say, `0x40000000`; the dummy `call` instruction above will be corrected by the linker as

```
call 0x40007000
```

Thus, the reference to `printf()` becomes resolved at link time, requiring no scanning through the shared library or object code adjustments when the application runs. This optimization carries with it a price tag: suppose an operating system vendor has modified the shared library, for example, to add a few new functions. Suppose the code of `printf()` itself remained intact; however, its offset in the new version of `libc.so` changed, from `0x7000` to `0x7010`. Obviously, our application must be relinked: otherwise, the instruction `call 0x40007000` would miss its target, the beginning of `printf()`. It has to be stressed again that even if the application code does not use the newly added libc`.so` functionality, even if `printf()` itself is unchanged, the application still has to be rebuilt: no matter how incidental changes are, they have rendered the early resolved address of `printf()` invalid.

Other examples were the over-eagerness can hurt include using absolute IP addresses rather than host names in internet applications. This certainly saves the trouble of performing a DNS lookup; on the other hand, if a host is moved to a different address (different network), the application would have to be rebuilt. By the same token, using absolute port numbers (rather than resolving service names with `getservbyname()`) saves CPU cycles but leaves the application vulnerable if the port mapping changes. Even referring to a document by its URL is a shortcut, easily broken when the document moves. More reliable is a lazy, indirect locator that either refers to the document through a glossary, or contains a search engine query to be executed on demand, yielding the document's current location at that moment.

"Precompiled" database queries is yet another example of jumping the gun. Normally, SQL queries are compiled right before they are executed. For example, when a database manager receives a query

```
SELECT Job, Avg(Salary) FROM Employee
WHERE Salary > 50000 AND DOB > 3/5/53
GROUP BY Job;
```

it first checks a database schema for a table `Employee`: suppose, the table does contain the attribute `Job` as column #2 of the table, `Salary` as column #3, and `DOB` as column #5. The database system then scans through the `Employee` table in storage, checking these three columns and creating the answer. The query compilation stage – looking up attribute names in the database schema and translating them into table column numbers, among other things – can be performed in advance. The compiled query can then be executed several times. However, if the `Employee` table is changed, for example, some attributes have been added so that `DOB` becomes column #7, the precompiled query would give the wrong result. The original query (in the source form) will not be affected by these changes: the database manager will find the new column number for the `DOB` attribute when it consults the schema. In fact, the original query can be executed as long as the table `Employee` still contains columns `Job`, `Salary` and `DOB`, no matter how the table layout has been modified. This late binding is one of the main advantages of the non-procedural nature of SQL, which allows for independence of data manipulation from data layout.

Fortifying derivation trees

The obvious (and probably sole) solution to the Fragile Base-Class problem is to eschew shortcuts and be lazy, putting off resolving names as late as possible. This may be relatively straightforward to follow in some cases: use demand/lazy linking as often as possible (not letting the linker take shortcuts in resolving external names in shared libraries); precompile only those SQL queries that are absolutely critical to performance; try to use symbolic internet service names (rather than hard-wired port numbers) and query a portmapper: the latter is default in Sun's RPC system. The use of symbolic host names (rather then dot IP addresses) has become the standard practice already. In case of URL, being lazy may mean, at the very least, using relative URLs, especially to references within the site. Better yet, one really ought to take advantage of link glossaries if a web site is maintained by Frontier, NetFusion, GlobeTrotter, etc. packages. The best solution would be using of Universal Resource Names, which *describe* resources without explicitly naming them; the URN proposal is presently being considered by IETF.

Incidentally, lazy does not necessarily mean inefficient: for example, a dynamic linking technique called "load-time code generation" eliminates not only the FBC problem, but also the run-time overhead typical of dynamic linking. This technique uses a so-called slim-binary object code representation, which is usually 2.5-3 times more compact, and also allows for the last-minute tweaking (e.g., instruction scheduling)[3]. These slim binaries are used extensively in an *Oberon/F* system.

The advice of binding late is much harder to follow in C/C++ etc. environments, which are designed to use static binding of variable/function/method names, manifest types, and to resolve as many symbols at compile time as possible. Although the late binding is not explicitly provided by the language, it always can be emulated: for example, by `getenv()`, querying a resource, an `.INI` file, etc. The system of nested dictionaries discussed in a previous section is yet another example of demand binding implemented and used in C/C++.

However "static" the C/C++ environment may look like, it does offer some tools for dynamic binding. Indeed, the late binding occurs whenever a data item is accessed *indirectly*, either via a call to a function, or a handle. In the former case, one explicitly calls some sort of a lookup/getter/setter function that finds or computes the value. The result of this lookup may be cached. Incidentally, this is precisely how most of dynamic linking schemes work: a call to a function in a shared library is represented by a *promise* to invoke this function. When the promise is called, the operating system would look up the address of the

[3] M.Franz, "Dynamic Linking of Software Components", Computer, March 1997, pp.74-81

function and transfer the control to its code. The OS will also cache the found address: hence the next call to the function will invoke the function's body directly. The same technique can be used with data members. Thus if you do not want to bother making a promise yourself, simply put the necessary data/functions in a dynamically loadable segment, and let the dynamic linker/loader do the job for you.

To an extent, the Fragile Base-Class problem in C++ can be alleviated (if not completely solved) by indirection, along with some discipline. Indeed, the main reason a modification of a base class compels recompilation of all derived classes is because a derived class object contains the base class object as its header, with derived object's data following. Thus any change in the base object size alters the offsets of the derived object's data within the object, the offsets that derived class' methods use to access the data. However, if the base object's body is placed somewhere else in memory, with the derived object containing merely a reference (pointer) to it, then no matter how the base object shrinks or expands, the derived object's layout will remain the same. Using the example of `transaction_ofstream` above, if a C++ stream library vendor has shipped a new version of the library with an enhanced class `filebuf` (e.g., supporting 64-bit file offsets), the `transaction_ofstream` class (and any code that uses it) does *not* have to be recompiled. One has to take a special care though to manipulate the `filebuf` object only through appropriate methods, rather than using known offsets to publicly accessible data. Otherwise, one runs into the same problem as the one described in the previous section, dealing with `ferror()` and a changed `FILE` structure. This problem will not occur however if new members are added at the end of an indirectly referenced "base" object: obviously, this procedure does not alter any offset to the existing data. As was mentioned above, the discipline is important.

"Deriving" an object from another one by merely pointing to it is a rather weak form of inheritance: indeed, none of the base object's methods will apply to thus "derived" object. One has to re-declare all needed base class methods in the "derived" class to forward appropriate messages to the base object. This forwarding (which Microsoft calls "aggregation") is the cornerstone of Microsoft's COM architecture. The discipline is needed here as well: to make the aggregation as general as possible, a "derived" class should declare the base class reference as the pointer to an abstract class. The derived class constructor should initialize the pointer by calling a *factory*, which returns the pointer to an object implementing the abstract class functionality. Therefore, no matter how this implementation changes, the derived class itself does not have to be recompiled.

One does not have to give up inheritance to use indirection: with virtual classes, one can get both. When an object is virtually inherited, it does not become a part of the derived object, only a pointer to it does. The compiler takes care of initializing the base object, calling destructors, forwarding all the relevant messages, and accessing object's public data via that pointer. This works exactly as the forwarding above, yet looks and feels just like the regular inheritance. Harking back to the `transaction_ofstream` example above, if the C++ library vendor inserts the mutex lock into an `ios` class (which is virtually inherited by all other C++ stream classes), none of the code that implements or uses the `transaction_ofstream` needs to be recompiled. Thus indirection cures the FBC problem caused by addition of new data members to a base class. The situation is more complex when a base class is endowed with new virtual functions: the layout of a `vtable` is very difficult to control. For example, if one adds a new data item at the end of a class declaration, he can be sure this will not affect the layout of any of the old class members. This is not the case with virtual methods: often `vtable` entries are arranged in the lexicographic order of the virtual function names. Therefore, addition of a new virtual function to a class may affect `vtable` offsets for many or all old virtual methods, in a way that is difficult predict or control.

It is possible however to fortify at least a significant chunk of the derivation tree, even against additions of virtual functions to a base class. This technique requires some discipline in building a hierarchy, but in return guarantees that only the top-most class needs to be recompiled to take into account modifications made to a base class. The rest of the hierarchy (which may have been compiled to a set of separate libraries) may be used as it is. Let us consider a `transaction_fofstream` class, which is almost the same as `transaction_ofstream` of the previous section:

```
class transaction_fofstream: virtual
public strstream
```

```
{
... the same as transaction_ofstream
above
};
```

only it inherits virtually from the `strstream` class. We also assume that `str()` etc. methods of the `strstream` class have been declared virtual: all this virtuality is very important. Let us suppose that the `strstream`, `filebuf` and the other iostream classes are compiled in a library `iostreams.LIB` (which usually comes with a C++ compiler itself). Suppose we compile the `transaction_fofstream` class in a separate library `transactions.LIB`. A user can then use this class by incorporating it in his own hierarchy, as in

```
class MyTransaction: public
transaction_fofstream
{
 ....
};
```

Now assume that a vendor of the C++ stream library has decided to enhance the library with multi-threading. To this end, he added a mutex to the `strstream` class to enable a thread to obtain an exclusive access to a `strstream`. To make the design more flexible, the vendor also added a virtual function to operate this mutex lock, so that a derived class would be able to take an advantage of knowing that control has entered or about to enter a critical section. Rather than modifying the existing `strstream` class, the vendor ought to create a new, extended version of it. This is the discipline one has to stick to:

```
class strstream_v2 : virtual public
strstream
{
  pthread_mutex_t mutex;
public:
    strstream_v2(void) {
pthread_mutex_init(&mutex,
pthread_mutexattr_default); }
    virtual ~strstream_v2(void) {
pthread_mutex_destroy(&mutex); }
   virtual void lock(const bool
onoff)
   { assert( (onoff ?
pthread_mutex_lock :
pthread_mutex_unlock)(&mutex) == 0);
}
   char *str(void) { lock(true);
return strstream::str(); }
void freeze(const bool n = true)
{ if(n) lock(true),
strstream::freeze(true);
  else strstream::freeze(false),
lock(false); }
    bool frozen(void) const;
};
```

The vendor has compiled this `strstream_v2` patch, along with the original `strstream` in a new version of `iostreams.LIB`, which he now distributes. Since the original `strstream` version is present in the library as it has been, the old code continues to compile, link, and run. If a user however wants to take advantage of the new functionality, he can update his class as

```
class MyTransaction: virtual public
strstream_v2, public
transaction_fofstream
{
 ....
};
```

Obviously `MyTransaction` can now use `lock()`, the new feature brought in by `strstream_v2`. Slightly less evident is that a message `freeze()` sent to a `MyTransaction` would be delivered to the `strstream_v2` object rather than to the original base `strstream`. But what is truly remarkable is that when `transaction_fofstream` applies the `str()` and `freeze()` methods to the `strstream`, in the process of `commit`-ing a `MyTransaction`, the message is actually intercepted and handled by `strstream_v2`. Therefore, when `MyTransaction` is being committed, the `strstream` is locked until the data are written into the file. The `strstream_v2` effectively ***supplants*** the original `strstream` beneath `transaction_fofstream`, pulling off ***a substitution of a base class in an existent hierarchy***, without breaking it, or requiring it to be recompiled. The user merely needs to change a top level class to take advantage of new features in the enhanced version of `iostreams.LIB`. The old `transactions.LIB` library can be used as it is: it does *not* have to be recompiled. Thus the presented approach allows one to bypass recompilation for a rather long limb of the derivation tree, when a base class is extended with new data members, overridden virtual methods, or *even* with new virtual functions. If the vendor decides to enhance the `strstream` class further, he ought to create a new class `strstream_v3` virtually derived from `strstream_v2`. Granted, after a few more steps, this patching will become too messy to be useful; furthermore, chasing pointers in long chains of virtual tables slows the code down. Sooner or later, the vendor has to collect all the patches and roll them in a new version of `strstream`, which would require recompilation of all derived classes. Still, it is better to have to recompile the entire code one time in a few years rather than a few times a year, and still be able to take advantage of incremental enhancements.

Additional References

Object-Orientation FAQ
`http://iamwww.unibe.ch:80/~scg/OOinfo/FAQ/oo-faq-S-1.1.html`
A treasure trove of definitions for object, class, inheritance, etc.

Encapsulating a C++ Library
by Mark Linton, 1992 USENIX C++ Conference Proceeding, (pp. 57-66; ISBN 1-880446-45-6)

Object-Relational
The Unification of object and relational database technology
A UniSQL Whitepaper by Dr. Won Kim
`http://www.unisql.com/tech_spot/tech_spot.html`

SOM, COM, and Fragile Base Classes
`http://www.byte.com/art/9405/sec6/art1.htm`
The article explains what a Fragile Base Class Problem is all about, mentioning a Microsoft's way of dealing with it (by doing away with inheritance as being "too powerful" and using aggregation instead). The article then concentrates on how an Oberon/F system handles the FBC problem, by emulating delegation via callbacks. Incidentally, a distinction between types and classes (and correspondingly, subtyping vs. subclassing) is well explained.

`http://iamwww.unibe.ch/~scg/OOinfo/FAQ/oo-faq-S-1.10.html#S-1.10`
On dynamic inheritance, and one-level systems (Self, NewtonScript) where an object can serve as a class, and objects can change their parents on the fly.

Towards Tractable Algebras for Bags, by Stephane Grumbach and Tova Milo, Journal of Computer and System Sciences, 1996, v.52, pp.570-588.
This rather technical and dry mathematical paper proves (among many others) a theorem that an algebra of nested bags along with a fixed-point operator is Turing complete, that is, can simulate every computable query. Thus a database *IS-A* (OO)P language.