Object-Oriented Design Case
Studies with Patterns and C++

Douglas C. Schmidt

http://www.cs.wustl.edu/~schmidt/
schmidt@isis-server.isis.vanderbilt.edu

Vanderbilt University
Case Studies Using Patterns

- The following slides describe several case studies using C++ and patterns to build highly extensible software.

- The examples include:
  1. System Sort
     - e.g., Facade, Adapter, Iterator, Singleton, Factory Method, Strategy, Bridge, Double-Checked Locking Optimization
  2. Sort Verifier
     - e.g., Strategy, Factory Method, Facade, Iterator, Singleton

Case Study 1: System Sort

- Develop a general-purpose system sort
- It sorts lines of text from standard input and writes the result to standard output

- e.g., the UNIX system sort

  % sort < big.file > sorted.file

- In the following, we’ll examine the primary forces that shape the design of this application

- For each force, we’ll examine patterns that resolve it
External Behavior of System Sort

- A “line” is a sequence of characters terminated by a newline

- Default ordering is lexicographic by bytes in machine collating sequence

- The ordering is affected globally by the following options:
  - Ignore case (-i)
  - Sort numerically (-n)
  - Sort in reverse (-r)
  - Begin sorting at a specified field (-f)
  - Begin sorting at a specified column (-c)

- Note, our program need not sort files larger than main memory
High-level Forces

• Solution should be both time and space efficient
  - e.g., must use appropriate algorithms and data structures
  - Efficient I/O and memory management are particularly important
  - Our solution uses minimal dynamic binding (to avoid unnecessary overhead)

• Solution should leverage reusable components
  - e.g., iostreams, Array and Stack classes, etc.

• Solution should yield reusable components
  - e.g., efficient input classes, generic sort routines, etc.
Top-level Algorithmic View of the Solution

- Note the use of existing C++ mechanisms like I/O streams

```cpp
// Reusable function
template <class ARRAY> void sort (ARRAY &a);

int main (int argc, char *argv[])
{
    parse_args (argc, argv);
    Input_Array input;

    cin >> input;
    sort (input);
    cout << input;
}
```

- Avoid the *grand mistake* of using top-level algorithmic view to structure the design...
General OOD Solution Approach

- Identify the classes in the application and solution space
  - *e.g.*, stack, array, input class, options, access table, sorts, etc.

- Recognize and apply common design patterns
  - *e.g.*, Singleton, Factory, Adapter, Iterator

- Implement a framework to coordinate components
  - *e.g.*, use C++ classes and parameterized types
C++ Class Components

- Tactical components
  - Stack
    * Used by non-recursive quick sort
  - Array
    * Stores pointers to lines and fields
  - Access_Table
    * Used to store and sort input
  - Input
    * Efficiently reads arbitrary sized input using only 1 dynamic allocation and 1 copy
C++ Class Components

- Strategic components
  - System_Sort
    * Integrates everything...
  - Sort_AT_Adapter
    * Integrates the Array and the Access_Table
  - Options
    * Manages globally visible options
  - Sort
    * e.g., both quicksort and insertion sort
Detailed Format for Solution

- Note the separation of concerns

```cpp
// Prototypes
template <class ARRAY> void sort (ARRAY &a);
void operator >> (istream &,
                 Access_Table<Line_Ptrs> &);
void operator << (ostream &,
                 const Access_Table<Line_Ptrs> &);

int main (int argc, char *argv[])
{
    Options::instance ()->parse_args (argc, argv);

    cin >> System_Sort::instance ()->access_table ();
    sort (System_Sort::instance ()->access_table ());
    cout << System_Sort::instance ()->access_table ();
}
```
Reading Input Efficiently

• **Problem**
  
  – The input to the system sort can be arbitrarily large (e.g., up to 1/2 size of main memory)

• **Forces**

  – To improve performance solution must minimize:

    1. Data copying and data manipulation

    2. Dynamic memory allocation

• **Solution**

  – Create an Input class that reads arbitrary input efficiently
Access Table Format

ACCESS BUFFER

ACCESS ARRAY
The Input Class

- Efficiently reads arbitrary-sized input using only 1 dynamic allocation

```cpp
class Input {
public:
    // Reads from <input> up to <terminator>,
    // replacing <search> with <replace>. Returns
    // pointer to dynamically allocated buffer.
    char *read (istream &input,
                int terminator = EOF,
                int search = '\n',
                int replace = '\0');
    // Number of bytes replaced.
    size_t replaced (void) const;

    // Size of buffer.
    size_t size (void) const;

private:
    // Recursive helper method.
    char *recursive_read (void);

    // ...
};
```
Design Patterns in System Sort

● Facade

  – “Provide a unified interface to a set of interfaces in a subsystem”

  * Facade defines a higher-level interface that makes the subsystem easier to use

  – *e.g.*, sort provides a facade for the complex internal details of efficient sorting

● Adapter

  – “Convert the interface of a class into another interface clients expect”

  * Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces

  – *e.g.*, make Access_Table conform to interfaces expected by sort and iostreams
Design Patterns in System Sort
(cont’d)

• Factory
  – “Centralize the assembly of resources necessary to create an object”
  – e.g., decouple initialization of Line_Ptrs used by Access_Table from their subsequent use

• Bridge
  – “Decouple an abstraction from its implementation so that the two can vary independently”
  – e.g., comparing two lines to determine ordering

• Strategy
  – “Define a family of algorithms, encapsulate each one, and make them interchangeable”
  – e.g., allow flexible pivot selection
Design Patterns in System Sort (cont’d)

- Singleton
  - “Ensure a class has only one instance, and provide a global point of access to it”
  - e.g., provides a single point of access for the system sort facade and for program options

- Double-Checked Locking Optimization
  - “Ensures atomic initialization or access to objects and eliminates unnecessary locking overhead”
  - e.g., allows multiple threads to execute sort

- Iterator
  - “Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation”
— *e.g.*, provides a way to print out the sorted lines without exposing representation or initialization
Sort Algorithm

- For efficiency, two types of sorting algorithms are used:

1. Quicksort
   - Highly time and space efficient sorting arbitrary data
   - $O(n \log n)$ average-case time complexity
   - $O(n^2)$ worst-case time complexity
   - $O(\log n)$ space complexity
   - Optimizations are used to avoid worst-case behavior

2. Insertion sort
   - Highly time and space efficient for sorting “almost ordered” data
   - $O(n^2)$ average- and worst-case time complexity
   - $O(1)$ space complexity
Quicksort Optimizations

1. *Non-recursive*
   - Uses an explicit stack to reduce function call overhead

2. *Median of 3 pivot selection*
   - Reduces probability of worse-case time complexity

3. *Guaranteed* (log n) *space complexity*
   - Always “pushes” larger partition

4. *Insertion sort for small partitions*
   - Insertion sort runs fast on almost sorted data
Selecting a Pivot Value

- **Problem**
  - There are various algorithms for selecting a pivot value
    * e.g., randomization, median of three, etc.

- **Forces**
  - Different input may sort more efficiently using different pivot selection algorithms

- **Solution**
  - Use the *Strategy* pattern to select the pivot selection algorithm
The Strategy Pattern

- **Intent**
  
  - Define a family of algorithms, encapsulate each one, and make them interchangeable

    * Strategy lets the algorithm vary independently from clients that use it

- **This pattern resolves the following forces**

  1. *How to extend the policies for selecting a pivot value without modifying the main quicksort algorithm*

  2. Provide a *one size fits all* interface without forcing a *one size fits all* implementation
Structure of the Strategy Pattern

Context

context_interface()

Strategy

algorithm_interface()

Concrete Strategy A

algorithm_interface()

Concrete Strategy B

algorithm_interface()

Concrete Strategy C

algorithm_interface()
Using the Strategy Pattern

quick_sort

Pivot Strategy
get_pivot()

pivot_strat->get_pivot (array, lo, hi)

Select First

Random

Median of Three
Implementing the Strategy Pattern

- ARRAY is the particular “context”

```cpp
template <class ARRAY>
void sort (ARRAY &array)
{
    Pivot<ARRAY> *pivot_strat = Pivot<ARRAY>::make_pivot
        (Options::instance ()->pivot_strat ());

    quick_sort (array, pivot_strat);
}

template <class ARRAY, class PIVOT_STRAT>
quick_sort (ARRAY &array, PIVOT_STRAT *pivot_strat)
{
    for (; ; ) {
        ARRAY::TYPE pivot; // typename ARRAY::TYPE pivot...

        pivot = pivot_strat->get_pivot (array, lo, hi);

        // Partition array[lo, hi] relative to pivot...
    }
}
Devising a Simple Sort Interface

• **Problem**
  
  – Although the implementation of the *sort* function is complex, the interface should be simple to use

• **Key forces**
  
  – Complex interface are hard to use, error prone, and discourage extensibility and reuse
  
  – Conceptually, sorting only makes a few assumptions about the “array” it sorts
    
    * e.g., supports `operator[]` methods, size, and element `TYPE`
    
  – We don’t want to arbitrarily limit types of arrays we can sort

• **Solution**
  
  – Use the *Facade* and *Adapter* patterns to simplify the sort program
Facade Pattern

• **Intent**

  – Provide a unified interface to a set of interfaces in a subsystem

  * Facade defines a higher-level interface that makes the subsystem easier to use

• This pattern resolves the following forces:

  1. Simplifies the sort interface

     – *e.g.*, only need to support `operator[]` and `size` methods, and element `TYPE`

  2. Allows the implementation to be efficient and arbitrarily complex without affecting clients
Structure of the Facade Pattern

**EXTERNALLY**

**VISIBLE**

**HIDDEN**

- Facade

- [Diagram]

  - EXTERNALLY
  - VISIBLE
  - HIDDEN
Using the Facade Pattern

- EXTERNALLY
  - VISIBLE

- HIDDEN

Sort

Quick Sort

Stack

Insert Sort
The Adapter Pattern

- **Intent**
  
  - “Convert the interface of a class into another interface clients expect”

  * Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces

- This pattern resolves the following forces:

  1. How to transparently integrate the Access_Table with the sort routine

  2. How to transparently integrate the Access_Table with the C++ iostream operators
Structure of the Adapter Pattern

1: request()

client

Target
request()

2: specific_request()

Adapter
request()

Adaptee
specific_request()
Using the Adapter Pattern

1: ARRAY::TYPE t = array[i]

typedef Line_Ptrs TYPE
make_table()
operator[]
size()

Sort_AT_Adapter

sort

"conforms to"

Access_Table

make_table()
length()
element()
Dynamic Array

- Defines a variable-sized array for use by the Access_Table

```cpp
template <class T>
class Array
{
    public:
        typedef T TYPE; // Type "trait"

        Array (size_t size = 0);
        int init (size_t size);
        T &operator[](size_t index);
        size_t size (void) const;
        // ...

    private:
        T *array_;
        size_t size_;
};
```
The Access_Table Class

- Efficiently maps indices onto elements in the data buffer

```cpp
template <class T>
class Access_Table
{

public:
    // Factory Method for initializing Access_Table.
    virtual int make_table (size_t num_lines,
                            char *buffer) = 0;

    // Release buffer memory.
    virtual ~Access_Table (void) { delete [] buffer_; }

    // Retrieve reference to <indexth> element.
    T &element (size_t index) {
        return access_array_[index];
    }

    // Length of the access_array.
    size_t length (void) const {
        return access_array_.size ();
    }

protected:
    Array<T> access_array_; // Access table is array of T.
    char *buffer_; // Hold the data buffer.
};
```
The Sort_AT_Adapter Class

- Adapts the Access_Table to conform to the ARRAY interface expected by sort

```cpp
struct Line_Ptrs {
    // Comparison operator used by sort().
    int operator< (const Line_Ptrs &);

    // Beginning of line and field/column.
    char *bol_, *bof_;
};

class Sort_AT_Adapter :
    // Note the use of the "Class form" of the Adapter
private Access_Table<Line_Ptrs> {
public:
    virtual int make_table (size_t num_lines, char *buffer);

    typedef Line_Ptrs TYPE; // Type "trait".

    // These methods adapt Access_Table methods...
    T &operator[] (size_t index) {
        return element (index);
    }

    size_t size (void) const { return length (); } 
};
```
Centralizing Option Processing

● **Problem**
  
  – Command-line options must be global to many parts of the sort program

● **Key forces**
  
  – Unrestricted use of global variables increases system coupling and can violate encapsulation

  – Initialization of static objects in C++ can be problematic

● **Solution**
  
  – Use the *Singleton* pattern to centralize option processing
Singleton Pattern

- **Intent**
  
  - “Ensure a class has only one instance, and provide a global point of access to it”

- This pattern resolves the following forces:
  
  1. Localizes the creation and use of “global” variables to well-defined objects
  2. Preserves encapsulation
  3. Ensures initialization is done after program has started and only on first use
  4. Allow transparent subclassing of Singleton implementation
Structure of the Singleton Pattern

```c
if (unique_instance_ == 0)
    unique_instance_ = new Singleton;
return unique_instance_;
```

Singleton

- `static instance()`
- `singleton_operation()`
- `get_singleton_data()`
- `static unique_instance_`
Using the Singleton Pattern

```cpp
if (unique_instance_ == 0)
    unique_instance_ = new Options;
return unique_instance_;
```

<table>
<thead>
<tr>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>static instance()</td>
</tr>
<tr>
<td>bool enabled()</td>
</tr>
<tr>
<td>field_offset()</td>
</tr>
<tr>
<td>static unique_instance_options_</td>
</tr>
</tbody>
</table>
Options Class

- This manages globally visible options

```cpp
class Options
{
public:
    static Options *instance (void);
    void parse_args (int argc, char *argv[]);

    // These options are stored in octal order
    // so that we can use them as bitmasks!
    enum Option { FOLD = 01, NUMERIC = 02,
        REVERSE = 04, NORMAL = 010 };
    enum Pivot_Strategy { MEDIAN, RANDOM, FIRST };

    bool enabled (Option o);

    int field_offset (void); // Offset from BOL.
    Pivot_Strategy pivot_strat (void);
    int (*compare) (const char *l, const char *r);

protected:
    Options (void); // Ensure Singleton.

    u_long options_; // Maintains options bitmask...
    int field_offset_;
    static Options *instance_; // Singleton.
};
```
Using the Options Class

- The following is the comparison operator used by sort

```c
int Line_Ptrs::operator< (const Line_Ptrs &rhs) {
    Options *options = Options::instance ();

    if (options->enabled (Options::NORMAL))
        return strcmp (this->bof_, rhs.bof_) < 0;

    else if (options->enabled (Options::FOLD))
        return strcasecmp (this->bof_, rhs.bof_) < 0;

    else
        // assert (options->enabled (Options::NUMERIC));
        return numcmp (this->bof_, rhs.bof_) < 0;
}
```
 Efficiently Avoiding Race Conditions for Singleton Initialization

• Problem
  – A multi-threaded program might have execute multiple copies of sort in different threads

• Key forces
  – Subtle race conditions can cause Singletons to be created multiple times
  – Locking every access to a Singleton can be too costly

• Solution
  – Use the Double-Checked Locking Optimization pattern to efficiently avoid race conditions when initialization Singletons
The Double-Checked Locking Optimization Pattern

• Intent
  
  – Ensures atomic initialization or access to objects and eliminates unnecessary locking overhead

• This pattern resolves the following forces:
  
  1. Ensures atomic initialization or access to objects, regardless of thread scheduling order

  2. Keeps locking overhead to a minimum

    – e.g., only lock on first access, rather than for the entire Singleton instance() method
Structure of the Double-Checked Locking Optimization Pattern

```c
if (unique_instance_ == NULL) {
    mutex_.acquire();
    if (unique_instance_ == NULL)
        unique_instance_ = new Singleton;
    mutex_.release();
}
return unique_instance_;```

Singleton

- static instance()
- static unique_instance_

Mutex
Using the Double-Checked Locking Optimization Pattern

- Uses the Adapter pattern to turn ordinary classes into Singletons optimized automatically with the Double-Checked Locking Optimization pattern

```cpp
template <class TYPE, class LOCK>
class Singleton {
public:
    static TYPE *instance (void);
protected:
    static TYPE *instance_;
    static LOCK lock_{
};

    template <class TYPE, class LOCK> TYPE *
Singleton<TYPE, LOCK>::instance (void) {
    // Perform the Double-Check.
    if (instance_ == 0) {
        Guard<LOCK> mon (lock_);
        if (instance_ == 0) instance_ = new TYPE;
    }
    return instance_;
}
```
Simplifying Comparisons

• **Problem**
  
  – The comparison operator shown above is somewhat complex

• **Forces**
  
  – It’s better to determine the type of comparison operation during the initialization phase
  
  – But the interface shouldn’t change

• **Solution**
  
  – Use the *Bridge pattern* to separate interface from implementation
The Bridge Pattern

- **Intent**
  
  - “Decouple an abstraction from its implementation so that the two can vary independently”

- This pattern resolves the following forces that arise when building extensible software

  1. *How to provide a stable, uniform interface that is both closed and open, i.e.*,

     - *Closed* to prevent direct code changes

     - *Open* to allow extensibility

  2. *How to simplify the Line_Ptrs::operator< implementation*
Structure of the Bridge Pattern

Abstraction

\[ \text{method}() \]

Implementor

\[ \text{method_impl()} \]

Concrete Implementor A

\[ \text{method_impl()} \]

Concrete Implementor B

\[ \text{method_impl()} \]
Using the Bridge Pattern
Using the Bridge Pattern

- The following is the comparison operator used by `sort`

```cpp
int Line_Ptrs::operator<(const Line_Ptrs &rhs) {
    return (*Options::instance ()->compare)
        (bof_, rhs.bof_);
}
```

- This solution is much more concise

- However, there's an extra level of function call indirection...  
  - Which is equivalent to a virtual function call
Initializing the Comparison Operator

● Problem

- How does the `compare` pointer-to-method get assigned?

```c
int (*compare) (const char *left, const char *right);
```

● Forces

- There are many different choices for `compare`, depending on which options are enabled

- We only want to worry about initialization details in one place

- Initialization details may change over time

- We’d like to do as much work up front to reduce overhead later on

● Solution

- Use a `Factory` pattern to initialize the comparison operator
The Factory Pattern

• **Intent**

  – “Centralize the assembly of resources necessary to create an object”
  
    * Decouple object creation from object use by localizing creation knowledge

• **This pattern resolves the following forces:**

  – Decouple initialization of the `compare` operator from its subsequent use

  – Makes it easier to change comparison policies later on

    * *e.g.*, adding new command-line options
Structure of the Factory Pattern

Factory

make_product()

creates

Product

Product product = ...

return product
Using of the Factory Pattern for Comparisons

Options

parse_args()

creates

initialize compare

Compare Function
Code for Using the Factory Pattern

- The following initialization is done after command-line options are parsed

```cpp
Options::parse_args (int argc, char *argv[]) {
  // ...
  if (this->enabled (Options::NORMAL))
    this->compare = &strcmp;
  else if (this->enabled (Options::FOLD))
    this->compare = &strcasecmp;
  else if (this->enabled (Options::NUMERIC))
    this->compare = &numcmp;
  // ...

  int numcmp (const char *s1, const char * s2) {
    double d1 = strtod (s1, 0), d2 = strtod (s2, 0);

    if (d1 < d2) return -1;
    else if (d1 > d2) return 1;
    else // if (d1 == d2)
      return 0;
  }
```
Initializing the Access_Table

• Problem
  – One of the nastiest parts of the whole system sort program is initializing the Access_Table

• Key forces
  – We don’t want initialization details to affect subsequent processing
  – Makes it easier to change initialization policies later on
    * e.g., using the Access_Table in non-sort applications

• Solution
  – Use the Factory Method pattern to initialize the Access_Table
Factory Method Pattern

- **Intent**

  - Define an interface for creating an object, but let subclasses decide which class to instantiate

    * Factory Method lets a class defer instantiation to subclasses

- **This pattern resolves the following forces:**

  - Decouple initialization of the Access_Table from its subsequent use

  - Improves subsequent performance by pre-caching beginning of each field and line

  - Makes it easier to change initialization policies later on

    * *e.g.*, adding new command-line options
Structure of the Factory Method Pattern

**Creator**

```cpp
factory_method() = 0
make_product()
```

```
Product *product = factory_method()
return product
```

**Concrete Creator**

```cpp
factory_method()
```

```
return new Concrete_Product
```

**Product**

**Concrete Product**

CREATES

CREATES
Using the Factory Method Pattern for Access_Table Initialization

```
make_table() = 0
```

```
// initialize the table
```

[Diagram of Access_Table, Sort_AT_Adapter, Line_Ptrs connections]
Using the Factory Method Pattern for the Sort_AT_Adapter

• The following iostream Adapter initializes the Sort_AT_Adapter access table

```cpp
template <class T>
void operator>>(istream &is, Access_Table<T> &access_table)
{
    Input input;

    // Read entire stdin into buffer.
    char *buffer = input.read (is);

    // Determine number of lines.
    size_t num_lines = input.replaced ();

    // Factory Method initializes Access_Table<T>.
    access_table.make_table (num_lines, buffer);
}
```
Implementing the Factory Pattern

- The Access_Table_Factory class has a Factory Method that initializes Sort_AT_Adapter.

```cpp
// Factory Method initializes Access_Table.
int Sort_AT_Adapter::make_table (size_t num_lines,
                                char *buffer)
{
    // Array assignment op.
    this->access_array_.resize (num_lines);
    this->buffer_ = buffer; // Obtain ownership.

    size_t count = 0;

    // Iterate through the buffer and determine
    // where the beginning of lines and fields
    // must go.

    for (Line_Ptrs_Iter iter (buffer, num_lines);
         iter.is_done () == 0;
         iter.next ())
    {
        Line_Ptrs line_ptr = iter.current_element ();
        this->access_array_[count++] = line_ptr;
    }
}
```
Initializing the Access_Table with Input Buffer

• Problem
  – We’d like to initialize the Access_Table without having to know the input buffer is represented

• Key force
  – Representation details can often be decoupled from accessing each item in a container or collection

• Solution
  – Use the Iterator pattern to scan through the buffer
*Iterator Pattern*

- **Intent**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

- The Iterator pattern provides a way to initialize the Access_Table without knowing how the buffer is represented:

  ```c
  Line_Ptrs_Iter::Line_Ptrs_Iter
  (char *buffer, size_t num_lines);

  Line_Ptrs
  Line_Ptrs_Iter::current_element (void)
  {
    Line_Ptrs lp;

    // Determine beginning of next line and next field...
    lp.bol_ = // ..... 
    lp.bof_ = // .....  

    return lp;
  }
  ```
Iterator Pattern (cont’d)

- The Iterator pattern also provides a way to print out the sorted lines without exposing representation

```cpp
template <class T>
void operator<<(ostream &os, const Access_Table<T> &at)
{
    if (Options::instance()->enabled (Options::REVERSE))
        for (size_t i = at.size (); i > 0; i--)
            os << at[i - 1];

    else
        for (size_t i = 0; i < at.size (); i++)
            os << at[i];
}
```

- Note that STL is heavily based on iterators
Summary of System Sort Case Study

- This case study illustrates using OO techniques to structure a modular, reusable, and highly efficient system.

- Design patterns help to resolve many key forces.

- Performance of our system sort is comparable to existing UNIX system sort.
  - Use of C++ features like parameterized types and inlining minimizes penalty from increased modularity, abstraction, and extensibility.
Case Study 2: Sort Verifier

• Verify whether a sort routine works correctly

  – *i.e.*, output of the sort routine must be an ordered permutation of the original input

• This is useful for checking our system sort routine!

  – The solution is harder than it looks at first glance…

• As before, we’ll examine the key forces and discuss design patterns that resolve the forces
General Form of Solution

- The following is a general use-case for this routine:

```cpp
template <class ARRAY> void sort (ARRAY &a);

template <class ARRAY> int check_sort (const ARRAY &o, const ARRAY &p);

int main (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  Input_Array input;
  Input_Array potential_sort;

  cin >> input;

  copy (input, potential_sort);
  sort (potential_sort);

  if (check_sort (input, potential_sort) == -1)
    cerr << "sort failed" << endl;
  else
    cout << "sort worked" << endl;
}
```
Common Problems

- Several common problems:
  - Sort routine may zero out data
    * though it will appear sorted...;-)
  - Sort routine may fail to sort data
  - Sort routine may erroneously add new values
Forces

* Solution should be both time and space efficient
  
  – *e.g.*, it should not take more time to check than to sort in the first place!
  
  – Also, this routine may be run many times consecutively, which may facilitate certain space optimizations

* We cannot assume the existence of a “correct” sorting algorithm…
  
  – Therefore, to improve the chance that our solution is correct, it must be simpler than writing a correct sorting routine
    
    * Quis costodiet ipsos custodes?*
Forces (cont’d)

- Multiple implementations will be necessary, depending on properties of the data being examined, *e.g.*,

  1. if data values are small (in relation to number of items) and integrals use …

  2. if data has no duplicate values use …

  3. if data has duplicate values use …

- This problem illustrates a simple example of “program families”

  - *i.e.*, we want to reuse as much code and/or design across multiple solutions as possible
Strategies

• Implementations of search structure vary according to data, e.g.,

1. *Range Vector*
   - $O(N)$ time complexity and space efficient for sorting “small” ranges of integral values

2. *Binary Search* (version 1)
   - $O(n \log n)$ time complexity and space efficient but does not handle duplicates

3. *Binary Search* (version 2)
   - $O(n \log n)$ time complexity, but handles duplicates

4. *Hashing*
   - $O(n)$ best/average case, but $O(n^2)$ worst case, handles duplicates, but potentially not as space efficient
General OOD Solution Approach

- Identify the “objects” in the application and solution space
  - e.g., use a search structure ADT organization with member function such as insert and remove

- Recognize common design patterns
  - e.g., Strategy, Template Method, and Factory Method

- Implement a framework to coordinate multiple implementations
  - e.g., use classes, parameterized types, inheritance and dynamic binding
General OOD solution approach (cont’d)

- C++ framework should be amenable to:
  - *Extension and Contraction*
    * May discover better implementations
    * May need to conform to resource constraints
    * May need to work on multiple types of data
  - *Performance Enhancement*
    * May discover better ways to allocate and cache memory
      * Note, improvements should be transparent to existing code…
  - *Portability*
    * May need to run on multiple platforms
High-level Algorithm

- e.g., pseudo code

```cpp
template <class ARRAY>
int check_sort (const ARRAY &original,
                const ARRAY &potential_sort)
{
    Perform basic sanity check to see if the potential_sort is actually in order
    (can also detect duplicates here)

    if basic sanity check succeeds then
        Initialize search structure srchstrct
        for i ← 0 to size − 1 loop
            insert (potential_sort[i])
            into srchstrct
        for i ← 0 to size − 1 loop
            if remove (original[i]) from srchstrct fails then
                return ERROR
            return SUCCESS
        else
            return ERROR
    end if
}
```
### C++ Class Interfaces

- **Search structure base class.**

  ```cpp
template <class T>
  class Search_Struct_Strategy
  {
  public:
    virtual int insert (const T &new_item) = 0;
    virtual int remove (const T &existing_item) = 0;
    virtual ~Search_Struct_Strategy (void) = 0;
  };
  ```

- **Strategy Factory class**

  ```cpp
template <class ARRAY>
  class Search_Strategy
  { 
  public:
    // Singleton method.
    static Search_Strategy *instance (void);

    // Factory Method
    virtual Search_Struct_Strategy<ARRAY::TYPE> * make_strategy (const ARRAY &);
  };
  ```
C++ Class Interfaces (cont’d)

- Strategy subclasses

```cpp
// Note the template specialization
class Range_Vector : public Search_Struct_Strategy<long>
{
    typedef long TYPE; /* ... */
};

template <class ARRAY>
class Binary_Search_Nodups : public Search_Struct_Strategy<ARRAY>
{
    typedef T TYPE; /* ... */
};

template <class ARRAY>
class Binary_Search_Dups : public Search_Struct_Strategy<ARRAY>
{
    typedef T TYPE; /* ... */
};

template <class T>
class Hash_Table : public Search_Struct_Strategy<T>
{
    typedef T TYPE; /* ... */
};
```
Design Patterns in Sort Verifier

- Factory Method
  - “Define an interface for creating an object, but let subclasses decide which class to instantiate”
  
    * Factory Method lets a class defer instantiation to subclasses

- In addition, the Facade, Iterator, Singleton, and Strategy patterns are used
Using the Strategy Pattern

- This pattern extends the strategies for checking if an array is sorted without modifying the `check_sort` algorithm
The Factory Method Pattern

- **Intent**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
    
    * Factory Method lets a class defer instantiation to subclasses

- This pattern resolves the following force:

  1. *How to extend the initialization strategy in the sort verifier transparently*
Structure of the Factory Method Pattern

**Creator**

\[ \text{factory} \_\text{method}() = 0 \]

\[ \text{make} \_\text{product}() \]

Product

Concrete Product

Concrete Creator

\[ \text{factory} \_\text{method}() \]

Product *product = factory_method()

return product

return new Concrete_Product

CREATES
Using the Factory Method Pattern

Search Strategy
make_strategy()

New Search Strategy
make_strategy()

return new New_Search_Struct

Search Struct

New Search Struct

creates
Implementing the check_sort Function

- e.g., C++ code for the sort verification strategy

```cpp
#include <typename>

template <class ARRAY> int check_sort (const ARRAY &orig, const ARRAY &p_sort) {
    if (orig.size () != p_sort.size ())
        return -1;

    auto_ptr < Search_Struct_Strategy<ARRAY::TYPE> > ss =
        Search_Strategy<ARRAY>::instance ()->make_strategy (p_sort);

    for (int i = 0; i < p_sort.size (); i++)
        if (ss->insert (p_sort[i]) == -1)
            return -1;

    for (int i = 0; i < orig.size (); i++)
        if (ss->remove (orig[i]) == -1)
            return -1;

    return 0;
    // auto_ptr's destructor deletes the memory...
}
```
Initializing the Search Structure

- Factory Method

```cpp
template <class ARRAY> Search_Struct_Strategy<ARRAY::TYPE> *
Search_Strategy<ARRAY>::make_strategy
    (const ARRAY &potential_sort)
{
    int duplicates = 0;

    for (size_t i = 1; i < potential_sort.size (); i++)
        if (potential_sort[i] < potential_sort[i - 1])
            return 0;
        else if (potential_sort[i] == potential_sort[i - 1])
            duplicates++;

    if (duplicates == 0)
        return new Binary_Search_Nodups<ARRAY>
            (potential_sort);
    else if (size % 2)
        return new Binary_Search_Dups<ARRAY>
            (potential_sort, duplicates)
    else return new Hash_Table<ARRAY::TYPE>
        (size, &hash_function);
}
```
Specializing the Search Structure for Range Vectors

template <Array<long> > Search_Struct_Strategy<long> * Search_Strategy<Array<long> >::make_strategy
    (const Array<long> &potential_sort)
{
    int duplicates = 0;

    for (size_t i = 1; i < size; i++)
        if (potential_sort[i] < potential_sort[i - 1])
            return 0;
        else if (potential_sort[i] == potential_sort[i - 1])
            duplicates++;

    long range = potential_sort[size - 1] - potential_sort[0];
    if (range <= size)
        return new Range_Vector (potential_sort[0],
            potential_sort[size - 1])
    else if (duplicates == 0)
        return new Binary_Search_Nodups<long>
            (potential_sort);
    else if (size % 2)
        return new Binary_Search_Dups<long>
            (potential_sort, duplicates)
    else return new Hash_Table<long>
            (size, &hash_function);
}
Summary of Sort Verifier Case Study

- The sort verifier illustrates how to use OO techniques to structure a modular, extensible, and efficient solution
  
  - The main processing algorithm is simplified
  
  - The complexity is pushed into the strategy objects and the strategy selection factory
  
  - Adding new solutions does not affect existing code
  
  - The appropriate ADT search structure is selected at run-time based on the Strategy pattern