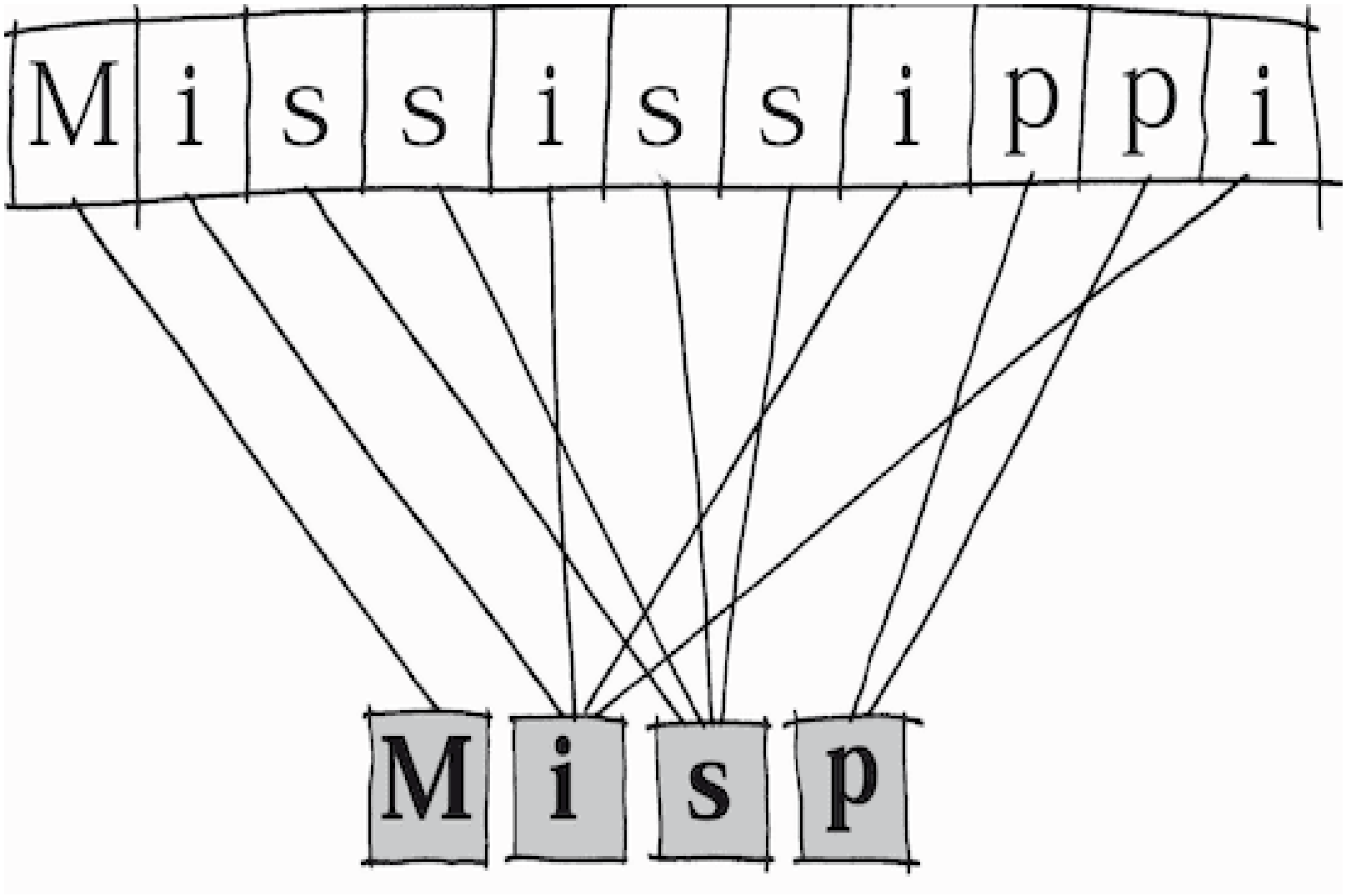
Flyweight Pattern

Use Sharing to Support Large Numbers of Fine-Grained Objects Efficiently


M i s s i s s i p p i
M i s p

Flyweight Pattern

Think of **sharing resources** efficiently when you need many similar objects:

- **Text editor**: Share font character **shapes**, vary **position** and **color**
- **Game forest**: Share tree **models**, vary **position** and **size**
- **Chess game**: Share piece **types** (6 types), vary **positions** (32 pieces)

- **Share** common intrinsic state among many objects.
- **Separate** intrinsic (shared) from **extrinsic** (context-specific) state.
- **Factory** manages shared instances to ensure efficient memory usage.

The Problem

- We need to display **large ASCII art characters** composed of many smaller characters.
- Each **big character** contains substantial **font data** (intrinsic state).
- Displaying **thousands of characters** would create massive memory overhead.

The challenge: how to support **large numbers** of character objects **efficiently**?

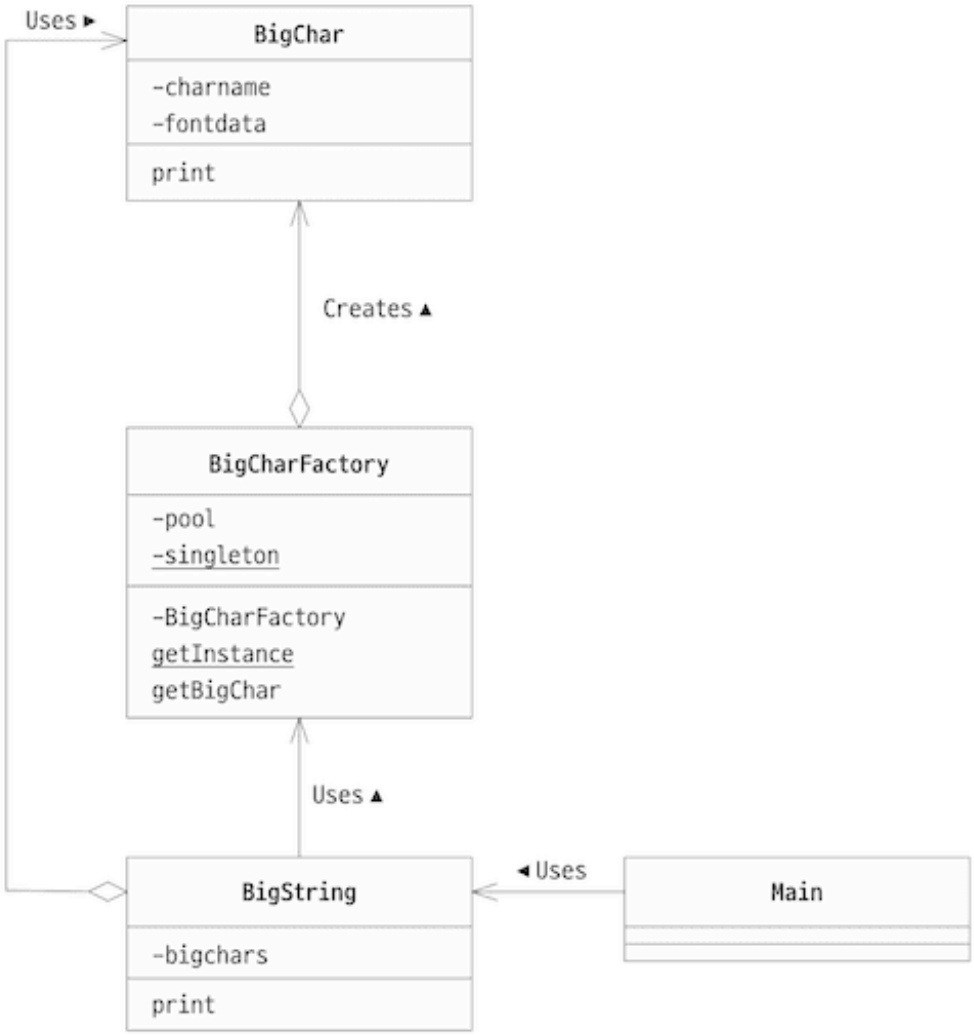
The *Flyweight* as the Solution

- **Share** the standard parts (font patterns) among all instances of the same character.
- **Separate** shared state (font data) from unique state (position, context).

We need a factory:

- **Factory** ensures only one instance per character type exists.

The Solution (Design)



Step 1: Understand the Players

In this design, we have key components:

The *Flyweight* (BigChar):

- Interface enabling flyweight to receive extrinsic state

The *FlyweightFactory* (BigCharFactory):

- Creates and manages flyweight instances

The *User/Context* (BigString):

- Uses flyweights and maintains extrinsic state

Step 2: Intrinsic vs Extrinsic state separation

- **Intrinsic**: Shared among flyweights (font patterns)
- **Extrinsic**: Unique per usage (position, context)

Step 3: Understand Intrinsic State

- **Intrinsic state** is **independent** of context and can be shared.
- In our example: **font patterns** for each character type ('A', 'B', etc.)
- **Stored inside** the flyweight object.
- **Immutable** once created.

Step 4: Understand Extrinsic State

- **Extrinsic state depends** on context and cannot be shared.
- In our example: **position** where character appears, **styling** information.
- **Passed as parameters** to flyweight methods.
- **Stored in context** objects, not flyweights.

Step 5: Understand the Factory

- **FlyweightFactory** creates and **manages** flyweight instances.
- **Ensures sharing**: Returns existing instance if already created.
- **Memory management**: Keeps a pool of created flyweights.
- **Thread safety**: Often needs synchronization for concurrent access.

Code

- Main Method
- Flyweight Factory
- Flyweight Implementation
- Context Usage

Main Method

```
from big_string import BigString
from big_char_factory import BigCharFactory

def main():
    # Create a string with repeated characters
    big_string = BigString("1212123")
    big_string.print()

    # Demonstrate memory efficiency
    factory = BigCharFactory.get_instance()
    print(f"\nMemory Efficiency:")
    print(f"Characters displayed: {len('1212123')}")
    print(f"BigChar instances created: {factory.get_pool_size()}")
    print(f"Memory savings: {len('1212123') - factory.get_pool_size()} reused")
```

Step 1: Create context with repeated characters

```
big_string = BigString("1212123")
```

- **BigString** acts as **context (user)** that uses multiple flyweights.
- **Repeated characters** ('1' and '2') will demonstrate sharing efficiency.

Step 2: Display the content

```
big_string.print()
```

- **Context** coordinates multiple **flyweights** to produce final output.
- **Extrinsic state** (positioning) managed by context.

Step 3: Analyze memory efficiency

```
print(f"Characters displayed: {len('1212123')}")      # 7 characters
print(f"BigChar instances created: {factory.get_pool_size()}")  # Only 3 instances
```

- **Flyweight sharing**: Only three instances ('1', '2', '3') for seven character displays.
- **Memory savings**: 4 character instances reused through sharing.

Flyweight Factory

It uses the Singleton DP.

```
class BigCharFactory:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
                    cls._instance._pool = {}
        return cls._instance

    def get_big_char(self, charname):
        with self._lock:
            if charname not in self._pool:
                self._pool[charname] = BigChar(charname)  # Create new
            return self._pool[charname]  # Return existing or new
```

Key Points: Factory

1. **Singleton pattern**: Only one factory instance exists
2. **Pool management**: Maintains dictionary of created flyweights
3. **Lazy creation**: Creates flyweight only when first requested
4. **Thread safety**: Uses locks for concurrent access protection

Flyweight Implementation

```
class BigChar:  # ConcreteFlyweight
    def __init__(self, charname):
        self.charname = charname  # Intrinsic state
        # Load font data from file - expensive operation done once
        try:
            with open(f"big{charname}.txt", 'r') as f:
                self.fontdata = f.read()  # Intrinsic state
        except IOError:
            self.fontdata = f"{charname}?\n"

    def print(self):  # Operation using intrinsic state
        print(self.fontdata, end='')
```

Key Points: Flyweight

1. **Intrinsic state only**: Stores sharable state (font patterns)
2. **Heavy initialization**: Expensive operations done once per type
3. **Stateless operations**: Methods don't modify internal state
4. **Context independence**: Behavior doesn't depend on external context

Context (User) Usage

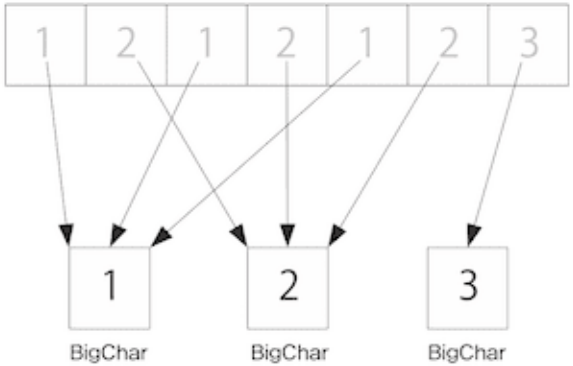
```
class BigString:  # Context
    def __init__(self, string):
        self.string = string  # Extrinsic state
        factory = BigCharFactory.get_instance()

        # Store references to flyweights - extrinsic state
        self.bigchars = []
        for char in string:
            big_char = factory.get_big_char(char)
            self.bigchars.append(big_char)

    def print(self):
        # Coordinate flyweights with extrinsic state (positioning)
        for line_num in range(max_lines):
            for big_char in self.bigchars:
                # Extrinsic state: position in combined output
                print_char_line(big_char, line_num)
```

Key Points: Context (User)

1. **Extrinsic state management**: Stores context-specific information
2. **Flyweight coordination**: Manages multiple flyweight instances
3. **Operation orchestration**: Combines flyweights to achieve complex behavior
4. **State separation**: Keeps what can't be shared separate from flyweights

Discussion

Memory Usage Analysis



Without Flyweight: 1000 'A' characters = 1000 BigChar instances

With Flyweight: 1000 'A' characters = 1 BigChar instance + 1000 references

Memory savings grow **linearly** with number of repeated objects.

Key Benefits

1. **Memory efficiency**: Dramatic reduction in memory usage for large object collections
2. **Performance**: Fewer object allocations and garbage collections
3. **Scalability**: Support virtually unlimited numbers of fine-grained objects
4. **Automatic sharing**: Factory handles sharing transparently

When to Use Flyweight

- **Application** uses **large numbers** of objects
- **Storage costs** are high due to **quantity** of objects
- Most **object state** can be made **extrinsic**
- Groups of objects can be **replaced** by **few shared** objects
- Application doesn't depend on **object identity**

Implementation Considerations

1. **State separation**: Carefully identify intrinsic vs extrinsic state
2. **Factory management**: Consider when to remove unused flyweights
3. **Thread safety**: Ensure the factory is thread-safe in concurrent environments
4. **Extrinsic state**: Design efficient ways to pass extrinsic state to operations

Related Patterns

- **Singleton**: Flyweight factory often implemented as a singleton
- **Factory Method**: Used by the flyweight factory to create instances
- **Composite**: Flyweights can serve as leaves in composite structures
- **State/Strategy**: Can be implemented as flyweights if they don't maintain state

Flyweight vs Singleton

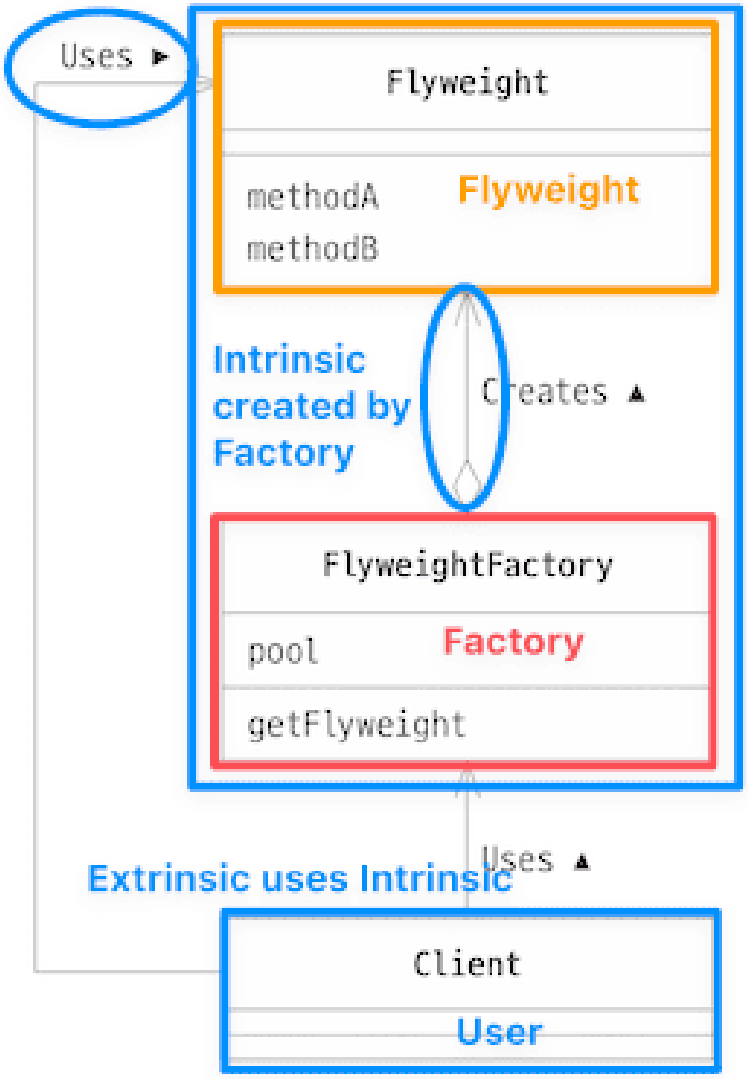
Singleton Pattern:

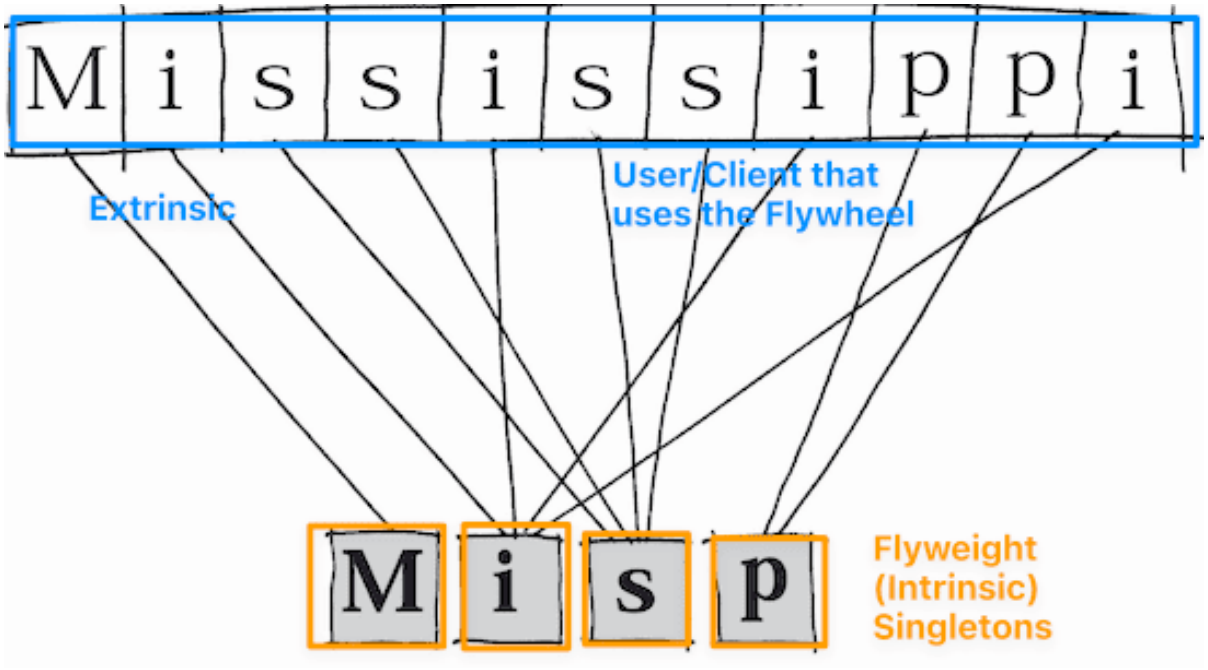
- **Ensures** single **instance** of a class
- **Class-level** uniqueness constraint

Flyweight Pattern:

- **Manages** multiple **shared instances** (one per intrinsic state)
- **Instance-level** sharing for efficiency

UML


Uses
Flyweight
methodA
methodB
Flyweight
Intrinsic
created by
Factory
Creates
FlyweightFactory
pool
Factory
getFlyweight
Uses
Extrinsic uses Intrinsic
Client
User


M i s s i s s i p p i
Extrinsic
User/Client that
uses the Flyweight
M i s p
Flyweight
(Intrinsic)
Singletons