



INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA EN COMPUTACIÓN  
ÁREA ACADÉMICA DE INGENIERÍA EN COMPUTADORES  
ALGORITMOS Y ESTRUCTURAS DE DATOS II (CE-2103)

TAREA EXTRACLASE #2  
Patrones de diseño de software en C++ y Rust

Realizado por:  
Eduardo Moya Bello - 2015096664  
María Fernanda Ávila Marín - 2014089607  
Gabriel Abarca Aguilar - 2017110442

Profesor:  
Ing. Isaac Ramírez, M.SI.

Cartago, 2 de Septiembre de 2018

## Tabla de contenidos

<b>Introducción</b>	<b>2</b>
<b>Facade</b>	<b>3</b>
C++	3
Rust	3
<b>Adapter</b>	<b>4</b>
C++	4
Rust	4
<b>Builder</b>	<b>5</b>
C++	5
Rust	5
<b>Observer</b>	<b>6</b>
C++	6
Rust	6
<b>Abstract Factory</b>	<b>7</b>
C++	7
Rust	7

# Introducción

En diseño de software, un patrón de diseño se refiere a una solución específica a un problema definido. En otras palabras, es una manera de programar que garantiza evitar un problema común. Dicha solución ha sido analizada por varios científicos de la computación por lo que es una solución eficiente.

Con el presente trabajo se busca investigar sobre los patrones de diseño de software para mejorar las prácticas de programación y también implementar los patrones de diseño en C++ y Rust para examinar las diferencias entre ambos lenguajes.

El presente trabajo aborda el uso de los patrones de diseño Facade, Adapter, Builder, Observer y Abstract Factory en los lenguajes de programación C++ y Rust, mediante el uso de ejemplos. El código fuente de los patrones de diseño Facade y Adapter, se encuentra disponible en [GitHub](#) (Adapter se encuentra únicamente en C++).

## Facade

La idea del patrón de diseño Facade es tomar un sistema complejo y hacerlo sencillo de utilizar para el cliente. Para el siguiente ejemplo, se toma una línea de producción de celulares, la cual lleva muchos procesos. Estos procesos se pueden unir en una clase Facade, para que el cliente solo deba decir que desea producir cierta cantidad de celulares.

### C++

Lo primero que se debe hacer en C++ es hacer importar y definir todo aquello a utilizar. Esta es una de las principales distinciones que tiene C++ con Rust, ya que para imprimir y el método `to_string()` se debe utilizar el *namespace* `std`. Además, el método de impresión en consola es diferente, como se analizará más adelante.

```
#include <iostream>
using namespace std;
#define println(x) cout << x << endl;
```

Posteriormente, se crea la primera clase en la línea de producción que consiste en la extracción de materia prima. Cabe destacar que la concatenación en C++ es únicamente en variables del mismo tipo (como sucede con Java) por lo que se debe usar el método `std::to_string()`. Además, a diferencia de Rust, la declaración de una clase o estructura finaliza con punto y coma.

```
struct MaterialFinder {
    static void find_metal(int amount) {
        println("Searching for " + to_string(500 * amount) + " aluminium grams...");
        println("Searching for " + to_string(10 * amount) + " silicon grams...");
        println("Searching for " + to_string(1 * amount) + " gold grams...");
    }

    static void find_other_materials(int amount) {
        println("Searching for " + to_string(200 * amount) + " plastic grams...");
        println("Searching for " + to_string(300 * amount) + " glass grams...");
    }
};
```

Después de la extracción de la materia prima, se debe transportar dicha materia prima. Además, el transporte se encargará de distribuir el producto una vez finalizada la etapa de producción.

```
struct Transport {
    static void transport_materials() {
        println("Transporting products to the factory...");
    }

    static void transport_product() {
        println("Tansporting the phones to the stores...");
    }
};
```

Los materiales extraídos se deben procesar para crear los componentes de los celulares.

```
struct MaterialProcessor {
    static void process_metals() {
        println("Processing aluminium for the phone structures...");
        println("Processing silicon for semiconductors...");
        println("Processing gold for semiconductors...");
    }

    static void process_other_materials() {
        println("Processing plastic for the phone cases, chargers and headphones...");
        println("Processing glass for the screens...");
    }
};
```

En la fábrica se crean todos los componentes con los componentes procesados.

```
struct Factory {
    static void create_processor() {
        println("Requesting silicon and gold...");
        println("Creating the processors...");
        println("Processors created.");
    }

    static void create_memory() {
        println("Requesting silicon and gold...");
        println("Creating the memories...");
        println("Memories created.");
    }
};
```

```

static void create_remaining() {
    println("Requesting plastic, glass and other materials...");
    println("Creating the screens...");
    println("Creating the cases...");
    println("Creating the motherboards...");
    println("Creating the cameras...");
    println("Creating the buttons...");
    println("Creating the chargers...");
    println("Creating the headphones...");
    println("All components created.");
}
};

```

Luego se procede a la sección de ensamblaje, donde los componentes se unen para producir un celular funcional.

```

struct Assembler {
    static void join_components() {
        println("Preparing components...");
        println("Requesting needed materials...");
        println("All the components are ready to assemble.");
    }

    static void assemble_materials() {
        println("Turning on assembling machines...");
        println("Requesting screws...");
        println("Assembling...");
        println("Assembling finished successfully.");
    }
};

```

Se deben hacer diversas pruebas a los celulares para comprobar su funcionamiento. Nótese la sintaxis del while en C++.

```

struct Tester {
    static void test_phones(int amount) {
        int counter = 1;
        while (counter <= amount) {
            println("Testing phone " + to_string(counter) + " of " + to_string(amount) + ".");
            counter += 1;
        }
        println("All phones tested successfully.");
        println("All phones are ready for distribution.");
    }
};

```

```
};
```

Finalmente se empaacan los celulares para transportarlos y venderlos.

```
struct Packager {
    static void request_boxes(int amount) {
        println("Requesting " + to_string(amount) + " boxes to manufacturer...");
        println("Boxes ready.");
    }

    static void package_phones(int amount) {
        int counter = 1;
        while (counter <= amount) {
            println("Packing phone " + to_string(counter) + " of " + to_string(amount) +
                ".");
            counter += 1;
        }
        println("All phones packaged successfully.");
        println("All phones are ready for distribution.");
    }
};
```

El hecho de que cada vez que se desee fabricar un celular, el cliente deba manejar todas esas clases, aumenta la posibilidad de errores, los cuales en un ambiente de producción, significan pérdidas millonarias. Por lo tanto, el Facade puede encargarse de tomar todos esos procesos, para simplificarle al cliente su tarea como se observa a continuación.

```
struct PhoneFactoryFacade {
    void develop_phones(int amount_of_phones) {

        println("=====");
        MaterialFinder::find_metal(amount_of_phones);
        println("-----");
        MaterialFinder::find_other_materials(amount_of_phones);

        println("=====");
        Transport::transport_materials();

        println("=====");
        MaterialProcessor::process_metals();
        println("-----");
        MaterialProcessor::process_other_materials();
    }
};
```

```

println("=====");
    Factory::create_processor();
    println("-----");
    Factory::create_memory();
    println("-----");
    Factory::create_remaining();

println("=====");
    Assembler::join_components();
    println("-----");
    Assembler::assemble_materials();

println("=====");
    Tester::test_phones(amount_of_phones);

println("=====");
    Packager::request_boxes(amount_of_phones);
    println("-----");
    Packager::package_phones(amount_of_phones);

println("=====");
    Transport::transport_product();

println("=====");
}
};

```

El main (que sería lo que manejaría el cliente) se simplificaría mucho, quedando como procede.

```

int main() {
    PhoneFactoryFacade facade;
    facade.develop_phones(10);
    return 0;
}

```

Lo que genera la siguiente salida.

```

"/home/edmob/TEC/DatosII/TE1/C++/Facade/cmake-build-debug/Facade"
=====
Searching for 5000 aluminium grams...
Searching for 100 silikon grams...
Searching for 10 gold grams...
-----

```



```

Searching for 2000 plastic grams...
Searching for 3000 glass grams...
=====
Transporting products to the factory...
=====
Processing aluminium for the phone structures...
Processing sillicon for semiconductors...
Processing gold for semiconductors...
-----
Processing plastic for the phone cases, chargers and headphones...
Processing glass for the screens...
=====
Requesting sillicon and gold...
Creating the processors...
Processors created.
-----
Requesting sillicon and gold...
Creating the memories...
Memories created.
-----
Requesting plastic, glass and other materials...
Creating the screens...
Creating the cases...
Creating the motherboards...
Creating the cameras...
Creating the buttons...
Creating the chargers...
Creating the headphones...
All components created.
=====
Preparing components...
Requesting needed materials...
All the components are ready to assemble.
-----
Turning on assembling machines...
Requesting screws...
Assembling...
Assembling finished successfully.
=====
Testing phone 1 of 10.
Testing phone 2 of 10.
Testing phone 3 of 10.
Testing phone 4 of 10.
Testing phone 5 of 10.
Testing phone 6 of 10.
Testing phone 7 of 10.
Testing phone 8 of 10.
Testing phone 9 of 10.

```

```
Testing phone 10 of 10.
All phones tested successfully.
All phones are ready for distribution.
=====
Requesting 10 boxes to manufacturer...
Boxes ready.
-----
Packing phone 1 of 10.
Packing phone 2 of 10.
Packing phone 3 of 10.
Packing phone 4 of 10.
Packing phone 5 of 10.
Packing phone 6 of 10.
Packing phone 7 of 10.
Packing phone 8 of 10.
Packing phone 9 of 10.
Packing phone 10 of 10.
All phones packaged successfully.
All phones are ready for distribution.
=====
Transporting the phones to the stores...
=====

Process finished with exit code 0
```

## Rust

En el caso de Rust usaremos de ejemplo una fábrica de Pizzas, donde el Facade va a crear las pizzas automáticamente, y el main o cliente solo le va a decir cuál tipo quiere. Se instancia un trait para clase abstracto y las diferentes pizzas que van a crear, en este caso circular y cuadrada.

```
//Facade con el cocinero de Pizzas
trait Pizza{
    fn bake(&self);
}
//Crear Pizzas Circulares
struct CircularPizza;

impl Pizza for CircularPizza {
    fn bake(&self) {
        println!("Baking Circular Pizza");
    }
}
//Crear Pizzas Cuadradas
struct SquarePizza;

impl Pizza for SquarePizza {
    fn bake(&self) {
        println!("Baking Square Pizza");
    }
}
//Cocinero de Pizzas
struct Facade{
    P1 : CircularPizza,
    P2 : SquarePizza
}

impl Facade{
    pub fn bakeCircle(&self){
        self.P1.bake();
    }
    pub fn bakeRect(&self){
        self.P2.bake();
    }
}
```

Para enfatizar en el Facade, debemos tener 2 variables, que van a ser las diferentes pizzas. después el facade va a tener las opción de cocinar esas pizzas, algo que el cliente no va a hacer, si no que ese trabajo le toca al Facade.

```
//Cocinero de Pizzas
struct Facade{
    P1 : CircularPizza,
    P2 : SquarePizza
}

impl Facade{
    pub fn bakeCircle(&self){
        self.P1.bake();
    }
    pub fn bakeRect(&self){
        self.P2.bake();
    }
}

//Pizzas
fn main(){
    let interface = Facade{ P1 : CircularPizza, P2 : SquarePizza };
    interface.bakeCircle();
    interface.bakeRect();
}
```

Recibimos los siguientes outputs:



The screenshot shows a Rust IDE window titled "Execution" with a "Close" button. The code editor displays the following code:

```
30 | /      pub fn bakeRect(&self){
31 | |          self.P2.bake();
32 | |      }
   | |____^
```

Below the code, the execution status is shown: "Finished dev [unoptimized + debuginfo] target(s) in 0.75s" and "Running 'target/debug/playground'". The "Standard Output" pane at the bottom displays the following output:

```
Baking Circular Pizza
Baking Square Pizza
```

Donde se ve que se están cocinando las pizzas.

# Adapter

El patrón adapter toma una interfaz y la convierte en otra que el cliente espera.

## C++

El código utilizado para realizar el patrón adapter se muestra a continuación.

```
#include <iostream>
using namespace std;

#define println(x) cout << x << endl;
class SpeedMeter {
public:
    virtual void calculateSpeed() = 0;
};

class EnglishSpeedMeter {
private:
    float miles_;
    float hours_;
public:
    EnglishSpeedMeter(float miles, float hours) {
        miles_ = miles;
        hours_ = hours;
    }

    void calculateMphSpeed() {
        println("The speed is " + to_string(miles_/hours_) + " miles per hour.");
    }
};

class SISpeedMeterAdapter: public SpeedMeter, private EnglishSpeedMeter{
public:
    SISpeedMeterAdapter(float meters_, float seconds_) : EnglishSpeedMeter(
meters_ / 1609.344, seconds_ / 3600) {}
    virtual void calculateSpeed() {
        calculateMphSpeed();
    }
};

int main() {
    SpeedMeter* sm = new SISpeedMeterAdapter(1000, 5);
    sm->calculateSpeed();
    return 0;
}
```

Como se observa en el ejemplo anterior, el cliente tiene los datos necesarios para calcular la velocidad en metros y segundos, pero necesita obtener el valor en millas por hora. Esto es algo muy común en la actualidad, y para los ingenieros mecánicos es una constante tarea el tener que convertir unidades.

De este ejemplo se rescatan algunas características. En el patrón Facade, se observa el uso de estructuras, mientras que en este, el de clases. Realmente la única diferencia entre ambas es la privacidad. En las clases, por defecto todo es privado. En el código se muestra qué se desea mantener privado y qué no. Se destaca también el uso de voids virtuales, que permiten a la clase que las hereda, utilizar los atributos. Además, al principio se define una función virtual pura, que se asemeja a una función abstracta en Java.

La salida obtenida es la siguiente:

```
"/home/edmob/TEC/Datos
II/TE1/C++/Adapter/cmake-build-debug/Adapter"
The speed is 447.387268 miles per hour.

Process finished with exit code 0
```

## Rust

En el lenguaje Rust hay ciertas diferencias en la implementación de este código. Primero, la inexistencia de interfaces (lo más similar son los traits). Además, como se ha visto anteriormente, la referencia que tienen las estructuras con el `self`. En Rust, no existe una manera directa para hacer herencias, por lo que para implementar el adaptador, se tuvo que crear una instancia del `EnglishSpeedMeter`.

```
use std::{i8, i16, i32, i64, u8, u16, u32, u64, isize, usize, f32, f64};
use std::io::stdin;

trait SpeedMeter {
    fn calculateSpeed(&self);
}

struct EnglishSpeedMeter {
```

```

    miles_: f64,
    hours_: f64,
}

impl EnglishSpeedMeter {
    fn calculateMphSpeed(&self) {
        println!("The speed is {} miles per hour.", self.miles_/self.hours_);
    }
}

struct SISpeedMeterAdapter {
    meters_: f64,
    seconds_: f64,
}

impl SpeedMeter for SISpeedMeterAdapter {
    fn calculateSpeed(&self) {
        let esm = EnglishSpeedMeter {
            miles_: self.meters_ / 1609.344,
            hours_: self.seconds_ / 3600.0
        };
        esm.calculateMphSpeed();
    }
}

fn main() {
    let adapter = SISpeedMeterAdapter {
        meters_: 1000.0,
        seconds_: 5.0,
    };
    adapter.calculateSpeed();
}

```

El resultado final es el siguiente:

```

edmobe@AW17edmobe:~/TEC/Datos II/TE1/Rust/Adapter$ ./adapter
The speed is 447.3872584108804 miles per hour.

```

# Builder

El patrón Builder es utilizado para crear objetos que tienen muchos atributos y a la hora de generar instancias es difícil ver qué parámetro corresponde a qué espacio en la llamada. a continuación veremos ejemplos con un creador de pizzas tanto en Rust como en C++.

## C++

Inicialmente se definen la clase a la cual se le hará el patrón, en este caso una pizza

```
class foo
{
public:
    class builder;
    foo(bool ConJamon, bool ConQueso, bool ConPinna, bool ConCarne, bool ConHongos) :
        ConJamon{ ConJamon },
        ConQueso{ ConQueso },
        ConPinna{ ConPinna },
        ConCarne{ ConCarne },
        ConHongos{ ConHongos } {}

    bool ConJamon;
    bool ConQueso;
    bool ConPinna;
    bool ConCarne;
    bool ConHongos;
};
```



Luego se crean las funciones que agregan los “ingredientes” a la pizza.

```
class foo::builder
{
public:
    builder & set_jamon(bool value)
    {
        ConJamon = value;
        return *this;
    };
    builder & set_queso(bool value)
    {
        ConQueso = value;
        return *this;
    };
    builder & set_pinna(bool value)
    {
        ConPinna = value;
        return *this;
    };
    builder & set_carne(bool value)
    {
        ConCarne = value;
        return *this;
    };
    builder & set_hongos(bool value)
    {
        ConHongos = value;
        return *this;
    };
    foo build()const
    {
        return
            foo
            {
                ConJamon, ConQueso, ConPinna, ConCarne, ConHongos };
    }
};
```

Finalmente en el main se llama a las funciones que crean la pizza con los ingredientes deseados

```
int main()
{
    foo f = foo::builder
    {
        }.set_jamon(true).set_queso(true).build();
    cout << "Ingredientes de su orden:" << endl;
    cout << "Jamon: " << f.ConJamon << endl;
    cout << "Queso: " << f.ConQueso << endl;
    cout << "Pinna: " << f.ConPinna << endl;
    cout << "Carne: " << f.ConCarne << endl;
    cout << "Hongos: " << f.ConHongos << endl;
    system("pause");
}
```

```

D:\Maria\Cosas Maria\Progras datos 2 why\Builder Pattern\...
Ingredientes de su orden:
Jamón: 1
Queso: 1
Pinna: 0
Carne: 0
Hongos: 0
Press any key to continue . . .

```

## Rust

En este caso se sigue la misma estructura que en C++. Primero se crea la clase base:

```

struct Pizza<'a> { // Clase basica con varios atributos
    jamon: &'a bool,
    queso: &'a bool,
    salsa: &'a bool,
    pinna: &'a bool,
    hongos: &'a bool,
    carne: &'a bool,
}

```

Luego se crea el struct y las funciones del builder:

```

struct PizzaBuilder<'a> { //se crea la estructura del builder, el cual posee los mismos atributos que la clase
    jamon: &'a bool,
    queso: &'a bool,
    salsa: &'a bool,
    pinna: &'a bool,
    hongos: &'a bool,
    carne: &'a bool,
}

impl<'a> PizzaBuilder<'a> { //se crea la funcion que crea las pizzas, inician vacias
    fn new() -> PizzaBuilder<'a> {
        PizzaBuilder {
            jamon:&false,
            queso:&false,
            salsa: &false,
            pinna: &false,
            hongos: &false,
            carne: &false,
        }
    }
}

```

Luego se crean las funciones para agregar los ingredientes

```
fn con_jamon(mut self, jamon: &'a bool) -> Self { //funcion para agregar Jamon
    self.jamon = jamon;
    println!("Agregando Jamon\n");
    self
}

fn con_queso(mut self, queso: &'a bool) -> Self { //funcion para agregar queso
    self.queso = queso;
    println!("Agregando Queso\n");
    self
}

fn con_salsa(mut self, salsa: &'a bool) -> Self { //funcion para agregar salsa
    self.salsa = salsa;
    println!("Agregando Salsa\n");
    self
}

fn con_pinna(mut self, pinna: &'a bool) -> Self { //funcion para agregar pinna
    self.pinna = pinna;
    println!("Agregando Pinna\n");
    self
}

fn con_hongos(mut self, hongos: &'a bool) -> Self { //funcion para agregar hongos
    self.hongos = hongos;
    println!("Agregando Hongos\n");
    self
}

fn con_carne(mut self, carne: &'a bool) -> Self { //funcion para agregar carne
    self.carne = carne;
    println!("Agregando Carne\n");
    self
}

fn build(self) -> Pizza<'a> { //funcion que tguarda los datos en la pizza final
    Pizza {
        jamon: self.jamon,
        queso: self.queso,
        salsa: self.salsa,
        pinna: self.pinna,
        hongos: self.hongos,
        carne: self.carne,
    }
}
```

Finalmente en el main se crea una instancia con builder

```
fn main() { //Main donde se llama al builder
    let pizza = PizzaBuilder::new().con_salsa(&true)
        .con_queso(&true)
        .con_jamon(&true)
        .con_pinna(&true)
        .con_hongos(&true)
        .con_carne(&true)
        .build();
}
```

Standard Output

Agregando Salsa  
Agregando Queso  
Agregando Jamon  
Agregando Pinna  
Agregando Hongos  
Agregando Carne

## Observer

Observer es un patrón utilizado en casos donde se necesita estar pendiente de algún cambio de valor, sin la necesidad de tener un monitoreo en tiempo real y constante, el cual consume muchos recursos.

### C++

Inicialmente se crea la clase que tiene el valor de interés, con funciones para agregar observadores, quitarlos y notificarlos, además de una lista de observadores

```
class Pizzeria { //clase que se encarga de notificar
    vector < class Cliente * > views; //Se crea una lista de observadores
    bool value; //Se crea un valor el cual si se modifica se notifica a los observadores
public:
    void attach(Cliente *obs) { //Funcion que agrega un observador a la lista
        views.push_back(obs);
    }
    void setVal(bool val) { //funcion que modifica el valor y llama a la funcion que notifica a los observadores
        value = val;
        notify();
    }
    int getVal() { //Funcion que devuelve el valor monitoreado
        return value;
    }
    void notify();
};
```

Luego se crea una clase que son los clientes, a los cuales les interesa el cambio del valor

```
class Cliente { //clase que es notificada
    Pizzeria *model; //se crea un sujeto al cual se le agrega el observador
public:
    Cliente(Pizzeria *mod) {
        model = mod;
        model->attach(this);
    }
    virtual void update() = 0;
protected:
    Pizzeria * getPizzeria() {
        return model;
    }
};
```

Aca podemos ver la función notificar mas a detalle, en la cual se le notifica a cada miembro de la lista de observadores:

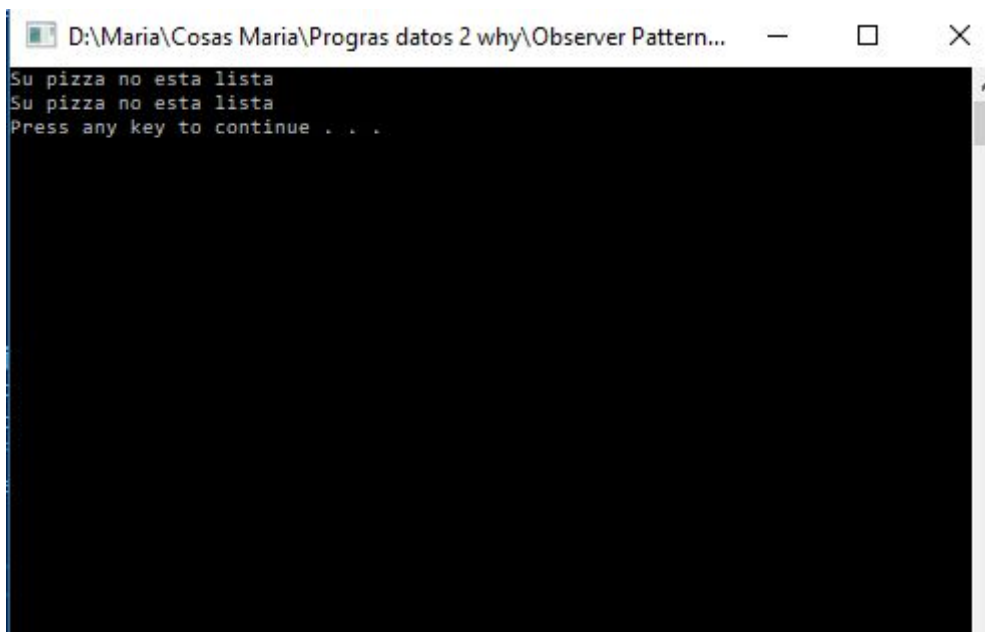
```
void Pizzeria::notify() { //Funcion que notifica a todos los observadores de que se cambio el valor
    for (int i = 0; i < views.size(); i++)
        views[i]->update();
}
```

Finalmente se crea una clase que se encarga de pasar el dato nuevo a los clientes

```
class estadoPizza : public Cliente { //clase que se encarga de mandar un mensaje en caso de cambio de valor
public:
    estadoPizza(Pizzeria *mod) : Cliente(mod) {}
    void update() {
        int v = getPizzeria()->getVal();
        if (v) {
            cout << "Su pizza esta lista" << '\n';
        }
        else {
            cout << "Su pizza no esta lista" << '\n';
        }
    }
};
```

Podemos ver en el main como cómo se crean las instancias:

```
int main() {
    Pizzeria subj; //Se crea un sujeto
    estadoPizza cliente1(&subj); //se crean observadores
    estadoPizza cliente2(&subj);
    subj.setVal(false); //se modifica el valor, lo que causa que sean notificados los observadores
    system("pause");
}
```



```
D:\Maria\Cosas Maria\Progras datos 2 why\Observer Pattern...
Su pizza no esta lista
Su pizza no esta lista
Press any key to continue . . .
```



## Rust

Se crea la “clase” observer y el trait evento para notar los cambios (sujeto)

```
#[allow(unused_variables)]
pub trait Events {
    fn on_value(&self, value: &str) {}
}
struct Observable {
    value: String,
    observers: Vec<Box<Events>>,
}
```

Luego se implementa observer de manera que tenga observadores y el valor de importancia. Cuando se cambia el valor notifica a los clientes. Posee una función para agregar observadores

```
impl Observable {
    fn new(value: &str) -> Observable {
        Observable {
            value: value.to_owned(),
            observers: Vec::new(),
        }
    }

    fn set_value(&mut self, value: &str) {
        self.value = value.to_owned();
        // send event to observers
        for observer in &self.observers {
            observer.on_value(value);
        }
    }

    fn register<E: Events + 'static>(&mut self, observer: E) {
        self.observers.push(Box::new(observer));
    }
}
```

Acá se ve que cuando se cambia el valor se imprime el nuevo valor en consola

```
struct Observer;

impl Events for Observer {
    fn on_value(&self, value: &str) {
        println!("received value: {:?}", value);
    }
}

fn main() {
    let mut observable = Observable::new("initial value");
    observable.register(Observer);
    observable.set_value("updated value");
}
```

Standard Output

received value: "updated value"



## Abstract Factory

El Abstract Factory es usado para crear una serie de objetos con una base común, en ambos casos (Rust y C++) vamos a tomar de ejemplo la creación de Pizzas de diferente forma.

### C++

Se define una clase abstracta, la cual va a ser el molde de los diferentes productos.

```
//Clase abstracta principal
class Pizza {
public:
    Pizza() {
        id_ = total_++;
    }
    virtual void draw() = 0;
protected:
    int id_;
    static int total_;
};
int Pizza::total_ = 0;
```

Se crean las diferentes clases que van a ser implementadas por las diferentes fábricas.

```
//Tipos de Pizzas
class Circle : public Pizza {
public:
    void draw() {
        cout << "Circular Pizza " << id_ << ": baking" << endl;
    }
};
class Square : public Pizza {
public:
    void draw() {
        cout << "Squared Pizza " << id_ << ": baking" << endl;
    }
};
class Ellipse : public Pizza {
public:
    void draw() {
        cout << "Elipctic Pizza " << id_ << ": baking" << endl;
    }
};
class Rectangle : public Pizza {
public:
    void draw() {
        cout << "Rectangular Pizza " << id_ << ": baking" << endl;
    }
};
```

Luego se crea una fábrica abstracta que sirve de molde para las fábricas que crean las diferentes Pizzas. Se obtendrá una para pizzas cuadradas y circulares, y para otra que dará pizzas alargadas, elípticas y rectangulares.

```
//Diferentes Fabricas
class Factory {
public:
    virtual Pizza* createCurvedInstance() = 0;
    virtual Pizza* createStraightInstance() = 0;
};

class SimplePizzaFactory : public Factory {
public:
    Pizza* createCurvedInstance() {
        return new Circle;
    }
    Pizza* createStraightInstance() {
        return new Square;
    }
};

class RobustPizzaFactory : public Factory {
public:
    Pizza* createCurvedInstance() {
        return new Ellipse;
    }
    Pizza* createStraightInstance() {
        return new Rectangle;
    }
};
```

Luego se procede a crear cada pizza.

```
//Main de la fabrica de pizzas
int main() {
    cout << "Simple Pizza Factory\n";
    Factory* factory = new SimplePizzaFactory;

    Pizza* shapes[3];

    shapes[0] = factory->createCurvedInstance(); // shapes[0] = new Ellipse;
    shapes[1] = factory->createStraightInstance(); // shapes[1] = new Rectangle;
    shapes[2] = factory->createCurvedInstance(); // shapes[2] = new Ellipse;

    for (int i=0; i < 3; i++) {
        shapes[i]->draw();
    }
    Factory* factory2 = new RobustPizzaFactory;

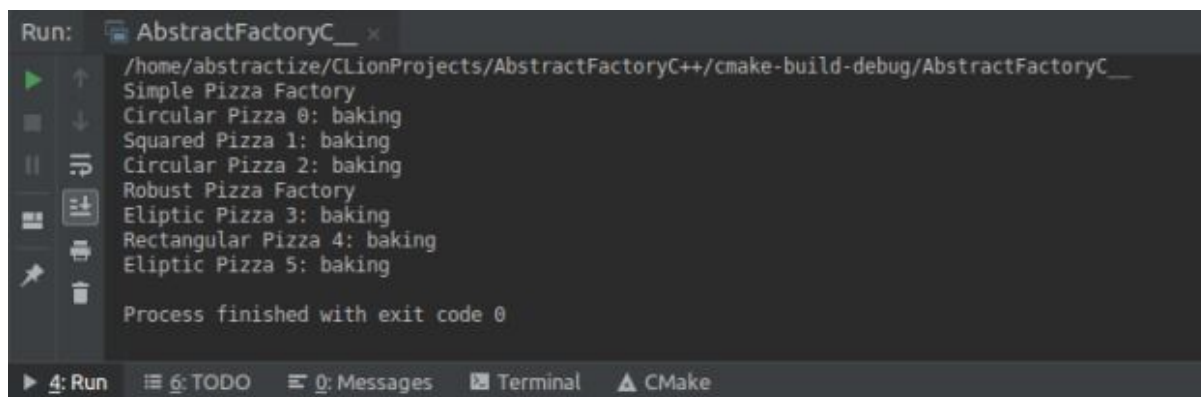
    cout << "Robust Pizza Factory\n";

    Pizza* shapes2[3];

    shapes2[0] = factory2->createCurvedInstance(); // shapes[0] = new Ellipse;
    shapes2[1] = factory2->createStraightInstance(); // shapes[1] = new Rectangle;
    shapes2[2] = factory2->createCurvedInstance(); // shapes[2] = new Ellipse;

    for (int i=0; i < 3; i++) {
        shapes2[i]->draw();
    }
}
```

Al crear las pizzas se obtiene:



```
Run: AbstractFactoryC_ x
/home/abstractize/CLionProjects/AbstractFactoryC++/cmake-build-debug/AbstractFactoryC_
Simple Pizza Factory
Circular Pizza 0: baking
Squared Pizza 1: baking
Circular Pizza 2: baking
Robust Pizza Factory
Eliptic Pizza 3: baking
Rectangular Pizza 4: baking
Eliptic Pizza 5: baking
Process finished with exit code 0
```

## Rust

Aquí se definen las 2 pizzas principales de Molde, una para pizzas de forma elíptica y otra de forma rectangular.

```
/*
 * Core Trait that defines a Curved Pizza
 */
trait Pizza1 {
    fn bake(&self);
    //Circular/redonda
}

/*
 * Core Trait that defines a Straighth Pizza
 */
trait Pizza2 {
    fn bake(&self);
    //Cuadrada
}
```

Se crea un trait de fábrica que va a servir como clase abstracta para moldear las fábricas concretas.

```
/*
 * Core Trait that defines a Pizza Factory
 */
trait Factory<P: Pizza1, T: Pizza2> {
    fn createCurvedInstance(&self) -> P;
    fn createStraightInstance(&self) -> T;
}
```

Luego se crean las fábricas concretas, la primera va a ser el de las Pizzas Simples con las diferentes Pizzas que va a crear.

```
/*  
 * Define the regular products  
 */  
struct CircularPizza;  
  
impl Pizza1 for CircularPizza {  
    fn bake(&self) {  
        println!("Baking Circular Pizza");  
    }  
}  
  
struct SquarePizza;  
  
impl Pizza2 for SquarePizza {  
    fn bake(&self) {  
        println!("Baking Square Pizza");  
    }  
}
```

```
/*  
 * Create SimpleFactory and implement SimplePizza type  
 */  
struct SimpleFactory;  
  
impl Factory<CircularPizza, SquarePizza> for SimpleFactory {  
    fn createCurvedInstance(&self) -> CircularPizza {  
        return CircularPizza;  
    }  
  
    fn createStraightInstance(&self) -> SquarePizza {  
        return SquarePizza;  
    }  
}
```

Se hace lo mismo con la segunda fábrica

```
struct EllipticPizza;

impl Pizza1 for EllipticPizza {
    fn bake(&self) {
        println!("Baking Elliptical Pizza");
    }
}

struct RectPizza;

impl Pizza2 for RectPizza {
    fn bake(&self) {
        println!("Baking Rectangular Pizza");
    }
}

/*
 * Create RobustFactory and implement the Robust Pizza Types
 */
struct RobustFactory;

impl Factory<EllipticPizza, RectPizza> for RobustFactory {
    fn createCurvedInstance(&self) -> EllipticPizza {
        return EllipticPizza;
    }

    fn createStraightInstance(&self) -> RectPizza {
        return RectPizza;
    }
}
```

Finalmente creamos las pizzas

```
fn main() {  
    // Create our two different factories  
    let VillaItalia = SimpleFactory;  
    let PizzaHut = RobustFactory;  
  
    // Both factories use the same interface, so let's just use them  
  
    // Test out Curved Pizzas  
    let pizza = VillaItalia.createCurvedInstance();  
    pizza.bake();  
  
    let pizza = PizzaHut.createCurvedInstance();  
    pizza.bake();  
  
    // Test out creating Straight Pizzas  
    let pizza = VillaItalia.createStraightInstance();  
    pizza.bake();  
  
    let pizza = PizzaHut.createStraightInstance();  
    pizza.bake();  
}
```

Obtenemos estos Outputs:



The screenshot shows a code editor window with a light blue background. The top section, labeled 'Execution', displays the following text:

```
98 | let PizzaHut = RobustFactory;  
    |          ^^^^^^^  
  
Finished dev [unoptimized + debuginfo] target(s) in 0.71s  
Running `target/debug/playground`
```

The bottom section, labeled 'Standard Output', displays the following text:

```
Baking Circular Pizza  
Baking Elliptical Pizza  
Baking Square Pizza  
Baking Rectangular Pizza
```