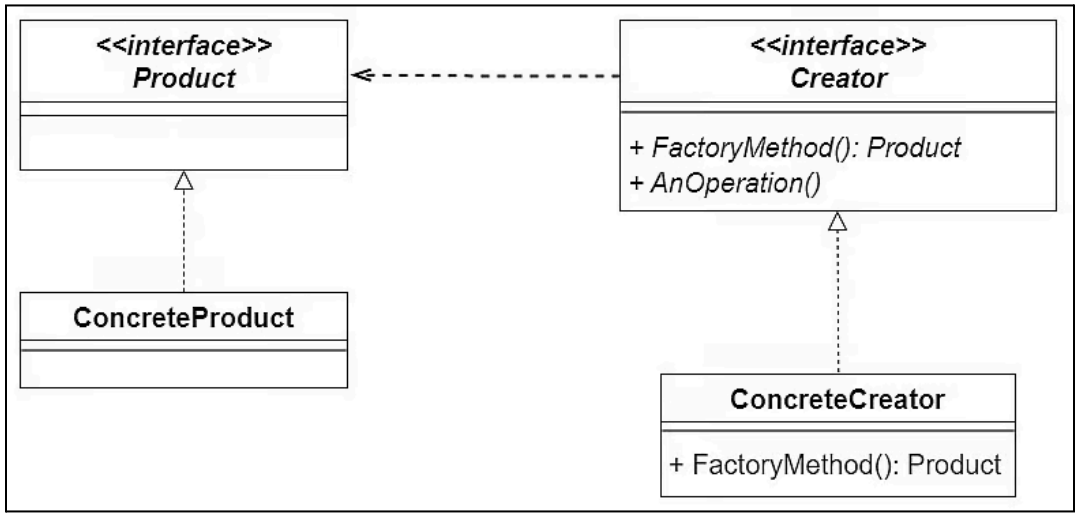
Factory

Purpose: Abstracting away details of objects we are creating
Motivation: Simplify creation of complex object with abstracting details



The solution: Defer instantiation process to subclasses

Extend the Base Factory Class + Provide creational logic for that type

Open-Closed: Open to Extension, Closed to Modification

Decoupling + Abstraction: We can provide different implementation of the creator and the client code won't need to change

---

Creator Interface or Abstract Class:
Abstract Class - if there are shared behaviors + attributes of the products
Interface Class - if other methods just need to be implemented

Concrete Creators:
There are subclasses that extend/implement the creator

Product Interface:
Contract of types of products the factory will produce

Limitations + Pitfalls:

1) Increased complexity: additional classes + interfaces
2) Harder to refactor

Closing Notes: 1) Encapsulate what varies, 2) Dependency Inversion Principle, 3) Open-Closed Principle

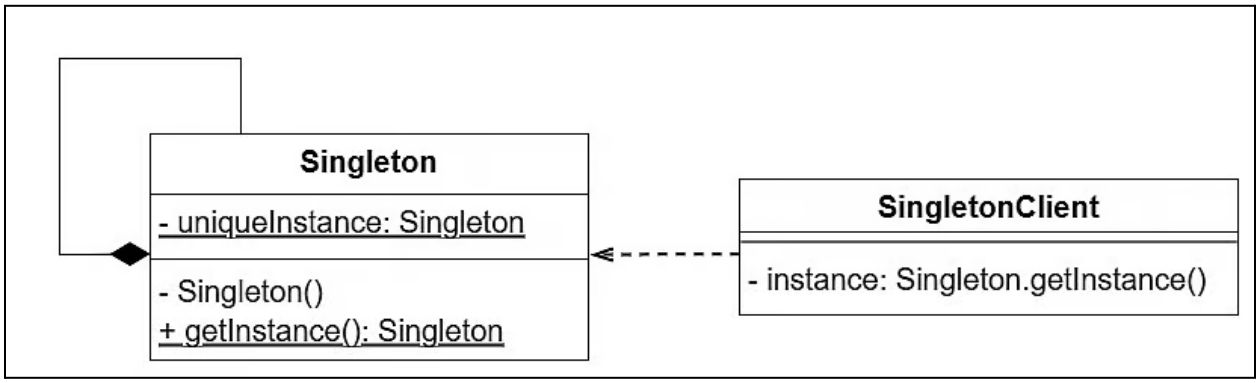
Singleton

Ensures a class has at most one instance and provides a global point of access to this instance.
Motivation: Access to one instance across the program

The Solution:
Initialize singleton to Null
Private Constructor prevents other instances from forming.
Get_Instance both sets and gets instance

Design and Implementation:

- Private static variable of its own type to hold the single instance
- Private Constructor which restricts public instantiation from the outside
- Public static method getInstance()
- (Optional) Ensure thread safety



Limitations and Pitfalls:

1. Violation of the single responsibility principle - manages its own instantiation
2. Difficulty in Unit Testing - testing may change the singleton's state, affecting other tests
3. Global Scope - difficult to track down where the state was changed b/c public access

Closing Notes:

1. Reduced memory usage
2. We are guaranteed one instance

Builder

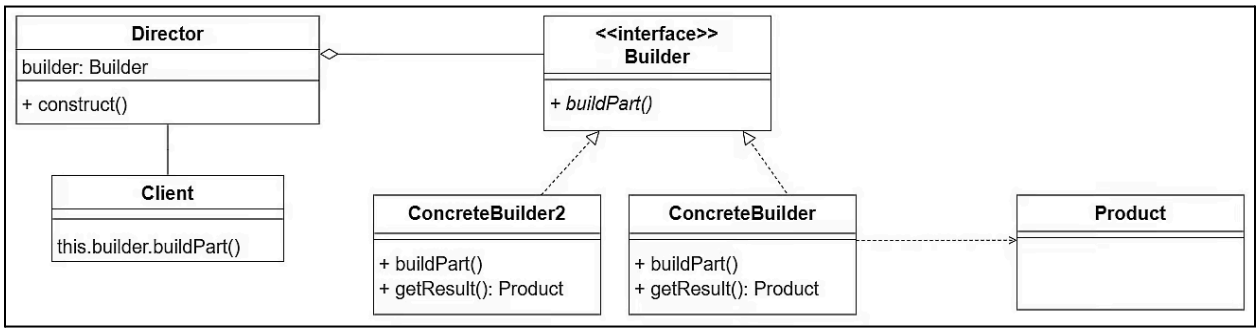
Separates the construction of a complex object from its representation.

Motivation: Rather than building an object all in one go, perhaps you get each new ingredient of that object in steps. You may want to perform 1 step to help build the object, then perform another step on that same object, then continue until it is completely finished.

Solution: The builder pattern is used to extract out the steps of constructing a product and delegate them to "Builders". This way, we can create multiple variations of a given product.

Design and Implementation:

1. **Product** - complex object (end product) that needs to be constructed
2. **Builder Interface** or Abstract Builder - declares the methods for constructing the product
3. **Concrete Builders** - implements the builder interface and provides specific implementations for each method
4. **Director** - manages correct sequence of the construction process, uses the builder interface to construct the product and shields the client from specifics of the product's construction.
5. **Client** - decides which type of product it needs and accordingly chooses the right concrete builder



Limitations + Pitfalls:

- **Complexity** - every product needs its own class and its own builder, code can grow quickly and be harder to manage.
- **Error-Prone** - allows for partial or step-wise construction so end product might miss some attributes, is in an inconsistent state, or has unexpected or erroneous behavior

Closing Notes:

1. **Encapsulate what varies** - hides internal representation and construction process of a product from the client (only valid products are constructed)
2. **Separation of Concerns** - pattern separates the construction from the representation, client does not need to know the intricate details of how object is put together
3. **Single Responsibility Principle** - every single one of the builders has one responsibility

Prototype

Duplicate an object with a hidden state, good for creating a complex object.

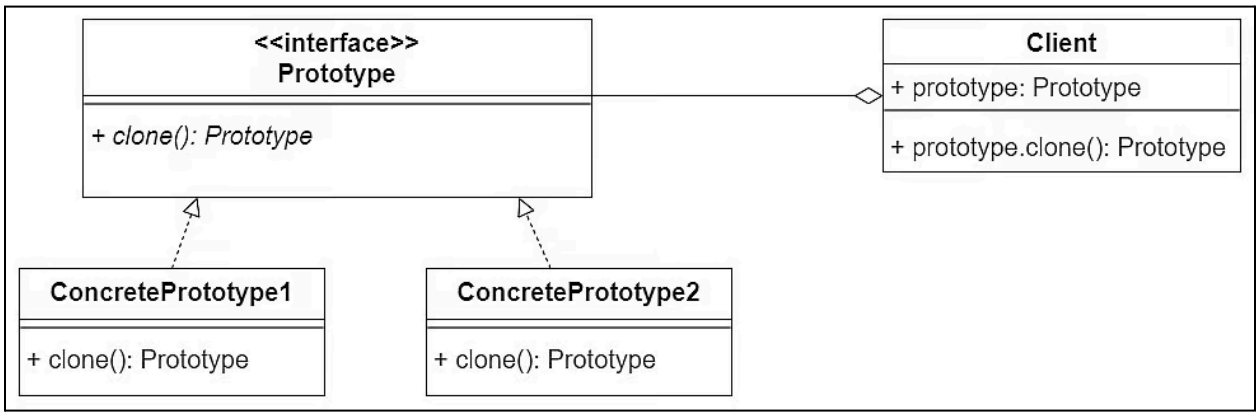
Motivation: don't want to recreate a complex object step by step (builder pattern too complex)

Solution: clients can simply call clone() to duplicate an object (less-error prone)

- Program to an Interface, not an implementation

Design + Implementation

1. Prototype (interface) - declares cloning method, concrete prototypes must implement this interface providing their own logic for clone(),
2. Concrete Prototype - implement the prototype interface and define clone logic
    - Each concrete prototype represents a different variation of the clone
3. Client - uses the Prototype interface to clone objects, holds a reference to a Prototype instance and uses it to create objects as needed.



Limitations and Pitfalls:

1. Complexity in performing deep copies
2. Limited Applicability

Closing Notes:

1. Program to an Interface, Not an Implementation
2. Efficient Creation of Complex Objects
3. Delegated Cloning Responsibility