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
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

• 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

[Diagram of 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

- 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 textual 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 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

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
• 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

• Who defines the traversal algorithm?
  – Iterator itself
    • may violate encapsulation
  – aggregate (in a “cursor”)
    • 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