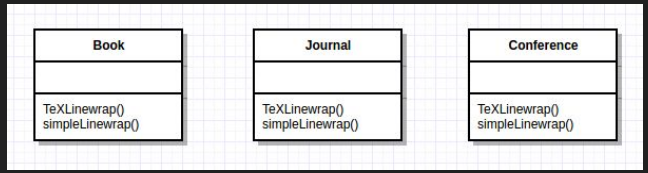
Strategy Pattern

Behavioral, Object focused

Motivation

- Many possible algorithms exist to solve every problem
- Hard-wiring each algorithm into each instance becomes intractable
  - For example, line wrapping in different documents can use the same algorithms

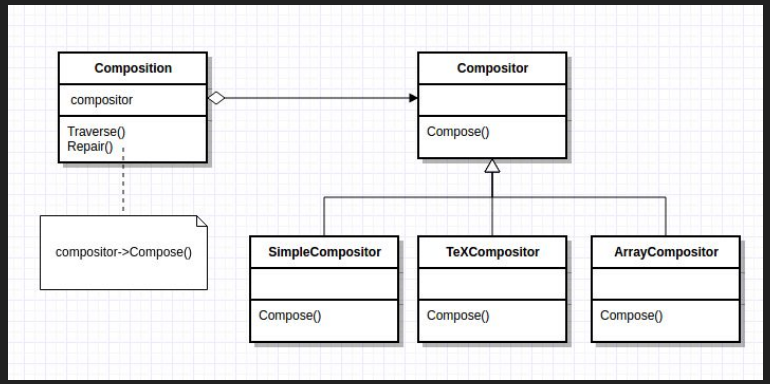


- Different algorithms may be appropriate in different situations
- Difficult to integrate new algorithms if code is hard-wired

Motivation

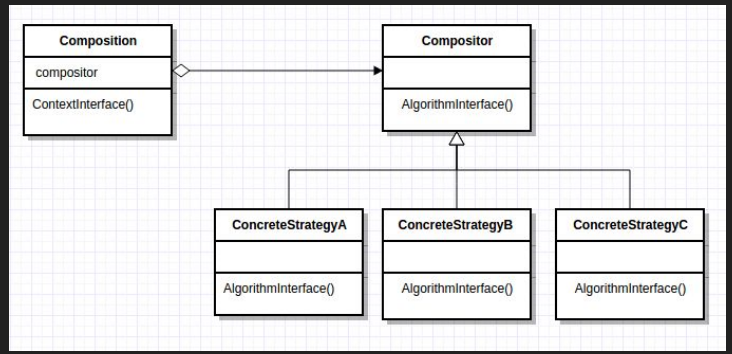
Define classes that encapsulate different algorithms (e.g. linebreaking)

An algorithm encapsulated this way is called a **strategy**



Structure

- **Strategy (compositor)**
  - Declares common interface for all supported algorithms
- **Concrete Strategy**
  - Implements the algorithms using the strategy interface
- **Context (composition)**
  - Configured with a concrete strategy object
  - Maintains reference to strategy object
  - May define interface to allow strategy to access its data



Consequences

- Pros:
  - Families of related algorithms
    - Inheritance factors out common functionality of algorithms
  - Alternative to subclassing
    - Allows varying the algorithm independently from the context
  - Strategies eliminate conditional statements
  - Offers different implementation of the *same* behavior
- Cons:
  - Incurs communication overhead between algorithm and context
  - Increases the number of objects

Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Formatting

Definition: Breaking a collection of glyphs into lines, lines into columns, and columns onto pages.

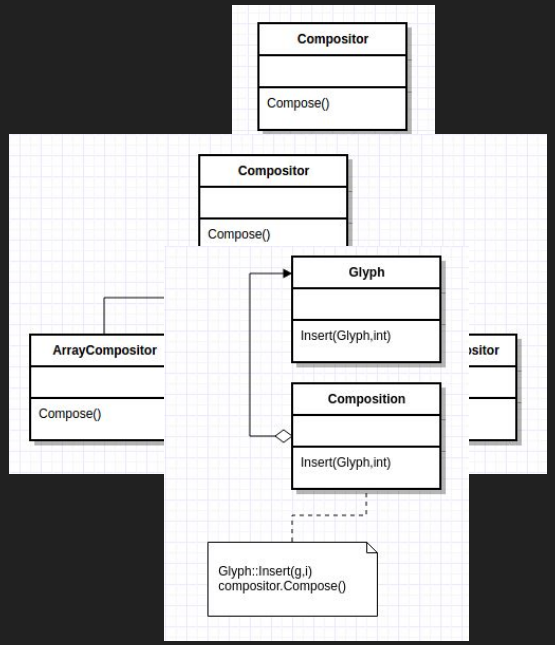
- Many, many complex methods (algorithms) exist for formatting
- We want to keep them self contained and independent of the document structure
  - Adding a new Glyph should not affect the code (*black-box reuse*)

This is done by ISOLATING and ENCAPSULATING the algorithm in an object

Formatting

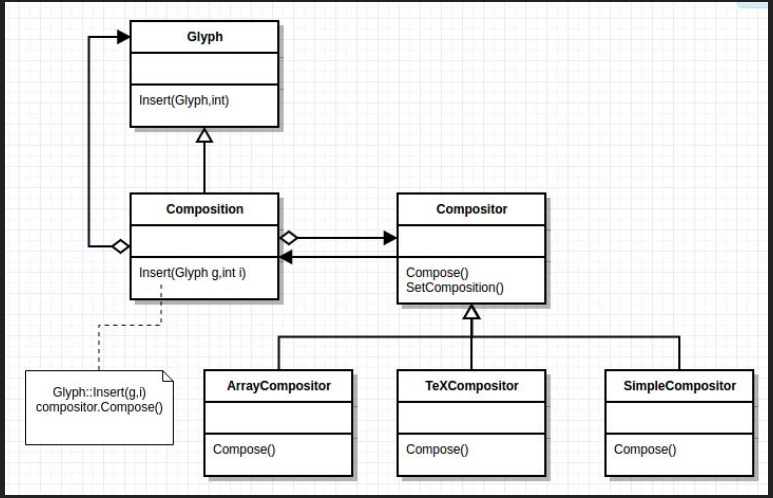
Let’s define a separate class hierarchy for formatting algorithms

- The root (compositor) will define the interface
- Subclasses implement a specific algorithm
- A Glyph subclass can structure its children for the given algorithm object



Compositor and Composition

- The `Compositor` class encapsulates the formatting algorithm
- The interface lets the compositor know ***what*** glyphs to format and ***when*** to do the formatting
- The glyphs formatted are children of the special `Composition` glyph subclass



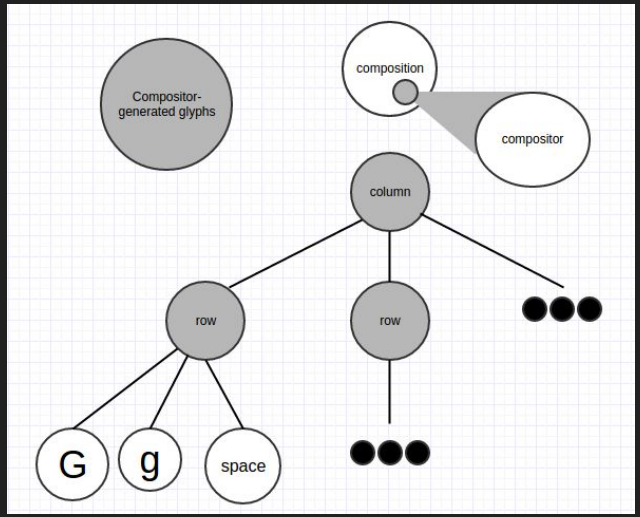
Basic Compositor Interface	
Responsibility	**Operations**
What to format	`void SetComposition(Composition*)`
How to format	`virtual void Compose()`

How to use the Composition and Compositor

1. A **`Composition`** gets an instance of a **`Compositor`** subclass
   a. The compositor subclass implements a specific formatting algorithm
2. An *unformatted* **`Composition`** object contains only visible glyphs making up the document's content
3. Glyphs determining the physical structure (Row and Column) are not contained
4. To format, the **`Composition`** calls **`compositor.Compose()`**
5. The **`Compositor`** iterates through the **`Compositions`** children and inserts new **`Row`** and **`Column`** glyphs according to the **`Compositor`** algorithm

How to use the Composition and Compositor

- The `Compositor` iterates through the `Compositions` children and inserts new `Row` and `Column` glyphs according to the `Compositor` algorithm



Benefits

- The `Compositor-Composition` class split ensures a separation between code for the physical structure, and formatting algorithm code
- Adding a new `Compositor` subclass does not affect the glyph classes
- Adding a new `Glyph` subclass does not affect the `Compositor` classes
- Using dynamic binding, the line-breaking algorithm can be changed at run-time through the `SetCompositor()` operation

Strategy Pattern Review

- Encapsulating an algorithm in an object is the intent of the **Strategy Pattern**
- The key participants in the pattern are **Strategy** objects (compositor) and the **context** in which they operate (composition)
- The key to applying the Strategy pattern is designing interfaces for the strategy and its context that are general enough to support a range of algorithms
- Follow the Three Strikes and Refactor rule