

Design Patterns

Douglas C. Schmidt

Washington University, St. Louis

**<http://www.cs.wustl.edu/~schmidt/>
schmidt@cs.wustl.edu**

1

Patterns of Learning

- Solutions to most challenging areas of human endeavor are deeply rooted in patterns
 - In fact, an important goal of education and training is to transmit patterns of learning from generation to generation
- In the following, we'll explore the use of patterns in learning to play chess
- In many ways, learning to develop good software is similar to learning to play good chess
 - Though the consequences of failure are often less dramatic!

2

Becoming a Chess Master

- First learn rules and physical requirements
 - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn principles
 - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- However, to become a master of chess, one must study the games of other masters
 - These games contain patterns that must be understood, memorized, and applied repeatedly
- There are thousands upon thousands of these patterns

3

Becoming a Master Software Designer

- First one learns the rules
 - e.g., the algorithms, data structures and languages of software
- Later, one learns the principles of software design
 - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- But to truly master software design, one must study the designs of other masters
 - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are thousands upon thousands of these patterns

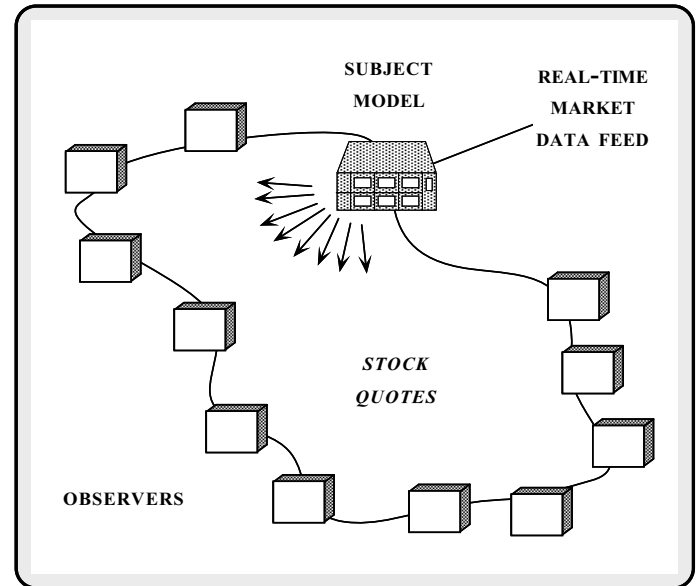
4

Design Patterns

- Design patterns represent *solutions* to *problems* that arise when developing software within a particular *context*
 - i.e., “Patterns == problem/solution pairs in a context”
- Patterns capture the static and dynamic *structure* and *collaboration* among key *participants* in software designs
 - They are particularly useful for articulating how and why to resolve *non-functional forces*
- Patterns facilitate reuse of successful software architectures and designs

5

Example: Stock Quote Service



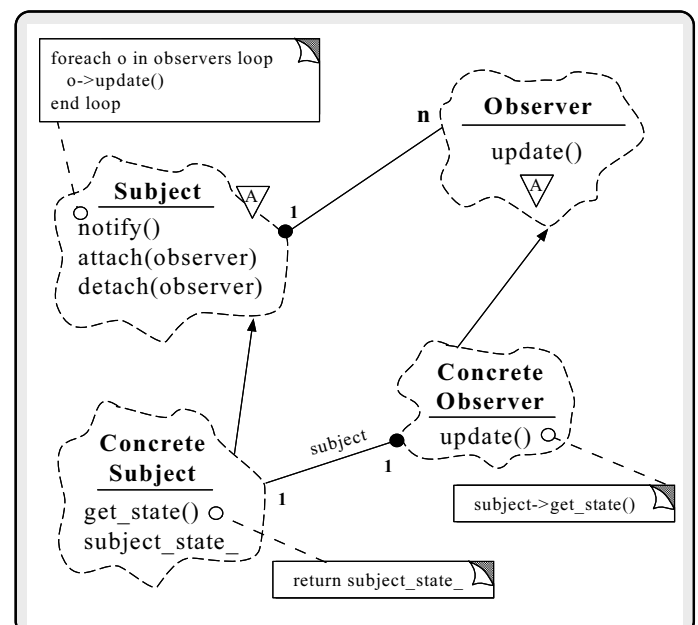
6

Observer Pattern

- *Intent*
 - “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”
- Key forces
 1. There may be many observers
 2. Each observer may react differently to the same notification
 3. The subject should be as decoupled as possible from the observers

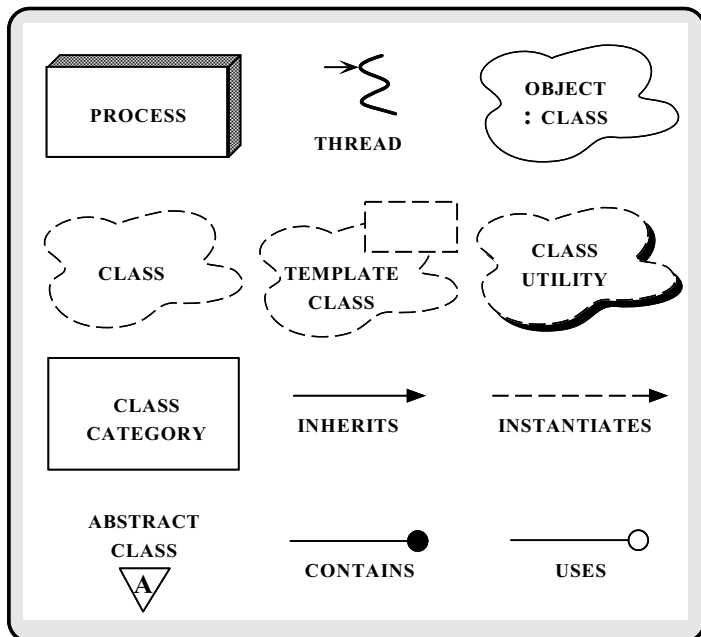
7

Structure of the Observer Pattern



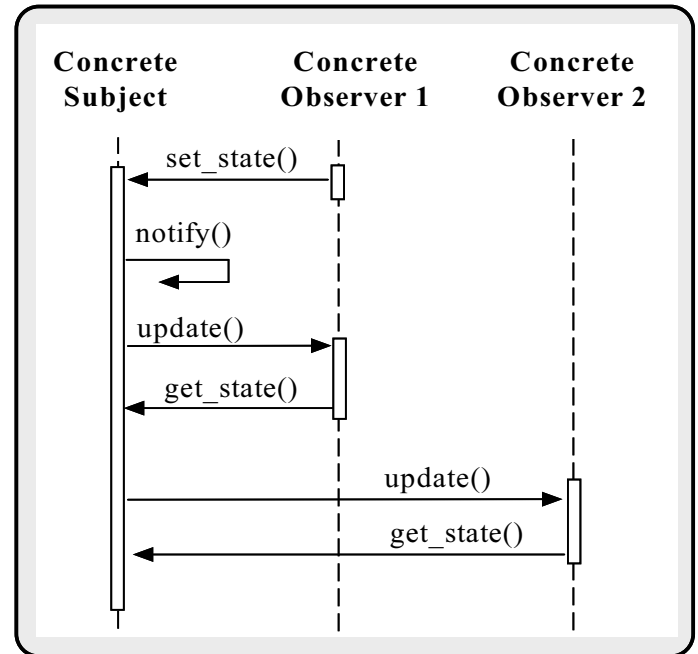
8

Graphical Notation



9

Collaboration in the Observer Pattern



10

Design Pattern Descriptions

- Main parts
 1. *Name*
 2. *Problem and context*
 3. *Force(s) addressed*
 4. *Abstract description of structure and collaborations in solution*
 5. *Positive and negative consequence(s) of use*
- Pattern descriptions are often independent of programming language or implementation details
 - Contrast with frameworks...

11

Challenges Addressed by Patterns

- *Communication of architectural knowledge among developers*
 - Provide a common vocabulary for common design structures
 - Reduce complexity
 - Enhance expressiveness
- *Codify good design*
 - Distill and disseminate experience
 - Aid novices and experts
 - Abstract the design process
- *Accommodate new design paradigms or architectural styles*

12

Challenges Addressed by Patterns (cont'd)

- *Resolve non-functional forces such as reusability, portability, and extensibility*
- *Avoid development traps and pitfalls that are usually learned only by experience*
 - Capture and preserve design information
 - Articulate design decisions concisely
 - Improve documentation
- *Facilitate restructuring/refactoring*
 - Patterns can be organized into systems or families

13

Case Study: System Sort

- The following slides describe a case study using design patterns to develop a general-purpose system sort program
 - This program sorts lines of text from standard input and writes the result to standard output
- 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

14

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)
- Program need not sort files larger than main memory

15

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
 - This solution uses minimal dynamic binding (to avoid 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.

16

General Form of Solution

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

```
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;
}
```

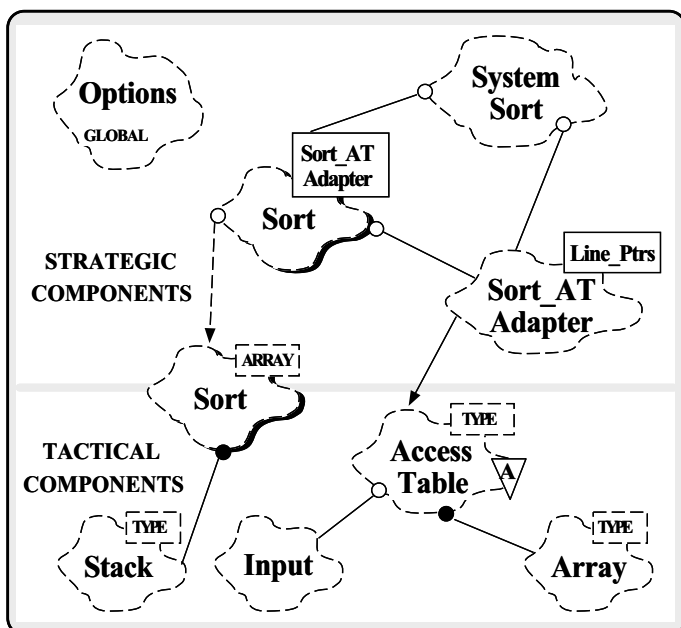
17

General OOD Solution Approach

- Identify the “objects” 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

18

C++ Class Model



19

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

20

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

21

Detailed Format for Solution

- Note the separation of concerns

```
// Prototypes
template <class ARRAY> sort (ARRAY &a);
void operator >> (istream &, Sort_AT_Adapter &);
void operator << (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 ();
}
```

22

Reading Input Efficiently

- *Problem*
 - The input to the system sort can be arbitrarily large (e.g., up to 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

23

The Input Class

- Efficiently reads arbitrary sized input using only 1 dynamic allocation

```
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);

    // ...
};
```

24

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 client expects”
 - ▷ 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`

25

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

26

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 system sort and for program options
- Double-Checked Locking
 - “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

27

Sort Algorithm

- For efficiency, two types of sorting algorithms are used:
 1. *Quicksort*
 - Highly time and space efficient sorting arbitrary data
 - $O(n \lg n)$ average-case time complexity
 - $O(n^2)$ worst-case time complexity
 - $O(\lg n)$ 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

28

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 ($\lg n$) space complexity*

- Always “pushes” larger partition

4. *Insertion sort for small partitions*

- Insertion sort runs fast on almost sorted data

29

The Strategy Pattern

• *Intent*

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

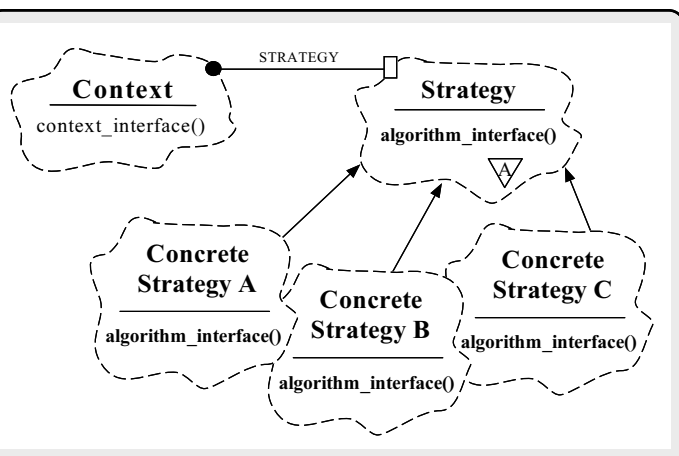
- Strategy lets the algorithm vary independently from clients that use it

• This pattern resolves the following force

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

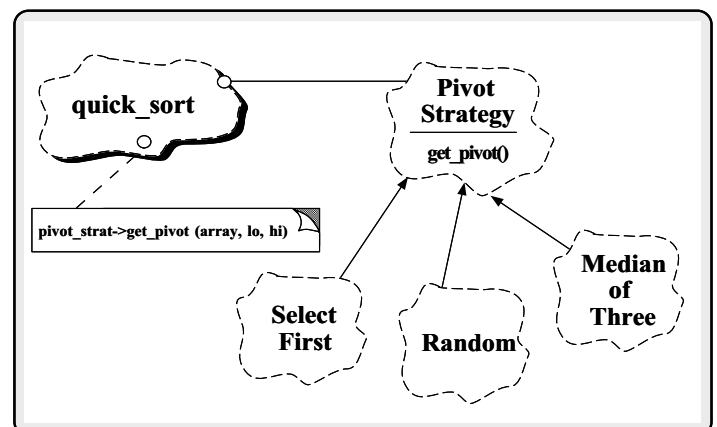
30

Structure of the Strategy Pattern



31

Using the Strategy Pattern



32

Implementing the Strategy Pattern

```
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;

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

        // Partition array[lo, hi] relative to pivot...
    }
}
```

33

Devising a Simple Sort Interface

- *Problem*
 - Although the implementation of the sort function is very 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<`, `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

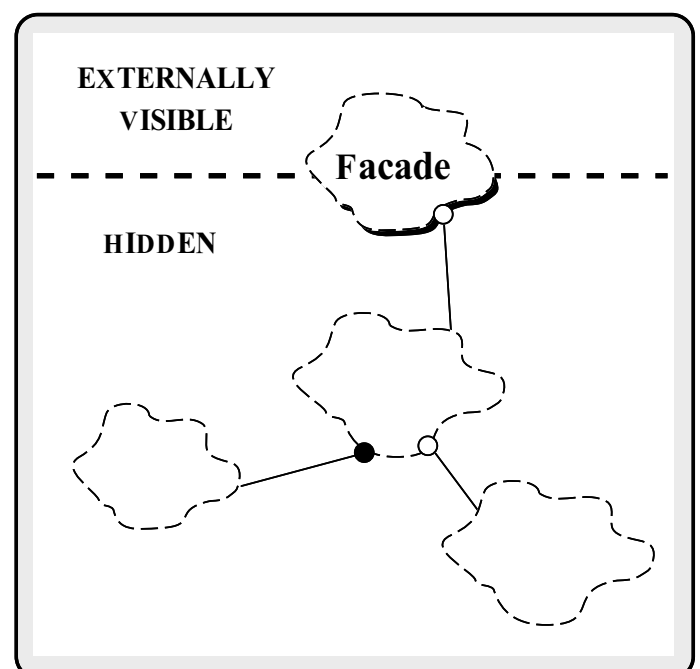
34

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<`, `operator[]` methods, size, and element `TYPE`
 2. Allows the implementation to be efficient and arbitrarily complex without affecting clients

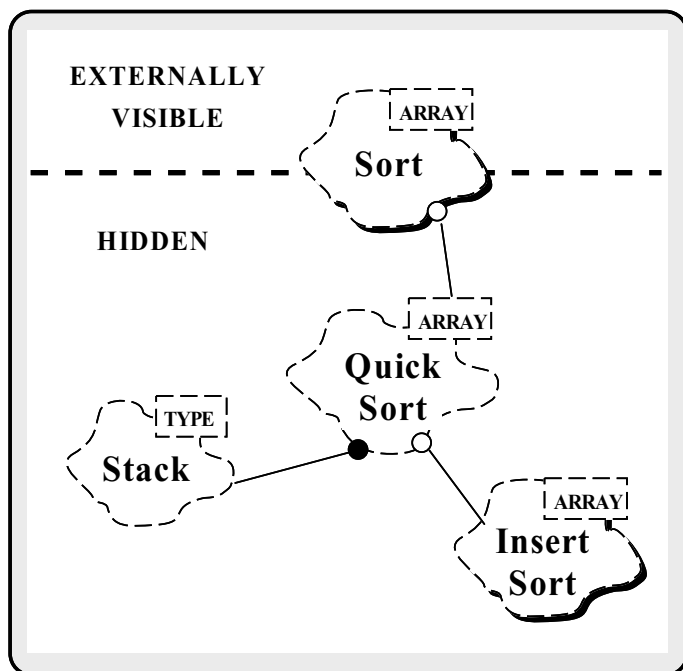
35

Structure of the Facade Pattern



36

Using the Facade Pattern



37

The Adapter Pattern

- *Intent*

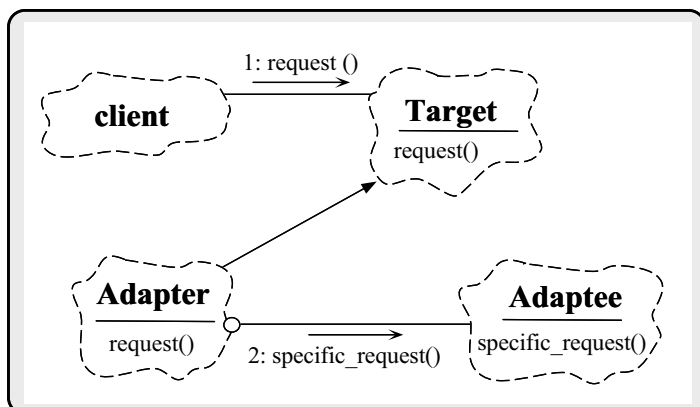
- Convert the interface of a class into another interface client expects
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

- 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

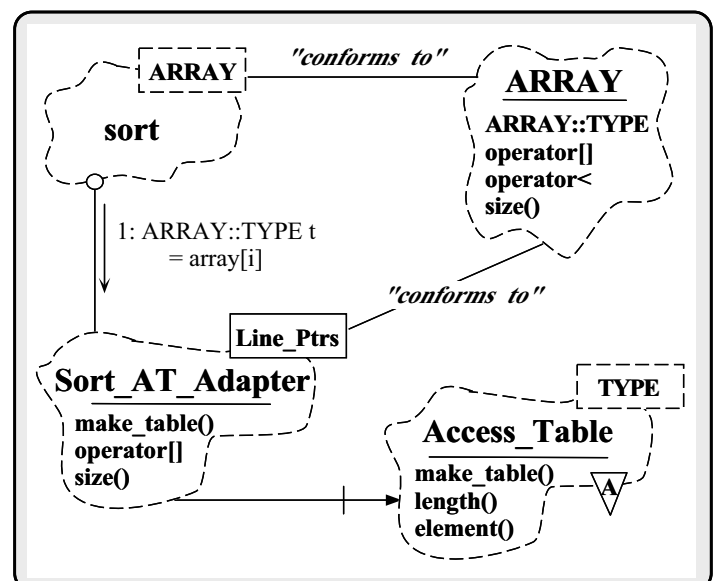
38

Structure of the Adapter Pattern



39

Using the Adapter Pattern



40

Dynamic Array

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

```
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_;
};
```

41

The Access_Table Class

- Efficiently maps indices onto elements in the data buffer.

```
template <class T>
class Access_Table
{
public:
    // Factory Method for initializing Access_Table.
    virtual int make_table (size_t size,
                           char *buffer) = 0;

    // Release buffer memory.
    ~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 an array of T
    char *buffer_; // Hold the data buffer.
};
```

42

The Sort_AT_Adapter Class

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

```
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 Adapter."
    private Access_Table<Line_Ptrs> {
    typedef Line_Ptrs TYPE; // Type "trait".

    T &operator[] (size_t index) {
        return element (index);
    }

    size_t size (void) const { return length (); }

    virtual int make_table (size_t size, char *buffer);
};
```

43

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

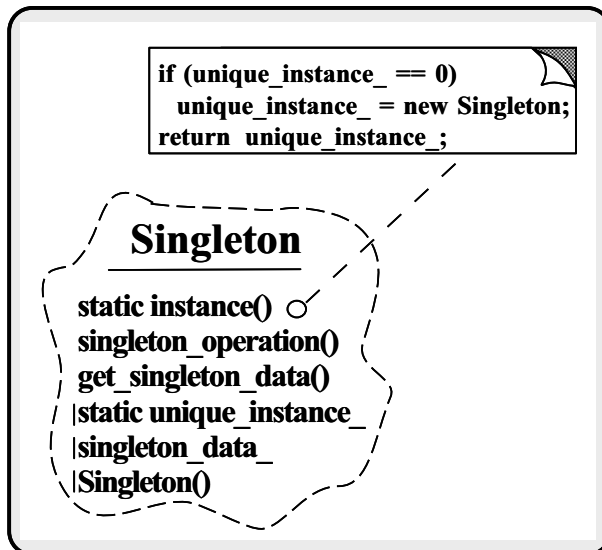
44

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

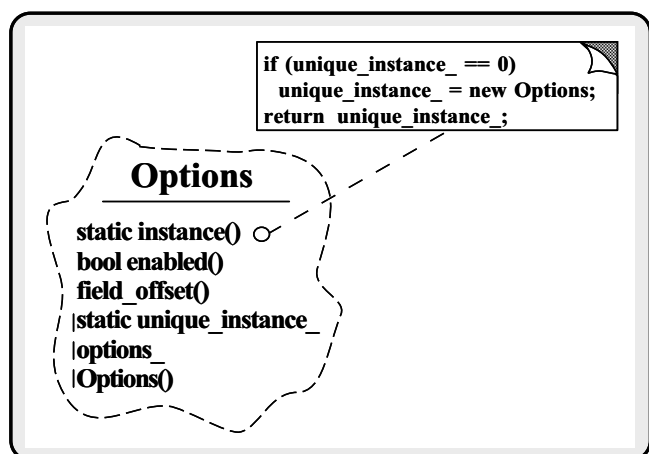
45

Structure of the Singleton Pattern



46

Using the Singleton Pattern



47

Options Class

- This manages globally visible options

```
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_options_;
    static Options *instance_; // Singleton.
};
```

48

Using the Options Class

- The following is the comparison operator used by sort

```
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;
}
```

49

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* pattern to efficiently avoid race conditions when initialization Singletons

50

The Double-Checked Locking Pattern

- *Intent*

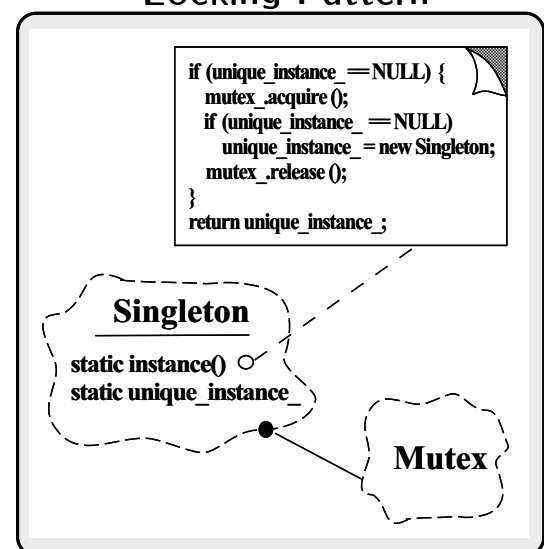
- Ensures atomic initialization or access to objects and eliminates unnecessary locking overhead

- 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

51

Structure of the Double-Checked Locking Pattern



52

Using the Double-Checked Locking Pattern

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

```
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> lock (lock_);
        if (instance_ == 0) instance_ = new TYPE;
    }
    return instance_;
}
```

53

Simplifying Comparisons

- *Problem*

- The comparison operator shown above is somewhat complex

- *Forces*

- The type of comparison operation can be determined during the initialization phase

- *Solution*

- Use the *Bridge pattern* to separate interface from implementation

54

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

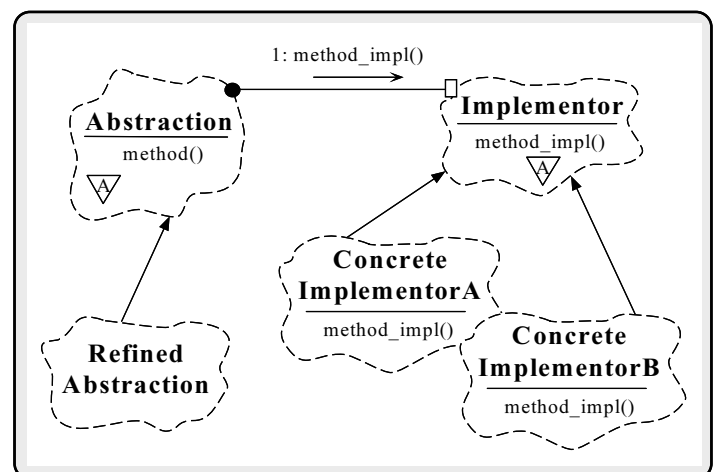
1. *How to provide a stable, uniform interface that is both closed and open, i.e.,*

- *Closed* to prevent direct code changes
- *Open* to allow extensibility

2. *How to simplify the implementation of operator<*

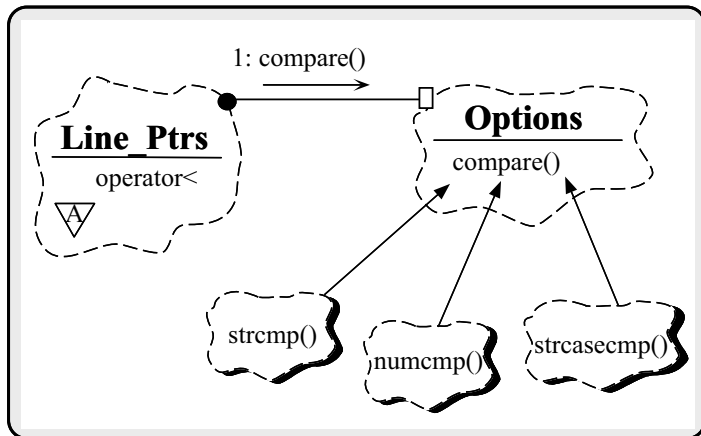
55

Structure of the Bridge Pattern



56

Using the Bridge Pattern



57

Using the Bridge Pattern

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

```

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...

58

Initializing the Comparison Operator

• Problem

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

```
int (*compare) (const char *l, const char *r);
```

• 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

59

The Factory Pattern

• Intent

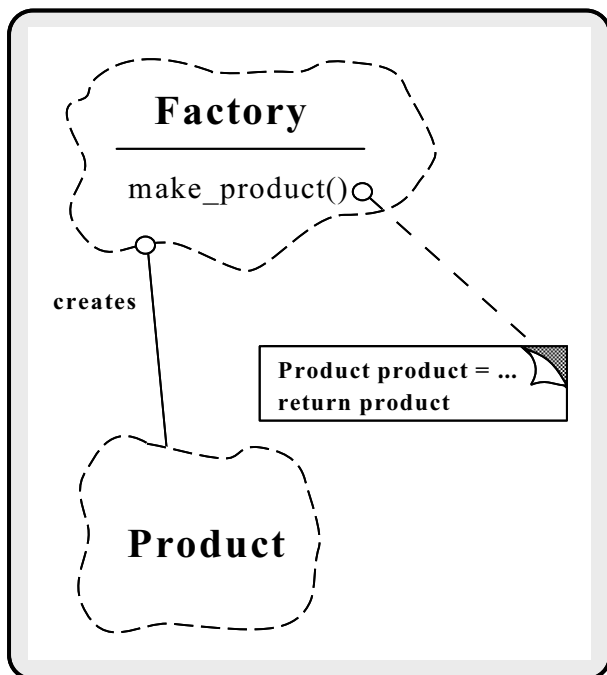
- "Centralize the assembly of resources necessary to create an object"
- Decouple object creation from object use by localizing creation knowledge

• 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

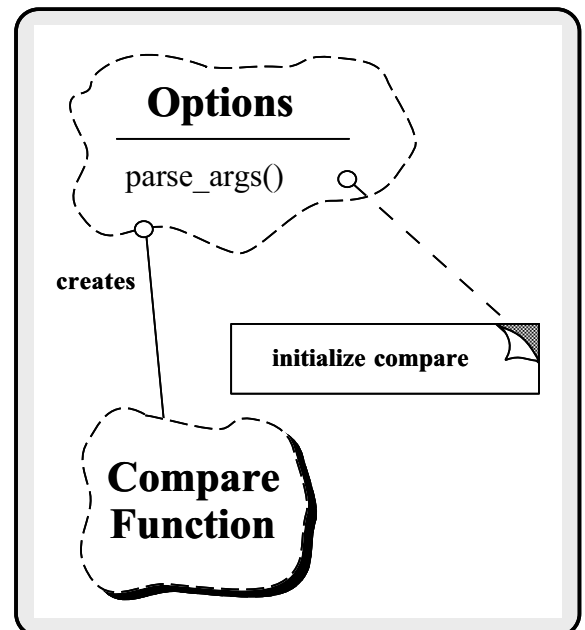
60

Structure of the Factory Pattern



61

Using of the Factory Pattern for Comparisons



62

Code for Using the Factory Pattern

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

```
Options::parse_args (int argc, char *argv[])
{
    // ...

    Options *options = Options::instance ();
    if (options->enabled (Options::NORMAL))
        options->compare = &strcmp;
    else if (options->enabled (Options::FOLD))
        options->compare = &strcasecmp;
    else if (options->enabled (Options::NUMERIC))
        options->compare = &numcmp;

    // ...
}
```

63

Initializing the Access_Table

- *Problem*
 - One of the nastiest parts of the whole 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`

64

Factory Method Pattern

- *Intent*

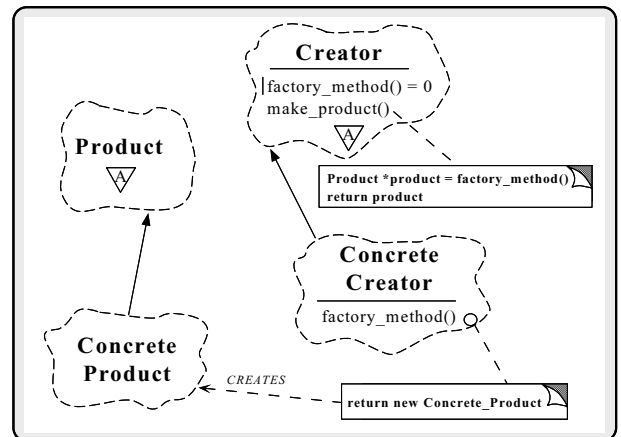
- Define an interface for creating an object, but let subclasses decide which class to instantiate
 - ▷ Factory Method lets a class defer instantiation to subclasses

- 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

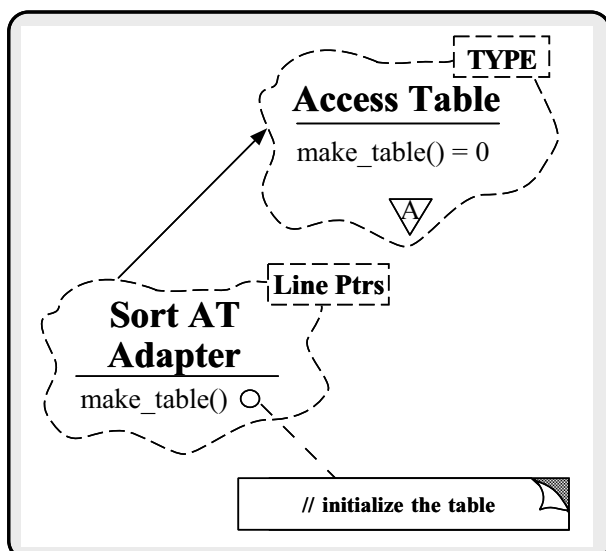
65

Structure of the Factory Method Pattern



66

Using the Factory Method Pattern for `Access_Table` Initialization



67

Using the Factory Method Pattern for the `Sort_AT_Adapter`

- The following `iostream` Adapter initializes the `Sort-"AT"-Adapter`

```

void operator>> (istream &is,
                Sort_AT_Adapter &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 Sort_AT_Adapter.
    access_table.make_table (num_lines, buffer);
}
  
```

68

Implementing the Factory Pattern

- The Access_Table_Factory class has a method that initializes Sort_at_Adapter

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

    Line_Ptrs line_ptr;
    size_t count = 0;

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

    for (Line_Ptrs_Iter iter (buffer, num_lines);
         iter (line_ptr) != -1;
         iter++)
    {
        this->access_array_[count++] = line_ptr;
    }
}
```

69

Initializing the Access_Table with Buffer

- *Problem*

- We'd like to initialize the Access_Table *without* having to know the 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

70

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:

```
Line_Ptrs_Iter::Line_Ptrs_Iter
(char *buffer, size_t num_lines)
{ /* ... */ }

int
Line_Ptrs_Iter::operator () (Line_Ptrs &lp)
{
    // Determine beginning of next line and next field...
    lp.bol_ = // .....
    lp.bof_ = // .....
    return done ? -1 : 0;
}
```

71

Iterator Pattern (cont'd)

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

```
void operator << (ostream &os,
                 const Sort_AT_Adapter &at)
{
    if (Options::instance ()->enabled (Options::REVERSE)) {
        for (size_t i = at.size (); i > 0; i--)
            os << at[i - 1].bol_;
    }
    else {
        for (size_t i = 0; i < at.size (); i++)
            os << at[i].bol_;
    }
}
```

72

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 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

73

Benefits of Design Patterns

- *Design patterns enable large-scale reuse of software architectures*
- *Patterns explicitly capture expert knowledge and design tradeoffs*
- *Patterns help improve developer communication*
- *Patterns help ease the transition to object-oriented technology*

74

Drawbacks to Design Patterns

- *Patterns do not lead to direct code reuse*
- *Patterns are deceptively simple*
- *Teams may suffer from pattern overload*
- *Patterns are validated by experience rather than by testing*
- *Integrating patterns into a software development process is a human-intensive activity*

75

Suggestions for Using Patterns Effectively

- *Do not recast everything as a pattern*
 - Instead, develop strategic domain patterns and reuse existing tactical patterns
- *Institutionalize rewards for developing patterns*
- *Directly involve pattern authors with application developers and domain experts*
- *Clearly document when patterns apply and do not apply*
- *Manage expectations carefully*

76