Lexi: 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

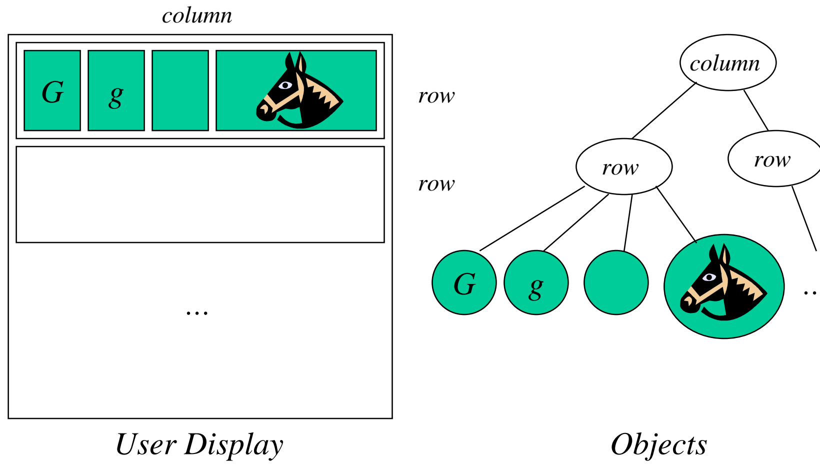
Structure of a Lexi Document

- 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

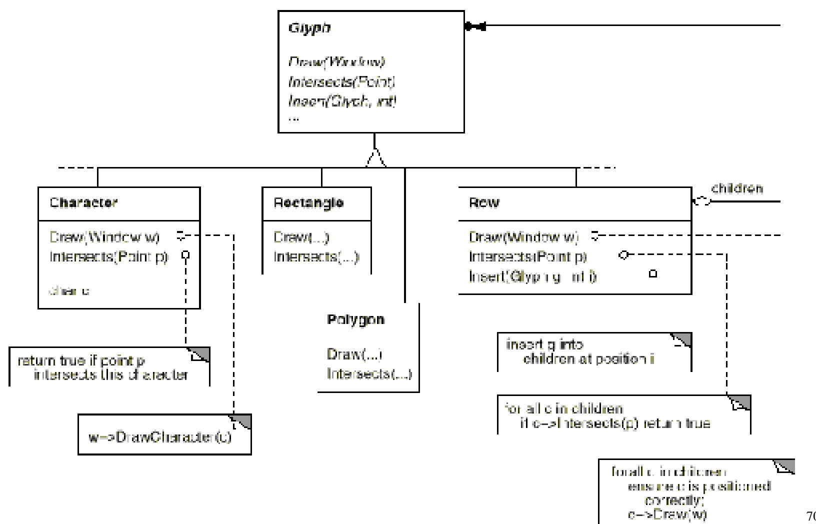
Structure of a Lexi Document

- Use **recursive composition** for defining and handling complex objects
  - Abstract class Glyph for all displayed objects
  - Glyph responsibilities:
    - know how to draw itself
    - knows what space it occupies
    - knows its children and parent
  - Glyph instances can recursively **compose** other Glyph instances

Recursive Composition



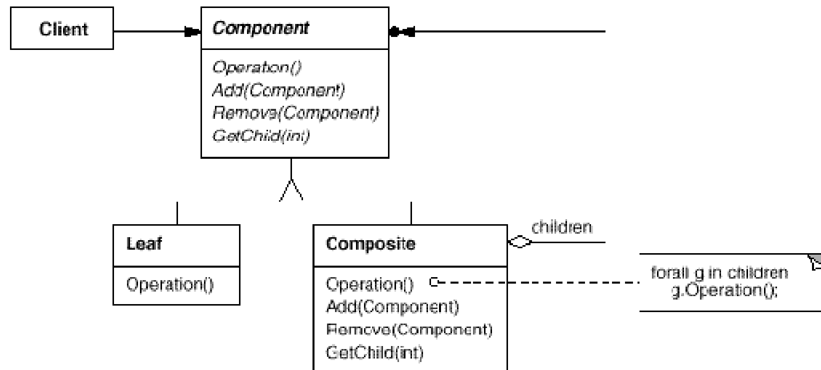
Glyph Class Diagram



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 …)

Composite Pattern Structure



Composite Pattern 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

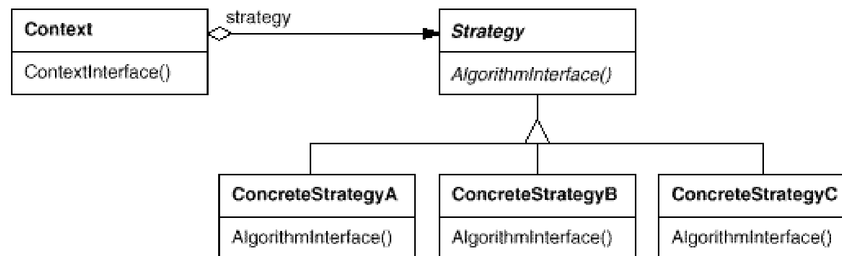
Formatting Lexi Documents: Strategy

- We know that documents are represented as Glyphs, but not how documents are constructed.
- Formatting:
  - Document structure will be determined based on rules for 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
  - define root class that supports many algorithms
  - each algorithm implemented in a subclass

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: 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
  - Like State, Template, …
  - Typical in OO programming

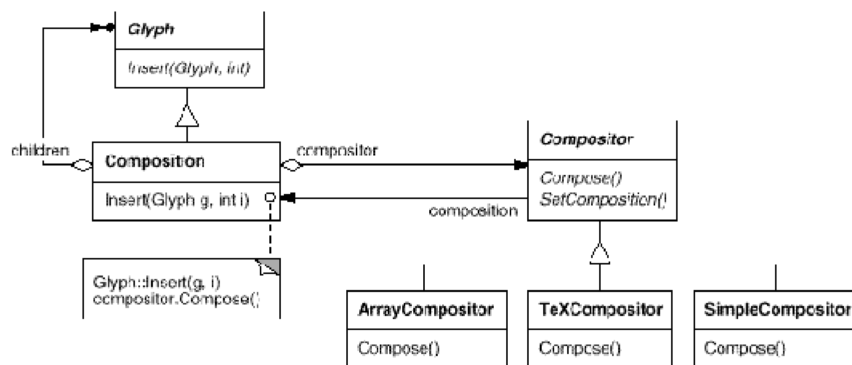
Strategy Pattern Consequences (cont'd)

- 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

Lexi: Using Strategy

- Compositor and Composition classes
  - Compositor: class encapsulating formatting algorithm
    - pass Composition objects to be formatted as parameters to Compositor methods
  - Composition: things being formatted
    - Glyph subclass
    - Each Composition object refers to its Compositor object
    - When a Composition needs to format itself, it sends a message to its Compositor instance

Class Diagram



Adding Scroll Bars and Borders: Decorator

- How to define classes for scrollbars and borders?
- Define as subclasses of Glyph
  - Scrollbars and borders are displayable objects
  - Will use 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")

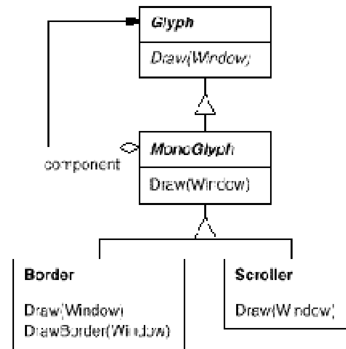
Transparent Enclosure

- Two features:
  - Single-child composition
    - Calls its child, then adds its own behavior
  - Compatible interfaces
    - Can use the enclosing object in place of the one it encloses
- Implemented by the Decorator pattern
  - Saw this earlier

Monoglyph class: a Decorator

```
Class Monoglyph { …
  void Draw (Window w)  {
      component.Draw(w);
  } …
}

Class Border extends Monoglyph { ..
  void Draw (Window w) {
     super.Draw(w);
     DrawBorder(w);
  } …
}
```



Changing look-and-feel: Abstract Factory

- Goal: easily change Lexi's look-and-feel
  - When new libraries are available (future variability)
  - At run-time by switching between them (present variability)
- Thoughtless implementation technique:
  - use distinct class for each widget and standard
  - let clients handle different instances for each standard
    - Button pb = new MotifButton(); // bad

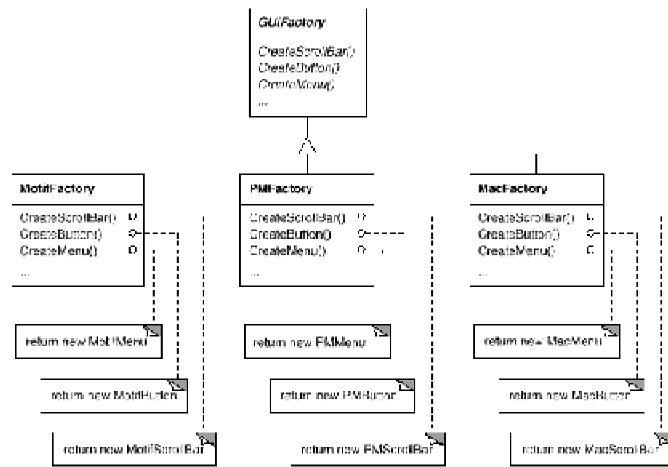
Abstracting Creation

- Concrete Creation problems:
  - Class of object is fixed at compile-time
    - can't change standard at run-time
  - Changing the class means making changes all over the code
- Instead:
  - Use a class to create abstract classes:
    - Button pb = guiFactory.createButton();// better

Solution: Use Abstract Factory

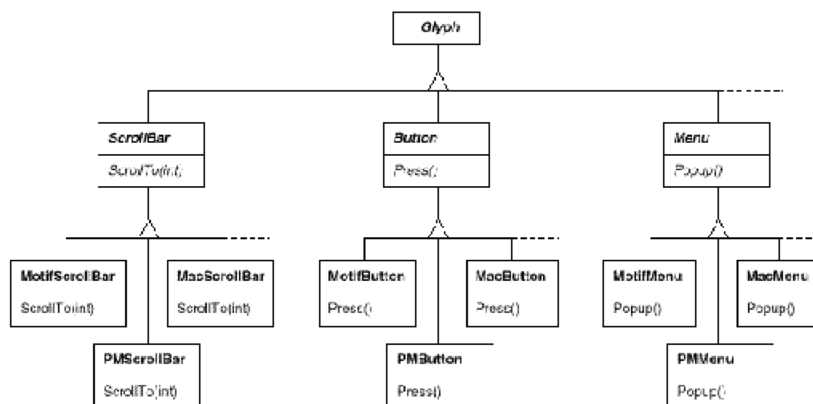
- Define abstract class GUIFactory with creation methods for widgets
  - Concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
    - MotifFactory, MacFactory, etc.
  - Specialize each widget into subclasses for each look-and-feel standard
- Thus, can easily change the kind of factory without changes all over the place

Class diagram for GUIFactory



CMSC 433, Fall 2002 87

Diagram for product classes

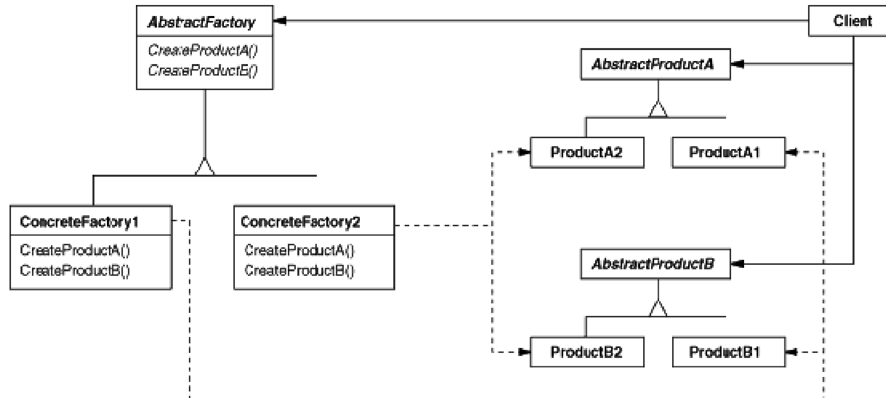


CMSC 433, Fall 2002 88

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: Consequences

- Isolate instance creation and handling from clients
- Can easily change look-and-feel standard
  - Reassign a global variable;
  - Recompute and redisplay the interface
- Enforce consistency among products in each family
- Adding to family of products is difficult
  - Have to update factory abstract class and all concrete classes

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.