DESIGN PATTERNS

CHAPTER 2: A Case Study: Designing a Document Editor

WYSIWYG: What-You-See-Is-What-You-Get

(Case study in the design of a) → Lexi: Document Editor

Design Problems

- Document Structure
- Formatting
- Embellishing the user interface
- Supporting multiple look-and-feel standars
- Supporting multiple window systems
- User operations
- Spelling checking and hyphenation

Document Structure

- Maintaining the document's physical structure, that is, the arrangement of text and graphics into lines, columns, tables, etc.
- Generating and Presenting the documen visually
- Making positions on the display to elements in the internal representation.

▲ Recursive Composition

- Entails building increasingly complex elements out of simpler ones.
- Gives us a way to compose a document out of simple graphical elements.
- This approach has two important implications.
  - The objects needs corresponding classes
  - The classes must have compatible interfaces, because we want to treat the objects uniformly.

▲ Glyph a

- Abstract Class for all objects that can appear in a document structure

Glyphs have 3 basic responsibilities.

- How to draw themselves
- What space they occupy
- Their children and parent

▲ Composite Pattern

Capture the essence of recursive composition in object-oriented terms.

Formatting

The ability to capture the document's physical structure doesn't tell us how to arrive at a particular structure

- Encapsolation the Formatting Algorith
  Formatting algorithms tend to be complex, it's also desirable to keep them wellcontained or - better yet - completely independent of the document structure.

- Compositor and Composition
  Compositor : class for objects that can encapsulate a Formatting argorithm.

  Composition : gets an instance of a Compositor subclass (specialized)

  The Compositor - Composition class split ensures a strong separation between code that supports the document's physical structure and the code for different formatting algorithms.

- Strategy Pattern
  The key to applying the strategy pattern is designing interfaces.

Embellishing the User Interface

- Transparent Enclosure
  - Extending existing code.
    (Use inheritance)
    (Add Composition)

- Monoglyph
  We can apply the concept of Transparent Enclosure to all glyphs that emebellishing other glyphs.

- Decarator Pattern.
  Capture class and Objects relationships that suppor embellishment by transparent enclosure.

Supporting Multiple look-and-Feel Standars.

- Abstracting Object Creation
  - A set of abstract Glyph subclasses for each category of widget glyph.
  - A set of concrete subclasses for each abstract subclass that implement different look-and-feel standars.

- Factories and Product Classes
- Abstract Factory Pattern
  Captures how to create families of related product objects without instantiating classes directly.

Supporting Multiple Window Systems

A platform's window system creates the illusion of multiple overlapping windows on a bitmapped display.

- Encapsulating Implementation Dependencies
  - They provide operations for drawing basic geometric shapes
  - They can iconify and de-iconify themselves
  - They can resize themselves
  - They can (re)draw their contents on demand
  1. Intersection Functionality (Window class interface)
  2. Union of Functionality (Create interface that incorporates the capabilities of all existing systems)

- Bridge Pattern
  The intent behind Bridge is allow separate class hierarchies to work together even as they evolve independently

  WindowImp : Window system implementations

User Operations

- Creating a new Document
- Opening, Saving, and printing an existing document
- cutting selected text out of the document and pasing it back in.
- changing the fond and style of selected text
- Changing the formatting text
- Quitting the application
- On and on

- Encapsulating a Request
- Command Class and Subclasses
- Undoability
- Command Hystory
- Command Pattern

Spelling Checking and Hyphenation

- Accesing Scattered Information
- Encapsulating Access and Traversal
- Iterator Class and Subclasses
- Iterator Pattern
- Encapsulating the Analisis
- Visitor Class and Subclasses
- Visitor Pattern

1 Composite to Represent the document's physical structure
2 Strategy to allow different formatting algorithms
3 Decorator for embellishing the user interface
4 Abstract Factory for supporting multiple look-and-feel standards
5 Bridge to allow multiple windowing platforms
6 Command for undoable user operations
7 Iterator for accessing and traversing object structures
8 Visitor for allowing an open-ended number of analytical capabilities without complicating the document structure's implementation.

DESIGN PATTERNS
CHAPTER 3 : Design Pattern Calalog

CREATIONAL PATTERNS
Creational design patterns abstract the instantiation process. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to anothe object.

Creational patterns become iniportant as systems evolve to depend more on object composition than class inheritance.

Abstract Factory
Applicability
Use the abstract factory pattern when
- A system should be independent of how its products are created, composed, and represented
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Participants
- AbstractFactory
- ConcreteFactory
- AbstractProduct
- ConcreteProduct
- Client

Consequences
- It isolates concrete classes
- It makes exchanging product families easy
- It promotes consistency among products.
- Supporting new kinds of products is difficult.

Implementation
- Factories as singletons
- Creating the products
- Defining extensible factories

Builder
Applicability
Use the builder pattern when:
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.

Participants
- Builder
- ConcreteBuilder
- Director
- Product

Consequences
- It lets you vary a product's internal representation
- It isolates code for construction and representation
- It gives you finer control over the construction process

Implementation
- Assembly and construction interface
- Why no abstract class for products?
- Empty methods as default in Builder

Factory Method

Applicability
Use the Factory Method pattern when
- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of wich helper subclass is the delegate

Participants
- Product
- ConcreteProduct
- Creator
- ConcreteCreator

Consequences
- Provides hooks for subclass
- Connects parallel class hierarchies

Implementation
- Two major varieties
- Parameterized factory Methods
- Language-specific variants and issues
- Using templates to avoid subclassing
- Naming Conventions.

Prototype

Applicability
Use de prototype pattern when a system should be independent of how its products are created, composed, and represented; and:
- when the classes to instantiate are specified at run-time
- To avoid building a class hierarchy of factories that parallels the class hierarchy of products
- when instances of a class can have one of only a few different combinations of state.

Participants
- Prototype
- ConcretePrototype
- Client

DD MM AA

Consequences
- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reduced subclassing
- Configuring an application with access dynamically

Implementation
- Using a prototype manager
- Implementing the Clone operation
- Initializing clones

Singleton

Applicability
Use the Singleton Pattern when
- There must be exactly one instance of a class, and it must be accesible to clients from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Participants
- Singleton

Consequences
- Controlled access to sole instance.
- Reduced name space.
- Permits a variable number of instances.
- Permits refinement of operations and representation.
- More Flexible than class operations.

Implementation
- Ensuring a unique instance
- Subclassing the Singleton class