Novel and Adapted Object-Oriented Design Patterns for Scientific Software

Jesus A. Izaguirre
University of Notre Dame
Department of Computer Science and Engineering
384 Fitzpatrick Hall
Notre Dame, IN 46556
izaguirr@cse.nd.edu

Thierry Matthey
University of Bergen
Para//ab
Thormohlens Gate 55
N-5008 Bergen, Norway
matthey@ii.uib.no

Trevor Cickovski
University of Notre Dame
Department of Computer Science and Engineering
325 Cushing Hall
Notre Dame, IN 46556
tcickovs@nd.edu

Abstract

The ABSTRACT is to be in fully-justified italicized text, at the top of the left-hand column, below the author and affiliation information. Use the word "Abstract" as the title, in 12-point Times, boldface type, centered relative to the column, initially capitalized. The abstract is to be in 10-point, single-spaced type. The abstract may be up to 3 inches (7.62 cm) long. Leave two blank lines after the Abstract, then begin the main text.

Special thanks to Joseph Coffland and Todd Schneider for their contributions to the design of CompuCell.

1. Introduction

While object-oriented design patterns have been explored for many years in software engineering [9, 3, 17], their application to scientific software is just beginning to unfold. As recently as 2002, the bulk of scientific software was written in either C or Fortran due to the computational overhead of object-oriented languages and heavy emphasis on fast mathematical algorithms in scientific computing [4]. However, as scientific software frameworks grow larger, so also does their need to be extensible, flexible, and maintainable. Maintainable software is highly desirable, as various studies have shown maintenance to consume on average 65 to 75 percent of the overall software life cycle cost [1].

Generic patterns help to address these issues by providing a solution to a problem that can be used across multiple application domains [18]. We present several design patterns for scientific platforms that we have implemented in the object-oriented language C++, along with the specific goals behind their use. Some are novel, and others are modifications of known design patterns. We also introduce two sample scientific frameworks on which we applied and tested these techniques. ProtoMol [14] is a framework for conducting molecular dynamics simulations, and CompuCell [6] is an engine for three-dimensional simulation of morphogenesis. We will explain how these techniques have made these frameworks more maintainable, as well as their direct performance benefits. For example, the application of some of these techniques to CompuCell yielded fourfold improvements in efficiency and tenfold improvements in memory consumption.

2. Adapted Patterns

2.1. Generic Automaton

Gamma *et al.* [9] provide a behavioral State pattern and apply it to a TCP connection example. Their pattern has the intent of allowing an object to change behavior depending on the value of its internal state. Simulation objects in scientific software often also need to maintain state and vary behavior in a similar fashion, with the state affected by both internal and external conditions. An example of this is cellular automata, which have been applied to various studies of the biological cell [2, 15].

Our generic automaton is a set of libraries which provide

an interface to a structure that (1) accepts objects whose operation and behavior depend on the value of some internal state, and (2) contains a set of rules that govern the transition of these objects from one state to another. To ensure maintainable software, these libraries must not make it difficult to add new states or rules. For the sake of flexibility, this interface also cannot cause complications if transition rules depend on varying numbers of inputs. We have designed our generic automaton with these goals in mind.

In cellular automaton terminology, cells are often said to transition between 'type's, and 'state' is subsequently used as the cell's set of internal variable values [5]. Although the application of our generic automaton is not limited to cellular automata, we do figure cellular automata to be a potentially common use of these libraries. As a result, in our generic automaton we use the term 'type' rather than 'state' to avoid potential confusion. For interfacing with our generic automaton, the user needs only to create some class to represent a simulation object, containing a variable of type `unsigned char` to hold the current type of the object (from now on we will use `Object` as the name of this class and `type` as the name of the variable, but these names are irrelevant). For a model with *n* different types we assume `type` to contain a value between 0 and *n-1* as an unsigned character, with each value representing one type.

Figure 1 expresses this design in UML. An abstract class `Transition` provides the interface for individual transitions. `Transition` contains a variable of type `unsigned char` which holds the type to which the transition is going, and a function `checkCondition()` which returns 'true' if the passed `Object` should switch to this type. `checkCondition()` accepts any necessary parameters to encompass all internal and external conditions relevant to the automaton. For example, it may be necessary to pass a grid of chemical concentrations or the value of a global timer.

A set of rules for changing type is contained within the class `ObjectType` (this name is also flexible). Each `ObjectType` contains an array of `Transition` objects, and by default the `checkCondition()` method of each added `Transition` is invoked in the `ObjectType::update()` function, and the `type` member of the passed `Object` changes to the type that the first true `Transition::checkCondition()` goes. If all invocations to `checkCondition()` return false, the type of the `Object` does not change.

The class `Automaton` supplies the highest-level interface, and each `Automaton` object may contain multiple `ObjectType`s. We allowed this in our libraries to give users more flexibility. For example, it may be desirable for an object to operate under different sets of rules depending on the current simulation status. The set of rules to use can be controlled in the `Automaton::update()` function which can be overridden accordingly in child classes. By default there is just one `ObjectType` data member whose `update()` method is invoked in `Automaton::update()`.

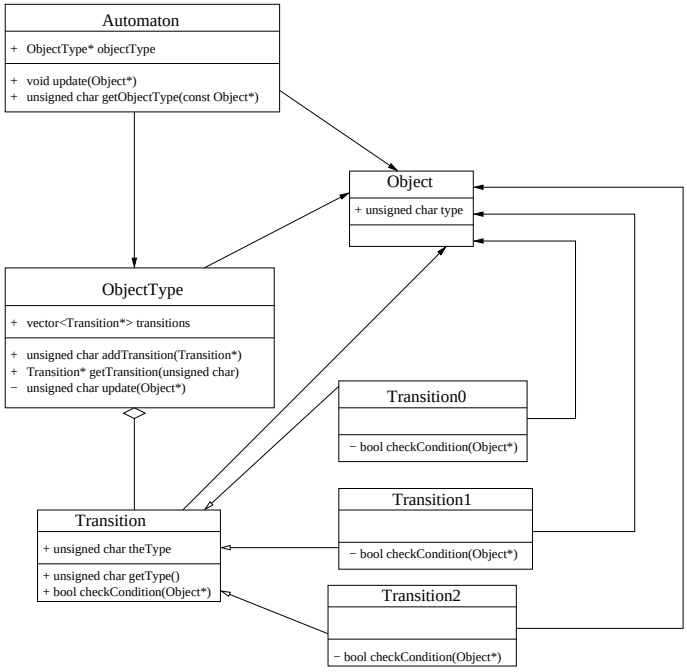

Figure 1. Implementation of the generic automaton in UML.

2.2. Offset Neighbor Evaluation

This technique specifically applies to simulations that operate on a three dimensional grid. It is often desirable, given a pixel in the grid, to quickly determine the location of a pixel's neighbor points and attributes (i.e. environmental conditions). An example application of this is the Cellular Potts Model [10]. For two-dimensional grids, two dimensional arrays can be used for quick access without consuming large amounts of memory. In three dimensions this is infeasible. We thus create an algorithm for finding neighbors that employs two known software techniques: lazy evaluation and information caching, with the goals of improving efficiency and at the same time consuming less memory.

The algorithm assumes that for a pixel to be a 'neighbor', it must be close enough, lying within some distance *D* (not necessarily directly bordering, though this can be the case by setting *D* to be small enough). Neighbors for the point *(X, Y, Z)* are then calculated with respect to the origin, and their coordinates are translated by *(X, Y, Z)*. Once calculated, the neighbors with respect to the origin are cached so that they can be used by future requests for neighbors and

Algorithm 1 Pseudo-code of offset neighbor evaluation.

Neighbor Finder:

1. Pre-processing: Initialize x := 0 and $neighbor\_array$ to be empty. $neighbor\_array$ is an array of pairs of points and integer distances, and n := 0;
2. **getNeighbor(int** n**, double &**D**)**
   (a) **while** length of $neighbor\_array < n$
      i. $x := x + 1$;
      ii. **for** each (X, Y, Z) such that $x = X^2 + Y^2 + Z^2$
         A. **for** each unique point Q that is a rotation of (X, Y, Z) around the axes
            Add (Q, $\sqrt{x}$) to $neighbor\_array$;
   (b) D := $neighbor\_array[n]$.distance;
   (c) **return** $neighbor\_array[n]$.point;
3. To look at level 1 neighbors, distance 1 from a point P:
   (a) **do**
      i. $neighbor$ = getNeighbor(n, D) + P;
      ii. ... Do something with neighbor ...
      iii. $n := n + 1$;
   while $D <= 1$

translated appropriately. More neighbors are computed only if more neighbors need to be calculated for a point than already have been cached. This saves computation time, and also saves memory since information on all pixel neighbors in the grid does not have to be stored.

Algorithm 1 provides pseudocode for our neighbor-finding algorithm.

2.3. Put MD Adapted Patterns Here

3. Novel Patterns

3.1. Plugins

It can be a desirable feature of scientific software to possess optional simulation functionality, i.e. functionality that can be easily added or removed from a simulation without any difficulty. For example, a biologist may want to view cellular behavior with and without mitosis; a chemist may want to view the affect of adding or removing a chemical from a certain medium. Even better, it would be good if decisions on whether or not to load certain object files that provide optional features could be postponed until runtime, minimizing executable size.

A plugin provides one general feature to a framework that is optionally included or excluded from particular simulations, and dynamically loaded at runtime so that modifications to the feature do not require recompilation of the entire framework. The method for adding or removing the feature provided by a plugin in a particular simulation can be left in the control of the user through for example a configuration file.

We provide three generic classes to handle plugins. The most basic is `BasicPluginInfo`, which contains four data members along with `get()` accessor methods for each one: `name` (the name of the plugin), `description` (a sentence description of what the plugin accomplishes), `numDeps` (the number of other plugins that this plugin depends on), and `dependencies` (an array of the plugin names that this plugin depends on - these are automatically loaded if this plugin is loaded). Plugins are managed by a `BasicPluginManager`, which holds an array of all plugins along with a corresponding array of factories for plugin object creation, and controls all dynamic loading. Each plugin within a simulation needs to define a *plugin proxy* which makes the final connection by registering the plugin with the manager. A `BasicPluginProxy` defines an `init` method to accomplish this task.

The UML inheritance structure for plugins is defined in figure 3.

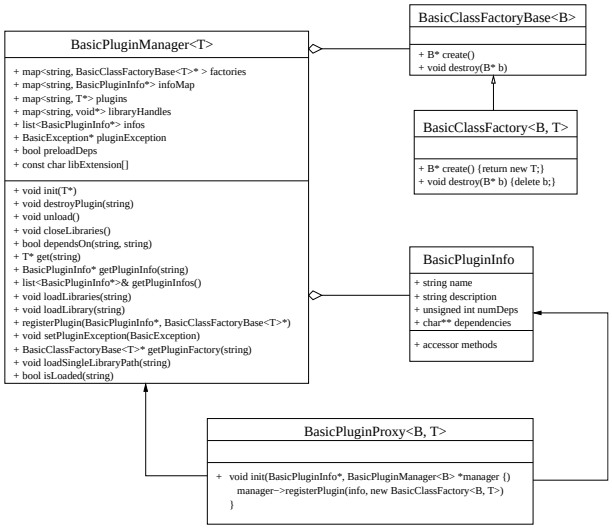

Figure 2. UML diagram of our Plugin libraries. A new plugin is created by a factory object, following the factory design pattern [3]. All plugins are managed by a PluginManager singleton. This object manages plugin attributes, factories and dynamically loaded libraries. Plugins are registered with the manager through their respective plugin proxy.

Plugins are dynamically loaded through the following C++ snippet:

```
BasicPluginManager<Plugin*> pluginMan-
ager;
char *pluginPath = "<path>/plugins";
pluginManager.loadLibraries(pluginPath);
```

The class `Plugin` can define any basic functionality common to all simulation plugins. A new plugin `X` can be defined and compiled into shared object files, in the directory `<path>/plugins/X`:

```
class XPlugin :  public Plugin {
....  functionality ....
}
```

Finally, a plugin proxy, accepting all `PluginInfo` (name, function, plugin dependencies) along with the manager, is linked into the shared object files. For example, if plugin `X` depended on two other plugins `Y` and `Z`:

```
BasicPluginProxy<Plugin, XPlugin>
xProxy("X", "Implements function X",
(const char *){"Y", "Z"}, &pluginMan-
ager);
```

This would imply that if plugin `X` is loaded, plugins `Y` and `Z` are also loaded, no matter what.

By providing this generic interface and the ability for a user to define a location for plugin libraries (for example, through an environment variable), this design yields more extensibility compared with a setup that forces direct source code modification for feature addition. The latter also leads to a high potential for disorganization and unclear code structure, hurting maintenance. To maintain a framework with plugins is cleaner because it is easy to find the necessary code to modify in order to correct faults or improve performance of a certain feature.

3.2. Dynamic Class Nodes

It is often necessary when running scientific simulations to keep track of large quantities of dynamic objects. In agent-based modeling [12] for example, there could be thousands to millions of agents with different attributes and properties, interacting with their surrounding environment(s) in various ways. Molecular dynamics often enforces the need to record data for large amounts of atoms. There is a similar application to biological cell simulations and tracking individual cells.

Large numbers of data cache misses and page faults can be generated by such applications. In particular, if individual simulation object attributes are noncontigously allocated in virtual memory and are needed consistently throughout a simulation, a high swap rate could be imminent, potentially high enough to lead to thrashing as even recently used data which will be used again soon gets swapped out. Contiguous allocation increases the likelihood of multiple attributes for a specific object to lie in the same memory block, leading to the likelihood of lying in the same page, or even cache block. One way of providing contiguous allocation is through C `structs`, but these are lacking in terms of flexibility. In order to modify, add, or delete attributes, you must access a global `struct` and recompile all dependent source files. Ideally, it would be best to implement attribute declaration in a way similar to plugins through registering, but still allow them to be contiguously allocated to avoid performance degradation due to thrashing and high cache miss rates.

Dynamic class nodes have been designed to provide this functionality. Each attribute is declared as a `DynamicClassNode`, and then registered through a `BasicDynamicClassFactory`. Each individual dynamic class node keeps track of an offset, and so when a simulation object `x` is accessed, if for example attribute `y` has offset `z`, the resulting address of attribute `y` is `<address of object x>+z`, for all simulation objects `x`.

Figure 3 shows UML for an example dynamic class node for some attribute `X`:

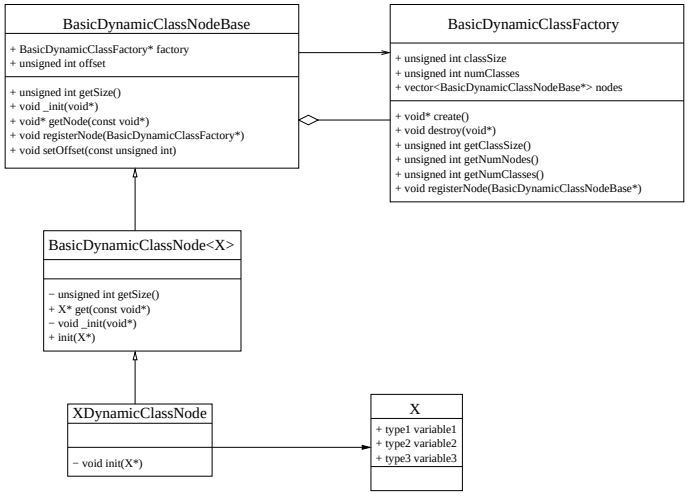

Figure 3. UML diagram of our dynamic class node interface.

3.3. Put Novel MD Patterns Here

4. Example Frameworks

CompuCell and ProtoMol are ongoing projects under the Laboratory for Computational Life Sciences [13] at the Uni-

versity of Notre Dame. Both software packages are freely distributed via SourceForge [7, 16].

4.1. CompuCell

CompuCell [8, 5, 6] implements a known mathematical model for morphogenesis, the Cellular Potts Model (CPM) [10], including a modified version of mathematical partial differential equations proposed by Hentschel *et al*. [11] to establish a surrounding activator chemical gradient. The CPM represents biological cells in a mathematical grid of pixels, giving each unique cell in the simulation an different integer value *index* and each grid pixel one of these indices. If pixel *(X, Y, Z)* possesses index σ, then the cell with index σ encompasses point *(X, Y, Z)*. At each CPM step, a random pixel is selected from the cell grid and a proposal is made to change its index to that of a neighboring pixel. Energy calculations are performed on the current state of the simulation and the new state with the proposed index change. If the energy for the new state is lower the change is made, otherwise the change is made with a certain probability.

To find a neighboring pixel, Offset Neighbor Evaluation is used in CompuCell. To illustrate the savings provided by this algorithm, we compare program A which runs a three-dimensional CPM algorithm that forces each pixel to maintain pointers to all neighbors, versus program B which uses offset evaluation. Both versions were run on a PC running Red Hat Linux 9.0, kernel 2.4.22, with an AMD Athlon XP 1800+ at 1.6 GHZ, and 512 MB of memory. The size of the cell grid was 71x36x211 and cells were initially 8 cubic pixels (2x2x2). To further restrict performance measurement solely to computation we turned off any visualization code, and used the Linux `time` command to measure wall clock time. We also only counted simulation times from after the second step so that the long initialization of all pixel neighbors for program A would not be accounted for. Even without accounting for this, program B ran four times as fast, while consuming one-tenth of the memory of program A.

	A	B	Ratio A/B
Time - 100 CPM Flips	1959 s	501 s	3.91
Memory Usage	70656 KB	6564 KB	10.76

Cell attributes (center of mass coordinates, volume, surface area, and type) are represented by dynamic class nodes, and type is controlled by a cellular automaton that depends directly on exterior chemical gradients. CompuCell also uses plugins to implement optional simulation functionality. Such options include: center of mass calculation, various types of energy calculations in the CPM, a density-dependent growth algorithm, polarity calculation, surface area calculation, and volume calculation. Whether or not these functionalities are included in a simulation rests on the decision of a user to include them in a configuration file input to CompuCell. Cellular automata are implemented as plugins as well, which inherit the interface of `Automaton`. When an automaton is indicated in the configuration file, it is registered and subsequently run at every successful CPM pixel index flip.

4.2. ProtoMol

5. Conclusions

References

[1] Software maintenance - fox pro wiki. `http://fox.wikis.com/wc.dll?Wiki\~{}SoftwareMaintenance\~{}SoftwareEng`.

[2] M. S. Alber, M. A. Kiskowski, J. A. Glazier, and Y. Jiang. On cellular automaton approaches to modeling biological cells. Technical Report 337, University of Notre Dame, Department of Mathematics, Sept. 2002.

[3] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, Reading, Massachusetts, 2001.

[4] C. Blilie. Patterns in scientifi c software; an introduction. *Computing in Science and Engineering*, 4(3):48–53, 2002.

[5] R. Chaturvedi, J. A. Izaguirre, C. Huang, T. Cickovski, P. Virtue, G. Thomas, G. Forgacs, M. Alber, G. Hentschell, S. Newman, and J. A. Glazier. Multi-model simulations of chicken limb morphogenesis. To appear in proceedings of the International Conference on Computational Science ICCS, 2003.

[6] T. Cickovski, C. Huang, R. Chaturvedi, T. Glimm, H. Hentschel, M. Alber, J. A. Glazier, S. A. Newman, and J. A. Izaguirre. A framework for three-dimensional simulation of morphogenesis. *IEEE/ACM Trans. Comp. Biol. and Bioinformatics*, 2004. Submitted.

[7] COMPUCELL. CompuCell: A framework for three-dimensional simulation of morphogenesis. `http://sourceforge.net/projects/compucell/`, Dec. 2003.

[8] COMPUCELL. `http://www.nd.edu/~lcls/compucell`, 2004. CompuCell home page.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[10] F. Graner and J. A. Glazier. Simulation of biological cell sorting using a two-dimensional extended potts model. *Phys. Rev. Lett.*, 69:2013–2016, 1992.

[11] H. G. E. Hentschel, T. Glimm, J. A. Glazier, and S. A. Newman. Dynamical mechanisms for skeletal pattern formation in the vertebrate limb. *Biological Sciences*, 271(1549):1713–1722, 2004.

[12] Y. Huang, X. Xiang, G. Madey, and S. Cabaniss. Agent-based scientifi c simulation using java/swarm, j2ee, rdbms and automatic management techniques. Preprint., 2004.

[13] J. A. Izaguirre. Laboratory for Computational Life Sciences. `http://www.nd.edu/~lcls`, July 2004.

[14] T. Matthey and J. A. Izaguirre. ProtoMol: A molecular dynamics framework with incremental parallelization. In *Proc. of the Tenth SIAM Conf. on Parallel Processing for Scientifi c Computing (PP01)*, Proceedings in Applied Mathematics, Philadelphia, Mar. 2001. Society for Industrial and Applied Mathematics.

[15] C. Picioreanu, M. C. M. van Loosdrecht, and J. J. Heijnen. A new combined differential-discrete cellular automaton approach for biofi lm modeling; application for growth in gel beads. *Biotechnl. Bioeng.*, 57:718–731, 1998.

[16] PROTOMOL. PROTOMOL: An object oriented framework for molecular dynamics. `http://sourceforge.net/projects/protomol/`, July 2004.

[17] A. Shalloway and J. R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, Boston, MA, 2002.

[18] J. P. Tremblay and G. A. Cheston. *Data Structures and Software Development in an Object-Oriented Domain*. Prentice Hall, Upper Saddle River, NJ, 2001.