**Flyweight Pattern**

* **Introduction:**
  + Our system needs a large number of objects of a particular class & maintaining these instances is a performance concern.
  + Flyweight allow us to share an object in multiple contexts. But instead of sharing entire object, which may not be feasible, we divide object states in two parts: intrinsic (state that is shared in every context) & extrinsic (context specific state). We create objects with only intrinsic state and share them in multiple contexts.
  + Client or user of object provides the extrinsic state to object to carry out its functionality.
  + We provide a factory so client can get required flyweight objects based on some key to identify flyweight.
* **UML:**

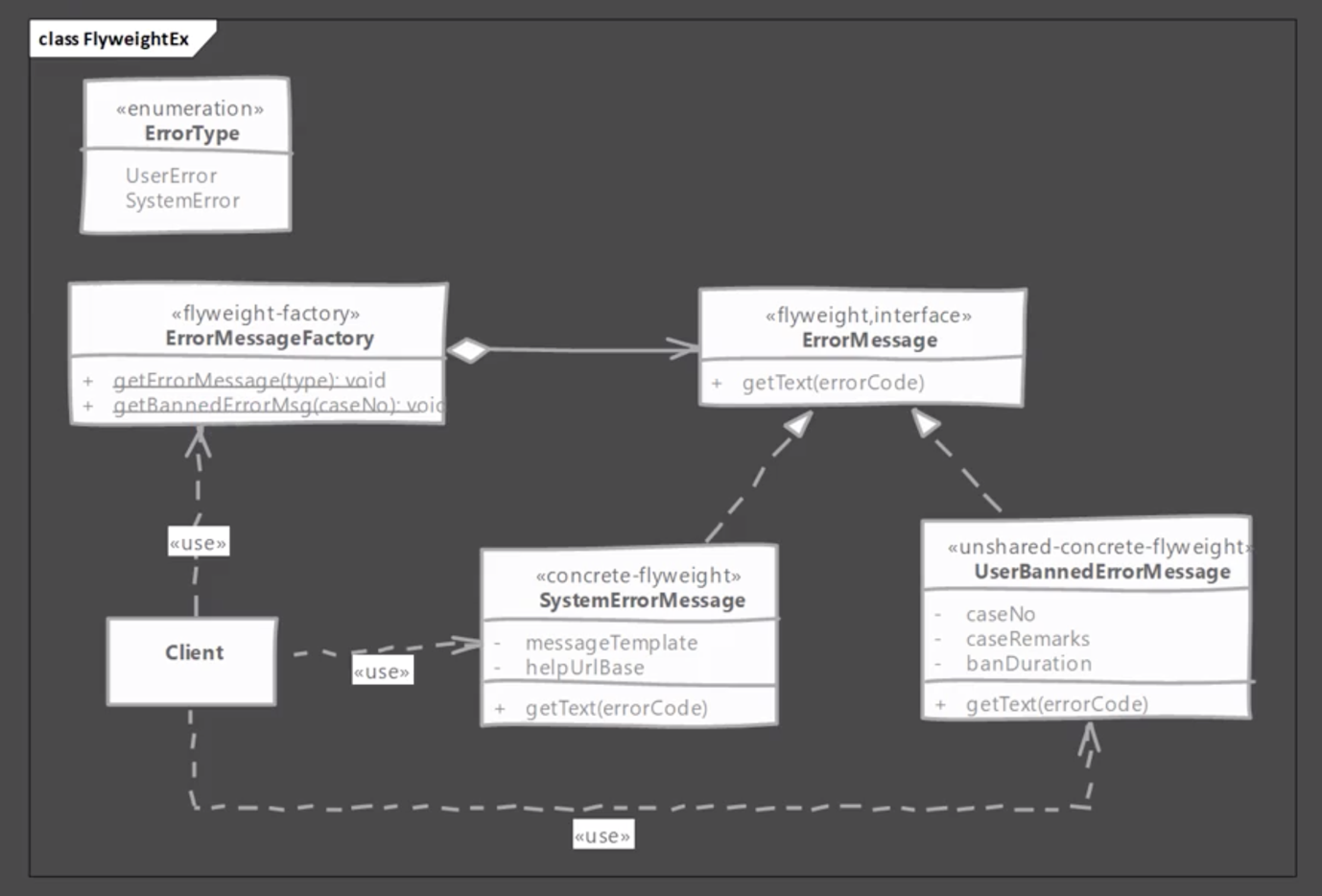
**A screenshot of a cell phone

Description automatically generated**

**A screenshot of a cell phone

Description automatically generated**

* **Implementation Steps:**
  + We start by identifying “intrinsic” & “extrinsic” state of our object.
    - We can create an interface for flyweight to provide common methods that can accept extrinsic state.
    - In implementation of shared flyweight we add intrinsic state & also implement methods.
    - In unshared flyweight implementation we simply ignore the extrinsic state argument as well we have all state within object.
  + Next we implement the flyweight factory which caches flyweights and also provides method to get them.
  + In our client we either maintain the extrinsic state or compute it on the fly when using flyweight.
* **Example UML:**

****

* **Implementation & Design Consideration:**
  + A factory is necessary with flyweight design pattern as client code needs easy way to get hold of shared flyweight. Also number of shared instances can be large so a central place is good strategy to keep track of all of them.
  + Flyweight’s intrinsic state should be immutable for successful use of flyweight pattern.

**Design Consideration:**

* + Usability of flyweight is entirely dependent upon presence of sensible extrinsic state in object which can be moved out of object without any issue.
  + Some other design patterns like state and strategy can make best use of flyweight.
* **Example:**
  + Java uses flyweight pattern for Wrapper classes like java.lang.Integer, Shor, Byte etc. Here the valueOf() static method serves as the factory method.

Note: Float and Double don’t use this particular design pattern.

**A screenshot of a cell phone

Description automatically generated**

* + String pool which is maintained by JVM is also an example of flyweight. We can call the intern() method on a String object to explicitly request this String object to be interned. This method will returned a reference to already cached object if present or else will create new String in cache if not present.
* **Comparison with Object Pool:**

|  |  |
| --- | --- |
| **Flyweight** | **Object Pool** |
| State of flyweight object is divided. Client must provide part of state to it. | A pooled object contains all of its state encapsulated within itself. |
| In a typical usage client will not change intrinsic state of flyweight instance as it is shared. | Clients can and will change state of pooled objects. |

* **Pitfalls:**
  + Runtime cost may be added for maintaining extrinsic state. Client code has to either maintain it or compute it every time it needs to use flyweight.
  + It is often difficult to find perfect candidate objects for flyweight. Graphical applications benefit heavily from this pattern however a typical web application may not have a lot of use for this pattern.
* **Quiz:**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**