Flyweight Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Design Aspect of Flyweight

Storage costs of objects

Outline

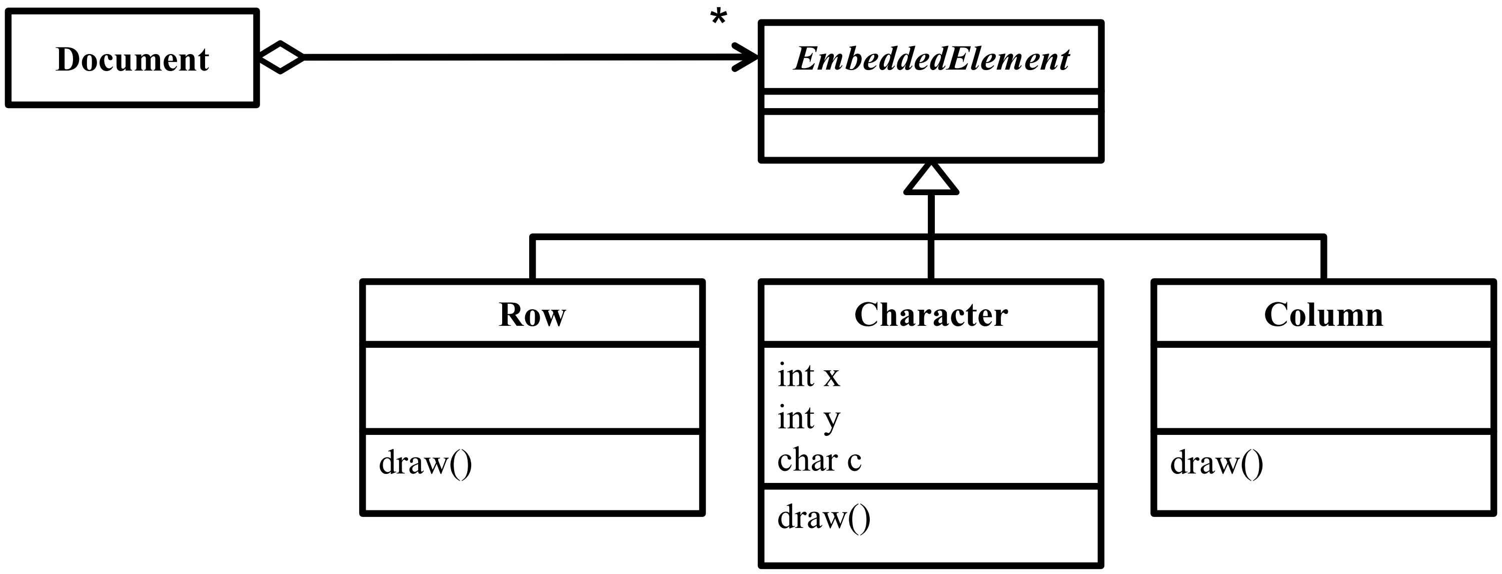
- Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Flyweight Pattern Structure

Document Editor (Flyweight)

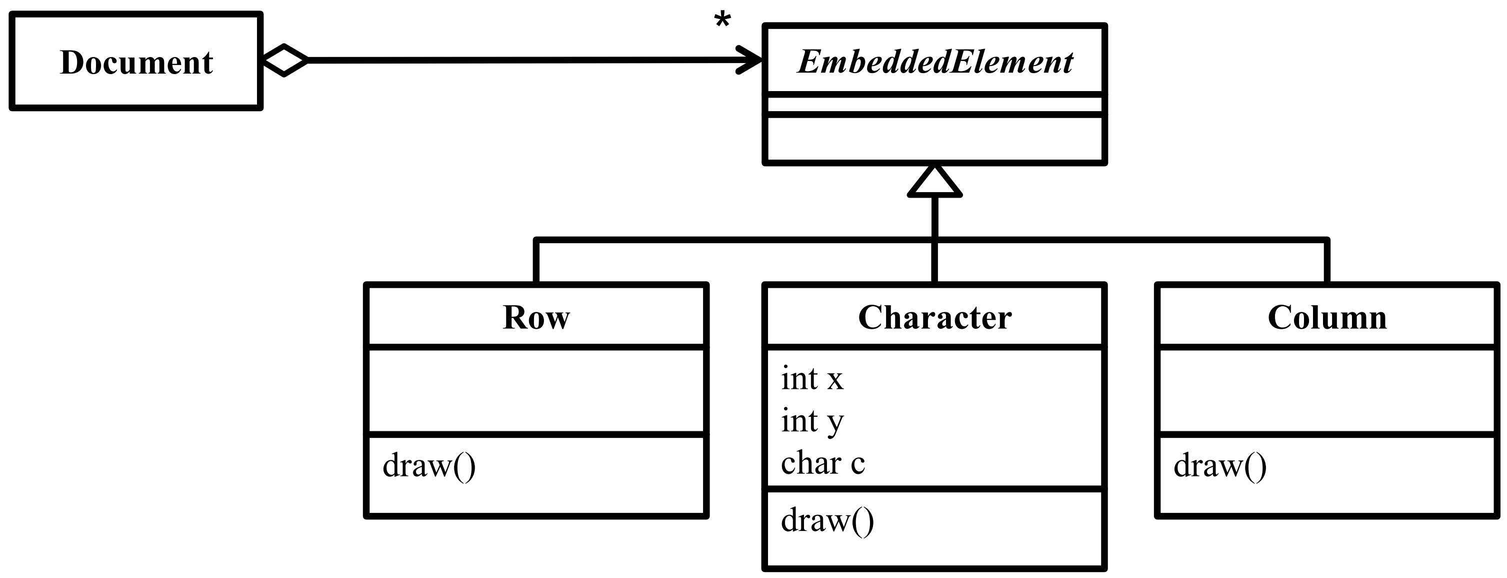
Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Requirements Statements$_1$

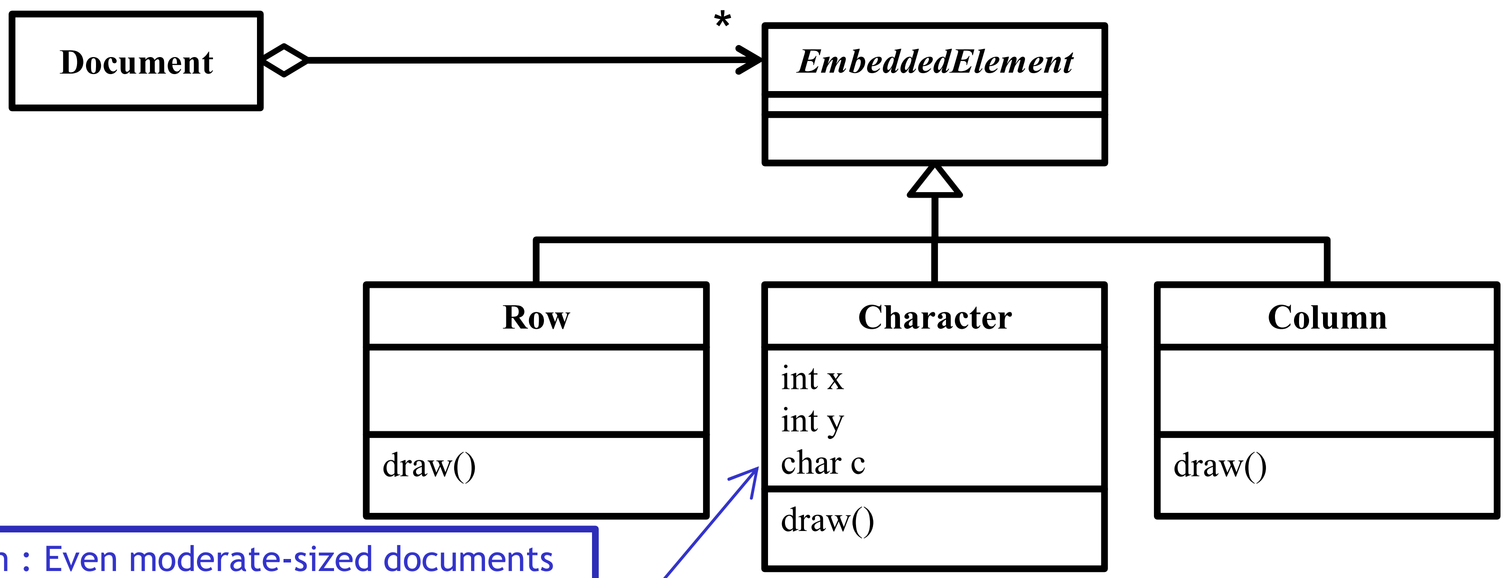
- A document editor uses objects to represent embedded elements like rows, columns and characters. The characters have X-Y locations, and they can draw themselves dynamically.



Initial Design



Problem with the Initial Design



Problem : Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.

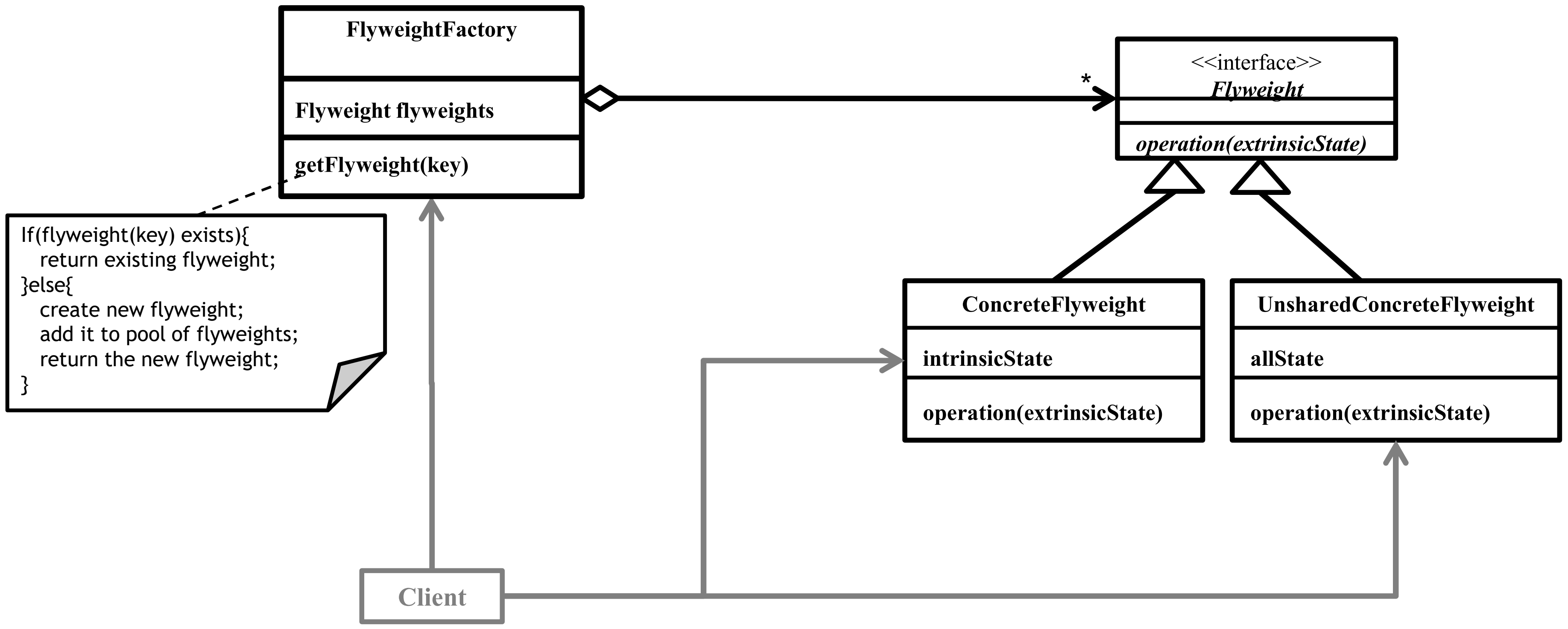
Recurrent Problem

- Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.

Intent

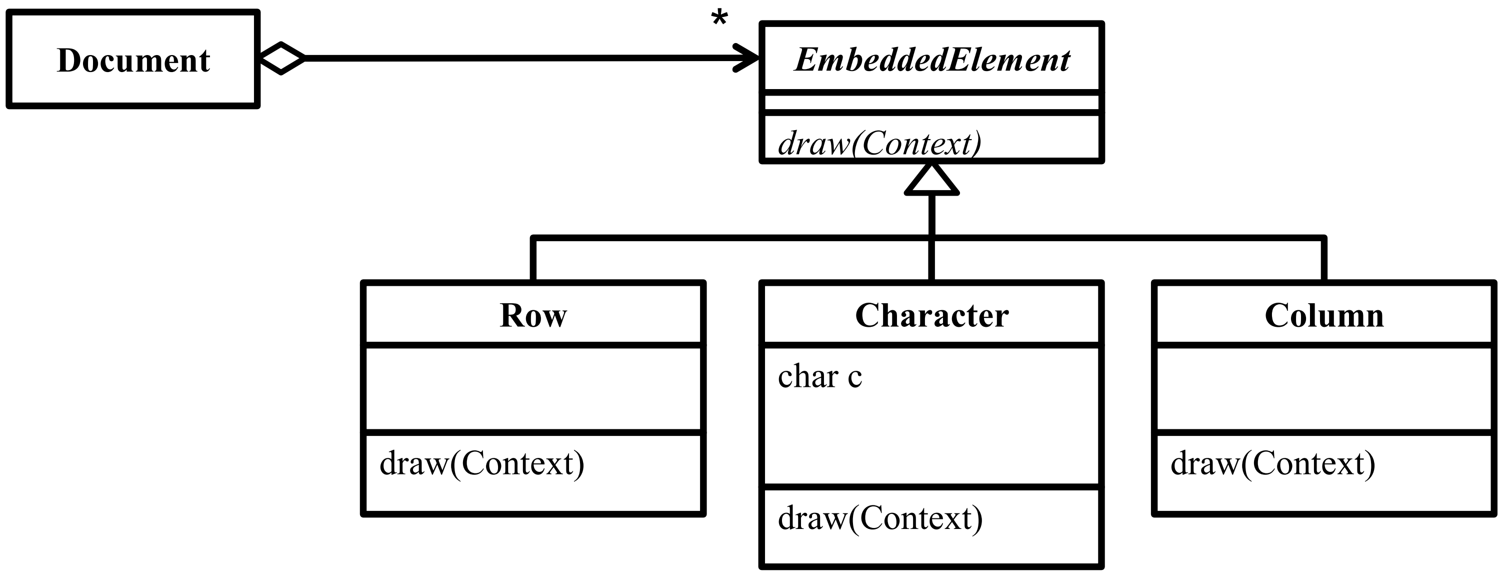
- Use sharing to support large numbers of fine-grained objects efficiently.

Flyweight Structure

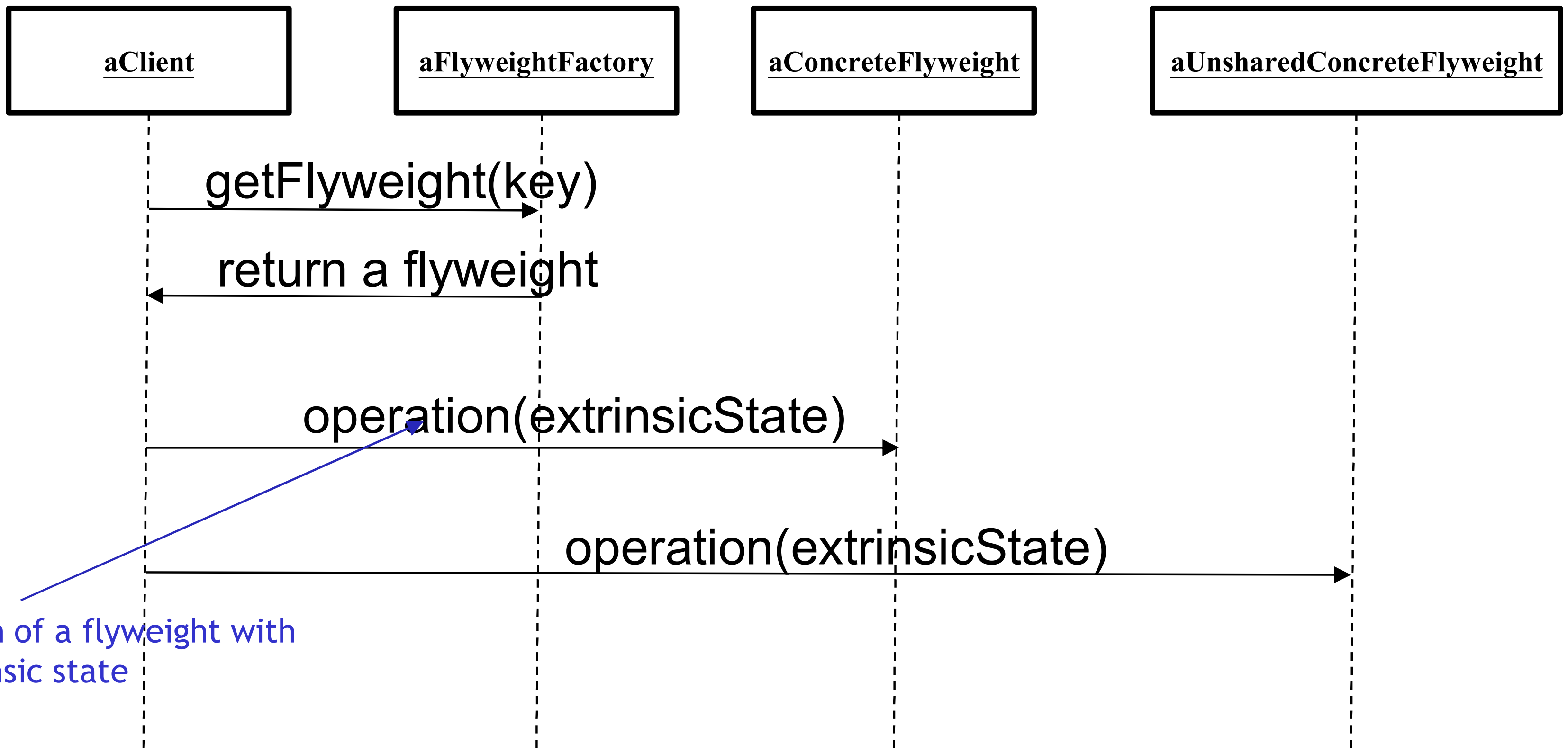


FlyweightFactory
Flyweight flyweights
getFlyweight(key)
If(flyweight(key) exists){
return existing flyweight;
}else{
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
*
<<interface>>
Flyweight
operation(extrinsicState)
ConcreteFlyweight
intrinsicState
operation(extrinsicState)
UnsharedConcreteFlyweight
allState
operation(extrinsicState)
Client

Example


Document
*
EmbeddedElement
draw(Context)
Row
draw(Context)
Character
char c
draw(Context)
Column
draw(Context)

Sequence Diagram



Invoke an operation of a flyweight with arguments of extrinsic state

Flyweight Pattern Structure$_3$

	Instantiation	Use	Termination
FlyweightFactory	Don't Care	The Client uses the operation of getFlyweight	Don't Care
Flyweight	X	The Flyweight Factory uses the interface of Flyweight for returning flyweight objects	X
ConcreteFlyweight	FlyweightFactory	The Client invokes an operation of a concrete flyweight with arguments of extrinsic state	FlyweightFactory
UnsharedConcreateFlyweight	FlyweightFactory	The Client invokes an operation of a concrete flyweight with arguments of extrinsic state	FlyweightFactory

Applicability

- Apply the Flyweight pattern 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 doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

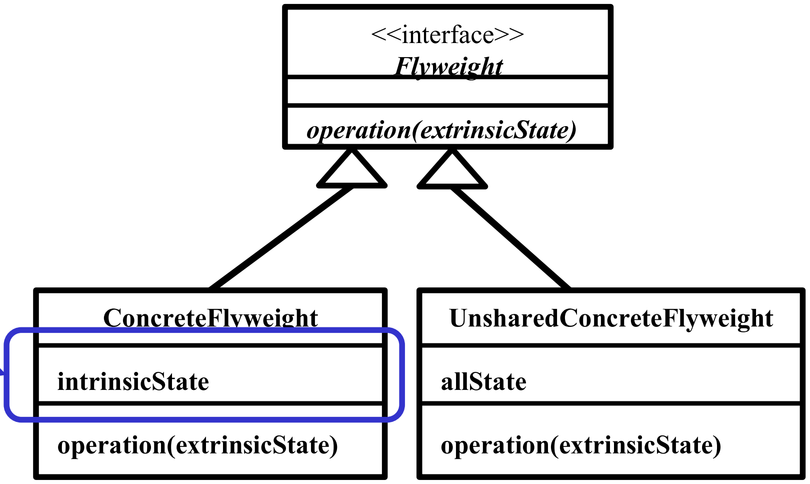
Consequences

- The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state. The greatest savings occur when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored. Then you save on storage in two ways:
  - Sharing reduces the cost of intrinsic state
  - You trade extrinsic state for computation time.

Flyweight objects are Immutable

- To enable safe sharing, Flyweight objects must be immutable.
  - An immutable object is an object whose state cannot be modified after it is created.

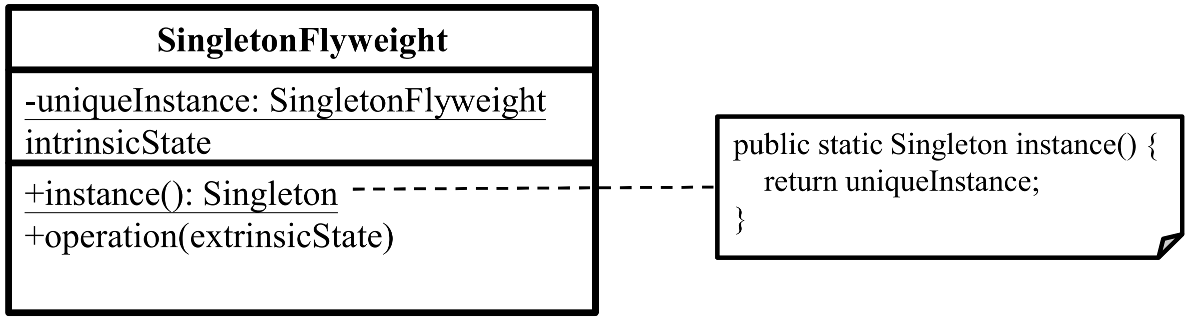
The intrinsicState object cannot be modified after it is created



<<interface>>
Flyweight
operation(extrinsicState)
ConcreteFlyweight
intrinsicState
operation(extrinsicState)
UnsharedConcreteFlyweight
allState
operation(extrinsicState)

Flyweight vs. Singleton

- Flyweight creates instances of a class with different values, and each instance can be used to provide many virtual instances.
  - It reduces the number of object instances at runtime, saving memory.
- Singleton creates only one instance of a class.
  - It provides a global point of access to that instance and is often used in multi-threaded environments to facilitate all threads using a single instance.
- Singleton can be used to create only one Flyweight instance for each class



SingletonFlyweight
-uniqueInstance: SingletonFlyweight
intrinsicState
+instance(): Singleton
+operation(extrinsicState)

```
public static Singleton instance() {
   return uniqueInstance;
}
```