Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life

Michael R. Wick
Computer Science Department
University of Wisconsin-Eau Claire
Eau Claire, WI 54701
wickmr@uwec.edu

ABSTRACT

Design patterns are an important element of today's undergraduate curricula. However, their inherent complexities often make them difficult for entry-level students to even partially grasp. In this paper, we describe the latest in our continuing efforts to build educational materials appropriate for infusing design patterns in entry-level computer science courses.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education*.

General Terms

Design.

Keywords

Design Patterns, Game of Life, CS1, Laboratory.

1 INTRODUCTION

Design patterns [1] have emerged over the last decade as a necessary component of a software educator's arsenal of design and implementation techniques (for example, [2]). Some authors argue that the use of design patterns can create designs that are far more complicated than necessary for entry-level computer science applications (for example, [3]). The key point is that the use of design patterns can add complexity that is called for. However, we have found that with the proper simplification and customization, many popular design patterns can be presented to entry-level computer science students in such a way that the resulting design is understandable and that the presence of the design pattern has real and significant advantages for the system. While we are the first to admit that the resulting designs are typically more complicated than those that entry-level students would develop on their own, we believe that it isn't the complication that students object to; it is the fruitless complication of using design patterns without significant and real value-added.

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.

Over the last few years, we have worked on developing a collection of exercises, lectures, and laboratories designed to introduce entry-level students to the power and elegance of design patterns (for example, [4] [5]). This paper presents the outcomes of our latest work in this area - a series of closed laboratory assignments designed to introduce students to the power and elegance of design patterns through their application to the classic Game of Life [6] program.

2 GUIDING PRINCIPLES

Before diving into the details of the Game of Life laboratory sequence, it is worth pausing for a moment to reflect on the guiding principles that steer us in our work to infuse design patterns in the entry-level computer science coursework. These principles fall into two categories for the purpose of this paper – General Principles and Design Pattern Principles.

2.1 General Assignment Selection Principles

Use assignments that involve graphical user interfaces. This doesn't mean that the students need to implement the graphics. Rather, given the ubiquitous presence of "sexy" computer applications in their lives, students will be more engaged in assignments that look and feel like the computer programs with which they are familiar.

Use assignments that include an element of chance, experimentation, or surprise. We have found that students are more engaged in the software development process when the end artifact is something with which the students can "play" and experiment.

Use assignments that have a connection to the student's perception of the "real world". Seemingly more so every year, our students want to see applications even in the first semester that have some kind of connection to the real world. We have found it much easier to keep students engaged if they can see some real-world application for the system they are developing.

2.2 Design Pattern Assignment Selection Principles

Use classic computer science assignments as the basis for the design pattern assignments. According to Webster's Dictionary, "classic" is defined as 1) a work of enduring excellence; 2) historically memorable; 3) a traditional event. Classic computer science examples by their very definition are excellent examples for illustrating key computer science concepts. We should harness the proven value of these classics even when attempting

to introduce students to additional concepts beyond the original intent.

Remove all unnecessary complication from the design pattern without removing the essential characteristics. Design patterns, in their full glory, typically involve the use of abstract classes, interfaces, inheritance, polymorphism, and so on. This can be a daunting list of concepts for an entry-level student to consume at one time. However, many specific applications of design patterns do not really call for this generality. Design patterns can be simplified for presentation to entry-level students without losing the essential characteristics that make the design patterns valuable.

Choose only design patterns that have a real value-added to the application. As with almost any software development concept, design patterns can be applied in places where they really don't have anything to add to the design. Doing so tends to leave students with the feeling that all design patterns add to a design is complexity [3]. It is important to carefully choose the design patterns for entry-level students so that the value-added of the design pattern is obvious and real.

Use refactoring as a mechanism for helping students to understand the power and impact of design patterns. Entry-level students are highly unlikely to come up with the designs suggested by typical design patterns. Rather, they tend to take the fastest and easiest solution to the problem (which doesn't mean the best solution). By starting students with a design that they find reasonable and understandable and then refactoring that design to introduce design patterns, students get a better appreciation for value-added by the design pattern.

The Game of Life fits particularly well with most of these guiding principles. The Game of Life is a classic computer science assignment with a proven record of teaching students important programming techniques. The graphical user interface for the Game of Life is simple enough to provide as a pre-cooked software component but sufficiently interesting in appearance so as to keep the students' attention. While the Game of Life is driven by deterministic rules, the behavior of those rules over sufficiently many generations is interesting and surprising. The Game of Life also provides a fertile ground for introducing important design patterns that have a real and significant value-added to the student. The Game of Life, however, does not score very well on the "real-world" metric. Certainly, one can show the types of biological populations that the Game of Life can be used to model, but for most students this is a stretch. However, given the other qualities of the Game of Life assignment, we have still found it to be a popular and engaging exercise for the students.

3 THE LABORATORY SEQUENCE

This section outlines a series of closed laboratory assignments based on the Game of Life that, one by one, introduce entry-level computer science students to the Observer, State, Singleton, Command, and Visitor design patterns [1].

3.1 Pre-Laboratory Design

At the beginning of the first laboratory, students are given a complete implementation of the Game of Life as summarized in the design shown in Figure 1. This design is best described as the "monolithic" design that an entry-level computer science might originally develop. The design centers on a single class that includes both the domain rules of the Game of Life and the user interface. The implementation is straight-forward enough that nearly all students can quickly and easily digest the code. However, it doesn't typically take very long before the students realize that this simplistic design lacks the kind of generality and robustness appropriate for the Game of Life application.

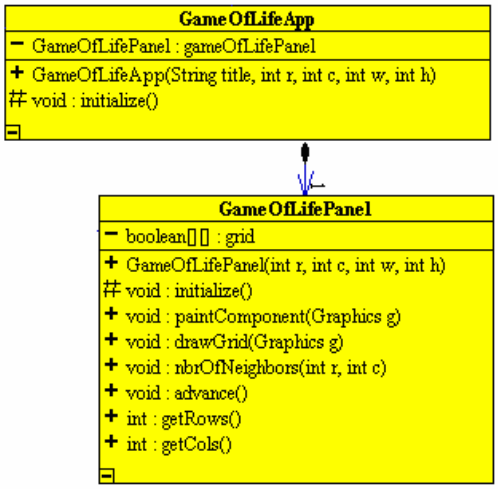

Figure 1: A Starting Design

3.2 The Observer Design Pattern

The first improvement that we introduce to the students is the concept of separating the presentation and domain layers of the system. In particular, we introduce the Observer design pattern [1,p.293]. The Observer design pattern is applicable and appropriate in many situations including when 1) the application has two separate aspects that can be varied independently of one another; or 2) the application involves objects that when changed require changing other objects.

The Game of Life has both of these characteristics. Students see that the visual representation of the Game of Life and the actual structure of "live" and "dead" cells are two separate aspects of the system. Further, the students are quick to point out that when the cells of the game change state (from "live" to "dead" or vice versa) the domain must notify the graphical user interface to allow it to update itself. Likewise, when the user clicks on a cell in the user interface to toggle the cell from "live" to "dead" or "dead" to "alive", the user interface must notify the domain so that it can record the appropriate changes to its model.

The students are then introduced to the refactored design summarized in Figure 2.

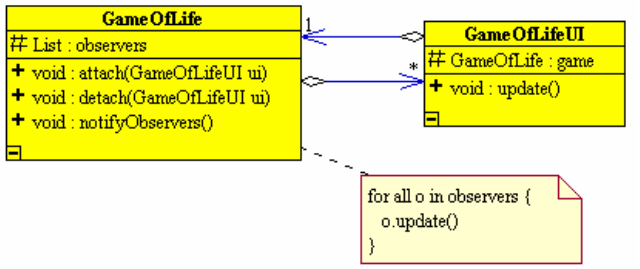

Figure 2: The Observer Pattern in the Game of Life

Students are given the code for this improved design but asked to complete several critical elements that achieve the desired realization of the Observer design pattern:

- Implement the "attach(GameOfLifeUI)" method of the GameOfLife class that allows additional user interfaces to be attached to the game.
- Implement the "detach(GameOfLifeUI)" method of the GameOfLife class that removes a user interface from the list of observers for the game.
- Implement the "notifyObservers()" method of the GameOfLife class which involves the for-loop shown in Figure 2.
- Modify the "advance()" method of the GameOfLife class so that it correctly causes all observers to be notified of changes to the domain.
- Modify the "update()" method of the GameOfLifeUI class so that it correctly retrieves the model from the GameOfLife and renders the new state on the screen.

Notice that while the students are given considerable portions of the design already coded, they are asked to gain first-hand experience with implementing the primary aspects of the Observer design pattern.

3.3 The State Design Pattern

The next stage of the laboratory is designed around two primary lessons: 1) software solutions should be designed around the language of the problem not the language of the solution; and 2) polymorphism is a powerful technique for enabling objects to change their behavior over their lifetime. To help instill these lessons into the students, we next introduce the State design pattern [1,p.305] which is appropriate in many situations including when an object's behavior depends on its state and it must change its behavior at run-time depending on its state.

In the previous implementation, the "state" of a cell ("alive" versus "dead") is represented as a matrix of Boolean values. This of course leads to conditionals that ask "if cells[i][j] is true then...". Clearly, the domain of the Game of Life does not involve Boolean values. Rather, the language of the problem talks about cells as either "alive" or "dead". We illustrate to students that by using a Boolean matrix, we have exposed a design decision.

Further, students also see that a given cell conceptually changes state some times as the program executes. Using the State design pattern we present the students with the refactored design summarized in Figure 3.

In this design, we replace the matrix of Boolean values with a matrix of Cells. Each Cell instance holds an instance of a CellState. The CellState is either an instance of DeadState or AliveState. When an instance of the Cell class receives a message, the Cell instance passes the message onto the CellState instance whose behavior is determined by its actual (dynamic) type. For example, an instance of AliveState, when requested to "toggle()" returns an instance of DeadState. This new instance is saved by the Cell. In the future, the Cell will now behave as if it were actually DeadState.

Again, students are given the code for this design and asked to complete a few critical methods involved in the concrete implementation of the State design pattern:

- Implement the "live()" method of the Cell class to simply call the "live()" method of the CellState class and save the returned CellState as the new CellState for the Cell object.
- Implement the "die()" method of the Cell class following the same mechanism as the "live()" method described above.
- Implement the "live()" method of the DeadState class so that it defines the behavior of a dead cell coming to life.
- Implement the "die()" method of the AliveState class so that it defines the behavior of an alive cell dieing.
- Modify the "advance()" method of the GameOfLife class so that it uses a matrix of Cells.

This specific set of exercises gives the students hands-on experience with the implementation of the delegation which lies at the heart of the State design pattern.

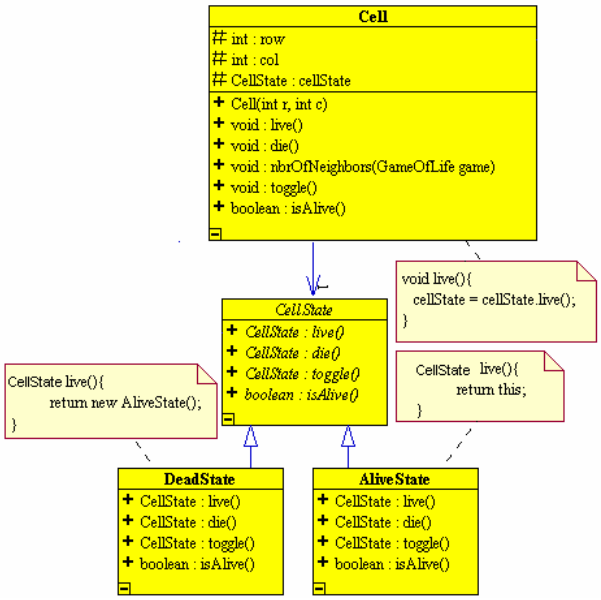

Figure 3: The State Pattern in the Game of Life

3.4 The Singleton Design Pattern

Next, the students are shown that AliveState and DeadState just added to the design have no local state themselves. That is, they have no attributes. It thus seems rather silly and inefficient to keep generating new instances of these two classes as the program proceeds from generation to generation. The Singleton design pattern [1, p.127] is applicable in many situations including when 1) there must be exactly one instance of a class; or 2) a class contains no local state

Figure 4 summarizes the application of the Singleton design pattern to the Game of Life.

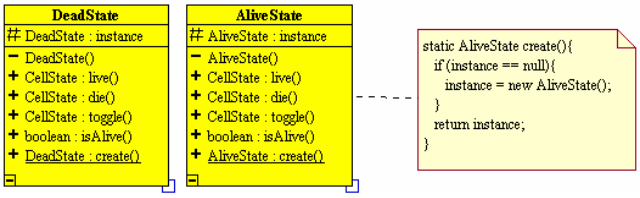

Figure 4: The Singleton Pattern in the Game of Life

Notice that this design involves the use of a private constructor and a public "create()" method to gain access to the single shared instance of each class. Again, the students are given code and asked to complete several key aspects:

- Implement the static "create()" method in the AliveState class as shown in Figure 4.
- Modify the "die()" method of the AliveState class so that it uses the "create()" method to get an instance of the DeadState rather than using the now private constructor.
- Initialize the static variable in the DeadState class that holds access to the one shared instance of the DeadState class.
- Modify the "live()" method of the DeadState class so that it uses the "create()" method.
- Modify the constructor of the Cell class so that it is implemented using the "create()" method of either the DeadState or AliveState class.

This particular set of exercises gives the students hands-on experience with the use of a static variable and a private constructor to control class instantiation – the essence of the Singleton design pattern.

3.5 The Command Design Pattern

Recall that the Game of Life uses the states of the surrounding cells to determine the state of each cell in the next generation. For example, based on the particular rules used, a live cell with a certain number of live neighbors dies (starvation). However, you can't simply use a pair of nested for-loops to walk through the matrix of cells changing them as appropriate. To do so would then change the number of alive and dead cells for the neighbors of the mutated cell and thus would destroy the environment that should have determined the states of the neighboring cells. The typical solution used by entry-level programmers is to create a second copy of the matrix, using the original matrix to decide if cells live or die and then actually mutating them only in the copy of the matrix. After all cells are processed, the original matrix is replaced with the new copy. This seems to strike students as silly and inefficient (because it is). The real problem is that we need to separate the time between when we decide that a live cell must die or a dead cell must live. The Command design pattern [1,p.233] is appropriate when you wish to specify, queue, and execute requests a different times. Figure 5 summarizes the application of the Command design pattern to the Game of Life.

The solution involves the creation of two classes that represent the "live" command given to a dead cell or the "die" command given to a live cell. As the GameOfLife moves through the matrix of cells, it creates instances of the LiveCommand or the DieCommand as appropriate. Notice that both LiveCommand and DieCommand are subclasses of LifeCommand which holds the actual cell involved in the command. When the matrix is completely processed, the "execute()" method of each saved LifeCommand is run which in turns sends the appropriate request (live() or die()) to the appropriate Cell.

The students are given an implementation of the Game of Life using the Command design pattern and are asked to:

- Implement the constructor of the LifeCommand class so that it saves the specific Cell instance for which the message is intended.
- Implement the "execute()" method of the DieCommand class as shown in Figure 5.
- Implement the "execute()" method of the LiveCommand class analogous to the "execute()" method of the DieCommand class.
- Modify the "advance()" method in the GameOfLife class so that it creates a list of LifeCommands as it moves through the Cell matrix. It must also include a loop to move through the resulting list asking each LifeCommand to "execute()".

This specific set of exercises allows the student to gain hands-on experience with the single most important aspect of the Command design pattern; namely the ability to separate the construction of a request from its actual execution.

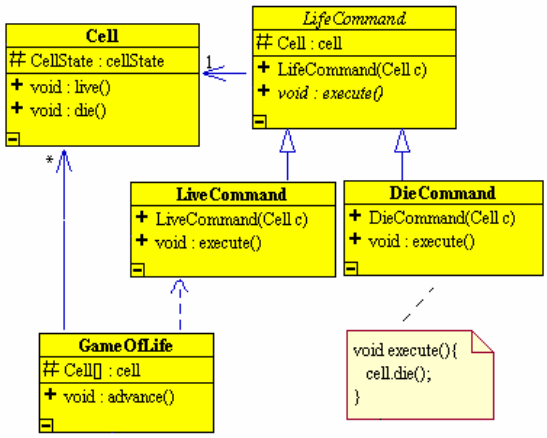

Figure 5: The Command Pattern in the Game of Life

3.6 The Visitor Design Pattern

At this point in the laboratory, the students have created a rather robust design for the Game of Life implementation. One (at least) serious defect still remains, however. Namely, the survival rules of the Game of Life, for which there exist numerous variations, have been coupled to the implementation of the production of new generations. This is where the Visitor design pattern comes in [1,p.331]. The Visitor design pattern can be used in several situations including when 1) many distinct operations need to be performed on objects in an object structure and you want to avoid "polluting" their classes with these operations; or 2) the classes defining the object structure rarely change, but you often want to define new operations over the structure.

For the Game of Life, the object structure is the matrix of cells. The distinct operations are the survival rules that we wish to apply to the matrix of cells to create the list of LifeCommands. Most variations of the Game of Life focus on variations in the survival rules and not on the states of a cell (alive vs. dead) and therefore the object structure doesn't need to change but the operations (survival rules) do need to change. Figure 6 summarizes the application of the Visitor design pattern to the Game of Life.

The basic idea is that each Cell in the matrix is given a method "accept(...)" that allows a particular LifeVisitor (survival rule) to be applied to each Cell. The Cell, as it always does, simply delegates the "accept(...)" request to its CellState instance. The CellState class (which is either an instance of DeadState or AliveState) invokes the appropriate "visit*X*(...)" method from the LifeVisitor (for example, "visitLiveCell(...)". This method

applies the particular rules which define how to visit a live cell and either places a new LifeCommand in a list or does not, as appropriate. Notice that in this design, the detail of the survival a rule are decoupled from the operation of the Game of Life and thus can be allowed to vary independently and dynamically as the game operates.

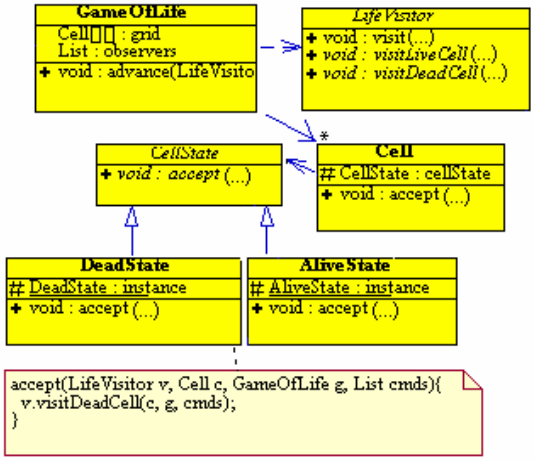

Figure 6: The Visitor Pattern in the Game of Life

Again, the students are given code for this design of the Game of Life and asked to:

- Implement the "accept(...)" method of the Cell class which simply delegates the method call to the current CellState instance.
- Implement the "accept(...)" method in the AliveState class by having it invoke the "visitAliveCell(...)" method of the LifeVisitor.
- Implement the "accept(...)" method of the DeadState class analogous to the same method in the AliveState class.
- Implement the "visit(...)" method of the LifeVisitor class using double dispatching to invoke the appropriate state-specific visit method.
- Modify the "advance()" method in the GameOfLife class to use the explicit LifeVisitor rather than the hard-coded survival rules.
- Implement the "visitDeadCell(...)" method in a LifeVisitor subclass to correctly define the appropriate survival rule for the traditional Game of Life implementation.

This set of exercises is by far the most extensive and challenging for the students. As is shown in the next section, for our specific laboratory sequence, this portion of the laboratory sequence is a single closed laboratory by itself.

3.7 A Sample Division of the Lab Sequence

Our entry-level course includes a 2-hour per week closed laboratory setting. The entire Game of Life sequence involves three of these laboratories periods (of a total of 15 laboratory periods). The laboratory sequenced is introduced in the final third of the semester after the students have already experienced laboratories on rather standard CS1 material. In the first laboratory, the students are introduced to the concept of design patterns, given the monolithic starting design, and asked to update the design to include the Observer design pattern and the State design pattern. In the second laboratory, the students are refreshed on the application and asked to incorporate the Singleton and Command design patterns. The third and most challenging laboratory is the third laboratory which involves the incorporation of the Visitor design pattern. Again, however, this is just a sequence that we have found useful and appropriate.

4 SUMMARY AND CONCLUSION

We have described a series of closed laboratory experiences that help students to learn the fundamentals of the five powerful design patterns within the context of the classic Game of Life computer application. We have developed a specific set of exercises that allow each student to gains hands-on experience with the essential characteristics of these design patterns without becoming overwhelmed by the need to implement the other aspects of the system. Further, by selecting core intertwined functionality for the students to implement, we avoid the kind of "blind coding" that can be a problem with "program-in-progess" assignments in which students add a single line here or there but never understand the bigger picture. The students don't need to understand the details of how the entire system works but rather are focused on understanding just those aspects that are affected by the design pattern under study.

We have also found that from this first-year experience, most students are well prepared to further study and apply these and other design patterns in subsequent courses. More importantly, each student has the case studies of this laboratory sequence in their background as a common example of how object-oriented design and separation of concerns can lead to robust and powerful designs.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley Publishing, (1994).

[2] D. Nguyen, and S. Wong. *Design Patterns for Sorting,* ACM SIGCSE Bulletin 33(1):263-267, Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education , 2001.

[3] O. Astrachan, *OO overkill: when simple is better than not,* ACM SIGCSE Bulletin 33(1): 302-306, Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, 2001.

[4] M. Wick, *An object-oriented refactoring of Huffman encoding using the Java collections framework,* ACM SIGCSE Bulletin 35(1): 283-287, Proceedings of the 34th SIGCSE technical symposium on Computer Science Education, 2003.

[5] M. Wick, *Kaleidoscope: using design patterns in CS1,* ACM SIGCSE Bulletin 33(1): 258-262, Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, 2001

[6] M. Gardner, *The fantastic combinations of John Conway's new solitaire game "life",* Scientific American, 223: 120-123, 1970.