## ‎‎Patrones de programación‎‎[

‎"Comprender es percibir patrones e implementarlos en el Diseño" ‎

‎Los patrones de diseño de software son abstracciones que ayudan a estructurar los diseños de sistemas. Si bien no es nuevo, dado que el concepto ya fue descrito por ‎[‎Christopher Alexander‎](https://en.wikipedia.org/wiki/Christopher_Alexander)‎ en sus teorías arquitectónicas, solo cobró algo de tracción en la programación debido a la publicación del libro ‎[‎Design Patterns: Elements of Reusable Object-Oriented Software‎](https://en.wikipedia.org/wiki/Design_Patterns)‎ en octubre de 1994 por ‎[‎Erich Gamma‎](https://en.wikipedia.org/wiki/Erich_Gamma)‎, ‎[‎Richard Helm‎](https://en.wikipedia.org/wiki/Richard_Helm)‎, ‎[‎Ralph Johnson‎](https://en.wikipedia.org/wiki/Ralph_Johnson)‎ y ‎[‎John Vlissides‎](https://en.wikipedia.org/wiki/John_Vlissides)‎, conocido como la ‎[**‎Banda de los Cuatro (GoF)‎**](https://en.wikipedia.org/wiki/Design_Patterns)‎ , que identifica y describe 23 patrones de diseño de software clásicos. ‎

‎Un patrón de diseño no es una solución estática, ni es un algoritmo. Un ‎**‎patrón‎**‎ es una forma de describir y abordar por su nombre una solución o enfoque repetible para un problema de diseño común, es decir, una forma común de resolver un problema genérico (qué tan genérico o específico es el patrón depende de cuán restringido esté el objetivo objetivo). Los patrones pueden surgir por sí solos o por diseño. Esta es la razón por la cual los patrones de diseño son útiles como una abstracción sobre la implementación y una ayuda en la etapa de diseño. Con este concepto, se da una forma más fácil de facilitar la comunicación sobre una elección de diseño como técnica de normalización para que cada persona pueda compartir el concepto de diseño. ‎

‎Dependiendo del problema de diseño que aborden, los patrones de diseño se pueden clasificar en diferentes categorías, de las cuales las categorías principales son: ‎

* [‎Patrones de creación‎](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns/Creational_Patterns)
* [‎Patrones estructurales‎](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns/Structural_Patterns)
* [‎Patrones de comportamiento‎](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns/Behavioral_Patterns)‎.‎

‎Los patrones se encuentran comúnmente en lenguajes de programación orientados a objetos como C ++ o Java. Se pueden ver como una plantilla sobre cómo resolver un problema que ocurre en muchas situaciones o aplicaciones diferentes. No es reutilización de código, ya que generalmente no especifica código, pero el código se puede crear fácilmente a partir de un patrón de diseño. Los patrones de diseño orientados a objetos suelen mostrar relaciones e interacciones entre clases u objetos sin especificar las clases de aplicación finales o los objetos que están involucrados. ‎

‎Cada patrón de diseño consta de las siguientes partes: ‎

**‎Problema/requisito‎**

‎Para usar un patrón de diseño, necesitamos pasar por un mini diseño de análisis que puede estar codificado para probar la solución. Esta sección establece los requisitos del problema que queremos resolver. Este suele ser un problema común que ocurrirá en más de una aplicación.‎

**‎Fuerzas‎**

‎Esta sección establece los límites tecnológicos, que ayudan y guían la creación de la solución.‎

**‎Solución‎**

‎En esta sección se describe cómo escribir el código para resolver el problema anterior. Esta es la parte de diseño del patrón de diseño. Puede contener diagramas de clases, diagramas de secuencia y/o lo que sea necesario para describir cómo codificar la solución.‎

‎Los patrones de diseño pueden considerarse como una estandarización de las mejores prácticas comúnmente acordadas para resolver problemas de diseño específicos. Uno debe entenderlos como una forma de implementar buenos patrones de diseño dentro de las aplicaciones. Hacerlo reducirá el uso de soluciones ineficientes y oscuras. El uso de patrones de diseño acelera su diseño y ayuda a comunicarlo a otros programadores.‎

Patrones de creación‎‎

‎En ingeniería de software, ‎**‎los patrones de diseño creacional son patrones‎**‎ de diseño que se ocupan de los mecanismos de creación de objetos, tratando de crear objetos de una manera adecuada a la situación. La forma básica de creación de objetos podría dar lugar a problemas de diseño o agregar complejidad al diseño. Los patrones de diseño creacional resuelven este problema controlando de alguna manera esta creación de objetos. ‎

‎En esta sección asumimos que el lector tiene suficiente familiaridad con las funciones, las variables globales, la pila frente al montón, las clases, los punteros y las funciones de miembro estático como se introdujo anteriormente. ‎

‎Como veremos hay varios patrones de diseño creacional, y todos se ocuparán de una tarea de implementación específica, que creará un mayor nivel de abstracción a la base de código, ahora cubriremos cada uno. ‎

**‎Constructor‎**‎[‎ **(Builder)**

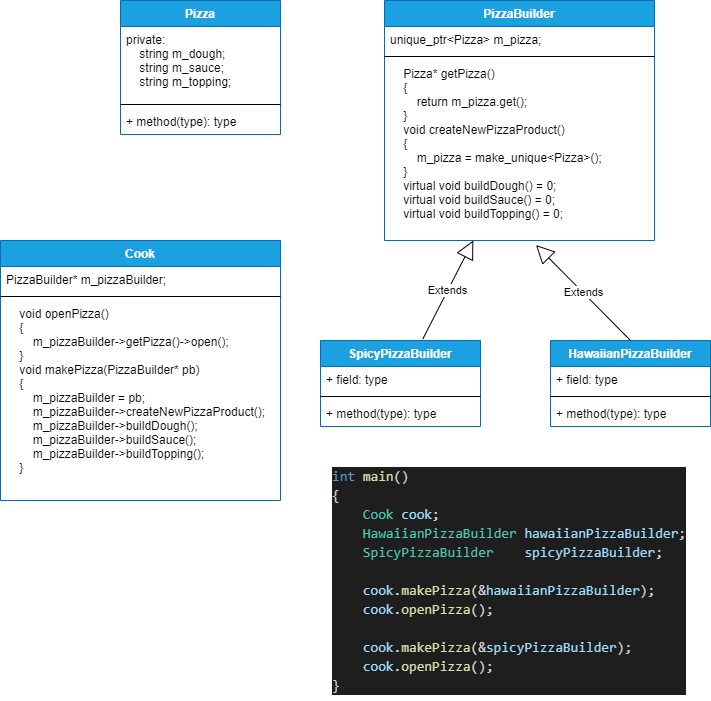
‎El Patrón de Creación del Constructor se utiliza para separar la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones de objetos. ‎

**‎Problema‎**

‎Queremos construir un objeto complejo, sin embargo, no queremos tener un miembro constructor complejo o uno que necesitaría muchos argumentos.‎

**‎Solución‎**

‎Defina un objeto intermedio cuyas funciones miembro definan el objeto deseado parte por parte antes de que el objeto esté disponible para el cliente. Builder Pattern nos permite aplazar la construcción del objeto hasta que se hayan especificado todas las opciones de creación.‎



#include <string>

#include <iostream>

#include <memory>

using namespace std;

// "Product"

class Pizza

{

public:

    void setDough(const string& dough){

        m\_dough = dough;

    }

    void setSauce(const string& sauce){

        m\_sauce = sauce;

    }

    void setTopping(const string& topping){

        m\_topping = topping;

    }

    void open() const {

        cout << "Pizza with " << m\_dough << " dough, " << m\_sauce << " sauce and "

            << m\_topping << " topping. Mmm." << endl;

    }

private:

    string m\_dough;

    string m\_sauce;

    string m\_topping;

};

// "Abstract Builder"

class PizzaBuilder

{

public:

    virtual ~PizzaBuilder() {};

    Pizza\* getPizza()

    {

        return m\_pizza.get();

    }

    void createNewPizzaProduct()

    {

        m\_pizza = make\_unique<Pizza>();

    }

    virtual void buildDough() = 0;

    virtual void buildSauce() = 0;

    virtual void buildTopping() = 0;

protected:

    unique\_ptr<Pizza> m\_pizza;

};

//----------------------------------------------------------------

class HawaiianPizzaBuilder : public PizzaBuilder

{

public:

    virtual ~HawaiianPizzaBuilder() {};

    virtual void buildDough()

    {

        m\_pizza->setDough("cross");

    }

    virtual void buildSauce()

    {

        m\_pizza->setSauce("mild");

    }

    virtual void buildTopping()

    {

        m\_pizza->setTopping("ham+pineapple");

    }

};

class SpicyPizzaBuilder : public PizzaBuilder

{

public:

    virtual ~SpicyPizzaBuilder() {};

    virtual void buildDough()

    {

        m\_pizza->setDough("pan baked");

    }

    virtual void buildSauce()

    {

        m\_pizza->setSauce("hot");

    }

    virtual void buildTopping()

    {

        m\_pizza->setTopping("pepperoni+salami");

    }

};

//----------------------------------------------------------------

class Cook

{

public:

    void openPizza()

    {

        m\_pizzaBuilder->getPizza()->open();

    }

    void makePizza(PizzaBuilder\* pb)

    {

        m\_pizzaBuilder = pb;

        m\_pizzaBuilder->createNewPizzaProduct();

        m\_pizzaBuilder->buildDough();

        m\_pizzaBuilder->buildSauce();

        m\_pizzaBuilder->buildTopping();

    }

private:

    PizzaBuilder\* m\_pizzaBuilder;

};

int main()

{

    Cook cook;

    HawaiianPizzaBuilder hawaiianPizzaBuilder;

    SpicyPizzaBuilder    spicyPizzaBuilder;

    cook.makePizza(&hawaiianPizzaBuilder);

    cook.openPizza();

    cook.makePizza(&spicyPizzaBuilder);

    cook.openPizza();

}

### ‎Fábrica‎‎[

**‎Definición‎**‎: una clase de utilidad que crea una instancia de una clase a partir de una familia de clases derivadas ‎

### ‎Abstract Factory‎‎

**‎Definición‎**‎: clase de utilidad que crea una instancia de varias familias de clases. También puede devolver una fábrica para un determinado grupo. ‎

‎El patrón de diseño de fábrica es útil en una situación que requiere la creación de muchos tipos diferentes de objetos, todos derivados de un tipo de base común. El método Factory define un método para crear los objetos, que las subclases pueden anular para especificar el tipo derivado que se creará. Por lo tanto, en tiempo de ejecución, al método Factory se le puede pasar una descripción de un objeto deseado (por ejemplo, una cadena leída de la entrada del usuario) y devolver un puntero de clase base a una nueva instancia de ese objeto. El patrón funciona mejor cuando se utiliza una interfaz bien diseñada para la clase base, por lo que no es necesario fundir el objeto devuelto. ‎

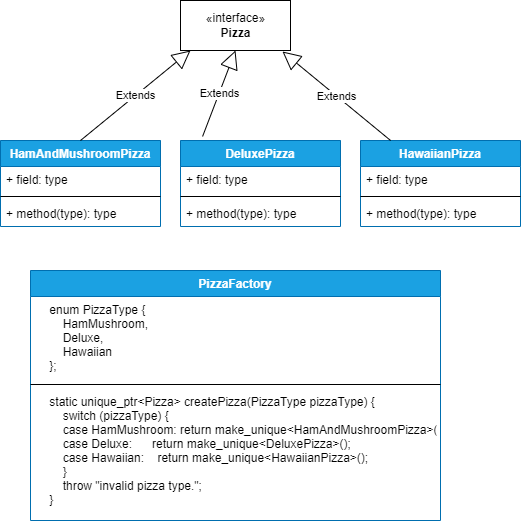
**‎Problema‎**

‎Queremos decidir en tiempo de ejecución qué objeto se va a crear en función de algún parámetro de configuración o aplicación. Cuando escribimos el código, no sabemos qué clase debe ser instanciada.‎

**‎Solución‎**

‎Defina una interfaz para crear un objeto, pero deje que las subclases decidan qué clase crear una instancia. Factory Method permite que una clase aplace la creación de instancias a subclases.‎

‎Imagina programar un videojuego, donde te gustaría agregar nuevos tipos de enemigos en el futuro, cada uno de los cuales tiene diferentes funciones de IA y puede actualizarse de manera diferente. Mediante el uso de un método de fábrica, el controlador del programa puede llamar a la fábrica para crear los enemigos, sin ninguna dependencia o conocimiento de los tipos reales de enemigos. Ahora, los futuros desarrolladores pueden crear nuevos enemigos, con nuevos controles de IA y nuevas funciones de miembros de dibujo, agregarlo a la fábrica y crear un nivel que llame a la fábrica, preguntando a los enemigos por su nombre. Combine este método con una descripción ‎[‎XML‎](https://en.wikibooks.org/wiki/XML)‎ de los niveles, y los desarrolladores podrían crear nuevos niveles sin tener que volver a compilar su programa. Todo esto, gracias a la separación de la creación de objetos del uso de objetos.‎



.‎

#include <stdexcept>

#include <iostream>

#include <memory>

using namespace std;

class Pizza {

public:

    virtual int getPrice() const = 0;

    virtual ~Pizza() {};  /\* without this, no destructor for derived Pizza's will be called. \*/

};

class HamAndMushroomPizza : public Pizza {

public:

    virtual int getPrice() const { return 850; };

    virtual ~HamAndMushroomPizza() {};

};

class DeluxePizza : public Pizza {

public:

    virtual int getPrice() const { return 1050; };

    virtual ~DeluxePizza() {};

};

class HawaiianPizza : public Pizza {

public:

    virtual int getPrice() const { return 1150; };

    virtual ~HawaiianPizza() {};

};

class PizzaFactory {

public:

    enum PizzaType {

        HamMushroom,

        Deluxe,

        Hawaiian

    };

    static unique\_ptr<Pizza> createPizza(PizzaType pizzaType) {

        switch (pizzaType) {

        case HamMushroom: return make\_unique<HamAndMushroomPizza>();

        case Deluxe:      return make\_unique<DeluxePizza>();

        case Hawaiian:    return make\_unique<HawaiianPizza>();

        }

        throw "invalid pizza type.";

    }

};

/\*

\* Create all available pizzas and print their prices

\* Remember Free Function ???

\* Multiparadign in Action.

\*/

void pizza\_information(PizzaFactory::PizzaType pizzatype)

{

    unique\_ptr<Pizza> pizza = PizzaFactory::createPizza(pizzatype);

    cout << "Price of " << pizzatype << " is " << pizza->getPrice() << std::endl;

}

int main()

{

    pizza\_information(PizzaFactory::HamMushroom);

    pizza\_information(PizzaFactory::Deluxe);

    pizza\_information(PizzaFactory::Hawaiian);

}

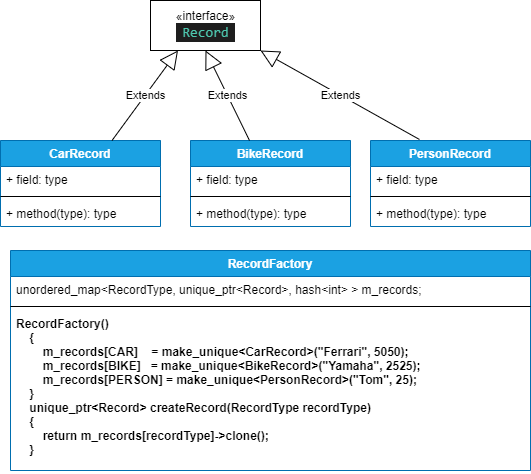
**‎Prototipo‎**‎

‎Un patrón prototipo se utiliza en el desarrollo de software cuando el tipo de objetos a crear está determinado por una instancia prototípica, que se clona para producir nuevos objetos. Este patrón se utiliza, por ejemplo, cuando el costo inherente de crear un nuevo objeto de manera estándar (por ejemplo, usando la palabra clave) es prohibitivamente caro para una aplicación determinada. ‎new

**‎Implementación‎**‎: declare una clase base abstracta que especifique un método virtual puro. Cualquier clase que necesite una capacidad de "constructor polimórfico" se deriva de la clase base abstracta e implementa la operación. ‎clone()clone()

‎Aquí el código de cliente invoca primero el método de fábrica. Este método de fábrica, dependiendo del parámetro, descubre la clase de concreto. En esta clase concreta, se llama al método y el objeto se devuelve mediante el método de fábrica. ‎clone()

* ‎Esta es una implementación de ejemplo del método Prototype. Tenemos la descripción detallada de todos los componentes aquí. ‎
  + Record‎ class, que es una clase virtual pura que tiene un método virtual puro.‎clone()
  + CarRecord‎, y como implementación concreta de una clase.‎BikeRecordPersonRecordRecord
  + ‎Un ‎**‎RecordType de enumeración‎**‎ como mapeo uno a uno de cada implementación concreta de clase.‎Record
  + RecordFactory‎ clase que tiene un método . Este método requiere un ‎**‎RecordType de enumeración‎**‎ como parámetro y, dependiendo de este parámetro, devuelve la implementación concreta de la clase.‎



/\*\* Implementation of Prototype Method \*\*/

/\*This is a sample implementation of Prototype method.

We have the detailed description of all the components here.

Record class, which is a pure virtual class that has a pure virtual method clone().

CarRecord, BikeRecord and PersonRecord as concrete implementation of a Record class.

An enum RecordType as one to one mapping of each concrete implementation of Record class.

RecordFactory class that has a Factory method CreateRecord(…).

This method requires an enum RecordType as parameter and depending on this parameter

it returns the concrete implementation of Record class.\*/

#include <iostream>

#include <unordered\_map>

#include <string>

#include <memory>

using namespace std;

/\*\* Record is the base Prototype \*/

class Record

{

public:

    virtual ~Record() {}

    virtual void print() = 0;

    virtual unique\_ptr<Record> clone() = 0;

};

/\*\* CarRecord is a Concrete Prototype \*/

class CarRecord : public Record

{

private:

    string m\_carName;

    int m\_ID;

public:

    CarRecord(string carName, int ID) : m\_carName(carName), m\_ID(ID)

    {

    }

    void print() override

    {

        cout << "Car Record" << endl

             << "Name  : "   << m\_carName << endl

             << "Number: "   << m\_ID << endl << endl;

    }

    unique\_ptr<Record> clone() override

    {

        return make\_unique<CarRecord>(\*this);

    }

};

/\*\* BikeRecord is the Concrete Prototype \*/

class BikeRecord : public Record

{

private:

    string m\_bikeName;

    int m\_ID;

public:

    BikeRecord(string bikeName, int ID) : m\_bikeName(bikeName), m\_ID(ID)

    {

    }

    void print() override

    {

        cout << "Bike Record" << endl

             << "Name  : " << m\_bikeName << endl

             << "Number: " << m\_ID << endl << endl;

    }

    unique\_ptr<Record> clone() override

    {

        return make\_unique<BikeRecord>(\*this);

    }

};

/\*\* PersonRecord is the Concrete Prototype \*/

class PersonRecord : public Record

{

private:

    string m\_personName;

    int m\_age;

public:

    PersonRecord(string personName, int age) : m\_personName(personName), m\_age(age)

    {

    }

    void print() override

    {

        cout << "Person Record" << endl

            << "Name : " << m\_personName << endl

            << "Age  : " << m\_age << endl << endl;

    }

    unique\_ptr<Record> clone() override

    {

        return make\_unique<PersonRecord>(\*this);

    }

};

/\*\* Opaque record type, avoids exposing concrete implementations \*/

enum RecordType

{

    CAR,

    BIKE,

    PERSON

};

/\*\* RecordFactory is the client \*/

class RecordFactory

{

private:

    unordered\_map<RecordType, unique\_ptr<Record>, hash<int> > m\_records;

public:

    RecordFactory()

    {

        m\_records[CAR]    = make\_unique<CarRecord>("Ferrari", 5050);

        m\_records[BIKE]   = make\_unique<BikeRecord>("Yamaha", 2525);

        m\_records[PERSON] = make\_unique<PersonRecord>("Tom", 25);

    }

    unique\_ptr<Record> createRecord(RecordType recordType)

    {

        return m\_records[recordType]->clone();

    }

};

int main()

{

    RecordFactory recordFactory;

    auto record = recordFactory.createRecord(CAR);

    record->print();

    record = recordFactory.createRecord(BIKE);

    record->print();

    record = recordFactory.createRecord(PERSON);

    record->print();

}

### ‎Singleton‎‎

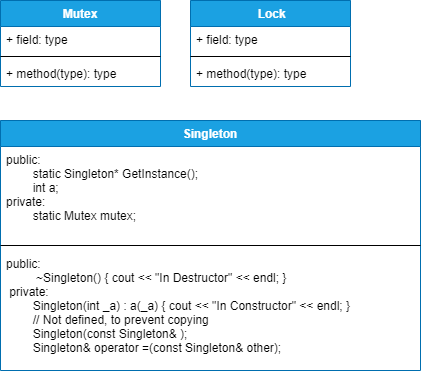
‎El ‎**‎patrón Singleton‎**‎ garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a esa instancia. Lleva el nombre del conjunto singleton, que se define como un conjunto que contiene un elemento. Esto es útil cuando se necesita exactamente un objeto para coordinar acciones en todo el sistema. ‎

**‎Lista de verificación‎**

* ‎Defina un atributo estático privado en la clase "instancia única".‎
* ‎Defina una función de descriptor de acceso estático público en la clase.‎
* ‎Haga "inicialización perezosa" (creación en el primer uso) en la función de accesorio.‎
* ‎Defina todos los constructores que se van a proteger o proteger.‎
* ‎Los clientes solo pueden usar la función de accesorio para manipular el Singleton.‎

‎Echemos un vistazo a cómo un Singleton difiere de otros tipos de variables. ‎

‎Al igual que una variable global, el Singleton existe fuera del alcance de cualquier función. La implementación tradicional utiliza una función de miembro estático de la clase Singleton, que creará una única instancia de la clase Singleton en la primera llamada y devolverá para siempre esa instancia. En el ejemplo de código siguiente se ilustran los elementos de una clase singleton de C++, que simplemente almacena una sola cadena.‎



#include <iostream>

using namespace std;

/\* Place holder for thread synchronization mutex \*/

class Mutex

{   /\* placeholder for code to create, use, and free a mutex \*/

};

/\* Place holder for thread synchronization lock \*/

class Lock

{   public:

        Lock(Mutex& m) : mutex(m) { /\* placeholder code to acquire the mutex \*/ }

        ~Lock() { /\* placeholder code to release the mutex \*/ }

    private:

        Mutex & mutex;

};

class Singleton

{   public:

        static Singleton\* GetInstance();

        int a;

        ~Singleton() { cout << "In Destructor" << endl; }

    private:

        Singleton(int \_a) : a(\_a) { cout << "In Constructor" << endl; }

        static Mutex mutex;

        // Not defined, to prevent copying

        Singleton(const Singleton& );

        Singleton& operator =(const Singleton& other);

};

Mutex Singleton::mutex;

Singleton\* Singleton::GetInstance()

{

    Lock lock(mutex);

    cout << "Get Instance" << endl;

    // Initialized during first access

    static Singleton inst(1);

    return &inst;

}

int main()