CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 21 Flyweight & Facade

Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 185-206

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 179-212

Flyweight
Motivation

When a program needs a large number of objects, the space requirements can be come too large

The example in the text uses objects to represent individual characters of the alphabet

Using objects allows the program to treat characters like any other part of the document - as an object

A character, like “G”, may require a fair amount of information:

- The character type - G not h or y
- Its font
- Its width, height, ascenders, descenders, etc.
- How to display/print the character
- Where it is in the document

Most of this information is the same for all instances of the same letter

Intrinsic State

- Information that is independent from the objects context
- The information that can be shared among many objects

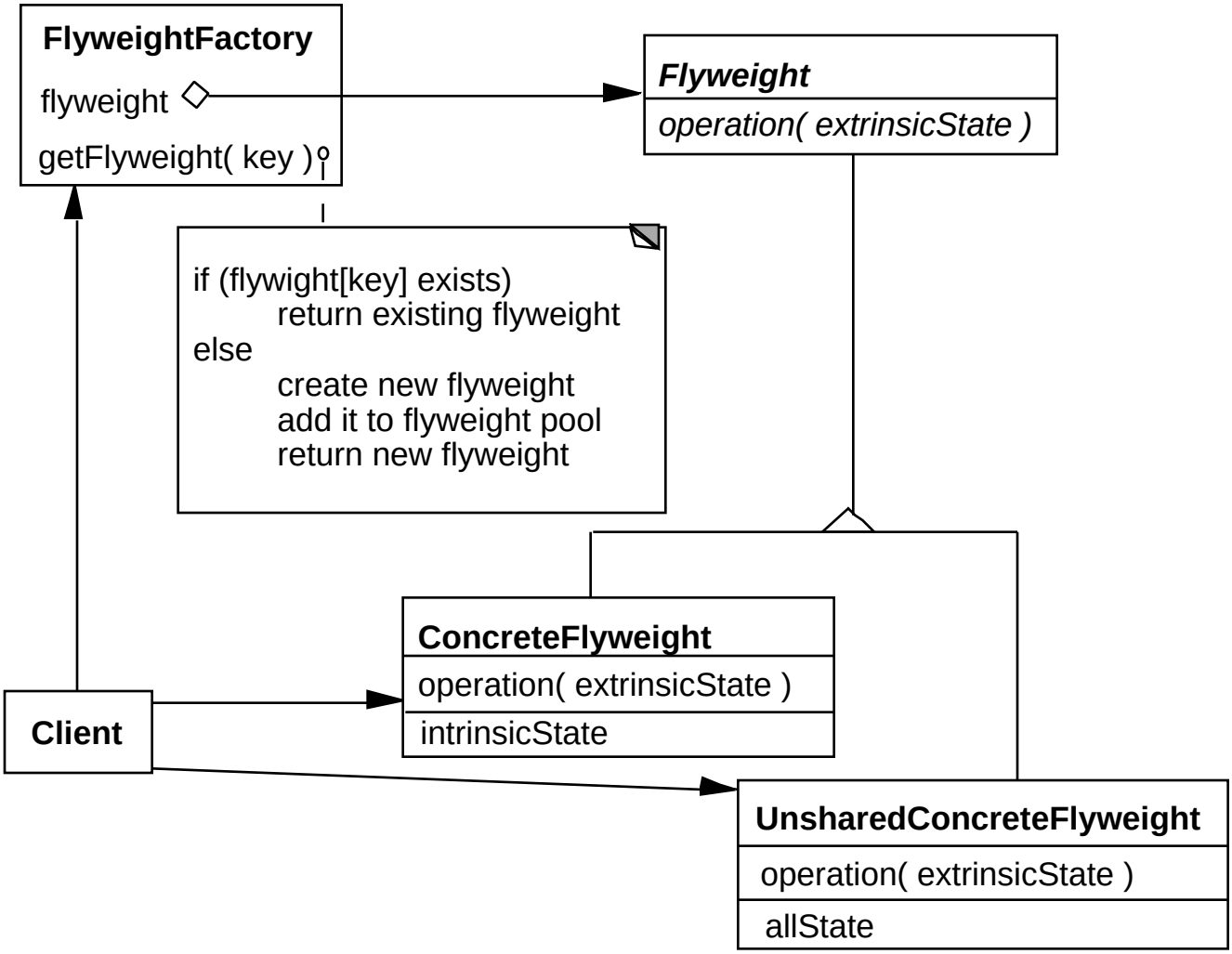
Extrinsic State

- Information that is dependent on the objects context
- The information that can not be shared among objects

So create one instance of the object and use the same object wherever you need an instance

The one object can store the intrinsic state, but someone else needs to store the extrinsic state for each context of the object

Structure


FlyweightFactory
flyweight
getFlyweight(key)
Flyweight
operation(extrinsicState)
if (flywight[key] exists)
return existing flyweight
else
create new flyweight
add it to flyweight pool
return new flyweight
ConcreteFlyweight
operation(extrinsicState)
intrinsicState
Client
UnsharedConcreteFlyweight
operation(extrinsicState)
allState

Applicability

The pattern can be used when all the following are true

- The program uses a large number of objects
- Storage cost are high due to the sheer quantity of objects
- The program does not use object identity

```
MyClass* objectPtrA;
MyClass* objectPtrB;

if ( objectPtrA == objectPtrB ) //testing object identity
```

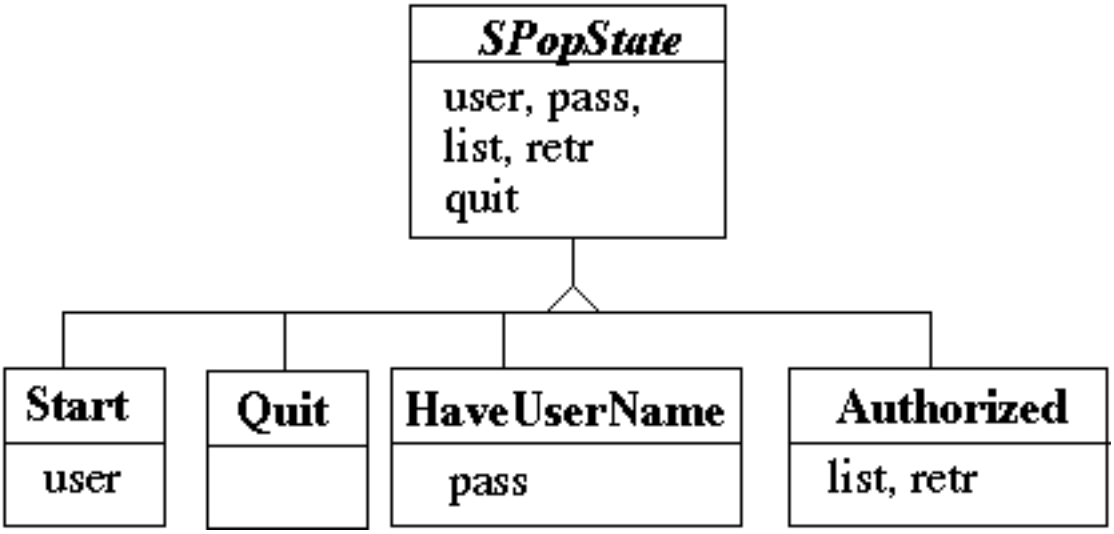
- Most object state can be made **extrinsic**

  Extrinsic state is data that is stored outside the object

  The extrinsic state is passed to the object as an argument in each method

- Many objects can be replaced by a relatively few shared objects, once the extrinsic state is removed

Example



State transitions

Start State comes first
When get user name transfer to HaveUserName State
When get correct password transfer to Authorized State

HaveUserName and Authorized states need the user name

In concurrent server we may have many (100's) different users connected at the same time.

State creation can become expensive, so create each state once

Each time use the state pass the user name to the state

```
class HaveUserName extends SPopState
   {
   private HaveUserName() {};
   private static HaveUserName instance;

   public getInstance( )
      {
      if ( instance == null )
         instance = new HaveUserName();
      return instance;
      }

   public SPopState pass( String userName, String password )
      {
      if ( validateUser( userName, password )
         return Authorized.getInstance();
      else
         return Start.getInstance();
      }
   }
```

Implementation

Separating state from the flyweight

This is the hard part

Must remove the extrinsic state from the object

Store the extrinsic state elsewhere

This needs to take up less space for the pattern to work

Each time you use the flyweight you must give it the proper extrinsic state

Managing Flyweights

Cannot use object identity on flyweights

Need factory to create flyweights, cannot create directly

How do we know when we are done with a flyweight?

Facade
Review of Wirfs-Brock (CRC Design Process)

- Exploratory Phase

  Who is on the team?

  What are their tasks, responsibilities?

  Who works with whom?

- Analysis Phase

  Who's related to whom?

  Finding sub teams

  Putting it all together

Exploratory Phase

- Who is on the team?

  What are the goals of the system?

  What must the system accomplish?

  What objects are required to model the system and accomplish the goals?

- What are their tasks, responsibilities?

  What does each object have to know in order to accomplish each goal it is involved with?

  What steps toward accomplishing each goal is it responsible for?

- Who works with whom?

  With whom will each object collaborate in order to accomplish each of its responsibilities?

  What is the nature of the objects' collaboration?

Analysis Phase

- Who's related to whom?

  Determine which classes are related via inheritance

  Finding abstract classes

  Determine class contracts

- Finding sub teams

  Divide responsibilities into subsystems

  Designing interfaces of subsystems and classes

- Putting it all together

  Construct protocols for each class

  Produce a design specification for each class and subsystem

  Write a design specification for each contract

Subsystems

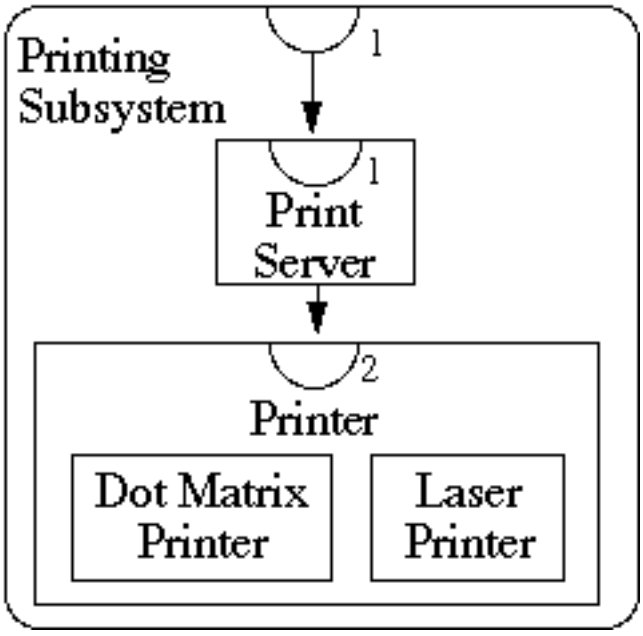
Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts

There is no conceptual difference between the responsibilities of a class and a subsystem of classes

The difference between a class and subsystem of classes is a matter of scale

A subsystem should be a good abstraction

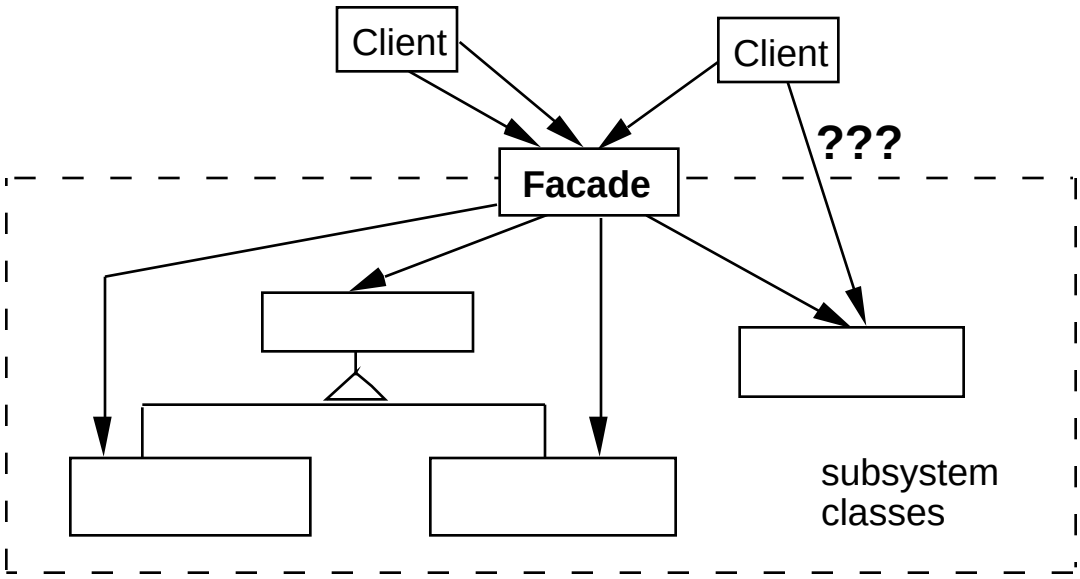
There should be as little communication between different subsystems as possible



The Facade Pattern - Basic Idea

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem