Flyweight Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the use of the Flyweight Design Pattern
- Use and implement the Flyweight Pattern

Inspiration

"… with proper design, the features come cheaply. This approach is arduous, but continues to succeed." –Dennis Ritchie

Bonus Pattern: Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Motivation
  - Imagine a text editor that creates one object per character in a document
  - For large documents, that is a lot of objects!
    - but for simple text documents, there are only 26 letters, 10 digits, and a handful of punctuation marks being referenced by all of the individual character objects

Flyweight, continued

- Applicability
  - Use flyweight when all of the following are true
    - An application uses a large number of objects
    - Storage costs are high because of the sheer quantity of objects
    - Most object state can be made extrinsic
    - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
    - The application does not depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

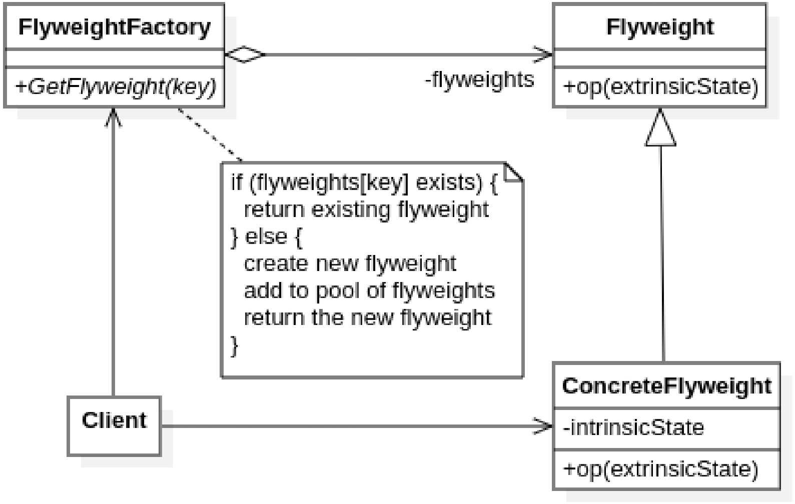
Idaho State University
Computer Science

Flyweight, continued

- Participants
  - `Flyweight`
    - declares an interface through which flyweights can receive and act on extrinsic state
  - `ConcreteFlyweight`
    - implements `Flyweight` interface and adds storage for intrinsic state
  - `UnsharedConcreteFlyweight`
    - not all flyweights need to be shared; unshared flyweights typically have children which are flyweights
  - `FlyweightFactory`
    - creates and manages flyweight objects
  - `Client`
    - maintains extrinsic state and stores references to flyweights

ROAR

Flyweight's Structure and Roles



```
if (flyweights[key] exists) {
  return existing flyweight
} else {
  create new flyweight
  add to pool of flyweights
  return the new flyweight
}
```

Flyweight, continued

- Collaborations
  - Data that a flyweight needs to process must be classified as intrinsic or extrinsic
    - Intrinsic is stored with flyweight; Extrinsic is stored with client
  - Clients should not instantiate `ConcreteFlyweights` directly
- Consequences
  - Storage savings is a trade-off between total reduction in number of objects versus the amount of intrinsic state per flyweight and whether or not extrinsic state is computed or stored
    - greatest savings occur when extrinsic state is computed

Flyweight, continued

- Demonstration
- Simple implementation of flyweight pattern
  - Focus is on factory and flyweight rather than on client
  - Demonstrates how to do simple sharing of characters

Wrapping Up

- The Flyweight Pattern is useful for managing situation where you need lots of “small” objects but you don’t want them taking up a lot of memory
  - It is an example of a “pattern of patterns” as it requires use of the Factory Pattern to control the creation of the “small” objects

Are there any questions?