CS 417
Design Patterns

Visitor, Flyweight, State

Dr. Chad Williams
Central Connecticut State University

Announcements

- **Final projects reminder**
  - UML diagrams
    - note while full UML diagram required it can become very difficult to read create all relationships on single diagram
    - It is very easy to create sub diagrams i.e. include just classes relevant to specific pattern
      - Please do this particularly for additional patterns added that were not in the homework assignments
  - Make sure you use packages where appropriate
  - All objects well behaved (equals, toString, hashCode)
  - Don’t forget your JUnit

Design pattern: Visitor

- **Category**: Behavioral design pattern
- **Intent**:
  - Represents an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates

Applicability

Use the Visitor pattern when:

- Object structure contains many different classes with different concrete interfaces and without a shared concrete parent class
- Many distinct operations need to be performed on different types of objects within the structure allowing you to avoid polluting the object structure multiple similar calls in all parts of the object structure – instead functionality put in single place in a concrete Visitor class
- Object structure rarely changes as changes in structure may be costly in Visitor structure.

Participants

- Visitor (NodeVisitor)
  - Declares a visit operation for each class of the ConcreteElement allowing subclasses to access all operations of concrete classes
- ConcreteVisitor
  - Implements each operation declared by Visitor (typically thus cost of changing object structure). Stores state typically collected across traversal.
- Element (Node)
  - Defines accept operation that takes Visitor as argument
- ConcreteElement
  - Implements accept operation
- ObjectStructure – composite, list – provides mechanism for traversing full structure

In class examples

- Vehicle
  - Count of manufactures
- House
  - Inventory
  - Insurance adjustment

- Sequence diagrams

Design pattern: Flyweight

- **Category**: Structural design pattern
- **Intent**:
  - Use sharing to support large numbers of fine-grained objects efficiently
- **Motivation**
  - Huge number of similar (or identical) objects created and referenced in multiple places
  - Huge memory overhead
  - Goal rather than multiple copies of similar objects reference common objects

Applicability

Should **only** be used when ***all*** of the following are true:

- Application uses a large number of these objects
- Storage costs high due to sheer quantity of objects
- Most contextual object state can be made ***extrinsic***
- Many objects or groups of objects may be replaced by relatively few shared objects once extrinsic state removed
- Application doesn't depend on object identity

Participants

- Flyweight
  - Declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight
  - Implements Flyweight interface adds storage of ***intrinsic*** state
- UnsharedConcreteFlyweight
  - Possible for unshared object to implement Flyweight interface, most common with multiple ConcreteFlyweight children
- FlyweightFactory
  - Creates and manages flyweight objects
  - Ensures shared properly
  - Often implemented using Singleton pattern

In class examples

- Word document
- Massive multiplayer chess
- Casino decks of cards

Design pattern: State

- **Category**: Behavioral design pattern
- **Intent**:
  - Allow an object to alter its behavior when internal state changes. The object appears to change its class
- **Motivation**
  - Significant changes in behavior of same object depending on state
  - Reduce complexity of long conditional logic

Applicability

Use in either of these cases:

- Object's behavior depends on its state, and it must change its behavior at runtime depending on state
- Operations have large multipart conditional logic with several containing same conditional structure

Participants

- Context
  - Class defines the interface of interest to client
  - Maintains an instance of ConcreteState subclass that defines current state
- State
  - Defines interface for encapsulating the behavior associated with particular state of the Context
- ConcreteState subclasses
  - Each subclass implements a behavior associated with a state of the Context

In class examples

- TCP connection
  - Open
  - PassiveOpen
  - Closed
- Phone
  - Off
  - Locked
  - On
  - Camera