Sunday, 6 March 2022

# Java Design Patterns & SOLID Design Principles

---

## Creational Design Patterns

### INTRODUCTION

*What Are They?*

- Creational design patterns deal with the process of creation of objects of classes.
- The following patterns are considered creational:
  - Builder
  - Simple Factory
  - Factory Method
  - Prototype
  - Singleton
  - Abstract Factory
  - Object Pool

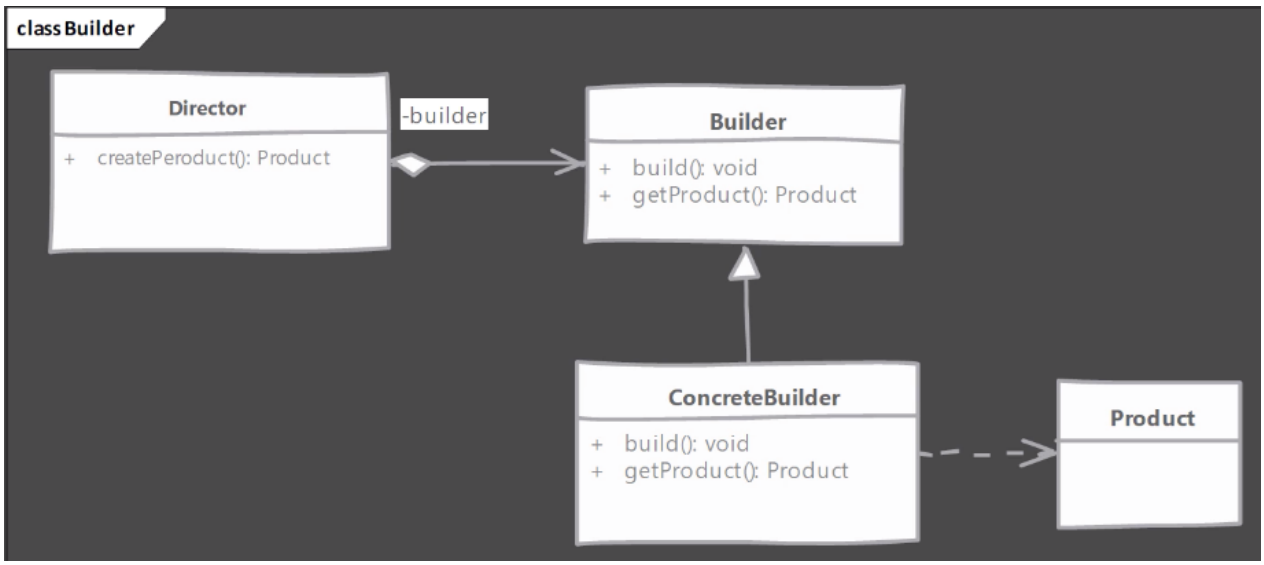
### BUILDER

*What Is It?*

- A **separate class** that is used to abstract and remove the logic related to **object construction** from client code.
- Used when there is a **complex process to construct an object** involving multiple steps.

*What Problems Does It Solve?*

- Class constructor contains a lot of information and instances of the class are immutable: **Builder** pattern allows to avoid writing constructors with a lot of arguments and still keep the instance immutable.
- Objects that need other objects or “parts” to construct them.



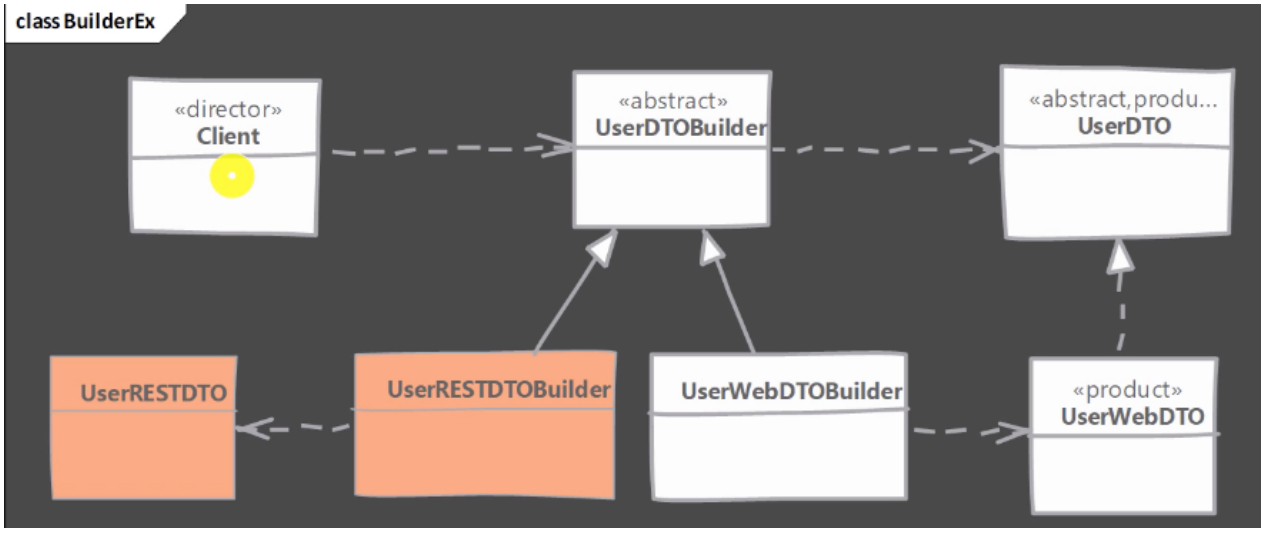

*UML Diagram*

In UML Diagram:

- **Product**: final complex object that we want to create.
- **Builder**: provides interface for creating parts of the product (defines methods)
  - Allows to specify the arguments one at a time.
  - Provides a method to assemble the final object (i.e., **build()** method).
  - Provides a method to query the already built object from the builder.
- **ConcreteBuilder**: implementation of the builder
  - Provides method implementations.
  - Optionally, keeps track of the product it creates.
- **Director**: provides logic as how the builder should be used.
  - Uses builder to construct objects
  - Knows the steps and their sequence to build product.

*What Are Builder Implementation Steps?*

1. Identify the **parts** of the product & provide **methods** to create those parts.
2. Provide a method to **assemble** or build the product/object.
3. Provide a way/method to get **fully built object out**. Optionally, builder can keep reference to a product it has built so the same can be returned in the future.
4. Create a separate class for the **director** or let the client play the role of director.




*Example UML Diagram*

*Notes on Builder Implementation:*

- Abstract Builder Interface provides methods returning the reference to the builder object to allow for **method chaining**.
- A Builder class is often implemented as an inner static class in the product class that it intends to build (note that the setter methods of the product class are often declared **private** for immutability, and the inner static class would be able to use them).

*Builder Implementation & Design Considerations*

- Easily create an immutable class by implementing builder as an inner static class.
- Director role is rarely implemented as separate class, typically the **consumer of the object instance** or the **client** handles that role.
- Abstract builder is also not required if ‘product' itself is not part of any inheritance hierarchy - you can directly create the concrete builder.
- If you are running into a “too many constructor arguments”problem, it’s a good indication for creating a builder.

*Builder Pitfalls:*

- Possibility of a partially initialised object; user code can set only a few or none of the properties using withXXX methods and call build(). If required properties are missing, build method should provide suitable **defaults** or **throw exception**.

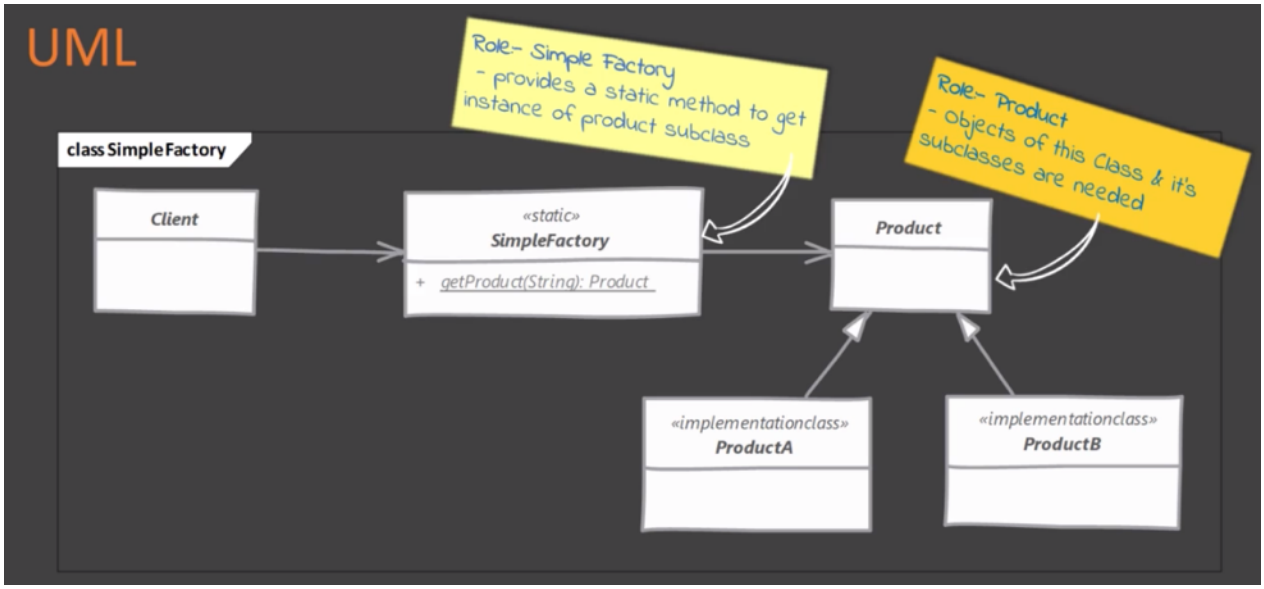
## SIMPLE FACTORY

*What Is It?*

- A design pattern to solve the problem of multiple-type instantiation where the choice is based on some simple criteria.
- You simply move the instantiation logic to a separate class (most commonly to a static method of this class).
- Some might not consider the simple factory a design pattern since it’s simply a method that encapsulates object instantiation - nothing complex going on in that method.

*What Are Simple Factory Implementation Steps?*

- Create a **separate class** for simple factory.
- Add a **static method** that accepts **argument** to decide which class to instantiate and returns the desired object instance.




*Simple Factory Implementation & Design Considerations:*

- Simple factory can be just a method in existing class but adding a separate class allows other parts of your code to use simple factory more easily.
- It doesn't need any state tracking so it's best to keep this as a static method.
- It can use other design patterns inside to construct objects.
- In case you want to **specialise** simple factory in sub classes, you need factory method design pattern instead.

*Simple Factory Pitfalls:*

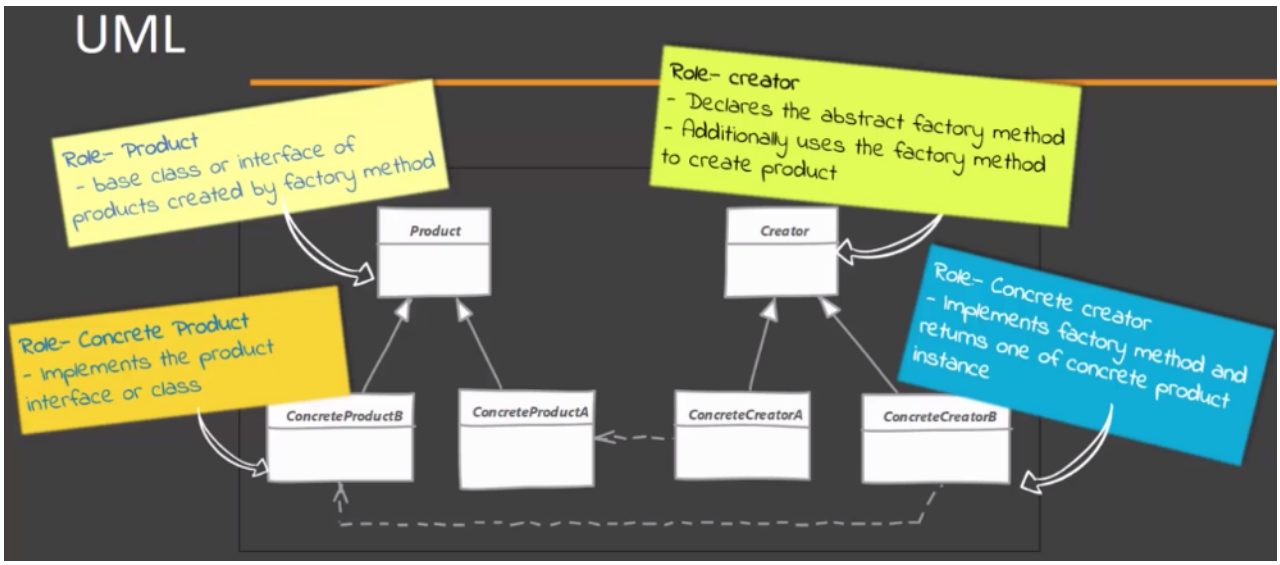
- The criteria used by simple factory to decide which object to instantiate can get more complex over time - if you find yourself in such situation, then use factory method instead.

Sunday, 6 March 2022

# FACTORY METHOD

*What Is It?*

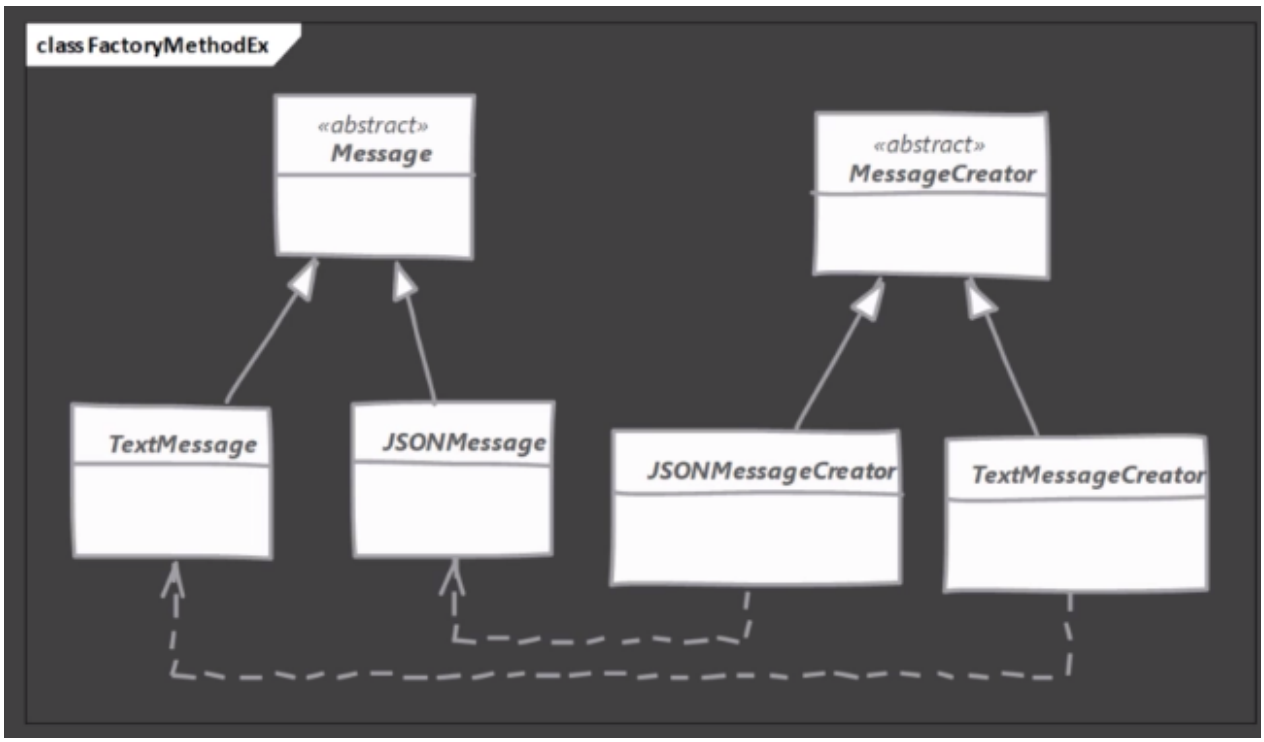
- A creational design pattern used when you want to move the object creation logic from the code to a separate class.
- You use this pattern when you do not know in advance which class you may need to instantiate beforehand & also to allow new classes to be added to system and handle their creation w/o affecting client code.
- You let the subclasses decide which object to instantiate by overriding the factory method.




*What Are Factory Method Implementation Steps?*

**1.** Start by creating a class of our creator:

- Creator itself can be concrete if it can provide a default object it can be abstract.
- Implementations (subclasses of the creator) will override the method and return an actual concrete object.




*Factory Method Implementation & Design Considerations:*

- Creator class can be a concrete class & provide a default implementation for the factory method (if there is a default product) - in that case you'll create some "default" object in base creator.
- You can also use the simple factory way of accepting additional arguments to choose between different object types - subclasses can then override factory method to selectively create different objects for some criteria.
- The creator hierarchy in factory method pattern reflects the product hierarchy! We typically end up with a concrete creator per product concrete implementation.
- Template method design patter and "Abstract Factory" creational design pattern makes use of factory method pattern.

*Notes on Factory Method Implementation:*

- The most defining characteristic of factory method is that the **subclasses are providing the actual instances** - the static method returning object instances are technically not GoF factory method.

*Factory Method Pitfalls:*

- More complex to implement than simple factory - more classes involved and unit testing is needed.
- You have to start with Factory method design pattern from the beginning as it's not easy to refactor the existing code into factory method pattern.
- Sometimes this pattern forces you to subclass just to create appropriate instance.

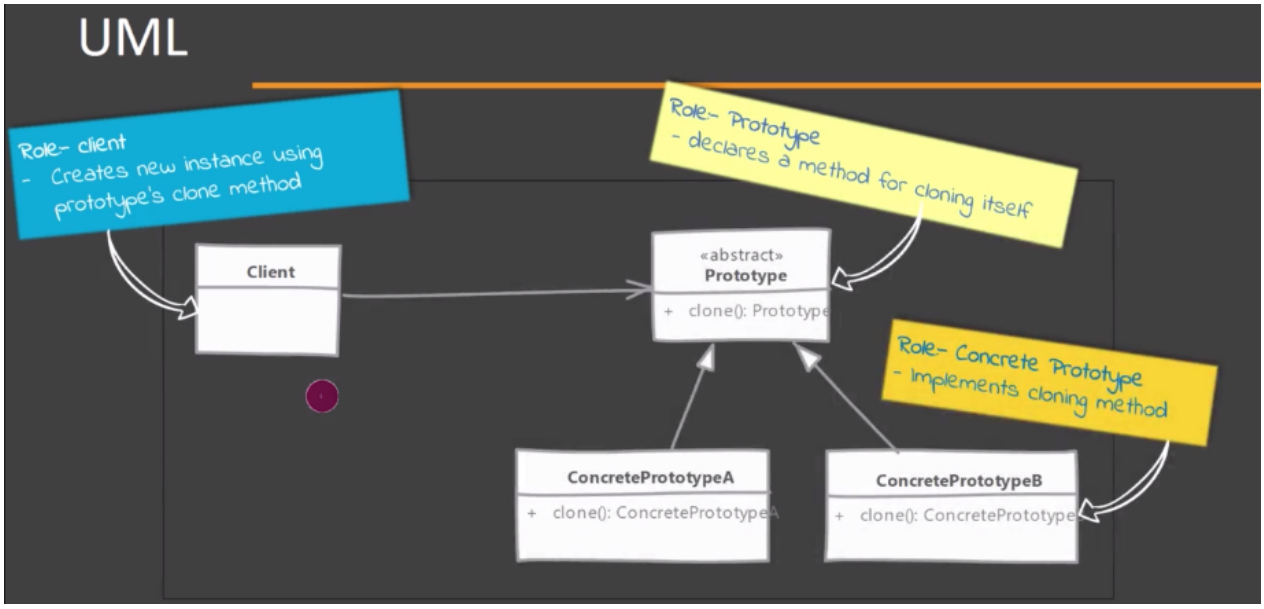
*Factory Method Summary:*

- Use factory method when you want to delegate object instantiation to subclasses - you'd want to do this when you have 'product' inheritance hierarchy and possibility of future additions to that.
- Adding a product and an appropriate creator implementation would be easy and without modifying the existing codebase.
- The creator usually performs additional stuff on the object it creates such as adding the headers, encrypting, etc.
- The client will often create the concrete instance of the creator and after that the interface methods (those of the base creator) will be used to operate.

# PROTOTYPE

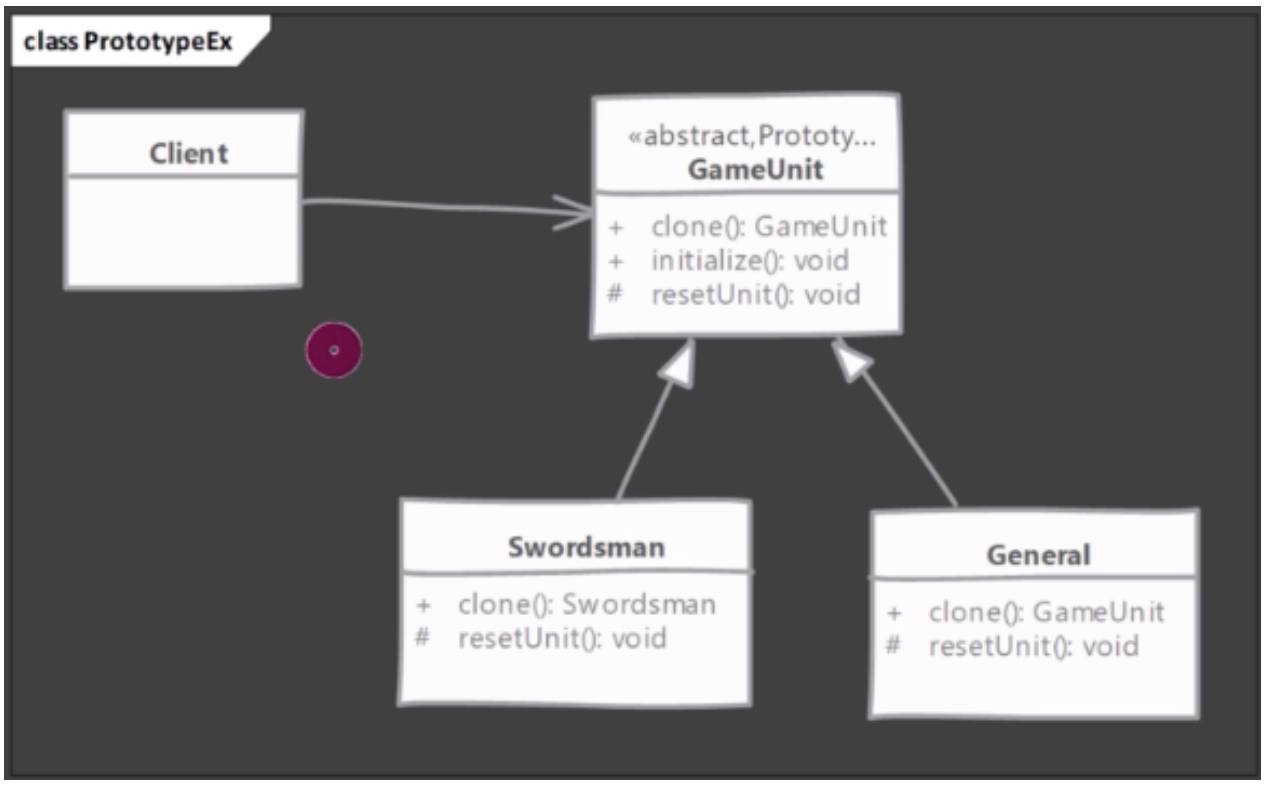
*What Is It?*

- A creational design patter used when you have a complex object that is costly (e.g., performance cost, using external resources) to create - to create more instances of such class, you use an **existing instance as the prototype**.
- It allows us to make copies of existing object & saves us from having to recreate objects from scratch.




*What Are Prototype Implementation Steps?*

1. Create a prototype class
   - Class must implement **Cloneable** (in-built Java interface) interface.
   - Class should override clone method and return a copy of itself.
   - Method should declare **CloneNotSupportedException** in throws clause to give subclasses chance to decide on whether to support cloning.
2. Clone method implementation should consider the deep & shallow copy and choose the one that is applicable.




*Prototype Implementation & Design Considerations:*

- Pay attention to the deep copy and shallow copy of references - immutable field on clones 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 subclasses to initialise themselves.
- **clone()** method is protected in Object class and must overridden to be public to be callable from outside the class.
- **Cloneable** is a "marker" interface - an indication that the class supports cloning.
- Prototypes are useful when 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 you can register various prototype instances which other code can access to clone out instances - this solves the issue of getting access to initial instance.
- Prototypes are useful when working with **Composite** and **Decorator** patterns.

*Prototype Pitfalls:*

- Usability depends upon the number of properties in state that are **immutable** or can be shallow copied - an object where state is comprised of large number of **mutable** objects is **complicated** to clone.
- In Java, the default clone operation will only perform the shallow copy so if you need a deep copy, you have to implement it.
- **Subclasses** 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.

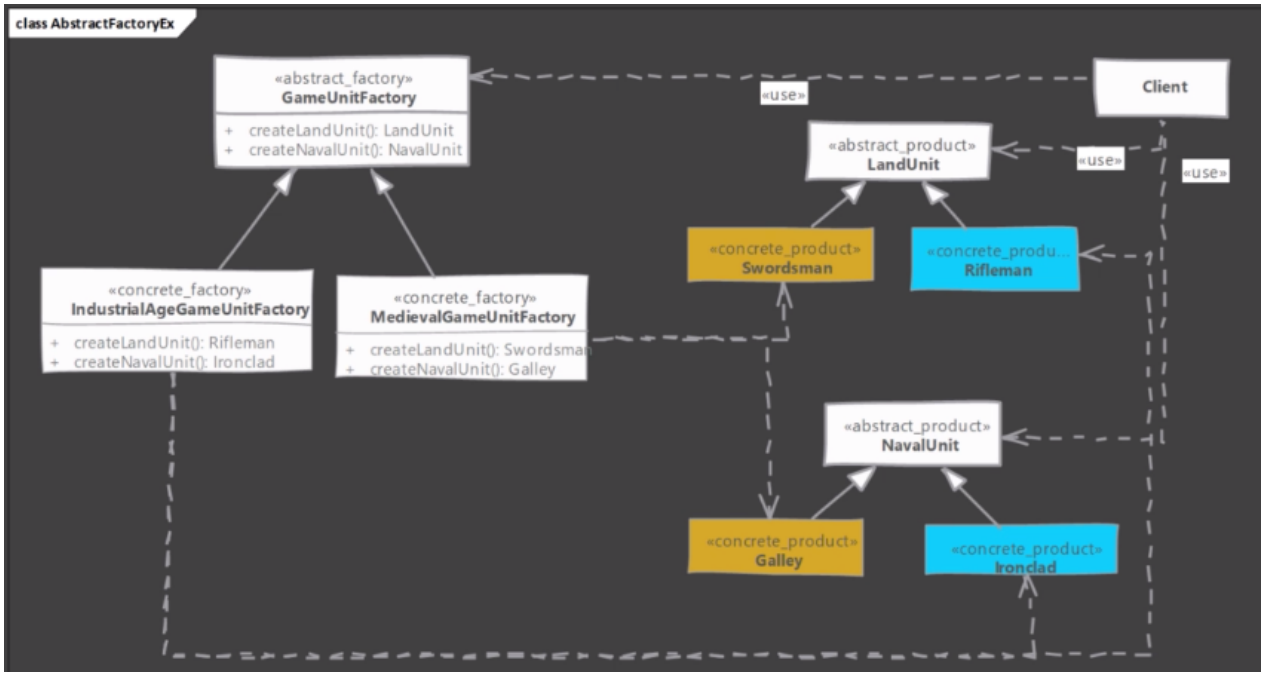
*Prototype Summary:*

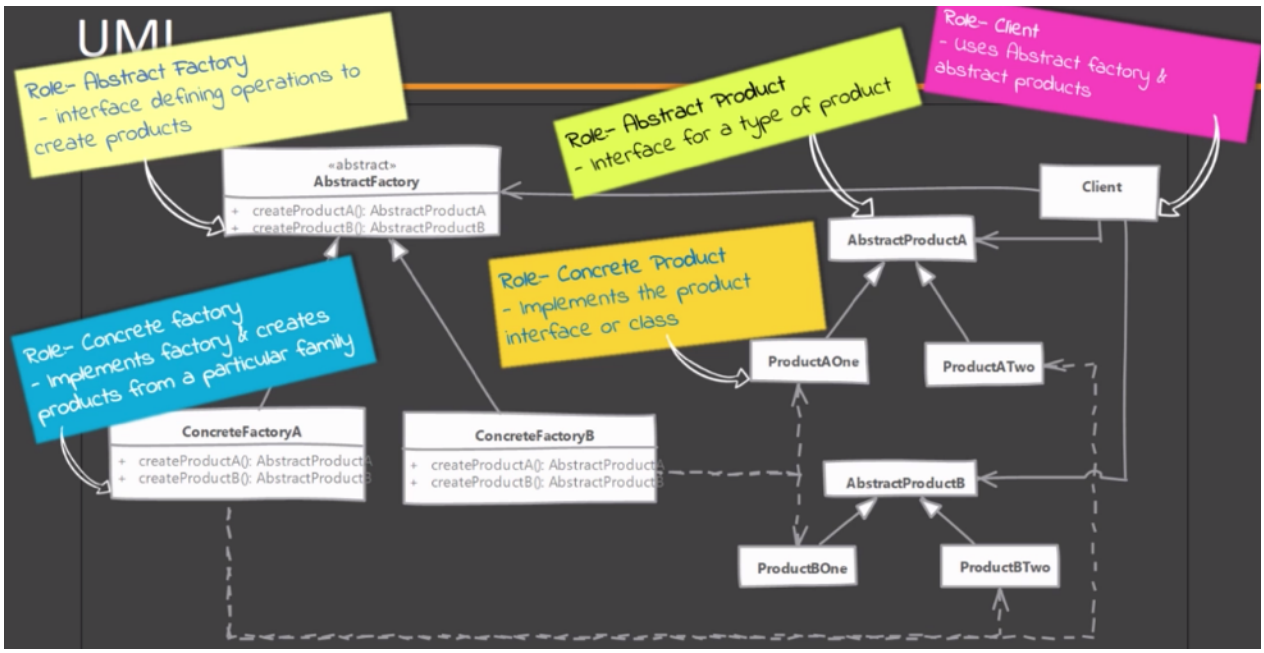
- Think of prototype pattern when you have an object where **construction** of new instance is **costly** or **not possible** (i.e., object is supplied to your code).
- In Java, you typically implement this pattern with **clone()** method.
- Objects with majority of their state **immutable** are good candidates for prototypes.
- Pay attention to the deep/shallow copy considerations.
- Ensure that clone is “initialized”: appropriate states are reset before returning the copy to outside world.

# ABSTRACT FACTORY

*What Is It?*

- A creational design pattern used when you have two or more objects which work together forming a **kit** or **set** and there can be multiple sets or kits that can be created by client code.
- The intent behind is to **separate** the **client code** from concrete objects forming such a set and also from the code which creates these sets.

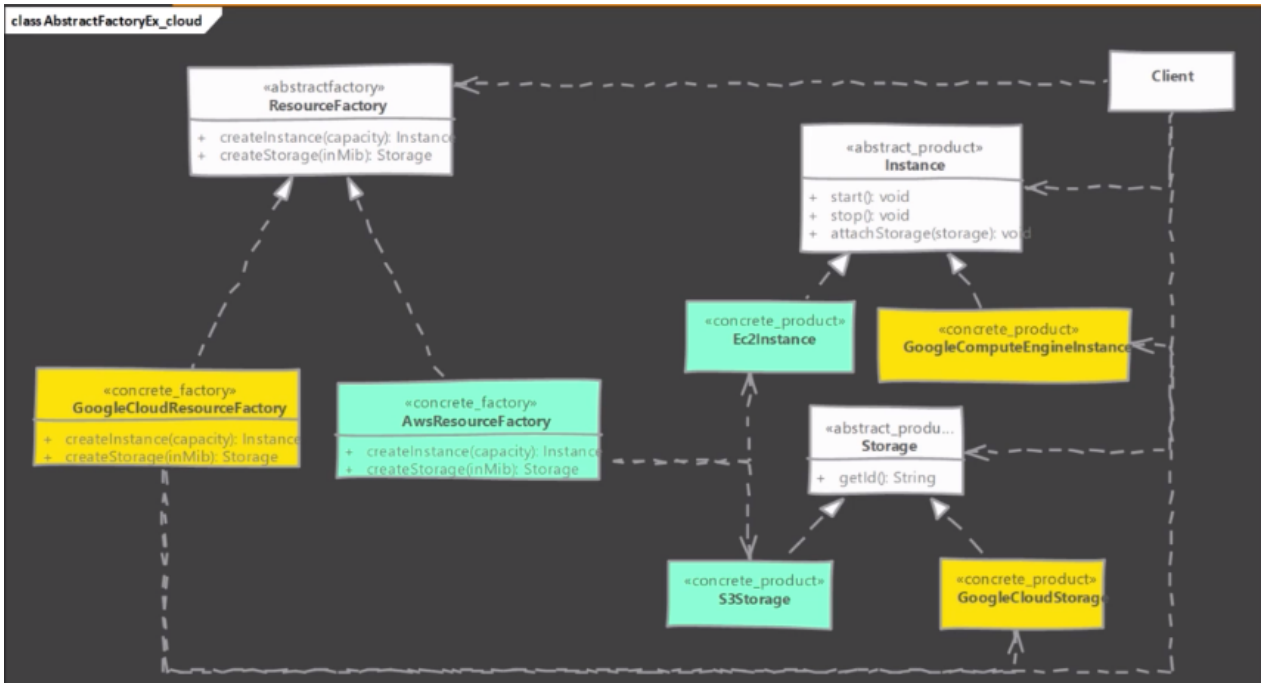


*What Are Prototype Implementation Steps?*

1. Start by studying the product 'sets'
   - Create abstract factory as an abstract class or an interface.
   - Abstract factory defines abstract method for creating products.
   - Provide concrete implementation of factory for each set of products.
2. Remember that factory makes use of factory method pattern - you can think of abstract factory as an object with multiple factory methods.

*Another Example of Abstract Factory Implementation:*




*Factory Implementation & Design Considerations:*

- Factories can be implemented as singletons - we typically ever need only one instance of it anyway.
- Adding a new product type requires changes to the **base factory** as well as **all implementations** of factory.
- Provide the client code with **concrete factory** so that it can create objects.

- When you want to constrain object creations so that they all work together, then abstract factory is your design pattern.
- Abstract factory itself uses factory method pattern.
- If objects are expensive to create then you can transparently switch factory implementations to use **prototype** design pattern to create objects.

*Abstract Factory vs. Factory Method:*

- Hide factories as well as concrete objects used from client code **vs.** hides only the concrete objects which are used from the client code.
- Suitable when multiple objects are designed to work together & client muse use products from single family at a time **vs.** concerned with one product and its subclasses, collaboration of product itself with other objects is irrelevant.

*Abstract Factory Pitfalls:*

- A lot more **complex** to implement than factory method.
- Adding a new product type requires changes to **base factory** AND **all implementations** of factory.
- Hard to visualise the need at start of development and often starts as factory method.
- This pattern is very **specific** to the problem of 'product families'.

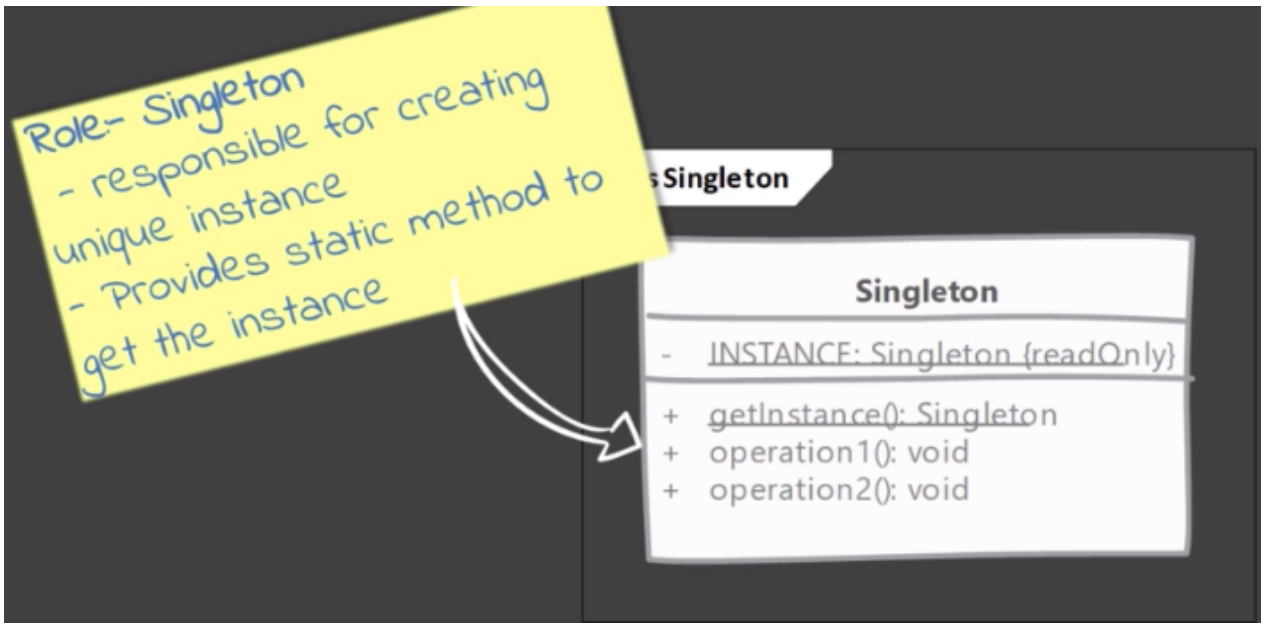
*Abstract Factory Summary:*

- When you have multiple sets of objects where objects in one set work together, use the abstract factory to isolate the client code from concrete objects and their factories.
- Abstract factory itself uses factory method and you can think of it as an object with multiple factory methods.
- Adding new product type requires changes. What are those?
- Concrete factory can be singletons as we need only one stateless instance.
- Client code is unaware of concrete factory class that it using - not tightly coupled (client code works with interface or abstract class references).

# SINGLETON

*What Is It?*

- A singleton class is a class that has only one instance, accessible globally through a single point (via method/field).
- Main problem that it solves is to ensure that only a single instance of this class exists.
- Any state you add in your singleton becomes part of “global state” of your application.




*What Are Singleton Implementation Steps?*

- Controlling instance creation:
  - Class constructors must not be accessible globally (outside of the class).
  - Subclassing/inheritance must not be allowed.
- Keeping track of instance:
  - Class itself is a good place to track the instance.
- Giving access to the singleton instance:
  - A **public static method** is a good choice.
  - Exposing instance as **final public static field** won’t work for all singleton implementations.

*Types of Singleton Implementations:*

- Early initialisation or **Eager Singleton**
  - Create singleton as soon as class is loaded.
- Lazy initialisation or **Lazy Singleton**
  - Singleton is created when it is first required.
  - Double-checked locking implementation: when working with multiple threads, ensure that the initialisation part has **double-checked locking** and make the constant referring to the instance **volatile** to avoid the case where threads are using the ‘cached' version of the variable instead of the 'main-memory' version (with the latest value).
  - Initialisation holder implementation (**preferred way**)**:** by declaring a **private static class** initialisation holder within your singleton class holding the singleton instance, you ensure the lazy initialisation as the instance within inner class is not initialised until that inner class is used (inside the **getInstance()** method).
- **Enum Singleton**:
  - Using enum to create a singleton.
  - Handels the serialisation using Java’s in-built mechanism and still ensure a single instance.

*Singleton Implementation & Design Considerations:*

- **Early/Eager initialisation** is the preferred way - try to use this approach first.
- The ‘classic' singleton pattern implementation uses double-checked locking and volatile field.
- The lazy **initialisation holder** idiom provides best of both worlds - you don’t deal with synchronisation issues directly and it is easy to implement.
- Due to pre-conceptions about what an enum is, it may be hard to sell an enum singleton, especially if it has mutable fields.
- Singleton creation doesn’t need parameters - otherwise, you need a simple factory or factory method pattern.
- Make sure that your singletons are not carrying a lot of mutable global state.

*Singleton vs. Factory Method:*

- Primary purpose or intent of singleton is to ensure that only one instance of a class is ever created **vs.** factory method is primarily used to isolate client code from object creation & delegate object creation to subclasses.
- Singleton instance is created w/o any need of arguments from client code **vs.** factory method allows to parameterise the object creation.

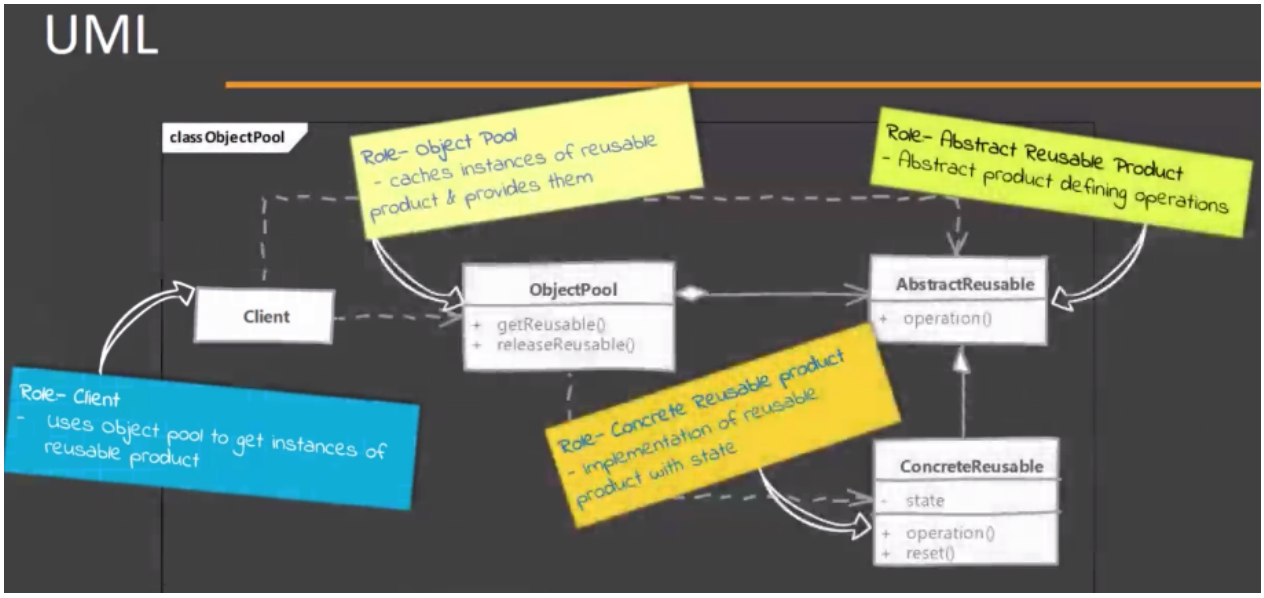
*Singleton Pitfalls:*

- Singleton can deceive you about true **dependencies** of your code - since they are globally accessible, its easy to miss dependencies.
- **Hard to unit test** - you cannot easily mock the instance that is returned.
- Most common way to implement singletons is through **static** variables and they are held per class loader and not per JVM - they may not be truly singleton in an OSGi or web app.
- A singleton carrying around a large **mutable** global state is an indication of abused singleton pattern.

# OBJECT POOL

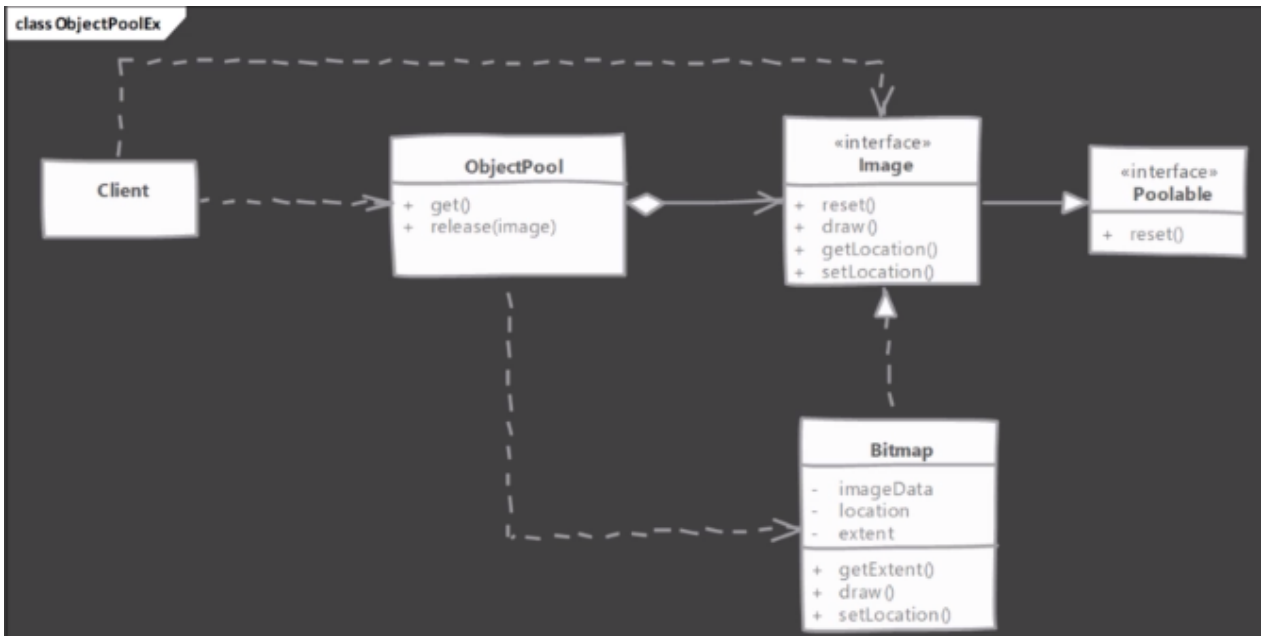
*What Is It?*

- A creation design pattern used when the cost of creating an instance of class is high and you need a large number of objects of this class for short duration.
- You either **pre-create objects** of the class or **collect unused instances** in the in-memory cache - when code needs an object of this class you provide it from this cache.
- One of the most complicated patterns to implement efficiently.




*What Are Object Pool Implementation Steps?*

1. Start by creating class for object pool
   - **A thread-safe caching** of objects should be done in pool.
   - Methods to acquire and release objects should be provided & pool should **reset** cached objects before giving them out.
2. The reusable object must provide methods to reset its state upon ‘release' by code.
3. Decide whether to create new pooled objects when pool is empty or to wait until an object becomes available - choice is influenced by whether the object is tied to a fixed number of external resources.




*Object Pool Implementation & Design Considerations:*

- **Resetting** object state should not be costly operation, otherwise, you may end up losing your performance savings.
- **Pre-caching objects**, meaning creating objets in advance, can be helpful as it won't slow down the code using these objects. However, it may add-up to start up time & memory consumption.
- **Object pool synchronisation** should consider the reset time needed & avoid resetting in synchronised context if possible.
- Object pool can be parameterised to cache & return multiple objects and the acquire method can provide selection criteria.
- Pooling objects is only beneficial if they involve costly initialisation of external resource (e.g., connection, thread) - don't pool objects just to save memory unless you are running into out of memory errors.
- **Don't pool** long lived object or just to save frequent call to new - pooling may actually negatively impact performance in such cases.

*Object Pool vs. Prototype:*

- Cached objects that frequently live throughout programs entire run **vs.** prototype creates object when needed and no caching is done.
- Code using objects from pool has to return them explicitly to the pool - failing to do that may lead to memory and/or resource leaks **vs.** once an object is cloned, no special treatment is needed by client code and object can be used like any regular object.

*Object Pool Pitfalls:*

- Successful implementation depends on **correct use by the client code** - releasing objects back to pool may be vital for proper working.
- The reusable object needs to take care of **resetting** its state in efficient way - some object may not be suitable for pooling because of that.
- **Difficult** to use in **refactoring** legacy code as the client code & reusable object both need to be aware of object pool.
- You have to decide what happens when the pool is empty and there is a demand for an object - you can either wait for an object to become free or create a new object, and both options have issues. Waiting can have severe negative impact on performance, while creating new object needs additional work to maintain or trim the pool size.