REFLECTIONS ON A DECADE OF OBJECT-ORIENTED PROGRAMMING IN ACCELERATOR PHYSICS

by Leo Michelotti

First viewgraph

Every talk should begin with a snappy one-liner. Here are a couple candidates.

- Those who don't see the problem can't appreciate the solution.

- To find something, it helps to look for it.

Mixed phases with ill-defined transitions

- **Phase 1: 1988-1992. The early years.** Terrible compilers, no debuggers; trying a whole new approach to programming in hostile technical and sociological environments; rapidly evolving "vision"; continually rewriting software; spectacularly unsuccessful proselytizing. Redeeming feature: C++ is a simple, powerful language.
- **Phase 2.1: 1992-1993: Communication: J.-F.Ostiguy, J.Holt, E.Forest, and J.Bengtsson** First contacts with like-minded people.
- **Phase 2.2: February 16-19, 1994. The Workshop.** Approximately 30 people show up to share viewpoints on OOP/C++ programming for studying accelerators and detectors.
- **Phase 2.3: 1995-1997. Collaboration: CLASSIC.** A collaboration is started, loosely under John Irwin's leadership. I attend one collaboration meeting and kibitz others. Christoph Iselin does a lot of hard work and writes MAD 9.
- **Phase 3: 1996-1999. Hunker down: beambeam and OLM/OAM.** Focus on doing local calculations and supporting local applications. Hide out on the sixth floor and add beam-beam classes (and others) to the libraries. Turn attention to supporting the OLM/OAM effort: connecting accelerator models with control consoles. (At a critical juncture, 2/4 of the OLM/OAM team leave Fermilab. It's still a good idea, but needs support, i.e., people.)
- **Phase 4: 1996-1998. FPCLTF (pronounced ZOOM).** CDF and D0 decide to rewrite software in C++ . Informally attending "brain-storming" sessions at D0, I eventually join the FPCLTF, a laboratory supported, organized, C++ support group. Regrettably, CD chooses to support only detector-related HEP activities.
- **Phase 5: 1999-present. To oblivion and beyond.** Ending the OLM/OAM team; beginning of FNAL's NLC team. Using the libraries to model and study linacs. Excellent compilers and debuggers, but C++ is no longer simple. Movement toward JAVA within controls and our group.

Top ten reasons for using FORTRAN, not C++

1. I wouldn't use C++ if you held a gun to my head force-fed me broccoli, and made me listen to Sheena Easton records. Go away and stop bothering me!
2. We have a super-stupendous project coming up, and C++ would be perfect for it. But *not now.* See me in six months. (recursively applied)
3. FORTRAN is OK: it said so in the Computing Division Newsletter. Besides, I don't want to throw away all those FORTRAN libraries.
4. Everybody else does it. Why should I "do the right thing?"
5. *This* has to be done in a hurry. I don't have time to learn a new language.
6. I'm going to learn C first.
7. There's no documentation. How do you expect me to learn this?
8. Have you ever tried getting a C++ program to compile and link? I'm no closet masochist.
9. I need standardization and portability: I don't think I'm going to be at Fermilab much longer, and who knows what computers I'll be using at my new job?
10. Hey!! Any language that was good enough for George Washington is good enough for me!

A theorem, stated and proved OOPly

Let $\underline{\underline{O}}$ be an orthogonal matrix: that is, $\underline{\underline{O}}^\dagger \cdot \underline{\underline{O}} = \underline{\underline{O}} \cdot \underline{\underline{O}}^\dagger = \underline{\underline{1}}$. Let $\underline{\underline{M}}$ be a non-singular square matrix.

If $\quad \underline{\underline{M}} \cdot \underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger = \underline{\underline{O}}, \quad$ then $\quad \underline{\underline{M}}^\dagger \cdot \underline{\underline{O}} \cdot \underline{\underline{M}} = \underline{\underline{O}}.$

Proof: It is obvious that $\underline{\underline{M}}^{-1} = \underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger \cdot \underline{\underline{O}}^\dagger$, for

$$\underline{\underline{M}} \cdot \underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger \cdot \underline{\underline{O}}^\dagger = \underline{\underline{O}} \cdot \underline{\underline{O}}^\dagger = \underline{\underline{1}} \quad .$$

Then, since $(\underline{\underline{M}}^\dagger)^{-1} = (\underline{\underline{M}}^{-1})^\dagger$, we have

$$\begin{aligned} \underline{\underline{O}} &= (\underline{\underline{M}}^{-1}) \cdot (\underline{\underline{M}} \cdot \underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger) \cdot (\underline{\underline{M}}^{-1})^\dagger \\ &= (\underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger \cdot \underline{\underline{O}}^\dagger) \cdot (\underline{\underline{O}}) \cdot (\underline{\underline{O}} \cdot \underline{\underline{M}}^\dagger \cdot \underline{\underline{O}}^\dagger)^\dagger \\ &= \underline{\underline{O}} \cdot (\underline{\underline{M}}^\dagger \cdot \underline{\underline{O}} \cdot \underline{\underline{M}}) \cdot \underline{\underline{O}}^\dagger \quad . \end{aligned}$$

Finally, multiplying on the left by $\underline{\underline{O}}^\dagger$ and right by $\underline{\underline{O}}$ provides the result

$$\underline{\underline{M}}^\dagger \cdot \underline{\underline{O}} \cdot \underline{\underline{M}} = \underline{\underline{O}} \quad .$$

A theorem, stated and proved non-OOPly

Let $\{ A_{ij} \mid i,j = 1 \ldots n \} \subset R$ and $\{ B_{ij} \mid i,j = 1 \ldots n \} \subset R$ be two sets of real numbers satisying

$$\text{(a)} \quad \sum_{k=1}^{n} A_{ik}A_{kj} - B_{ik}B_{kj} = \delta_{ij} \quad \text{and}$$

$$\text{(b)} \quad \sum_{k=1}^{n} B_{ik}A_{kj} + A_{ik}B_{kj} = 0,$$

for all i and j.

Let $\{ U_{ij} \mid i,j = 1 \ldots n \} \subset R$ and $\{ V_{ij} \mid i,j = 1 \ldots n \} \subset R$ be "any" two sets of numbers. If

$$A_{ij} = \sum_{k=1}^{n} \sum_{m=1}^{n} (U_{ik}U_{jm}A_{km} + U_{ik}V_{jm}B_{km} + V_{ik}V_{jm}A_{km} - V_{ik}U_{jm}B_{km})$$

and

$$B_{ij} = \sum_{k=1}^{n} \sum_{m=1}^{n} (V_{ik}V_{jm}B_{km} + V_{ik}U_{jm}A_{km} + U_{ik}U_{jm}B_{km} - U_{ik}V_{jm}A_{km})$$

for all i and j, then

$$A_{ij} = \sum_{k=1}^{n} \sum_{m=1}^{n} (U_{ki}U_{mj}A_{km} + U_{ki}V_{mj}B_{km} + V_{ki}V_{mj}A_{km} - V_{ki}U_{mj}B_{km})$$

and

$$B_{ij} = \sum_{k=1}^{n} \sum_{m=1}^{n} (V_{ki}V_{mj}B_{km} + V_{ki}U_{mj}A_{km} + U_{ki}U_{mj}B_{km} - U_{ki}V_{mj}A_{km})$$

for all i and j.

Proof: *Do it yourself!!*

CLASSIC

Problem: Devise an interface, a set of header files that "everyone" could agree on.

Solutions: The CLASSIC collaboration was formed to come up with such an interface. One of its early accomplishments was designing a logo.

Informally under John Irwin's leadership, several collaboration meetings were held as people tried to hammer out a consistent set of header files. Christoph Iselin, the most prolific member of the collaboration, authored a set of classes used by MAD 9. Nick Walker left the collaboration – or perhaps it left him; I don't know which – to write Merlin, which provides its own programming language. Others have gone their own ways and written their own codes and/or libraries. The collaboration has ceased to exist in any meaningful sense.

Goals for BEAMLINE/MXYZPTLK libraries

- Provide alternative to monolithic code paradigm.
  - Do not create a new language.
  - Do not expect one program to do everything.
  - Use same language used for description and calculations: no special parser, no special input decks.
- Separate physics cleanly from geometry and/or hardware attributes.
  - Local, not global, coordinates used throughout.
  - Default propagator functors for each beamline element class.
  - Specialized functors can be "plugged into" instantiated objects.
  - Allows modifying the physics of beamline element classes without detailed knowledge of the whole class.
- Same software used for tracking is used for all modelling and calculations.
  - Becomes statement about polymorphism.
- Automated map-based perturbation algorithms are implemented "expressively."
  - Basically, this means operator overloading for algebraic classes: **Jet**, **Mapping**, **LieOperator**, ...
  - Hierarchy of libraries: the algebraic classes are assumed by the beamline classes.
- Special purpose "calculator" (**Sage**) and "controller" classes.

Useful books

By now, "tons" of books have been written on OOP/C++ programming. The five I personally found most useful were these:

- Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley. 1986.
- James O. Coplien. *Advanced C++ : Programming Styles and Idioms.* Addison-Wesley. 1992.
- Scott Meyers. *Effective C++ : 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley. 1992.
- Ralph Johnson, John Vlissides, Erich Gamma, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1994.
- Scott Meyers. *More Effective C++ : 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley. 1996.

Programming issues

Using C++ "from the beginning" meant running head on into problems and anticipating solutions that did not yet exist.

- RUN TIME TYPE IDENTIFICATION (RTTI)
- OWNERSHIP, CONTAINERS, AND ITERATORS
- MEMORY MANAGEMENT
- PERSISTENCE
- VISITORS
- NAMESPACES (& NESTED CLASSES)
- BARNACLES
- FUNCTORS
- DYNAMIC POLYMORPHISM (?)
- SAGES
- BROADCASTING
- TEMPLATES
- EFFICIENCY

RTTI

Problem: From the beginning, long before RTTI was implemented, it was necessary to be able to determine the type of an object from its address.

Solutions:

- Member functions — Clone, Name, and Type — pure virtual in **class bmlnElmnt**, enable a basic level of emulating RTTI-like behavior.

```
-----------------------------------------------------
class bmlnElmnt {
  ...
  inline char* Name() { return ident; }
  virtual bmlnElmnt* Clone() = 0;
  virtual char* Type() = 0;
  ...
};
-----------------------------------------------------
class sbend : public bmlnElmnt {
  ...
  bmlnElmnt* Clone() { return new sbend( *this ); }
  char* Type() { return "sbend"; }
};
-----------------------------------------------------
```

- **Clone** return a new beamline element which is a copy of the one invoking it; **Type** returns a string which names the particular, derived type of beamline element. (It has been useful in setting up criterion functions for Calculator objects.)
- Notice the *error* of inlining the derived functions. This was eventually fixed.
- Recasting of returned pointers of cloned objects offended purists.

Ownership, Containers, and Iterators

Problem: A **beamline** contains a set of elements – quadrupoles, dipoles, cavities, and so forth – any one of which may be other beamlines. Does the beamline "own" its elements? What container should be used?

Solutions:

- My first class: extending **slist** to **dlist**.
- Early mistake: use inheritance.
  - ties representation too closely to object
  - public inheritance increased possibility of error
- Early, but probably correct decision: a **beamline** should not "own" its elements.
  - in opposition to later STL philosophy.
  - **beamline** does not have responsibility for object.
  - handle available for outside influence.
  - same object may appear in several places.
- Should I now use the STL?
  - negates the "ownership" decision (?)
  - still must decide on container
  - nodal objects require extra functionality (e.g., ordering)
  - time tests indicate no clear advantage for iterator usage
  - Bottom line: I'll probably eventually convert my old libraries by using the STL but am in no hurry to do so.

Memory Management 101: Garbage Collection

Problem: Large containers of volatile objects are created and destroyed on the fly. Copy constructors waste time reproducing data unnecessarily.

Solutions:

- Early attempt (c.1991) involved inheriting from a base class **Junk**, which possessed a *virtual destructor*.
  - All dynamic, volatile objects were derived from **Junk**.
  - Their pointers were stored in an **Attic** object.
  - When the number of **Junk** objects reached some critical value, a percentage of them – the oldest – were deleted. This was, of course, a risky scheme.
- In 1993-4 I read Coplien's *Advanced C++ Programming Styles and Idioms* and learned about the "envelope-letter" scheme. This fundamental procedure was an eye-opening introduction to reference counting in garbage collection.
  - Nothing deleted until it was no longer needed.
  - Garbage collection occurred on the fly.
  - Data existed only in one place; no large objects need ever be copied.
- Equally serious problem: efficiency of dynamically creating and destroying large numbers of objects.
  - Precreate pools of objects to draw from.
  - This works for objects of well-defined sizes.

Persistence

Problem: It is essential that beamlines created by one program be retrievable by another. Thus, beamlines and their elements must exist outside the programs; they must "persist." One (and only one) key issue involves objects containing pointers to other objects that may or may not be unique.

Solutions:

- Important words: "pickling" and "swizzling."
- Earliest solutions utilized streaming and virtual "writeTo" functions.
  - base class data is first streamed; derived class data is then written with **writeTo**
  - essential problem of pointers to objects or functions was not handled correctly
- Our most recent solution, written by a Summer student, uses Joseph Hesse's EZSave.
- A "Factory" object becomes either essential or exceedingly convenient.
- "Best" solution: an object oriented data base?
- Unfortunately, the C++ language does not provide persistence.

Visitors

Problem: Adding functionality to **class beamline** required constantly changing header files and source code.

Solution:

- Many calculations proceeed in a sequential manner, by processing one beamline element at a time.
- An early CLASSIC idea was the "Ferret" or "Algorithm," a base class for objects that could be propagated through beamlines and did specific calculations by changing their states as they traversed the elements.
- This anticipated and was superseded by the "visitor" pattern.
  - Base class BmlVisitor contains virtual functions, **.visitXXX**, that get overwritten.

```
class BmlVisitor
{
public:
  virtual void visitBeamline( beamline* );
  virtual void visitBmlnElmnt( bmlnElmnt* );
  virtual void visitDrift( drift* x )
    { visitBmlnElmnt( (bmlnElmnt*) x ); }
  virtual void visitRbend( rbend* x )
    { visitBmlnElmnt( (bmlnElmnt*) x ); }
  ...
};
```

  - Base class bmlnElmnt contains a pure virtual function, **.accept**, that is overwritten.

```
class rbend : public bmlnElmnt
{
...
  void accept( BmlVisitor& v ) { v.visitRbend( this ); }
};
```

Barnacles

Problem: Different calculator objects (e.g., visitors) may be able to use information generated by earlier calculations. Without invoking centralized organization, how can a later calculator know whether the earlier calculation has been done and, if so, where to find the information?

Solution:

- **Barnacle** was a base class for a "Post-It" note that could be attached to beamline elements.
- A calculator would attach its own information, in an object derived from **Barnacle**, to a beamline element.
- The information attached to an element is retrieved via a keyword search.
- This was something akin to an STL multimap, except that the "data" could be anything whatsoever.

Functors

Problem: Separate the physics cleanly from the geometry – or any other considerations. Allow for dynamic specialization.

Solution: The concept of a functions that is also an object was very useful. Default propagators could be instantiated for every class of beamline element. Specialized propagators could be written for special cases (e.g., high order symplectic integrators for very thick, very strong magnets). Their states could be adjusted by control objects, They could even be plugged into the element dynamically at run time.

Dynamic Polymorphism

Problem: It would be very convenient if a variable should be able to change its type in the middle of a program: for example, from a **Jet** and **double**.

Solutions:

- Again use "envelope-letter," where the high level object is only a holder for a pointer to the data.
- Double dispatch provides a mechanism for "doubly virtual" operators.
- There are several ways to do this. The one that I chose was to maintain "doubly indexed function tables."

```
----------------------------------------------------
MatrixD operator* ( const MatrixD& x, const MatrixD& y ) {
  return MatrixD( MLD::ProdVtbl[ x._pml->vtblIndex() ]
                               [ y._pml->vtblIndex() ]
                               ( x._pml, y._pml )      );
}
----------------------------------------------------
void MLDDiag::initOps()
{
  for( int i = FPCL_INDEX_D_BASE;
       i < FPCL_MLD_TYPES;
       i++ ) {
    SumVtbl [ FPCL_INDEX_D_DIAG ][ i ] =  SumDiagBase;
    SumVtbl [ i ][ FPCL_INDEX_D_DIAG ] =  SumBaseDiag;
    DiffVtbl[ FPCL_INDEX_D_DIAG ][ i ] = DiffDiagBase;
    DiffVtbl[ i ][ FPCL_INDEX_D_DIAG ] = DiffBaseDiag;
    ProdVtbl[ FPCL_INDEX_D_DIAG ][ i ] = ProdDiagBase;
    ProdVtbl[ i ][ FPCL_INDEX_D_DIAG ] = ProdBaseDiag;
  };
  SumVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] =  SumDiagDiag;
  DiffVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] = DiffDiagDiag;
  ProdVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] = ProdDiagDiag;
}
----------------------------------------------------
```

Sages

Problem: People misuse the software, sometimes by not thinking correctly about the physics (e.g., not turning off RF before calculating a closed orbit). Without resorting to the monolithic code paradigm, can calculations be made more foolproof?

Solution:

- **Sage** is a class that knows how to do a particular category of calculations correctly.
- It uses lower level calculator and beamline classes.
- This comes dangerously close to writing monolithic codes.
- It fundamentally violates an original intent of the BEAMLINE/MXYZPTLK libraries: they were not supposed to be "foolproof." They were meant to be used by those who understood the physics (and mathematics), knew what they were doing, and would do it correctly.
- The terms "wizard" and "expert system" were already taken, so I used "Sage."

Broadcasting; event handling

Problem: It is necesary to inform categories of objects about changes in a beamline. You must "broadcast" information to all existing objects satisfying some criterion.

Solution:

- This arises in connection with graphics applications constantly.
- There is a central message server which keeps a registry of all objects that respond to particular kinds of events.
- Objects are registered and deregistered by their constructors and destructors.
- Ask Jean-Francois Ostiguy for details.

Some of the things I've learned: aphorisms and potential epitaphs

- GLOBAL
  - Those who don't see the problem can't appreciate the solution. (You find what you look for.)
  - What you think is not important usually is.
  - Good, clean, simple ideas have a short half-life before decaying to an incomprehensible state.
  - The monolithic code paradigm rules.
  - Do something wrong five times before doing it right.
  - Emergence of the ANSI standard enables us to state unambiguously, "My compiler is not ANSI compliant."
  - Puritanical programmers are a pain.
  - Whoever said, "It is better to have failed than never to have tried," was wrong.
- LOCAL
  - Computing Division will support OOP/C++ software development for HEP but not accelerator modeling.
  - Unless it is directly related to controls, Beams (Accelerator) Division will provide at best Level 1 Support for software development (esp. OOP/C++).
  - Only hardware, not software, developers can talk publicly about things that don't work and expect a sympathetic audience.