Design Patterns for OO Development

NOTE: I'm still working on slides 52-74

Objectives

- History and the relevance of design patterns
- Detailed coverage of some popular existing patterns
  - a simple example: The Proxy pattern
  - case study

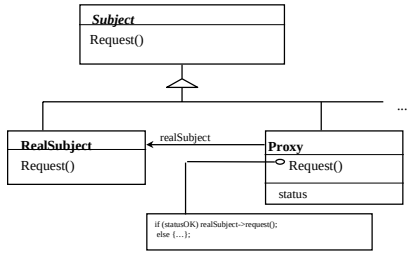
The Proxy Pattern

- Context:
  - situations in which access to an object should be controlled
    - In graphical application, protect objects whose display is expensive or slow (e.g., a picture)
    - In network application, facilitate access to distributed objects (e.g., make it easy for a client to find a server)
    - In OS application, protect objects from unauthorized access

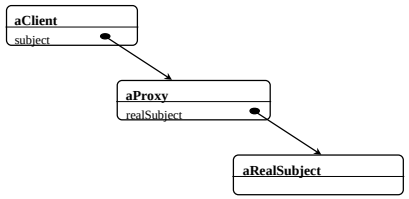
The Proxy Pattern (cont'd)

- Problem:
  - Prevent an object from being accessed directly by its clients
- Solution:
  - Use an additional object, called a proxy
  - Clients access to protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

Class Diagram



Object Diagram



Components of a Pattern

- Pattern name
  - identify this pattern; distinguish from other patterns
  - define terminology
- Pattern alias - *also known as*
- Real-world example
- Context
- Problem

Components of a Pattern (cont'd)

- Solution
  - typically natural language notation
- Structure
  - class (and possibly object) diagram in solution
- Interaction diagram (optional)
- Consequences
  - advantages and disadvantages of pattern
  - ways to address residual design decisions

Components of a Pattern (cont'd)

- Implementation
  - critical portion of plausible code for pattern
- Known uses
  - often systems that inspired pattern
- References - *See also*
  - related patterns that may be applied in similar cases

Taxonomy of Patterns

- Three kinds of patterns, based on purpose:
  - creational patterns
    - describe how to create objects
    - often by delegation of responsibility from an abstract class to concrete subclasses
  - structural patterns
    - describe how to define structures and their relationships
  - behavioral patterns
    - describe object behavior and interactions among objects

Principles Underlying Patterns

- Rely on abstract classes to hide differences between subclasses from clients
  - object class vs. object type
    - class defines how an object is implemented
    - type defines an object's interface (protocol)
- *Program to an interface, not an implementation*

Principles (cont'd)

- Black-box vs. white-box reuse
  - black-box relies on object references, usually through instance variables
  - white-box reuse by inheritance
  - black-box reuse preferred for information hiding, run-time flexibility, elimination of implementation dependencies
  - disadvantages: Run-time efficiency (high number of instances, and communication by message passing)
- *Favor composition over class inheritance*

Principles (cont'd)

- Delegation
  - powerful technique when coupled with black-box reuse
  - Allow delegation to different instances at run-time, as long as instances respond to similar messages
  - disadvantages:
    - sometimes code harder to read and understand
    - efficiency (because of black-box reuse)

Lexi: A Simple GUI-Based Editor

- Lexi is a WYSIWYG editor
  - supports documents with textual and graphical objects
  - scroll bars to select portions of the document
  - be easy to port to another platform
  - support multiple look-and-feel interfaces
- Highlights several OO design issues
- Case study of design patterns in the design of Lexi

Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
- Handle multiple windowing systems
- Support user operations
- Advanced features
  - spell-checking and hyphenation

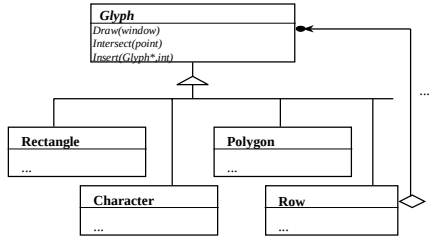
Structure of a Lexi Document

- Goals:
  - store text and graphics in document
  - generate visual display
  - maintain info about location of display elements
- Caveats:
  - treat different objects uniformly
    - e.g., text, pictures, graphics
  - treat individual objects and groups of objects uniformly
    - e.g., characters and lines of text

Structure of a Lexi Document

- Solution:
  - define abstract class Glyph for all displayed objects
  - glyph responsibilities:
    - know how to draw itself
    - knows what space it occupies
    - knows its children and parent
  - use recursive composition for defining and handling complex objects
  - define composition of Glyph as instances of Glyph

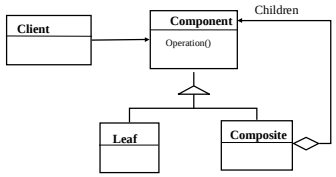
Glyph Class Diagram



The Composite Pattern

- Motivation:
  - support recursive composition in such a way that a client need not know the difference between a single and a composite object (as with Glyphs)
- Applicability:
  - when dealing with hierarchically-organized objects (e.g., columns containing rows containing words …)

Glyph Class Diagram



Composite Pattern (cont'd)

- Consequences:
  - class hierarchy has both simple and composite objects
  - simplifies clients
  - aids extensibility
    - clients do not have to be modified
  - too general a pattern?
    - difficult to to restrict functionality of concrete leaf subclasses

Formatting Lexi Documents

- Handle justification, margins, line breaking, etc.
- Many good algorithms exist;
  - different tradeoffs between quality and speed
  - design decision: implement different algorithms, decide at run-time which algorithm to use
- Goal: maintain orthogonality between formatting and representation (glyphs)
- Solution
  - define root class that supports many algorithms
  - each algorithm implemented in a subclass

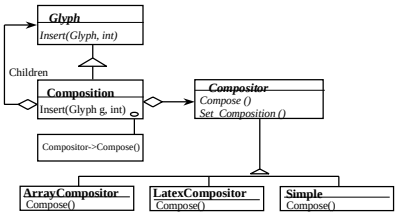
Compositor and Composition

- Relevant design decisions:
  - compositor: class containing formatting algorithm
  - pass objects to be formatted as parameters to compositor methods
  - parameters are instances of a Glyph subclass called Composition
  - uniform interface between formattable objects and compositor algorithms

Compositor and Composition

- Relevant design decisions (cont'd):
  - each Composition instance has a reference to a compositor object
  - when a composition needs to format itself, it sends a message to its compositor instance

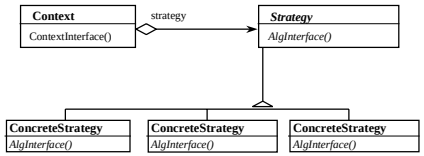
Class Diagram



Strategy Pattern

- Name
  - Strategy (*aka* Policy)
- Applicability
  - many related classes differ only in their behavior
  - many different variants of an algorithm
  - need to encapsulate algorithmic information

Strategy Pattern: Structure



Strategy Pattern (cont'd)

- Consequences
  - clear separation of algorithm definition and use
    - glyphs and formatting algorithms are independent
    - alternative (many subclasses) is unappealing
      - proliferation of classes
      - algorithms cannot be changed dynamically
  - elimination of conditional statements
    - as usual with OO programming

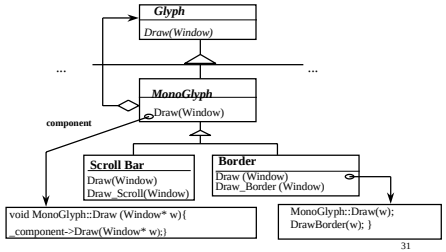
Strategy Pattern (cont'd)

- 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")

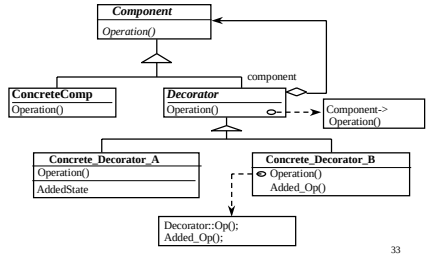
Monoglyph class



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

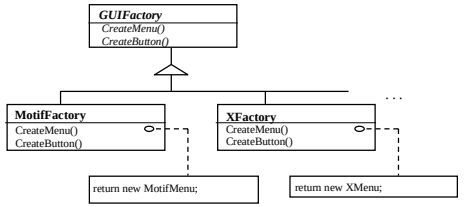
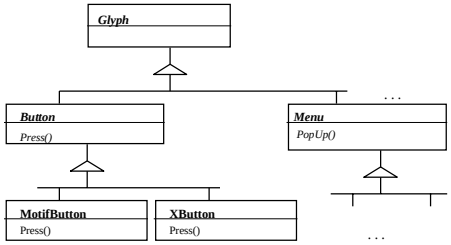


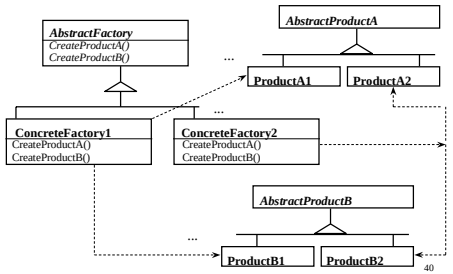
Diagram for product classes



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



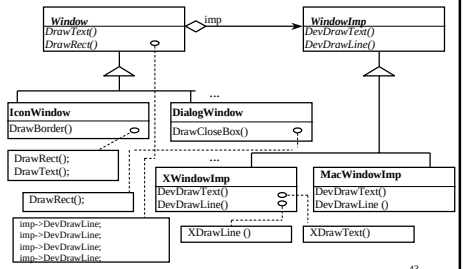
Abstract Factory (cont'd)

- 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

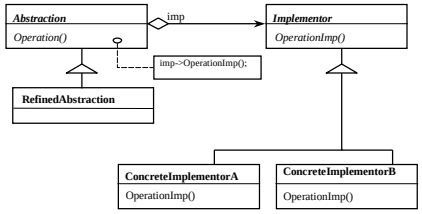
Implementation



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

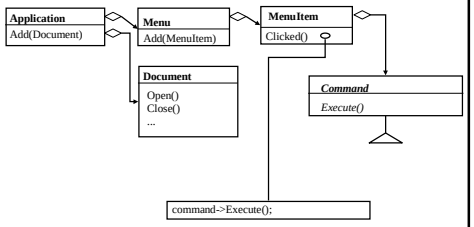
Supporting User Commands

- 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

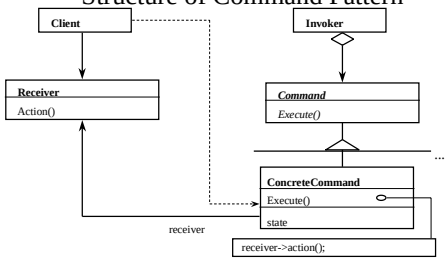
Structure of Command Pattern



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

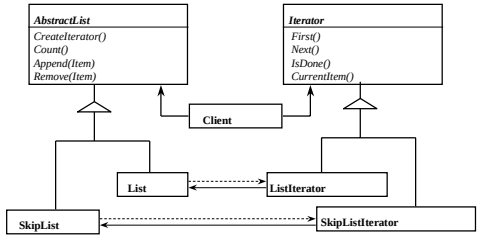
Iterator Pattern

- 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



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 roubust 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

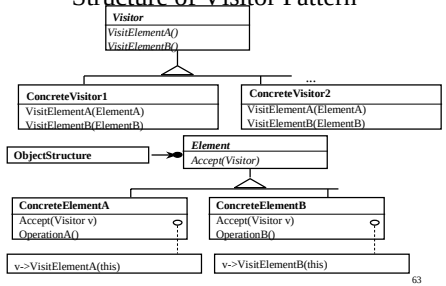
Visitor pattern

- 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 operation family
  - compiler example:
    - Object subclasses VariableRef, AssignmentNode.
    - Operation subclasses TypeCheck, GenerateCode, PrettyPrint

Structure 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
  - visiting across class hierachies
    - *Iterator* needs a common superclass
  - accumulating state

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

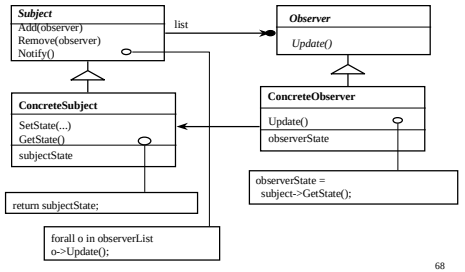
Observer Pattern

- Name
  - Observer (aka Dependents, Publish-Subscribe)
- Applicability
  - changes to master cause updates to dependent objects
  - master doesn't know number or identity of dependents
  - coupling between master and dependents is low

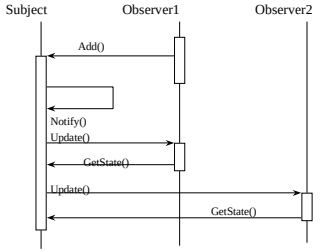
Observer Pattern (cont'd)

- Problem
  - dependent's state must be consistent with master's state
- Solution structure
  - define four kinds of objects:
    - abstract subject
      - maintain list of dependents; notifies them when master changes
    - abstract observer
      - define protocol for updating dependents
    - concrete subject
      - manage data for dependents; notifies them when master changes
    - concrete observers
      - get new subject state upon receiving update message

Structure of Observer Pattern



Run-time behavior of Observer



Observer Pattern (cont'd)

- Consequences
  - low coupling between subject and observers
    - subject unaware of dependents
  - support for broadcasting
    - dynamic addition and removal of observers
  - unexpected updates
    - no control by the subject on computations by observers

Observer pattern (cont'd)

- Implementation issues
  - storing list of observers
    - typically in subject
    - expensive when many subjects have no observers
  - observing multiple subjects
    - typically add parameters to update()
  - who triggers update?
    - State-setting operations of subject
      - Possibly too many updates
    - client
      - Error-prone if an observer forgets to send notification message

Observer pattern (cont'd)

- Implementation issues (cont'd)
  - possibility of dangling references when subject is deleted
    - easier in garbage-collected languages
    - otherwise, have subject notify observers before dying
  - possibility of premature notifications
    - typically, method in Subject subclass calls inherited method which does notification
    - solve by using Template method pattern
      - method in abstract class calls deferred methods, which is defined by concrete subclasses

Observer pattern (cont'd)

- Implementation issues (cont'd)
  - how much information should subject send with *update()* messages?
    - Push model: Subject sends all information that observers may require
      - May couple subject with observers (by forcing a given observer interface)
    - Pull model: Subject sends no information
      - Can be inefficient
  - registering observers for certain events only
    - use notion of an *aspect* in subject
    - Observer registers for one or more aspects

Observer pattern (cont'd)

- Implementation issues (cont'd)
  - complex updates
    - use change managers
    - change manager keeps track of complex relations among (possibly) many subjects and their observers and encapsulates complex updates to observers