

**OO Software Engineering,  
Modeling, and Design**

**Douglas C. Schmidt**

**Washington University, St. Louis**

**<http://www.cs.wustl.edu/~schmidt/>  
[schmidt@cs.wustl.edu](mailto:schmidt@cs.wustl.edu)**

**What is Object-Orientation (OO)**

- OO is a set of principles and methods for decomposing, abstracting, and layering system architectures
  - Various notations and tools have evolved to express, “externalize,” and implement an OO architectural design
- An OO design is characterized by structuring the system architecture based on the *objects* (and classes of objects) in the problem domain
  - Rather than the *actions* performed by the system

## Benefits of OO

- OO is an *enabling technology* that enhances key *software quality factors* of a system and its components
  - *i.e.*, OO techniques do not enhance system functionality per se
  - However, they make it more convenient to enhance functionality and resolve non-functional forces
- *Increased software stability and reuse*
  - Systems evolve and functionality changes, but data objects, interfaces, and components relations tend to remain relatively stable over time...
  - ▷ Therefore, basing software structure on objects rather than actions increases reuse and flexibility

## The OO Development Process

- The OO development process supports iterative and incremental system evolution
- There are two distinct levels at which an OO development process operates:
  1. *Macro process*
    - Concerned with long-term development lifecycle issues
  2. *Micro process*
    - Concerned with short-term specification and development of a particular phase of the macro process

## The OO Development Process (cont'd)

- *Macro process*
  - Establish core requirements (conceptualization)
  - Develop a model of desired behavior (analysis)
  - Create an architecture (design)
  - Evolve the implementation (evolution)
  - Manage post delivery evolution (maintenance)
- *Micro process*
  - Identify classes and objects at a given level of abstraction
  - Identify the semantics of these classes and objects
  - Identify the relationships among these classes and objects
  - Specify the interface and then the implementation of these classes and objects

## Elements of OO Methods

- A set of unifying *principles*
  - e.g., encapsulation, decomposition, design patterns, hierarchical relations, deferred decision-making
- A systematic development *process* or *methodology* that leads to the generation of relevant products and artifacts
  - e.g., functional specs, documentation, reviews, estimates, pattern descriptions, code modules
- A *notation* for capturing and reasoning about the strategic and tactical analysis and design decisions and recurring patterns
- *Tools* that support the notation and the development process
  - e.g., programming languages and development environments

## Differences between OOA and OOD

- Object-oriented techniques may be applied to several stages (*e.g.*, *analysis*, *design*, *implementation*, etc.) in the software lifecycle
- OO analysis (OOA) is a process of *discovery*
  - Where the development team models and understands the requirements of the system
- OO design (OOD), in contrast, is primarily a process of *invention* and *adaptation*
  - Where the development team creates the *abstractions* and *mechanisms* necessary to meet the system's behavioral requirements that are determined via analysis

## Differences between OOA and OOD (cont'd)

- It is useful to distinguish between *strategic* and *tactical* analysis and design decisions
  - *Strategic decisions* have broad, sweeping architectural implications
    - ▷ *e.g.*, decision to utilize a relational database, separation of tasks in a client/server application, choice of IPC mechanisms
  - *Tactical decisions* represent essential details that complete the strategic architectural decisions
    - ▷ *e.g.*, the database schema, “type signature” of a member function, particular sorting algorithm, etc.
- Note, the distinction between *strategies* and *tactics* is somewhat fuzzy...

## Differences between OOD and OOP

- It is also useful to distinguish between object-oriented programming (OOP) and object-oriented design (OOD)
  - Each level of abstraction emphasizes different aspects of OO
- OOP tends to be more related to programming language issues
  - e.g., dynamic vs. static typing, compiled vs. interpreted languages, etc.
- OOD tends to be relatively independent of the language used
  - e.g., design patterns help to transcend programming language-centric viewpoints
  - Obviously, the more consistent/related the OOP and OOD techniques, the easier they are to apply in real-life...

## Differences between OOD and OOP (cont'd)

### • Basic Definitions

#### 1. *Object-Oriented Programming*

- The construction of software systems as structured collections of *Abstract Data Type* (ADT)s, combined with *inheritance* and *dynamic binding*

#### 2. *Object-Oriented Design*

- A method for decomposing software architectures based on the *objects* and *classes of objects* every system or subsystem manipulates
  - ▷ Rather than “the” function it is meant to ensure

## Object-Oriented Programming Topics

- Object-oriented programming features and techniques include
  - *Data abstraction and information hiding*
  - *Active (rather than passive) types and objects*
  - *Genericity/parameterized types*
  - *Inheritance and dynamic binding*
  - *Programming by contract*
  - *Assertions and exception handling*
- These OOP features and techniques help improve software quality factors
  - *e.g., correctness, reusability, extensibility, reliability, etc.*

## Object-Oriented Design Topics

- Object-oriented design concepts include:
  - *Design patterns*
  - *Decomposition/Composition*
  - *Abstraction*
    - ▷ *Modularity*
    - ▷ *Information Hiding and encapsulation*
    - ▷ *Separating Policy and Mechanism*
    - ▷ *Subset Identification and Program Families*
  - *Hierarchy*
    - ▷ *Virtual Machines*
- These design concepts help to manage software system complexity and improve key *software quality factors*

## Software Quality Factors

- Object-oriented techniques enhance key *external* and *internal* software quality factors, e.g.,
  1. External (visible to end-users)
    - (a) *Correctness*
    - (b) *Robustness and reliability*
    - (c) *Performance*
    - (d) *Compatibility (via standard/uniform interfaces)*
  2. Internal (visible to developers)
    - (a) *Modularity*
    - (b) *Flexibility/Extensibility*
    - (c) *Reusability*
    - (d) *Type-safety*

## Principles of Decomposition/Composition

- Decompose so as to limit the effect of any one design decision on the rest of the system
  - Remember, anything that permeates the system will be expensive to change...
  - Since design decisions transcend execution time, modules may not correspond to execution steps...
- Components should be specified by all information needed to use the module and *nothing more*
- Try to compose the system by reusing existing components if possible

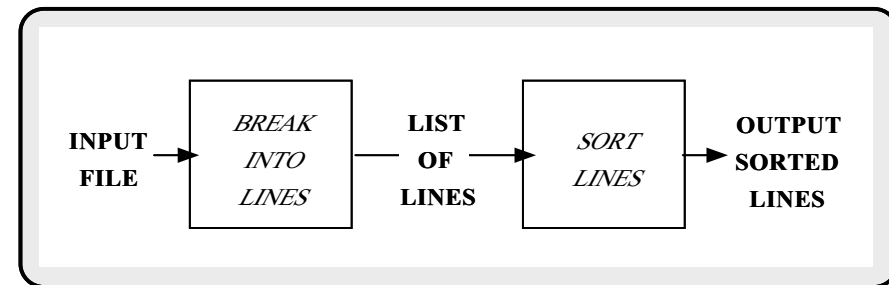
## System Sort Example

- A system sort utility is used to sort lines of files in various ways specified to the program, e.g.,

```
% sort -r /usr/dict/words  
% sort -n grades
```

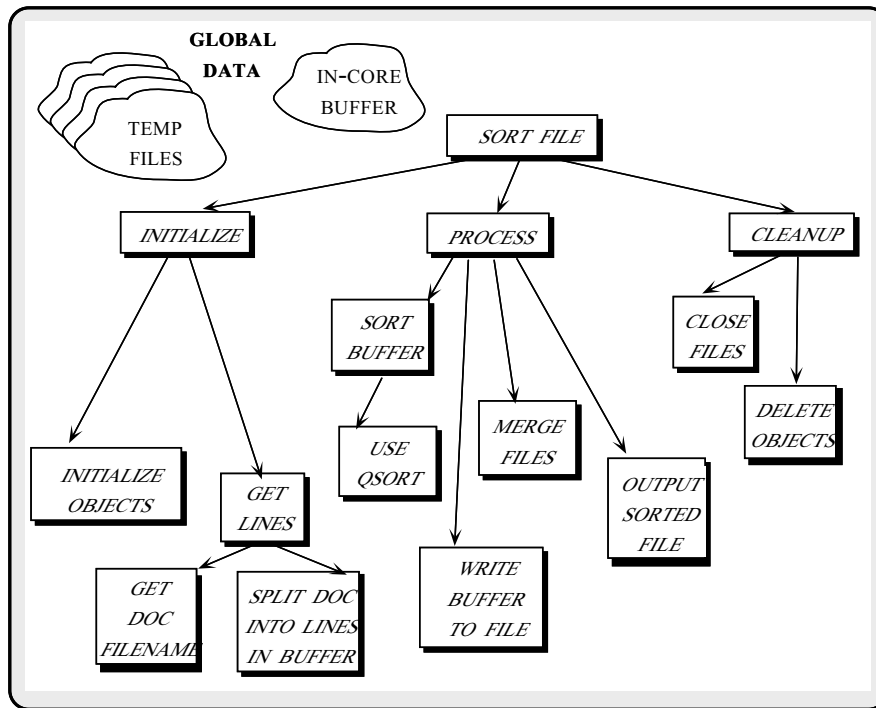
- This utility must work for files that are larger than main memory
  - Typically this is accomplished via some type of hybrid architecture that combines several different sorting techniques
    - ▷ e.g., memory sorting vs. disk sorting

## Data Flow Decomposition of System Sort



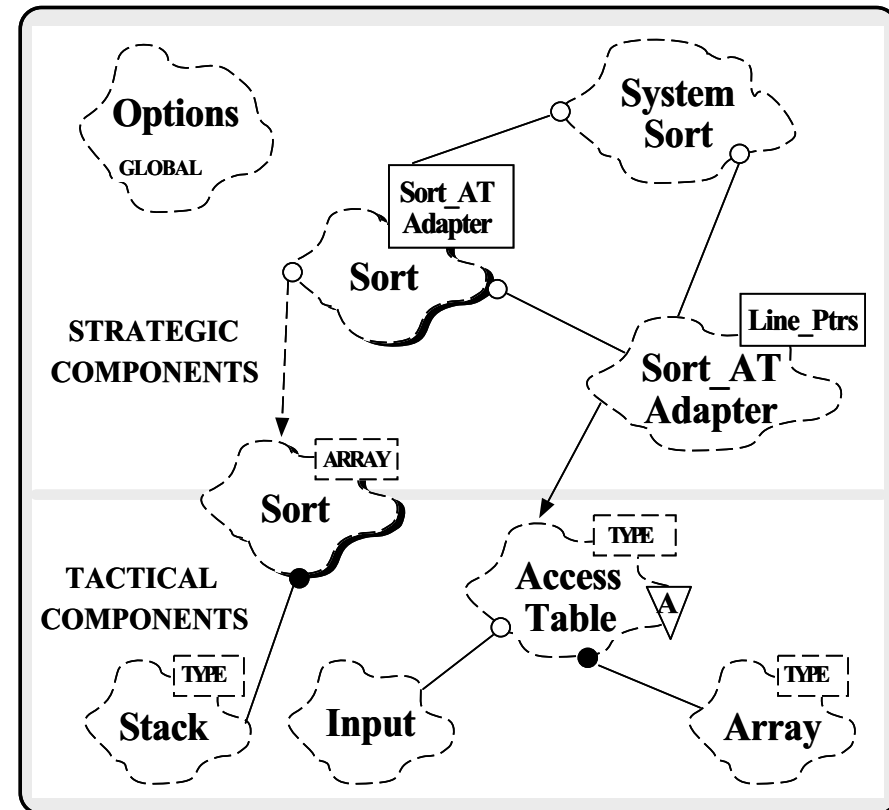


## Algorithmic Decomposition of System Sort



17

## OO Decomposition of System Sort

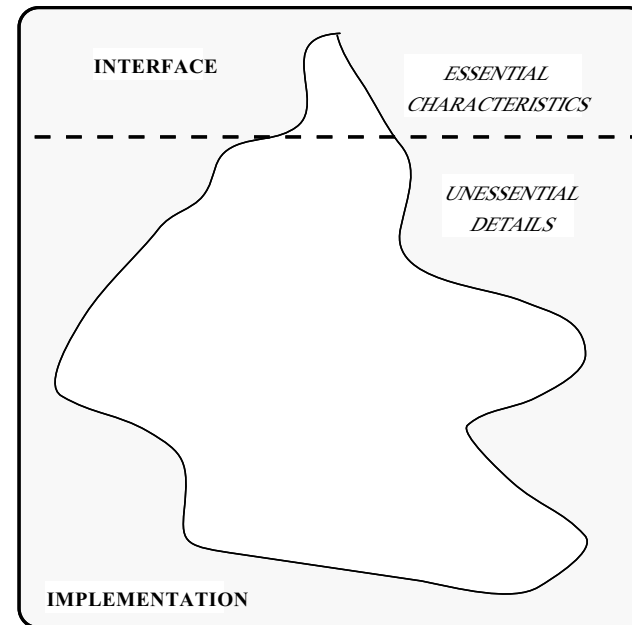


18

## Evaluating Decompositions of System Sort

- *Data flow*
  - It is not typically appropriate to use this view as the basis for decomposing the software architecture
    - ▷ e.g., it tightly couples the components with the processing order
- *Algorithmic decomposition*
  - This is also not an appropriate basis for decomposing the actual system architecture
- *Object-Oriented*
  - Forms the basis for decomposing the actual system architecture

## Abstraction



- Abstraction provides a way to manage complexity by emphasizing essential characteristics and suppressing unessential details at the current unit of analysis

## Abstraction (cont'd)

- e.g., evolution of programming language abstraction mechanisms

- *Name abstraction*

```
int foo (void) {  
    int bar = 1; // mov #1, @sp(4)  
}
```

- *Expression abstraction*

```
int i = x * 10;
```

```
mov x, r1  
mul #10, r1  
mov r1, i
```

- *Procedural abstraction*

- ▷ e.g., closed subroutines

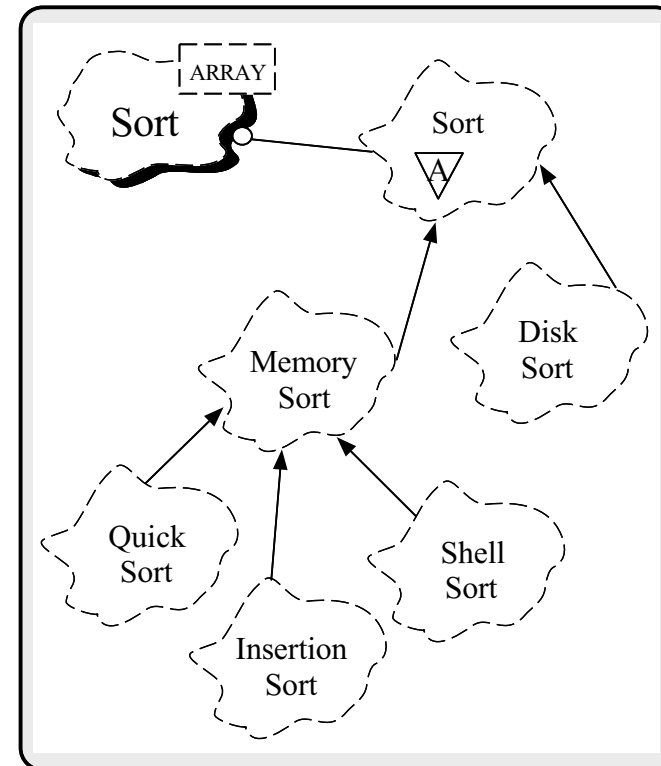
- *Data abstraction*

- ▷ e.g., ADTs, components

- *Control abstraction*

- ▷ **for/while** loops, iterators, event-loops, active objects, multitasking, etc.

## Abstraction (cont'd)



- This figure depicts the abstract class relations for the system sort

## Modularity

- *Motivation*

- Modularity is an essential characteristic of a good design since it:

1. Enables developers to reduce overall system complexity via *decentralized* software architectures, e.g.,

- ▷ *Divide and conquer*

- ▷ *Separation of concerns*

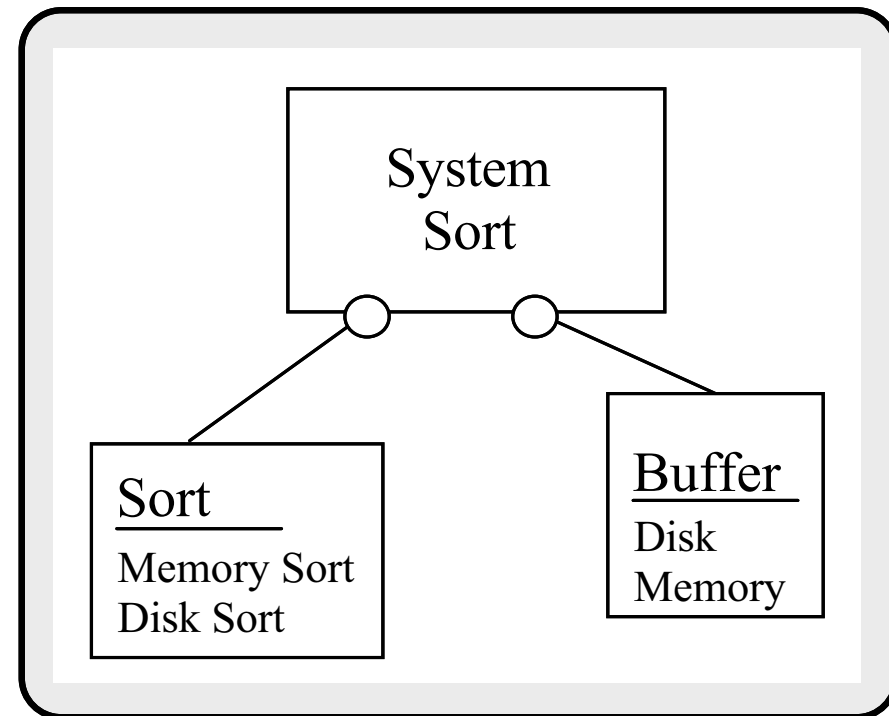
2. Enhances *scalability* by supporting independent and concurrent development by multiple personnel

- To be both useful and reusable, modules should possess

- \* Well-specified *abstract interfaces*
  - \* High *cohesion* and low *coupling*

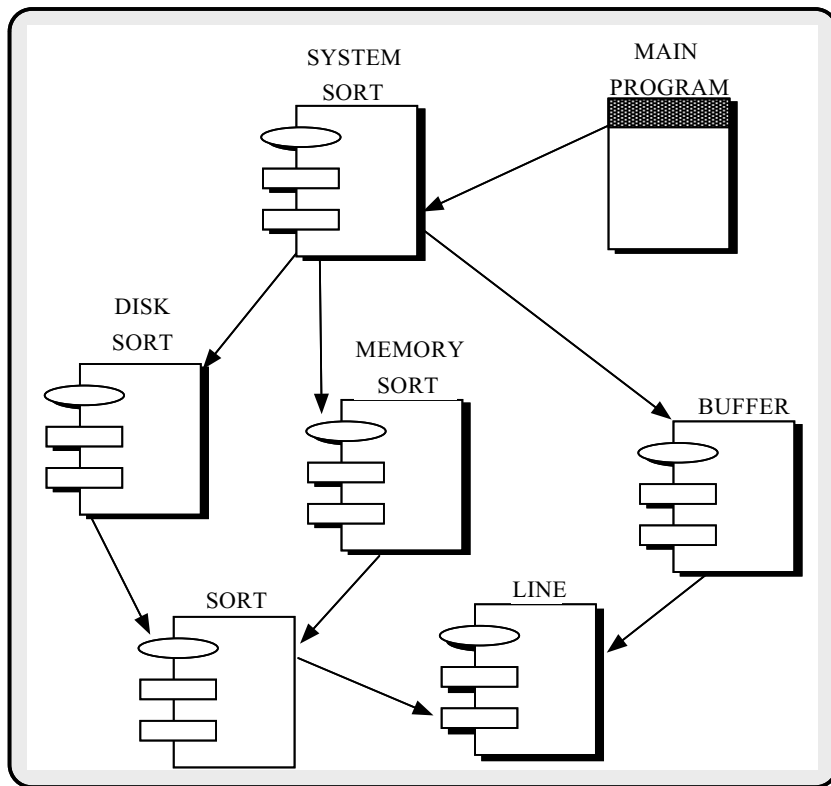
- Modularity exists at both a *logical* and *physical* level of abstraction

## Modularity (cont'd)



- Class categories

## Modularity (cont'd)



- Module structure

## Principles for Ensuring Modular Designs

- *Language Support for Modular Units*
  - If possible, use languages where modules correspond to syntactically and semantically enforced programming units, e.g.,
    - ▷ Namespaces and classes in C++
    - ▷ Packages in Ada and Java
    - ▷ Modules in Modula 2
- *Information Hiding* (described below)
  - All information about a module should be private to the module unless it is specifically declared public

## Principles for Ensuring Modular Designs (cont'd)

- *Few Interfaces*

- Every module should communicate with as few others as possible to reduce coupling
- By employing “weak coupling” the opportunity for reuse, extensibility, maintainability, and understandability are enhanced

- *Small Interfaces*

- If any two modules communicate at all, they should exchange as little information as possible
  - ▷ e.g., avoid passing pointers to control blocks...
- However, be careful of the ramifications of interface changes in hierarchical designs...

## Principles for Ensuring Modular Designs (cont'd)

- *Explicit Interfaces*

- Whenever two modules A and B communicate, this must be obvious from the text of A or B or both
  - ▷ Either via comments or language constructs
- e.g., implicit interfaces

```
/* File globals.c */  
int global = 0; /* Global variable */  
/* File foo.c */  
extern int global;  
int foo (void) { if (global == 0) global++; ... }  
/* File bar.c */  
extern double global; /* Error!!!! */  
int bar (void) { if (global == 10) global = 0; ... }  
– e.g., explicit interfaces
```

```
package Globals is  
    global : integer;  
end package;  
package Foo is  
    with Globals;  
end package;  
package Bar is  
    with Globals;  
end package;
```

## Information Hiding and Encapsulation

- *Motivation*

- Details of design decisions that are subject to change should be hidden behind abstract interfaces
  - ▷ *i.e.*, encapsulated
- Information hiding is one means to enhance abstraction
  - ▷ Note that information hiding is somewhat orthogonal to modularity
  - ▷ *e.g.*, some languages support modularity but not encapsulation
    - *e.g.*, Turbo Pascal

## Information Hiding and Encapsulation (cont'd)

- Typical information to hide includes:

- *Data representations*
  - ▷ *e.g.*, stack implementation
- *Algorithms*
  - ▷ *e.g.*, sorting routines
- *Input and Output Formats*
  - ▷ *e.g.*, network vs. host byte ordering
- *Lower-level module interfaces*
  - ▷ *i.e.*, how the module itself is implemented in terms of components used from other modules

## Information Hiding and Encapsulation (cont'd)

- *e.g.*,

- Contrast implicit information hiding support

```
struct String {  
    int length; /* Please use the str_len() function! */  
    char *str;  
} s;  
int str_len (struct String *s);
```

```
s.length = 100 /* Oops, forgot to use str_len()! */
```

- with explicit information hiding support

```
class String {  
public:  
    int str_len (void);  
private:  
    int length;  
    char *str;  
} s;  
s.length = 100; /* Compile-time error!!!! */
```

## Separate Policies and Mechanisms

- *Motivation*

- Clearly distinguishing between the *what/when* and the *how* at both the design and implementation phases enhances reuse and extensibility

- Multiple policies may be implemented via a set of shared mechanisms

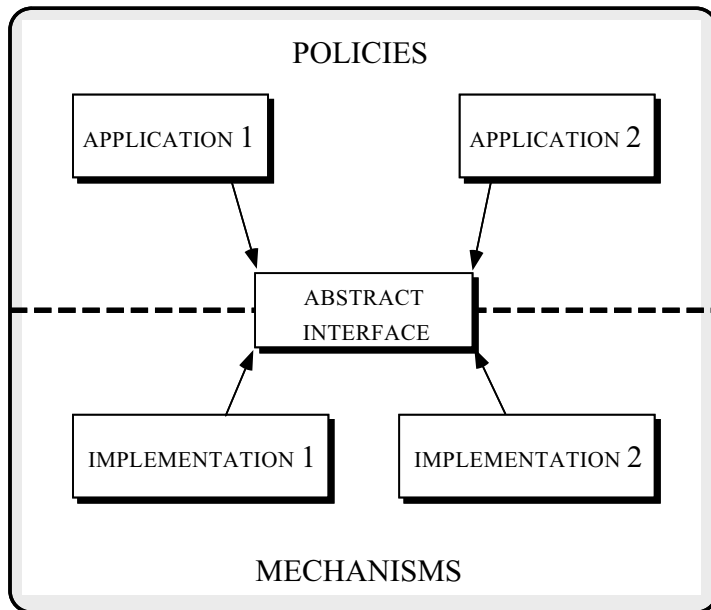
- *e.g.*, OS scheduling and virtual memory paging

- Same policy can be implemented by multiple mechanisms

- *e.g.*, reliable, non-duplicated, bytestream service can be provided by multiple communication protocols

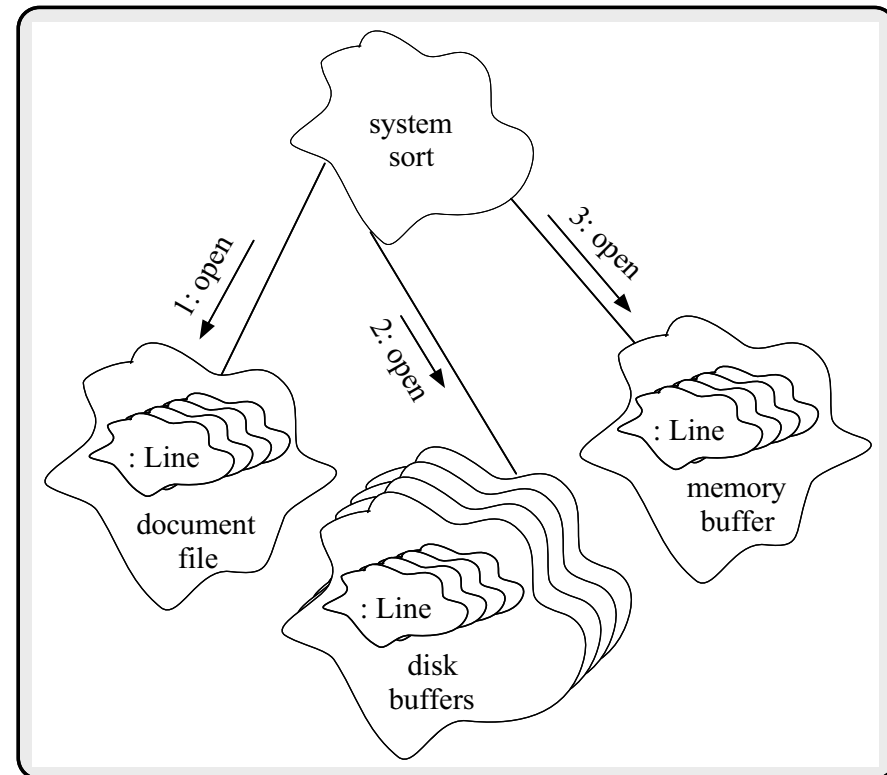


## Separate Policies and Mechanisms (cont'd)



- What is a policy and what is a mechanism is a matter of perspective...

## Separate Policies and Mechanisms (cont'd)



## Program Families and Subsets

- Program families are a collection of related modules or subsystems that form a reusable *framework*
  - e.g., UNIX System V STREAMS I/O subsystem, Sun Solaris 2.x OS (runs on both Intel and Sparc platform)
- The components in a program family are similar enough that it makes sense to emphasize their similarities before discussing their differences
- *Motivation*
  - Program families are useful for implementing *subsets* of system functionality
  - Reasons for providing subsets include cost, time, personnel resources, etc.

## Program Families and Subsets (cont'd)

- Identifying subsets:
  - Analyze requirements to identify minimally useful subsets
  - Also identify minimal increments to subsets
- Advantages of subsetting:
  - Facilitates software system extension and contraction
  - Promotes reusability and modularity
  - Anticipates potential changes

## Program Families and Subsets (cont'd)

- Program families support:
  - Different services for different markets
    - ▷ e.g., different alphabets, different I/O formats
  - Different hardware or software platforms
    - ▷ e.g., compilers or OSs
  - Different resource trade-offs
    - ▷ e.g., speed vs. space
  - Different internal resources
    - ▷ e.g., shared data and library routines
  - Different external events
    - ▷ e.g., UNIX I/O device interface
  - Backward compatibility
    - ▷ e.g., sometimes it is important to retain bugs!

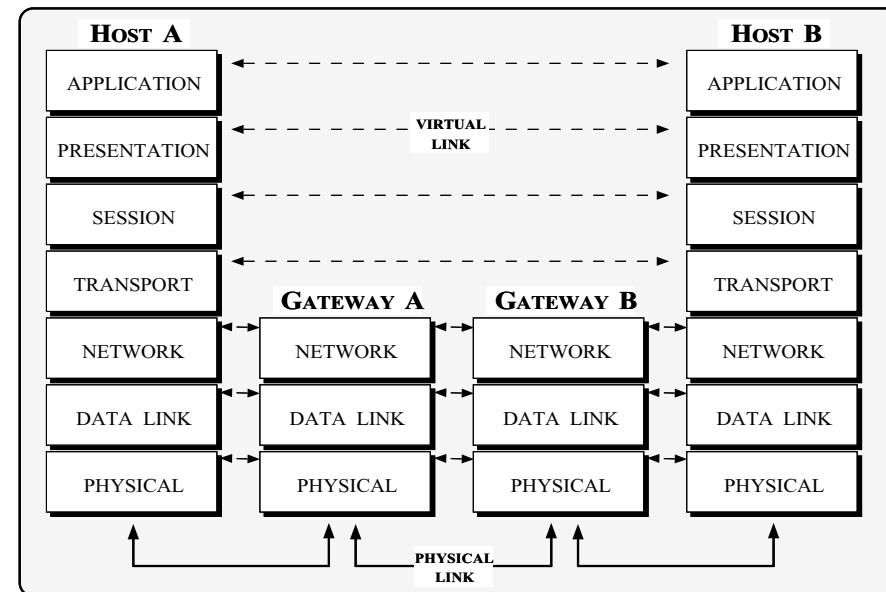
## Virtual Machines

- *Motivation*
  - To reduce overall complexity, software system architectures may be decomposed into hierarchically structured “virtual machines”
- A virtual machine provides an extended “software instruction set”
  - Provides additional data types and associated “software instructions” that extend the underlying hardware instruction set, e.g.,  
  
**int** qsort (**void** \*, **int**, **int**, **int** (\*)(**void** \*, **void** \*));
- Benefits
  - Enables developers to work at higher-levels of abstraction
    - ▷ e.g., RPC vs. sockets
  - Virtual machines provide the basis for incremental extensions to existing “application programmatic interfaces” (APIs)

## Virtual Machines (cont'd)

- Common examples of virtual machines include
  - *Computer Architectures*
    - ▷ e.g., compiler → assembler → object code → microcode → gates, transistors, signals, etc.
  - *Communication protocol stacks*
    - ▷ e.g., ISO OSI reference model, Internet reference model

## Virtual Machines (cont'd)

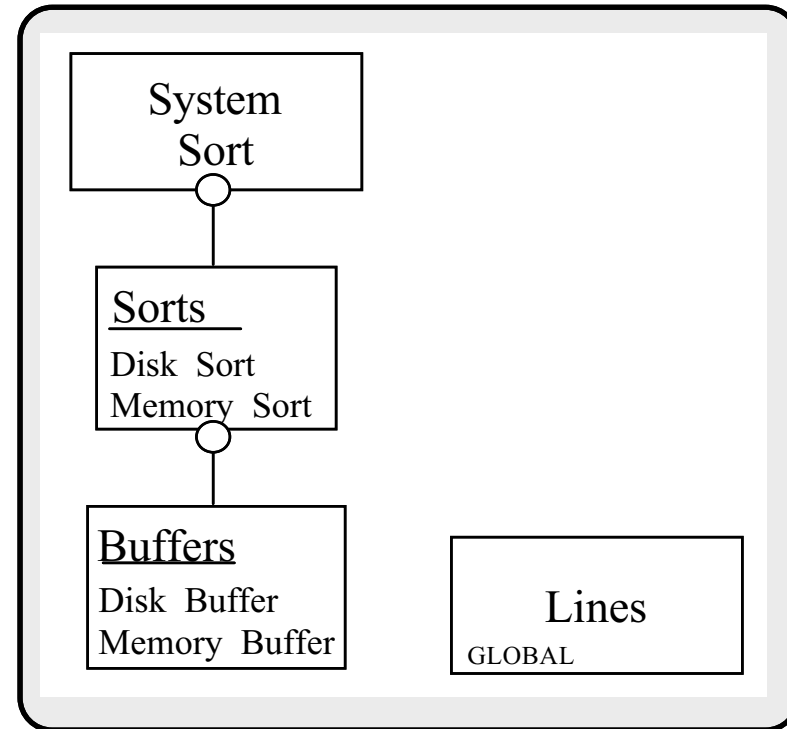


- The OSI reference model

## Virtual Machines (cont'd)

- It is important to distinguish *virtual machine layering* from *partitioning*
  - Layering results from applying horizontal slices to a system architecture
    - ▷ e.g., the OSI layers illustrated above
  - Partitioning results from applying vertical slices to a particular layer in a system architecture
    - ▷ e.g., the sender and receiver partitions of the transport layer
- Class categories may be used to indicate logical system layering

## Virtual Machines (cont'd)



- Class categories for the system sort

## Virtual Machines (cont'd)

- Several challenges must be overcome to effectively use virtual machines as an architectural structuring technique:
  - *Ensuring Adequate Performance:*
    - ▷ It is difficult to obtain good performance at level  $N$ , if levels below  $N$  are not implemented efficiently
    - ▷ This often requires *implementing* the virtual machine differently than the design may initially suggest...
  - *Alleviating Inter-level Dependencies*
    - ▷ To maximize reuse, it is essential to eliminate/reduce dependencies “between” virtual machine levels...
    - ▷ Therefore, virtual machines are often organized into hierarchical *layers* or *levels of abstraction*

## Virtual Machines (cont'd)

- A “hierarchy” may be defined to reduce module interactions by restricting the topology of relationships between virtual machines
  - e.g., consider the OSI reference model!
- A relation defines a hierarchy if it partitions units into levels
  - Level 0 is the set of all units that use no other units
  - Level  $i$  is the set of all units that use at least one unit at level  $< i$  and no unit at level  $> i$
- Advantages of hierarchical structuring
  - *Facilitates independent development of levels or layers*
  - *Isolates ramifications of change*
  - *Enables rapid prototyping*

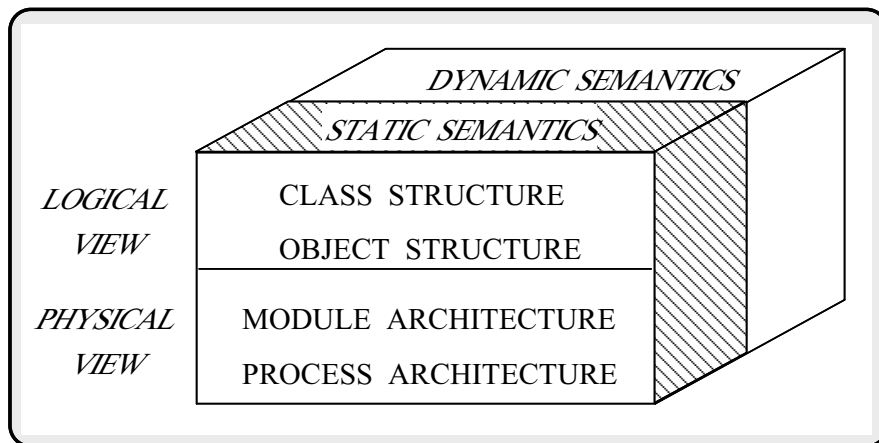
## OO Modeling Issues

- Developers must address the following issues:
  1. What classes exist and how are these classes related?
  2. How are individual objects managed and what mechanisms are used to specify how societies of objects collaborate?
  3. In which modules should each class and object be defined?
    - What components are visible?
    - How are large systems partitioned into subsystems?
  4. How should the objects be allocated to the processes and/or threads of control?

## Modeling Complex Software

- The four primary types of components in an OO design are:
  1. *Classes*
  2. *Objects*
  3. *Modules*
  4. *Processes*
- The OO design process results in a set of diagrams that express the following “views” of the system architecture
  1. *Class relationships*
  2. *Object collaboration*
  3. *Module partitioning*
  4. *Process interactions*

## Multiple Views of System Structure



- Complex systems require multiple views to capture their static and dynamic characteristics

## Four Main Views

### 1. *Class Diagrams*

- (a) Illustrates individual classes and collections of *class categories* (static view)
- (b) State charts depict actions that member functions perform (dynamic view)

### 2. *Object Diagrams*

- (a) Object-instance diagrams (static view)
- (b) Interaction diagrams (dynamic view)

### 3. *Module Diagrams*

- Illustrates partitioning of classes/objects into *sub-systems*

### 4. *Process Diagrams*

- Illustrate allocation of objects to processes, as well as processes to processors



## Logical and Physical Views

- Class and object diagrams form the “logical” view of a system
  - *i.e.*, they describe the existence, structure, and meaning of key abstractions and mechanisms in the system
- Module and process diagrams form the “physical” view of a system
  - *i.e.*, they describe the concrete software and hardware components of the implementation
- The logical and physical views may be either tightly-coupled or loosely-coupled

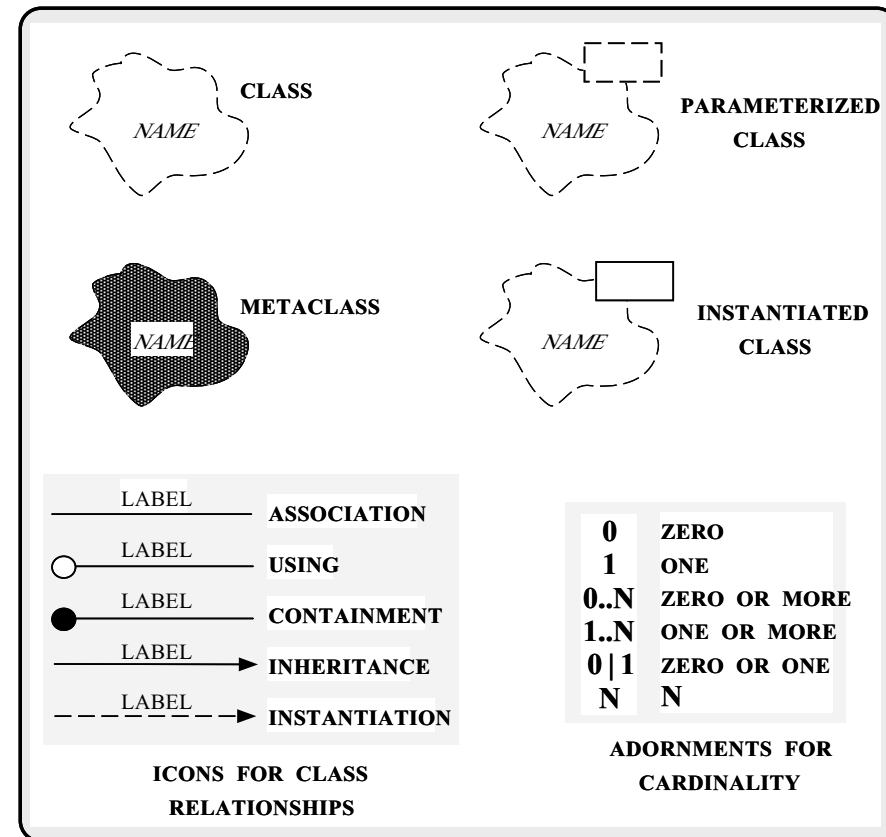
## The Need for Notation

- During both the analysis and design phases, it is useful to have a systematic notation that captures the essential elements and decisions of the development process
  - The analysis phase typically requires less detailed aspects of the notation
  - The design phases typically require much more detailed aspects of the notation
- The notation should be relatively language independent
  - However, certain portions of the notation may be replaced with language-specific components
    - ▷ Particularly the “class and module specification” portions

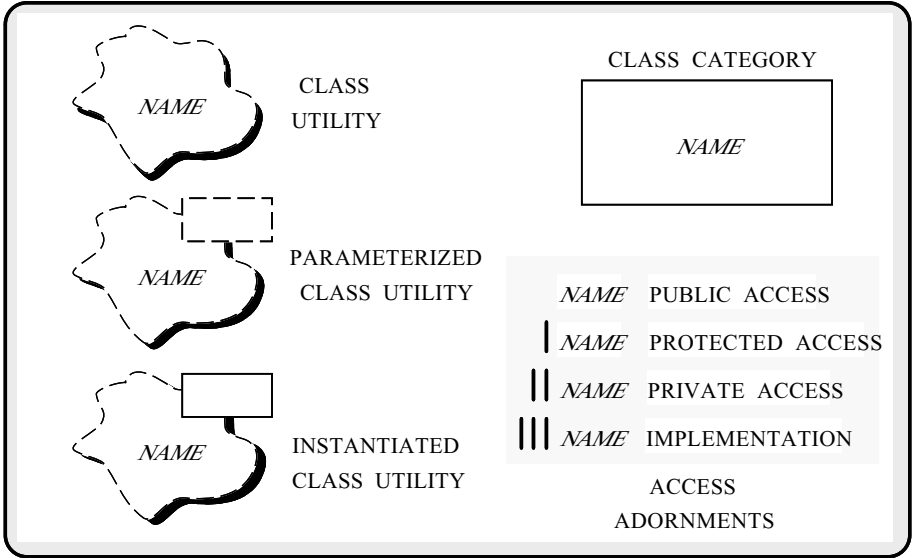
## Important Caveat

- OO system development is *not* simply the act of drawing a picture using a CASE tool or a graphical notation!!!
  - Analysis and design are fundamentally *creative* activities
  - A notation only serves to help a development team formulate a model and communicate this model to others
- However, given a sufficiently rich notation and a set of associated tools, it may be possible to provide *consistency*, *completeness*, and *correctness* checks on the design

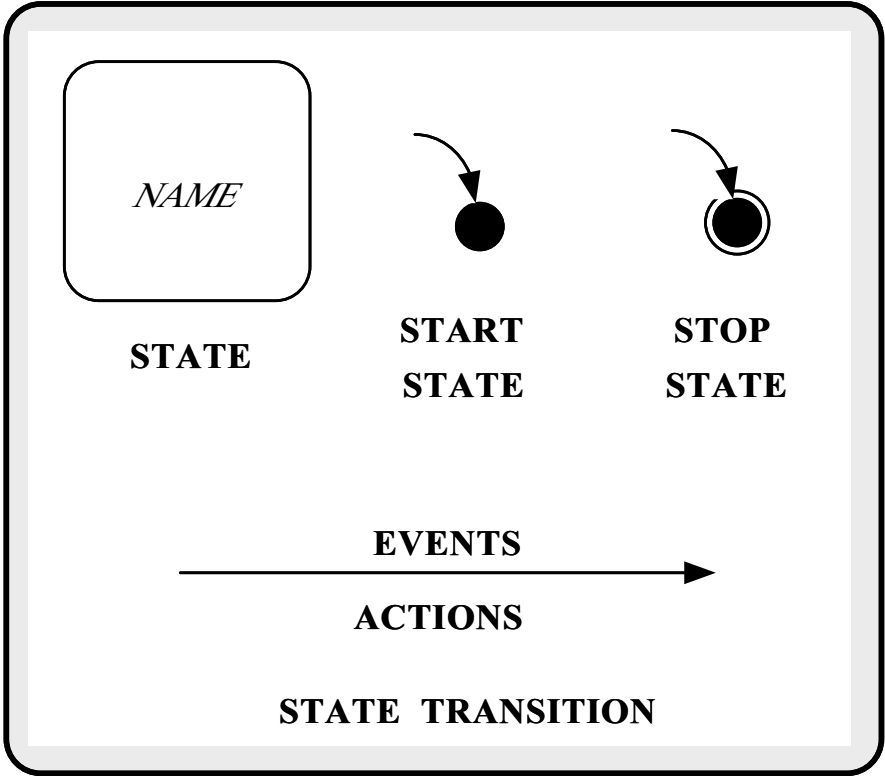
## Icons for Classes



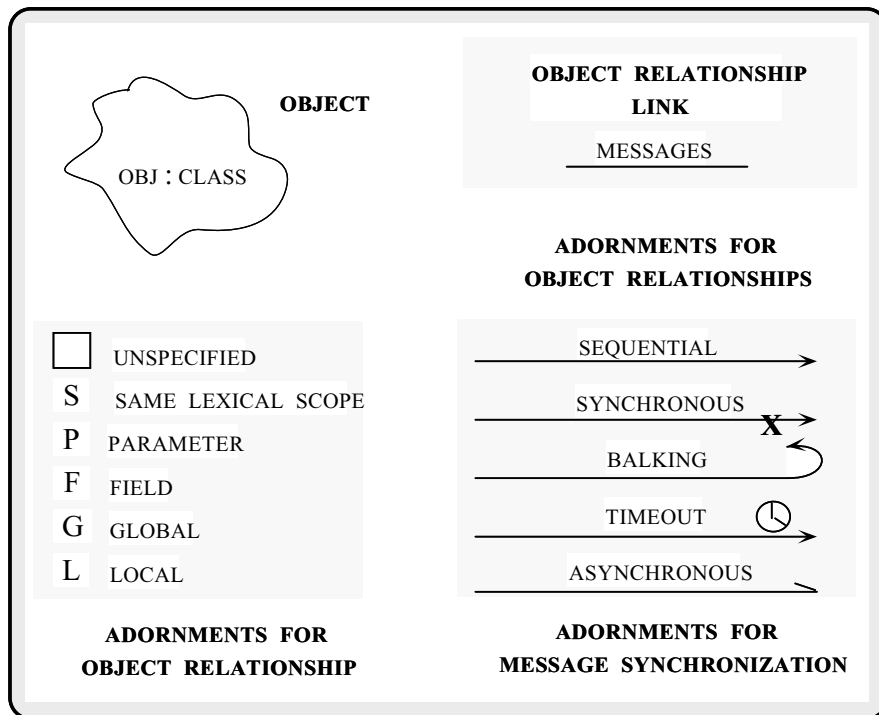
# Icons for Class Utilities and Class Categories



# Icons for State Machines and Transitions



## Icons for Objects



## OO Design Alternatives

- A key question facing software architects and designers is:
  - Should software systems be structured by *actions* or by *data*?
  - This decision cannot be postponed indefinitely
    - ▷ Eventually, a designer must settle on one or the other
    - ▷ Note, the source code reveals the final decision...
- *Observation:*
  - The tasks and functions performed by a software system are often highly volatile and subject to change
- *Conclusion:*
  - Structuring systems around classes and objects increases continuity and improves maintainability over time for large-scale systems

## Algorithmic Design

- Top-down design based on the *functions* performed by the system
- Generally follows a “divide and conquer” strategy based on functions
  - *i.e.*, more general functions are iteratively/recursively decomposed into more specific ones
- The primary design components correspond to processing steps in the execution sequence
  - Similar to a recipe for cooking a meal
    - ▷ Consider the objects used in cooking...

## Object-oriented Design

- Design based on modeling objects in the application domain
  - Which may or may not reflect the “real world”
- Generally follows a “hierarchical data abstraction” strategy where the design components are based on classes, objects, modules, and processes
- Operations are related to specific objects and/or classes of objects

## Structured Design

- Design is based on data structures input and output during system operation
  - Generally follows a decomposition strategy based on data flow between processing components
  - Primary design components correspond to flow of data
    - ▷ Program structure is derived from data structure
    - ▷ Data structure charts show decomposition of input/output streams

## Structured Design (cont'd)

- Often used as the basis for designing data processing systems
  - Note that these types of systems are often very similar...
- Design tends to be overly dependent upon temporal ordering of processing phases
  - e.g., initialize, process, cleanup
- Changes in data representations ripple through entire structure due to lack of information hiding

## Transformational Systems

- Design is based on specifying the problem, rather than specifying the solution
  - The solution is automatically derived from the high-level specification
  - Note, each transformation component may be implemented via other design alternatives
- Limited to well-understood domains where flexibility, rather than raw efficiency, is the primary concern:
  - e.g., parser-generators, GUI layouts

## Criteria for Evaluating Design Methods

- *Component Decomposability*
  - Does the method aid decomposing a new problem into several separate subproblems?
    - ▷ e.g., top-down algorithmic design
- *Component Composability*
  - Does the method aid constructing new systems from existing software components?
    - ▷ e.g., bottom-up design
- *Component Understandability*
  - Are components separately understandable by a human reader
    - ▷ e.g., how *tightly coupled* are they?

## Criteria for Evaluating Design Methods (cont'd)

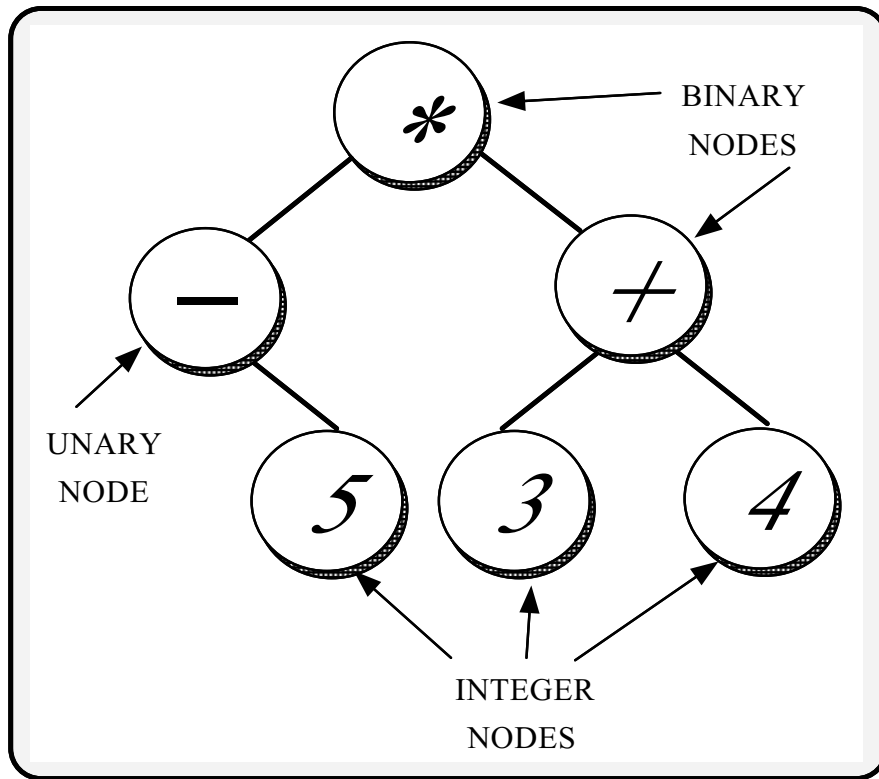
- *Component Continuity*
  - Do small changes to the specification affect a localized and limited number of components?
- *Component Protection*
  - Are the effects of run-time abnormalities confined to a small number of related components?
- *Component Compatibility*
  - Do the components have well-defined, standard and/or uniform interfaces?
    - ▷ e.g., “principle of least surprise”

## Case Study 1: Expression Tree Evaluator

- The following inheritance and dynamic binding example constructs *expression trees*
  - Expression trees consist of nodes containing operators and operands
    - ▷ Operators have different *precedence levels*, *different associativities*, and different *arities*, e.g.,
      - Multiplication takes precedence over addition
      - The multiplication operator has two arguments, whereas unary minus operator has only one
    - ▷ Operands are integers, doubles, variables, etc.
      - We'll just handle integers in the example...



## Expression Tree Diagram



## Expression Tree Behavior

- *Expression trees*
  - These trees may be “evaluated” via different traversals
    - ▷ e.g., in-order, post-order, pre-order, level-order
  - The evaluation step may perform various operations..., e.g.,
    - ▷ Traverse and print the expression tree
    - ▷ Return the “value” of the expression tree
    - ▷ Generate code
    - ▷ Perform semantic analysis

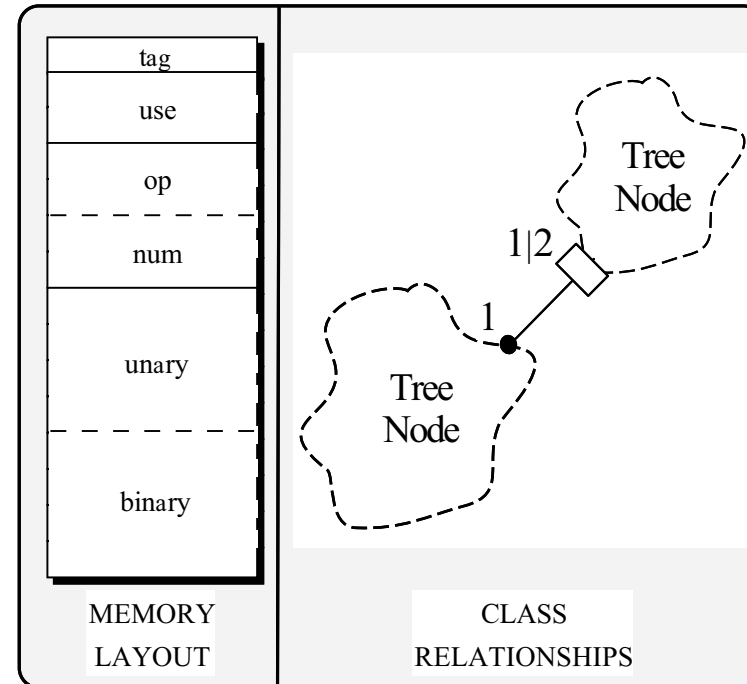
## C Version

- A typical functional method for implementing expression trees in C or Ada involves using a **struct/union** to represent data structure, e.g.,

```
typedef struct Tree_Node Tree_Node;
struct Tree_Node {
    enum {
        NUM, UNARY, BINARY
    } tag;
    short use; /* reference count */
    union {
        int num;
        char op[2];
    } o;
#define num o.num
#define op o.op
    union {
        Tree_Node *unary;
        struct { Tree_Node *l, *r; } binary;
    } c;
#define unary c.unary
#define binary c.binary
};
```

67

## Memory Layout of C Version



- Here's what the memory layout of a **struct** Tree\_Node object looks like

68

## Print\_Tree Function

- Typical C or Ada implementation (cont'd)

- Use a **switch** statement and a recursive function to build and evaluate a tree, e.g.,

```
void print_tree (Tree_Node *root) {  
    switch (root->tag) {  
        case NUM: printf ("%d", root->num); break;  
        case UNARY:  
            printf ("%s", root->op[0]);  
            print_tree (root->unary);  
            printf (")"); break;  
        case BINARY:  
            printf ("(");  
            print_tree (root->binary.l);  
            printf ("%s", root->op[0]);  
            print_tree (root->binary.r);  
            printf (")"); break;  
        default:  
            printf ("error, unknown type\n");  
            exit (1);  
    }  
}
```

## Limitations with C Approach

- Problems or limitations with the typical design and implementation approach include
  - Language feature limitations in C and Ada
    - ▷ e.g., no support for inheritance and dynamic binding
  - Incomplete modeling of the problem domain that results in
    1. Tight coupling between nodes and edges in **union** representation
    2. Complexity being in algorithms rather than the data structures
      - ▷ e.g., **switch** statements are used to select between various types of nodes in the expression trees
        - compare with binary search!
  - ▷ Data structures are “passive” in that functions do most processing work explicitly

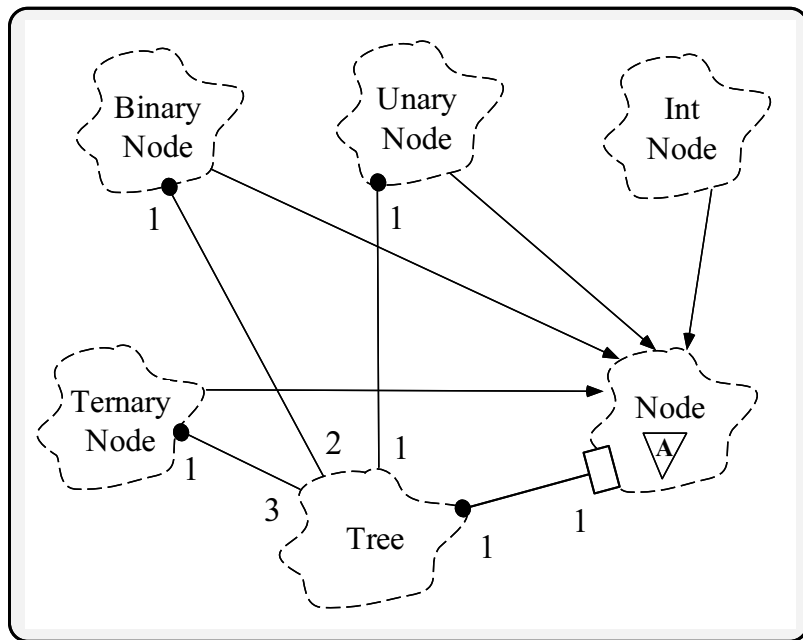
## Limitations with C Approach (cont'd)

- Problems with typical approach (cont'd)
  - The program organization makes it difficult to extend, *e.g.*,
    - ▷ Any small changes will ripple through the entire design and implementation
      - *e.g.*, see the ternary extension below
    - ▷ Easy to make mistakes **switching** on type tags..
  - Solution wastes space by making worst-case assumptions *wrt* **structs** and **unions**
    - ▷ This not essential, but typically occurs
    - ▷ Note that this problem becomes worse the bigger the size of the largest item becomes!

## OO Alternative

- Contrast previous functional approach with an object-oriented decomposition for the same problem:
  - Start with OO modeling of the “expression tree” problem domain:
    - ▷ *e.g.*, go back to original picture
  - There are several classes involved:
    - class** Node: base class that describes expression tree vertices:
      - class** Int\_Node: used for implicitly converting **int** to Tree node
      - class** Unary\_Node: handles unary operators, *e.g.*,  $-10$ ,  $+10$ ,  $!a$ , or  $\sim\text{foo}$ , etc.
      - class** Binary\_Node: handles binary operators, *e.g.*,  $a + b$ ,  $10 - 30$ , etc.
    - class** Tree: “glue” code that describes expression tree edges
  - Note, these classes model elements in the application domain
    - ▷ *i.e.*, nodes and edges (or vertices and arcs)

## Relationships Between Trees and Nodes



## Design Patterns in the Expression Tree Program

- Adapter
  - “Convert the interface of a class into another interface client expects”
  - ▷ e.g., make **Tree** conform to interfaces expected by C++ iostreams operators
- Factory
  - “Centralize the assembly of resources necessary to create an object”
  - ▷ e.g., decouple **Node** subclass initialization from their subsequent use
- Bridge
  - “Decouple an abstraction from its implementation so that the two can vary independently”
  - ▷ e.g., printing the contents of a subtree

## C++ Node Interface

- // node.h

```
#ifndef _NODE_H
#define _NODE_H
#include <iostream.h>
#include "Tree.h"
```

```
// Describes the Tree vertices.
```

```
class Node {
friend class Tree;
```

```
protected: // Only visible to derived classes.
    Node (void)
        : use (1) {}
```

```
// pure virtual
```

```
virtual void print (ostream &) const = 0;
```

```
// Important to make destructor virtual!
```

```
virtual ~Node (void);
```

```
private:
```

```
    int use; // Reference counter.
```

```
};
```

```
#endif
```

## C++ Tree Interface

- // Tree.h

```
#include "Node.h"
```

```
// Describes the Tree edges.
```

```
class Tree {
```

```
public:
```

```
    ~Tree (void);
```

```
    Tree (int);
```

```
    Tree (char *, Tree &);
```

```
    Tree (char *, Tree &, Tree &);
```

```
    Tree (const Tree &t); // Copy constructor
```

```
    void operator= (const Tree &t); // Assignment
```

```
    void print (ostream &);
```

```
private:
```

```
    Node *node_; // pointer to a rooted subtree.
```

```
};
```

## C++ Int\_Node and Unary\_Node Interface

- // Int\_Node.h

```
#include "Node.h"
```

```
class Int_Node : public Node {  
public:  
    Int_Node (int k);  
    virtual void print (ostream &stream) const;  
private:  
    int num; // operand value.  
};
```

- // Unary\_Node.h

```
#include "Node.h"
```

```
class Unary_Node : public Node {  
public:  
    Unary_Node (const char *op, const Tree &t);  
    virtual void print (ostream &stream) const;  
private:  
    const char *operation;  
    Tree operand;  
};
```

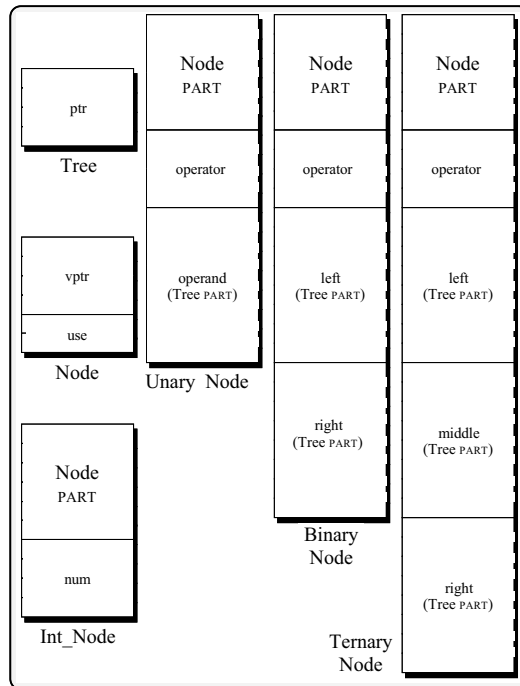
## C++ Binary\_Node Interface

- // Binary\_Node.h

```
#include "Node.h"
```

```
class Binary_Node : public Node {  
public:  
    Binary_Node (const char *op,  
                 const Tree &t1,  
                 const Tree &t2);  
    virtual void print (ostream &s) const;  
private:  
    const char *operation;  
    Tree left;  
    Tree right;  
};
```

## Memory Layout for C++ Version



- Memory layouts for different subclasses of Node

## C++ Int\_Node and Unary\_Node Implementations

- // Int\_Node.C

```
#include "int-node.h"
```

```
Int_Node::Int_Node (int k): num (k) { }
```

```
void Int_Node::print (ostream &stream) const {
    stream << this->num;
}
```

- // Unary\_Node.C

```
#include "unary-node.h"
```

```
Unary_Node::Unary_Node (const char *op, const Tree &t1)
    : operation (op), operand (t1) { }
```

```
void Unary_Node::print (ostream &stream) const {
    stream << "(" << this->operation << " "
        << this->operand // recursive call!
        << ")";
}
```



## C++ Binary\_Node Implementation

- // Binary\_Node.C

```
#include "binary-node.h"
```

```
Binary_Node::Binary_Node (const char *op,  
                           const Tree &t1,  
                           const Tree &t2):  
    operation (op), left (t1), right (t2) { }
```

```
void Binary_Node::print (ostream &stream) const {  
    stream << "(" << this->left // recursive call  
        << " " << this->operation  
        << " " << this->right // recursive call  
        << ")";  
}
```

## Initializing the Node Subclasses

- *Problem*

- How to ensure the Node subclasses are initialized properly

- *Forces*

- There are different types of Node subclasses
  - ▷ e.g., take different number and type of arguments
- We want to centralize initialization in one place because it is likely to change...

- *Solution*

- Use a “Factory” pattern to initialize the Node subclasses

## The Factory Pattern

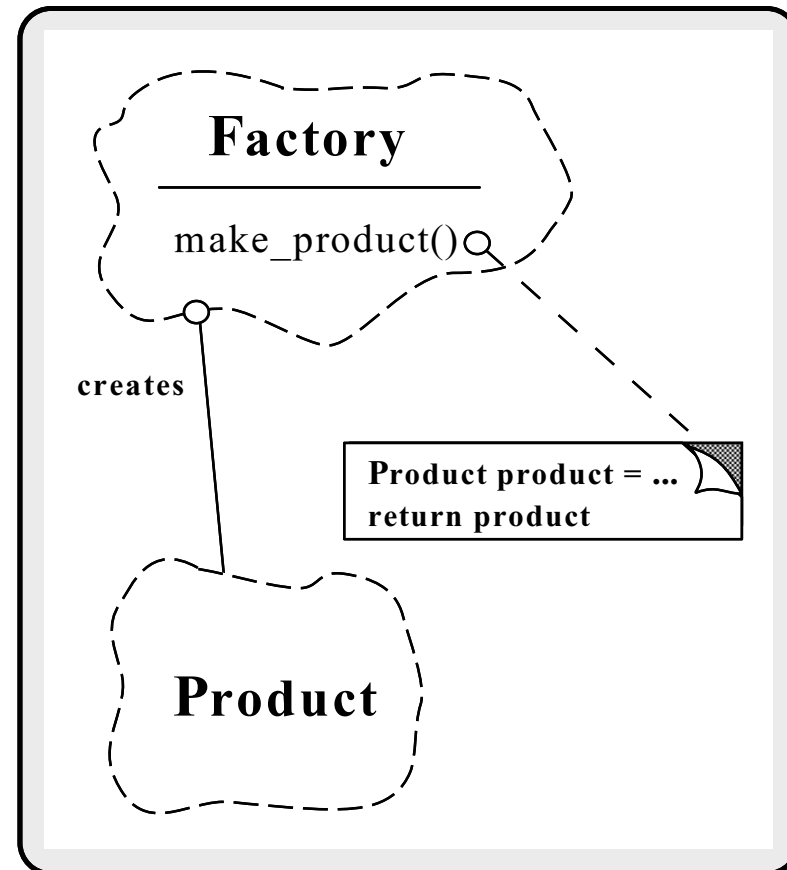
- *Intent*

- “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 **Node** subclasses from their subsequent use
- Makes it easier to change add new Node subclasses later on
  - ▷ e.g., Ternary nodes...

## Structure of the Factory Pattern



## Using the Factory Pattern

- The Factory pattern is used to initialize node subclasses:

```
Tree::Tree (int num)
: node_ (new Int_Node (num)) {}
```

```
Tree::Tree (const char *op, const Tree &t)
: node_ (new Unary_Node (op, t)) {}
```

```
Tree::Tree (const char *op,
            const Tree &t1,
            const Tree &t2):
: node_ (new Binary_Node (op, t1, t2)) {}
```

## Printing Subtrees

- *Problem*

- How do we print subtrees without revealing their types?

- *Forces*

- The **Node** subclass are hidden within the **Tree** instances and we don't want to become dependent on the use of **Nodes**, inheritance, and dynamic binding, etc.

- *Solution*

- Use the “Bridge” pattern to shield the use of inheritance and dynamic binding

## The Bridge Pattern

- *Intent*

- Decouple an abstraction from its implementation so that the two can vary independently

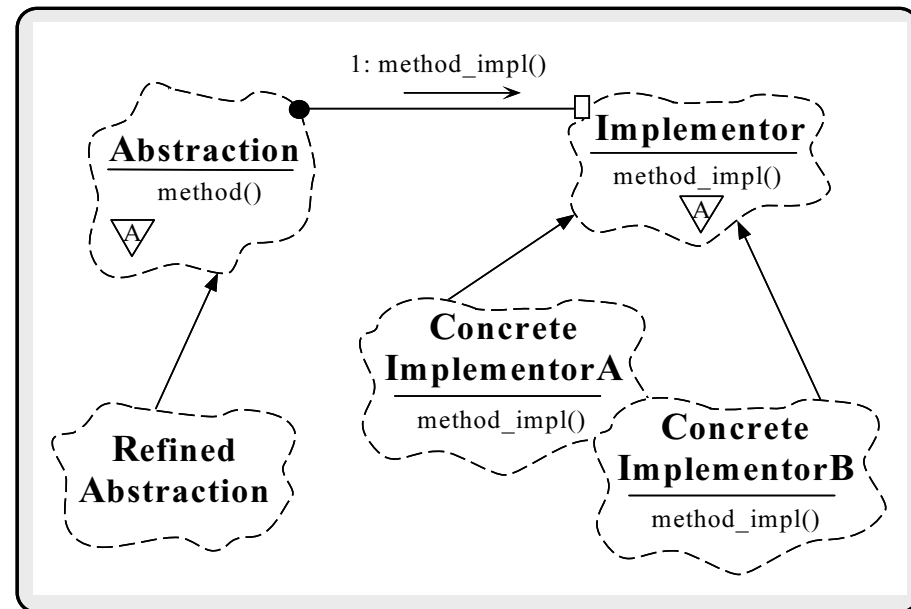
- This pattern resolves the following forces that arise when building extensible software

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<*

## Structure of the Bridge Pattern



## Using the Bridge Pattern

- The Bridge pattern is used for printing expression trees:

```
void Tree::print (ostream &os)
{
    this->node_->print (os);
}
```

- Note how this decouples the **Tree** interface for printing from the **Node** subclass implementation
  - *i.e.*, the **Tree** interface is *fixed*, whereas the **Node** implementation varies
  - However, clients need not be concerned about the variation...

## Integrating with C++ I/O Streams

- *Problem*
  - Our **Tree** interface uses a **print** method, but most C++ programmers expect to use I/O Streams
- *Forces*
  - How to integrate our existing C++ **Tree** class into the I/O Stream paradigm without modifying our class or C++ I/O
- *Solution*
  - Use the *Adapter* pattern to integrate **Tree** with I/O Streams

## The Adapter Pattern

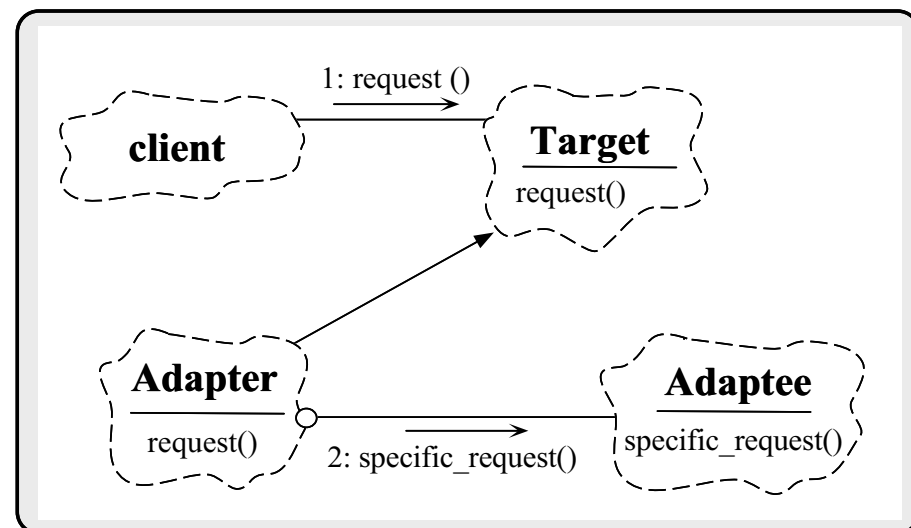
- *Intent*

- 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 force:

1. How to transparently integrate the **Tree** with the C++ iostream operators

## Structure of the Adapter Pattern



## Using the Adapter Pattern

- The Adapter pattern is used to integrate with C++ I/O Streams

```
ostream &operator<< (ostream &s, const Tree &tree) {  
    tree.print (s);  
    // This triggers Node * virtual call!.  
    // (*tree.node_>vp[1]) (tree.node_, s);  
    return s;  
}
```

- Note how all the C++ code shown above uses I/O streams, which “adapts” the Tree interface...

## C++ Tree Implementation

- Reference counting

```
Tree::~Tree (void) {  
    // Ref-counting, garbage collection  
    if (--this->node_>use <= 0)  
        delete this->node_;  
}  
  
Tree::Tree (const Tree &t): node_ (t.node_) {  
    // Sharing, ref-counting.  
    ++this->node_>use;  
}  
  
void Tree::operator= (const Tree &t) {  
    // order important here!  
    ++t.node_>use;  
    if (--this->node_>use == 0)  
        delete this->node_;  
    this->node_ = t.node_;  
}
```

## C++ Main Program

- // main.C

```
#include <stream.h>
#include "tree.h"

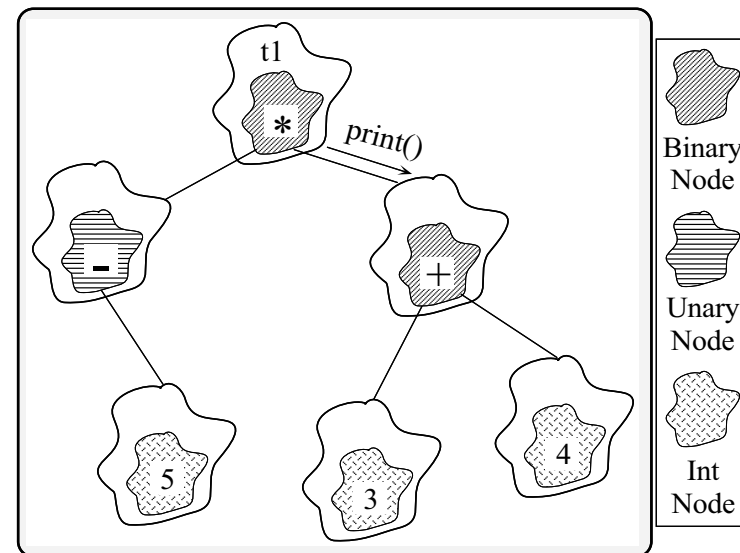
int main (void)
{
    const Tree t1 = Tree ("*",
                          Tree("-", 5),
                          Tree("+", 3, 4));
    // Tree ("*",
    // Tree("-", Tree(5)),
    // Tree("+", Tree(3), Tree(4)));

    // prints ((-5) * (3 + 4)).
    cout << t1 << endl;
    const Tree t2 = Tree ("*", t1, t1);

    // prints (((-5) * (3 + 4)) * ((-5) * (3 + 4))).
    cout << t2 << endl;

    // Destructors of t1 and t2 recursively
    // delete entire tree leaving scope.
}
```

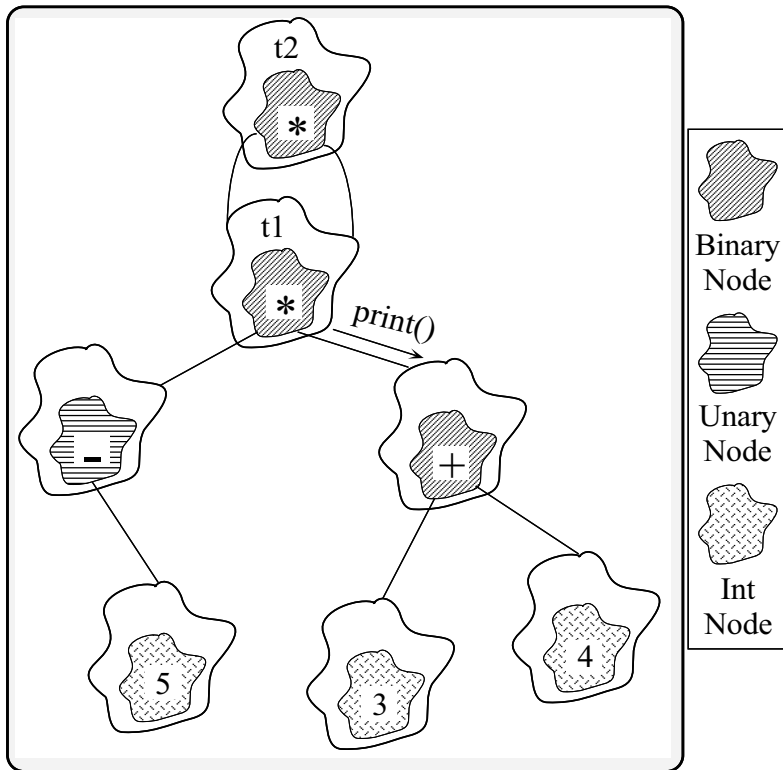
## Expression Tree Diagram 1



- Expression tree for  $t1 = ((-5) * (3 + 4))$



## Expression Tree Diagram 2



- Expression tree for  $t2 = (t1 * t1)$

## Extending Solution with Ternary\_Nodes

- Extending the existing solution to support ternary nodes is straightforward

– *i.e.*, just derived new class Ternary\_Node

**class** Ternary\_Node: handles ternary operators, *e.g.*,  $a == b ? c : d$ , etc.

- `// ternary-node.h`

**#include** "node.h"

```
class Ternary_Node : public Node {
public:
    Ternary_Node (const char *, const Tree &,
                  const Tree &, const Tree &);
    virtual void print (ostream &) const;
private:
    const char *operation;
    Tree left, middle, right;
};
```

## C++ Ternary\_Node Implementation

- // ternary-node.C

```
#include "ternary-node.h"
Ternary_Node::Ternary_Node (const char *op,
                           const Tree &a,
                           const Tree &b,
                           const Tree &c)
    : operation (op), left (a), middle (b), right (c) {}

void Ternary_Node::print (ostream &stream) const {
    stream << this->operation << "("
        << this->left // recursive call
        << "," << this->middle // recursive call
        << "," << this->right // recursive call << ")";
}
```

- // Modified class Tree

```
class Tree { // add 1 class constructor
// Same as before
public:
// Same as before
    Tree (const char *, const Tree &,
          const Tree &, const Tree &);
};
Tree::Tree (const char *op, const Tree &l,
            const Tree &m, const Tree &r):
    : node_ (new Ternary_Node (op, l, m, r)) {}
```

99

## Differences from C Implementation

- On the other hand, modifying the original C approach requires changing:

- The original data structures, e.g.,

```
struct Tree_Node {
    enum {
        NUM, UNARY, BINARY, TERNARY
    } tag;
    // same as before
    union {
        // same as before
        // add this
        struct {
            Tree_Node *l, *m, *r;
        } ternary;
    } c;
};
#define ternary c.ternary
```

- and many parts of the code, e.g.,

```
void print_tree (Tree_Node *root) {
    // same as before
    case TERNARY: // must be TERNARY.
        printf "(";
        print_tree (root->ternary.l);
        printf "%c", root->op[0];
        print_tree (root->ternary.m);
        printf "%c", root->op[1];
        print_tree (root->ternary.r);
        printf (")"); break;
    // same as before
}
```

100

## Summary

- OO version represents a more complete modeling of the problem domain
  - e.g., splits data structures into modules that correspond to “objects” and relations in expression trees
- Use of C++ language features simplify the design and facilitate extensibility
  - e.g., the original source was hardly affected
- Use of patterns helps to motivate and justify design choices

## Summary (cont'd)

- Potential Problems with OO approach
  - Solution is very “data structure rich”
    - ▷ e.g., Requires configuration management to handle many headers and .C files!
  - May be somewhat less efficient than original C approach
    - ▷ e.g., due to virtual function overhead
  - In general, however, virtual functions may be no less inefficient than large **switch** statements or **if/else** chains...
  - As a rule, be careful of micro vs. macro optimizations
    - ▷ i.e., always profile your code!

## Case Study 2: Spell Checker

### Example

- System Description
  - *Collect words from the named document, and look them up in a main dictionary or a private, user-defined dictionary composed of words. Display words on the standard output if they do not appear in either dictionary, or cannot be derived from those that do appear by applying certain inflections, prefixes, or suffixes*
- We first examine the *algorithmic* approach, then the *object-oriented* approach
  - Note carefully how changes to the specification affect the design alternatives in different ways...

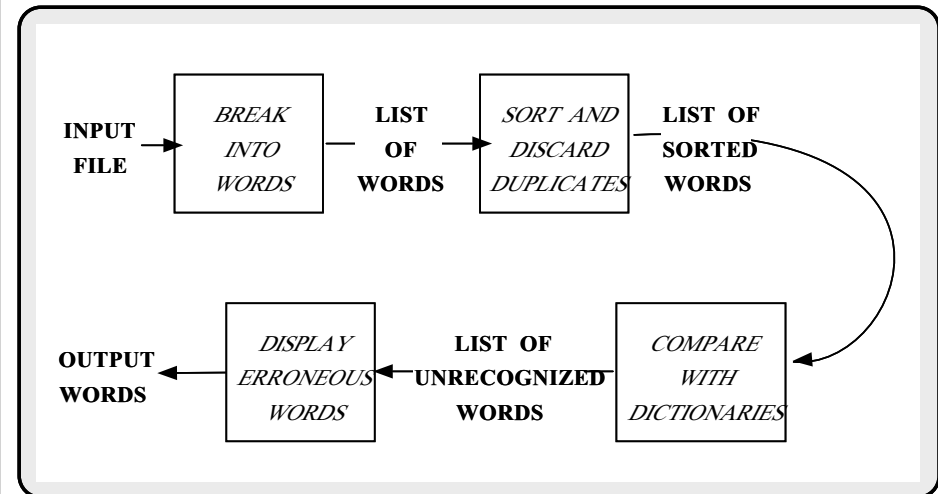
## High-level Application Description

- Pseudo-code algorithmic description
  1. Get document file name
  2. Splits document into words
  3. Look up each word in the main dictionary and a private dictionary
    - (a) If the word appears in either dictionary, or is derivable via various rules, it is deemed to be spelled correctly and ignored
    - (b) Otherwise, the “misspelled” word is output
- Note, avoid the temptation to directly refine the algorithmic description into the software architecture...

## Program Requirements

- Initial program requirements and goals:
  1. Must handle ASCII text files
  2. Document must fit completely into main memory
  3. Must run “quickly”
    - Note, document is processed in “batch” mode
  4. Must be smart about what constitutes misspelled words (that’s why we need prefix/suffix rules and a private dictionary)
- Two common mistakes:
  1. Failure to flag misspelled words
  2. Incorrectly flag correctly spelled words

## Data Flow Diagram



- While this diagram is useful for “describing” high-level flow of data and control, avoid temptation to refine it into system design and implementation...

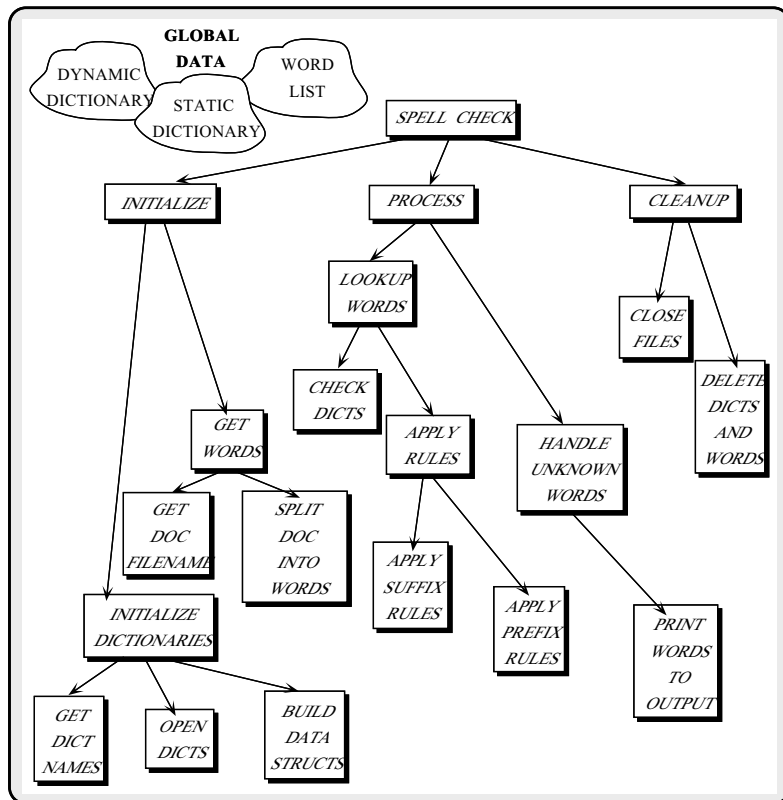
## Algorithmic Design

- Spell checker program is organized according to activities carried out during program execution
  - *i.e.*, system is completely specified by the functions that it performs
- Function refinement precedes and guides data refinement
- *Important questions:*
  1. How is design affected by subsequent changes to the specification and/or implementation?
  2. How reusable are the algorithmic components developed via the approach?

## Algorithmic Design (cont'd)

- Top-down, iterative “step-wise” refinement of functionality:
  1. Break the overall “top” system function into sub-functions
  2. Determine data flow between these functions, *then* determine data structures
  3. Iterate recursively over subfunctions until implementation is immediate and “obvious”
- Structure chart shows function hierarchy and data flow
  - Hierarchical organization is a tree with one functional activity per node

## Algorithm Design Structure



109

## Algorithm Design Program

- Example program

```

struct List {
    char *word;
    struct List *next, *prev;
} *list = 0;
extern struct Static_Dictionary main_dict;
extern struct Dynamic_Dictionary private_dict;

int main (int argc, char *argv[]) {
    initialize (); /* Init files, dictionaries and build list */
    process (); /* Perform lookups */
    cleanup (); /* Close files and deallocate resources */
}

int process (void) {
    for (struct List *ptr = list;
        ptr != 0; ptr = ptr->next)
        if (static_lookup (&main_dict, ptr->word) ||
            dynamic_lookup (&private_dict, ptr->word)) {
            struct List *tmp = ptr;
            ptr->prev->next = ptr->next;
            free ((char *) tmp);
        }
    handle_unknown_words ();
}
  
```

110

## Advantages of Algorithmic Design

- Suitable for small-scale, algorithmic-intensive programs
  - e.g., Eight-Queens problem, Towers of Hanoi, 8-tiles problem, sort, and searching, etc.
- Easy to understand for small problems
  - Since system structure matches verbal, algorithmic description
- “Intuitive” to many designers and programmers
  - Due to emphasis in early training...

## Disadvantages of Algorithmic Design

- Fails to account for long-term system evolution
  - i.e., changes in algorithms and data structures ripple through entire program structure (and the documentation...)
  - Implementation often typified by lack of information hiding, combined with an over-abundance of global variables
    - ▷ These characteristics are not inherent, but are often related...
- Does not promote reusability
  - The design is specifically tailored for the requirements and specifications of a particular application
- Data structure aspects are often underemphasized
  - They are postponed until activities have been defined and ordered



## Object Oriented Design

- Development begins with extensive domain analysis on the problem space
  - *i.e.*, OOD is not a “cookbook” solution
- Decompose the spell checker by *classes* and *objects*, not by overall processing *activities*
- Organize the program to hide implementation details of information that is likely to change
  - *i.e.*, use abstract data types and information hiding
- The order of overall system *activities* are not considered until later in the design phase
  - However, activities are *not* ignored!

## Object Oriented Design (cont'd)

- Note, at first glance the object-oriented design appears to be incomplete since it does not seem to address the overall system *actions*...
- However, this is intentional, and supports the software design principle of “underspecification”
  - The goal is to develop reusable components that support a “program family” of potential solutions to this and other related problems
- In fact, the main processing algorithm may be quite similar in both algorithmic and OO solutions...

## Key Challenges of OO Design

- A key challenge facing developers is finding the objects and classes
  - One approach: *Use parts of speech in requirements specification statements to:*
    1. Identify the Objects
    2. Identify the Operations and Attributes
    3. Establish the Interactions and Visibility
  - Note, this is not perfect, but it is a good place to start...
    - ▷ *i.e.*, apply this methodology at various levels of abstraction during development

## Classifying Parts of Speech

- Example: **spell checker**
  - *Collect **words** from the **named document**, and look them up in a **main dictionary** or a **private, user-defined dictionary** composed of **words**. Display **words** on the **standard output** if they do not appear in either **dictionary**, or cannot be *derived* from those that do appear by *applying* certain **inflections, prefixes, or suffixes**.*
- Relevant parts of speech:
  - **common nouns** → classes
  - **proper nouns** → objects
  - *verbs* → actions on objects

## Identifying Classes and Objects

- Common noun → class
  - *e.g.*, **spell checker**, **dictionary**, **document**, **words**, **output**
- Proper noun or direct reference → object
  - **named document**, **main dictionary**, **private dictionary**, **standard output**
- Describe using Booch or Rumbaugh notation, CRC cards (“class, responsibility, collaborators”), C++ classes, etc.

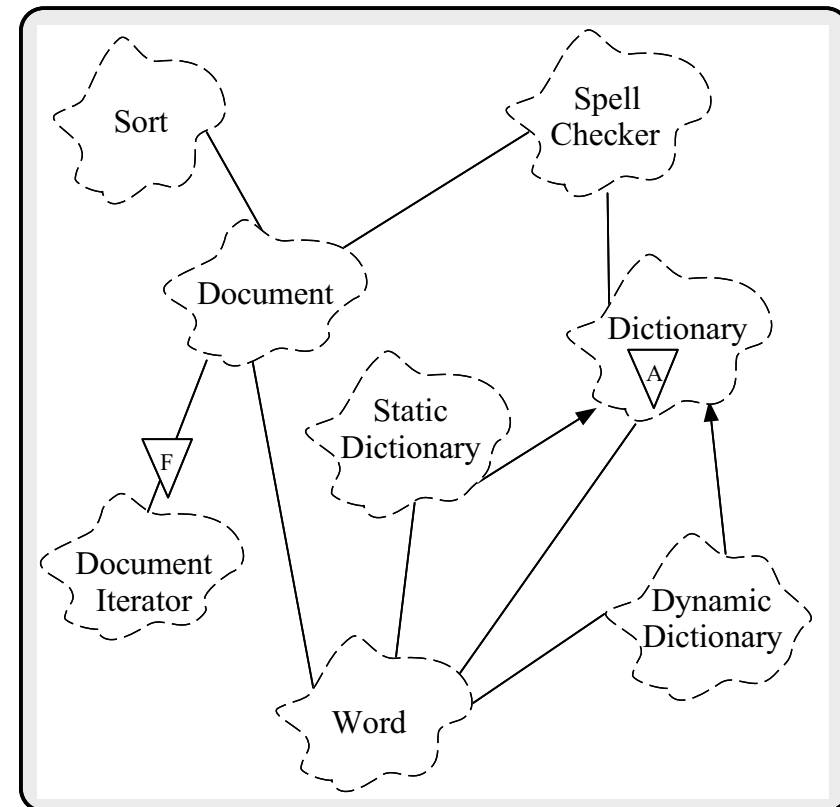
## Identifying Operations and Attributes

- Verb → operations performed on a class or by an object of a class
  - *e.g.*, **collect** (document), **look up** (dictionary), **display** (word)
- Adverb → constraint on an operation
  - *e.g.*, **insert\_quickly** (*i.e.*, no range checking)
- Adjective → attribute of an object
  - *e.g.*, “**large**” dictionary → size field
- Object of verb → object dependencies
  - *e.g.*, “**A dictionary composed of words**”

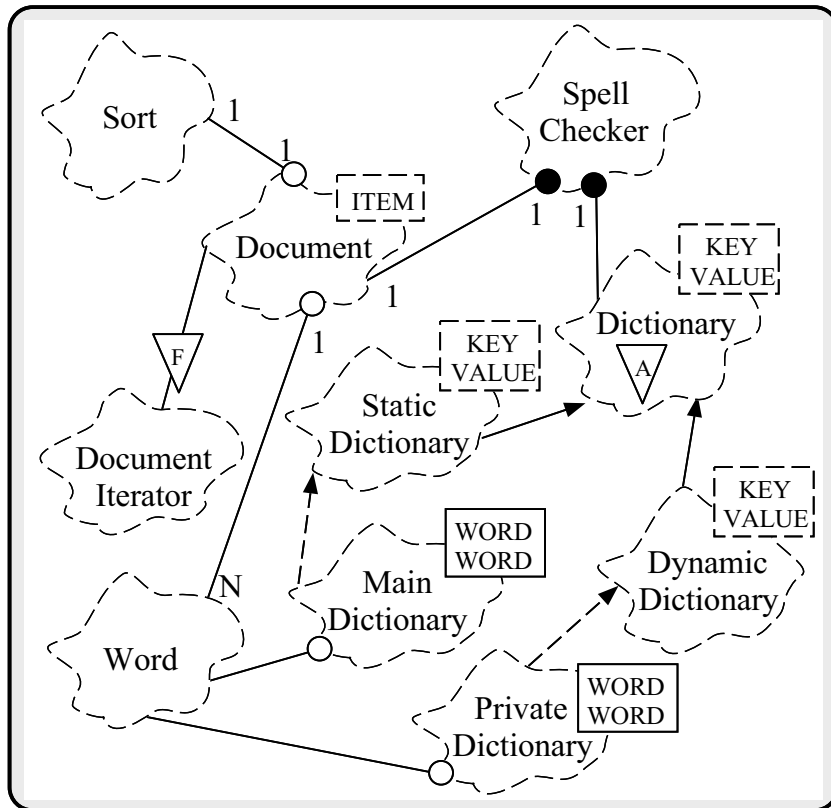
## Comments

- Visibility should satisfy dependencies and no more
  - In general, reduce global visibility, de-emphasize coupling, emphasize cohesion
  - In particular, Document and Dictionary shouldn't be visible outside context of Spell\_Checker...
- Develop a set of diagrams that graphically illustrate class, object, module, and process relationships from various perspectives

## High-level Class Diagram

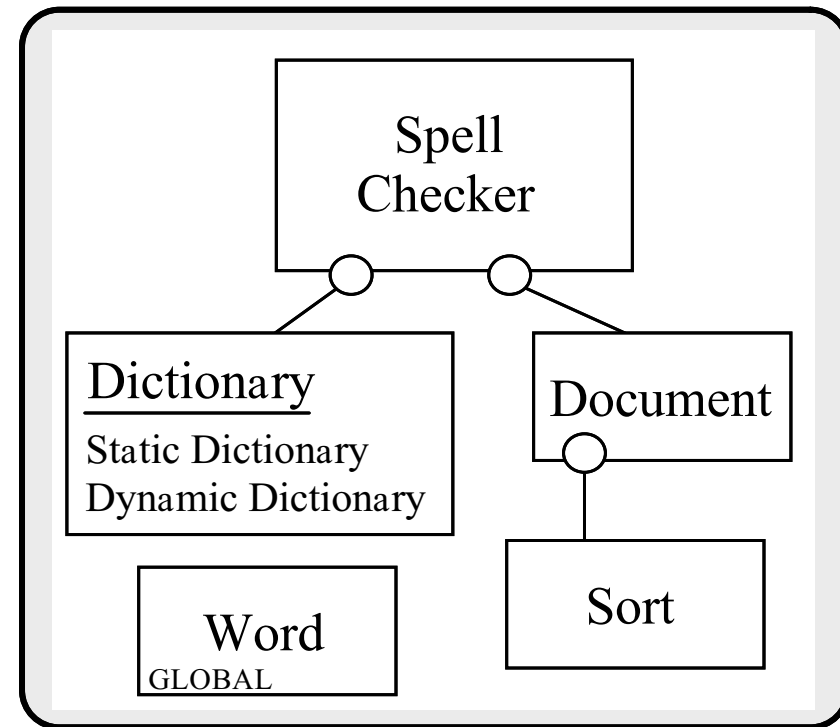


## Detailed Class Diagram



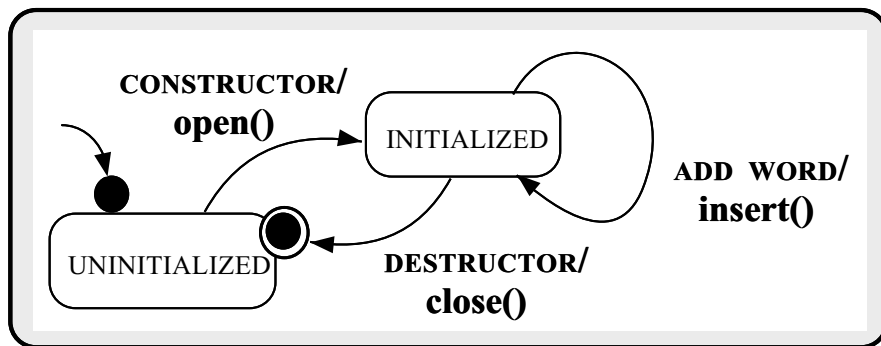
121

## Class Categories



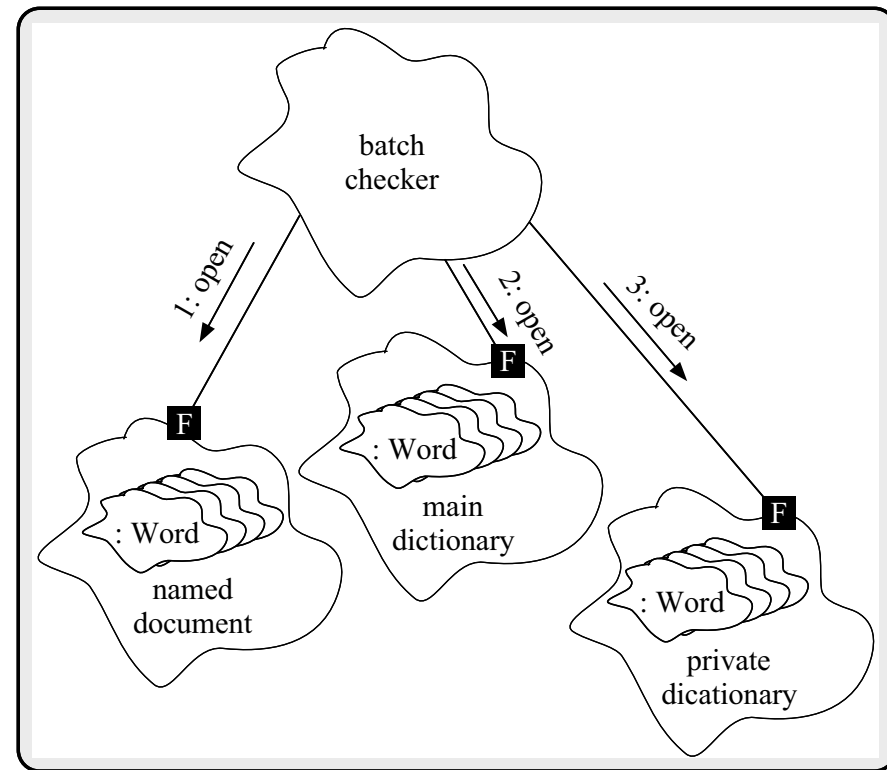
122

## State Machine Diagram for Dictionary class



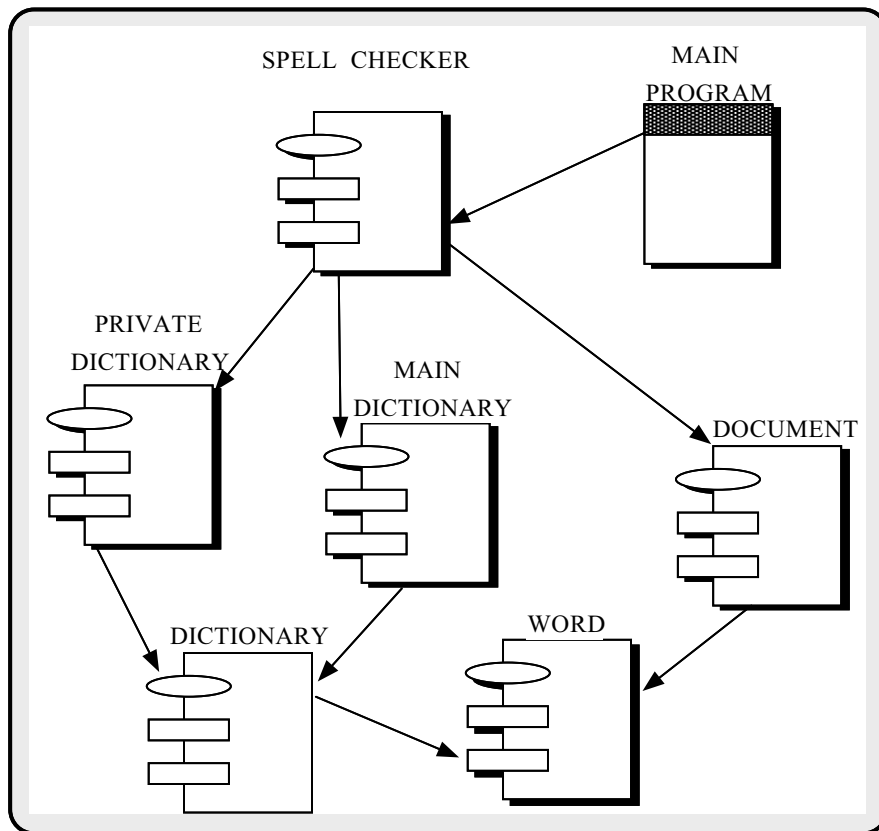
123

## Object Diagram



124

## Module Diagram



## General Class Descriptions

- Building block classes (abstract notation for class interface description)

NAME	Word
ACCESS	Exported
CARDINALITY	Unlimited
MEMBERS	construct/destruct
	insert/remove characters
	clone
	concatenate
	compare
	...

NAME	Document
ACCESS	Exported
CARDINALITY	Unlimited
MEMBERS	construct/destruct
	next word iterator
	sort
	...

NAME	Spell_Checker
ACCESS	Exported
CARDINALITY	Unlimited
MEMBERS	construct/destruct
	spell_check
	...

## General Class Descriptions (cont'd)

- Building block classes (cont'd)

NAME	Dictionary
QUALIFICATIONS	Abstract class
ACCESS	Exported
CARDINALITY	Unlimited
MEMBERS	construct/destruct
open/close	

insert word  
find word  
remove word  
next word iterator  
...

NAME	Dynamic Dictionary
ACCESS	Exported
CARDINALITY	Unlimited
SUPERCLASS	Dictionary
MEMBERS	construct/destruct
	...

NAME	Static Dictionary
ACCESS	Exported
CARDINALITY	Unlimited
SUPERCLASS	Dictionary
MEMBERS	construct/destruct
	...

## Concrete Class Descriptions

- Building block classes (C++ notation for class interface description)

```
class Word {
public:
    Word (char *);
    Word (void);
    int insert (int index, char c);
    int clone (Word &);
    int concat (const Word &);
    int compare (const Word &);
    // ...
};
```

```
template <class ITEM>
class Document {
public:
    Document (void);
    ~Document (void);
    int open (char filename[]);
    int sort (int options);
    // ...
};
```

```
template <class ITEM>
class Document_Iterator {
public:
    Document_Iterator (Document &);
    int next_item (ITEM &);
    // ...
};
```



## Concrete Class Descriptions (cont'd)

- Building block classes (C++ notation for class interface description)

```
namespace ACE_Dictionary {
template <class KEY, class VALUE>
class Dictionary {
public:
    virtual int open (char filename[]) = 0;
    virtual find (KEY, VALUE &) = 0;
    virtual insert (KEY, VALUE &) = 0;
    virtual remove (KEY) = 0;
    // ...
};

template <class KEY, class VALUE>
class Dynamic_Dictionary : public Dictionary {
public:
    virtual int open (char filename[]);
    virtual find (KEY, VALUE &);
    // ...
};

template <class KEY, class VALUE>
class Static_Dictionary : public Dictionary {
public:
    virtual int open (char filename[]);
    virtual find (KEY, VALUE &);
    // ...
};
}
```

## Concrete Class Descriptions (cont'd)

- Building block classes (C++ notation for class interface description)

```
#include "Document.h"
#include "Static_Dictionary.h"
#include "Dynamic_Dictionary.h"

using namespace ACE_Dictionary;

typedef Static_Dictionary<Word, Word> Main_Dictionary;
typedef Dynamic_Dictionary<Word, Word> Private_Dictionary;

class Spell_Checker {
public:
    Spell_Checker (char *doc_name,
                  char *main_dict_name,
                  char *private_dict_name);
    ~Spell_Checker (void);
    int open (char *doc_name, char *main_dict_name,
             char *private_dict_name);
    INT spell_check (ostream &standard_output);
private:
    Document<Word> named_document;
    Main_Dictionary main_dictionary;
    Private_Dictionary private_dictionary;
};
```

## Spell checker Implementation

- Main class

```
Spell_Checker::Spell_Checker (char *doc_name,
                              char *main_dict_name,
                              char *private_dict_name)
{
    if (named_document.open (doc_name) == -1
        || main_dictionary.open (main_dict_name) == -1
        || private_dictionary.open (private_dict_name) == -1) {
        cerr << "initialization problem";
        ::exit (1)
    }
}

int
Spell_Checker::spell_check (ostream &standard_output)
{
    int result = 0;
    Word word;
    this->named_document.sort (REMOVE_DUPS);
    for (Document_Iterator<Word> doc_iter (this->named_document);
         doc_iter.next_item (word) != -1; )
        if (this->main_dictionary.find (word) != -1
            || this->private_dictionary.find (word) != -1)
            continue; /* found word */
        else {
            standard_output.write (word);
            /* erroneous word */
            result = -1;
        }
}

// ...
```

## Spell Checker Driver

- The main program is:

```
int main (int argc, char *argv[])
{
    if (argc != 4) {
        cerr << "usage: " << argv[0]
              << ": doc-name, main-dict, private-dict"
              << endl;
        ::exit (1);
    }
    Spell_Checker batch_checker (argv[1], argv[2], argv[3]);
    if (batch_checker.spell_check (cout) == -1)
        return -1;
    else
        return 0;
}
```

- Note how the OO decomposition uses essentially the same algorithm as the original spell-checker...

– However, the architecture is *totally* different

## Advantages of OO Design

- Increased modularity:
  1. Easier to understand the components in isolation, since data coupling and visibility have been reduced
    - e.g., modules and classes are composed of related activities
  2. More adaptive to specification and implementation changes, since changes are localized
    - e.g., most changes occur in representations, rather than interfaces
    - By hiding objects' representational details, changes will not ripple through design (unless class specification changes)

## Advantages of OO Design

- Class data and member functions are equally emphasized
  - However, higher-level structuring of activities is postponed
- Object behavior is independent of temporal ordering
  - i.e., the *shopping list* approach
  - Easier to reuse and extend classes in other systems, since emphasis is on stable interfaces
    - ▷ e.g., reuse sort from system sort application

## Disadvantages of OO Design

- Certain problem domains do not necessarily benefit from an object-oriented approach
  - *e.g.*, mathematical routines for numerical analysis, where there is no need for shared state...
- Requires more work in the upstream activities
  1. *e.g.*, analysis, modeling, and architectural design to determine architectural components, relations, and interfaces
  2. Often not as intuitive to determine the objects (without training and practice)
- Requires an object-oriented language for best results

## Potential Modifications

- Make the program run interactively, rather than in “batch” mode
  - *e.g.*, integrate with a text editor and make the program work on user-selected regions of the document (*e.g.*, GNU emacs):
    1. Query the user to check if an unrecognized word is misspelled
    2. If it is misspelled then
      - (a) Replace the word in the document
      - (b) Potentially add the word to the private dictionary, if user specifies this action
    3. Produce an updated private dictionary
- Remove arbitrary limits on input document size
  - *i.e.*, does not need to fit into memory

## Potential Modifications

- Make the program handle multiple input files
- Make the program handle multiple dictionaries
- Modify the program to perform other text oriented tasks, *e.g.*,
  - Build a document index or cross-referencer
  - Build an interactive thesaurus
- Make the program work on other types of files, *e.g.*,
  - LaTeX or TeX files
  - nroff files
  - MS Word files
  - postscript or dvi files

## Parting Thought

- Sometimes the “best” design is the least elaborate one:

```
% cat tex-spell-check
dextex $1 | \           # strip tex formatting commands
tr A-Z a-z | \         # map upper case to lower case
tr -cs a-z '\012' | \   # remove all non-words
sort -u >! /tmp/words    # remove duplicates
                        # omit lines only in filename 2
comm -2 /tmp/words /usr/dict/words
```

- Advantage:
  - Easy to get right (once you understand UNIX tools ;-)), since it is very decoupled...
- Disadvantages
  - Doesn't work very well for prefixes/suffixes
  - Slow...(many processes, many stages)

## Summary

- Object-oriented design differs from algorithmic design in several ways:
  - Structure of the system is organized around classes/objects rather than functions
  - Objects are typically more “complete” abstractions than are functions (e.g., they include data emphasis as well as control flow emphasis)
  - Algorithmically decomposed components have *verb* names, while object-oriented components have *noun* names
- Advantages of OO are most evident in
  1. *Large-scale systems*
  2. *Evolving systems*