**Flyweight Pattern**

### 1/ Before using Flyweight Pattern

Trying to use objects at very low levels of granularity is nice, but the overhead may be prohibitive. Flyweight suggests removing the non-shareable state from the class, and having the client supply it when methods are called. This places more responsibility on the client, but, considerably fewer instances of the Flyweight class are now created. Sharing of these instances is facilitated by introducing a Factory class that maintains a "cache" of existing Flyweights.

In this example, the "X" state is considered shareable (within each row anyways), and the "Y" state has been externalized (it is supplied by the client when report() is called).

**class** **Gazillion**

{

**public**:

Gazillion()

{

m\_value\_one = s\_num / Y;

m\_value\_two = s\_num % Y;

++s\_num;

}

**void** report()

{

cout << m\_value\_one << m\_value\_two << ' ';

}

**static** **int** X, Y;

**private**:

**int** m\_value\_one;

**int** m\_value\_two;

**static** **int** s\_num;

};

**int** Gazillion::X = 6, Gazillion::Y = 10, Gazillion::s\_num = 0;

**int** **main**()

{

Gazillion matrix[Gazillion::X][Gazillion::Y];

**for** (**int** i = 0; i < Gazillion::X; ++i)

{

**for** (**int** j = 0; j < Gazillion::Y; ++j)

matrix[i][j].report();

cout << '\n';

}

}

#### Output

00 01 02 03 04 05 06 07 08 09

10 11 12 13 14 15 16 17 18 19

20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39

40 41 42 43 44 45 46 47 48 49

50 51 52 53 54 55 56 57 58 59

**2/ Flyweights:**

***History:*** The flyweight pattern was first coined and extensively explored by [Paul Calder](https://en.wikipedia.org/w/index.php?title=Paul_Calder&action=edit&redlink=1) and [Mark Linton](https://en.wikipedia.org/w/index.php?title=Mark_Linton&action=edit&redlink=1) in 1990 to efficiently handle glyph information in a [*WYSIWYG (What you see is what you get) document editor*](https://en.wikipedia.org/wiki/WYSIWYG)*.*

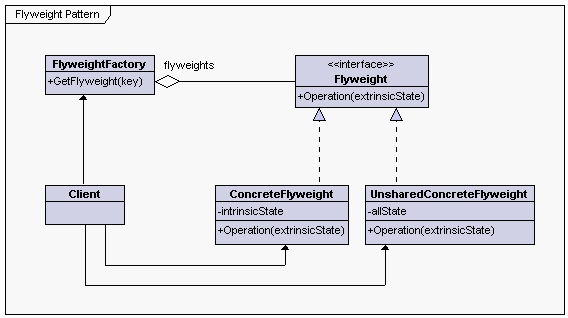
- Used when you need to create a large number of similar objects

- To reduce memory usage, you share objects that are similar in some ways rather than always creating new ones.

***Flyweight state:*** Flyweight describes how to share objects, so that their use at fine granularity is not cost prohibitive. A key concept is the distinction between "intrinsic" and "extrinsic" state. Intrinsic state consists of information that is independent of the flyweight's context - information that is sharable (i.e. each Icon's name, width, and height).

***Flyweight factories:*** Clients should not instantiate Flyweights directly, they should obtain them exclusively from a Flyweight factory object to ensure they are shared properly.

***Class Diagram:***

****

***Flyweight*** - declares an interface through which flyweights can receive and act on extrinsic state

***ConcreteFlyweight*** - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context

***UnsharedConcreteFlyweight*** - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it does not enforce it.

***FlyweightFactory*** - creates and manages flyweight objects

- ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

***Client***- maintains a reference to flyweight(s)

- computes or stores the extrinsic state of flyweight(s)

### 3/ After using Flyweight Pattern

**class** **Gazillion**

{

**public**:

Gazillion(**int** value\_one)

{

m\_value\_one = value\_one;

cout << "ctor: " << m\_value\_one << '\n';

}

~Gazillion()

{

cout << m\_value\_one << ' ';

}

**void** report(**int** value\_two)

{

cout << m\_value\_one << value\_two << ' ';

}

**private**:

**int** m\_value\_one;

};

**class** **Factory**

{

**public**:

**static** Gazillion \*get\_fly(**int** in)

{

**if** (!s\_pool[in])

s\_pool[in] = **new** Gazillion(in);

**return** s\_pool[in];

}

**static** **void** clean\_up()

{

cout << "dtors: ";

**for** (**int** i = 0; i < X; ++i)

**if** (s\_pool[i])

**delete** s\_pool[i];

cout << '\n';

}

**static** **int** X, Y;

**private**:

**static** Gazillion \*s\_pool[];

};

**int** Factory::X = 6, Factory::Y = 10;

Gazillion \*Factory::s\_pool[] =

{

0, 0, 0, 0, 0, 0

};

**int** **main**()

{

**for** (**int** i = 0; i < Factory::X; ++i)

{

**for** (**int** j = 0; j < Factory::Y; ++j)

Factory::get\_fly(i)->report(j);

cout << '\n';

}

Factory::clean\_up();

}

#### Output

ctor: 0

00 01 02 03 04 05 06 07 08 09

ctor: 1

10 11 12 13 14 15 16 17 18 19

ctor: 2

20 21 22 23 24 25 26 27 28 29

ctor: 3

30 31 32 33 34 35 36 37 38 39

ctor: 4

40 41 42 43 44 45 46 47 48 49

ctor: 5

50 51 52 53 54 55 56 57 58 59

dtors: 0 1 2 3 4 5

**4/ Intent**

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

- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

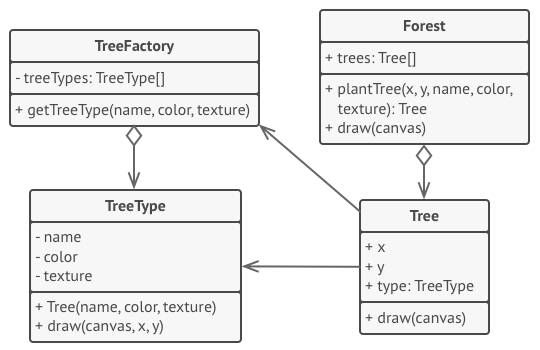
\* When representing large text documents, for example, creating an object for each character in the document would result in a huge number of objects that could not be processed efficiently.

### 5/ 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.

**6/ Another Example**

In this example, the Flyweight pattern helps to reduce memory usage when rendering millions of tree objects on a canvas.



The pattern extracts the repeating intrinsic state from a main Tree class and moves it into the flyweight class TreeType.

Now instead of storing the same data in multiple objects, it’s kept in just a few flyweight objects and linked to appropriate Tree objects which act as contexts. The client code creates new tree objects using the flyweight factory, which encapsulates the complexity of searching for the right object and reusing it if needed.

// The flyweight class contains a portion of the state of a

// tree. These fields store values that are unique for each

// particular tree. For instance, you won't find here the tree

// coordinates. But the texture and colors shared between many

// trees are here. Since this data is usually BIG, you'd waste a

// lot of memory by keeping it in each tree object. Instead, we

// can extract texture, color and other repeating data into a

// separate object which lots of individual tree objects can

// reference.

**class** **TreeType** **is**

**field** name

**field** color

**field** texture

**constructor** TreeType(name, color, texture) { ... }

**method** draw(canvas, x, y) **is**

// 1. Create a bitmap of a given type, color & texture.

// 2. Draw the bitmap on the canvas at X and Y coords.

// Flyweight factory decides whether to re-use existing

// flyweight or to create a new object.

**class** **TreeFactory** **is**

**static** **field** treeTypes: collection of tree types

**static** **method** getTreeType(name, color, texture) **is**

type = treeTypes.find(name, color, texture)

**if** (type == **null**)

type = **new** TreeType(name, color, texture)

treeTypes.add(type)

**return** type

// The contextual object contains the extrinsic part of the tree

// state. An application can create billions of these since they

// are pretty small: just two integer coordinates and one

// reference field.

**class** **Tree** **is**

**field** x,y

**field** type: TreeType

**constructor** Tree(x, y, type) { ... }

**method** draw(canvas) **is**

type.draw(canvas, **this**.x, **this**.y)

// The Tree and the Forest classes are the flyweight's clients.

// You can merge them if you don't plan to develop the Tree

// class any further.

**class** **Forest** **is**

**field** trees: collection of Trees

**method** plantTree(x, y, name, color, texture) **is**

type = TreeFactory.getTreeType(name, color, texture)

tree = **new** Tree(x, y, type)

trees.add(tree)

**method** draw(canvas) **is**

**foreach** (tree in trees) do

tree.draw(canvas)

### 7/ References

<http://wiki.western.edu/mcis/index.php?title=James_Martin/CIS410/HMWK_6>

<https://sourcemaking.com/design_patterns/flyweight/cpp/1>

<https://refactoring.guru/design-patterns/flyweight>