

Design Patterns for OO Development

1

Course Objectives

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

2

The Proxy Pattern

- Context:
 - situations in which access to an object should be deferred
 - 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

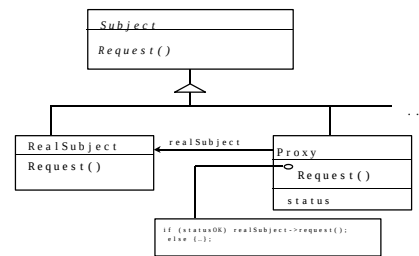
3

The Proxy Pattern (cont'd)

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

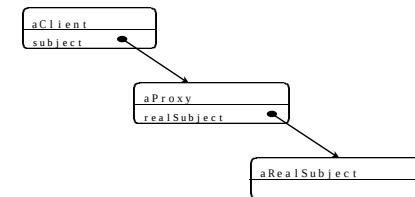
4

Class Diagram



5

Object Diagram



6

Components of a Pattern

- Pattern name
 - identify this pattern; distinguish from other patterns
 - define a lexicon for OO design
- Pattern alias - *also known as*
- Real-world example
- Context
- Problem

7

Components of a Pattern (cont'd)

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

8

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

9

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

10

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*

11

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*

12

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)

13

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

14

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

15

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

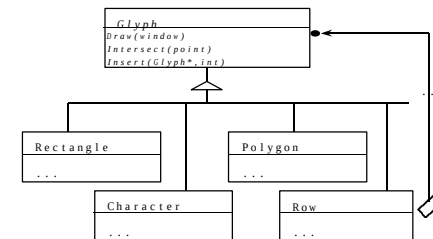
16

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

17

Glyph Class Diagram



18

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 ...)
- Structure: Same as previous picture

19

Composite Pattern (cont'd)

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

20

Composite Pattern (cont'd)

- Implementation Issues:
 - parent references
 - facilitate traversal of hierarchy
 - sharing components
 - save space, but at the price of increased programming complexity (e.g., difficult to implement with a single parent pointer)

21

Composite Pattern (cont'd)

- overgenerality
 - how do components avoid defining methods that don't make sense for some leafs?
 - especially methods that make sense only for composites, e.g., format, group, ungroup?
 - possibly view leafs as components with no children
- where do you define child management methods?
 - In root: transparent, but leafs may do silly things
 - In composites: safer because leafs do not respond to method, but loss of uniformity

22

Composite Pattern (cont'd)

- should components implement list of components?
 - e.g., as an instance variable?
 - probably yes, except that leafs would never use this variable
- child ordering
 - sometimes an issue (e.g., a row of Lexi text)
- caching to improve performance
 - save I/O data to avoid recomputation
- who should delete components?
 - probably the responsibility of the composite

23

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

24

Compositor and Composition

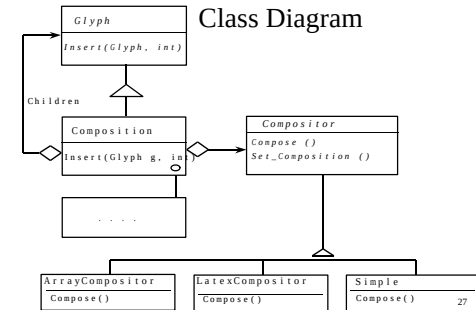
- Relevant design decisions:
 - compositor: class containing formatting algorithm
 - pass objects to be formatted as parameters to compositor methods
 - parameters are instances of a Glyph subclass called Composition
 - uniform interface between formattable objects and compositor algorithms

25

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

26



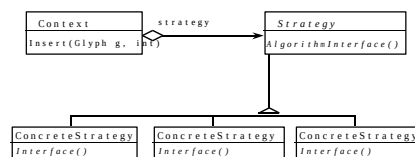
27

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

28

Strategy Pattern: Structure



29

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

30

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

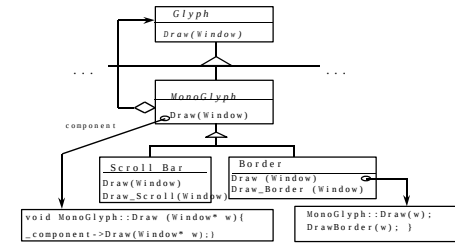
31

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")

32

Monoglyph class



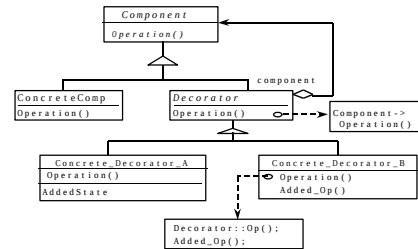
33

Decorator Pattern

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

34

Decorator Pattern: Structure



35

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

36

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(); // good
```

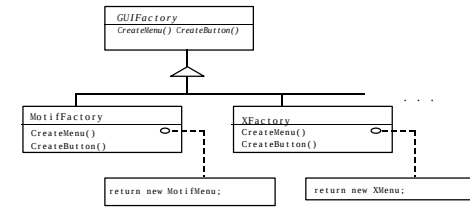
37

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, XFactory, etc.
 - must still specialize each widget into subclasses for each look-and-feel standard

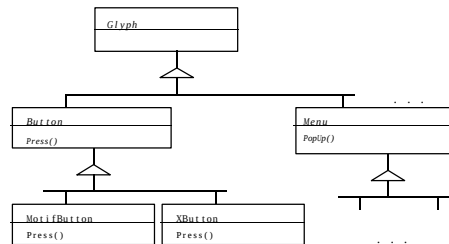
38

Class diagram for GUIFactory



39

Diagram for product classes



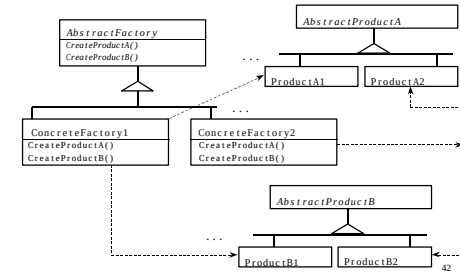
40

Abstract Factory pattern

- Name
 - Abstract Factory or Kit
- Applicability
 - different families of components (products)
 - must be used in mutually exclusive fashion and consistently
 - hide existence of multiple families from clients
- Structure
 - combine two previous diagrams and add arcs signifying object creation from factory classes to product classes

41

Structure of Abstract Factory



42

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
 - interface is fixed

43

Multiple Window Systems

- Want portability to different window systems
 - may seem 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 with specific types of windows (e.g., dialog, application, icon, etc.)
 - define WindowImp hierarchy to handle window implementation by a vendor

44

Multiple Window Systems (cont'd)

- Problem: how does one define the protocol of Window?
 - the intersection of functionality too restrictive
 - the union too large and incoherent
- Solution:
 - use common sense
 - identify generic behaviors
 - define “logical” and “physical” class hierarchies

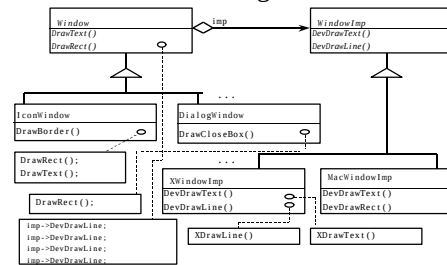
45

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

46

Structure of Bridge Pattern



47

Bridge Pattern

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

48

Bridge Pattern

- Implementation notes
 - only one implementor
 - don't need abstract class
 - choosing the implementor
 - abstract class
 - default and change at run-time
 - create a factory class

49

Supporting User Commands

- Support execution of Lexi commands
- Complications
 - different commands have different interfaces
 - same command can be invoked in different ways
 - Undo and Redo for some, but not all, commands (print)
- One solution - tie operations to UI widgets
 - no support for undo and redo
 - hard to associate state with the function
 - hard to extend functions

50

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

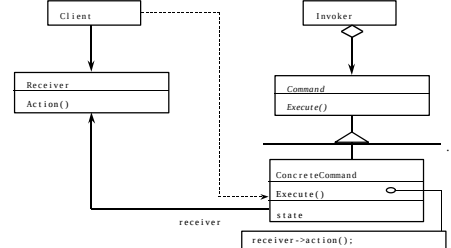
51

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
 - transactions in a database

52

Structure of Command Pattern



53

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

54

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

55

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 traversal actions from traversal

56

Iterator Pattern

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

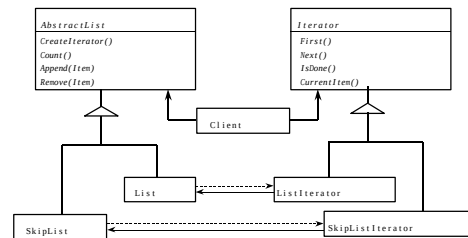
57

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

58

Structure of Iterator Pattern



59

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

60

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

61

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

62

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

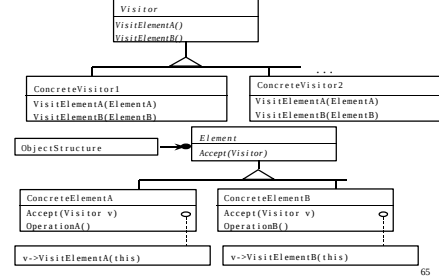
63

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

64

Structure of Visitor Pattern



65

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 hierarchies
 - *Iterator* needs a common superclass
 - accumulating state

66

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

67

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

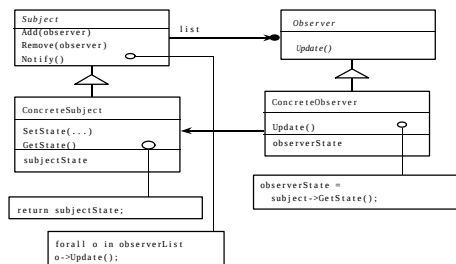
68

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

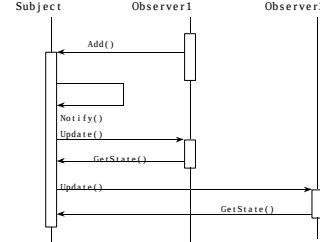
69

Structure of Observer Pattern



70

Run-time behavior of Observer



71

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

72

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

73

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

74

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

75

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

76

Conclusions

References

1. Go4 book: Gamma, Helm, Johnson, Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
 2. Go5 book: Buschmann, Meunier, Rohnert, Sommerlad, Stal. *Pattern-Oriented Software Architecture*, Wiley, 1996.
- See also:
3. *Pattern Languages of Program Design*. Conference Proceedings, Addison-Wesley, yearly.
 4. Pree. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

77

Conclusions

References (cont'd)

5. CACM, 10/96
6. Conference proceedings (e.g., OOPSLA, ICSE).
7. Tons of stuff on the web. Start from official pattern URL:
<http://st-www.cs.uiuc.edu/users/patterns/patterns.html>

78