COMP2511

Creational Patterns:

Factory Method
Abstract Factory Pattern
Builder Pattern
Singleton Pattern

Prepared by

Dr. Ashesh Mahidadia

Design Patterns

Creational Patterns

- Factory Method
- Abstract Factory
- Builder
- Singleton

Structural Patterns

- Adapter *discussed*
- Composite *discussed*
- Decorator *discussed*

Behavioral Patterns

- Iterator *discussed*
- Observer *discussed*
- State *discussed*
- Strategy *discussed*
- Template *discussed*
- Visitor

The lecture slides use material from the websites
https://refactoring.guru/design-patterns/
and the wikipedia pages.

Creational Patterns

Creational Patterns

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

- Factory Method
  - provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- Abstract Factory
  - let users produce families of related objects without specifying their concrete classes.
- Builder
  - let users construct complex objects step by step. The pattern allows users to produce different types and representations of an object using the same construction code.
- Singleton
  - Let users ensure that a class has only one instance, while providing a global access point to this instance.

Factory Method

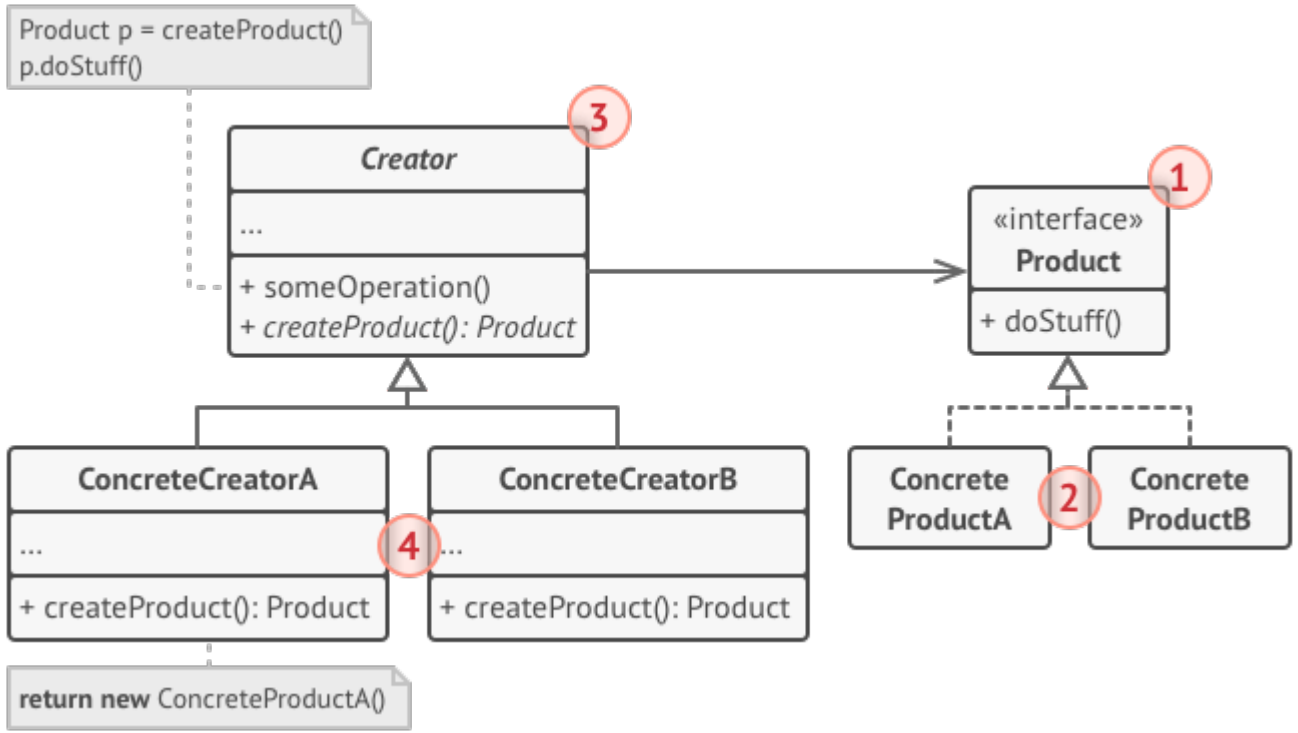
Factory Method

- Factory Method is a creational design pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

- **Problem:**
  - creating an object directly within the class that requires (uses) the object is inflexible
  - it commits the class to a particular object and
  - makes it impossible to change the instantiation independently from (without having to change) the class.

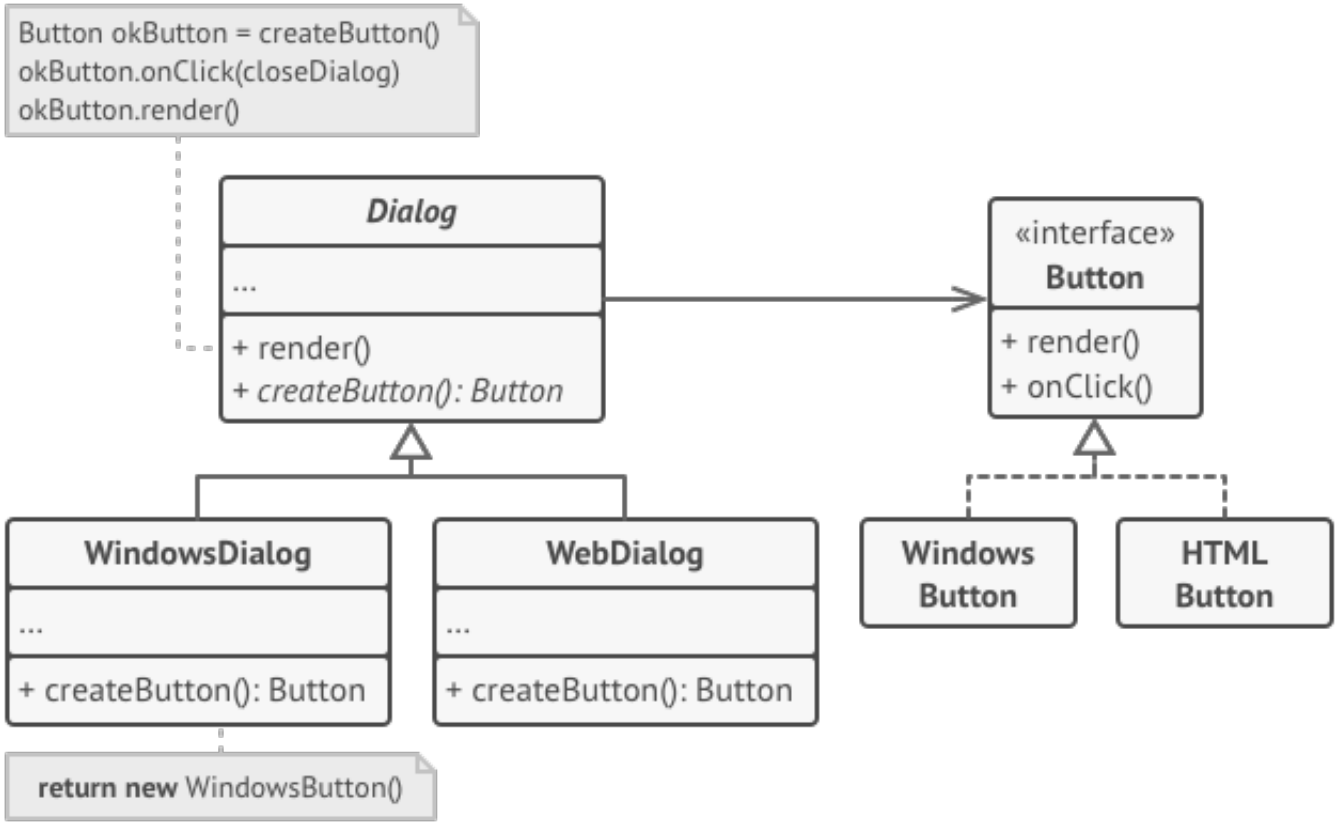
- **Possible Solution**:
  - Define a **separate** operation (factory **method**) for creating an object.
  - Create an object by calling a factory method.
  - This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate).

Factory Method : Structure



1. The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. Concrete Products are different implementations of the product interface.
3. The Creator class declares the factory method that returns new product objects.
4. Concrete Creators override the base factory method so it returns a different type of product.

Factory Method : Example



Example in **Java (MUST read)**:

https://refactoring.guru/design-patterns/factory-method/java/example

Factory Method

For more, read the following:

https://refactoring.guru/design-patterns/factory-method

Abstract Factory Pattern

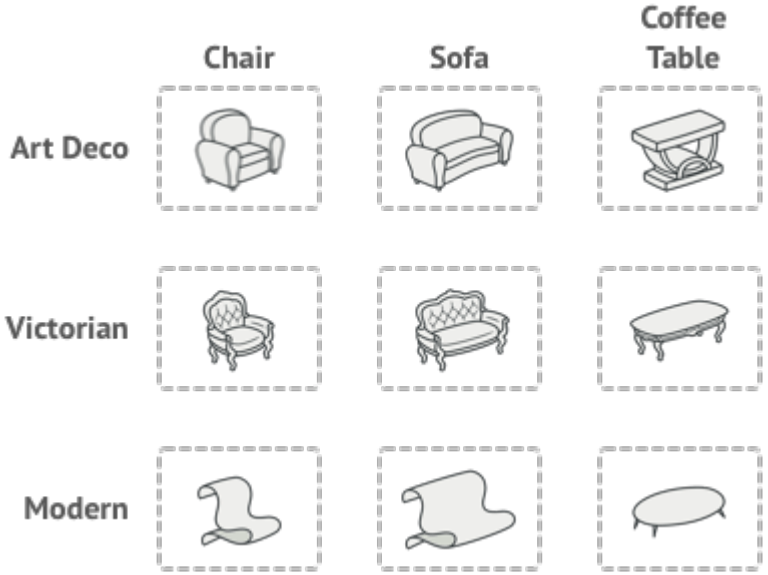
Abstract Factory Pattern

Intent: Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problem:

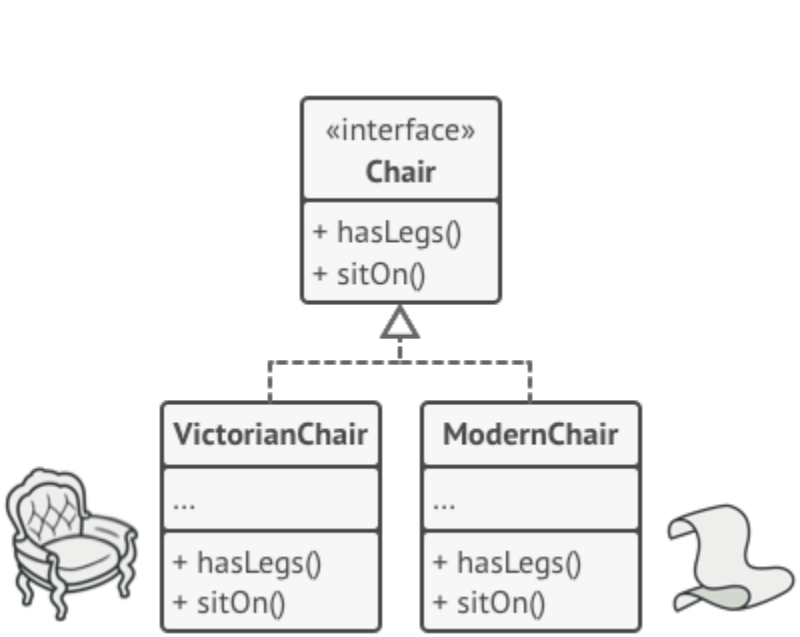
Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

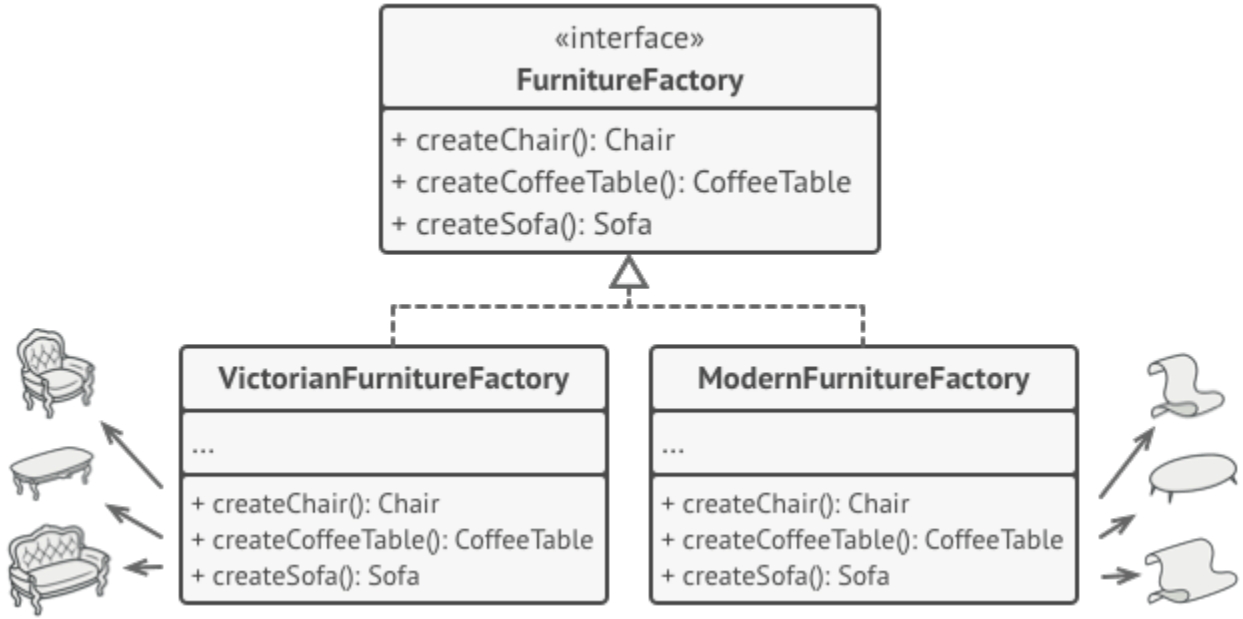
- A family of related products, say: Chair + Sofa + CoffeeTable.
- Several variants of this family.
- For example, products Chair + Sofa + CoffeeTable are available in these variants:



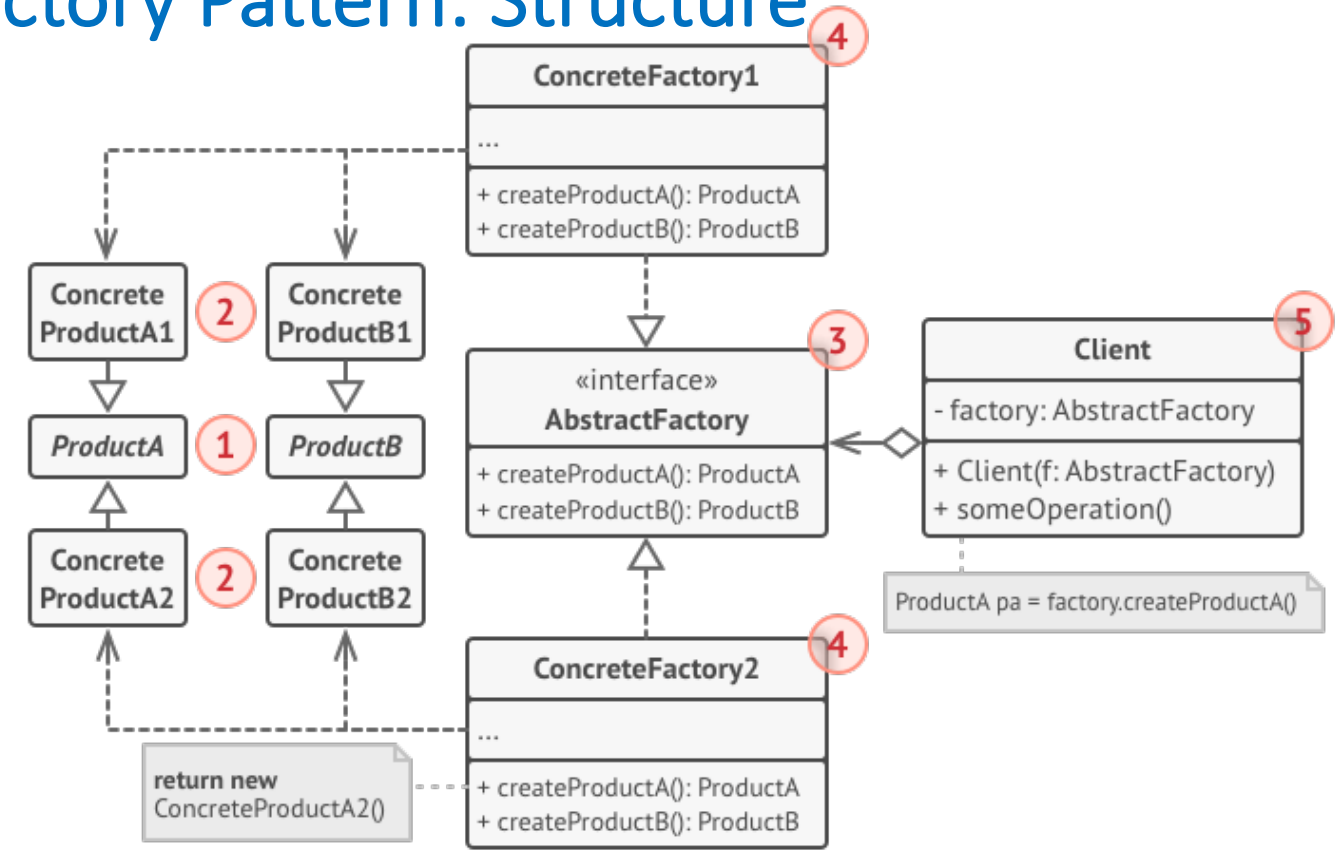
Abstract Factory Pattern:

Possible **Solution**:



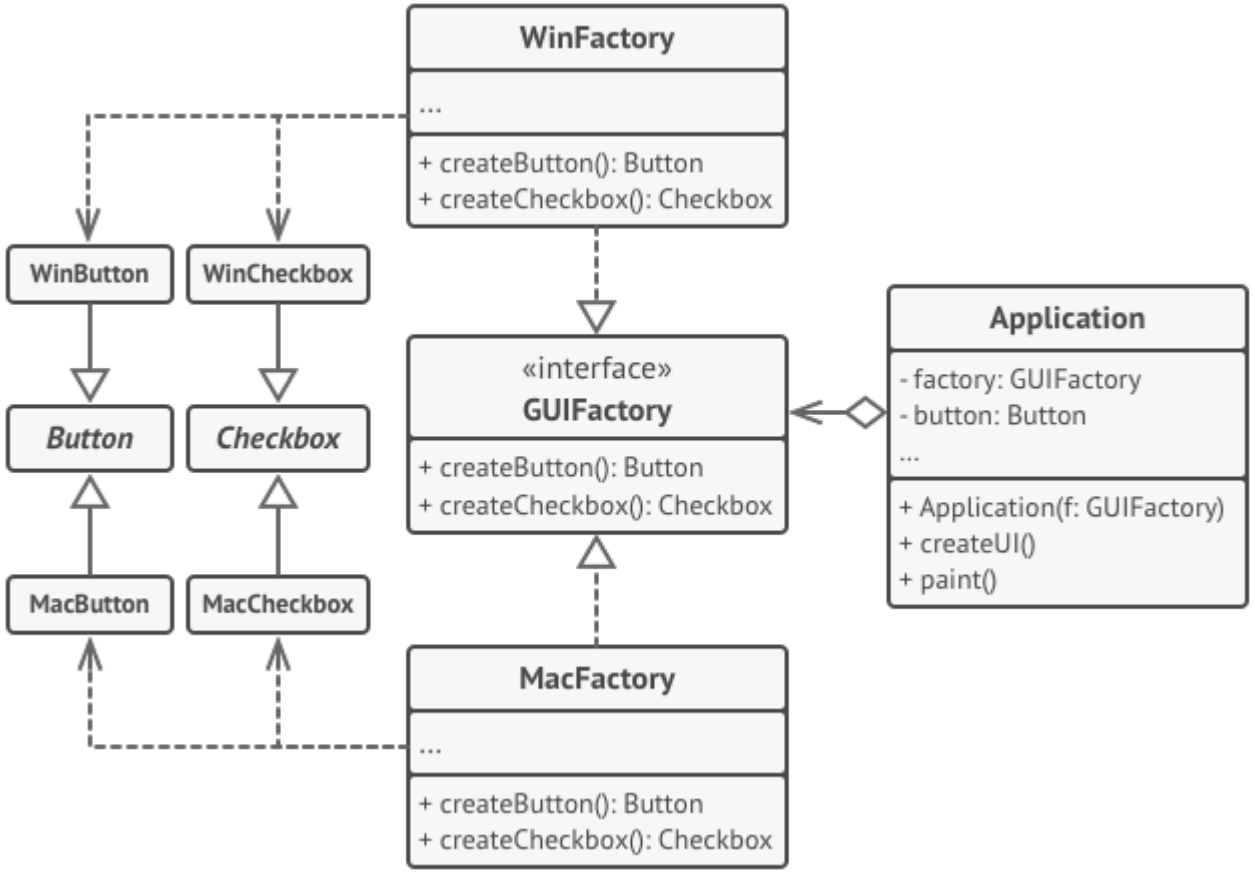


Abstract Factory Pattern: Structure



1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.
5. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

Abstract Factory Pattern: Example



Example in **Java (MUST read)**:

https://refactoring.guru/design-patterns/abstract-factory/java/example

Abstract Factory Pattern

For more, read the following:

https://refactoring.guru/design-patterns/abstract-factory

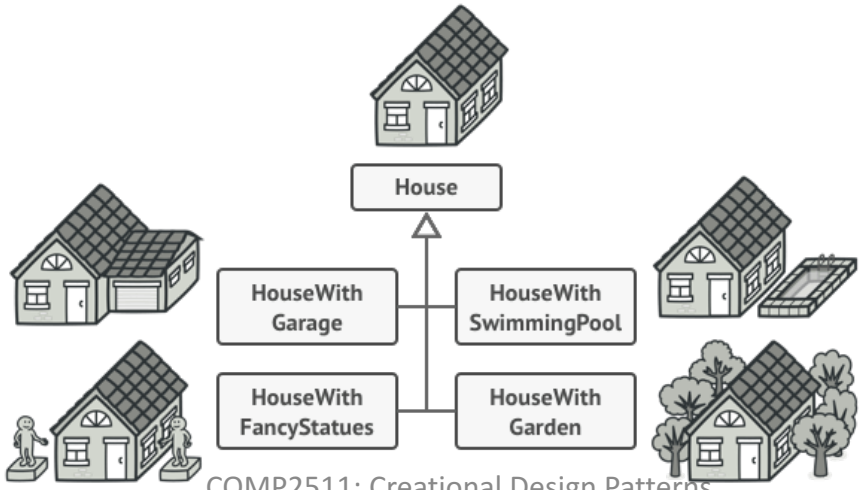
Builder Pattern

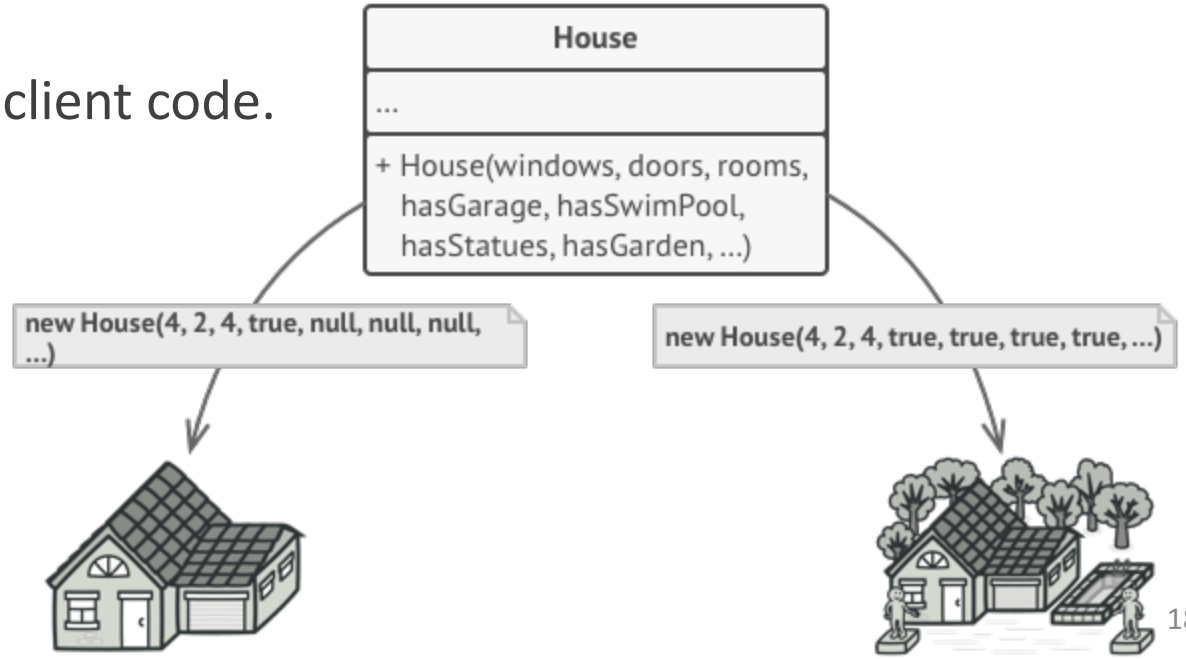
Builder Pattern

Intent: Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem:

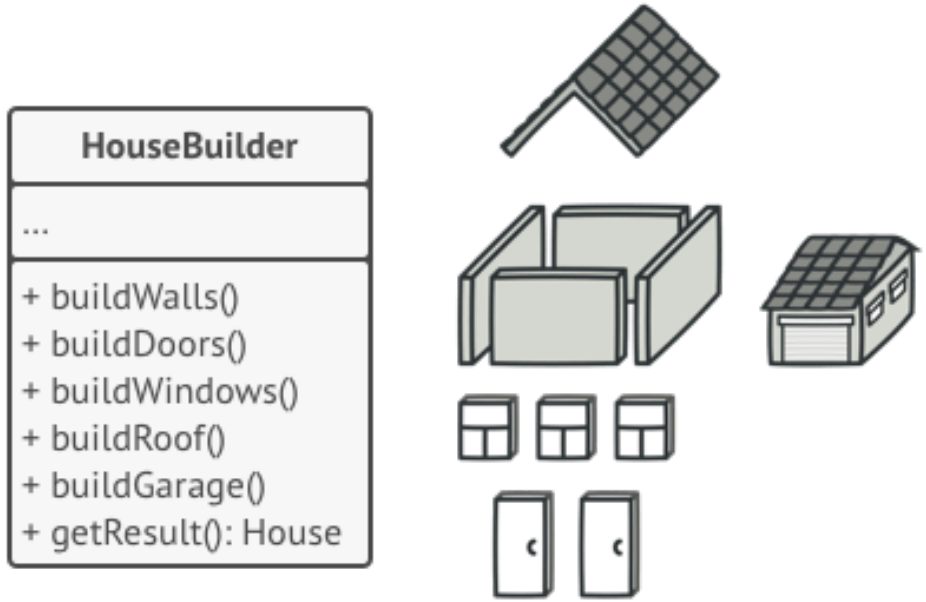
- Imagine a complex object that requires laborious, step-by-step initialization/construction of many fields and nested objects.
- Such initialization/construction code is usually buried inside a monstrous constructor with lots of parameters.
- Or even worse: scattered all over the client code.





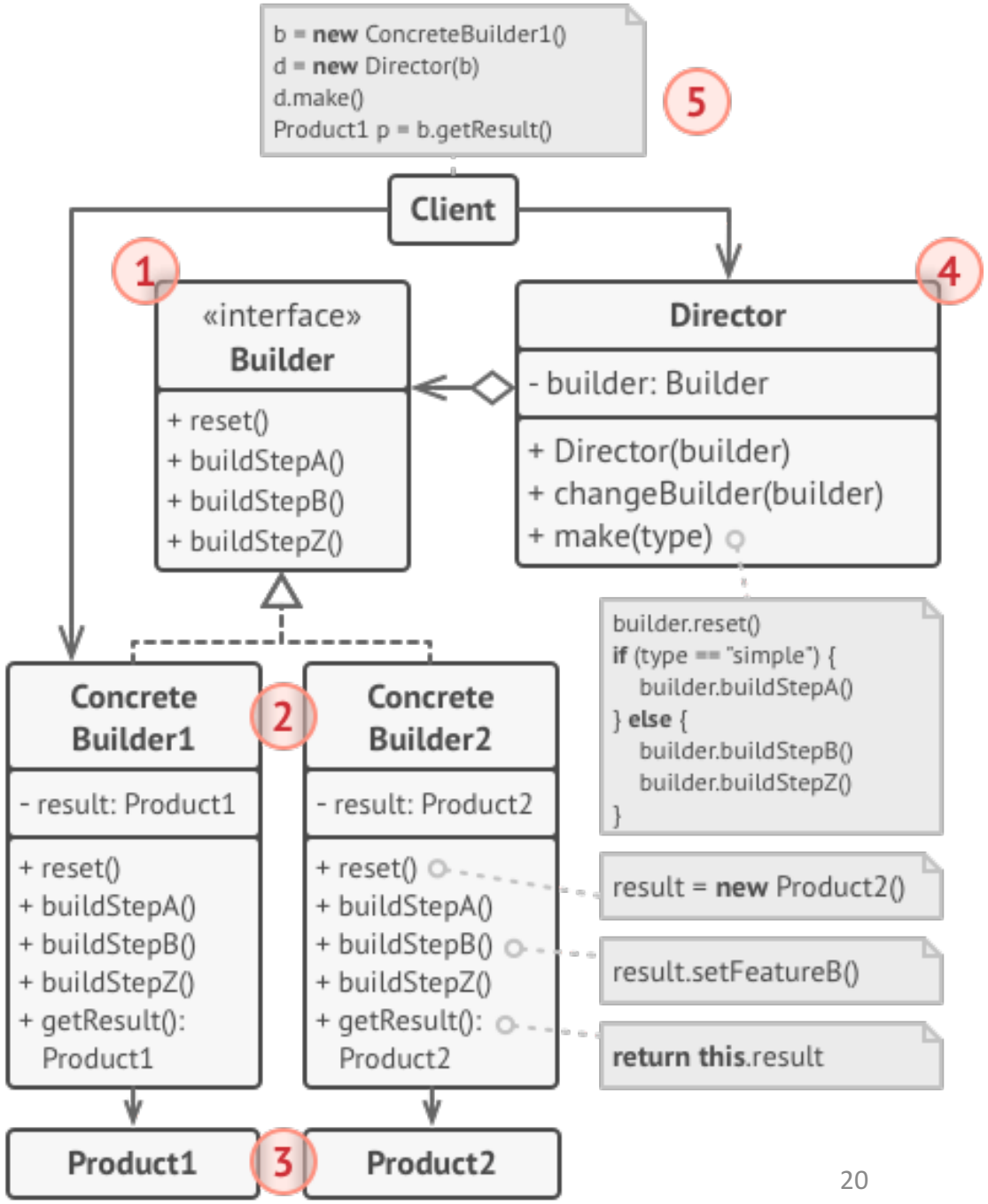
Builder Pattern

- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called **builders**.
- The Builder pattern lets you construct complex objects step by step.
- The Builder doesn't allow other objects to access the product while it's being built.
- **Director**: The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



Builder Pattern: Structure

- The **Builder** interface declares product construction steps that are common to all types of builders.
- **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
- **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
- The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- The **Client** must associate one of the builder objects with the director.



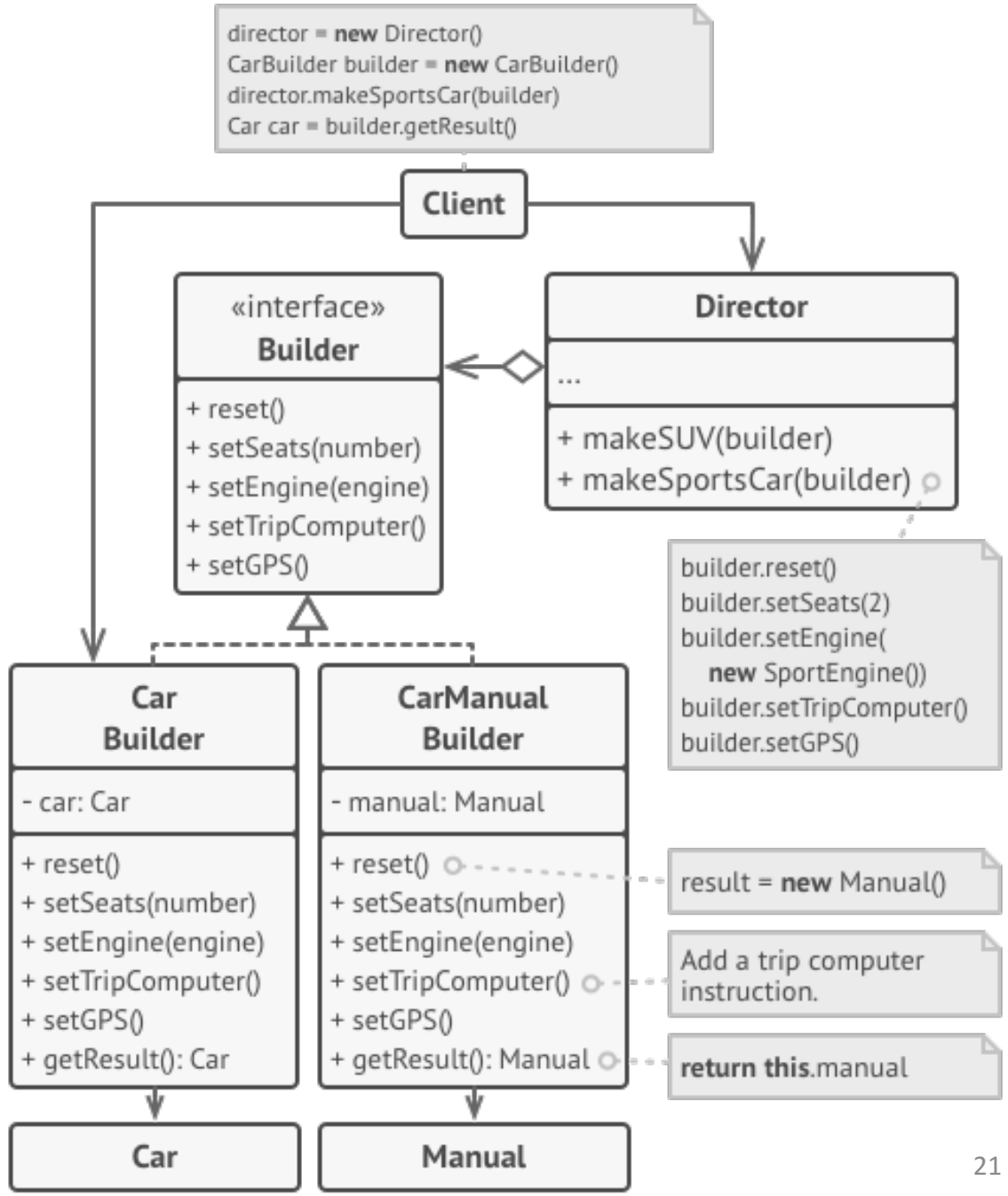
Builder Pattern: Example

This example illustrates how you can reuse the same object construction code when,

- building different types of cars, and
- creating the corresponding manuals for them.

Example in **Java (MUST read)**:

https://refactoring.guru/design-patterns/builder/java/example



Relations with Other Patterns

- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, or Builder (more flexible, but more complicated).
- Builder focuses on constructing complex objects step by step.
- Abstract Factory specializes in creating families of related objects.
- Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.

Builder Pattern

For more information, read:

https://refactoring.guru/design-patterns/builder

Singleton Pattern

Singleton Pattern

Intent: Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem: A client wants to,

- ensure that a class has just a single instance, and
- provide a global access point to that instance

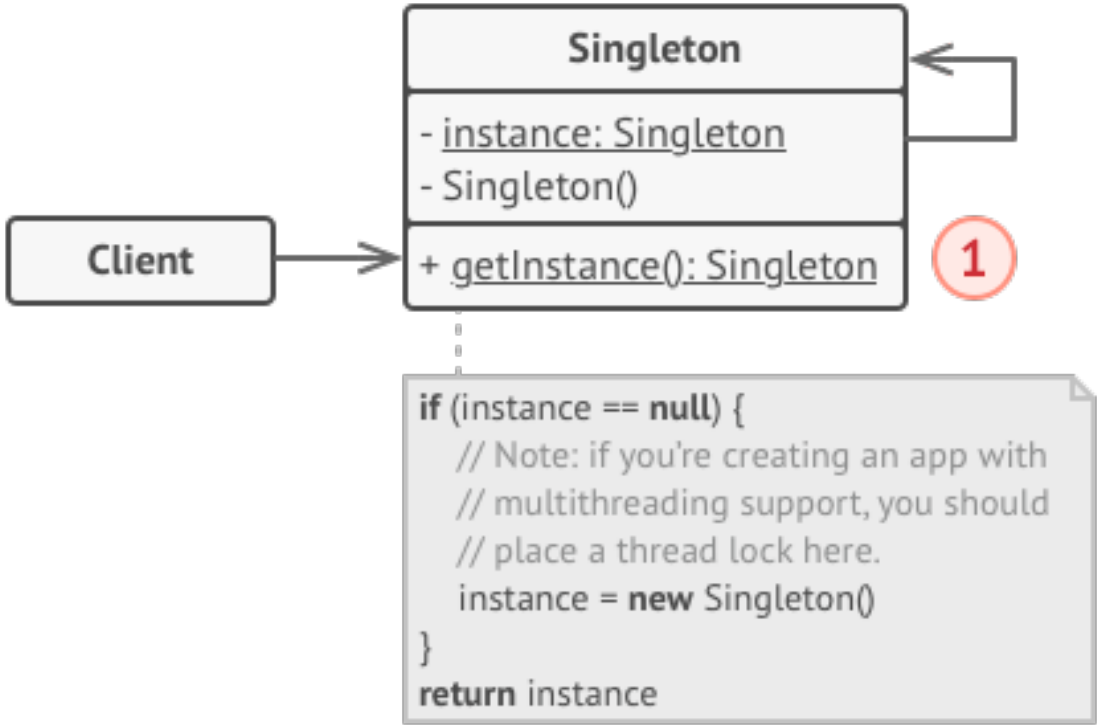
Solution:

All implementations of the Singleton have these two steps in common:

- Make the default constructor **private**, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.
- If your code has access to the Singleton class, then it's able to call the Singleton's static method.
- Whenever Singleton's static method is called, the same object is always returned.

Singleton: Structure

- The Singleton class declares the static method *getInstance* (1) that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code.
- Calling the *getInstance* (1) method should be the only way of getting the Singleton object.



Singleton: How to Implement

- Add a private static field to the class for storing the singleton instance.
- Declare a public static creation method for getting the singleton instance.
- Implement “lazy initialization” inside the static method.
  - It should create a new object on its first call and put it into the static field.
  - The method should always return that instance on all subsequent calls.
- Make the constructor of the class private.
  - The static method of the class will still be able to call the constructor, but not the other objects.
- In a client, call singleton’s static creation method to access the object.

Example in **Java (MUST read)**:

https://refactoring.guru/design-patterns/singleton/java/example

Singleton Pattern

For more information, read:

https://refactoring.guru/design-patterns/singleton

End