Top-Down Design Example

CS 1025 Computer Science Fundamentals I

Stephen M. Watt

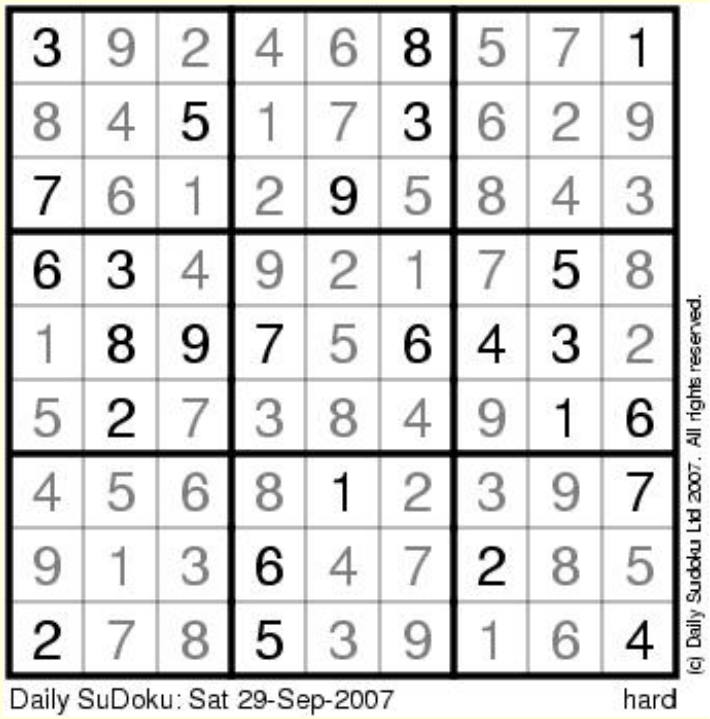
University of Western Ontario

Example of Top-Down Design

- *Write a program to verify that a 9 x 9 grid is a Sudoku solution.*

- Each row and column and 3x3 square (indicated by heavy lines) must have each of the digits 1-9.

- Because each row, column and 3x3 square have exactly 9 places, this implies that each digit appears exactly once in each of them.



3	9	2	4	6	8	5	7	1
8	4	5	1	7	3	6	2	9
7	6	1	2	9	5	8	4	3
6	3	4	9	2	1	7	5	8
1	8	9	7	5	6	4	3	2
5	2	7	3	8	4	9	1	6
4	5	6	8	1	2	3	9	7
9	1	3	6	4	7	2	8	5
2	7	8	5	3	9	1	6	4

Daily SuDoku: Sat 29-Sep-2007 hard

Top-Down Approach

- Imagine you had only 2 minutes to split the problem into big logical chunks. What would you do?
- You might come up with something like this:

```
1. Get the input somehow.
2. Check the various constraints:
   2a. Check rows.
   2b. Check columns.
   2c. Check 3x3 squares.
3. Write output.
```

- Then you would start to think about how to check the rows etc.
- At this point you would try to describe each of the steps in terms of its own high-level sub-problems.
- Keep doing this until the steps are easy to program.

A Little Bit of Bottom-Up

- At any given level, it is useful to think about whether any of the sub-problems can be described in a common way.
- This can lead to code sharing and consequent easier development and maintenance.
- In the Sudoku problem we should ask ourselves
  Can we describe checking rows, columns and 3x3 squares in a common way?

Designing a Class Hierarchy

- Once we have identified steps that should have a common solution, we use these relations to develop a class hierarchy.

  That is, things that should have common operations should have a common base class.

- In the Sudoku example, we see that rows, columns and 3x3 squares should share a base class.

  We will call this a **`Slice`**, and call the subclasses **`RowSlice`**, **`ColumnSlice`** and **`SquareSlice`** respectively.

A High-Level Solution

- At the top level, the solution might look like:

```
public class Sudoku {
   public static void main(String[] args) {
            Tableau t = new Tableau( ...);    t.print();
            boolean ok = checkSudoku(t);
            if (ok) System.out.println("The Sudoku is OK.");
            else    System.out.println("The Sudoku is not OK.");
   }
   public static boolean checkSudoku(Tableau t) {
            Slice s;
            for (int i = 0; i < t.size(); i++) {
                     s = new RowSlice (t, i);      if (!checkSlice(s)) return false;
                     s = new ColumnSlice(t, i);  if (!checkSlice(s)) return false;
                     s = new SquareSlice(t, i);   if (!checkSlice(s)) return false;
            }
            return true;
   }
   public static boolean checkSlice(Slice s) { ... }
}
```

Solving Sub-Problems

- We now need to determine how to:
  1. Input a Sudoku
  2. Check a Slice.

- For convenience, we will input a Sudoku as an array of strings, one for each row, and ignore non-digits (e.g. for spacing).

- To check a Slice, we will require that a Slice be able to return the digits it contains.

  The positions in a slice will be numbered 0..8 (more generally 0..size()-1) in some order depending on the kind of slice.

Solving Sub-Problems II

- To check a Slice, we need to verify that each of the digits 1..size() occurs exactly once. How to do this?
- Three ways come to mind:
  1. Use an array of integers with one slot for each digit, and count how many times each digit occurs.
  2. Use an array of booleans with one slot for each digit, and keep track of whether each digit occurs.
  3. For each digit, make a pass through the Slice to see whether the digit occurs.
- Evaluate the methods:
  1. O(n) space, O(n) time. [i.e. Space and time proportional to size()]
  2. O(n) space, O(n) time.
  3. O(1) space, O(n^2) time.
- Guess which might be better, then verify if it is important.

Sub-Problem Solution

- The solution for checking a slice would then look like:

```
public static boolean checkSlice(Slice s) {
        for (int i = 1; i <= s.size(); i++) {
                // Check that value i is there.
                boolean found = false;
                for (int j = 0; j < s.size() && !found; j++) {
                        if (s.getValue(j) == i) found = true;
                }
                // If not found, then the check fails.
                if (!found) return false;
        }
        // All were found so check succeeds.
        return true;
}
```

The Slice Classes

- We now need to give implementations for the class hierarchy.
- We will never have any object that is just a Slice.
  All will be either a RowSlice, ColumnSlice or SquareSlice.
- Java has a notion of an “abstract” class to say there will never be any objects that belong exactly to this base class. You can then declare methods as “abstract” to say that the subclass must implement them.

```
public abstract class Slice {
        abstract public int size();
        abstract public int getValue(int i);
}
```

The Slice Classes

- We can think of two easy ways to implement the subclasses:
  - Have a private array into which we copy the relevant numbers.
  - Keep the original Sudoku and some info saying which part is needed.
- We choose the second method to avoid creating lots of arrays.

```
public class RowSlice extends Slice {
        private Tableau  _t;
        private int          _rowno;

        public RowSlice(Tableau t, int rowno) { _t = t; _rowno = rowno; }

         final public int size() { return _t.size(); }
         final public int getValue(int i) { return _t.getValue( _rowno,  i ); }
 }
```

The Slice Classes II

- The columns can be done in exactly the same way as the rows.

```
public class ColumnSlice extends Slice {
        private Tableau  _t;
        private int           _colno;

        public ColumnSlice(Tableau t, int colno) { _t = t; _colno = colno; }

         final public int size() { return _t.size(); }
         final public int getValue(int i) { return _t.getValue( i, _colno); }
 }
```

The Slice Classes III

- The SquareSlice class is trickier.
- We can number the 3x3 subsquares as 0..8 as follows:

```
0 1 2
3 4 5
6 7 8
```

- Then, for example, the 5th 3x3 square will consist of positions

```
(3,6)  (3,7)  (3,8)
(4,6)  (4,7)  (4,8)
(5,6)  (5,7)  (5,8)
```

  in the original Sudoku.

- The starting squares for all of the 3x3 subsquares are:

```
(0,0) (0,3) (0,6)      (0,0) (0,1) (0,2)      (0/3, 0%3) (1/3, 1%3) (2/3, 2%3)
(3,0) (3,3) (3,6) = 3x (1,0) (1,1) (1,2) = 3x (3/3, 3%3) (4/3, 4%3) (5/3 ,5%3)
(6,0) (6,3) (6,6)      (2,0) (2,1) (2,2)      (6/3, 6%3) (7/3, 7%3) (8/3, 8%3)
```

- We can therefore keep track of the starting square as

```
_row0 = (n / _t.subsize()) * _t.subsize();
_col0 = (n % _t.subsize()) * _t.subsize();
```

The Slice Classes III

- We can number the squares within each of the subsquares using the same pattern.

```
public class SquareSlice extends Slice {
        private Tableau _t;
        private int _row0, _col0;

        public SquareSlice(Tableau t, int n) {
                _t = t;
                _row0 = (n / _t.subsize()) * _t.subsize();
                _col0 = (n % _t.subsize()) * _t.subsize();
        }
        final public int getValue(int i) {
                int r = _row0 + i / _t.subsize();
                int c = _col0 + i % _t.subsize();
                return _t.getValue(r, c);
        }
        final public int size() { return _t.size(); }
}
```

The Tableau Class

- Notice that we have not said how the Sudoku tableau should be implemented yet.
- We have merely supposed that it should support the methods:

```
int size();        // number of rows or cols in the Sudoku
int subsize();  // number of rows or cols in a subsquare. = sqrt  of size()
int getValue(int r, int c);  // value in row r, col c  using 0-based indexing.
```

- We could implement using a 2 D array, a 1 D array or something else.

- Always start simple. We choose a 2 D array.

The Tableau Class II

- Ignoring constructors for the moment, we have:

```
public class Tableau {
    private int _subsize;
    private int _size;

    private int[][] _tableau;

    public int size()    { return _size; }
    public int subsize() { return _subsize; }
    public int getValue(int r, int c) { return _tableau[r][c]; }
}
```

The Tableau Class III

- Before writing code to fill tableaux, we write a constructor to create an empty tableau, and one to output tableaux (useful for debugging).

```
private Tableau(int n) {
      _size    = n;
      _subsize = isqrt(n);
      _tableau = new int[n][n];
 }
 // Compute the square root of a small square integer.
 private static int isqrt(int n) { int i = 0; while (i*i < n) i++; return i; }

 // We put blank lines and columns between sub-squares.
 public void print() {
      for (int i = 0; i < _size; i++) {
          if (i > 0 && i % _subsize == 0) System.out.println();
          for (int j = 0; j < _size; j++) {
              if (j > 0 && j % _subsize == 0) System.out.print(" ");
              System.out.print(_tableau[i][j]);
          }
          System.out.println();
      }
  }
```

The Other Sub-Problem: Input

- We could write a constructor that takes an array of arrays of integers as its arguments. That would be easy to write.

- If we anticipate reading input from a file at some point, it would be useful to write a constructors that take strings as input. I.e.

  ```
  public Tableau(String[ ] rows) { ... }
  public Tableau(int nrows, String contents) { ... }
  ```

- To do this, we make use of the length() method from String and the static methods isDigit(c) and digit(c, base) from Character.

- A useful component would be a method that fills an empty row by consuming part of a string. We can use this to implement *both* contructors.

Input II

- This row-filling string muncher can be written as follows:

```
// Fill a row from a string, starting at index ix.
// Return the position of the first unused index.
// If the string isn't long enough, the rest of the row is filled with
// Non-digit characters are ignored.  This allows spacing.
private static int fillRow(String s, int ix, int[] row) {
      int rix = 0;
      for ( ; ix < s.length() && rix < row.length; ix++) {
          char c = s.charAt(ix);
          if (!Character.isDigit(c)) continue;
          row[rix++] = Character.digit(c, 10);
      }
      for ( ; rix < row.length; rix++)
          row[rix] = 0;
      return ix;
}
```

Input III

- The public constructors can then be written as:

```
// Create a Sudoku tableau from an array of rows.
 public Tableau(String[] rows) {
     this(rows.length);   // Call the private emtpy-tableau constructo
     for (int i = 0; i < size(); i++)
         fillRow(rows[i], 0, _tableau[i]);
 }

 // Create a Sudoku tableau from a single string.
 public Tableau(int n, String initial) {
     this(n);  // Call the private empty-tableau constructor.
     for (int rix = 0, ix = 0; rix < n; rix++) {
         // Each iteration advances the starting position in the string
         ix = fillRow(initial, ix, _tableau[rix]);
     }
 }
```

Input IV

- A Tableau can then be created as:

```
Tableau t = new Tableau(
              new String[] {
                  "534 678 912",
                  "672 195 348",
                  "198 342 567",

                  "859 761 423",
                  "426 853 791",
                  "713 924 856",

                  "961 537 284",
                  "287 419 635",
                  "345 286 179"
              }
          );
```

Conclusions

- Thinking “top-down” can give an understandable high-level view of a program.
- Sometimes we get it right the first time.
  Sometimes we need to back up and try again.
- At points we consider the set of sub-problems and ask whether there is common ground among them.
  This allows for a little bit of “bottom-up” design to share code.
- Common solutions to parts of problems can be realized using common base-classes.