ECE450 – Software Engineering II

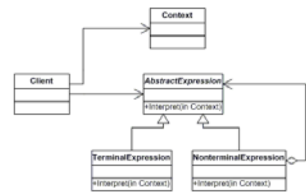
Today: **Design Patterns IX**
Interpreter, Mediator,
Template Method recap

The *Interpreter* pattern

- Need to know it exists, but it is probably the pattern with the lowest general applicability of all
  – Though it's very helpful if you deal with language creation, allowing users to come up with domain formulas, small compilers, etc.
- *Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language*
- Example: Regular expressions
  – The Interpreter pattern uses a class to represent each grammar rule…
    - I.e., what constitutes an "expression", a "literal", etc.
  – …and a Composite structure to construct the language

Structure and participants

- AbstractExpression
  – Declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree
- TerminalExpression
  – Implements an Interpret operation associated with terminal symbols in the grammar
  – An instance is required for every terminal symbol in a sentence
- NonterminalExpression
  – One such class is required for every rule in the grammar
  – Implements an Interpret operation for nonterminal symbols, typically calling itself recursively on the variables of the rule
- Context
  – Contains information that is global to the interpreter
- Client
  – Builds, or is given, an abstract syntax tree representing a sentence in the language that the grammar defines
  – Invokes the Interpret operation



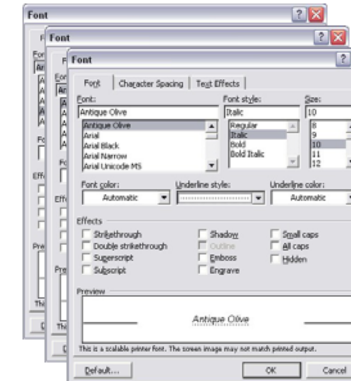
Applicability

- Use Interpret when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. It works best when:
  – The grammar is simple (otherwise the class hierarchy may become unmanageable)
  – Efficiency is not a critical concern

Consequences

- It's easy to change and extend the grammar
  - ...since it uses classes to represent grammar rules
- Complex grammars are hard to maintain
  - The pattern defines at least one class for every rule in the grammar. Grammars containing many rules can be hard to manage and maintain.
- Adding new ways to interpret expressions
  - It's easier to evaluate an expression in a new way (e.g., pretty printing)
  - If you keep creating new ways of interpreting an expression, consider using Visitor to avoid changing the grammar classes

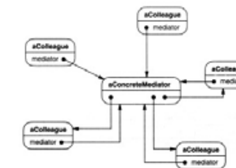
Coordinating GUI objects



How can we coordinate the interactions of all those objects?

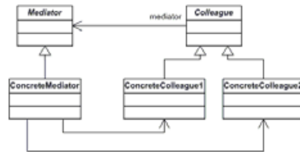
- Idea #1: Each object notifies of its changes directly to those that would be interested
  - For example, the Font list lets the Outline checkbox know whether it should be disabled
  - Too tight coupling!
- Idea #2: Implement Observer
  - Every object has a list of Observers and notifies them of changes
    - Objects don't need to know the details of who is observing them...
    - ...nor how the change will impact others
  - Much better approach...
  - ...but a bit clumsy: Every object is both an observer and a subject!
- Idea #3: The *Mediator* pattern
  - *Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently*

The mediator sits in the middle of all interactions



Structure and Participants

- Mediator
  - Defines an interface for communicating with Colleague objects
- ConcreteMediator
  - Implements cooperative behavior by coordinating Colleague objects
- Colleague classes
  - Each Colleague class knows its Mediator object
  - Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague



Applicability

- Use the Mediator pattern when…
  - A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand
  - Reusing an object is difficult because it refers to and communicates with many other objects
  - A behavior that's distributed between several classes should be customizable without a lot of subclassing

Consequences

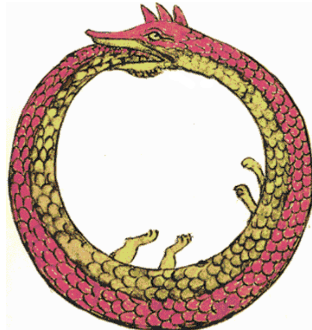
- It limits subclassing
  - Mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleagues can be reused as is
- It decouples colleagues
- It simplifies object protocols
  - Replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues
- It abstracts how objects cooperate
  - Since all mediation happens through a single object, it helps clarify how objects interact
- It centralizes control
  - Trades complexity of interaction for complexity in the mediator
  - May become more complex than any individual colleague

Implementation

- Do we need the abstract Mediator class?
  - We do **not** need it when colleagues work with only one mediator
- How can we implement the Colleague-Mediator communication?
  - We can implement the Mediator as an Observer of all the Colleagues
    - Instead of having everyone be **both** an Observer and a Subject

Finishing the tour at the beginning…

- Remember our old pizza example?
  - When we started this topic, we discussed the *Template Method* pattern, but we never defined it nor explained it properly
  - It's the only pattern left to define from the original *Design Patterns* book
  - …so let's get through with it:
- ***Template Method**: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure*



Example recap

```
public abstract class Pizza {
  public final void cook() {
    placeOnCookingSurface();
    placeInCookingDevice();
    int cookTime = getCookTime();
    letItCook(cookTime);
    removeFromCookingDevice();
  }
  protected abstract void placeOnCookingSurface();
  protected abstract void placeInCookingDevice();
  protected abstract int  getCookTime();
  protected abstract void letItCook(int min);
  protected abstract void removeFromCookingDevice();
}
```


Participants and structure

- AbstractClass
  - Defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm
  - Implements a template method defining the skeleton of an algorithm
- ConcreteClass
  - Implements the primitive operations to carry out subclass-specific steps of the algorithm

Applicability

- Use the Template Method pattern…
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication
    - "Refactoring to generalize"
  - To control subclasses extensions
    - You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points

Consequences

- Fundamental technique for code reuse
  - Particularly important in class libraries, since they are the means for factoring out common behavior in library classes
- Lead to an inverted control structure sometimes referred to as "the Hollywood principle"
  - "Don't call us, we'll call you"
  - The parent class calls the operations of a subclass and not the other way around