Case Studies Using Patterns

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

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

Case Study: 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
- 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 (e.g., ASCII)
- 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 (-k)
  - Begin sorting at a specified column (-e)
- Your program need not sort files larger than main memory
High-level Forces

- Solution should be both time & space efficient
  - *e.g.*, must use appropriate algorithms and data structures
  - Efficient I/O & memory management are particularly important
  - Our solution uses minimal dynamic binding (to avoid unnecessary overhead)
- Solution should leverage reusable components
  - *e.g.*, istd::ostreams, Array & 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 <typename ARRAY> void sort (ARRAY &a);

int main (int argc, char *argv[])
{
    parse_args (argc, argv);
    Input input;
    cin >> input;
    sort (input);
    cout << input;
}
```

Top-level Algorithmic View of the Solution (cont’d)

- Avoid the *grand mistake* of using top-level algorithmic view to structure the design . . .
  - Structure the design to resolve the forces!
  - Don’t focus on algorithms *or* data, but instead look at the problem, its participants, & their interactions!

General OOD Solution Approach

- Identify the classes in the application/problem space & solution space
  - *e.g.*, stack, array, input class, options, access table, sorts, *etc.*
- Recognize & apply common design patterns
  - *e.g.*, Singleton, Factory, Adapter, Iterator
- Implement a framework to coordinate components
  - *e.g.*, use C++ classes & parameterized types
**C++ Class Components**

- **Tactical components**
  - Stack
    - Used by non-recursive quick sort
  - Array
    - Stores/sorts pointers to lines & fields
  - Access_TABLE
    - Used to store input
  - Input
    - Efficiently reads arbitrary sized input using only 1 dynamic allocation & 1 copy

- **Strategic components**
  - System_Sort
    - Facade that integrates everything...
  - Sort_AT_Adapter
    - Integrates Array & Access_TABLE
  - Options
    - Manages globally visible options
  - Sort
    - *e.g.*, both quicksort & insertion sort

---

**Detailed Format for Solution**

```cpp
// Prototypes
template <typename ARRAY> void sort (ARRAY &a);
void operator>> (std::istream &, Sort_AT_Adapter &);
void operator<< (std::ostream &, const Sort_AT_Adapter &);

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 & data manipulation
    2. Dynamic memory allocation

- **Solution**
  - Create an `Input` class that reads arbitrary input efficiently

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 dynamically allocated buffer.
    char *read (std::istream &input, int terminator = EOF, int search = '\n', int replace = '\0');
    // Number of bytes replaced.
    size_t replaced () const;
    // Size of buffer.
    size_t size () const;
private:
    // Recursive helper method.
    char *recursive_read ();
    // . . .
};
```

The Input Class (cont’d)

```cpp
char *Input::read (std::istream &i, int t, int s, int r)
{
    // Initialize all the data members...
    return recursive_read ();
}
```

```cpp
char *Input::recursive_read () {
    char buffer[BUFSIZ];
    // 1. Read input one character at a time, performing search/replace until EOF is reached or buffer is full.
    // 1.a If buffer is full, invoke recursive_read() recursively.
    // 1.b If EOF is reached, dynamically allocate chunk large enough to hold entire input
    // 2. On way out of recursion, copy buffer into chunk
    // . . .
}
Design Patterns in the 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()* function 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 & istd::ostreams*

Design Patterns in System Sort (cont’d)

- **Singleton**
  - *Ensure a class has only one instance, & provide a global point of access to it*
    - *e.g., provides a single point of access for the system sort facade & for program options*
- **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 & space efficient sorting arbitrary data
     - O(n log n) average-case time complexity
     - O(n²) worst-case time complexity
     - O(log n) space complexity
     - Optimizations are used to avoid worst-case behavior
  2. **Insertion sort**
     - Highly time & space efficient for sorting “almost ordered” data
     - O(n²) average- & 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, & 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
Using the Strategy Pattern

Implementing the Strategy Pattern

- ARRAY is the particular “context”

```cpp
template <typename ARRAY> 
void sort (ARRAY &array) {  
Pivot_Strategy<ARRAY> *pivot_strat =  
    Pivot_Factory<ARRAY>::make_pivot  
    (Options::instance ()->pivot_strat ());  
    std::auto_ptr <Pivot_Strategy<ARRAY> >  
    holder (pivot_strat);  

    // Ensure exception safety.  
    ARRAY temp = array;  
    quick_sort (temp, pivot_strat);  
    // Destructor of <holder> deletes <pivot_strat>.  
    array = temp;  
}
```

Fixed-size Stack

- Defines a fixed size stack for use with non-recursive quicksort

```cpp
template <typename T, size_t SIZE> 
class Fixed_Stack  
{
    public:  
        bool push (const T &new_item);  
        bool pop (T &item);  
        bool is_empty ();  

    private:  
        T stack_[SIZE];  
        size_t top_;  
};
```
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 & reuse
  - Conceptually, sorting only makes a few assumptions about the “array” it sorts
    * e.g., supports `operator[]` methods, size, & trait `TYPE`
  - We don’t want to arbitrarily limit types of arrays we can sort

- **Solution**
  - Use the *Facade & 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[]` & `size` methods, & element `TYPE`
    2. Allows the implementation to be efficient and arbitrarily complex without affecting clients

Structure of the Facade Pattern

Using the Facade Pattern
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 & 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, & provide a global point of access to it*

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

---

Structure of the Singleton Pattern

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

---

Using the Singleton Pattern

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

---

Options

```
static instance()
bool enabled()
field_offset()
static unique_instance_options
```
Options Class

- This manages globally visible options

class Options
{
    public:
        static Options *instance();
        bool 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 };
    }

#define SET_BIT(WORD, OPTION) (WORD |= OPTION)
#define CLR_BIT(WORD, OPTION) (WORD &= ~OPTION)

bool Options::parse_args (int argc, char *argv[]) {
    for (int c;
         (c = getopt (argc, argv, "nrfs:k:c:t:")) != EOF;
            switch (c) {
            case 'n': {
                CLR_BIT (options_, Options::FOLD);
                CLR_BIT (options_, Options::NORMAL);
                SET_BIT (options_, Options::NUMERIC);
                break;
            }
            // . . .
        }
}

Options Class (cont’d)

bool enabled (Option o);

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

protected:
    Options (); // Ensure Singleton.
    u_long options_; // Maintains options bitmask . . .
    int field_offset_; // Singleton.
    static Options *instance_; // Singleton.

Using the Options Class

- One way to implement sort() comparison operator:

```cpp
int Line_Ptrs::operator< (const Line_Ptrs &rhs) const {
    Options *options = Options::instance();
    if (options->enabled (Options::NORMAL))
        return strcmp (this->bof_, rhs.bof_) < 0;
    else if (options->enabled (Options::NUMERIC))
        return numcmp (this->bof_, rhs.bof_) < 0;
    else // if (options->enabled (Options::FOLD))
        return strcasecmp (this->bof_, rhs.bof_) < 0;
    }
```

- We’ll see another approach later on using Bridge
**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 & open*, i.e.,
     - *Closed* to prevent direct code changes
     - *Open* to allow extensibility
  2. *How to simplify the Line_Ptrs::operator< implementation & reference counting for Access_Table buffer*

**Structure of the Bridge Pattern**

- **Abstraction**
  - method()
- **Implementor**
  - method_impl()
- **Concrete ImplementorA**
  - method_impl()
- **Concrete ImplementorB**
  - method_impl()

**Using the Bridge Pattern**

- **Line_Ptrs**
  - operator<
  - 1: compare()
  - **Options**
    - compare()
    - strcmp()
    - strcasecmp()
    - numcmp()
    - strcasecmpmp()
Using the Bridge Pattern

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

```cpp
int Line_Ptrs::operator<(const Line_Ptrs &rhs) const {
  return (*Options::instance ()->compare)
  (bof_, rhs.bof_) < 0;
}
```

- 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?

```cpp
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 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

```
class client {
  public:
    void request() {
      // Request from Target
    }

  private:
    Adapter& target_; // Adapter
}
```

```
class Target {
  public:
    void request() {
      // Specific request
    }
}
```

```
class Adaptee {
  public:
    void request() {
      // Specific request
    }
}
```
Using the Adapter Pattern

Dynamic Array

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

```cpp
template <typename T>
class Array {
public:
    Array (size_t size = 0); // Constructor
    int init (size_t size); // Initialization
    T &operator[](size_t index); // Array access
    size_t size () const; // Array size
    T *begin () const; // STL iterator methods.
    T *end () const;
private:
    T *array_; // Pointer to array
    size_t size_; // Array size
};
```

The Access_Table Class

- Efficiently maps indices onto elements in the data buffer

```cpp
template <typename T>
class Access_Table {
public:
    // Factory Method for initializing Access_Table.
    virtual int make_table (size_t lines, char *buffer) = 0;
    // Release buffer memory.
    virtual ~Access_Table () = 0;
    T &element (size_t index); // Reference to <indexth> element.
    size_t length () const; // Length of the access_array.
    Array<T> &array (void) const; // Return reference to array.
protected:
    Array<T> access_array_; // Access table is array of T.
    Access_Table_Impl *access_table_impl_; // Ref counted 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 &) const;

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

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

Code for Using the Factory Pattern (cont’d)

- We need to write a `numcmp()` adapter function to conform to the API used by the `compare` pointer-to-function

```cpp
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 & line
  - Makes it easier to change initialization policies later on
    * e.g., adding new command-line options

Structure of the Factory Method Pattern

Using the Factory Method Pattern for Access_Table Initialization

- The following istd::ostream Adapter initializes the Sort_AT_Adapter access table

```cpp
void operator>>(std::istream &is, Sort_AT_Adapter &at)
{
    Input input;
    // Read entire stdin into buffer.
    char *buffer = input.read (is);
    size_t num_lines = input.replaced ();

    // Factory Method initializes Access_Table<>
    at.make_table (num_lines, buffer);
}
```
Implementing the Factory Method 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 & determine
    // where the beginning of lines & 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 C++ Standard Library (STL) is heavily based on the iterator pattern, e.g.,**

```cpp
int main (int argc, char *argv[]) {
    std::vector <std::string> args;
    for (int i = 1; i < argc; ++i) {
        args.push_back (std::string (argv [i]));
    }
    for (std::vector<std::string>::iterator j = args.begin ();
         j != args.end (); ++j)
    cout << (*j) << endl;
}
```
**Iterator Pattern (cont’d)**

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

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

Line_Ptrs Line_Ptrs_Iter::current_element () { Line_Ptrs lp;

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

    return lp;
}
```

**Case Study: 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 & discuss design patterns that resolve the forces

**Summary of System Sort Case Study**

This case study illustrates using OO techniques to structure a modular, reusable, & 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, & extensibility.
General Form of Solution

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

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

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

int main (int argc, char *argv[]) {
    Options::instance ()->parse_args (argc, argv);
    Input original;
    Input potentially_sorted;
    cin >> input;
    std::copy (original.begin (),
               original.end (),
               potentially_sorted.begin ());
    sort (potentially_sorted);
    if (check_sort (original, potentially_sorted) == -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 & 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?
      · (Who shall guard the guardians?)
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) & 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 & space efficient for sorting “small” ranges of integral values
  2. *Binary Search (version 1)*  
     - $O(n \log n)$ time complexity & 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 & solution space
  - *e.g.*, use a *search structure* ADT organization with member function such as *insert* & *remove*
- Recognize common design patterns
  - *e.g.*, Strategy & Factory Method
- Implement a framework to coordinate multiple implementations
  - *e.g.*, use classes, parameterized types, inheritance & dynamic binding

General OOD solution approach (cont’d)

- C++ framework should be amenable to:
  - *Extension & 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 & 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 <typename 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)
```

High-level Algorithm (cont'd)

```cpp
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
        else
            return SUCCESS
    end if
}
```

UML Class Diagram for C++ Solution

C++ Class Interfaces

- Search structure base class.

```cpp
template <typename T>
class Search_Strategy
{
public:
    virtual bool insert (const T &new_item) = 0;
    virtual bool remove (const T &existing_item) = 0;
    virtual ~Search_Strategy () = 0;
};
```
C++ Class interfaces (cont’d)

• Strategy Factory class

```cpp
template<typename ARRAY>
Search_Struct
{
public:
  // Singleton method.
  static Search_Struct<ARRAY> *instance();

  // Factory Method
  virtual Search_Strategy<typename ARRAY::TYPE> *make_strategy(const ARRAY &);
};
```

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, & Strategy patterns are used

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

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

template<typename T>
class Hash_Table :
  public Search_Strategy<T>
{
  typedef typename ARRAY::TYPE TYPE; /* . . . */
};
```
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**
  - `factory_method() = 0`
  - `make_product()`
- **Product**
  - `Product *product = factory_method()`
  - `return product`
- **Concrete Creator**
  - `factory_method()`
  - `return new Concrete_Product`
Implementing the check_sort Function

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

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

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

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

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

    return 0;
}
```

Initializing the Search Structure

- Factory Method

```cpp
template <typename ARRAY>
Search_Strategy<typename ARRAY::TYPE> *
Search_Struct<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 (typeid (potential_sort[0]) == typeid (long)
        && range <= size)
        return new Range_Vector (potential_sort[0],
                                 potential_sort[size - 1])
    else 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<typename ARRAY::TYPE>
            (size, &hash_function);
}
```
Specializing the Search Structure for Range Vectors

```cpp
template <Array<long> > * Search_Struct<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, & efficient solution
  - The main processing algorithm is simplified
  - The complexity is pushed into the strategy objects & 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