

Empirical Evaluation of Design Patterns in Scientific Application

Kedar Aras
CSE Department
University of Notre Dame
325 Cushing Hall
Notre Dame, IN 46556
karas@nd.edu

Trevor Cickovski
CSE Department
University of Notre Dame
325 Cushing Hall
Notre Dame, IN 46556
tcickovs@nd.edu

Jesus A. Izaguirre
CSE Department
University of Notre Dame
325 Cushing Hall
Notre Dame, IN 46556
izaguirr@cse.nd.edu

ABSTRACT

Design patterns have been widely adopted for building flexible and extensible applications. This can come at a cost of reduced performance which may not be acceptable for computationally intensive scientific applications. We claim that there are certain design patterns that when used properly can enhance both system performance and flexibility and thus greatly benefit scientific software. Software engineers can use these insights to design flexible systems that also deliver on performance, potentially saving days of runtime on long simulations as well as reducing memory consumption significantly.

We investigate the effects of some of the design patterns on performance of scientific applications through a detailed measurement and profiling of CompuCell3D, which is a software framework for three-dimensional (3D) modeling of morphogenesis, a stage in embryonic development where cells cluster into tissues and organs. By examining CompuCell3D subsystems with and without design patterns, we evaluate their impact on application performance and maintainability.

We find that as the application is continually refactored to support additional functionality, not using design patterns significantly degrades application performance. We also present a scientific design pattern Dynamic Class Node (DCN), discovered during our experimentation. The pattern showed that contiguous allocation of object attributes offers significant benefits in performance by reducing page faults and cache misses, while still maintaining flexibility.

Categories and Subject Descriptors

C.4 [Performance of Systems]: *design studies, performance attributes.*

General Terms

Performance, Measurement, Design, Experimentation

Keywords

CompuCell3D, design patterns, performance, dynamic class node, strategy, factory, singleton.

1. INTRODUCTION

Design patterns have been widely adopted for building flexible and extensible applications. While they have been explored for many years in software engineering, their application to scientific software is just beginning to unfold. Until recently, the bulk of scientific software was written either in FORTRAN or C due to computational overhead of object-oriented languages and heavy emphasis on fast mathematical algorithms in scientific computing [1].

This paper investigates the effects of various design patterns on speed, memory and maintainability of CompuCell3D [2], [3], by redesigning subsystems with and without the use of design patterns. We implement seven different versions of CompuCell3D. We have posted all software, configuration files, and full experimental reports on our web site <http://www.nd.edu/~lcls/CompuCell3D> to allow others to reproduce the results.

We show that as applications get refactored to support additional functionality, not using design patterns can result in performance degradation. We also present a scientific design pattern Dynamic Class Node (DCN) which allocates object attributes contiguously. We find improvements in speed and memory consumption for DCNs over a noncontiguous allocator alternative while retaining maintainability over C structs.

We provide some background on CompuCell3D in Section 2. Section 3 provides detailed description of the alternative designs and design pattern combinations. Section 4 presents our experimental environment, including hardware and software. Section 5 discusses our experimental results. Related work is presented in Section 6, and finally Section 7 provides a discussion of results and future work.

2. BACKGROUND

CompuCell3D is a software framework for three-dimensional simulation of morphogenesis using a combination of the Cellular Potts Model (CPM) [4] and reaction-diffusion equation solvers [5], [6]. It is an extension of a two-dimensional framework called CompuCell [2]. The CPM uses cells as the lowest unit of modeling [7], representing them as groups of pixels in a three-dimensional lattice. Each lattice pixel contains an index representing a cell number (or '0' for the surrounding medium). At each step of the CPM, a random lattice location is selected and a proposal is made to change (or *flip*) its index to that of a neighboring pixel. A hypothetical effective energy change is calculated if this flip were to occur, and is accepted with a

probability that tends the system towards a state of lower energy. The effective energy contains multiple terms which drive cells towards certain behaviors which are obtained through biological experimentation [8].

CompuCell3D is comprised of subsystems as shown in Figure 1.

- Basic Utilities provides basic data structures and utility functions.
- Automaton encapsulates different cell types. This provides a method of categorizing cells by behavior.
- Field3D implements the mathematical grid for representing cells and the surrounding medium.
- Potts3D implements the Cellular Potts Model (CPM)
- Plugins manage access to a dynamically loaded set of features whose inclusion is optional for particular simulations and thus do not implement core functionality.

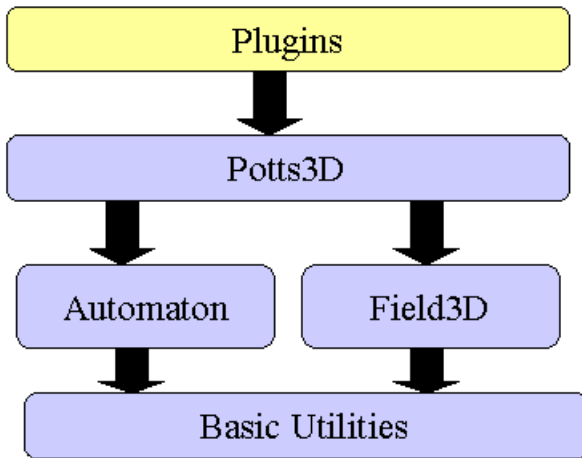


Fig 1. CompuCell3D Architecture

3. DESIGN ALTERNATIVES

We modified core subsystems of CompuCell3D and created different versions, some of which use design patterns and others do not. The versions that do not use design patterns were implemented with simple and easy to understand design choices, reflecting something a programmer could implement without being familiar with design patterns. Accordingly, we designed seven versions of the application: three without design patterns and four with design patterns. We now describe the designs.

3.1 Reference Field3D Implementation Without Design Patterns (A)

Field3D is responsible for the mathematical grid that simulates the environment, including cells and the external medium. This reference implementation uses a No Flux Boundary algorithm that allows the grid to simulate a finite boundary for cells. Figure 2 shows the class diagram for the implementation.

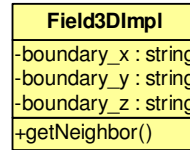


Fig 2. Reference Implementation

The getNeighbor() function is responsible for selecting valid neighbors for a pixel and also implements the No Flux Boundary algorithm. This algorithm determines the selected neighbor's position and if it lies outside the boundary, discards it and reselects.

3.2 Refactored Field3D With Design Patterns (B)

Reference implementation A, although simple, tightly couples the grid with boundary conditions. The design motivation here is to decouple them, allowing the grid to incorporate different types of boundary conditions. We use a combination of Factory, Strategy and Singleton [9] patterns to achieve this goal. The design is shown in Figure 3.

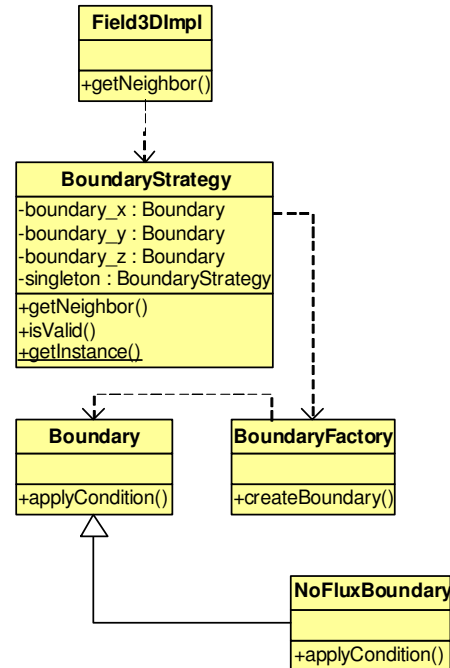


Fig 3. Field3D refactored with design patterns

NoFluxBoundary is implemented as a strategy and instantiated using a factory. Field3DImpl delegates the responsibility of selecting valid neighbors to BoundaryStrategy, which is implemented as a singleton. This ensures that only one instance of the boundary strategy is created and is globally accessible.

3.3 Extend Field3D Without Design Patterns (C)

The reference implementation of Field3D (A) is extended here to support additional boundary algorithms without using design patterns. In this case, the implementation now supports a Periodic Boundary that implements wraparound. Meaning, if a neighbor is

found outside of the grid boundaries, it is wrapped around to the other side of the grid. The class diagram is as depicted in Figure 4.

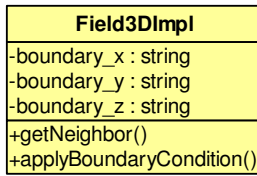


Fig 4. Reference implementation extended without design patterns.

When a neighbor is selected, applyBoundaryCondition() is invoked to validate it. The boundary conditions are implemented as part of the function. It uses a boundary flag (boundary_x, boundary_y, boundary_z) to determine the appropriate algorithm to apply using a conditional clause.

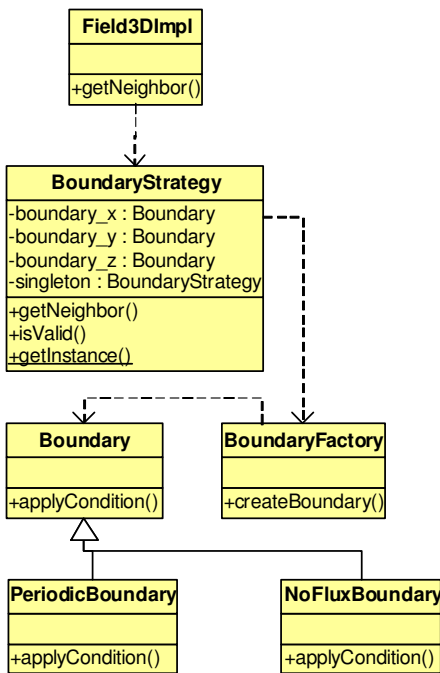


Fig 5. Field3D extended with design patterns

3.4 Extend Field3D With Design Patterns

(D)

The Field3D design B is extended here to support Periodic Boundary algorithm. The algorithm is implemented as a strategy similar to the No Flux algorithm. This diagram is thus identical to B except for one more class PeriodicBoundary which like NoFluxBoundary inherits from Boundary and overrides applyCondition(). The design is as shown in Figure 5.

3.5 Dynamic Class Nodes

3.5.1 Intent

Ensure contiguous allocation of individual sets of attributes to avoid page faults and cache misses.

3.5.2 Motivation

In a given scientific simulation, there can be thousands of simulated objects each containing multiple attributes (e.g. in the case of CompuCell3D, cells contain volume, surface area, type, etc.). Per simulation step, there can be millions of accesses made to these objects, potentially requiring access to these attributes each time. The quantity and unpredictable sizes of these attributes leads to a potential for a high frequency of cache misses, or page faults, the latter of which can lead to thrashing. These can be avoided by building a pattern that takes advantage of likely temporal locality of an object's attributes. The pattern should allocate all attributes contiguously for each object, imposing spatial locality on attribute sets. As a result, when an attribute is brought into the page table or cache, all attributes for that object could be brought in as well, avoiding the need to reaccess main memory.

Although FORTRAN [10] or C structs would provide contiguous allocation, they are lacking in flexibility. A struct could be used to encapsulate cell attributes, but it would need to be global for all objects, and would entail modification for each simulation that requires attribute changes. This motivated creation of the Dynamic Class Node (DCN) to represent object attributes which can be associated with a dynamically loaded plugin library in the case of CompuCell3D [11]. These are added or removed from a simulation through a reference in an input configuration file. This process is much easier than modifying a global struct and recompiling the framework at every attribute change. By using DCNs to represent cell attributes in CompuCell3D, we can thus maintain a flexible framework while also allocating attributes contiguously.

3.5.3 Applicability

Use the DCN pattern when:

- you have multiple simulation objects that each have the same set of attributes (although values can differ), and
- the data types of attributes are primitive

3.5.4 Structure

UML for the DCN pattern is shown in Figure 6. When an object attribute is specified, its data type is used to template the BasicDynamicClassNode interface.

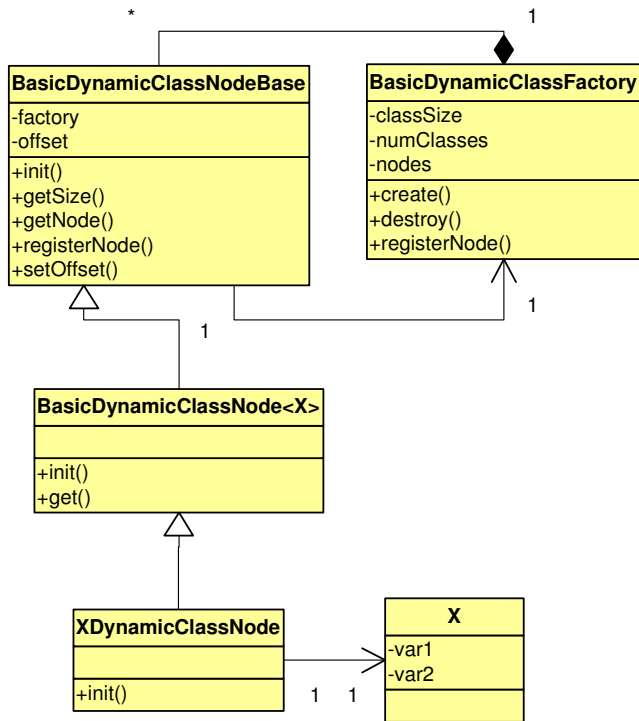


Figure 6. UML for the Dynamic Class Node design pattern.

3.5.5 Participants

- **BasicDynamicClassNodeBase**
 - The highest interface in a two-level hierarchy to set up an individual DCN.
 - Contains an address offset, which is added to the address of simulation object pointers to obtain the attribute.
 - Contains accessor methods for the size of the DCN in bytes, its offset, and its pointer, and also a method to register the DCN with a **BasicDynamicClassFactory** which will create objects with the appropriate DCN attributes.
- **BasicDynamicClassNode<X>**
 - Middle level in the DCN hierarchy. Simple interface.
 - Templated with the data type of the DCN.
 - Overrides methods in **BasicDynamicClassNodeBase** for objects of the given type.
- **XDynamicClassNode**
 - Lowest level in the DCN hierarchy.
 - DCN for an object of data type **X**.
 - Overrides the `init()` method, to initialize an **X** object appropriately (for example, **X** may be a struct containing three data members; each of these must be initialized). This method is implicitly called by the pattern, not the constructor, and thus objects with constructors cannot be used as DCNs.

- **X**
 - A DCN attribute data type. Encapsulates whatever data members are necessary for an attribute (for example, volume could contain one integer; center of mass could contain three integer coordinates, etc.)
- **BasicDynamicClassFactory**
 - Uses the Factory [12] pattern to create and destroy simulation objects, where each object is allocated an amount of memory amounting to the sum of all registered DCNs.
 - Contains a method to register a DCN.
 - Contains data members for the total simulation object size (in bytes) and also the number of registered DCNs, as well as accessor methods.

3.5.6 Collaborations

- **BasicDynamicClassNodeBase** relies on its subclasses to define the `init()` method, so that a DCN of the appropriate type is created and initialized properly.
- **XDynamicClassNode** relies on **X** to only encapsulate data members with types that do not require constructor invocation.
- **BasicDynamicClassNodeBase** depends on **BasicDynamicClassFactory** to register attributes properly for each Cell object in a simulation.

3.5.7 Consequences

When an attribute is declared as a DCN, it is subsequently registered with a Dynamic Class Node factory, which uses the Factory design pattern. At the point of registration, this attribute becomes associated with every object. Upon registration, the attribute gets an offset which is the sum in bytes of all DCNs registered thus far. All objects are represented as pointers, and to access attribute *y* of object *x*, assuming *y* has offset *z*, the location in memory is defined as `<address of object x>+z`. Figure 7 shows an example with three DCNs associated with a simulated cell in **CompuCell3D** with sizes of 12, 8, and 12 bytes respectively, assuming a block size of 4 bytes.

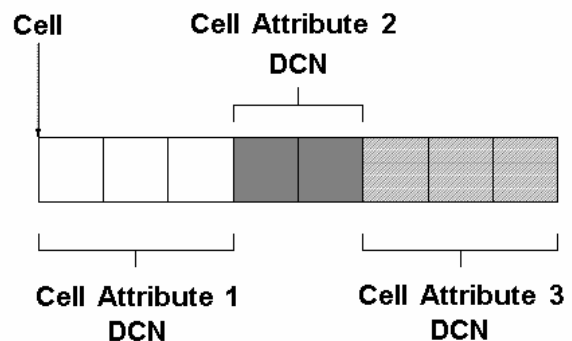


Figure 7. Allocation of a Cell with three DCNs of sizes 12, 8, and 12 bytes respectively, assuming a block size of 4 bytes. From [3].

3.5.8 Implementation

We now give an example of the DCN usage in CompuCell3D. To create a new Cell attribute using the DCN pattern, the first requirement is to encapsulate all data members for the attribute within a structure. For example, to represent cell center of mass (COM), we would need to allocate 12 bytes for each cell, one for each integer coordinate x, y and z, as the CPM operates on a lattice with integer coordinate locations. Since all data member types are primitive we can encapsulate this with the following:

```
class COM {
public:
    int x;
    int y;
    int z;
};
```

The next step is to template a dynamic class node with the COM data type, and override the init() method. For now we will initialize each coordinate to zero.

```
class COMDynamicClassNode :
public BasicDynamicClassNode<COM> {
    virtual void init(COM* com) {
        com->x = com->y = com->z = 0;
    }
};
```

Next, it is necessary to create a factory for cells, and register this new DCN. At this point, any cells created by this factory will have 12 bytes allocated for its center of mass, plus memory for any other registered DCNs.

```
BasicDynamicClassFactory<Cell*>cellFactory;
COMDynamicClassNode comDCN;
comDCN.registerNode(&cellFactory);
```

Finally, we can create a cell by invoking the create() method of the factory, and access the center of mass DCN by invoking its get() method. The center of mass DCN knows its own offset from the passed Cell pointer, and its value can be accessed or changed accordingly. For example, the following code snippet will create a cell and set the z-coordinate of its center of mass to 3:

```
Cell* myCell = (Cell*)cellFactory.create();
comDCN.get(myCell)->x = 3;
```

3.5.9 Availability

The DCN libraries are available within the CompuCell3D package and also the BasicUtils package [13].

3.6 Class Groups

We compare DCNs to a noncontiguous attribute allocator, the ClassGroup [13]. ClassGroups offer the tradeoff of allowing attributes to invoke constructors or to be of a virtual class (neither of which DCNs allow), but sacrificing contiguous allocation. The class diagram is depicted in figure 8.

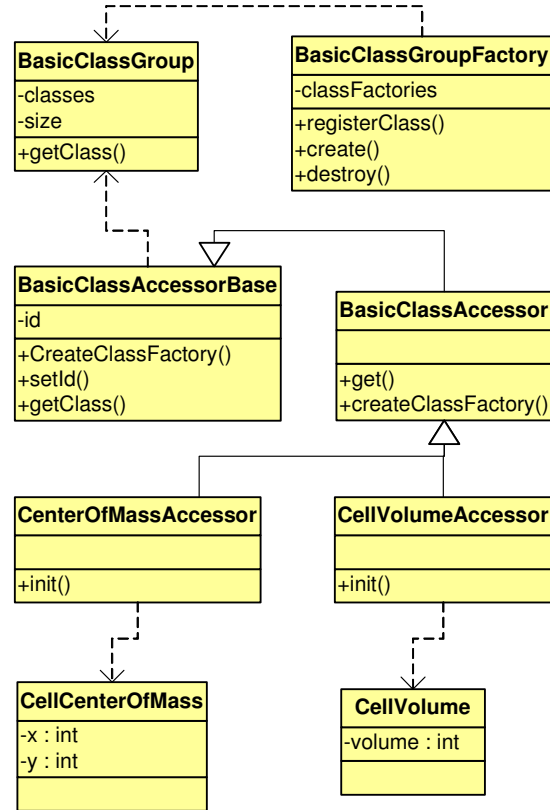


Figure 8. Using ClassGroups

A BasicClassGroup keeps an array of pointers to its classes and the size of this array. A group is allocated by a BasicClassGroupFactory in the create() method, and deallocated by the destroy() method. Classes for created BasicClassGroups are registered by invoking registerClass(). For classes with constructors, corresponding factories are kept in the array classFactories.

An additional BasicClassAccessor interface is provided for getting and setting classes. Each class has its own ID and factory (which can be overridden if a constructor must be invoked). Accessors are then customized for attributes like CenterOfMass and Volume as shown in the above UML.

3.7 C Structs

Another design option that we compare DCNs with is the simple C struct representation of cell attributes. All cell attributes are encapsulated within this struct, as shown in Figure 9.

| Cell |
|---------------|
| -volume : int |
| -x : int |
| -y : int |
| -z : int |

Figure 9. Using C Structs

4. EXPERIMENTAL ENVIRONMENT

4.1.1 Hardware

All tests were run on an HP Workstation xw4100 with an Intel Pentium 4 3.2 GHZ processor and 1 GB of memory.

4.1.2 Software

The HP Workstation ran Red Hat Linux 9.0, kernel 2.4.21. The C++ compiler used was g++, version 3.2.3. We used the flags: Wall, Werror, O2, c, o and -DHAVE_CONFIG_H.

5. EXPERIMENTAL RESULTS

To check speed we measure wall clock execution time of CompuCell3D, and for memory management we use the Linux 'top' command. For maintainability, we review steps for projected common extensions to CompuCell3D. We also perform impact analysis of incremental changes [14] that result from these extensions.

5.1 Field3D Designs

We compare the results for the four implementations of the Field3D subsystem. For each implementation, we evaluate three different configurations:

- 1000 step simulation with 2 flips per pixel (fpp), per step. With the grid dimension set to 71 x 36 x 211, we execute $2 \times 71 \times 36 \times 211 = 1078632$ iterations per step.
- 1000 step simulation with 4 flips per pixel, per step. Or, 2157264 iterations per step
- 2000 step simulation with 2 flips per pixel, per step. Like the first, there are 1078632 iterations per step, but the simulation is just run twice as long.

We selected these metrics for variation because Boundary algorithms are applied when neighbors are searched for in CompuCell3D, which is necessary at every flip attempt. Therefore, by increasing the flips per pixel or increasing the simulation length, we will increase the total number of flip attempts in the simulation as a whole. We execute 3 runs for each configuration to yield a total of 9 simulations for each implementation.

- Design A is the reference Field3D implementation (without design patterns).

- Design B is the refactored Field3D with design patterns.
- Design C is the extended Field3D without design patterns.
- Design D is the extended Field3D with design patterns.

5.1.1 Performance

Figures 10, 11, and 12 present the simulation results. In all graphs:

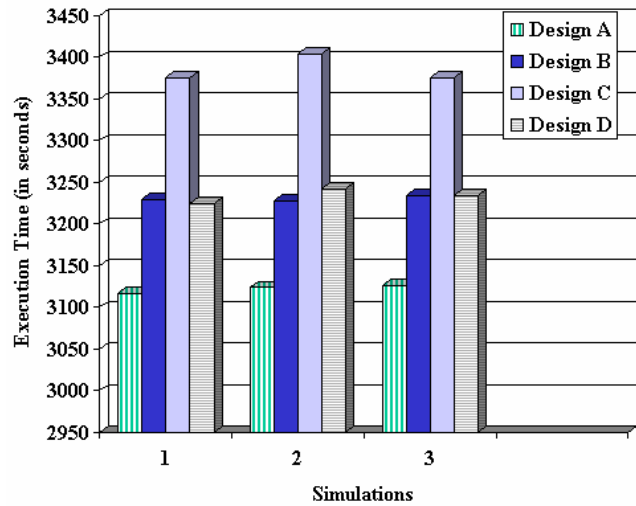


Figure 10. Execution times (in seconds) for each Field3D design implementation for the configuration: 1000 steps with 2 flips per pixel per step.

The reference implementation of Field3D for the No Flux Boundary algorithm (A) is faster than the implementation that uses design patterns (B). This is in line with our expectations, as the use of design patterns creates an overhead due to additional classes being involved.

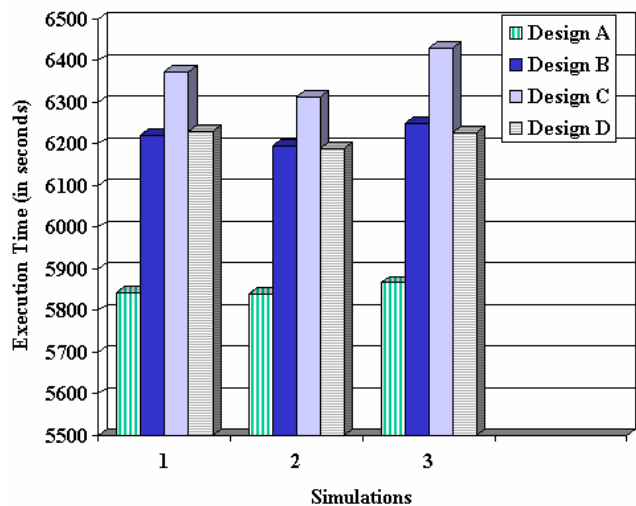


Figure 11. Execution times (in seconds) for each Field3D design implementation for the configuration: 1000 steps with 4 flips per pixel per step.

However, as Field3D is extended to support additional Boundary algorithms, the performance of Field3D not using design pattern (C) degrades substantially, whereas that of Field3D extended using design patterns (D) is comparable to B. As one provides support for additional algorithms, the Field3D implementation not using design patterns must now compensate by creating and checking flags for different algorithms. This additional conditional clause in the code causes a performance penalty.

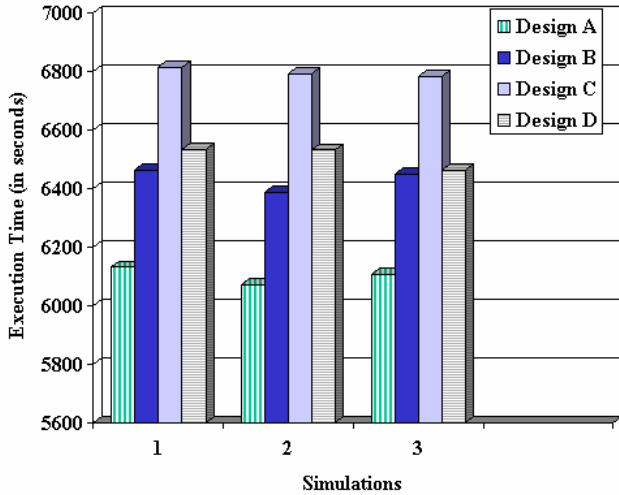


Figure 12. Execution times (in seconds) for each Field3D design implementation for the configuration: 2000 steps with 2 flips per pixel per step.

The Field3D implementations with design patterns do not suffer a huge performance penalty because they get around some additional code by instantiating the appropriate algorithm at run time, which is then referenced throughout the simulation. This explains their performance gain over C.

This behavior is consistently exhibited across different configuration sets, which provides a situation where not using design patterns can in fact degrade the performance over the life of the application, particularly as it gets extended to support new functionality.

5.1.2 Memory Consumption

Figure 13 presents memory consumption for each implementation. The configuration used is 1000 steps with 2 flips per pixel, per step. We can see that memory consumption is relatively the same whether or not you choose to use the design patterns. Adding support for additional algorithms will pose a slightly higher memory requirement (as shown by the consumption of C and D versus A and B).

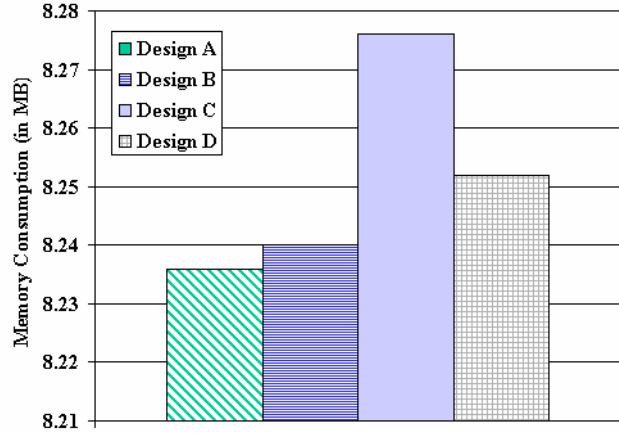


Figure 13. Memory Footprint (in MB) for each Field3D design implementation for the configuration: 1000 steps with 2 flips per pixel per step.

5.1.3 Maintainability

The loose coupling of grid implementation and boundary algorithms in Field3D with patterns offers a high level of flexibility compared to designs without patterns. This is strongly demonstrated when extending Field3D without design patterns to support additional boundary algorithms.

Using the strategy pattern allows implementation of new boundaries without affecting the grid implementation. This is in contrast to when the pattern was not used, and the grid implementation had to be refactored to support additional boundaries, making the implementation error prone and progressively difficult to keep up with additional changes.

5.2 Dynamic Class Nodes

We now compare the DCN with ClassGroups [13] and C structs. To ensure no performance penalty due to constructor invocation or forcing a class to create its own factory in ClassGroups, we use only primitive type attributes and do not use any virtual classes.

5.2.1 Performance

To evaluate performance, we ran three simulations on the afore mentioned platforms: (1) a simulation with 2593 cells for 1000 steps, (2) a simulation with 2593 cells for 2000 steps, and (3) a simulation with 20449 cells for 1000 steps. In this way we can view the effects of the DCN pattern as simulation length and cell count vary. Results are given in Figures 14, 15 and 16.

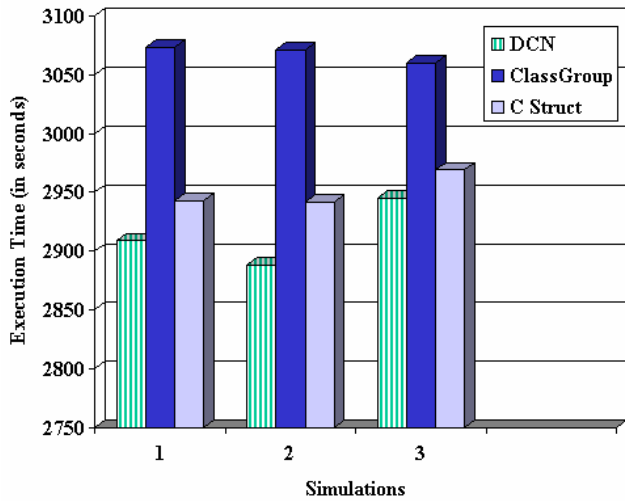


Figure 14. Execution times (in seconds) for DCN, ClassGroup and C Struct implementations for the configuration: 1000 steps with 2 flips per pixel per step for 2593 cells.

We can see that particularly for larger amounts of cells and longer simulations that the DCN consistently performs better than the ClassGroup, by avoiding performance penalties that result from page faults and cache misses.

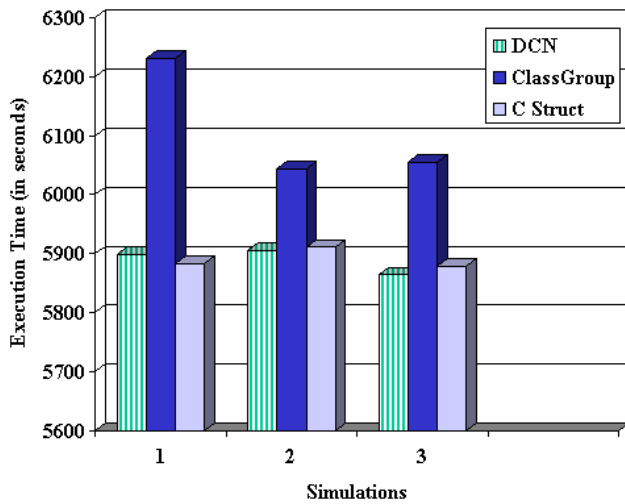


Figure 15. Execution times (in seconds) for DCN, ClassGroup and C Struct implementations for the configuration: 2000 steps with 2 flips per pixel per step for 2593 cells.

In particular, for very long simulations this savings becomes highly significant. C structs also offer significant savings over ClassGroups, and yield comparable results to DCNs although seem to save more as more cells are created, since the small savings resulting from attribute access directly through the struct versus a method invocation in the DCN becomes more significant.

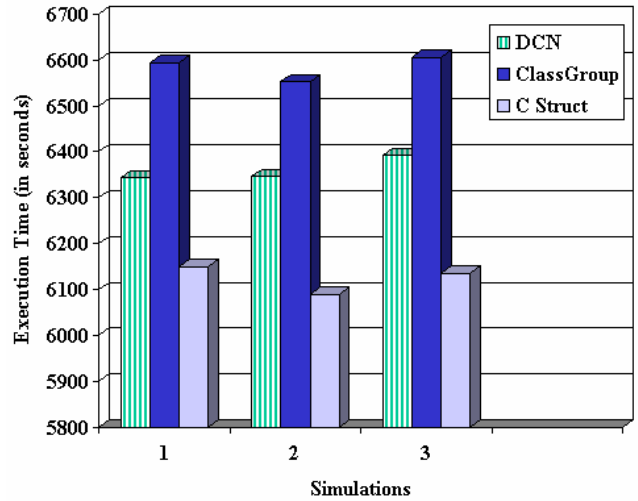


Figure 16. Execution times (in seconds) for DCN, ClassGroup and C Struct implementations for the configuration: 1000 steps with 2 flips per pixel per step for 20449 cells.

5.2.2 Memory Consumption

The two frameworks were run on the same afore mentioned platforms, and we run three example simulations: (1) a simulation with 2593 cells, (2) a simulation with 8587 cells, and (3) a simulation with 20,449 cells. In this way we view the effects of the DCN pattern as the number of simulated cells (and thus number of attribute sets to allocate) increases. We keep the step count consistent at 1000 across simulations. The results for memory consumption are summed up in figure 17.

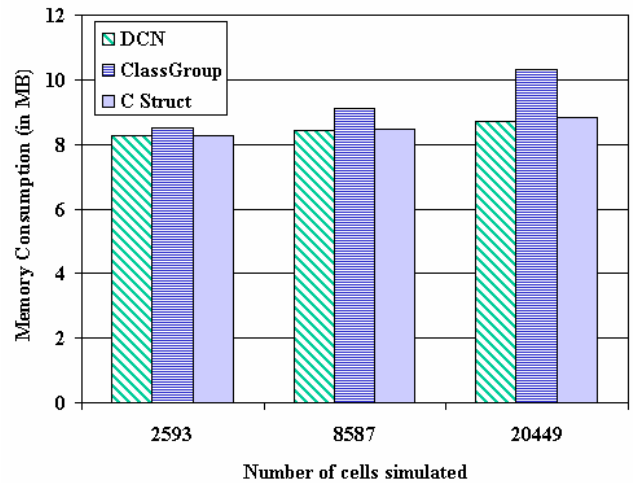


Figure 17. Memory footprint for DCN, ClassGroup and C Struct implementations.

We see that for smaller amounts of cells the DCN saves minimally over ClassGroups, but as the number of simulated cells grows, the difference becomes more significant, up to about a 16% savings for a simulation with 20,449 cells. ClassGroups consume more memory because a list of pointers to classes is kept in each BasicClassGroup. We can see that memory consumption between

DCNs and C Structs (which also perform contiguous allocation) is about the same.

5.2.3 Maintainability

To illustrate the advantages in terms of flexibility for DCNs over C structs, we show the requirements for performing four key tasks regarding cell attributes. The first is adding a new cell attribute to the framework (meaning, it is not already a part of CompuCell3D nor has it ever been). The second is removing a cell attribute from the framework. The third is adding an attribute that is a part of the framework to a particular simulation, and the fourth is removing an attribute that is a part of the framework from a simulation. Flexibility advantages are particularly strong in these latter two cases, because in terms of C structs, there is no distinction between them and the first two (i.e. to remove or add an attribute from a simulation requires removing or adding it to the framework as a whole). For DCNs, when they can work with dynamically loaded CompuCell3D plugin libraries, this is not the case as we show in Table 1. We show in parentheses the number of class dependencies that are affected by the necessary changes to the framework.

| <i>Extension</i> | <i>DCN</i> | <i>C STRUCT</i> |
|----------------------------------|---|--|
| Add attribute to framework | 1. Interface to a plugin (3) 2. Compile just new plugin libraries | 1. Modify Cell struct (33) 2. Recompile framework |
| Remove attribute from framework | 1. Remove registration invocation from plugin (3) 2. Recompile just plugin libraries | 1. Modify Cell struct (33) 2. Recompile framework |
| Add attribute to simulation | 1. Add plugin reference to configuration file (0) | 1. Modify Cell struct (33) 2. Recompile framework |
| Remove attribute from simulation | 2. Remove plugin reference from configuration file (0) | 1. Modify Cell struct (33) 2. Recompile framework |

Table 1. Extending CompuCell3D using DCN and C Struct

As can be seen, the latter two extensions are made much simpler by DCNs. If an attribute is already in the framework, its associated plugin can be added or removed from the configuration file, and because plugins are dynamically loaded and instantiated [3], it saves the need to recompile the framework. Modifying the C struct would require recompilation of all interfaces which depend on the Cell, which is a large percentage of the CompuCell3D framework.

Although plugin code is more complex than a C struct, compilation would be easier when adding a new attribute to the framework since it would only be necessary to compile a small set of plugin libraries as opposed to a large percentage of the framework. Removing an attribute completely is much easier than adding one, as the DynamicClassNode class which encompasses the attribute along with its registration call can be removed. Again, the most significant savings in maintainability come in once attributes are a part of the framework and need to be added or removed from particular simulations.

These extensions can be viewed as incremental changes to the framework [14]. Doing an impact analysis of the change in Cell struct, we see that 33 classes in the CompuCell3D framework depend on Cell, and so would be affected by any of the four extensions above. Contrasting, the four extensions above affect 3 (plugin class, DCN class and Attribute class), 3 (same three), 0, and 0 classes. There are far fewer dependencies when extending the framework using DCNs versus C structs.

Comparing DCNs to ClassGroups, ClassGroups would receive a slight edge in flexibility since they can encompass attributes that require constructor invocations. This is a direct tradeoff with the advantages in terms of speed and memory consumption of DCNs versus ClassGroups.

6. RELATED WORK

Prechelt et al [15] conducted a controlled experiment to study whether deploying design patterns could facilitate maintenance. In the majority of the cases, positive results were obtained confirming the usefulness of design patterns. However, in certain cases, negative results were found in a few cases where the design solutions without deploying patterns were less error prone or subject to shorter maintenance time. Bieman et al [16] conducted an industrial case study to explore the relationships between design patterns, design structures and program changes. No concrete evidence was found to suggest that design patterns ease adaptability. However, the two studies do not evaluate the performance impact of using design patterns. Billard [17] examined the execution performance of design patterns implemented in different languages as a pure object structure independent of the application. The patterns demonstrate a wide variety of performance between languages. However, the patterns' performance over simpler solutions in an application setting is not covered.

A different combination of known object-oriented design patterns is shown in Modelica [18], [19], which similarly uses a combination of known design patterns to build a useful tool for modeling multiple domains in mathematics and engineering. Modelica uses a combination of the Adapter, Strategy and Factory patterns [9], the latter two of which we also use in CompuCell3D. They use the Strategy pattern to encapsulate time-dependent signal calculation algorithms; similar to the way we use it to encapsulate boundary condition point validity determination algorithms. They also use the Factory method to create different parameter sets and equations, similar to our use in creating different boundary strategies and DCNs.

In addition, we have witnessed methodologies in software development which address similar issues as DCNs. For example, Chilimbi et al. [20] provide two software techniques called clustering and coloring which improve temporal and spatial locality of data structure elements. The former places elements likely to be accessed near the same time within a cache block, the latter separates elements by frequency into different blocks. Their techniques change memory (and cache) locations of these elements without affecting program semantics, and they even build a technique cmalloc which attempts to allocate a new element close to an existing element, which is exactly what DCNs do with cell attributes. DCNs do not take into account temporal locality, but have more flexibility provided by the factory which allows attributes to be easily added/removed from simulations. Beyls [21] provides a summary of various software methods to improve data locality and cache behavior.

7. CONCLUSIONS

We have provided some examples of design patterns and demonstrated situations where they helped performance along with providing inherent flexibility. We have also presented a DCN design pattern that provides substantial improvements in speed and memory consumption, while retaining maintainability benefits.

In the future we would like to explore further combinations of design patterns and their potential for similar benefits. We also look forward to the future uses of these patterns, both more extensively in CompuCell3D and in other frameworks. We would also like to catalog the performance of other design patterns. We feel this would be a good addition to the field of computational science as a whole by providing a solid tool for software developers in the domain.

8. ACKNOWLEDGEMENTS

This research was partially funded by NSF grants IBN-0083653, IBN-0313730, and an Arthur J. Schmitt Fellowship.

Special thanks to Joseph Coffland for his contributions to the DCN design pattern.

9. REFERENCES

- [1] Blilie, C. Patterns in scientific software; an introduction. *Computing in Science and Engineering*, 4(3):48-53, 2002.
- [2] Izaguirre, J.A, Chaturvedi, R, Huang, C, Cickovski, T, Coffland, J, Thomas, G, Forgacs, G, Alber, M, Newman, S.A, Glazier, J.A. CompuCell: A multi-model framework for simulations of morphogenesis. *Bioinformatics*, 20(7):1129-1137, 2004.
- [3] Cickovski, T, Huang, C, Chaturvedi, R, Glimm, T, Hentschel, G, Alber, M, Glazier, J.A, Newman, S.A, Izaguirre, J.A. A framework for three-dimensional simulation of morphogenesis. *ACM Transactions on Computational Biology and Biocomplexity*, under revision, 2005.
- [4] Graner, F. and Glazier, J.A. Simulation of biological cell sorting using a two-dimensional extended Potts model. *Phys. Rev. Lett.*, 69:20-13-2016, 1992.
- [5] Hentschel, H.G.E, Glimm, T, Glazier, J.A, Newman, S. A. Dynamical mechanisms for skeletal pattern formation in the vertebrate limb. *Proc R Soc Lond B Biol Sci*, 271(1549):1713-1722, 2004.
- [6] Panfilov, A.V. and Pertsov, e.A.M. Vortex ring in a three-dimensional active medium described by reaction-diffusion equations. *Dokl. Akad. Nauk*, 274:1500-1503, 1984.
- [7] Merks, R.M.H. and Glazier, J.A. A cell-centered approach to developmental biology. *Phys. A*, In press.
- [8] Foty, R.A, Pflieger, C. M, Forgacs, G, Steinberg, M.S. Surface tensions of embryonic tissues predict their mutual envelopment behavior. *Development*, 122:1611-1620, 1996.
- [9] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns – Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading MA, 1995.
- [10] UNFP. User notes on FORTRAN programming (unfp): An open cooperative practical guide. <http://www.ibiblio.org/pub/languages/fortran/unfp.html>.
- [11] Cickovski, T, Matthey, T, Izaguirre, J.A. *Design Patterns For Generic Object-Oriented Software*, Notre Dame Technical Report 2004-29, Nov. 2004.
- [12] Alexandrescu, A, *Modern C++ Design: Programming and Design Patterns Applied*. Addison-Wesley, Reading, Massachusetts, 2001.
- [13] Coffland, J, BasicUtils, <http://compuCell.sourceforge.net/phpwiki/index.php/BasicUtils>, December 2003.
- [14] Rajlich, V, and Gosavi, P. Incremental Change in Object-Oriented Programming, *IEEE Software* 21(4):62-69, 2004.
- [15] Prechelt, L, and Tichy, W. F. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*. 27(12):1134-1144, 2001.
- [16] Bieman, M. J, Jain, D, Yang, H. J. OO Design Patterns, Design Structure, and Program Changes: And Industrial Case Study. *Proceedings International Conference on Software Maintenance (ICSM 2001)*, Florence, November, 2001, pp. 580-589.
- [17] Billard, A. E. Language-Dependent performance of design patterns. *ACM SIGSOFT Software Engineering Notes*. 28(3):3, May 2003.
- [18] Claub, C, Leitner, T, Schnider, A, Schwarz, P. Object-oriented modeling of physical systems with Modelica using design patterns. http://www.eas.iis.fhg.de/publications/papers/2000/006/index_de.html, *Workshop on System Design Automation*, March 13-14, 2000.
- [19] Modelica. Modelica website. <http://www.modelica.org>.
- [20] Chilimbi, T. M, Hill, M. D, Larus, J. R, Cache-conscious structure layout. In *Proceedings of ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1-12, May 1999.
- [21] Beyls, K. Software Methods to Improve Data Locality and Cache Misses. <http://citeseer.ist.psu.edu/beyls04software.html>.