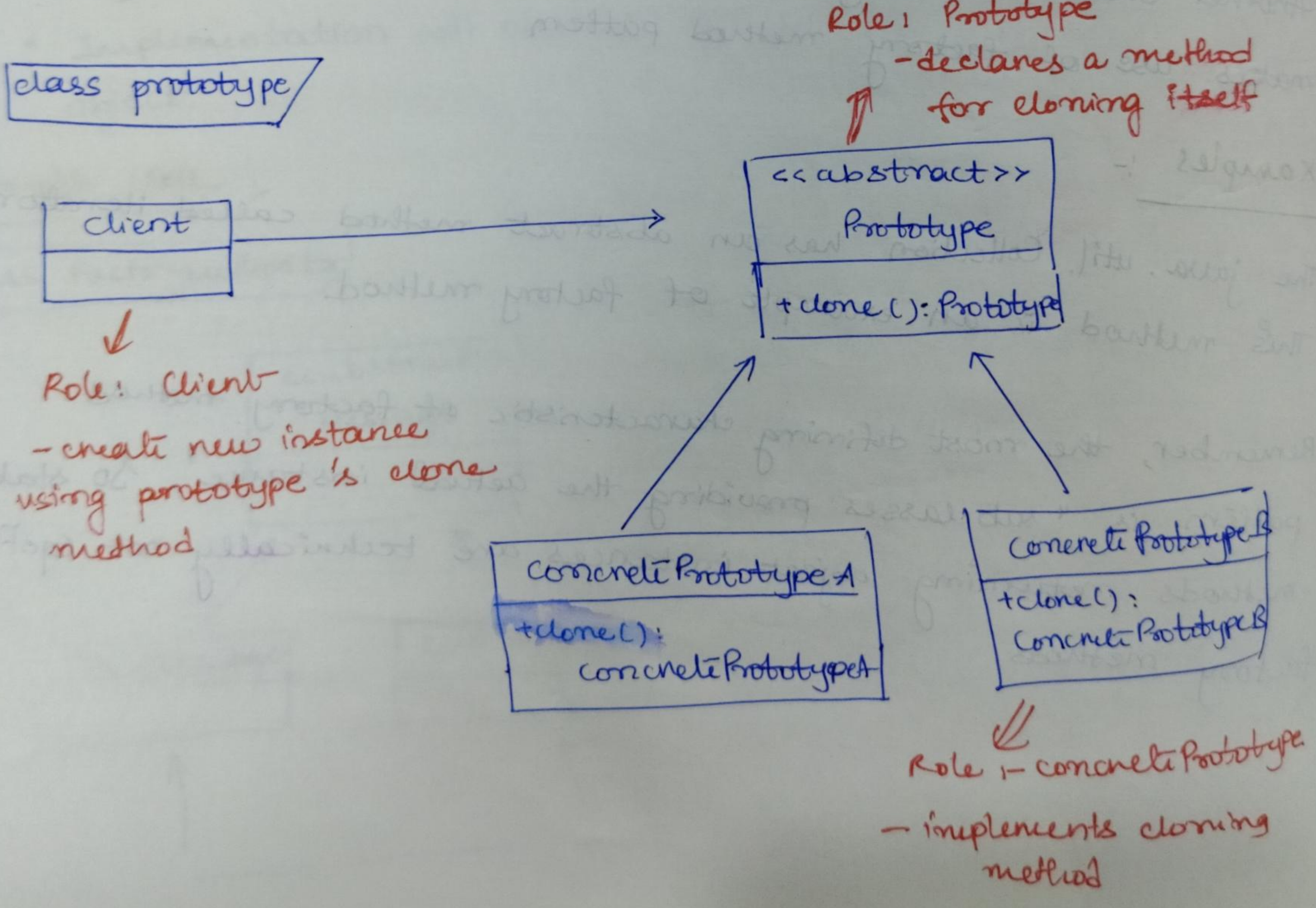
Prototype :-

→ We have a complex object that is costly to create. To create more instances of such class, we use an existing instance of our prototype.

→ Prototype will allow us to make copies of existing objects and save us from having to recreate objects from scratch.



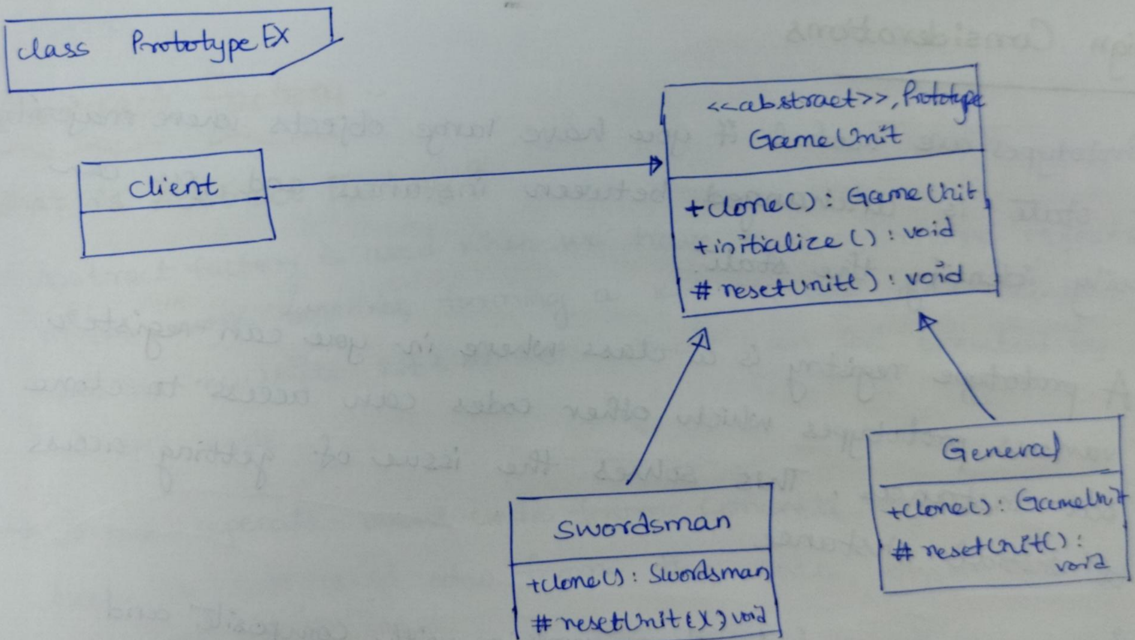
class prototype
Role: Prototype
- declares a method for cloning itself
<<abstract>>
Prototype
+ clone(): Prototype
Client
Role: Client
- create new instance using prototype's clone method
concretePrototypeA
+clone(): concretePrototypeA
ConcreteBPrototypeB
+clone(): ConcretePrototypeB
Role :- concretePrototype
- implements cloning method

Implement a Prototype :-

→ We start by creating a class which will be a prototype
- The class must implement cloneable Interface
- class should override clone method and return copy of itself.
- The method should declare CloneNotSupported Exception in throws clause to give subclass chance to decide on whether to support cloning.

→ clone method implementation should consider the deep & shallow copy and choose whichever is applicable.

Example UML :-



Implementation Considerations :-

→ Pay attention to the deep copy and shallow copy of references. Immutable fields on clone save the trouble of deep copy.

→ Make sure to reset the mutable state of object before returning the prototype. It's a good idea to implement this in method to allow subclass to initialize themselves.

→ clone ( ) method is protected in Object class and must be overridden to be public to be callable from outside the class.

→ Cloneable is a "marker" interface, an indication that the class supports cloning.

Design Considerations

→ Prototypes are useful if you have large objects where majority of state is unchanged between instances and you can easily identify the state.

→ A prototype registry is a class where in you can register various prototypes which other codes can access to clone out instances. This solves the issue of getting access to initial instance.

→ Prototypes are useful when working with composite and decorater patterns.

Pitfalls :-

→ Usability depends upon the no. of properties in state that are immutable or can be shallow copied.. An object where state is compromised of large number of mutable objects is complicated to clone.

→ In java the default clone operation will only perform shallow copy so if you need a deep copy you've to implement it.

→ Subclass may not be able to support clone and so the code becomes complicated as you have to code for situations where an implementation may not support clone.