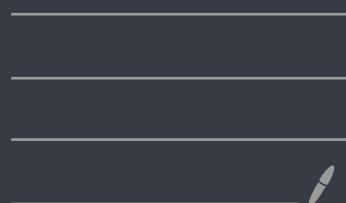


# Designs Patterns

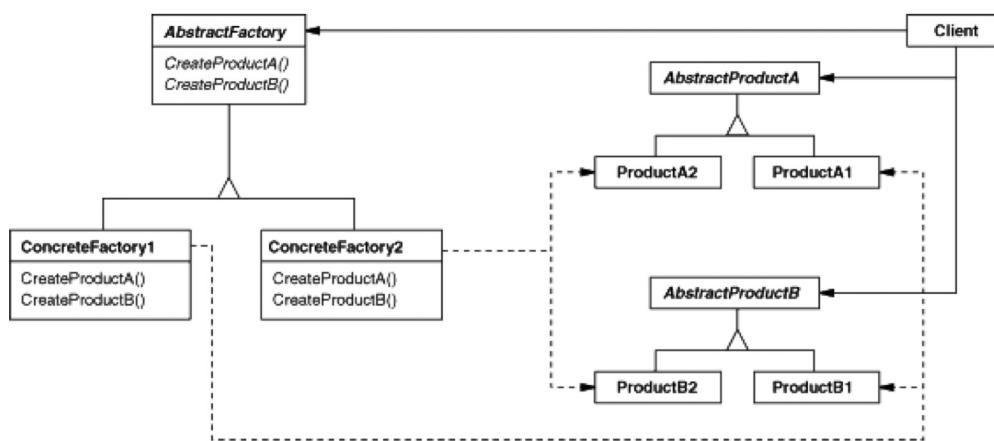


# Abstract Factory:

Used to separate the family building from the actual implementation of the program, making it transparent and remaining the same.

Usually the object implementation is at the start of the program

## Structure



## Possible use cases:

- Separate Two or more Families of UI Elements, depending on certain conditions.
- The paid and free version separation of an application.
- Distinguish between certain Format options

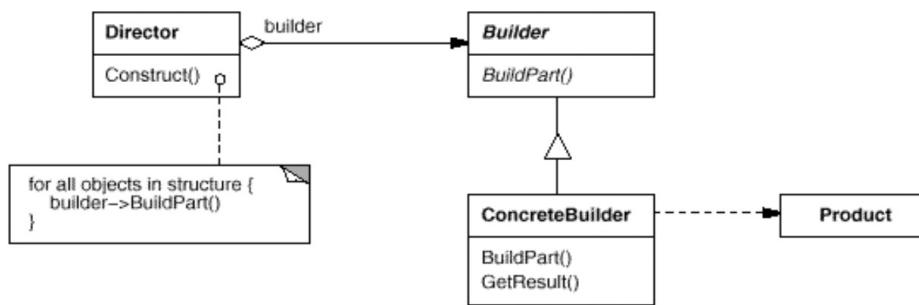
## Implementation:

[https://github.com/alexsotocx/design\\_patterns/tree/master/AbstractFactory](https://github.com/alexsotocx/design_patterns/tree/master/AbstractFactory)

# Builder Pattern:

Used to create complex objects in a structured way and produce a result. It separates the construction from its representation.

## Structure



112

If guarantees the construction of a valid and usable object.

Used to:

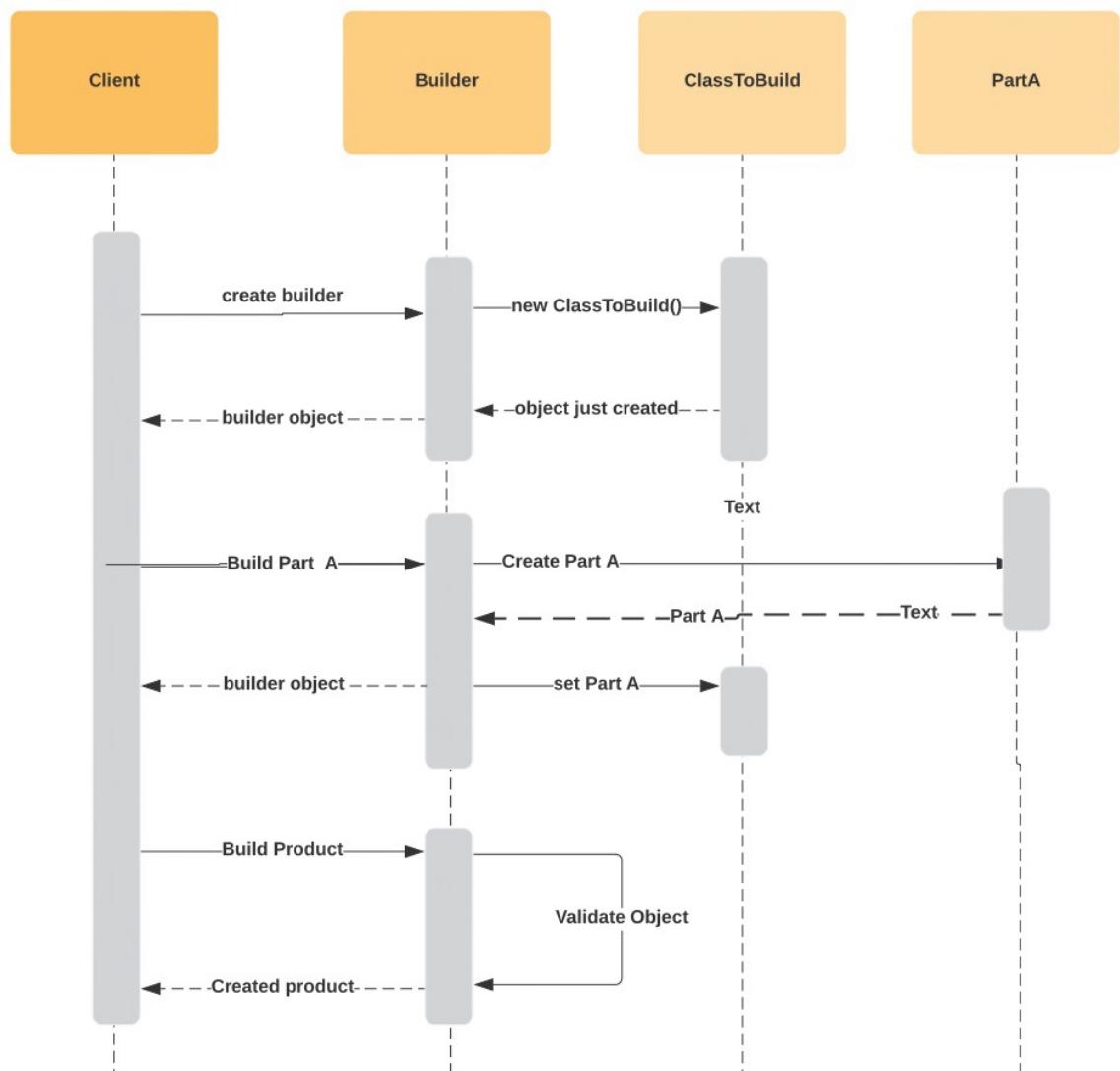
- Allow multiple representation from the same data using the same algorithm.
- Make the algorithm independent from the actual result creation.

## Possible usages:

- Signal conversions (Format conversion in general).
- Hide constructor of a class in a library

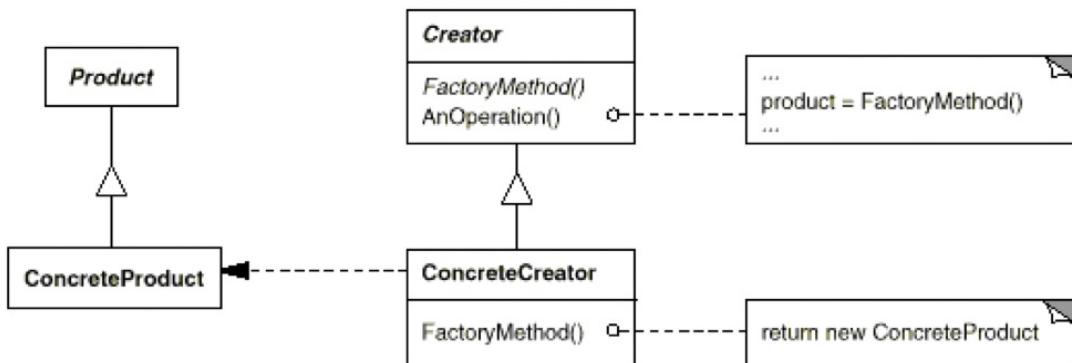
## Interactions:

The client is in charge of creating the builder, with it, the client will select dynamically or statically the parts (Components) to build, while the builder class will keep the context and it will build the target class to build.



## Factory pattern:

- Used to hide a complex object creation or dynamically create objects of certain type.
- A Factory function decides to create a new object, return an already created one or clone a certain prototype and return it.
- Encapsulates and provides a polymorphism so the code is not tied to something concrete.
- Structure :



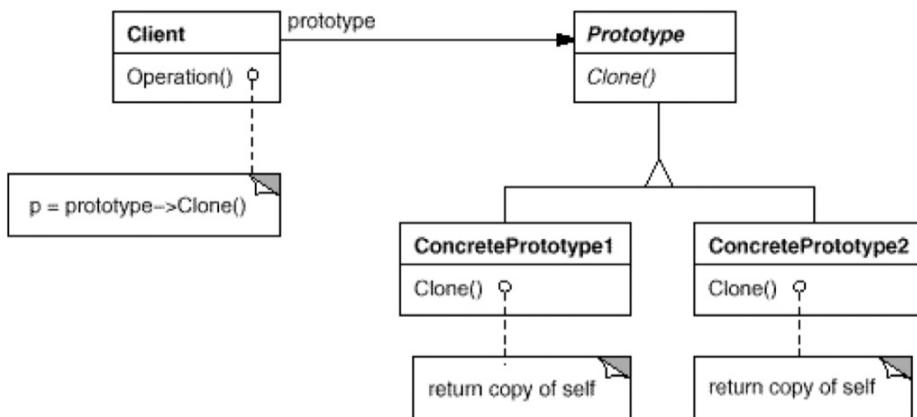
Use it when:

- Dynamic object creation. Makes the program decide which instances to create on realtime.
- Create singleton objects.

# Prototype Pattern

- \* Used to decouple the object creation from its representation
- \* Reduce the number of subclasses by cloning already predefined ones.
- \* Enable "dynamic" creation of "classes" (composition) for later reuse

## Structure



- The prototype works by cloning predefined classes, modifying its state, helping in that way the creation of new object with new behaviours.
- Can be used with the composition pattern, but the class needs to assure the deep clone.

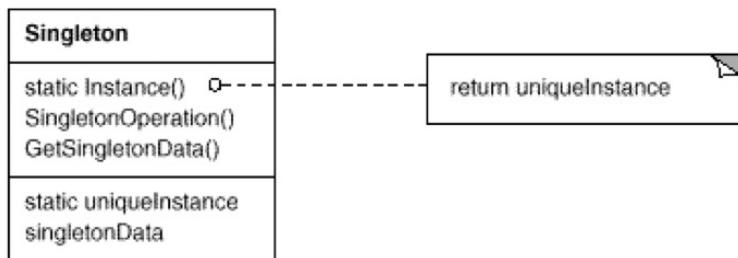
# Singleton pattern.

Used when an object of a specific class needs to exist just one time during the runtime.

The singleton pattern in itself, is an application of the Factory method, where the creator method checks if an object already exists or creates it and returning it.

- Allows subclassing,
- More usable than class static methods, because it can have state.
- "Don't" populate the global namespace.

## ▼ Structure

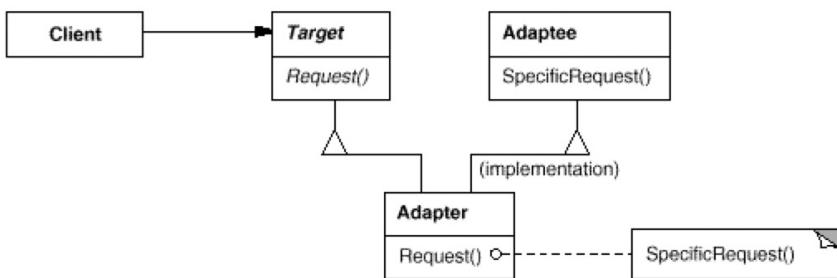


# Adapter Pattern (Wrapper)

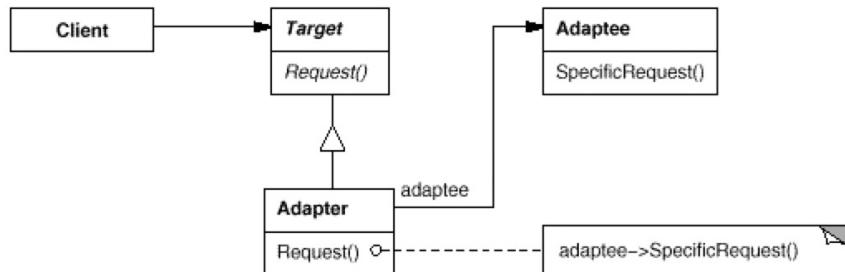
Used to convert the interface of a class into another interface needed by the client to work in another domain.

## Structure

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



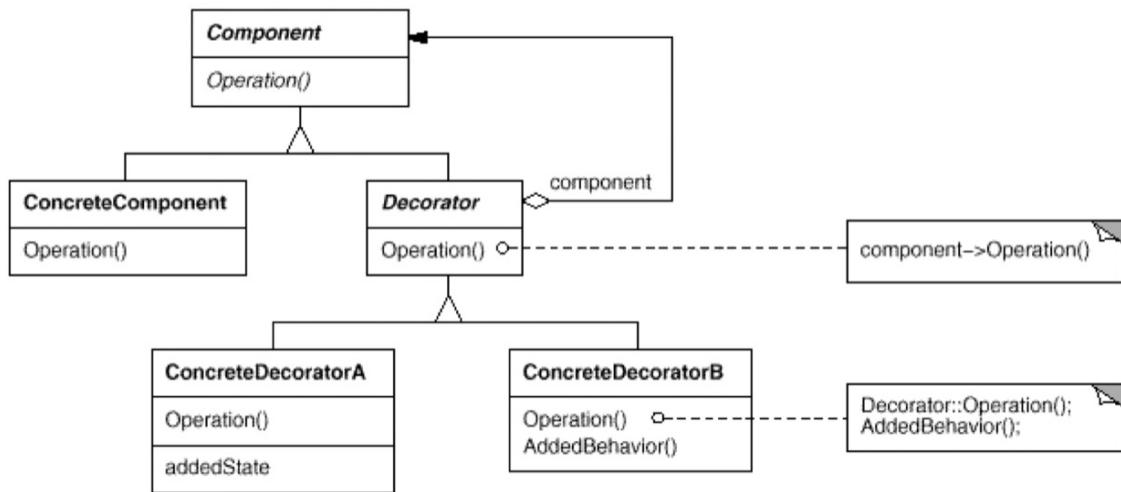
- Used when a class from another domain, package, library needs to be transformed into one our domain needs to work.  
→ (meet the needs)
- Isolate a dependency so it is easier to remove or replace.

# Decorator pattern

Used to attach new responsibilities to objects without the need of creating new classes to accomplish the same effect.

The decorator pattern uses inheritance or composition to first have the same interface the class it wants to decorate, and then extend the behaviour by adding method calls before or after the main call.

## Structure



## When to use:

- Add or remove responsibilities to a particular object.
- Class or superclass are light weight.
- Avoid big class tree.