Strategy Pattern (cont'd)

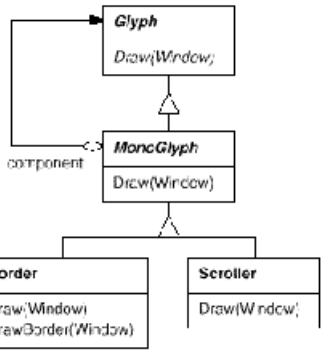
- Consequences (continued)
  - clients must be aware of different strategies
    - when initializing objects
  - proliferation of instances at run-time
    - each glyph has a strategy object with formatting information
    - if strategy is stateless, share strategy objects

Adding scroll bars and borders

- Where do we define classes for scrollbars and borders?
- Define as subclasses of Glyph
  - scrollbars and borders are displayable objects
  - supports notion of transparent enclosure
    - clients don't need to know whether they are dealing with a component or with an enclosure
- Inheritance increases number of classes
  - use composition instead ("has a")

Monoglyph class

```
void MonoGlyph::Draw (Window* w)  {
   component->Draw(w);
}
```

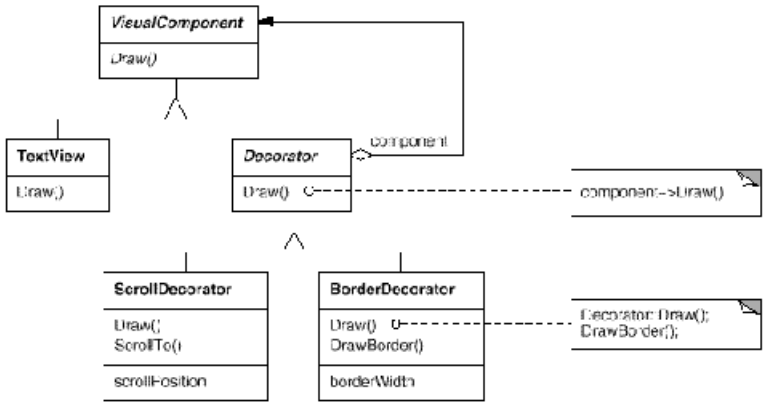



```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```


Decorator Pattern

- Name
  - Decorator (aka Wrapper)
- Applicability
  - add responsibilities to objects dynamically and transparently
  - handle responsibilities that can be withdrawn at run-time

Decorator Pattern: Structure



Decorator Pattern (cont'd)

- Advantages
  - fewer classes than with static inheritance
  - dynamic addition/removal of decorators
  - keeps root classes simple
- Disadvantages
  - proliferation of run-time instances
  - abstract Decorator must provide common interface
- Tradeoffs:
  - useful when components are lightweight
  - otherwise use Strategy

Multiple look-and-feel standards

- Change Lexi's look-and-feel at run-time
- Obvious solution has clear disadvantages
  - use distinct class for each widget and standard
  - let clients handle different instances for each standard
- Problems:
  - proliferation of classes
  - can't change standard at run-time
  - code for look-and-feel standard visible to clients
- Code example:
    - Button* pb = new MotifButton; // bad
    - Button* pb = guiFactory->createButton();// better

Multiple look-and-feel standards (cont'd)

- Solution:
  - define abstract class GUIFactory with creation methods (deferred) for widgets
  - concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
    - MotifFactory, MacFactory, etc.
  - must still specialize each widget into subclasses for each look-and-feel standard

Class diagram for GUIFactory

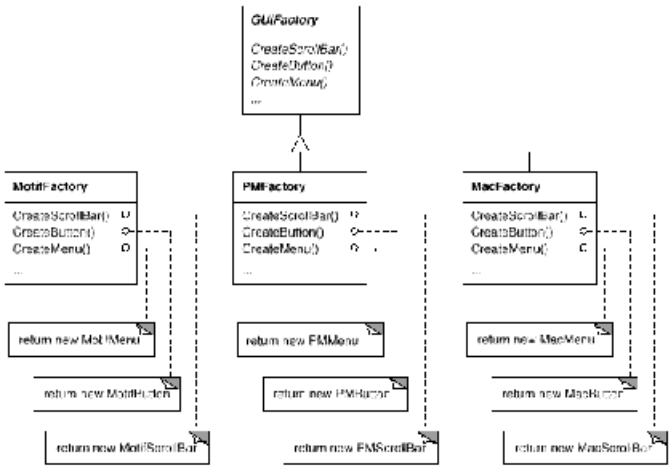
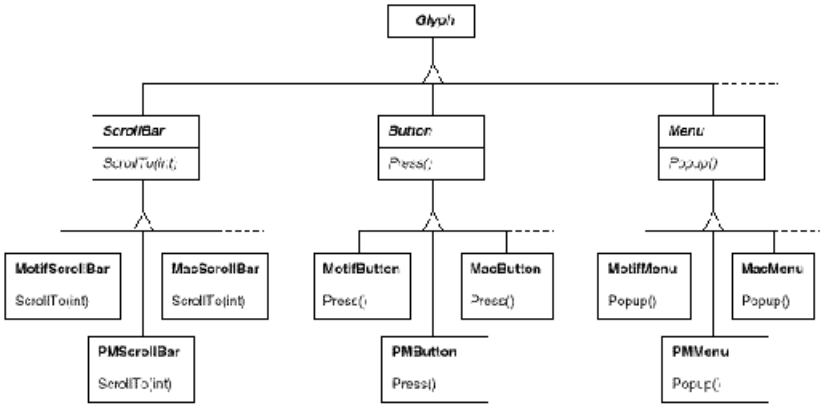


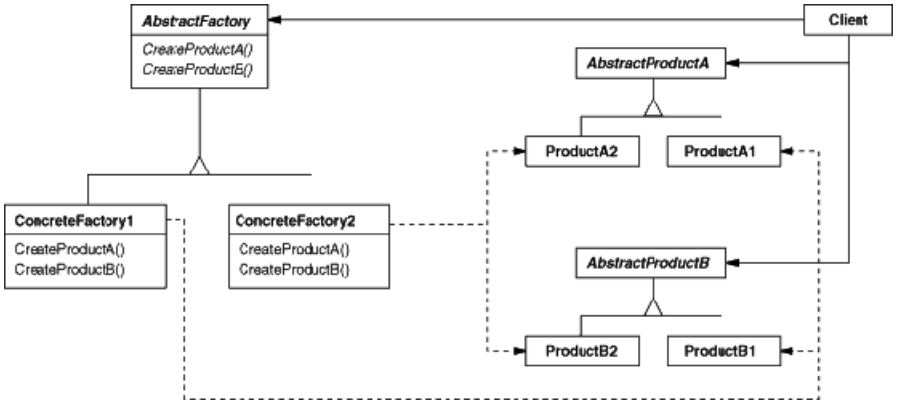
Diagram for product classes



Abstract Factory pattern

- Name
  - Abstract Factory or Kit
- Applicability
  - different families of components (products)
  - must be used in mutually exclusive and consistent way
  - hide existence of multiple families from clients

Structure of Abstract Factory



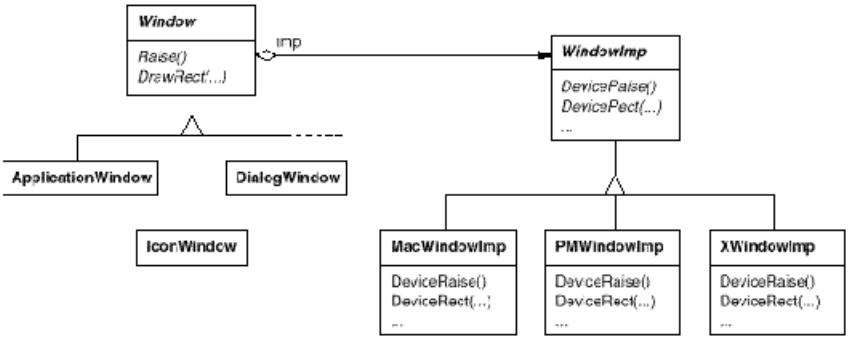
Abstract Factory (cont'd)

- Consequences
  - isolate creation and handling of instances from clients
  - changing look-and-feel standard at run-time is easy
    - reassign a global variable;
    - recompute and redisplay the interface
  - enforces consistency among products in each family
  - adding new family of products is difficult
    - have to update all factory classes

Multiple Window Systems

- Want portability to different window systems
  - similar to multiple look-and-feel problem, but different vendors will build widgets differently
- Solution:
  - define abstract class Window, with basic window functionality (e.g., draw, iconify, move, resize, etc.)
  - define concrete subclasses for specific types of windows (e.g., dialog, application, icon, etc.)
  - define WindowImp hierarchy to handle window implementation by a vendor

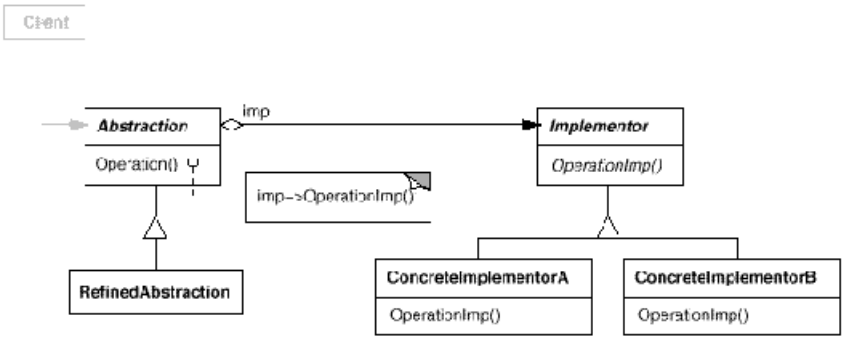
Implementation



Bridge Pattern

- Name
  - Bridge or Handle or Body
- Applicability
  - handles abstract concept with different implementations
  - implementation may be switched at run-time
  - implementation changes should not affect clients
  - hide a class's interface from clients
- Structure: use two hierarchies
  - logical one for clients,
  - physical one for different implementations

Structure of Bridge Pattern



Bridge Pattern

- Consequences:
  - decouple interface from impl.and representation
  - change implementation at run-time
  - improve extensibility
    - logical classes and physical classes change independently
    - hides implementation details from clients
      - sharing implementation objects and associated reference counts

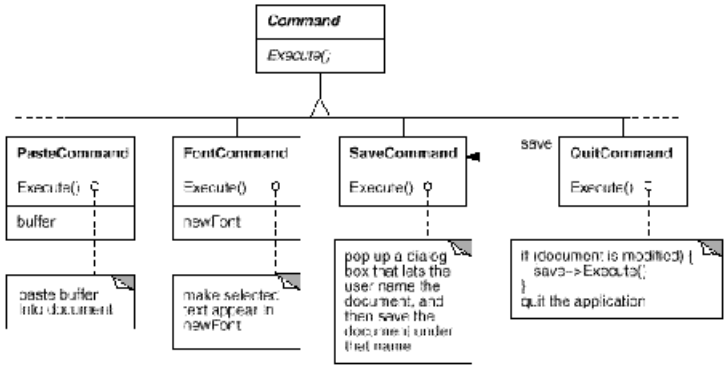
Supporting User Commands

- Support execution of Lexi commands
  - GUI doesn't know
    - who command is sent to
    - command interface
- Complications
  - different commands have different interfaces
  - same command can be invoked in different ways
  - Undo and Redo for some, but not all, commands (print)

Supporting User Commands (cont'd)

- An improved solution
  - create abstract "command" class
    - command must have reversible method
  - create action-performing glyph subclass
  - delegate action to command
- Key ideas
  - pass an object, not a function
  - pass context to the command function
  - store command history

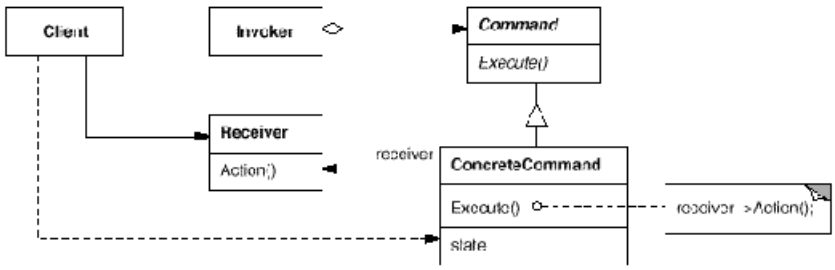
Command Objects



Command Pattern

- Name
  - Command or Action or Transaction
- Applicability
  - parameterize objects by actions they perform
  - specify, queue, and execute requests at different times
  - support undo by storing context information
  - support change log for recovery purposes
  - support high-level operations
    - macros

Structure of Command Pattern



Command Pattern

- Consequences:
  - decouple receiver and executor of requests
    - Lexi example: Different icons can be associated with the same command
  - commands are first class objects
  - easy to support undo and redo
    - must add state information to avoid hysteresis
  - can create composite commands
    - Editor macros
  - can extend commands more easily

Command Pattern

- Implementation notes
  - how much should command do itself?
  - support undo and redo functionality
    - operations must be reversible
    - may need to copy command objects
    - don't record commands that don't change state
  - avoid error accumulation in undo process

Spell-Checking and Hyphenation

- Must do texual analysis
  - multiple operations and implementations
- Must add new functions and operations easily
- Must efficiently handle scattered information and varied implementations
  - different traversal strategies for stored information
- Should separate traversal actions from traversal

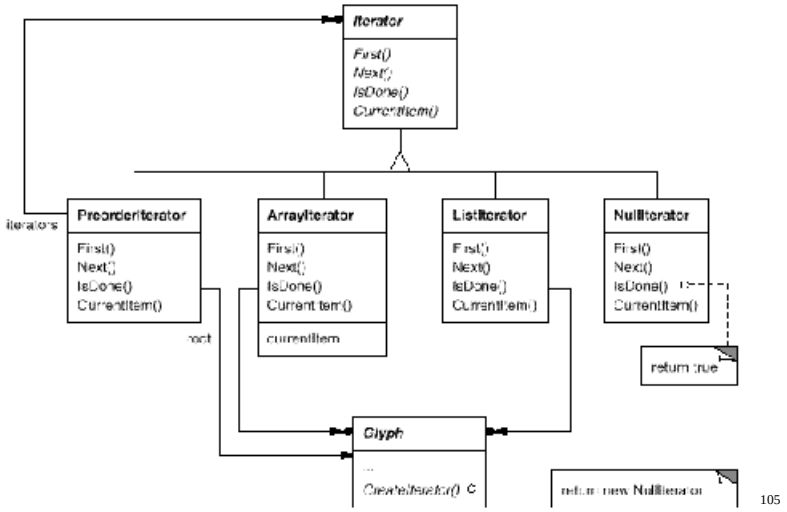
Iterator Pattern

- Name
  - Iterator or Cursor
- Applicability
  - access aggregate objects without exposing internals
  - support multiple traversal strategies
  - uniform interface for traversing different objects

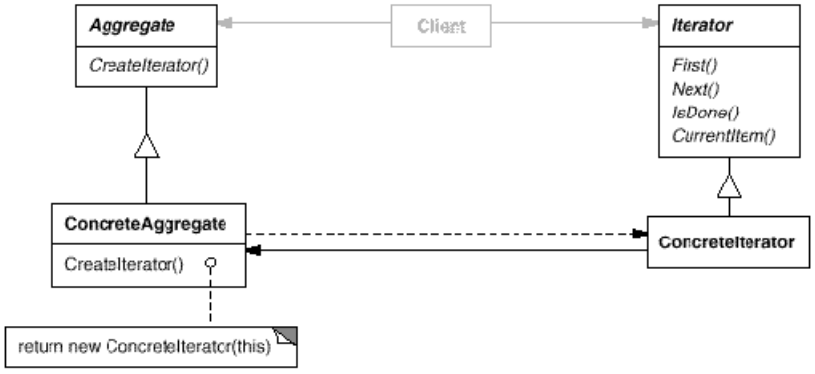
Iterator Pattern

- Key ideas
  - separate aggregate structures from traversal protocols
  - support addition of traversal functionality
  - small interfaces for aggregate classes,
  - multiple simultaneous traversals
- Pattern structure
  - abstract Iterator class defines traversal protocol
  - concrete Iterator subclasses for each aggregate class
  - aggregate instance creates instances of Iterator objects
  - aggregate instance keeps reference to Iterator object

Structure of Iterator Pattern



Structure of Iterator Pattern



Iterator Pattern (cont'd)

- Consequences
  - support different kinds of traversal strategies
    - just change Iterator instance
  - simplify aggregate's interface
    - no traversal protocols
  - supports simultaneous traversals

Iterator Pattern (cont'd)

- Implementation issues
- Who controls iteration?
  - external vs. internal iterators
    - external:
      - client controls the iteration via "next" operation
      - very flexible
      - some operations are simplified - logical equality and set operations are difficult otherwise
    - internal:
      - Iterator applies operations to aggregate elements
      - easy to use
      - can be difficult to implement in some languages

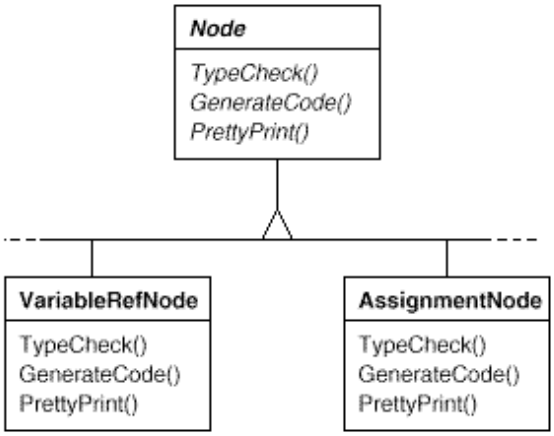
Iterator Pattern (cont'd)

- Who defines the traversal algorithm?
  - Iterator itself
    - may violate encapsulation
  - aggregate
    - Iterator keeps only state of iteration
- How robust is the Iterator?
  - are updates or deletions handled?
  - don't want to copy aggregates
  - register Iterators with aggregate and clean-up as needed
  - synchronization of multiple Iterators is difficult

Implementation of Operations

- Compiler represents prog. as abstract syntax tree
  - Different nodes for different components for a language
    - Different classes for assignment stmts, vars, expressions.
  - Operations on abstract syntax trees
    - Type checking, code generation, pretty-printing, etc

Traditional



Problems

- Distributing operations across the node classes leads to a system that's:
  - Hard to understand, maintain, and change
- Adding a new op will require recompiling all classes
- Would be better if each new operation
  - Could be added separately, and
  - Node classes were independent of op that apply to them

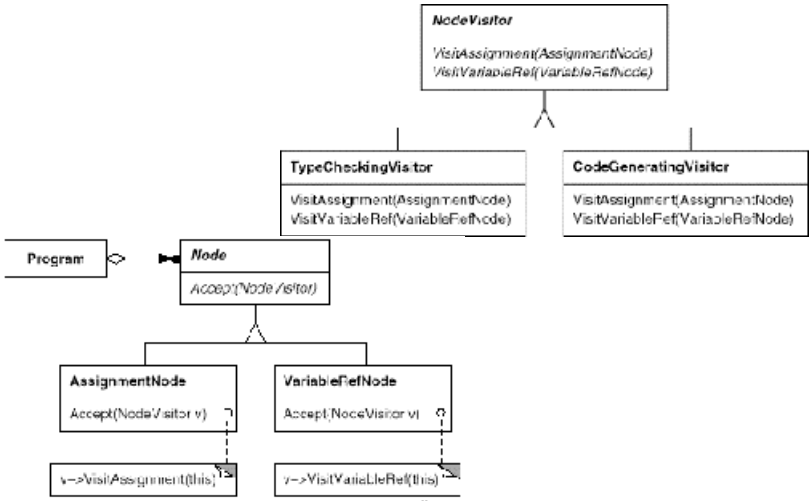
Visitor pattern

- Name
  - Visitor or double dispatching
- Applicability
  - related objects must support different operations and actual op depends on both the class and the op type
  - distinct and unrelated operations pollute class defs
  - object structure rarely changes, but ops changed often

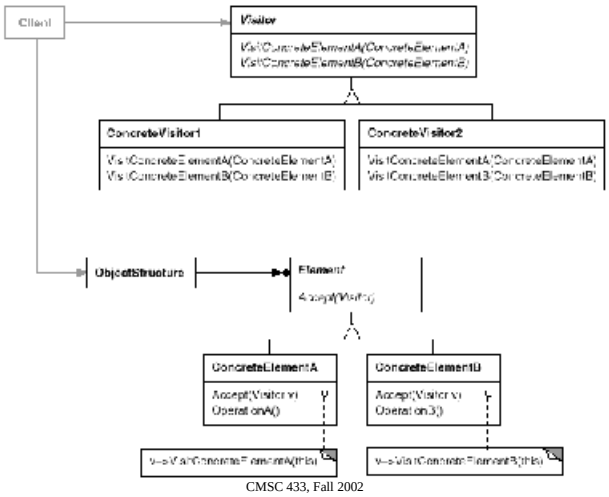
Visitor Pattern (cont'd)

- Structure
  - define two class hierarchies,
    - one for object structure
    - one for each operation family (called visitors)
  - compiler example:
    - Object subclasses VariableRef, AssignmentNode.
    - Operation subclasses TypeCheck, GenerateCode, PrettyPrint

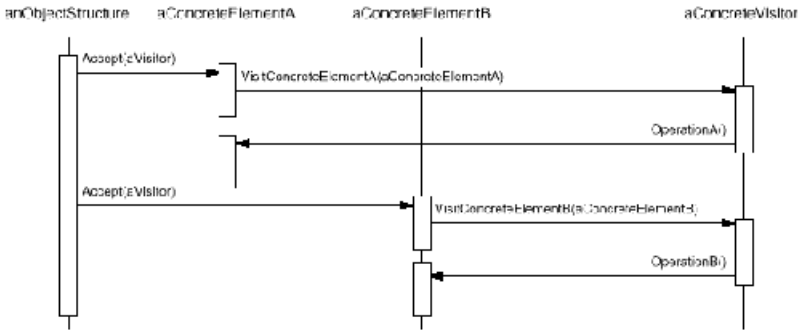
Visitor pattern



Structure of Visitor Pattern



Use of Visitor Pattern



Visitor Pattern (cont'd)

- Consequences
  - adding new operations is easy
    - add new operation subclass with a method for each concrete element class
    - easier than modifying every element class
  - gathers related operations and separates unrelated ones
  - adding new concrete elements is difficult
    - must add a new method to each concrete Visitor subclass
  - allows visiting across class hierachies
    - Iterator needs a common superclass
  - can accumulate state rather than pass it a parameters

Visitor Pattern (cont'd)

- Implementation issue:
  - Who is responsible for traversing object structure?
- Plausible answers:
  - visitor
    - must replicate traversal code in each concrete visitor
  - object structure
    - define operation that performs traversal while applying visitor object to each component
  - Iterator
    - Iterator sends message to visitor with current element as arg

Pattern hype

- Patterns get a lot of hype and fanatical believers
  - We are going to have a design pattern reading group, and this week we are going to discuss the Singleton Pattern!
- Patterns are sometimes wrong or inappropriate for a particular language or environment
  - Patterns developed for C++ can have very different solutions in Smalltalk or Java

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.