Design patterns for beginners (2/2)

Mustapha Tachouct

2017/02/13

Contents

3 Types in 23 Design Patterns

1. • Creational Patterns
2. • Structural Patterns
3. • Behavioral Patterns

[3 Types : Creational Patterns (1/3)]

- provide a way to create objects while hiding the creation logic
- replace the using of new operator

[3 Types : Structural Patterns (2/3)]

- Manage realtionships between entities
- Define ways to add new functionalities

[3 Types : Behavioral Patterns (3/3)]

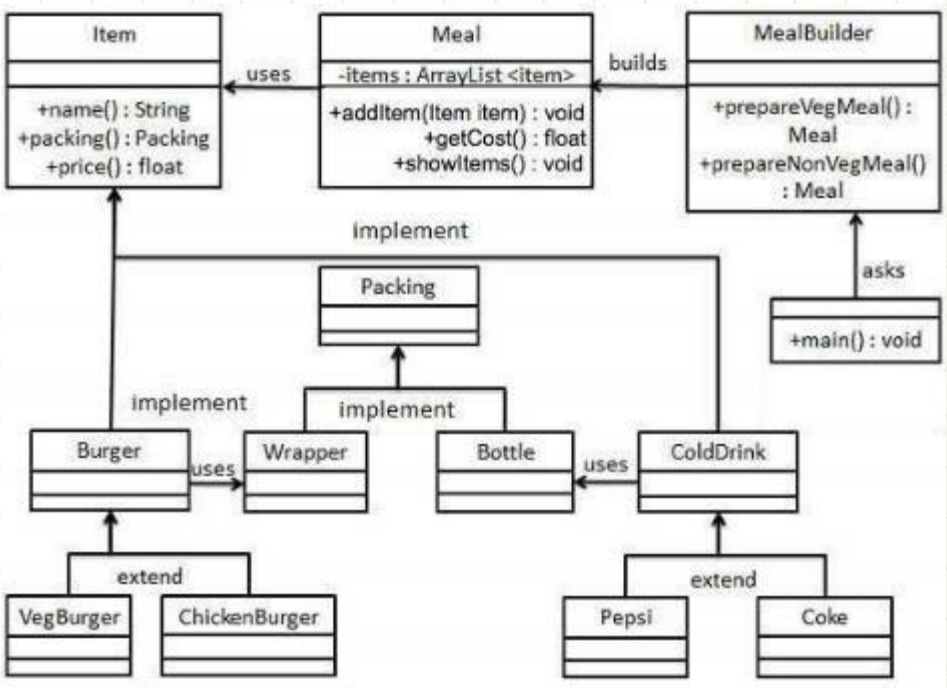
- concerne with communication between instances
- Increase flexibility the perform of this communication

[11 Patterns]

- Builder, Abstract factory, Composite, Decorator
- Facade, Flyweight, Command, Interpreter
- State, Template, Visitor

[Builder (13/23)]

- Builder pattern builds a complex object using simple objects and using a step by step approach
- type : creational pattern



[Builder (13/23)]

```
// Create an interface Item representing food item and packing.

class Item {
public:
    virtual const char* name() = 0;
    virtual Packing* packing() = 0;
    virtual float price() = 0;
};

class Packing {
public:
    virtual const char* pack() = 0;
};

// Create concrete classes implementing the Packing interface.
class Wrapper : public Packing {
public:
    const char* pack() {
        return "Wrapper";
    }
};

class Bottle : public Packing {
public:
    const char* pack() {
        return "Bottle";
    }
};

// Create abstract classes implementing the item interface
// providing default functionalities.

class Burger : public Item {
    Wrapper wrapper;
public:
    Packing* packing() {
        return &wrapper;
    }

    virtual float price() = 0;
};
```

```
class ColdDrink:  public Item {
    Bottle bottle;
public:
    Packing* packing() {
        return &bottle;
    }

    virtual float price() = 0;
};

// Create concrete classes extending Bu
class VegBurger : public Burger {
public:
    float price() {
        return 25.0f;
    }

    const char*  name() {
        return "Veg Burger";
    }
};

class ChickenBurger : public Burger {
public:
    float price() {
        return 50.5f;
    }

    const char* name() {
        return "Chicken Burger";
    }
};
class Coke : public ColdDrink {
public:
    float price() {
        return 30.0f;
    }

    const char* name() {
        return "Coke";
    }
};

class Pepsi : public ColdDrink {
public:
    float price() {
        return 35.0f;
    }
```

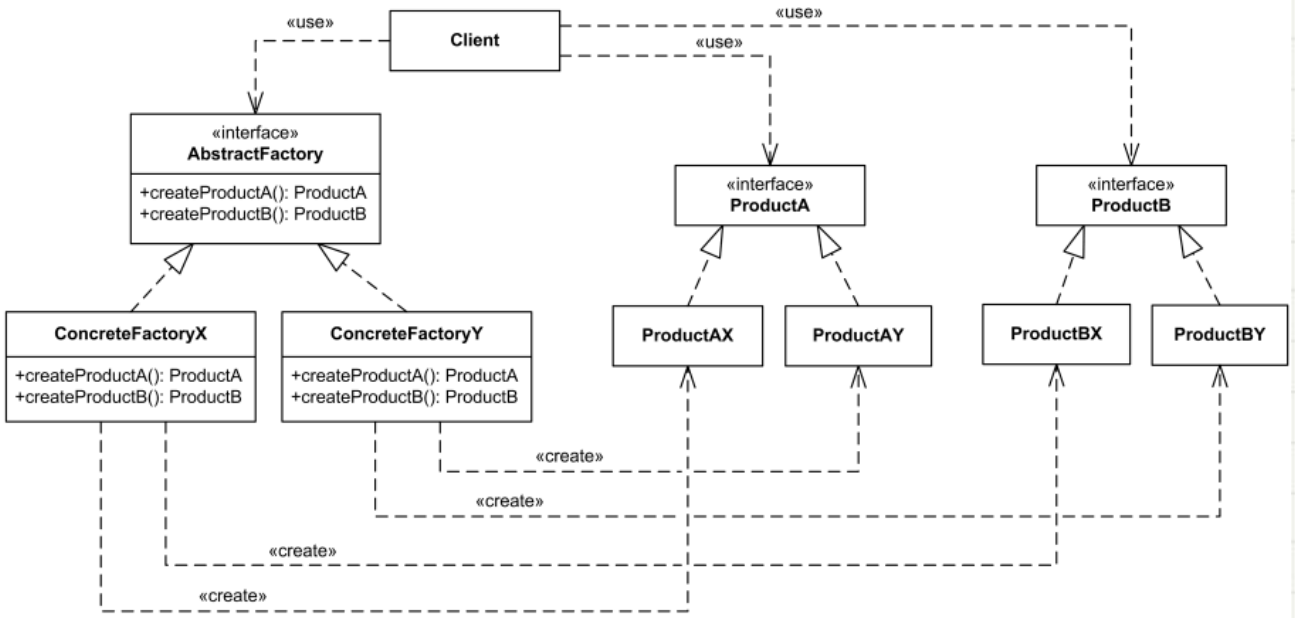
[Builder (13/23)]

```
// Create a Meal class having Item objects defined above.

class Meal {
private:
    std::list<Item*> items;

public:
    virtual ~Meal()
    {
        for (Item* item : items) {
            delete item;
        }
        items.clear();
    }

    void addItem(Item* item) {
        items.push_back(item);
    }

    float getCost() {
        float cost = 0.0f;

        for (Item* item : items) {
            cost += item->price();
        }
        return cost;
    }

    void showItems() {
        for (Item* item : items) {
            cout << "Item : " << item->name();
            cout << ", Packing : " << item->packing()->pack();
            cout << ", Price : " << item->price();
            cout << endl;
        }
    }
};
```

```
// Create a MealBuilder class, the actual builder class responsible to create Meal objects.
class MealBuilder {

public:
    Meal* prepareVegMeal() {
        Meal* meal = new Meal();
        meal->addItem(new VegBurger());
        meal->addItem(new Coke());
        return meal;
    }

    Meal* prepareNonVegMeal() {
        Meal* meal = new Meal();
        meal->addItem(new ChickenBurger());
        meal->addItem(new Pepsi());
        return meal;
    }
};
// BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

void main(int argc, char ** argv) {

    MealBuilder* mealBuilder = new MealBuilder();

    Meal* vegMeal = mealBuilder->prepareVegMeal();
    cout << "Veg Meal" << endl;
    vegMeal->showItems();
    cout << "Total Cost: " << vegMeal->getCost() << endl;
    delete vegMeal;

    cout << endl;
    cout << endl;

    Meal* nonVegMeal = mealBuilder->prepareNonVegMeal();
    cout << "Non-Veg Meal" << endl;
    nonVegMeal->showItems();
    cout << "Total Cost: " << nonVegMeal->getCost() << endl;
    delete nonVegMeal;

    delete mealBuilder;
}

/*
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost : 55.0


Non - Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost : 85.5
*/
```

[Abstract Factory (14/23)]

- Abstract Factory is a also called factory of factories
- Provide an interface for creating families of objects without specifying their concrete classes
- type : creational pattern



[Abstract Factory (14/23)]

```
class Client {
public:
    void draw() {
#ifdef LINUX
        Widget *w = new LinuxButton;
#else // WINDOWS
        Widget *w = new WindowsButton;
#endif
        w->draw();
        display_window_one();
        display_window_two();
    }

    void display_window_one() {
#ifdef LINUX
        Widget *w[] = {
            new LinuxButton,
            new LinuxMenu
        };
#else // WINDOWS
        Widget *w[] = {
            new WindowsButton,
            new WindowsMenu
        };
#endif
        w[0] ->draw();
        w[1]->draw();
    }

    void display_window_two() {
#ifdef LINUX
        Widget *w[] = {
            new LinuxMenu,
            new LinuxButton
        };
#else // WINDOWS
        Widget *w[] = {
            new WindowsMenu,
            new WindowsButton
        };
#endif
        w[0]->draw();
        w[1]->draw();
    }
};

int main() {
    Client *c = new Client();
    c->draw();
    return 0;
}
```

Without Abstract Factory

=

a lot of #ifdef/#else

[Abstract Factory (14/23)]

With Abstract Factory (1/2)

```
/**
* Abstract base product. It should define an interface
* which will be common to all products. Clients will
* work with products through this interface, so it
* should be sufficient to use all products.
*/
class Widget {
public:
    virtual void draw() = 0;
};

/**
* Concrete product family 1.
*/
class LinuxButton : public Widget {
public:
    void draw() { cout << "LinuxButton\n"; }
};
class LinuxMenu : public Widget {
public:
    void draw() { cout << "LinuxMenu\n"; }
};

/**
* Concrete product family 2.
*/
class WindowsButton : public Widget {
public:
    void draw() { cout << "WindowsButton\n"; }
};
class WindowsMenu : public Widget {
public:
    void draw() { cout << "WindowsMenu\n"; }
};
```

```
/**
* Abstract factory defines methods to create all
* related products.
*/
class Factory {
public:
    virtual Widget *create_button() = 0;
    virtual Widget *create_menu() = 0;
};

/**
* Each concrete factory corresponds to one product
* family. It creates all possible products of
* one kind.
*/
class LinuxFactory : public Factory {
public:
    Widget *create_button() {
        return new LinuxButton;
    }
    Widget *create_menu() {
        return new LinuxMenu;
    }
};

/**
* Concrete factory creates concrete products, but
* returns them as abstract.
*/
class WindowsFactory : public Factory {
public:
    Widget *create_button() {
        return new WindowsButton;
    }
    Widget *create_menu() {
        return new WindowsMenu;
    }
};
```

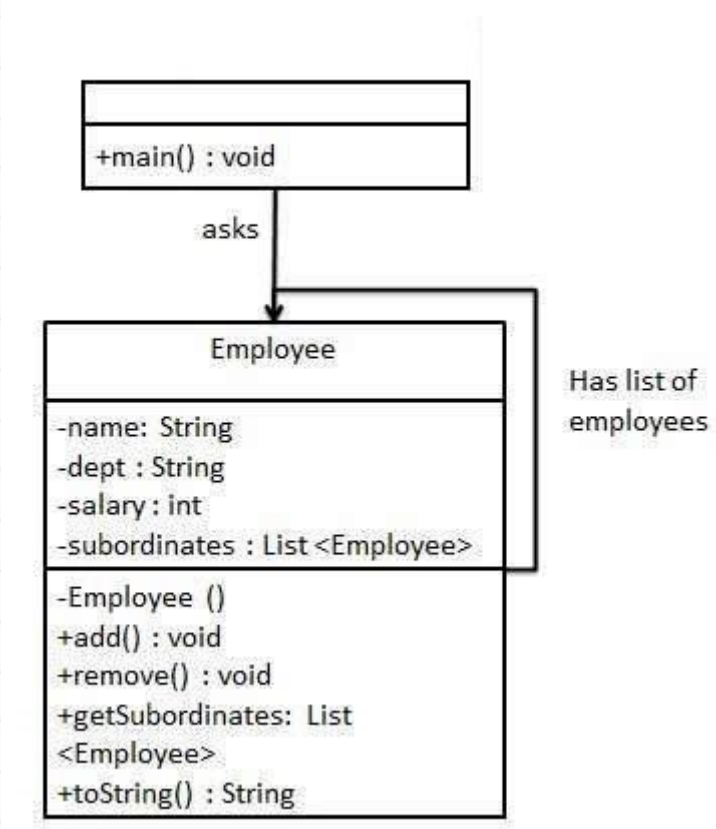
[Abstract Factory (14/23)]

With Abstract Factory (2/2) = easier to maintain

```
/**
/**
* Client receives a factory object from its creator.
*
* All clients work with factories through abstract
* interface. They don't know concrete classes of
* factories. Because of this, you can interchange
* concrete factories without breaking clients.
*
* Clients don't know the concrete classes of created
* products either, since abstract factory methods
* returns abstract products.
*/
class Client {
private:
    Factory *factory;

public:
    Client(Factory *f) {
        factory = f;
    }

    void draw() {
        Widget *w = factory->create_button();
        w->draw();
        display_window_one();
        display_window_two();
    }

    void display_window_one() {
        Widget *w[] = {
            factory->create_button(),
            factory->create_menu()
        };
        w[0]->draw();
        w[1]->draw();
    }
    void display_window_two() {
        Widget *w[] = {
            factory->create_menu(),
            factory->create_button()
        };
        w[0]->draw();
        w[1]->draw();
    }
};

/**
* Now the nasty switch statement is needed only once to
* pick and create a proper factory. Usually that's
* happening somewhere in program initialization code.
*/
int main() {
    Factory *factory;
#ifdef LINUX
    factory = new LinuxFactory;
#else // WINDOWS
    factory = new WindowsFactory;
#endif

    Client *c = new Client(factory);
    c->draw();

    return 0;
}
```

[Composite (15/23)]

- treat a group of objects in similar way as a single object
- type : structural pattern



[Composite (15/23)]

```
class Employee {
private:
    String name;
    String dept;
    int salary;
    list<Employee*> subordinates;

public:
    // constructor
    Employee(const String& name, const String& dept, int sal) {
        this->name = name;
        this->dept = dept;
        this->salary = sal;
    }

    virtual ~Employee()
    {
        for (Employee* e : subordinates)
        {
            delete e;
        }
        subordinates.clear();
    }

    void add(Employee* e) {
        subordinates.push_back(e);
    }

    void remove(Employee* e) {
        subordinates.remove(e);
    }

    const list<Employee*>& getSubordinates() {
        return subordinates;
    }

    void printEmployees() {
        cout << "Employee :[ Name : " << name.c_str() << ", dept : " << dept.c_str() << ", salary :" << salary << " ]" << endl;
        for (Employee* e : subordinates)
        {
            e->printEmployees();
        }
    }
};
```

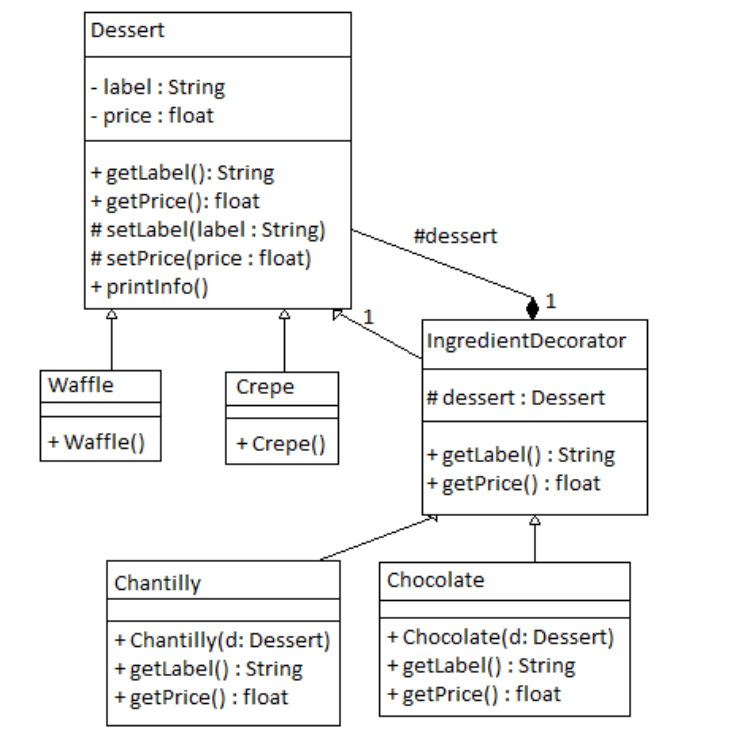
[Composite (15/23)]

```
void main(int argc, char ** argv) {

    Employee* CEO = new Employee("John", "CEO", 30000);

    Employee* headSales = new Employee("Robert", "Head Sales", 20000);

    Employee* headMarketing = new Employee("Michel", "Head Marketing", 20000);

    Employee* clerk1 = new Employee("Laura", "Marketing", 10000);
    Employee* clerk2 = new Employee("Bob", "Marketing", 10000);

    Employee* salesExecutive1 = new Employee("Richard", "Sales", 10000);
    Employee* salesExecutive2 = new Employee("Rob", "Sales", 10000);

    CEO->add(headSales);
    CEO->add(headMarketing);

    headSales->add(salesExecutive1);
    headSales->add(salesExecutive2);

    headMarketing->add(clerk1);
    headMarketing->add(clerk2);

    //print all employees of the organization
    CEO->printEmployees();

    // recursive deletion
    delete CEO;
}
```

```
/* Output :
    Employee :[Name:John, dept : CEO, salary : 30000]
    Employee :[Name:Robert, dept : Head Sales, salary : 20000]
    Employee :[Name:Richard, dept : Sales, salary : 10000]
    Employee :[Name:Rob, dept : Sales, salary : 10000]
    Employee :[Name:Michel, dept : Head Marketing, salary : 20000]
    Employee :[Name:Laura, dept : Marketing, salary : 10000]
    Employee :[Name:Bob, dept : Marketing, salary : 10000]
*/
```

[Decorator (16/23)]

- allows a user to add new functionality to an existing object without altering its structure
- type : structural pattern



[Decorator (16/23)]

```
// Create a "Abstract class" Dessert
class Dessert
{
private:
    String label;
    float price;

public:
    virtual ~Dessert() {}

    virtual const String getLabel()
    {
        return label;
    }

    virtual float getPrice()
    {
        return price;
    }

    void printInfo()
    {
        cout << getLabel().c_str() << " : " << ((int)getPrice())<< "."<< (((int)(100*getPrice()))%100) << " dollars" << endl;
    }

protected:
    void setLabel(const String& label)
    {
        this->label = label;
    }

    void setPrice(float price)
    {
        this->price = price;
    }
};
```

[Decorator (16/23)]

```
// 2 implementations of Dessert
class Waffle : public Dessert
{
public:
    Waffle()
    {
        setLabel("Waffle");
        setPrice(1.80);
    }
};

class Crepe : public Dessert
{
public:
    Crepe()
    {
        setLabel("Crepe");
        setPrice(1.50);
    }
};

// Abstract Class abstraite IngredientDecorator that inherits from Dessert
class IngredientDecorator: public Dessert
{
protected:
    Dessert* dessert;// Dessert sur leuquel on applique l'ingrédient.

    // On oblige les ingrédients à implémenter la méthode getLibelle().
    virtual const String getLabel() = 0;
    // On oblige les ingrédients à implémenter la méthode getPrix().
    virtual float getPrice() = 0;
};
```

[Decorator (16/23)]

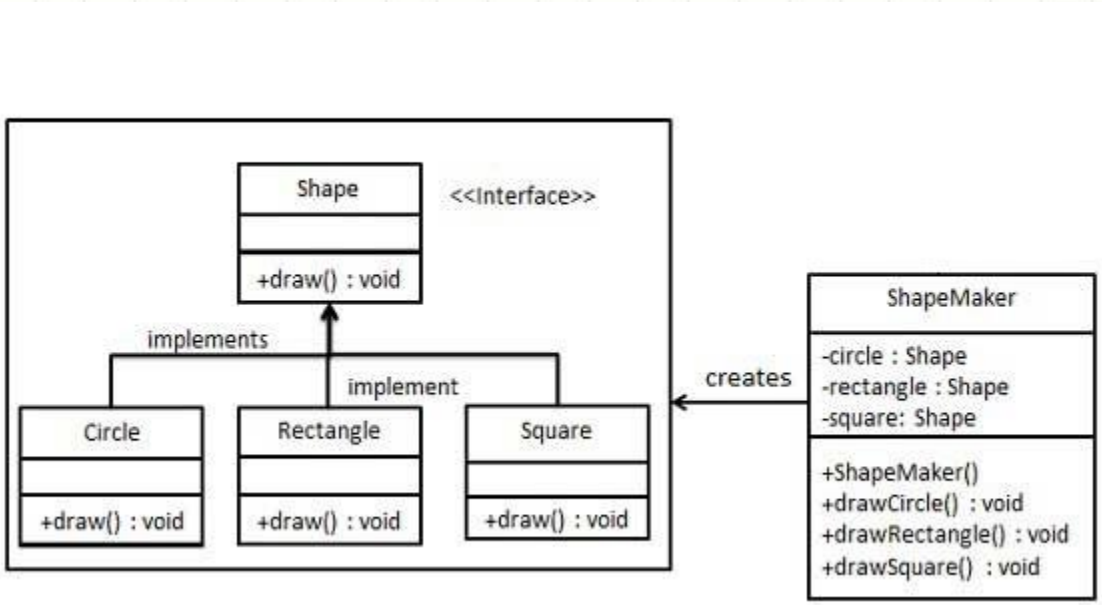
```
// Implémentation of Chantilly and Chocolat classes
class Chantilly : public IngredientDecorator
{
public:

    Chantilly(Dessert* d)
    {
        dessert = d;
    }

    // Show the label of the dessert and add the label of the chantilly ingredient
    const String getLabel()
    {
        return dessert->getLabel() + ", chantilly";
    }

    // Add the price of the dessert and the price of the chantilly ingredient
    float getPrice()
    {
        return dessert->getPrice() + 0.50;
    }
};

class Chocolate : public IngredientDecorator
{
public:
    Chocolate(Dessert* d)
    {
        dessert = d;
    }

    // Show the label of the dessert and add the label of the chocolate ingredient
    const String getLabel()
    {
        return dessert->getLabel() + ", chocolate";
    }

    // Add the price of the dessert and the price of the chocolate ingredient
    float getPrice()
    {
        return dessert->getPrice() + 0.20;
    }
};
```

[Decorator (16/23)]

```
void main(int argc, char ** argv)
{
    // Create and show a Waffle with chocolate
    Dessert* waffle = new Waffle();
    waffle->printInfo();
    Dessert* waffleWithChocolate = new Chocolate(waffle);
    waffleWithChocolate->printInfo();

    // Create and show a Waffle with chocolate and chantilly
    Dessert* crepe = new Crepe();
    crepe->printInfo();
    Dessert* crepeWithChocolate = new Chocolate(crepe);
    crepeWithChocolate->printInfo();
    Dessert* crepeWithChantilly = new Chantilly(crepe);
    crepeWithChantilly->printInfo();

    delete waffle;
    delete waffleWithChocolate;
    delete crepe;
    delete crepeWithChocolate;
    delete crepeWithChantilly;
}

/* Output :
Waffle: 1.80 dollars
Waffle, chocolate : 2.0 dollars
Crepe : 1.50 dollars
Crepe, chocolate : 1.70 dollars
Crepe, chantilly : 2.0 dollars
*/
```

[Facade (17/23)]

- hides the complexities of the system and provides an interface to the client
- type : structural pattern



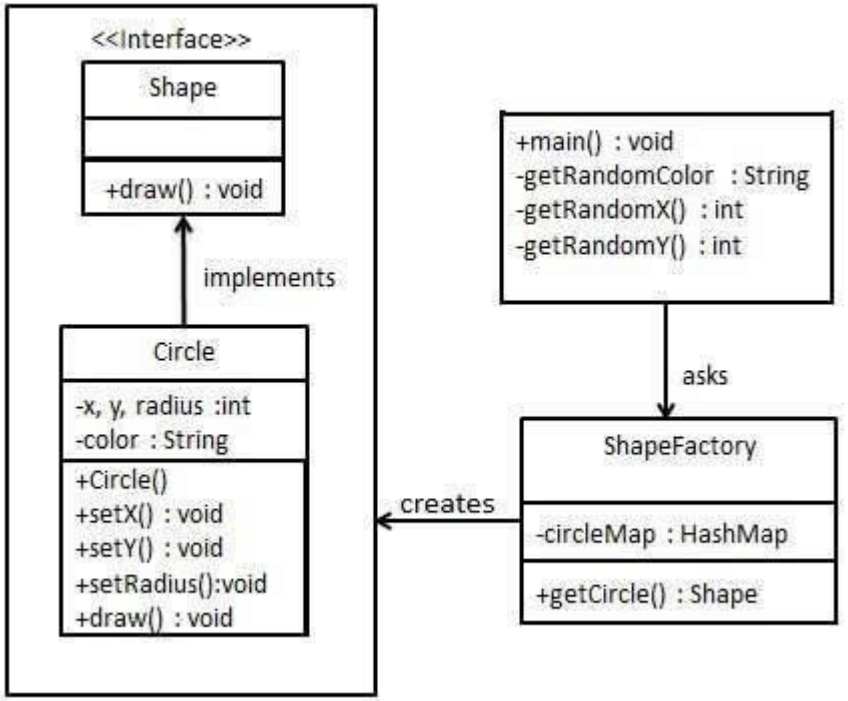
[Facade (17/23)]

```
// Create an "interface"
class Shape
{
public:
    virtual ~Shape() {}
    virtual void draw() = 0;
};

// Create concrete classes implementing the same interface
class Circle : public Shape
{
public:
    Circle() {}
    virtual ~Circle() { }
    virtual void draw() { cout << "draw circle" << endl; }
};

class Rectangle : public Shape
{
public:
    Rectangle() {}
    virtual ~Rectangle() {}
    virtual void draw() { cout << "draw rectangle" << endl; }
};

class Square : public Shape
{
public:
    Square() {}
    virtual ~Square() {}
    virtual void draw() { cout << "draw square" << endl; }
};
```

```
// Create a facade class
class ShapeMaker {
private:
    Shape* circle;
    Shape* rectangle;
    Shape* square;

public:
    ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
    }

    virtual ~ShapeMaker()
    {
        delete circle;
        delete rectangle;
        delete square;
    }

    void drawCircle() {
        circle->draw();
    }
    void drawRectangle() {
        rectangle->draw();
    }
    void drawSquare() {
        square->draw();
    }
};

// Use the facade to draw various types of shapes.
void main(int argc, char ** argv) {
    ShapeMaker* shapeMaker = new ShapeMaker();

    shapeMaker->drawCircle();
    shapeMaker->drawRectangle();
    shapeMaker->drawSquare();

    delete shapeMaker;
}
```

[Flyweight (18/23)]

- used to reduce the number of instances created
- goals : decrease memory footprint and increase performance
- type : structural pattern



[Flyweight (18/23)]

```
// Create an "interface"
class Shape
{
public:
    virtual ~Shape() {}
    virtual void draw() = 0;
};

// Create concrete classes implementing the same interface
class Circle : public Shape
{
private:
    String color;
    int x;
    int y;
    int radius;

public:
    Circle(const String &color)
    {
        this->x = 0;
        this->y = 0;
        this->radius = 0;
        this->color = color;
    }

    virtual ~Circle() { }

    void setX(int x) {
        this->x = x;
    }

    void setY(int y) {
        this->y = y;
    }

    void setRadius(int radius) {
        this->radius = radius;
    }

    virtual void draw() {
        cout << "Circle: Draw() [Color : " << color.c_str()
            << ", x : " << x << ", y :" << y
            << ", radius :" << radius << endl;
    }
};
```

```
// Create a factory to generate object of concrete class based on given information
class ShapeFactory {
private:
    static map<String, Shape*> circleMap;

public:
    static Shape* getCircle(const String& color) {
        Circle* circle = (Circle*)circleMap[color];
        if (circle == nullptr) {
            circle = new Circle(color);
            circleMap[color] = circle;
            cout << "Creating circle of color : " << color.c_str() << endl;
        }
        return circle;
    }

    static void clean()
    {
        for (auto iter : circleMap)
        {
            delete iter.second;
        }
        circleMap.clear();
    }
};
map<String, Shape*> ShapeFactory::circleMap;
```

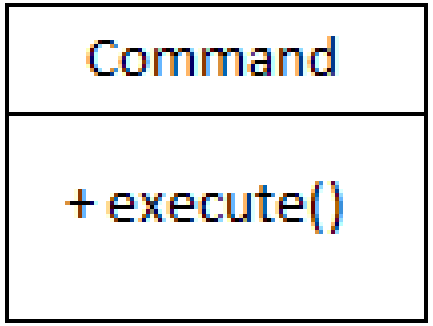
[Flyweight (18/23)]

```
int random(int max)
{
    return rand() % max;
}

const String& getRandomColor() {
    static vector<String> colors = { "Red", "Green", "Blue", "White", "Black" };
    return colors[random(colors.size())];
}

int getRandomX() {
    return random(100);
}

int getRandomY() {
    return random(100);
}

void main(int argc, char ** argv) {

    for (int i = 0; i < 20; ++i) {
        Circle* circle = (Circle*)ShapeFactory::getCircle(getRandomColor());
        circle->setX(getRandomX());
        circle->setY(getRandomY());
        circle->setRadius(100);
        circle->draw();
    }

    ShapeFactory::clean();
}
```

```
/* Output:
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
*/
```

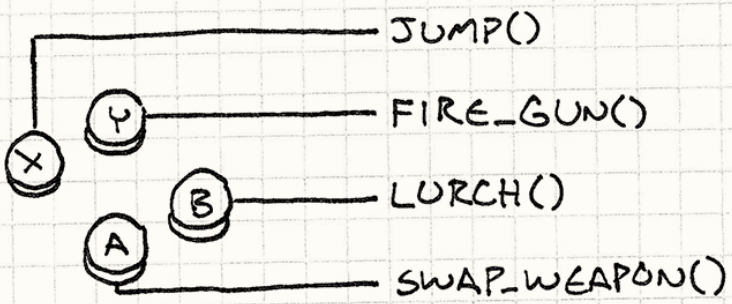
[Command (19/23)]

- it's an object-oriented callback
- example: can be attached/detached to a button of a gamepad or of a file menu
- type : behavioral pattern



[Command (19/23)]

Without Command Pattern (schema)



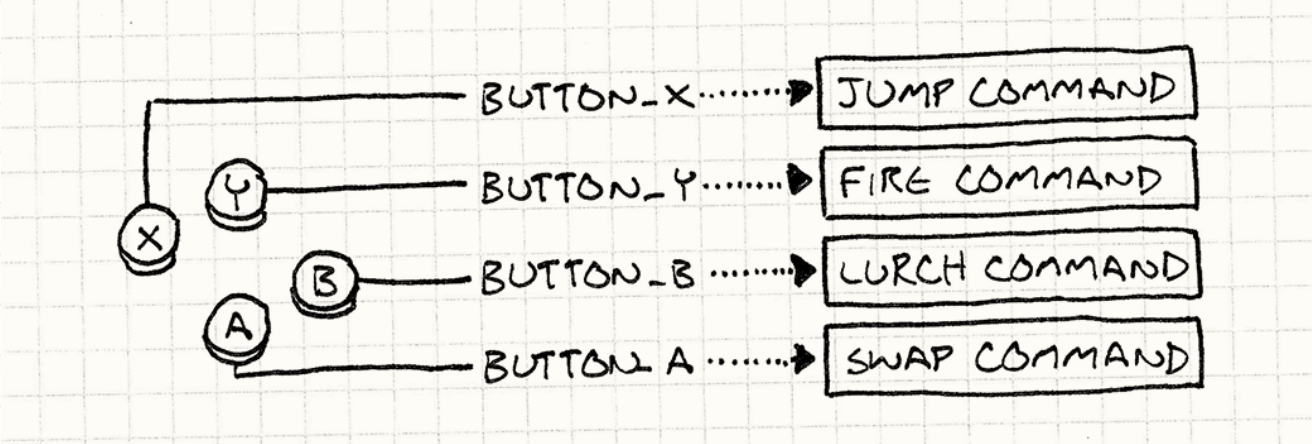
[Command (19/23)]

```
enum BUTTON
{
    BUTTON_X = 0,
    BUTTON_Y,
    BUTTON_A,
    BUTTON_B,
    BUTTON_COUNT
};

class Hero
{
public:
    void jump() {}
    void fireGun() {}
    void swapWeapon() {}
    void lurchIneffectively() {}
};

class InputHandler
{
public:

    void handleInput(Hero * hero)
    {
        if (isPressed(BUTTON_X)) hero->jump();
        else if (isPressed(BUTTON_Y)) hero->fireGun();
        else if (isPressed(BUTTON_A)) hero->swapWeapon();
        else if (isPressed(BUTTON_B)) hero->lurchIneffectively();
    }

    bool isPressed(enum BUTTON button)
    {
        return buttonStates[button];
    }

private:
    bool buttonStates[BUTTON_COUNT];
};
```

Without Command Pattern (code)

[Command (19/23)]

With Command Pattern (schema)



[Command (19/23)]

```
enum BUTTON
{
    BUTTON_X = 0,
    BUTTON_Y,
    BUTTON_A,
    BUTTON_B,
    BUTTON_COUNT
};


class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};

class Hero
{
public:
    void jump() {}
    void fireGun() {}
    void swapWeapon() {}
    void lurchIneffectively() {}
};

class JumpCommand : public Command
{
public:
    JumpCommand(Hero* h) :hero(h) {}
    virtual void execute() { hero->jump(); }
private:
    Hero* hero;
};

class FireCommand : public Command
{
public:
    FireCommand(Hero* h) :hero(h) {}
    virtual void execute() { hero->fireGun(); }
private:
    Hero* hero;
};

class SwapWeaponCommand : public Command
{
public:
    SwapWeaponCommand(Hero* h) :hero(h) {}
    virtual void execute() { hero->swapWeapon(); }
private:
    Hero* hero;
};
```

With Command Pattern 1/2 (code)

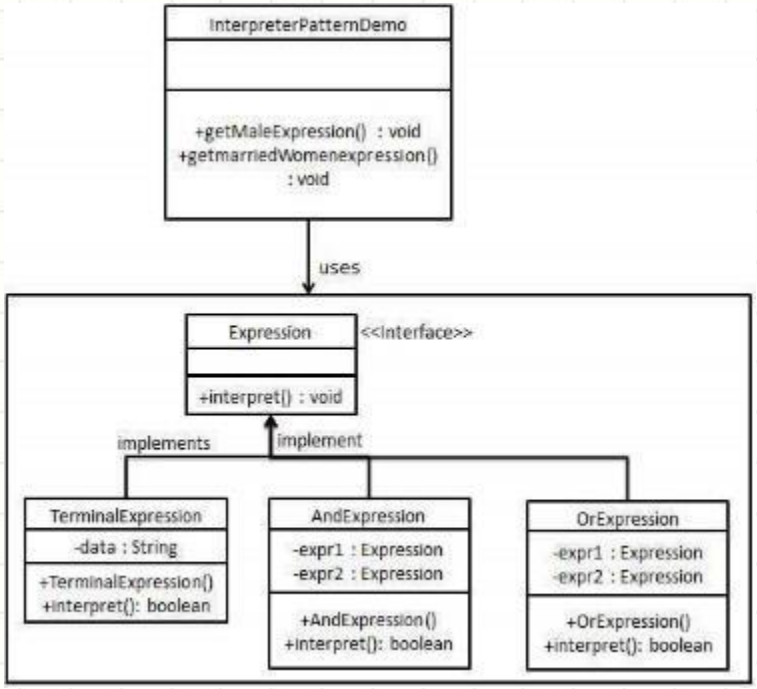
[Command (19/23)]

```
class LurchIneffectivelyCommand : public Command
{
public:
    LurchIneffectivelyCommand(Hero* h) :hero(h) {}
    virtual void execute() { hero->lurchIneffectively(); }

private:
    Hero* hero;
};


class InputHandler
{
public:

    InputHandler(Hero * h)
    {
        buttonX = new JumpCommand(h);
        buttonY = new FireCommand(h);
        buttonA = new SwapWeaponCommand(h);
        buttonB = new LurchIneffectivelyCommand(h);
    }

    ~InputHandler()
    {
        delete buttonX;
        delete buttonY;
        delete buttonA;
        delete buttonB;
    }
    void handleInput(Hero * hero)
    {
        if (isPressed(BUTTON_X)) buttonX->execute();
        else if (isPressed(BUTTON_Y)) buttonY->execute();
        else if (isPressed(BUTTON_A)) buttonA->execute();
        else if (isPressed(BUTTON_B)) buttonB->execute();
    }

    bool isPressed(enum BUTTON button)
    {
        return buttonStates[button];
    }

private:
    bool buttonStates[BUTTON_COUNT];

    Command* buttonX;
    Command* buttonY;
    Command* buttonA;
    Command* buttonB;
};
```

With Command Pattern 2/2 (code)

[Interpreter (20/23)]

- provides a way to evaluate a language grammar or an expression
- example : used in SQL parsing
- type : behavioral pattern



[Interpreter (20/23)]

```
// Create an expression "interface".
class Expression {
public:
    virtual ~Expression() {}
    virtual bool interpret(const String &context) = 0;
};


// Create concrete classes implementing the above interface.
class TerminalExpression : public Expression {

private:
    String data;

public:
    TerminalExpression(const String & data)
    {
        this->data = data;
    }

    ~TerminalExpression() {}

    bool interpret(const String & context)
    {
        if (context.find(data) >=0)
        {
            return true;
        }
        return false;
    }

};
```

[Interpreter (20/23)]

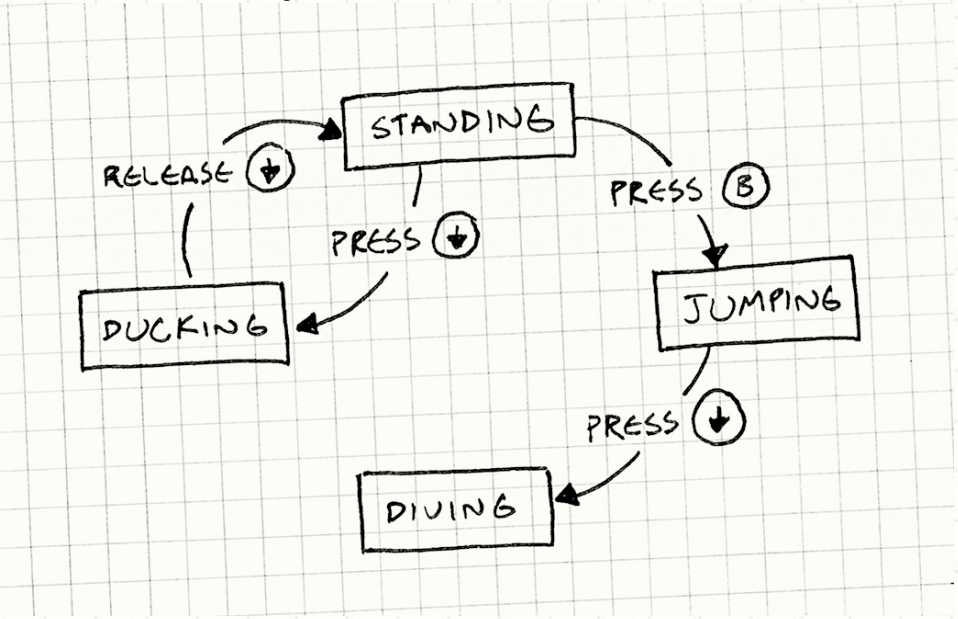
```
class OrExpression: public Expression {
private:
    Expression* expr1 = nullptr;
    Expression* expr2 = nullptr;

public:
    OrExpression(Expression* expr1, Expression* expr2) {
        this->expr1 = expr1;
        this->expr2 = expr2;
    }

    ~OrExpression() { delete expr1; delete expr2;}

    bool interpret(const String& context) {
        return expr1->interpret(context) || expr2->interpret(context);
    }
};

class AndExpression : public Expression {
private:
    Expression* expr1 = nullptr;
    Expression* expr2 = nullptr;

public:
    AndExpression(Expression* expr1, Expression* expr2) {
        this->expr1 = expr1;
        this->expr2 = expr2;
    }

    ~AndExpression() { delete expr1; delete expr2; }

    bool interpret(const String& context) {
        return expr1->interpret(context) && expr2->interpret(context);
    }
};
```

[Interpreter (20/23)]

```
class InterpreterPatternDemo {
public:
    //Rule: Robert and John are male
    static Expression* getMaleExpression() {
        Expression* robert = new TerminalExpression("Robert");
        Expression* john = new TerminalExpression("John");
        return new OrExpression(robert, john);
    }

    //Rule: Julie is a married women
    static Expression* getMarriedWomanExpression() {
        Expression* julie = new TerminalExpression("Julie");
        Expression* married = new TerminalExpression("Married");
        return new AndExpression(julie, married);
    }
};
int main()
{
    Expression* isMale = InterpreterPatternDemo::getMaleExpression();
    Expression* isMarriedWoman = InterpreterPatternDemo::getMarriedWomanExpression();

    cout << std::noboolalpha  << "John is male? " << std::boolalpha << isMale->interpret("John") << endl;
    cout << "Julie is a married women? " << std::boolalpha << isMarriedWoman->interpret("Married Julie") << endl;

    delete isMale;
    delete isMarriedWoman;

    return 0;
}

// Verify the output.
// John is male ? true
// Julie is a married women ? true
```

[State (21/23)]

- encapsulate varying behavior for the same object
- cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements (if/else or switch)
- goal : improve maintainability
- type : behavioral pattern



[State (21/23)]

```
class Heroine
{
public:
    void setGraphics(Animate animate) {}
    void handleInput(Input input);
    double yVelocity_;
    bool isJumping_;
    bool isDucking_;
};
//^spaghetti-5
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_ && !isDucking_)
        {
            // Jump...
        }
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            isDucking_ = true;
            setGraphics(IMAGE_DUCK);
        }
        else
        {
            isJumping_ = false;
            setGraphics(IMAGE_DIVE);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        if (isDucking_)
        {
            // Stand...
        }
    }
}
//^spaghetti-5
```

without state pattern

=

spaghetti code

[State (21/23)]

```
//^state-switch
void Heroine::handleInput(Input input)
{
    switch (state_)
    {
    case STATE_STANDING:
        if (input == PRESS_B)
        {
            state_ = STATE_JUMPING;
            yVelocity_ = JUMP_VELOCITY;
            setGraphics(IMAGE_JUMP);
        }
        else if (input == PRESS_DOWN)
        {
            state_ = STATE_DUCKING;
            setGraphics(IMAGE_DUCK);
        }
        break;

    case STATE_JUMPING:
        if (input == PRESS_DOWN)
        {
            state_ = STATE_DIVING;
            setGraphics(IMAGE_DIVE);
        }
        break;

    case STATE_DUCKING:
        if (input == RELEASE_DOWN)
        {
            state_ = STATE_STANDING;
            setGraphics(IMAGE_STAND);
        }
        break;
        //^omit
    case STATE_DIVING:
        break;
        //^omit
    }
}
//^state-switch

//^switch-update
void Heroine::update()
{
    if (state_ == STATE_DUCKING)
    {
        chargeTime_++;
        if (chargeTime_ > MAX_CHARGE)
        {
            superBomb();
        }
    }
}
//^switch-update
```

With Finite State Machine

=

Big Swicth

[State (21/23)]

```
class Heroine
{
    friend class JumpingState;
public:
    void setGraphics(Animate animate) {}
    void changeState(HeroineState* state) {}
private:
    HeroineState* state_;
};

//^heroine-static-states
class HeroineState
{
public:
    static StandingState standing;
    static DuckingState ducking;
    static JumpingState jumping;
    static DivingState diving;

    virtual void handleInput(Heroine& heroine, Input input) {}

    // Other code...
};
//^heroine-static-states

class StandingState : public HeroineState {};
class DuckingState : public HeroineState {};

class JumpingState : public HeroineState {
public:
    void handleInput(Heroine& heroine, Input input)
    {
        //^jump
        if (input == PRESS_B)
        {
            heroine.state_ = &HeroineState::jumping;
            heroine.setGraphics(IMAGE_JUMP);
        }
        //^jump
    }
};
class DivingState : public HeroineState {};
```

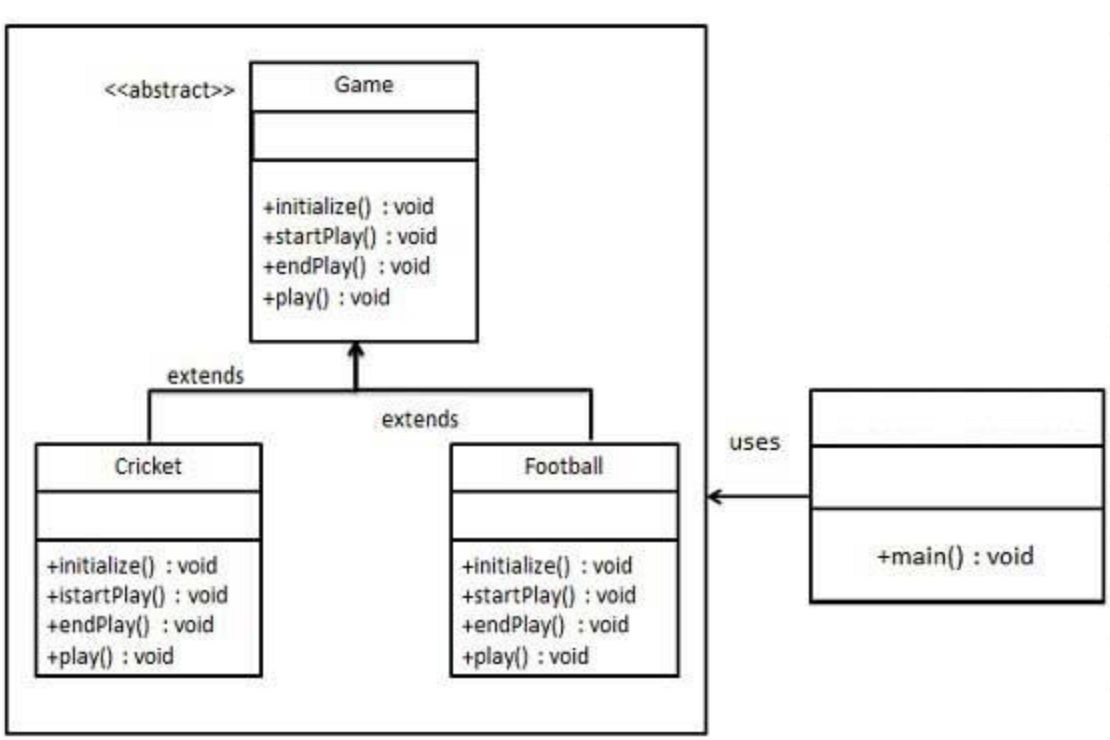
With State Pattern

=

“More maintainable”

[Template (22/23)]

- a template method calls virtual pure
- type : behavioral pattern



[Template (22/23)]

```
// Create an abstract class with a template method
class Game {
private:
    virtual void initialize() = 0;
    virtual void startPlay() = 0;
    virtual void endPlay() = 0;

public:
    //template method
    void play() {

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
};
```

[Template (22/23)]

```
// Create concrete classes extending the above class
class Cricket: public Game {
private:
    void endPlay() {
        cout << "Cricket Game Finished!" << endl;
    }

    void initialize() {
        cout << "Cricket Game Initialized! Start playing." << endl;
    }

    void startPlay() {
        cout << "Cricket Game Started. Enjoy the game!" << endl;
    }
};

class Football: public Game {
private:
    void endPlay() {
        cout << "Football Game Finished!" << endl;
    }

    void initialize() {
        cout << "Football Game Initialized! Start playing." << endl;
    }

    void startPlay() {
        cout << "Football Game Started. Enjoy the game!" << endl;
    }
};
```

[Template (22/23)]

```
int main(int argc, char ** argv)
{
    Game* game = new Cricket();
    game->play();
    delete game;
    cout << endl;

    game = new Football();
    game->play();
    delete game;

    return 0;
}

/* output :

Cricket Game Initialized!Start playing.
Cricket Game Started.Enjoy the game!
Cricket Game Finished!

Football Game Initialized!Start playing.
Football Game Started.Enjoy the game!
Football Game Finished! */
```

[Visitor (23/23)]

- visitors are used to implement type-testing without sacrificing type-safety
- type : behavioral pattern

[Visitor (23/23)]

```
void sort()
{
    list<Fruit*> fruits = getFruits();

    list<Orange*> oranges;
    list<Apple*> apples;
    list<Banana*> bananas;

    for(Fruit* fruit : fruits)
    {
        if (Orange* orange = dynamic_cast<Orange*>(fruit))
            oranges.push_back(orange);
        else if (Apple* apple = dynamic_cast<Apple*>(fruit))
            apples.push_back(apple);
        else if (Banana* banana = dynamic_cast<Banana*>(fruit))
            bananas.push_back(banana);
    }

    // result
    cout << "Oranges count " << oranges.size() << endl;
    cout << "Apples count " << apples.size() << endl;
    cout << "Bananas count " << bananas.size() << endl;
}
```

Without Visitor Pattern:

- Works but ugly code
- Catch type errors until runtime
- Not maintainable

[Visitor (23/23)]

```
class IFruitVisitor
{
public:
    virtual void Visit(Orange* fruit) = 0;
    virtual void Visit(Apple* fruit) = 0;
    virtual void Visit(Banana* fruit) = 0;
};

class Fruit { public: virtual void Accept(IFruitVisitor* visitor) = 0; };
class Orange : public Fruit { public: void Accept(IFruitVisitor* visitor) { visitor->Visit(this); } };
class Apple : public Fruit { public: void Accept(IFruitVisitor* visitor) { visitor->Visit(this); } };
class Banana : public Fruit { public: void Accept(IFruitVisitor* visitor) { visitor->Visit(this); } };

class FruitPartitioner : public IFruitVisitor
{
public:
    list<Orange*> Oranges;
    list<Apple*> Apples;
    list<Banana*> Bananas;


    FruitPartitioner() {}

    void Visit(Orange* fruit) { Oranges.push_back(fruit); }
    void Visit(Apple* fruit) { Apples.push_back(fruit); }
    void Visit(Banana* fruit) { Bananas.push_back(fruit); }
};
```

With Visitor Pattern:

- Relatively clean
- Type-safety (compile time)
- Maintainability (for adding/removing)

[Visitor (23/23)]

Usage:

```
int main(int argc, char ** argv)
{
    list<Fruit*> fruits = {
        new Orange(), new Apple(), new Banana(),
        new Banana(), new Banana(), new Orange() };

    FruitPartitioner partitioner;
    for (Fruit* fruit : fruits)
    {
        fruit->Accept(&partitioner);
    }

    // result
    cout << "Oranges count "<< partitioner.Oranges.size() <<endl;
    cout << "Apples count " << partitioner.Apples.size() << endl;
    cout << "Bananas count " << partitioner.Bananas.size() << endl;

    // clear fruits
    for (Fruit* fruit : fruits)
    {
        delete fruit;
    }
    fruits.clear();

    return 0;
}
/* output :
Oranges count 2
Apples count 1
Bananas count 3
*/
```

[To Go Further]

BOOKS :

- "Game Programming Patterns" by Bob Nystrom
- "Design Patterns for Embedded Systems in C" by Bruce Powel Douglass

CONCLUSION

Ressources

- https://sourcemaking.com/design_patterns
- https://www.tutorialspoint.com
- http://gameprogrammingpatterns.com/
- http://design-patterns.fr
- http://stackoverflow.com