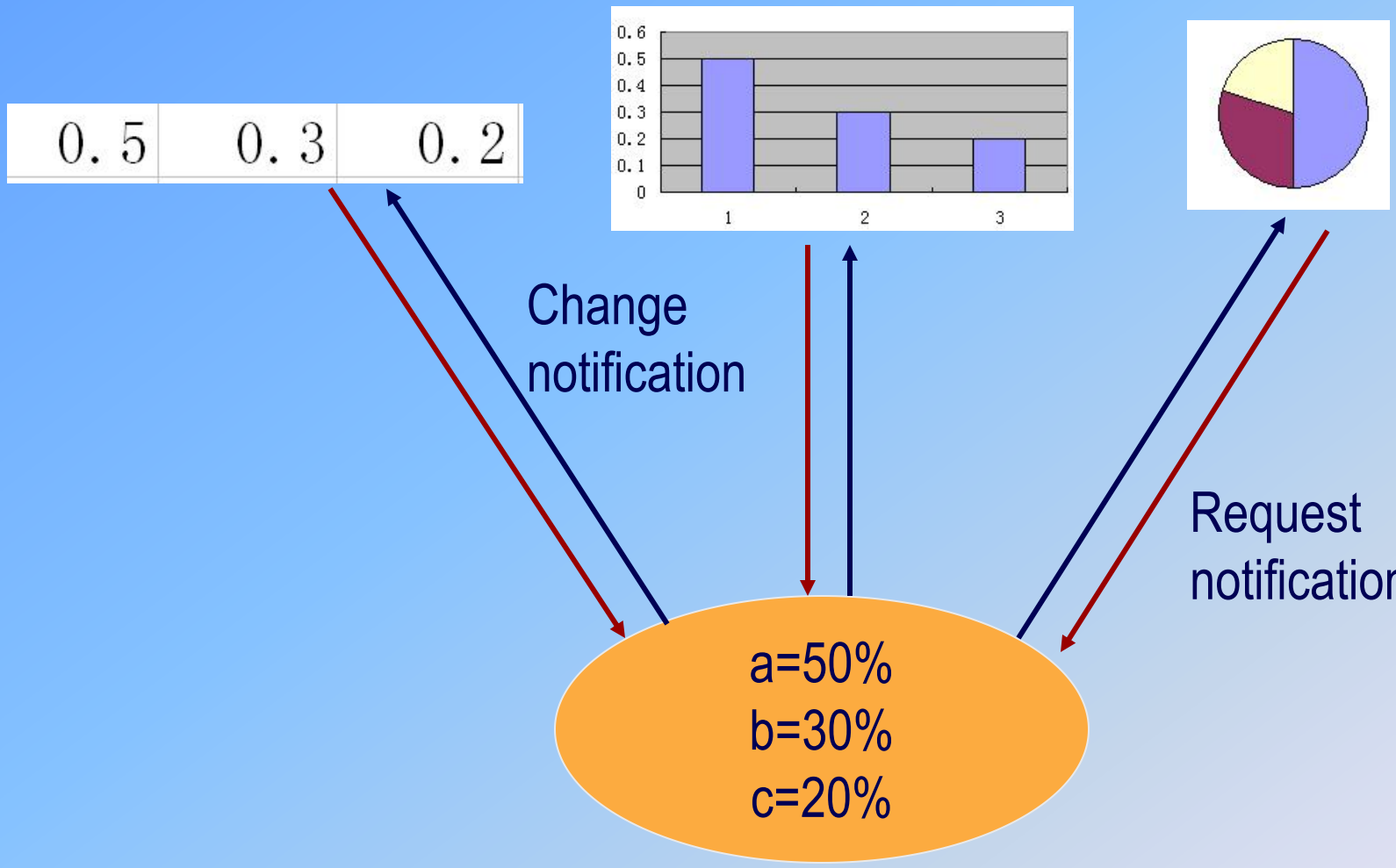
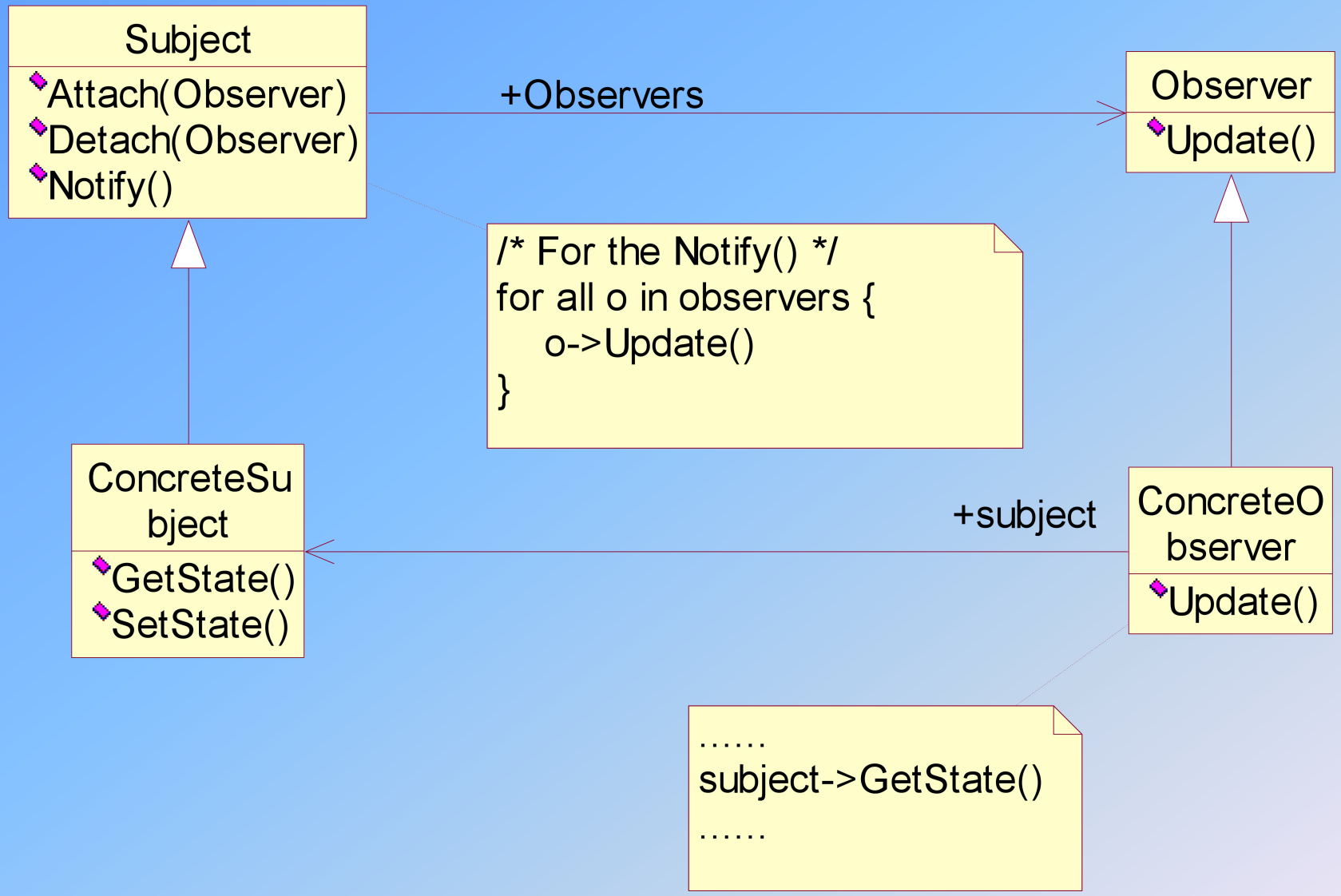
Outline

- Observer pattern
- A case study : sort

Observer

- Motivation
  - Separate data and their presentation
  - Example: excel data visualization


0.5 0.3 0.2
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
Change notification
Request notification
a=50%
b=30%
c=20%



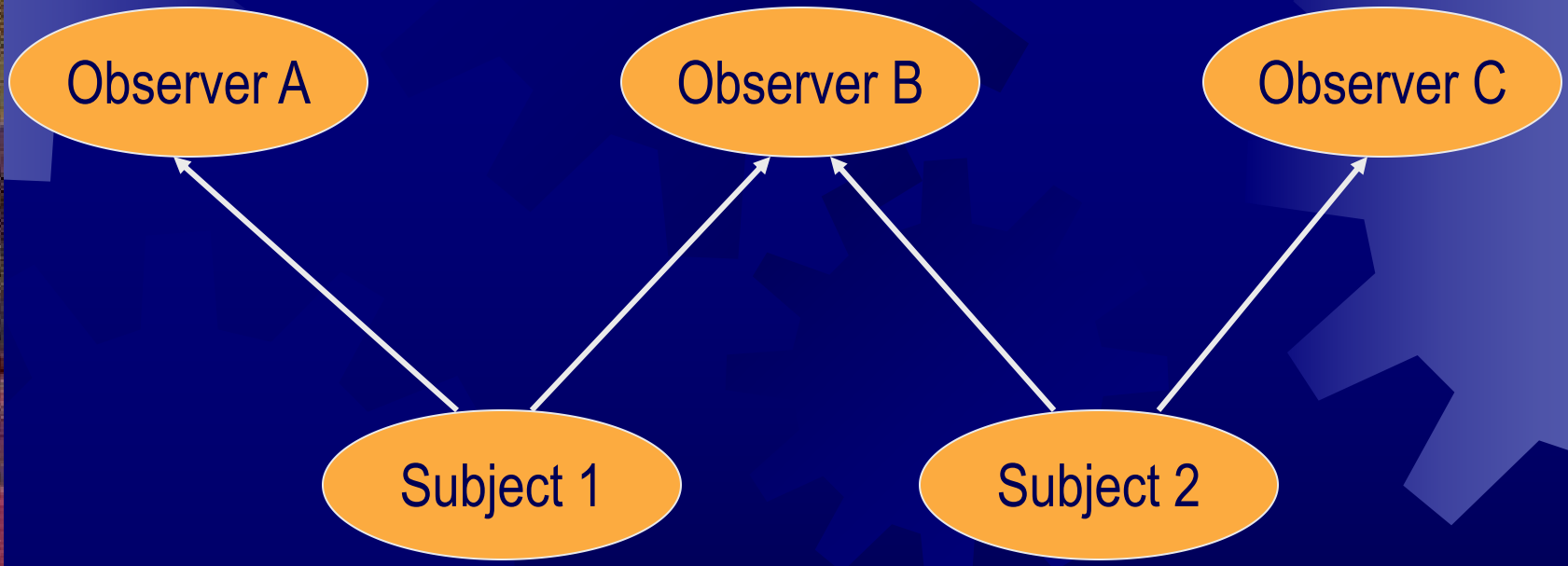
General structure of the Observer pattern

Consequence

- Abstract coupling between subject and Observer
  - Subject and observer can belong to different layers of a software system.
- Support for broadcast communication
  - an observer can ignore the Update message

Implementation

- An observer can observe more than one subject



Implementation

- Who triggers the Update
  - The subject object trigger the Update message immediately.
  - The client trigger the Update message after a series of state changes ➔ efficient
- What has been changed
  - The push model

    All information is included in the argument of the Update method
  - The pull model

Implementation

- Registering observers only for specific events of interest
  - void Subject::Attach(Observer*, Aspect & interest)

案例sort的需求定义

- 概述

  对一个文件的所有行进行排序。

- 支持的运行参数

  -i : 忽略大小写

  -n: 将关键字看作数字，按照数字大小进行排序

  -f k: 关键字从第k个field开始。默认情况为整个一行

  -c k:关键字从第k列开始。默认情况为整个一行

  -p [first | random | median3]：指定 pivot 值

  -r : 降序输出，默认情况为升序输出

案例sort的需求定义(补充)

- 补充说明
  - field的定义为：一行中靠空格、TAB分隔形成若干个field
  - -i 和 –n 同时出现的时候， -n有效
  - -f 和 –c 同时出现的时候， -c有效
  - 如果所指定的 field/列数大于一行总的fields数目/列数，则该行的关键字为空串

案例sort的设计要求

- 性能要求
  - 速度快：需要选择好的排序算法，I/O操作要快
  - 占用尽量少的内存
- 系统扩展性要求
  - 不使用 STL 中的类。
  - 系统中的类应该具有 reusability，细分各个功能，导致多个类，各司其职
  - ➔ 好像显得小题大做，但是教学中我们不能够使用太大的系统作为案例。因此，在设计每个类的时候，**我们要从手头的问题跳出来，使得设计出来的类尽量“通用”（不用修改就可以被其他项目使用！）**，这是理解这个案例的关键

使用singleton管理运行参数

- 运行参数格式举例
  - 一般格式：sort [options] 文件名字
  - sort -n -f 4 -p median3 ..\test_files\student_scores.txt
  - sort -f 4 -p median3 ..\test_files\student_scores.txt
- 设计要求
  - 对运行参数进行集中管理，而不是使用很多零散的全局变量来管理
  - 只能够有一个对象来管理，不能够出现第二个
  - 需要把表示运行参数的数据以及函数封装起来

```
class Options {
  public:
    enum Pivot_Strategy { FIRST, RANDOM, MEDIAN3 };

    static Options * instance( );
    void parse_args (int argc, char **argv);
    bool ignore_case( );
    bool key_is_numeric( );
    bool reverse_output( );
    int  field_offset  ( );
    int  column_offset ( );
    Pivot_Strategy pivot_strat( );
    char * file_name ( );
```

```
private:
    Options ( );
    bool  _ignore_case;
    bool  _key_is_numeric;
    bool  _reverse_output;
    int  _field_offset, _column_offset;
    Pivot_Strategy _pivot_strat;
    char * _file_name;

    static Options  _instance;
};
```

类Input的设计

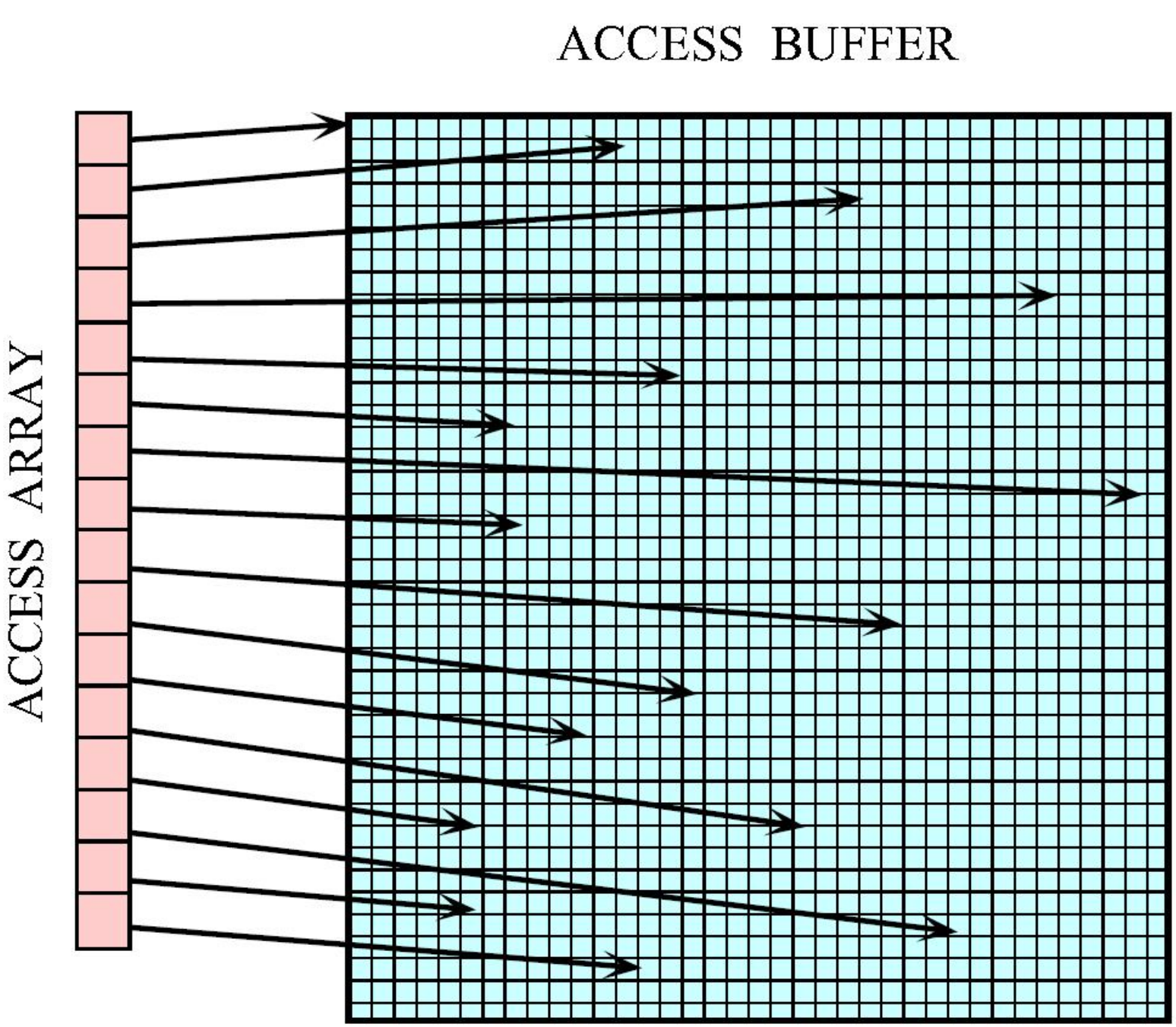
- 目标

  在内存中形成一个个“行”

- 设计要求
  - I/O要快：不能够使用 gets 函数一行行读
  - 避免多次的内存分配：只用一次内存分配

- 解决方案
  - 一次内存分配
  - 一次读入
  - 将“\n”替换为“\0”

  → 参下张幻灯片中的 Access Buffer


ACCESS BUFFER
ACCESS ARRAY

应该具有的类

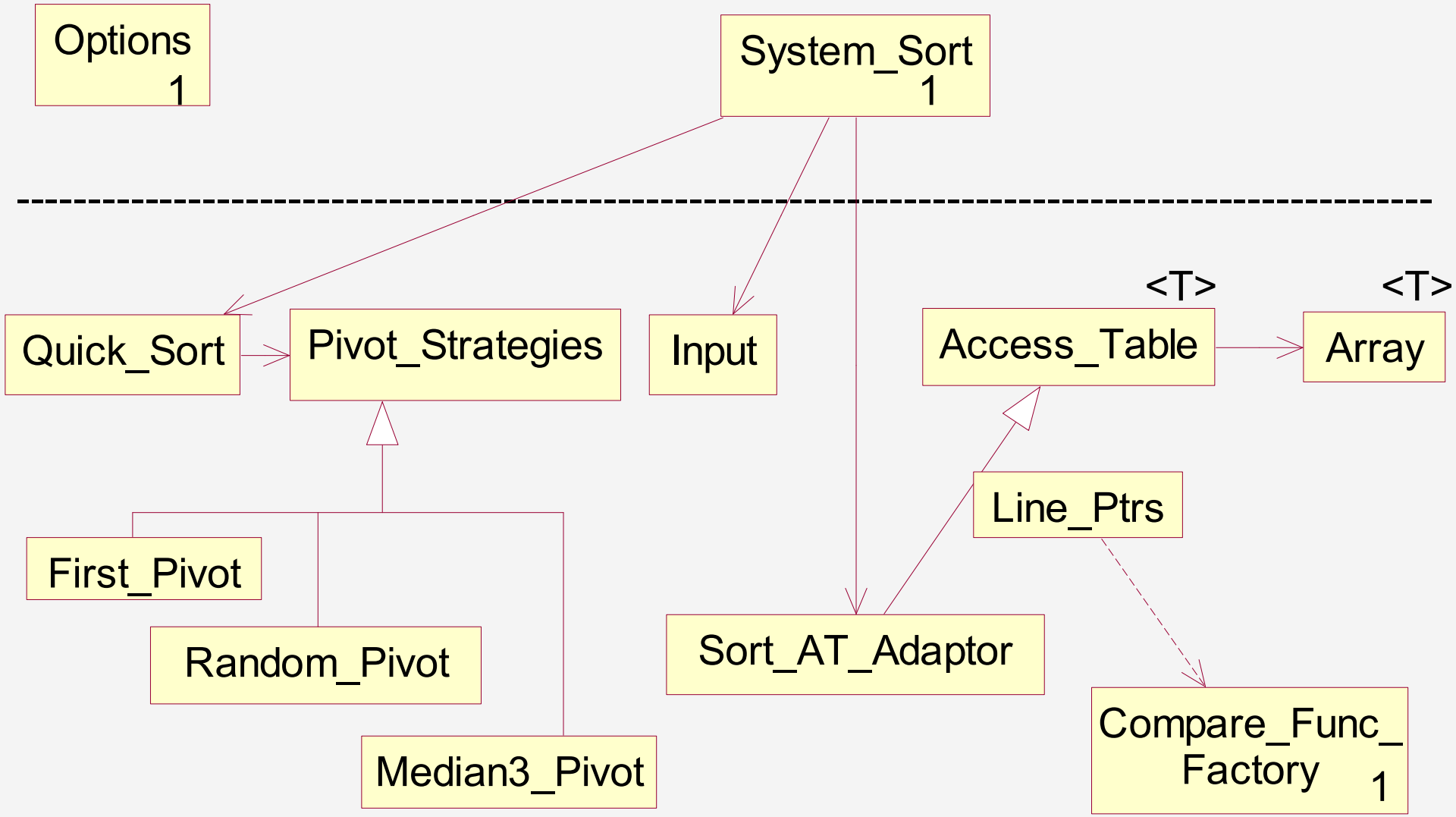
- 类 Input 的功能
  - 负责Access Buffer的分配（但不负责释放）
  - 负责高效地读入数据，形成若干文本行
- 类 Array 的功能
  - 在运行时候才指定长度的数组
- 类 Access_Table 的功能
  - 构建 Access Table（纯虚函数，使得该类能通用）
  - 访问 Access Table中的元素
  - 系统结束时候，释放 Access Buffer

```
class Input  {
  public:
        Input();
        char * read (char* file_name);
        char * buffer( ) const;
        size_t num_lines() const;

  private:
       size_t   _num_lines;
        char *   _buffer;
};
```

```
template <class T>
class Array {
  public:
        void Resize (size_t size = 0){
                _size = size;  _array = new T [ _size ];
        }
        ~Array( ){  delete [ ] _array;    };
        T &operator[ ](size_t index){
                return _array [index];
       }
        size_t size ( ) const{return _size;}
  private:
        T *    _array;
        size_t _size;
};
```

```
template <class T>
class Access_Table {
  public:
      virtual int make_table (char *buffer, size_t num_lines) =0;
      T & element (size_t index){
          return _access_array [ index ];
      };
      size_t length( ) const{ return _access_array.size( );   }
      virtual ~Access_Table ( ){ delete _access_buffer;      };

  protected:
     Array<T>  _access_array;
     char *    _access_buffer;
};
```

最后使用 Façade Pattern封装



排序算法设计：快速排序示意

{ 49　38　65　97　76　13　27　50 }

{ 27　38　13}　49　{76　97　65　50 }

{13}　27　{38}　　{50　65}　76　{97}

50　{65}

{13　27　38　49　50　65　76　97

Pivot 值的选择影响算法效率

{ 10 20 30 40 50 60 70 80 }

10 {20 30 40 50 60 70 80 }

20 {30 40 50 60 70 80}

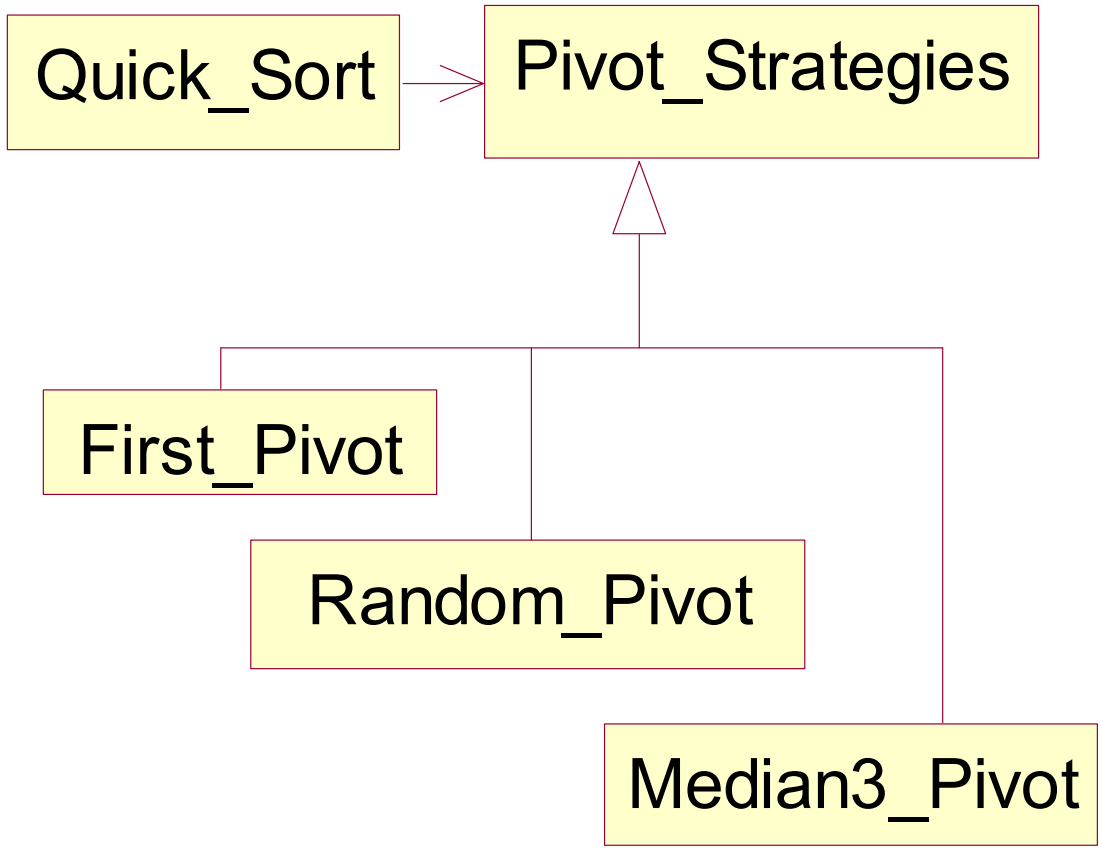
30 {40 50 60 70 80}

.....

“三者取中”来选择 pivot 的值

{ 10 20 30 40 50 60 70 80 }
{ 40 20 30 10 50 60 70 80 }
{ 10 20 30} 40 {50 60 70 80 }
{ 20 10 30} {60 50 70 80 }
{10} 20 {30} {50} 60 {70 80}
{70 80}
70 {80}

使用 Strategy Pattern

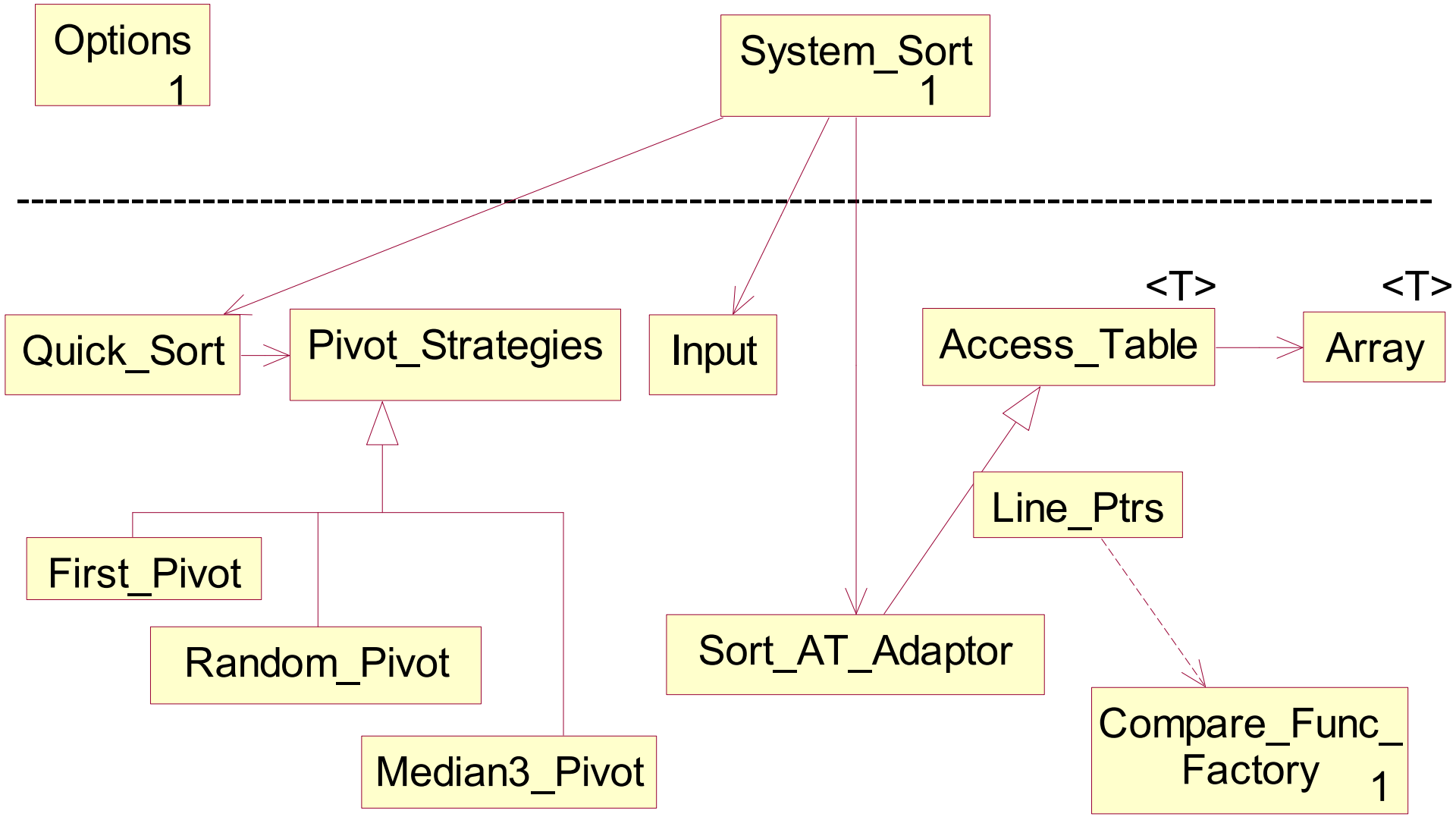


```
class Pivot_Strategies {
  public:
        virtual int get_pivot( Sort_AT_Adaptor & , int lo, int hi) =0;
};
class First_Pivot:public  Pivot_Strategies {
  public:
        virtual int get_pivot( Sort_AT_Adaptor & , int lo, int hi);
};
class Random_Pivot:public  Pivot_Strategies {
  public:
        Random_Pivot();
        virtual int get_pivot( Sort_AT_Adaptor & , int lo, int hi);
};
class Median3_Pivot:public  Pivot_Strategies {
  public:
        virtual int get_pivot( Sort_AT_Adaptor & , int lo, int hi);
};
```

Adaptor pattern --- 使得 System_Sort 能够操作Access_Table

- 由于Access_Table提供的接口和System_Sort所期望的接口不同，所以后者没有办法直接操作它 ➔ 从 Access_Table 派生出子类Sort_AT_Adaptor，提供后者所期望的接口
- 作为一个子类，必须要提供实用的功能了，所以，需要把 Line_Ptrs 作为一个模板参数，提供给 Access_Table。同时，需要实现虚函数 make_table。
- 提供System_Sort 所希望的输出流功能。

最后使用 Façade Pattern封装



```
class Line_Ptrs {
  public:
        // Comparison operator used by sort().
        int operator < (const Line_Ptrs &) const;

        // Beginning of line and key (field or column).
        char *_bol,  *_bok;
};
```

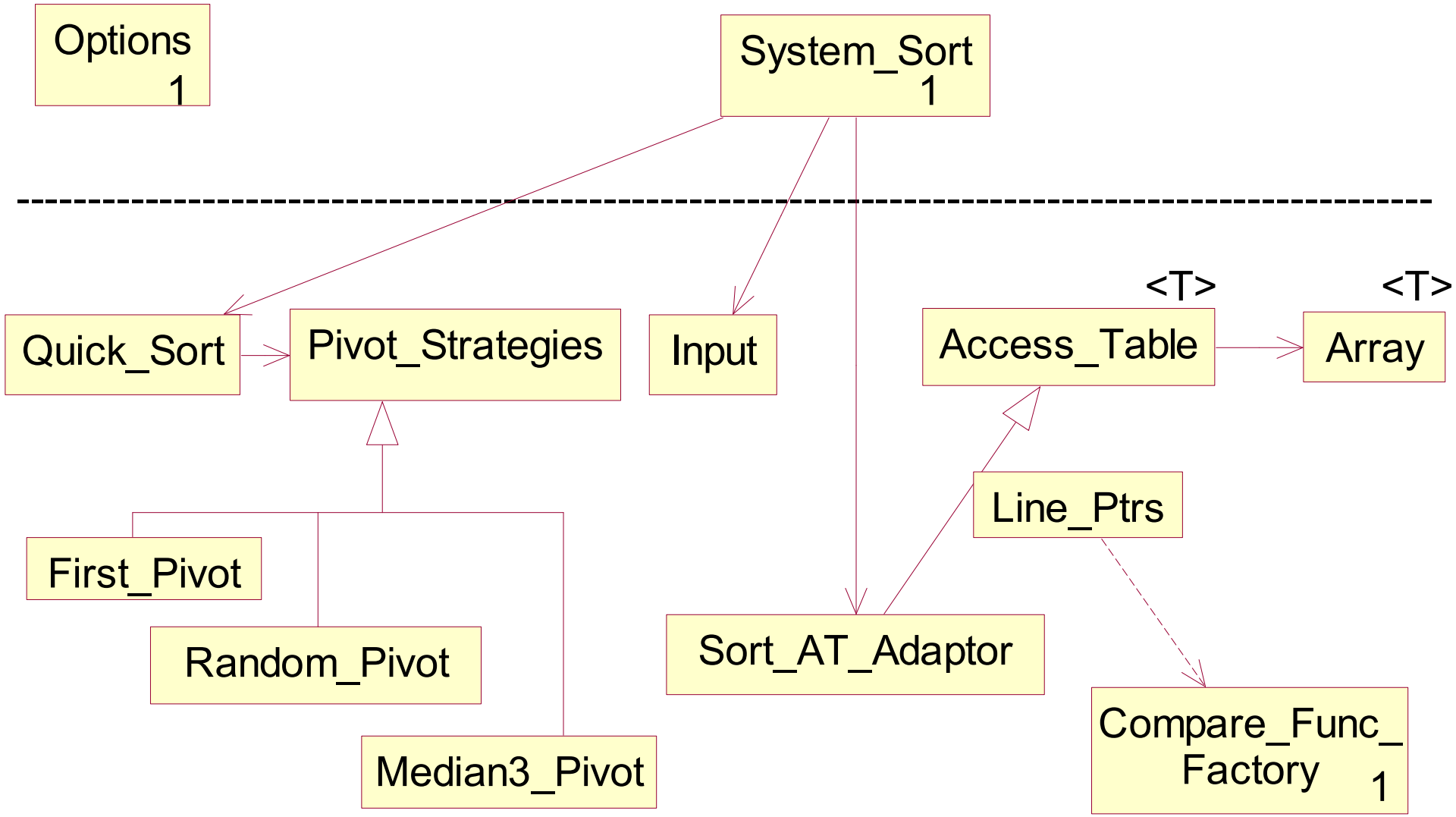
```
class Sort_AT_Adaptor:
                private Access_Table<Line_Ptrs>  {
public:
        virtual int make_table (char *buffer,
                                        size_t num_lines);
         Line_Ptrs &operator[ ] (size_t index){
                return element(index);
         }
        size_t size ( ) const{
                return length();
         }
         friend std::ostream & operator<< (
                 std::ostream & os,
                 Sort_AT_Adaptor & sort_at_adaptor );
};
```

比较函数

```
int Line_Ptrs::operator< (const Line_Ptrs & r) const
 {
    Compare_Func_Factory::Func_Pointer  fp;
    fp = Compare_Func_Factory::instance()
              ->get_func_pointer();
    return ( (*fp) (this->_bok, r._bok ) <= 0 );
 }
```

```
class Compare_Func_Factory
{
public:
        typedef int (*Func_Pointer) (const char*,
                        const char*);
        static Compare_Func_Factory * instance( );
        void set_func_pointer( );
        Func_Pointer get_func_pointer();
private:
        Compare_Func_Factory ( );
        static Compare_Func_Factory  _instance;

        Func_Pointer _func_pointer;
};
```

最后使用 Façade Pattern封装



所使用的设计模式总结

- Singleton
  - Options, System_Sort, Compare_Func_Factory
- Strategies
  - Pivot_Strategies以及其三个子类
- Adaptor
  - Sort_AT_Adaptor
- Façade
  - System_Sort