Sudoku — A little lesson in OOP.

Axel T. Schreiner  
Rochester Institute of Technology  
102 Lomb Memorial Drive  
Rochester NY 14623-5608  
+1 (585) 475-4902  
ats@cs.rit.edu

James E. Heliotis  
Rochester Institute of Technology  
102 Lomb Memorial Drive  
Rochester NY 14623-5608  
+1 (585) 475-6133  
jeh@cs.rit.edu

ABSTRACT

Paying only lip service to the principles of object-oriented programming rarely results in the expected benefits. This paper chronicles two implementations of a graphical user interface for machine-assisted Sudoku puzzle solving. The first implementation seems to be object-oriented but ends up with rather confusing and inefficient code. The second implementation looks deeper into the philosophy of object orientation and produces a significantly better result. The paper concludes with some general pointers why and how the initial mistakes could have been avoided.

Categories and Subject Descriptors

D.2.3 [Coding Tools and Techniques]: Object-oriented programming, D.2.11 [Software Architectures]: Information hiding, Patterns.

General Terms

Algorithms, Design, Languages.

Keywords

Sudoku, Design.

1. INTRODUCTION

Sudoku [1], a simple logical puzzle invented in the United States, has turned into a global craze. A 9x9 grid is divided into nine non-overlapping boxes each containing a 3x3 grid of cells. Each row and each column of the original grid as well as each box must contain each of the digits from 1 to 9 exactly once. For a puzzle, a few digits are already entered in the grid and the remaining digits must be deduced.

Sudoku provides an intriguing story line for programming problems. For example, solvers [2,3,4] can serve to discuss approaches to problem solving mandated by different languages. On a simpler level, a Sudoku worksheet [5] should be a relatively simple example of a dynamically changing graphical user interface with undo capabilities, combined with a program state modeled by an array with various slices; the state of a grid cell can be an integer or a more sophisticated object controlling data entry.

Adding a solving algorithm might be a bit ambitious for a more introductory programming course because it is likely to require backtracking in a relatively complex data structure, but students might enjoy being asked to implement some machine assistance for the more tedious aspects of Sudoku solving.

In this paper we present a series of designs for the Sudoku application that will lead introductory students through the all-important process of trial and error. They will see examples of design analysis, criticism, and improvement.

2. SUDOKU SOLVING PRIMER

Initially, a cell can contain any digit from 1 to 9. However, the cell belongs to exactly three grid slices: one row, one column, and one box, and the cell may not contain any digit whose value is already in another cell in one of these slices. This observation leads to the notion of a candidate list for each empty cell. Computing the lists is a major benefit of machine-assisted solving — and of course an anathema to all Sudoku aficionados.

There are a couple of small steps that even a limited solver can perform. First, if any candidate list is empty the puzzle has no solution. Second, if a candidate list is a singleton it obviously can decide which digit has to be entered into a cell.

Given a cell with a singleton candidate list, the singleton can be pruned from all candidate lists in the slices to which the cell belongs — a more significant step in machine-assisted solving.

Any puzzle worthy of the name will require more sophisticated pruning. For example, if two candidate lists in the same slice are equal pairs, they leave a choice of two digits for two cells in the slice, but the pair can be pruned from the other lists in the slice — often a leap of sophistication in automated solving.

3. IMPLEMENTATIONS

The steps of the programming assignment [5] are intended to encourage an object-oriented solution:

1. Create an observable model (puzzle state and some algorithms, in particular, undo), and test it from the command line.
2. Create a read-only view to experiment with layout and observing.
3. Finally, show the candidate lists in a dynamically changing view, which allows candidate selection based on the lists and undo and sends requests back to the model.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA ’06, October 22-26, 2006, Portland, OR, USA.  
Copyright 2006 ACM 1-58113-000-0/00/0004…$5.00.
3.1 Half-Baked Programming

The assignment is structured to include OOP paradigms such as Model-View-Controller and the observer pattern. To make it even simpler, the model is required to have methods such as

```java
int[] get (int row, int col)
which returns an array of candidate digits for the indicated position, and
```

```java
void set (int row, int col, int digit)
which enters the digit at the indicated position, remembers the operation for undo, and discards any operations remembered for redo.
```

The only significant design decision seems to be when to deal with the candidate lists — assume for the moment that each cell is represented as an `Integer.set` could inform the other cells in its context, meaning in all its affected slices, that a digit has been found or get could compute the lists as needed — following one or more undo operations, get will likely have to ask the cells for current information anyhow. The code should look something like the following:

```java
canBe = new BitSet();
for (int digit = 1; digit <= dim; ++ digit) {
    // row
    for (int c = 0; c < board[row].length; ++ c)
        if (c != col && board[row][c].equals(digit))
            continue search;
    // column
    for (int r = 0; r < board.length; ++ r)
        if (r != row && board[r][col].equals(digit))
            continue search;
    // box
    int r = (row/boxDim) * boxDim,
        c = (col/boxDim) * boxDim;
    for (int i = 0; i < boxDim; ++ i)
        for (int j = 0; j < boxDim; ++ j)
            if ((r+i != row || c+j != col)
                && board[r+i][c+j].equals(digit))
                continue search;
    // digit is a viable candidate
    canBe.set(digit);
}
```

It looks gruesome, but array slicing has been dealt with. The model could even prune singletons, i.e., if get returns an array of length 1 this could be considered equivalent to a set operation. However, this is likely to confuse the user interface for undo.

It is now time to revisit an earlier assumption as the next design decision: how to model a cell, i.e., what are the elements of each row of board? The idea for now is to make each cell an object that implements an interface `Digit` containing the following methods:

```java
boolean equals (int digit)
which returns true if the digit was entered by set,
```

```java
int[] digits ()
which returns the (cached) result of get, and
```

```java
boolean canBe (int digit)
which returns true if the digit is a candidate.
```

There are two implementations of the interface: `Move` can be the class that represents the effect of set and returns true for the proper argument to equals and false for all calls to canBe. On the other hand, before a set operation is successfully performed, there would be a class `Digits` that holds a candidate list. Digits would return false for equals and true if the argument of canBe is a candidate.

Unfortunately, once all the pieces are put together, the result is messy and looks impossible to extend. As discussed above, pruning singleton candidate lists can be done more or less silently, but there is no reasonable way to prune pairs.

If model and view are based on Java’s classes `Observable` and `Observer`, the view will use get to inquire about the state of a cell but the result cannot distinguish between a digit entered by set or a singleton candidate list. Depending on the user interaction to be implemented, the view might have to track (and undo) all set operations itself!

3.2 OOP

Where did the approach go wrong -- in spite of MVC, the observer pattern, and cell objects with different behaviors? There seem to be three basic mistakes: The model is not informative enough, the code for computing candidate lists exhibits some information leakage from the cell objects to the code in the model, and the slice iteration should be uniform enough to employ the for-each loop introduced in Java 5 [7].

Communication between model and view was based on Java’s `Observable` and `Observer`, i.e., the model sends update to the view and expects the view to use `get` to acquire the relevant information. This makes the model a rather passive participant in the object conversation and get does not even reveal enough. Here is a problem-specific observer interface:

```java
void move (int row, int col, int digit)
describes a users move.
```

```java
void ok (int row, int col, int digit)
describes a candidate list.
```

```java
void queues (int undos, int redos)
describes the number of undo and/or redo operations still possible.
```

Any number of these messages can be sent from the model to each observer as a response to a `set`, undo or redo operation. This hugely simplifies the implementation of a view because the view now only needs to visualize cell states as indicated by each move and ok it receives. queues messages can be used to control undo and redo button activation.

It turns out that the observer interface does not make the job described by the model interface more complicated. On the contrary, `set` is still used to enter a digit into a cell, change the undo state, and trigger recomputation of the candidate lists. But now as things change several messages are sent to the observers. An undo operation complements the last `set` which again results in a number of messages, and a redo operation acts like `set` but changes the undo state differently.

Plus, as an unanticipated benefit, machine-assisted solving can now be naturally added to the model interface: just like `set`, a request to infer moves from singletons, i.e., to turn singletons into selected digits, can also result in a number of `move` and `ok` messages, as can a request to prune singletons, or pairs, etc. The view simply visualizes the resulting changes in the state of the puzzle.

Turning now to the problem of information leakage, table 1 shows part of what a cell (a `Digit`) should implement, depending on whether it represents the effect of `set` or a candidate list. The key aspect is that the representation of the candidate list has been moved into the cell object. While the constituents of a candidate
array slicing is still an important way to illustrate reusability. In our example, others makes it look like a pure programming concern, it is important to realize that iteration is built into the Java language and that it is a part of the language syntax. Finally on the agenda for improvement is slice iteration. Although a single slice can be based on the following abstract class:

```java
abstract class Slice implements Iterator<Digit> {
    protected int n = 0; // state
    public boolean hasNext () { // default
        return n < dim;
    }
    public void remove () {
        throw new UnsupportedOperationException();
    }
    public Digit next () {
        return slice.next();
    }
    Iterator<Digit> slice = slices[n].iterator();
}
```

Based on this class, iterators for rows, columns, and boxes can easily be implemented. For each requires an Iterator or an array but an array cannot be converted to an Iterator. Therefore, even a row has to be wrapped as an Iterator. Iterators can even be constructed from other iterators. Here is how an iterator over the context of a cell is constructed:

```java
Iterable<Digit>[] slices = slices(row, col);
```

Table 1: Digit implementations.

<table>
<thead>
<tr>
<th>Move</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int digit;</td>
<td>BitSet digits;</td>
</tr>
<tr>
<td>int digit()</td>
<td>return digit;</td>
</tr>
<tr>
<td>BitSet digits()</td>
<td>return digits;</td>
</tr>
<tr>
<td>boolean isKnown()</td>
<td>return true;</td>
</tr>
</tbody>
</table>

Similar pairs of methods can be implemented to find unique digits in a context or to prune pairs, see [6] for details.

Finally on the agenda for improvement is slice iteration. Although the fact that iteration is built into the Java language and many others makes it look like a pure programming concern, iterator use is still an important way to illustrate reusability. In our example, array slicing into rows, columns, and boxes seems to require quite a bit of code duplication and therefore introduces a potential for error. In addition, you will note that, thus far, the methods shown were coded with the understanding that they should all benefit from the linguistic simplicity of the for-each loop introduced in Java 5 [7].
view to find out what it needs to know. Unfortunately, the informa-

tion function get did not provide sufficient detail.

- The first model used cell objects but did not involve them in

algorithms such as candidate list pruning. This resulted in in-

formation leakage and complicated algorithms.

- The first model contained a lot of code duplication because

there was no uniform approach to array slicing, i.e., there was

no systematic use of the iterator pattern.

Through the apparently simple problem of a popular puzzle, we

have succeeded in demonstrating in a very practical way the ad-

vantages of following basic rules of object-oriented design, and

even several of the more advanced design tenets. Here are some

examples:

- When done properly, information hiding expands the usefulness

  of the objects you design.

- The instanceof operator can and should be avoided.

- In OOD, algorithms should be distributed over the involved

  objects through method invocation (object-to-object communi-

  cation).

- Solve large problems by breaking them down into small prob-

  lems (divide-and-conquer).

- As you gain experience with your design, be prepared to refac-

  tor it: change the level of an interface, shift responsibilities, etc.

  Perhaps more fundamental is the famous rule, “Plan to throw

  one away; you will anyhow.” [8]

The Sudoku puzzle has shown itself to be an application domain

that can be visited once or as many times as appropriate for a

laboratory course involving programming and design with objects.

REFERENCES


[2] Schreiner, A. Sudoku Solver (Scheme) Sep. 21, 2005


<http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.html>.

[8] Brooks, F. The Mythical Man Month (2nd ed) Addison-

Wesley 1995.