Solution of Deck of Cards

Maoxu Li
li@maoxuli.com

This is a solution of Deck of Cards problem focused on object-oriented design and shuffling algorithm. It is described in five steps. However, the processes of analysis, design, and implementation are actually iterative and incremental.

Step 1. Specify the problem

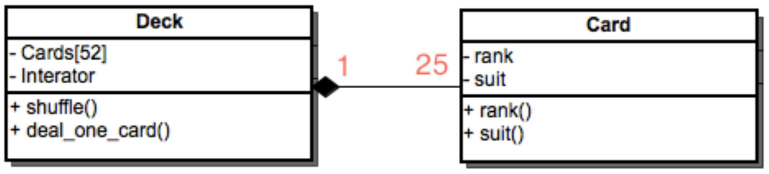
What should be included in solution has been described in the question. Here I specify the problem in details with some constrains (or what will not be included) as the basis of following analysis and design.

1. Only one deck of cards is used at any time during game. But new deck of cards can be used after the last one is discarded.
2. Only the operations of "shuffle" and "deal one card" are considered for a deck of cards. No extra game related actions are discussed.
3. A deck of cards can be shuffled for any times. The order of cards is changed with each shuffling. "When" and "how many" are determined by callers.
4. A deck of cards can be dealt one by one in its current order. Once the last card is dealt, there is no card dealt for the following requests.
5. A deck of cards can be reset with shuffling. After a shuffling, current position as to dealing cards will return to the first card.
6. Shuffling can occur at any time. It is possible to shuffle a deck of cards when a part (not all) of the cards have been dealt.

Step 2. Object-oriented modeling (v1)

The problem will be resolved with object-oriented paradigm. In general, object-oriented modeling is one of the key parts of object-oriented software engineering. In practice it is usually an iterative and incremental process of analysis and design. Some techniques, like "noun extraction" and "class-responsibility-collaboration (CRC) cards" can be used to draw classes from use cases. However, the result of this process is greatly dependent on the domain expertise of the developers.

As to this problem, the use case is so simple and clear that we may draw up the basic model of objects and their relationships with little effort. As is shown in figure below.



We will start with this simple model to implement an initial version of code (v1). Further discussion in Step 5 will refine this design and reach a second version of code (v2).

Step 3. Shuffling algorithm

An effective and high-efficient algorithm is needed to shuffle cards. "Don't reinvent the wheel!" There are many discussions about shuffling algorithms in textbooks and on Internet. Here is just a mini review as well as a bit analysis. In following discussion, we are supposed to shuffle n elements of an array.

1. Radom selection

In this algorithm, elements in original array are randomly selected one by one and put into a result array sequentially.

A naïve implementation generates a random number from 1 to n each time as the position of selection in original array. If the element at that position has been selected already, another random number is generated. Obviously it will be hard to find an unselected element when most elements in original array have been selected. A possible optimization is removing the element from original array once it is selected and adjusting the range of random number correspondingly. The drawback is removing an element of array results in the moving of all its following elements. Time complexity of these algorithms is up to O(n^2) and auxiliary space is O(n).

A further optimization is to put the selected element at the last unselected position of original array and move the last unselected element to the position just selected. This results in a swap algorithm that will be discussed later.

2. Random insertion

The idea of this algorithm is similar to that in random selection. Now elements in original array are picked sequentially and put into a result array with random positions. The performance analysis and possible optimization of this algorithm is almost same as that of random selection.

3. Random swap

- *Algorithm 3.1. Swap with two random positions*

The idea of this algorithm is to select and swap two random elements each time until all elements are shuffled to satisfaction. The algorithm is given with below code:

```
for(int i=0; i<SWAP_ROUNDS; i++)
{
        swap(a[rand() % n], a[rand() % n]); //Random range [0, n-1]
}
```

Obviously, auxiliary space of this algorithm is zero and time complexity is linear to the rounds number of swap. It is important to determine an appropriate such number. If shuffling result is measured with the possibility of "A CERTAIN ELEMENT is not selected", the possibility must be small enough. The possibility can be calculated with $p = ((n-2)/n)^m$, where m is rounds number of swap. Given $n = 52$ and $p < 1/1000$, m is about 176. More strictly, if shuffling result is measured with the possibility of "ANY ELEMENT is not selected", m is about 280 (* from Internet).

- *Algorithm 3.2. Swap with one random position*

To ensure all elements are swapped within finite rounds, in this algorithm, one element of swap is picked sequentially and the other is randomly selected. As to the range of random number, it is proven that always select a random number from 1 to n in each round is not perfect, if not incorrect, from the view point of equal possibility for each permutation of all elements (* from Internet). Perfect solution selects the random element in each round from those elements have not been randomly selected and swapped already. The algorithm is given with below two equivalent codes:

```
for(int i=0; i<n; i++)
{
        //Random range [i, n-1]
        swap(a[i], a[rand() % (n-i) + i]);
}
```

```
for(int i=n-1; i>=0; i--)
{
        //Random range [0, i]
        swap(a[i], a[rand() % (i+1)]);
}
```

Time complexity of this algorithm is O(n) and auxiliary space is zero.

It is worth to mention that STL (standard library of C++) provides a function to shuffle elements of a container (std::random_shuffle()), which follows Algorithm 3.2.

Step 4. Coding and testing (v1)

An initial solution based on above analysis and design is implemented with C++ and STL. You may find a complete copy of code (v1) at https://github.com/maoxuli/cards . Please note that this is just a proof-of-concept implementation. Some techniques that may be used in practice, e.g., namespace, smart pointer, inline function, and so on are not considered, as well as unit test and serious testing with test automation tools or profiling tools.

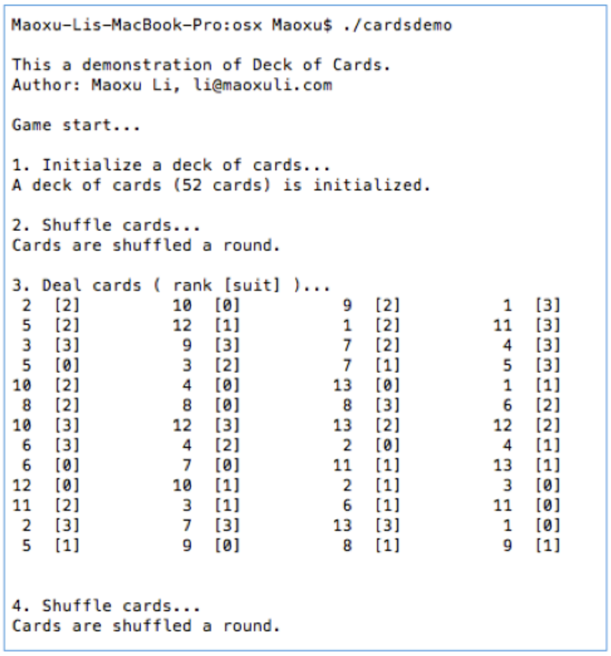
The code is compiled and debugged on Windows XP + VC2010 Express, Mac OS X 10.6 + gcc/g++/gdb, and Debian Linux 6 + gcc/g++/gdb.

- **Demo program (cardsdemo)**

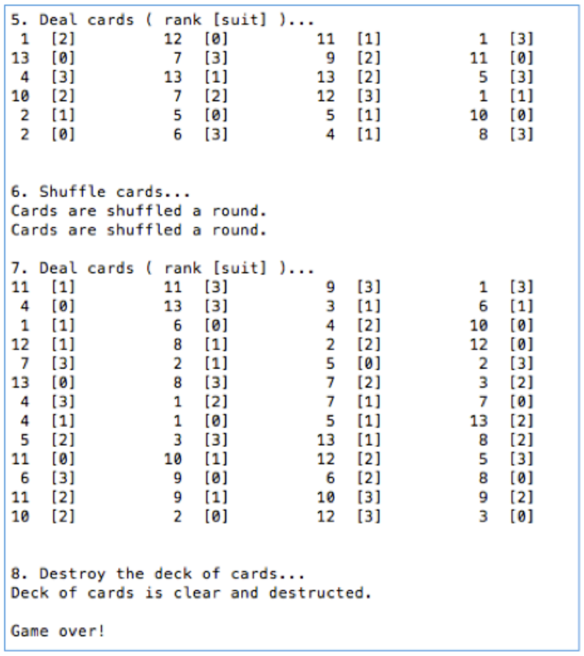
The demo program is implemented to test and demonstrate the design and implementation of class Card and Deck. It simulates below actions in sequence:

1. Open a deck of cards (Instantiate Deck)
2. Shuffle the deck of cards once (Deck::shuffle())
3. Deal cards one by one until finish the deck of cards (Deck::deal_one_card())
4. Shuffle the deck of cards once again
5. Deal a part (not all) of the cards
6. Shuffle the deck of cards twice again
7. Deal cards one by one until finish the deck of cards
8. Discard the deck of cards (destroy the instance of Deck).

Here is a snapshot of screen:



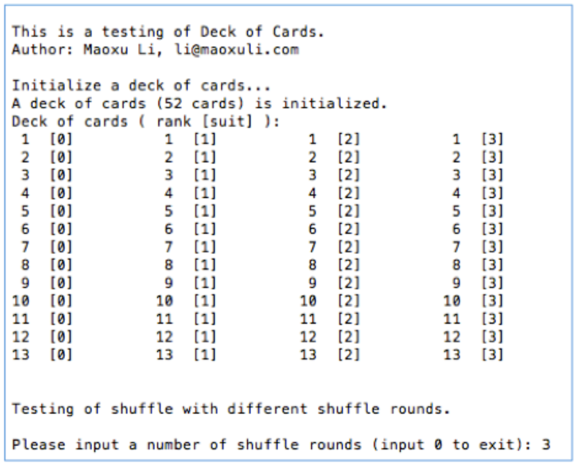
```
Maoxu-Lis-MacBook-Pro:osx Maoxu$ ./cardsdemo

This a demonstration of Deck of Cards.
Author: Maoxu Li, li@maoxuli.com

Game start...

1. Initialize a deck of cards...
A deck of cards (52 cards) is initialized.

2. Shuffle cards...
Cards are shuffled a round.

3. Deal cards ( rank [suit] )...
 2  [2]         10  [0]          9  [2]          1  [3]
 5  [2]         12  [1]          1  [2]         11  [3]
 3  [3]          9  [3]          7  [2]          4  [3]
 5  [0]          3  [2]          7  [1]          5  [3]
10  [2]          4  [0]         13  [0]          1  [1]
 8  [2]          8  [0]          8  [3]          6  [2]
10  [3]         12  [3]         13  [2]         12  [2]
 6  [3]          4  [2]          2  [0]          4  [1]
 6  [0]          7  [0]         11  [1]         13  [1]
12  [0]         10  [1]          2  [1]          3  [0]
11  [2]          3  [1]          6  [1]         11  [0]
 2  [3]          7  [3]         13  [3]          1  [0]
 5  [1]          9  [0]          8  [1]          9  [1]


4. Shuffle cards...
Cards are shuffled a round.
```



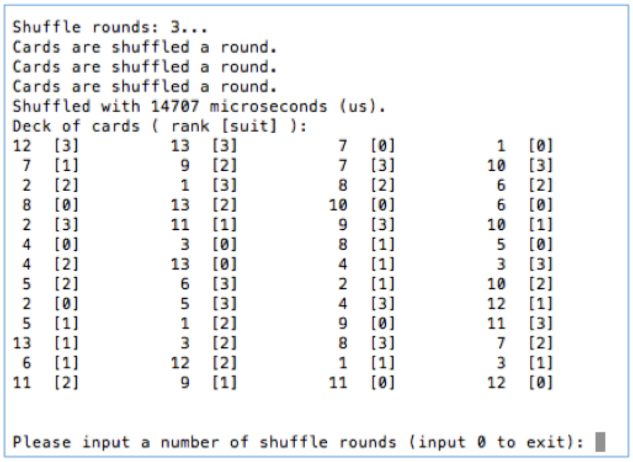
```
5. Deal cards ( rank [suit] )...
 1  [2]         12  [0]         11  [1]          1  [3]
13  [0]          7  [3]          9  [2]         11  [0]
 4  [3]         13  [1]         13  [2]          5  [3]
10  [2]          7  [2]         12  [3]          1  [1]
 2  [1]          5  [0]          5  [1]         10  [0]
 2  [0]          6  [3]          4  [1]          8  [3]


6. Shuffle cards...
Cards are shuffled a round.
Cards are shuffled a round.

7. Deal cards ( rank [suit] )...
11  [1]         11  [3]          9  [3]          1  [3]
 4  [0]         13  [3]          3  [1]          6  [1]
 1  [1]          6  [0]          4  [2]         10  [0]
12  [1]          8  [1]          2  [2]         12  [0]
 7  [3]          2  [1]          5  [0]          2  [3]
13  [0]          8  [3]          7  [2]          3  [2]
 4  [3]          1  [2]          7  [1]          7  [0]
 4  [1]          1  [0]          5  [1]         13  [2]
 5  [2]          3  [3]         13  [1]          8  [2]
11  [0]         10  [1]         12  [2]          5  [3]
 6  [3]          9  [0]          6  [2]          8  [0]
11  [2]          9  [1]         10  [3]          9  [2]
10  [2]          2  [0]         12  [3]          3  [0]


8. Destroy the deck of cards...
Deck of cards is clear and destructed.

Game over!
```

- **Test Program (cardstest)**

The test program is implemented to test the performance of shuffle algorithm in terms of its running speed. Here is some snapshot of screen:



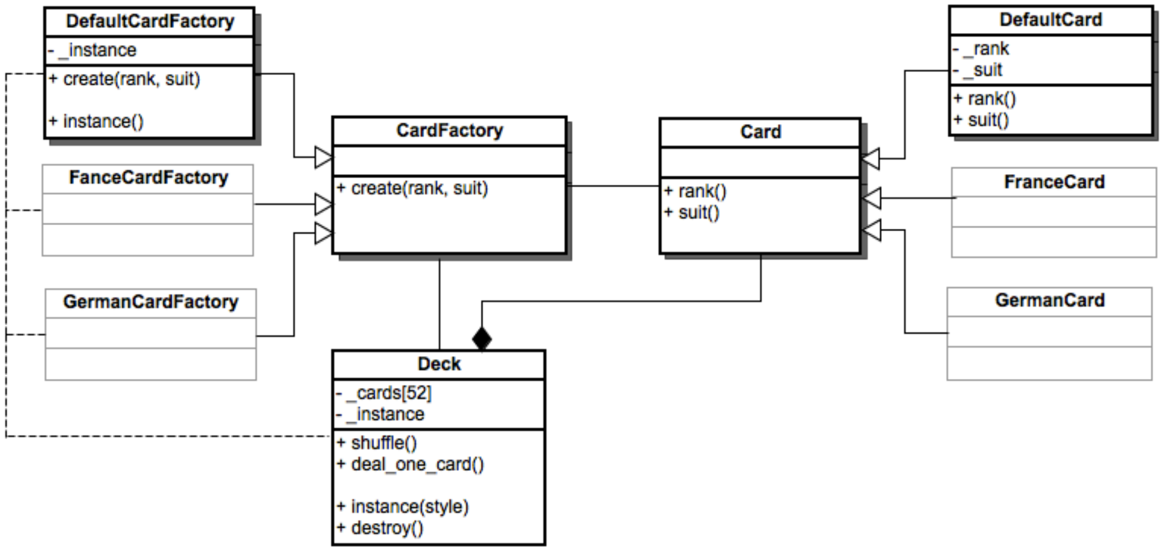
```
This is a testing of Deck of Cards.
Author: Maoxu Li, li@maoxuli.com

Initialize a deck of cards...
A deck of cards (52 cards) is initialized.
Deck of cards ( rank [suit] ):
 1  [0]           1  [1]           1  [2]           1  [3]
 2  [0]           2  [1]           2  [2]           2  [3]
 3  [0]           3  [1]           3  [2]           3  [3]
 4  [0]           4  [1]           4  [2]           4  [3]
 5  [0]           5  [1]           5  [2]           5  [3]
 6  [0]           6  [1]           6  [2]           6  [3]
 7  [0]           7  [1]           7  [2]           7  [3]
 8  [0]           8  [1]           8  [2]           8  [3]
 9  [0]           9  [1]           9  [2]           9  [3]
10  [0]          10  [1]          10  [2]          10  [3]
11  [0]          11  [1]          11  [2]          11  [3]
12  [0]          12  [1]          12  [2]          12  [3]
13  [0]          13  [1]          13  [2]          13  [3]


Testing of shuffle with different shuffle rounds.

Please input a number of shuffle rounds (input 0 to exit): 3
```



```
Shuffle rounds: 3...
Cards are shuffled a round.
Cards are shuffled a round.
Cards are shuffled a round.
Shuffled with 14707 microseconds (us).
Deck of cards ( rank [suit] ):
12  [3]          13  [3]           7  [0]           1  [0]
 7  [1]           9  [2]           7  [3]          10  [3]
 2  [2]           1  [3]           8  [2]           6  [2]
 8  [0]          13  [2]          10  [0]           6  [0]
 2  [3]          11  [1]           9  [3]          10  [1]
 4  [0]           3  [0]           8  [1]           5  [0]
 4  [2]          13  [0]           4  [1]           3  [3]
 5  [2]           6  [3]           2  [1]          10  [2]
 2  [0]           5  [3]           4  [3]          12  [1]
 5  [1]           1  [2]           9  [0]          11  [3]
13  [1]           3  [2]           8  [3]           7  [2]
 6  [1]          12  [2]           1  [1]           3  [1]
11  [2]           9  [1]          11  [0]          12  [0]


Please input a number of shuffle rounds (input 0 to exit):
```

Step 5. Refactoring with design patterns (v2)

Refactoring is to alter the internal design and implementation of software to improve its maintainability and extendibility without changing its behavior and interface. A refined design (shown in figure below) can be reached with further analysis as following.



First, a card in reality is far more complicated than an entity of deck of cards identified with rank and suit. On one hand, a card has many attributes that distinguish a deck of cards from the other. Here we use "style" to denote "what a card looks like". Typically, two decks of cards may have different back images and a "France style" card has totally different representation of suits with a "German style" one. On the other hand, different styles of cards have same interface as to their functions. Design patterns of strategy and factory can be used in expansion and instantiation of such objects as shown in figure. If new styles of cards are implemented with dynamic library, it is easy to define a “plug-in” framework to support the expansion of new card styles.

In addition, it is reasonable that there is a single instance of concrete card factory for each card style. Concrete card factory should be designed with singleton pattern (as is shown in figure). Similarly, as is supposed at the beginning, there is only one deck of cards is in use at any time. A deck of cards can be discarded and another deck of cards is used thereafter. However, the second deck of cards cannot be used before the first one is discarded. Deck class should be designed with singleton pattern and it needs a explicit release interface (as is shown in figure).

This design results in a second implementation (v2) at https://github.com/maoxuli/cards . As requirements of refactoring, class Card and Deck are totally redesigned and recoded while their interfaces stay no change. Hence demo program and test program need only a little modification.