JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 3, January 2006

The C++ Design Patterns for Solvers Used in Object-Oriented Computing and Simulation Models

Alexander I. Kozynchenko*
Mid Sweden University, Sundsvall, SE-851 70, Sweden

I. Introduction

THIS note touches a problem of developing generic mathematical software for object-oriented modeling and simulation of complex dynamic systems. More specifically, it focuses on designing the C++ *class* templates that could communicate with source classes comprising an object model and could process their data. This problem is of great importance for models involving a large number of different classes (types) with complex data processing. In many instances, aerospace applications fall under this category. For example, navigation, control, and stabilization systems of an aircraft model require, beyond the continuous updating of the state variables, estimation and filtering of noisy data from sensors, which, in turn, have to be simulated somehow. These specialized tasks could be effectively solved by using corresponding class templates, or *design patterns.* C++ seems to be a language of choice for this purpose because of its sound template support, as compared with such popular object-oriented languages as Java and C#. The importance of the C++ generic programming attracts growing attention of software developers to this theme.[1] The C++ language includes the Standard Template Library (STL) that provides a great number of global function templates, so-called *generic algorithms*, designed for processing collections of objects in containers of different kinds. These algorithms implement such general purpose routines as sorting, searching, and merging. In order to meet the needs of more complex algorithms for scientific computing in C++, a library of global *functions* has been created for a great variety of numerical methods from linear algebra to partial differential equations (PDEs).[2] Unfortunately, this vast library cannot be used in object models immediately, without extra programming efforts. In the textbook[3] the instance of a solver *class* template was described, which is intended for solving initial-value problems (IVPs) for first-order ordinary differential equations (ODEs). The main feature of the solver is the usage of function pointers to the source *global* functions describing the right-hand sides of ODEs. However, this solver pattern cannot interact with classes of an object model through its inherent inability to access class *members*. Summing the background of the problem, one can come to a conclusion that there is still no any adequate approach to designing solvers for scientific computing to be embedded into an object model for processing the *object* data.

The objective of the current note is to show how to use available C++ facilities for creating design patterns for solvers to be acting as equal parts of generic object models, thus to fill the gap. Such solvers must meet the following general requirements:

- be designed as class templates rather than function templates to ensure that intermediate data are retained;
- provide access to public data of a source class;
- have a mechanism of returning the newly calculated data to the source class at each step of processing (e.g., integration of ODE), since such a functionality is essential in many simulation models.

The note exemplifies these requirements in designing templates for two kinds of ODE solvers intended for solving both IVPs and two-point boundary-value problems (BVPs). The reason for the choice is that generic models including dynamic objects described by ODEs are often used for pure computing or continuous simulation in research works

Received 29 June 2005; revision received and accepted for publication 22 November 2005.
* Lecturer in Computer Science, Department of Information Technology and Media, Mid Sweden University, Sundsvall, SS-851 70, Sweden. alexander.kozynchenko@miun.se

and many applications ranging from aerospace systems and optimal control to game development. The basic idea is the use of mechanism of pointers to member functions that allows design patterns to be effectively incorporated into generic object models. In case of ODE solvers, member pointers provide access to source class member functions that describe the right-hand sides of ODEs, have control over the integration process, and make specialized computations.

II. Design Pattern of the ODE Solver for IVP

From a variety of numerical methods for solving initial-value problems for systems of ODEs, the standard 4th order fixed-step-sized Runge-Kutta method has been chosen. The declaration of the corresponding IVP solver template involves, together with the customary private and public sections, a set of type definitions (**typedef**) for pointers to source class member functions laying down minimum requirements on the public interface of a *source* class. Thus proper matching is established between a solver and a source class. Pointers to member functions can be passed to a solver template by two ways. The first way, which is adopted in the note, uses arguments of the solver *constructor*, whereas another, less convenient scheme passes member pointers as *template parameters*. Now we can consider an example of solver template declaration in detail:

```
template<typename DynamicSystem, int order>
class RungeKutta
{
    typedef double (DynamicSystem::*PMemberFunc) (double* x);
    typedef void (DynamicSystem::*PUpdate) (double* x);
    typedef bool (DynamicSystem::*PCond) (int jc);
private:
    DynamicSystem * pDS;
    double h, x0[order+1], x[order+1];
    double kx[order][4];
    PMemberFunc pMF[order];
    PUpdate pUpdate;
    PCond pCond;
public:
    RungeKutta(double h, DynamicSystem * pds, double* xi,
               PMemberFunc* pmf, PUpdate pUp, PCond pCond);
    double& GetX0(int i) { return x0[i]; }
    void Run(int jc);
};
```

In its template part, it has a template argument, `DynamicSystem`, referring to a source class of an object-oriented model that uses this solver. The non-template argument `order` indicates an order of ODEs system to be integrated. The declaration starts with defining pointers to `DynamicSystem's` member functions, thus displaying required prototypes of the source class member functions. Here, `PMemberFunc` is declared as a pointer to a member function representing the right-hand side of ODE, the member pointer `PUpdate` gives access to a member function that updates state variables at each step, and, then the member pointer `PCond` is bound to the predicate that provides the termination condition of integration process. The integer argument `jc` indexes the variable (dependent or independent) used in the termination condition.

In the private section, the data members of the class template `RungeKutta` include the pointer `pDS` to the corresponding source object with which the solver is to interact. The variable `h` is a step size of integration, and the elements of arrays `x0` and `x` are initial and current values of the independent variable (usually time) and state variables of ODEs (e.g., distances, velocities, accelerations). The auxiliary array `kx` keeps temporary values used in the 4th-order Runge-Kutta method. Pointers to the right-hand side `DynamicSystem`'s member functions are represented as elements of the array `pMF`. The member pointers `pUpdate` and `pCond` of corresponding types are also declared.

In the public section, a parameterized constructor is declared that creates solver objects that match corresponding model ones. The member function `Run`, being the core of the solver, integrates ODEs by the Runge-Kutta method in point, starting from initial boundary values. At each step, the advanced values are returned to the source object by calling the `DynamicSystem`'s helper function pointed to by the member pointer `pUpdate`. This helper function could also make some control decisions concerning both the use of intermediate data and the process rate. Access to the `DynamicSystem`'s member functions is provided by the operator $->^*$ that binds corresponding pointers to member functions with the source object pointed to by `pDS`. The integration process is going on in the `while`-loop until the termination condition is valid, that is, the function call (`pDS->*pCond`) (`jc`) returns `true`.

III. Design Pattern of the ODE Solver for Two-Point BVP

An objective of this section is to show the structure of a solver template for a two-point BVP, taking as an example the pure "shooting" method. A class template `Shooting` has been developed for this purpose. Specifically, this pattern is intended for a simple free BVP defined as follows. For a system of ODEs of order n,

$$\frac{d\vec{x}}{dt} = \vec{f}(t, \vec{x}),$$

where $\vec{x} = (x_1, \ldots, x_n)$—$n$-vector of dependent (state) variables, $\vec{f} = (f_1, \ldots, f_n)$—$n$-vector of nonlinear right-hand side functions, there are specified:

1) n-1-vector of initial conditions $\vec{x}(t = t_0) = \vec{x}_0$,
2) two boundary conditions $x_i(t = t_f) = x_{if}$ and $x_j(t = t_f) = x_{jf}$, $i, j = \overline{1, n}$, $i \neq j$, at some unspecified final point t_f. After eliminating the time t_f, we get one condition to be satisfied at the right end: $x_i(x_j = x_{jf}) = x_{if}$.

To solve the described BVP means to determine a freely specifiable initial value, say x_{k0}, initially supposed to lie in some interval $[x_{k0\,\min}, x_{k0\,\max}]$. This initial value x_{k0} may be found as a root of the nonlinear equation $\Delta x_{if}(x_{k0}) = 0$, where the discrepancy is $\Delta x_{if} = x_i(x_j = x_{jf}) - x_{if}$. The dependence $\Delta x_{if}(x_{k0})$ cannot be expressed analytically in the general case, but we can calculate it by repeatedly solving the IVP until $x_j = x_{jf}$. If $\Delta x_{if}(x_{k0\,\min})$ and $\Delta x_{if}(x_{k0\,\max})$ have opposite signs, the iterative process of finding a root must converge. In the class `Shooting`, the *false position* method is chosen for determining the root (see, for example,[2], pp. 358–361).

The declaration of the class `Shooting` involves one template argument, namely, `DynamicSystem`, concerning a source object of a model. It means that it is the source object that is responsible for providing the IVP solver to be used in the "shooting" algorithm. Like the class `RungeKutta`, the class `Shooting` places some requirements upon the interface of the class `DynamicSystem`, which have to contain two member functions with prototypes being defined by pointers to member functions of types `RunIVP` and `GetX`. The first function pointed to by `RunIVP` accomplishes the integration of IVP, taking three arguments—the index `k` and initial value `xk0` of the freely specifiable state variable x_k, as well as the index `j` of the state variable x_j used in the second boundary (termination) condition. The second function pointed to by `GetX` gets access to state variables used in the BVP solution. The declaration of the class template `Shooting` is shown below:

```
template<typename DynamicSystem>
class Shooting
{
    typedef void (DynamicSystem::*RunIVP)(int k, double xk0, int j);
    typedef double& (DynamicSystem::*GetX)(int);
private:
    double xmin, xmax, ymin, ymax;
    int i, j, k;
    double xf, error;
    DynamicSystem* pDS;
    RunIVP runIVP;
    GetX getX;
```

```
public:
    Shooting(DynamicSystem * pds, double xmin, double xmax,
             int i, int j, int k, double xf, double error,
             RunIVP runIVP, GetX getX0)
    void Run();
};
```

The data members `xmin` and `xmax` denote a range of varying the unspecified initial value of the state variable x_k. The data members `ymin` and `ymax` denote corresponding misses Δx_{if}. The integer data members `i`, `j`, and `k` are indexes of the state variables x_i, x_j, and x_k. The data member `xf` indicates the desired value x_{if} at the final point t_f, and the data member `error` sets the admissible error Δx_{iferr} in determining the root x_{k0}. All data members, including pointers, are initialized in the parameterized constructor. The "shooting" algorithm is performed by the member function `Run` using the false position method. The function `Run` calls the member function of a source class `(pDS->*runIVP)(k, x, j)` to be making successive trial "shots" for finding discrepancies `ymin` and `ymax`. The argument `x` is to be iteratively recalculated according to the formula

$$x_{n+1} = x_n - y_n \cdot (x_n - x_{n-1})/(y_n - y_{n-1}).$$

After the condition `abs(y) < error` has been satisfied, the approximate root value x_{k0} is passed to the source object in the statement `(pDS->*getX)(k) = x;`.

IV. A Case Study: Using the BVP Solver in Trajectory Control for Re-Entry of a Space Vehicle

Aerospace systems are based on laws of mechanics and therefore provide a variety of guidance and control problems convenient for testing BVP solvers–from spacecraft rendezvous to soft landing on planets. Here, it is considered an example related to trajectory control for a re-entry vehicle. This problem has been investigated over the past 45 years (see, e.g.,[4]). The two-dimensional model of center-of-mass motion of an unpowered low-lift re-entry vehicle in the stationary atmosphere of a non-rotating spherical Earth is described by the following ODEs system of the 4th order:

$$\begin{aligned}
\dot{V} &= -\kappa \cdot \rho(y) \cdot V^2 - g(y) \cdot \sin\gamma \\
\dot{\gamma} &= \kappa \cdot u \cdot \rho(y) \cdot V - [g(y)/V - V/(R_e + y)] \cdot \cos\gamma \\
\dot{y} &= V \cdot \sin\gamma \\
\dot{x} &= [R_e/(R_e + y)] \cdot V \cdot \cos\gamma
\end{aligned} \tag{1}$$

where overdot indicates differentiation with respect to time t, V—velocity, γ—flight-path angle, y—altitude, x—range, κ—ballistic coefficient, u—lift-to-drag ratio,

$$g(y) = g_0 \cdot [R_e/(R_e + y)]^2 \tag{2}$$

is acceleration of gravity, $g_0 = 9.81\,\mathrm{m/sec^2}$ is acceleration of gravity at sea level, $R_e = 6371\,\mathrm{km}$ is the Earth's radius,

$$\rho(y) = \rho_0 \cdot e^{-\beta \cdot y} \tag{3}$$

is the atmospheric density if an isothermal atmosphere is assumed, $\rho_0 = 1.225\,\mathrm{kg/m^3}$ is the atmospheric density at sea level, $\beta = 0.00014\,\mathrm{m^{-1}}$ is the logarithmic gradient of density.

We will consider the two-point BVP of determining the flight-path angle $\gamma(t_0) = \gamma_0$ at entry point under specified boundary conditions. The initial conditions of the BVP includes initial velocity at entry point $V(t = t_0) = V_0$, altitude $y(t = t_0) = y_0$, and range $x(t = t_0) = x_0$. The second boundary (terminal) conditions are $x(t = t_f) = x_f$ and $y(t = t_f) = y_f$, or, having eliminated t_f, $x(y = y_f) = x_f$. Such a problem can be effectively solved by the

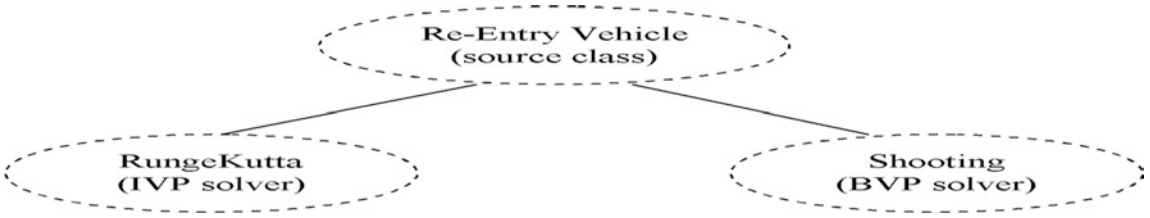

Fig. 1 Class diagram of the simulation model of atmospheric re-entry.

developed design patterns `Shooting` and `RungeKutta`. The class diagram for the simulation model involving both the solvers and the source class `Reentry` is depicted in Fig. 1.

The declaration of the class `Reentry` (with some functions defined inline) is presented below:

```
class Reentry
{
    typedef double (Reentry::*F)(double* var);
private:
    enum {SIZE = 5, ORDER = SIZE-1};
    Shooting<Reentry>* pSh;
    RungeKutta<Reentry, ORDER>* pRK;
    const double PI, G0, RE, dens0, beta;
    double ht, kappa, u;
    double var[SIZE]; //variables t, v, gamma, y, x;
    double varf[SIZE]; //values of variables at the final point
    double Dens(double y);
    double G(double y);
    double FV(double* var);
    double FGamma(double* var);
    double FY(double* var);
    double FX(double* var);
    F f[SIZE-1];
    void RunRK(int k, double xk, int j) { pRK-> GetX0(k) = xk;
               pRK-> Run(j); }
    void Print();
public:
    const double D_R; //conversion from degrees to radians
    Reentry(double gamma0, double v0, double y0, double x0, int ivarf,
            double varfp, double ht, double t0 = 0);
    ~Reentry();
    F* GetF() { return f; }
    void Exchange(double* varp);
    bool Stop(int j) { return var[j] < varf[j]; }
    double& GetVar(int i) { return var[i]; }
    double* GetVar() { return var; }
    void RunShooting(double varMin, double varMax, int i, intj, int k,
                     double varf, double miss);
    //part of interface containing access functions GetT, GetV, etc.
    ...
};
```

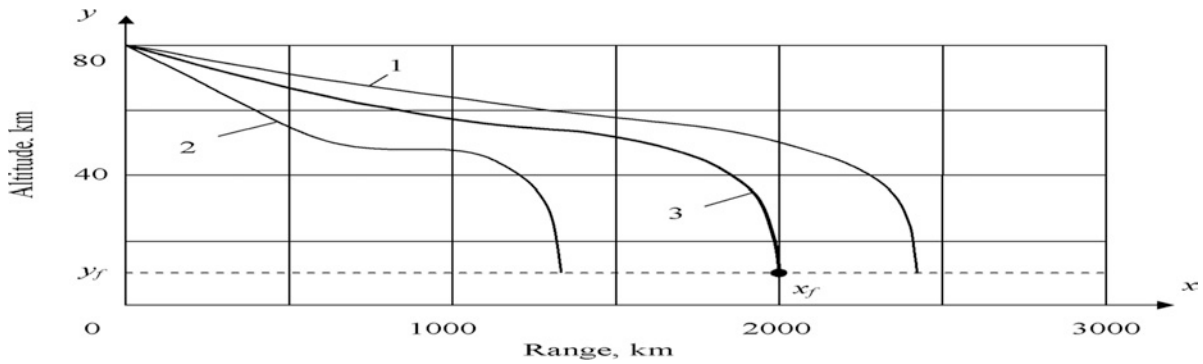

Fig. 2 Initial and final trial trajectories in atmospheric re-entry guidance.

The class contains pointers `pRK` and `pSh` to classes `RungeKutta` and `Shooting` correspondingly, which are bound to objects created dynamically in the constructor:

```
pRK = new RungeKutta <Reentry, ORDER> (ht, this, var, f, Exchange, Stop);
```

as well as in the member function `RunShooting`:

```
pSh = new Shooting <Reentry> (this, gammaMin*D_R, gammaMax*D_R, i, j, k,
                              varf, miss, RunRK, GetVar);
```

The member functions `FV`, `FGamma`, `FY`, and `FX` compute right-hand sides of the set of ODEs (1). The helper functions `G` and `Dense` compute the acceleration of gravity and the atmospheric density on formulas (2) and (3), correspondingly.

A numerical example aimed to determine the initial flight-path angle when re-entry is illustrated in Fig. 2, where the trial trajectories 1, 2, and 3 relate to $\gamma_0 = \gamma_{0\min}$, $\gamma_0 = \gamma_{0\max}$, and required $\gamma_0 = \gamma_{0f}$, correspondingly. Data used in the test are as follows: $\kappa = 0.001\,\mathrm{m}^2/\mathrm{kg}$, $u = 0.2$, $V_0 = 7800\,\mathrm{m/sec}$, $y_0 = 80\,\mathrm{km}$, $x_0 = 0\,\mathrm{m}$, $\gamma_{0\min} = -1^\circ$, $\gamma_{0\max} = -3^\circ$, $y_f = 10\,\mathrm{km}$, $x_f = 2000\,\mathrm{km}$, admissible range miss $\Delta x_f = 10\,\mathrm{km}$. The driver function `main` has the following view:

```
int main()
{
    Reentry rv(-2, 7800., 80e3, 0., 3., 10e3, 1.);
    rv.RunShooting(-1.0, -3.0, 4, 3, 2, 2000e3, 500.0);
    return 0;
}
```

The sought-for initial flight-path angle $\gamma_{0f} = -1.522^\circ$ has been found after 5 iterations with the miss $\sim$5 km.

V. Conclusions

The class templates presented in the note may be considered as the first ODEs IVP and BVP solvers to be completely incorporated in diverse generic object-oriented computing and simulation models written in C++. Usage of the mechanism of pointers to member functions gives the solvers a capability to communicate effectively with dynamic objects forming a model. The design patterns may serve as examples for developing great families of object-oriented solvers using other algorithms and applicable not only to ODEs but also to PDEs, linear algebraic equations, optimization, and many other fields of numeric computing. Their usage would be especially convenient in the aerospace simulation models which are frequently populated with numerous dynamic objects involving optimal control, navigation, and decision-making systems.

References

[1]Vandevoorde, D., and Josuttis N. M., *C++ Templates: The Complete Guide*, Addison-Wesley, Boston, 2002.

[2]Press, W. H. (Ed.), Teukolsky, S. A. (Ed.), Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in C++: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, Cambridge, 2002.

[3]Yang, D., *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*, Springer, New York, 2001, pp. 194–198.

[4]Regan, F. J., and Anandakrishnan, S. M., *Dynamics of Atmospheric Re-Entry*, AIAA Education Series, AIAA, 1993.