Flyweight Pattern

CSCI-4448 - Boese

University of Colorado **Boulder**

Objectives

- Problem
- Definition
- Why
- Examples
- How
- Comparisons

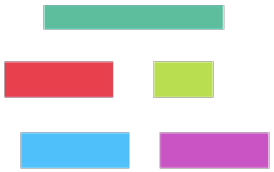
Problem

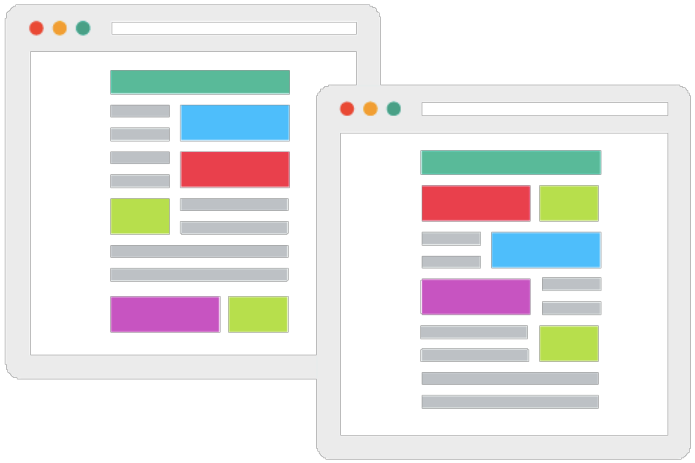
Example

Modern web browsers

- Prevent loading same images twice.
- Loads a web page,
  - New Images:: Loads and places into internal cache.
  - Already loaded images::
    - Create a flyweight object
      - Unique data like position within the page, but everything else is referenced to the cached one.

Browser loads images just once and then reuses them from pool:





Definition

Definition

"Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.."

Definition

Name “Flyweight”

- Flyweight is a boxing category, for light weight people.
- Flyweight pattern is for "light weight" objects (though you will have many of them)

Type

- Structural

Intent

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

Problem

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage

Why

Why use Flyweight Pattern?

- Many similar objects are used and the storage cost is high
- The majority of each object's state data can be made *extrinsic*
- A few shared objects would easily replace many unshared objects
- The identity of each object does not matter

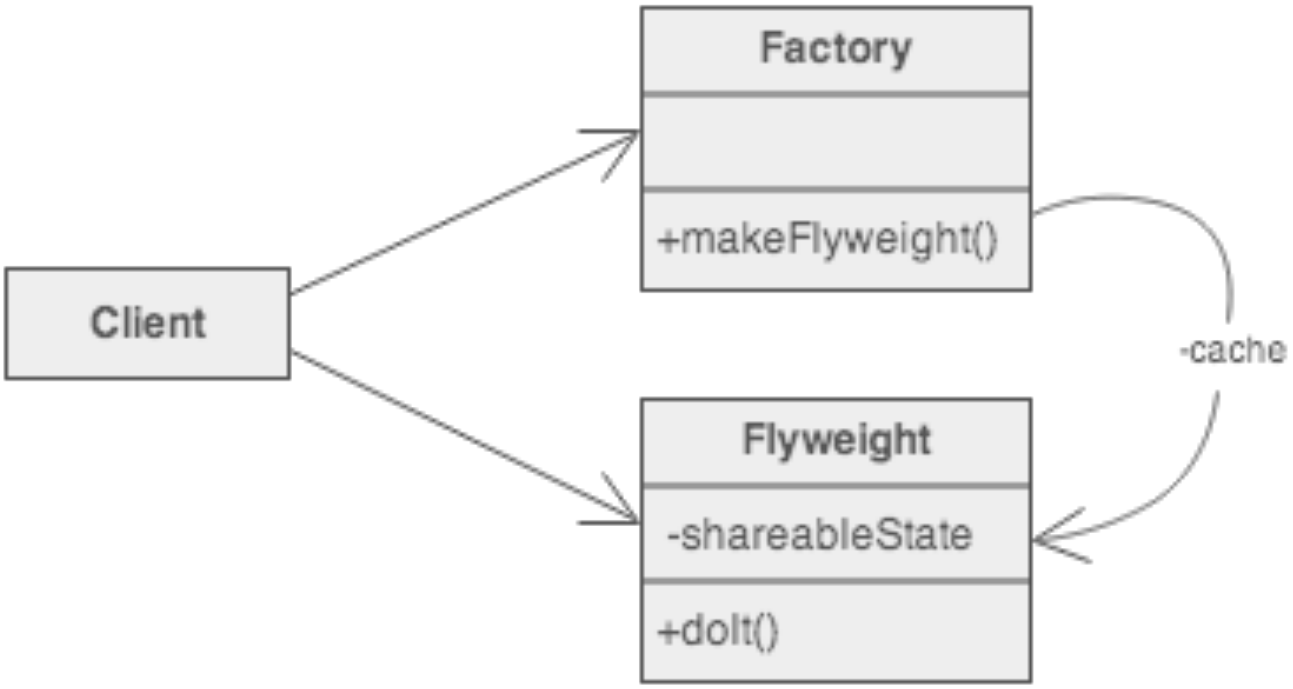
- Typical case
  - System resources.
    - Icons or folders are good candidates for use of this pattern.

Composition

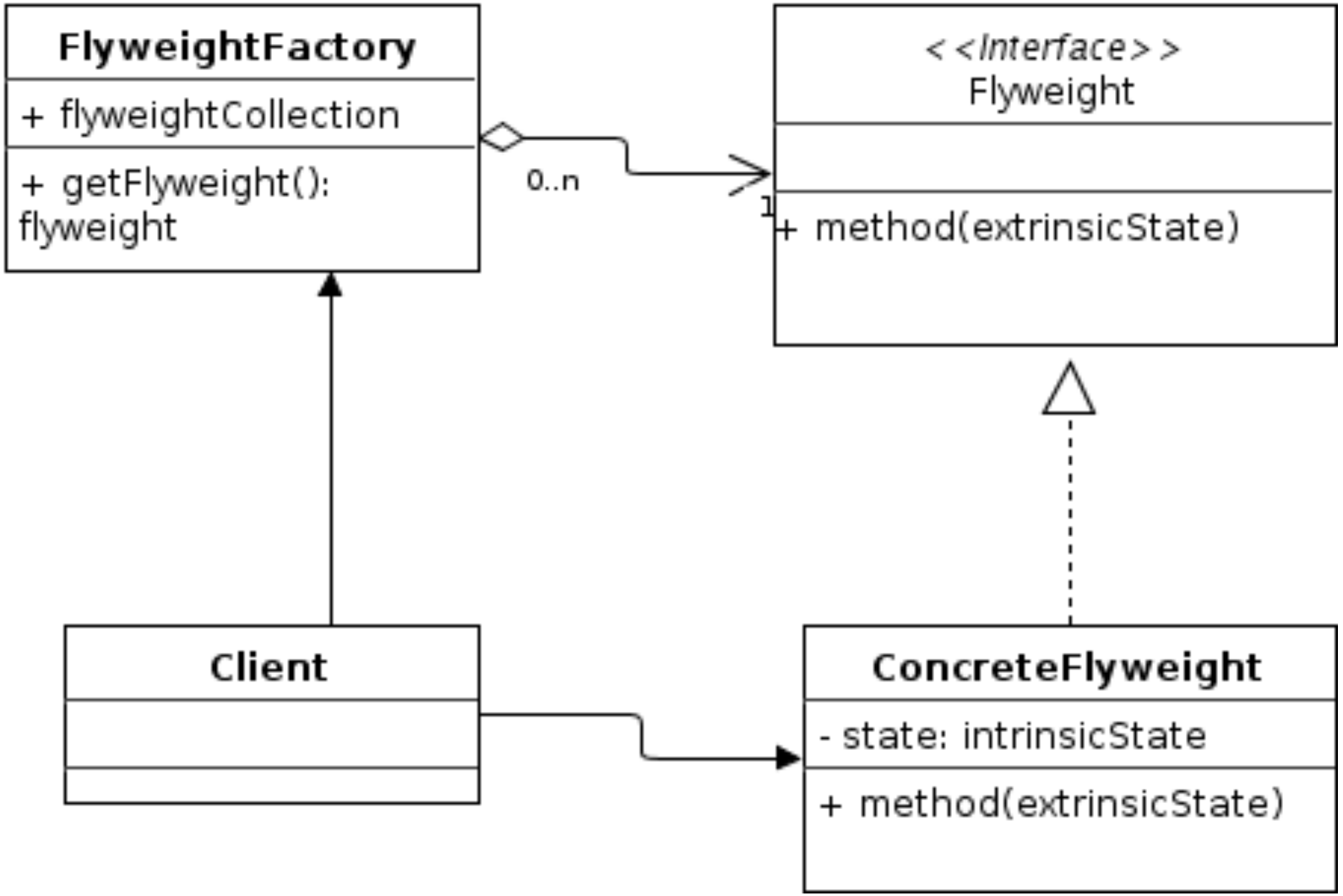
- Each "flyweight" object is divided into 2 pieces:
  - **State-dependent** (extrinsic) part,
    - Stored or computed by client objects, and passed to the Flyweight when its operations are invoked
  - **State-independent** (intrinsic) part.
    - Stored (shared) in the Flyweight object.

Flyweight UML

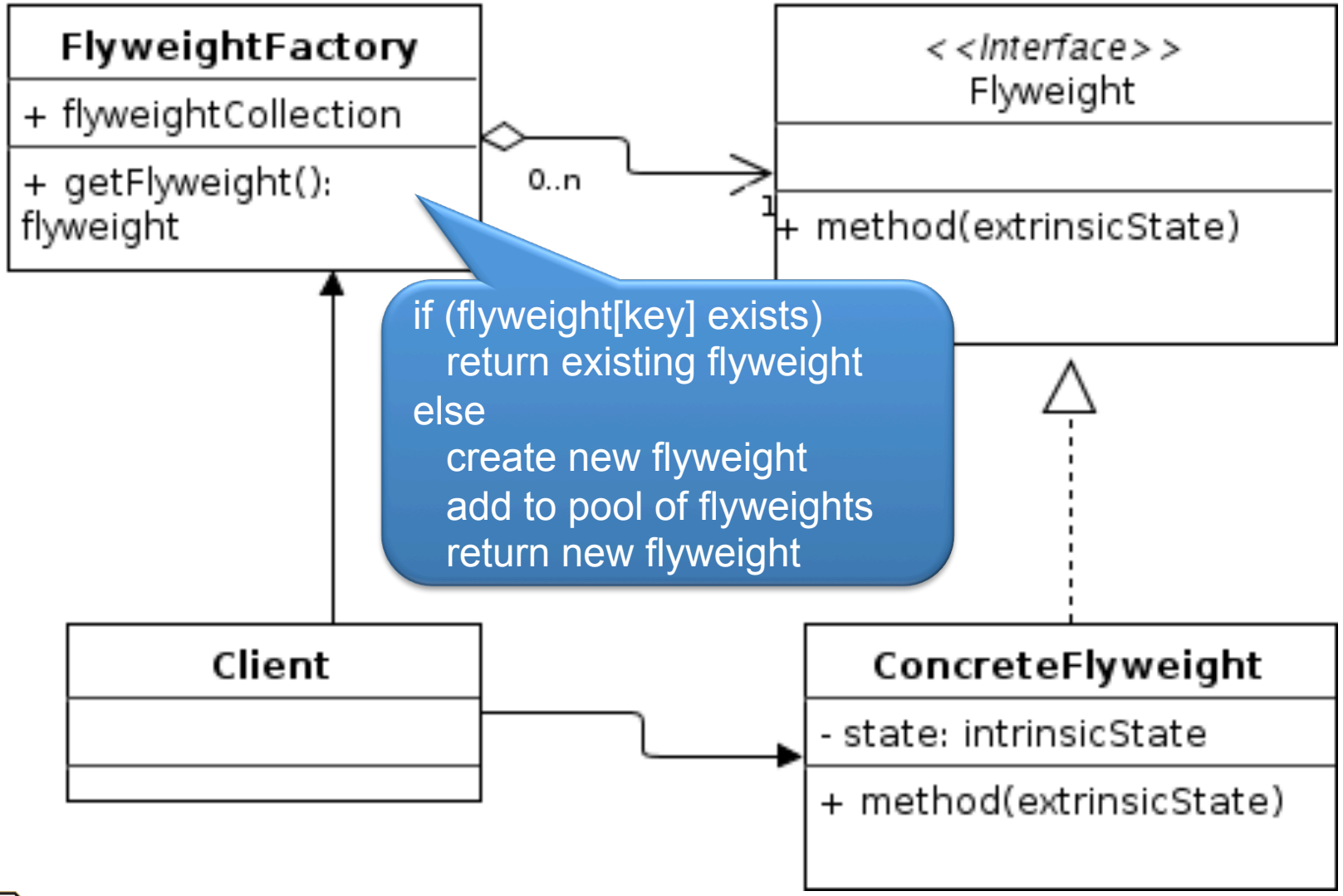
Structure



Flyweight UML



Flyweight UML



Generic FlyweightFactory

```
public class FlyweightFactory
{
    private Map<Integer, Flyweight> flyweights
                        = new HashMap<Integer, Flyweight>();

    public Flyweight get(Integer key)
    {
        Flyweight flyweight = flyweights.get(key);

        if (flyweight == null)
        {
            flyweight = new ConcreteFlyweight(key);
            flyweights.put(key, flyweight);
        }

        return flyweight;
    }
}
```

Example

Letters in a Document

Example

Text Editor

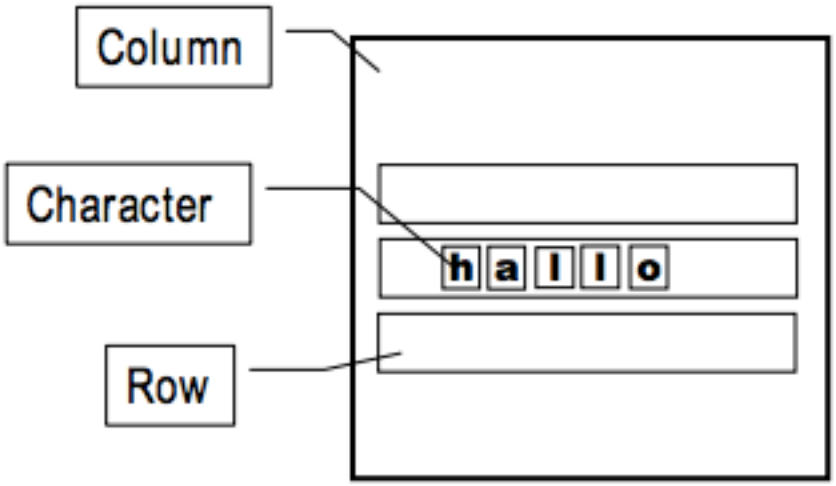
- Only alphabet set A to Z.
- If create 100 page document, we may have 200000 (2000 X 100) characters (assuming 2000 characters / page).
  - Without flyweight we will create 200000 objects to have fine grained control. With such fine control, every character can have its own characteristics like color, font, size, etc.

Example

Solution

- Intrinsic State
  - Create only 26 objects for (A to Z) mapping every unique character.
    These 26 objects will have intrinsic state as its character.
    That is object ‘a’ will have state as character ‘a’.
- Extrinsic State
  - Color, font and size
  - Passed by client code.
  - 26 objects will be in store, client code will get the needed character/ object and pass the extrinsic state to it with respect to the context. With respect to context means, ‘a’ in first line may come in red color and the same character may come in blue color in different line.

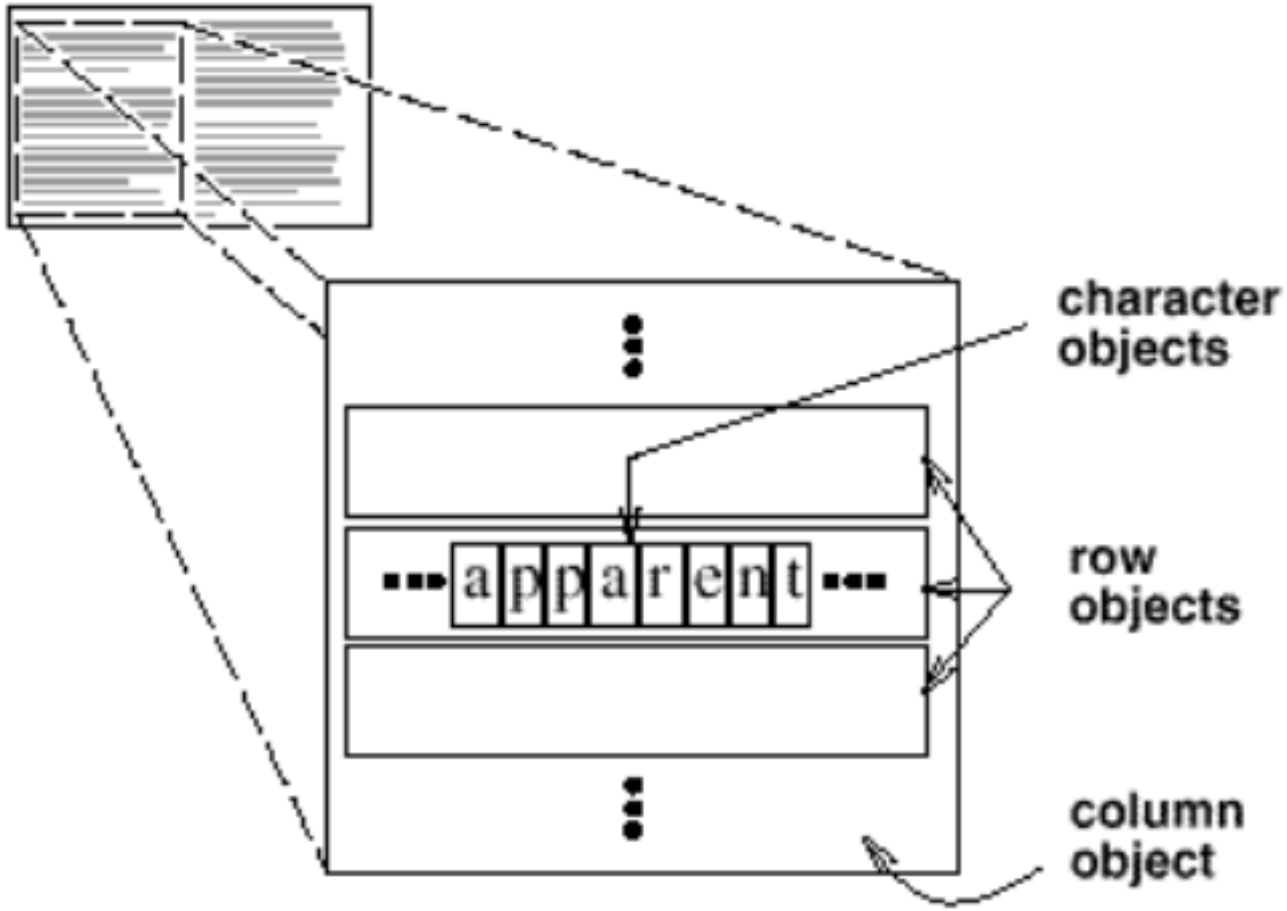
Text Problem



- use an object for each character in a text document editor
- use a layout object for each widget in a GUI

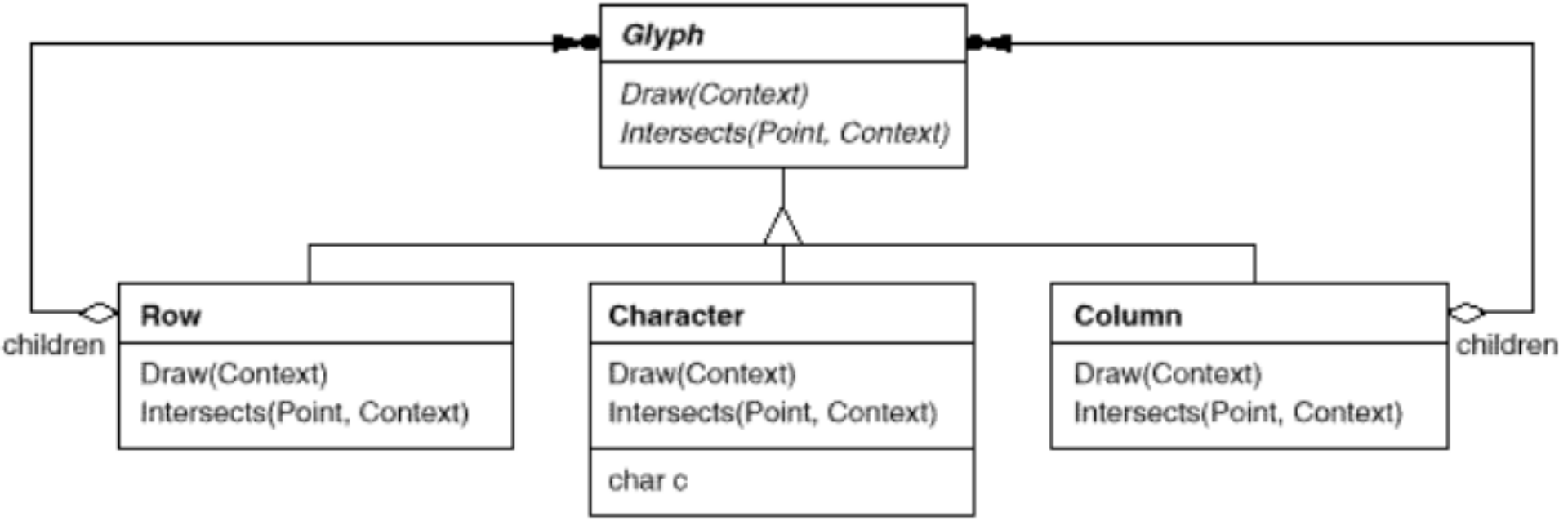
http://home.gwu.edu/~blankeng/Classes/CSCI253/Flyweight%20Pattern.pdf

Page Objects



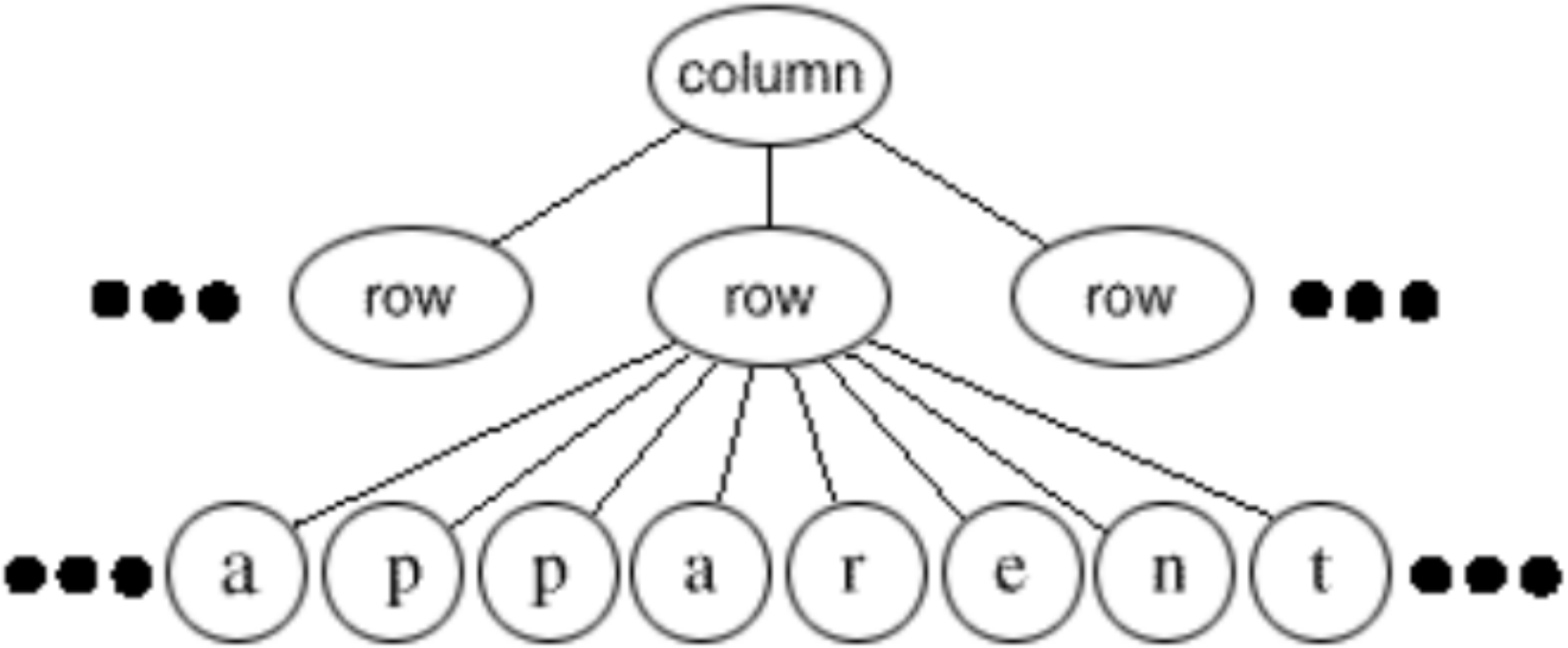
http://home.gwu.edu/~blankeng/Classes/CSCI253/Flyweight%20Pattern.pdf

Page Classes



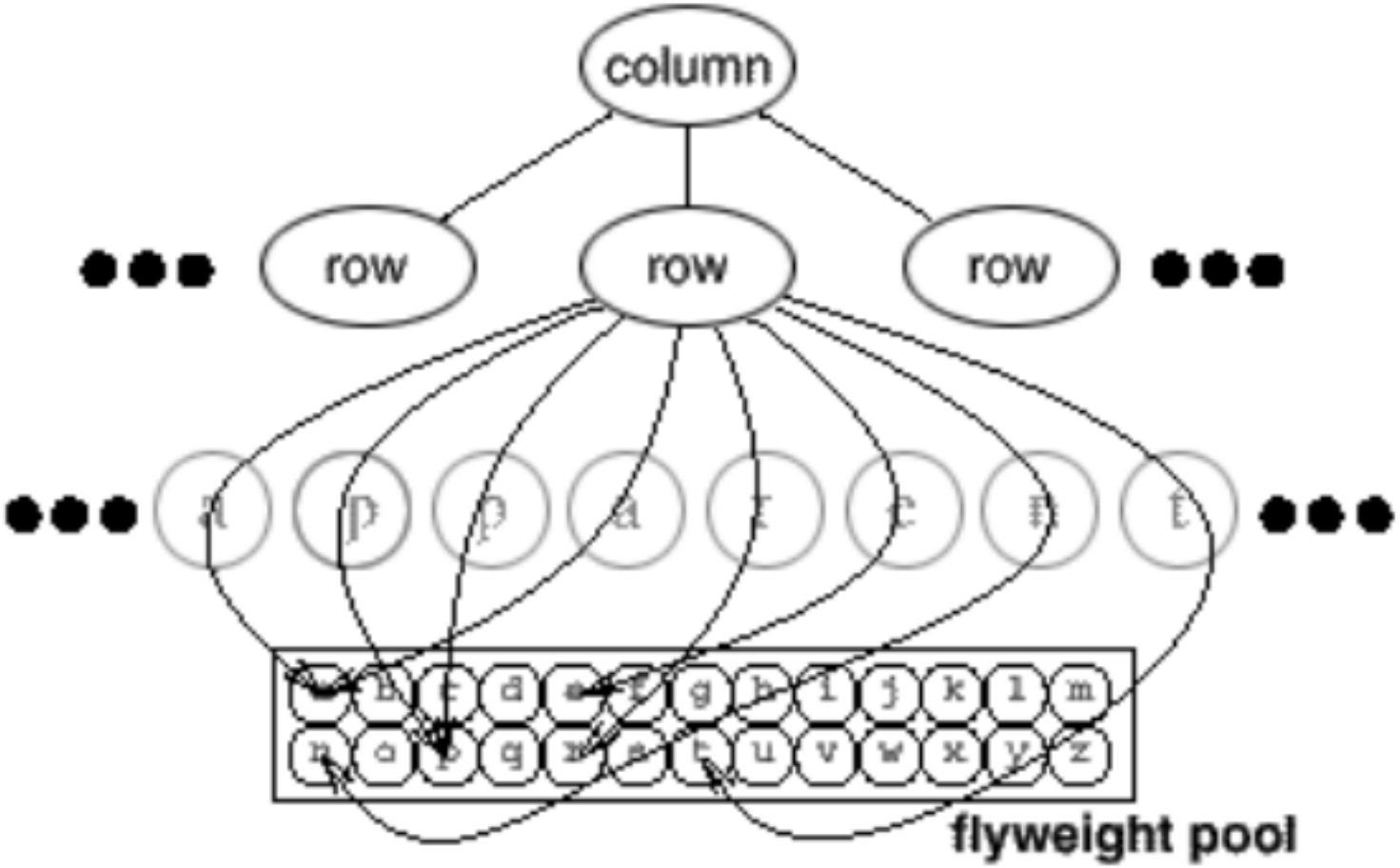
http://home.gwu.edu/~blankeng/Classes/CSCI253/Flyweight%20Pattern.pdf

Object Hierarchy



http://home.gwu.edu/~blankeng/Classes/CSCI253/Flyweight%20Pattern.pdf

Flyweight Pool



http://home.gwu.edu/~blankeng/Classes/CSCI253/Flyweight%20Pattern.pdf

Example

Java API

Example

Java API

valueOf

```
public static Integer valueOf(String s,
                              int radix)
                       throws NumberFormatException
```

Returns an `Integer` object holding the value extracted from the specified `String` when parsed with the radix given by the second argument. The first argument is interpreted as representing a signed integer in the radix specified by the second argument, exactly as if the arguments were given to the `parseInt(java.lang.String, int)` method. The result is an `Integer` object that represents the integer value specified by the string.

In other words, this method returns an `Integer` object equal to the value of:

```
new Integer(Integer.parseInt(s, radix))
```

Parameters:

`s` - the string to be parsed.

`radix` - the radix to be used in interpreting `s`

Returns:

an `Integer` object holding the value represented by the string argument in the specified radix.

Throws:

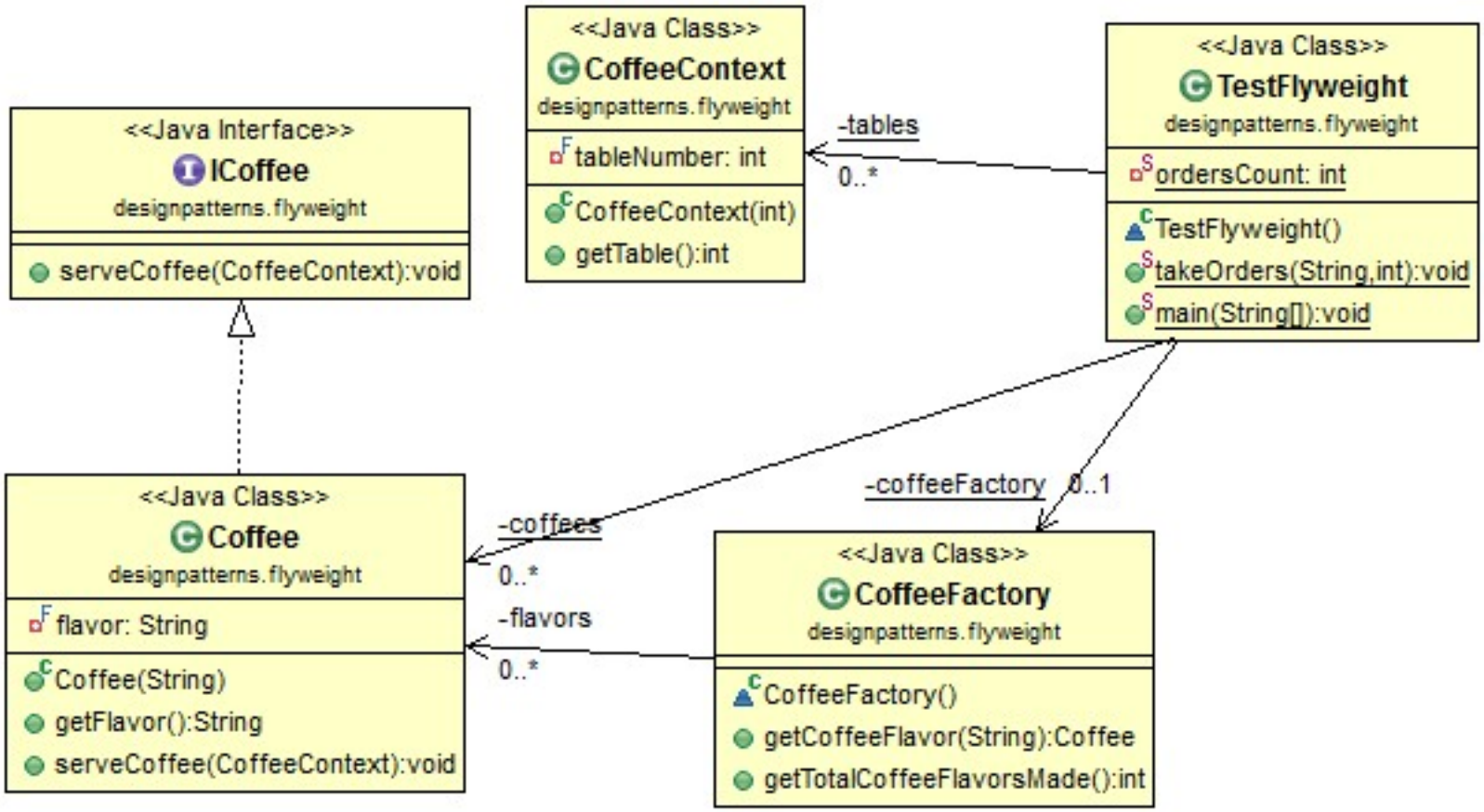
`NumberFormatException` - if the `String` does not contain a parsable `int`.

Java API

```
/**
* Returns an {@code Integer} instance representing the specified {@code int} value.
* If a new {@code Integer} instance is not required, this method should generally be used
* in preference to the constructor {@link #Integer(int)}, as this method is likely
* to yield significantly better space and time performance by
* caching frequently requested values.
*
* This method will always cache values in the range -128 to 127,
* inclusive, and may cache other values outside of this range.
*/
public static Integer valueOf(int i)
{
    assert IntegerCache.high >= 127;
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

Example

Coffee!


<<Java Interface>>
ICoffee
designpatterns.flyweight
serveCoffee(CoffeeContext):void
<<Java Class>>
CoffeeContext
designpatterns.flyweight
tableNumber: int
CoffeeContext(int)
getTable():int
-tables
0..*
<<Java Class>>
TestFlyweight
designpatterns.flyweight
ordersCount: int
TestFlyweight()
takeOrders(String,int):void
main(String[]):void
<<Java Class>>
Coffee
designpatterns.flyweight
flavor: String
Coffee(String)
getFlavor():String
serveCoffee(CoffeeContext):void
-coffees
0..*
-flavors
0..*
-coffeeFactory
0..1
<<Java Class>>
CoffeeFactory
designpatterns.flyweight
CoffeeFactory()
getCoffeeFlavor(String):Coffee
getTotalCoffeeFlavorsMade():int

Implementation *(for your reference)*

```
// Flyweight object interface
interface ICoffee {
    public void serveCoffee(CoffeeContext context);
}
```

```
/// Concrete Flyweight object
class Coffee implements ICoffee {
    private final String flavor;

    public Coffee(String newFlavor) {
        this.flavor = newFlavor;
    }

    public String getFlavor() {
        return this.flavor;
    }

    public void serveCoffee(CoffeeContext context) {
        System.out.println("Serving " + flavor + " to table " + context.getTable());
    }
}
```

Implementation *(for your reference)*

```
// A context, here is table number
class CoffeeContext {
   private final int tableNumber;

   public CoffeeContext(int tableNumber) {
      this.tableNumber = tableNumber;
   }

   public int getTable() {
      return this.tableNumber;
   }
}
```

```
// CoffeeFactory: it only create a new coffee when necessary.
//The FlyweightFactory
class CoffeeFactory {

   private HashMap<String, Coffee> flavors
                              = new HashMap<String, Coffee>();

   public Coffee getCoffeeFlavor(String flavorName) {
      Coffee flavor = flavors.get(flavorName);
      if (flavor == null) {
         flavor = new Coffee(flavorName);
         flavors.put(flavorName, flavor);
      }
      return flavor;
   }

   public int getTotalCoffeeFlavorsMade() {
      return flavors.size();
   }
}
```

Implementation *(for your reference)*

```
public class Waitress {                                          //Waitress serving coffee
   private static Coffee[ ] coffees = new Coffee[20];
   private static CoffeeContext[ ] tables = new CoffeeContext[20];
   private static int ordersCount = 0;
   private static CoffeeFactory coffeeFactory;

   public static void takeOrder(String flavorIn, int table) {
       coffees[ordersCount] = coffeeFactory.getCoffeeFlavor(flavorIn);
       tables[ordersCount] = new CoffeeContext(table);
       ordersCount++;
   }
   public static void main(String[] args) {
      coffeeFactory = new CoffeeFactory();
       takeOrder("Cappuccino", 2);       takeOrder("Cappuccino", 2);       takeOrder("Regular Coffee", 1);    takeOrder("Regular Coffee", 2);
      takeOrder("Regular Coffee", 3);    takeOrder("Regular Coffee", 4);   takeOrder("Cappuccino", 4);        takeOrder("Cappuccino", 5);
      takeOrder("Regular Coffee", 3);    takeOrder("Cappuccino", 3);
       for (int i = 0; i < ordersCount; ++i) {
                        coffees[i].serveCoffee(tables[i]);
       }
       System.out.println("\nTotal Coffee objects made: " +  coffeeFactory.getTotalCoffeeFlavorsMade());
   }
}
```

How

How

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.
2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
3. Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
4. Create a Factory that can cache and reuse existing class instances.
5. The client must use the Factory instead of the new operator to request objects.
6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Comparisons

Comparisons

- **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object represent an entire subsystem.
- **Flyweight** is often combined with **Composite** to implement shared leaf nodes.
- Terminal symbols within Interpreter's abstract syntax tree can be shared with **Flyweight**.
- **Flyweight** explains when and how State objects can be shared.