Design Pattern

Dominik Panzarella

September 1, 2023

Contents

Chapter 1

Creational Pattern

Introduction

Creational design patterns abstract the **instantiation** proess. *They help make a system indipendent of how its objects are created, composed and represented.*

There are two recurring themes in these patterns. First, the all encapsulate knowledge about which concrete classes the system uses. Second, the hide how instances of these classes are created and put together. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be **static** (that is, specified at compile-time) or dynamic (at run-time).

Creational patterns are divided into:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Because the creational patterns are closely related, we'll study all five of them to highlight their similarities and differences, using a common example - building a maze for a computer game - to illustrate their implementations We'll ignore many details of what can be in a maze and whether a maze game has a single or multiple players. Instead, we'll just focus in how mazes get created

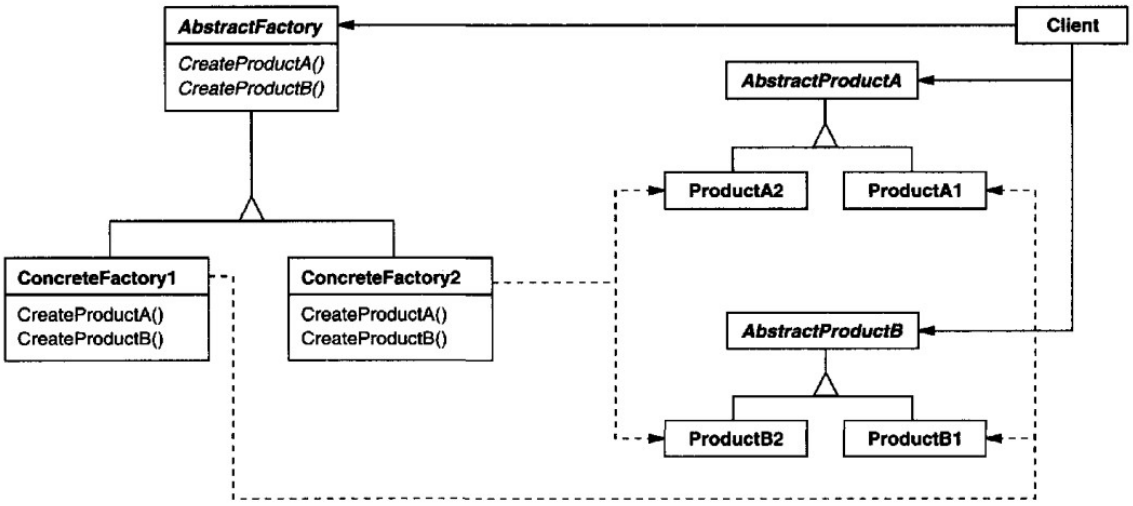

Figure 1.1: Structure of the Abstract Factory Pattern

Abstract Factory

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also Know As

Kit

Applicability

Use the abstract factory when:

- a system should be indipendent of how its products are creates, composed, and represented.
- a system should be configured with one of multiple families of products.
- a famility of related product objects is designed to be used together, and you need to enforce this constraint,
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Structure

See **Figure 1.1**

Partecipants

- AbstractFactory
  - declares an interface for operations that create abstract product.
- ConcreteFactory
  - implements the operations to create concrete product objects.

- AbstractProduct
  - declares an interface for a type of product object.
- ConcreteProduct
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- Client
  - uses only interface declared by AbstractFactory and AbstractProduct classes.

Consequences

Benefits and liabilites of the Abstract Factory pattern:

- It isolates concrete classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new kinds of products is difficult.

Implementation

Useful techniques for implementing the AbstractFactory pattern:

- Factories as singletons.
- Creating the products: define a factory method for each product. If many product families are possibile, the concrete factory can be implemented using the Prototype pattern
- Defining extensible factories: add a parameter to operations that create objects.

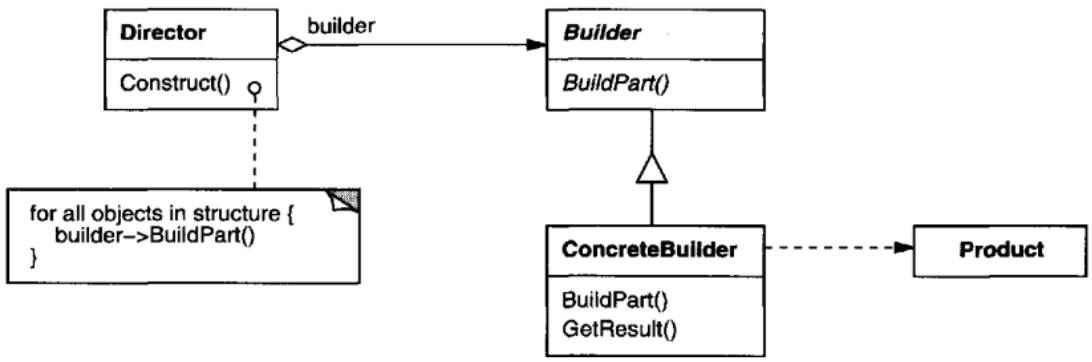

Figure 1.2: Structure of the Builder Pattern

Builder

Intent

Separate the construction of complex object from its representation so that the same construction process can create different representations.

Applicability

Use the Builder pattern when

- the algorithm for creating a complex object should be indipendent of the parts that make up the object and how they are assembled.
- the construction process must allows different representation for the object that is constructed.

Structure

See **Figure 1.2**

Partecipants

- Builder
  - specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keep track of the representation it creates.
  - provides an interface for retrieving the product
- Director
  - constructs an object using the Builder interface.
- Product
  - represents the complex object under construction. ConcreteBuilder build the product's internal representation and defines the process by which it is assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result-

Consequences

Key consequences:

1. It lets you vary a product's internal representation.
2. It isolates code for construction and representation.
3. It gives you finer control over the construction process.

Implementation

Typically, there is an AbstractFactory class that defines an operation for each component that a director may ask it to create - they do nothing by default. A ConcreteBuilder class overrides operations for components it is interested in creating.
Other implementation issue to consider:

1. Assembly and construction interface
2. Why no abstract class for products? In the common case, the products produced by the concrete builders differ so greatly that there is a little or null gain from using a common parent class.
3. Empty methods as default in Builder.

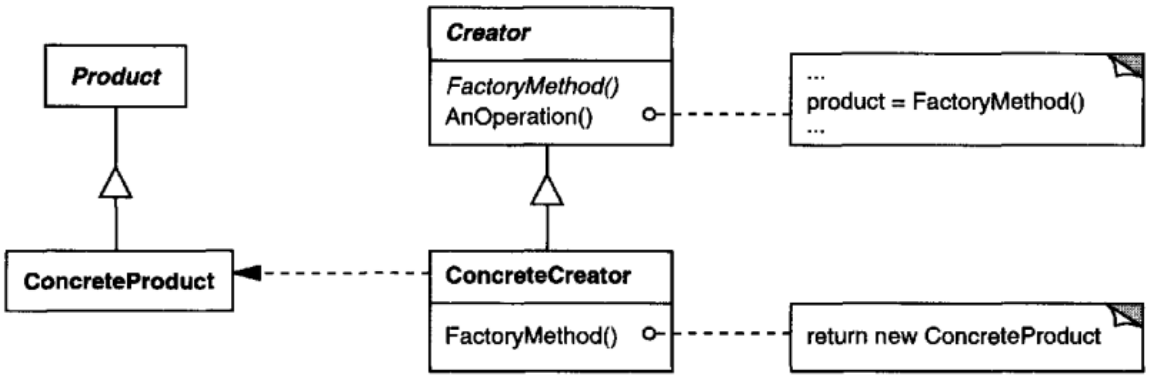

Figure 1.3: Strucre of the Factory Method Pattern.

Factory Method

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Know As

Virtual Constructor

Applicability

Use the Factory Method Pattern when:

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsability to one of several helper subclasses, and you want to localize the knoledge of which helper subclass is the delegate.

Structure

See **Figure 1.3**.

Partecipants

- Product
  - defines the interface of objects the factory method creates.
- ConcreteFactory
  - implements the Product interface.
- Creator
  - declares the factory method, which returns an object of type Prodcut. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- ConcreteCreator
  - overrides the factory method to return an instance of a ConcreteProduct.

Consequences

A potential disadvantage: client may have to subclass the Creator class just to create a particular ConcreteProduct object.
Two additional consequences of the Factory Method pattern:

1. Provides hooks for subclasses.
2. Connects parallel class hierarchies.

Implementations

Consider the following issue:

1. Two major varieties
   (a) the Creator class is an abstract class and do not provide an implementation for the factory method it declares (it is also possible to have an abstract class that define a default implementation).
   (b) the Creator is a concrete class and provides a default implementation for the factory method .
2. Parameterized factory methods.
3. Language-specific variants and issue.
4. Using templates to avoid subclassing.
5. Naming conventions.

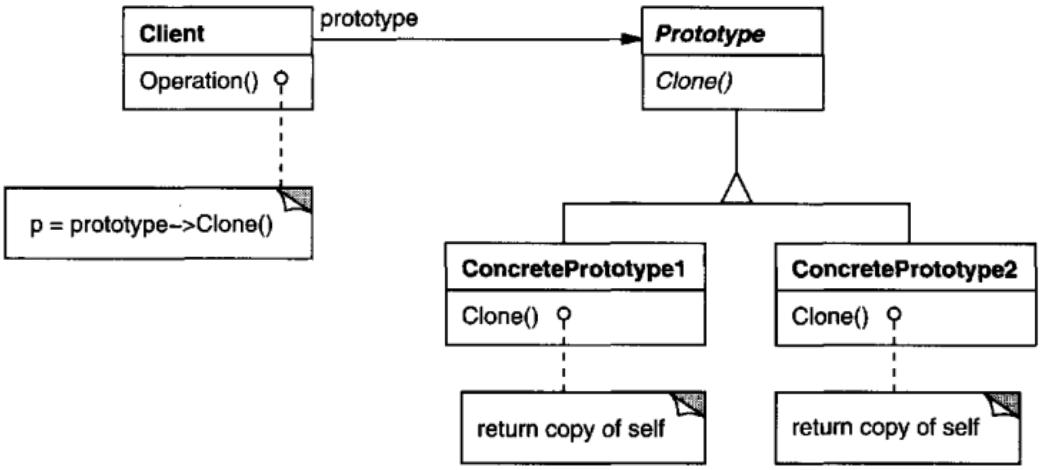

Figure 1.4: Structure of the Prototype Pattern.

Prototype

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

Use the Prototype pattern when a system should be indipendent of how its products are create, composed, and represented; **and**:

- when the classes to instantiate are specified at run-time; *or*
- to avoid building a class hierarchy of factories that parallels the class hierarchy of product; *or*
- when instances of a class can one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Structure

See **Figure 1.4**.

Partecipants

- Prototype
  - declares an interface for cloning itself.
- ConcretePrototype
  - implements an operation for cloning itself.
- Client
  - creates a new object by asking a prototype to clone itself.

Consequences

Prototype has as many of the same consequences that Abstract Factory and Builder have: it hides the concrete product classes from the client. Here additional benefits:

1. Adding and removing products at run-time.
2. Specifying new objects by varying values.
3. Specifying new objects by varying structure.
4. Reduces subclassing.
5. Configuring an application with classes dynamically

Implementations

Consider the following issues when implementing prototypes:

1. Using a prototype manager.
2. Implementing the Clone operation.
3. Initializing clones

Singleton

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Applicability

Use the Singleton pattern when:

- there must be exactly one instance of a class, and it must be accessible to client from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure

See **Figure 1.5**.

Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - may be responsible for creating its own unique instance.

Consequences

The Singleton pattern has several benefits:

- Controlled access to sole instance.
- Reduced name space.
- Permits refinement of operations and representation.
- Permits a variable number of instance.
- More flexible than class operations.

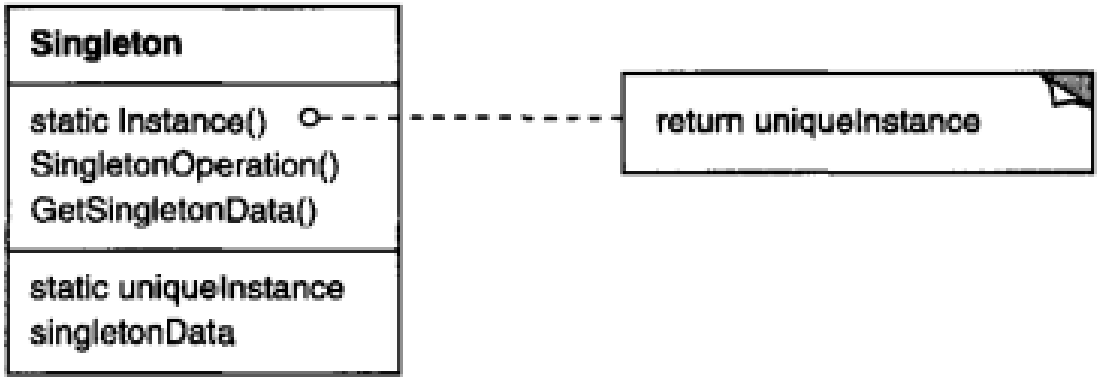

Figure 1.5: Structure of the Singleton Pattern.

Implementation

Here are implementation issues to consider when using the Singleton pattern:

- Ensuring a unique instance.
- Subclassing the Singleton class. The main issue is not defining the subclass but installing its unique instance so that clients will be able to use it. In essence, **the variable that refers to the singleton instance must be initialized with an instance of the subclass**. There are three different implementations:

  1. Determine which singleton you want to use in the Singleton's Instance operation.
  2. Take the implementation of Instance out of the parent class and put it in the subclass.
  3. Uses a **registry of singleton**. the Singleton classes can register their singleton instance by name in a well-known registry;the registry maps between string names and singleton. This approach frees Instance operation from knowing all possible Singleton class or instances, all it requires is a common interface for all Singleton classes.