Visitor and Factory Patterns

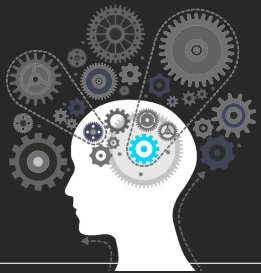
Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will be able to:

- Have a good understanding about the design and use of the following patterns:
  - Visitor
  - Factory Method
  - Abstract Factory
- Understand the design principles that these patterns enforce/work with
- Be capable of implementing these patterns in your own code



Visitor Pattern

CS 2263

Visitor Pattern

Idaho State University | Computer Science

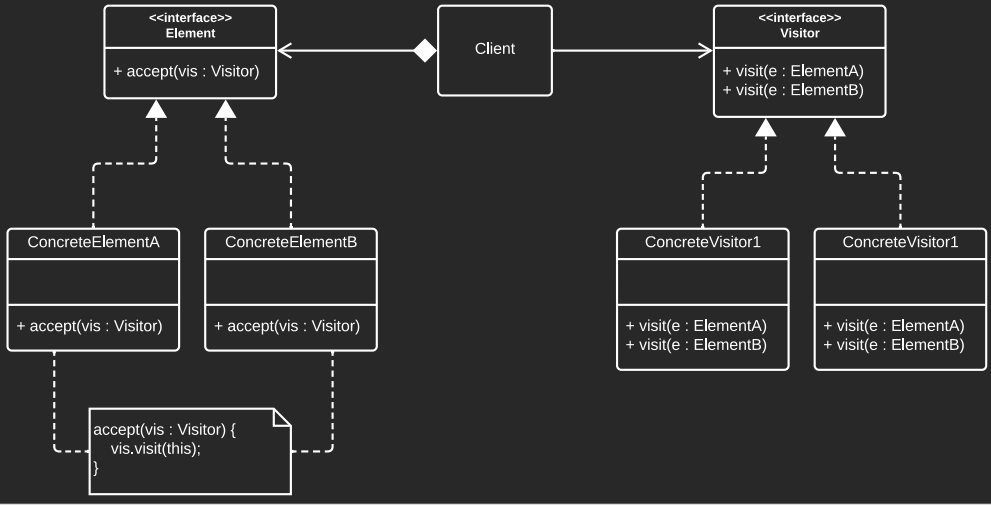
Intent

The Visitor Pattern enables us to add new behavior to existing classes in a fixed class hierarchy without changing this hierarchy

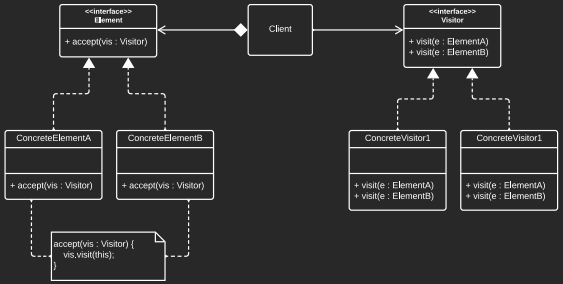
Solution Idea

Represent additional operations to be performed on the elements of an object structure (additional feature) as objects (of type Visitor)

Structure


<<interface>>
Element
+ accept(vis : Visitor)
Client
<<interface>>
Visitor
+ visit(e : ElementA)
+ visit(e : ElementB)
ConcreteElementA
+ accept(vis : Visitor)
ConcreteElementB
+ accept(vis : Visitor)
ConcreteVisitor1
+ visit(e : ElementA)
+ visit(e : ElementB)
ConcreteVisitor1
+ visit(e : ElementA)
+ visit(e : ElementB)
accept(vis : Visitor) {
vis.visit(this);
}

Structure



- The `Visitor` interface declares a `visit` method per element type in the object structure
- A `Visitor` interface describes how to "treat" the element types
- Concrete visitor classes implement the interface specifically, i.e., treat elements differently

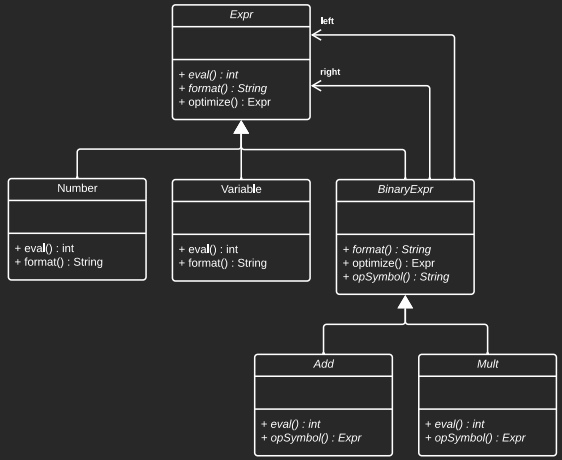
- A concrete visitor class corresponds to a particular feature to be added to the object structure
- Elements in the object structure provide the method `accept(visitor)`
- On being asked to `accept` a visitor passed to it as a parameter, an element asks the visitor to `visit` it.

Case Study: Arithmetic Expressions

Idaho State University | Computer Science

Requirements:

- A library for (arithmetic) expressions must provide different functionality for:
  - Formatting an expression to a string
  - Computing the value of an expression
  - Optimizing an expression
- The library must be extensible with new functionality
  - Generate code for different machines
  - Various refactorings, e.g., rename variables
- The library must be extensible with new kinds of expressions

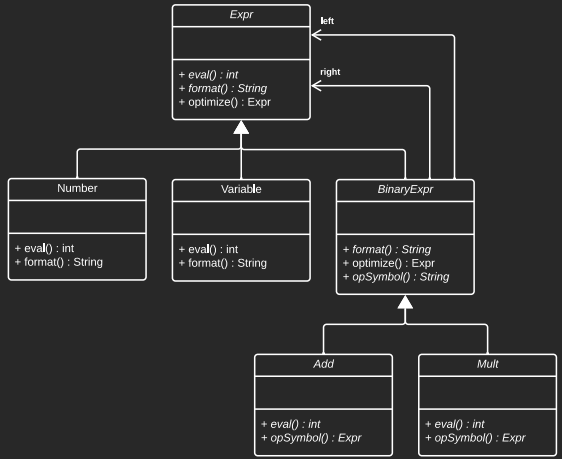


Case Study: Arithmetic Expressions

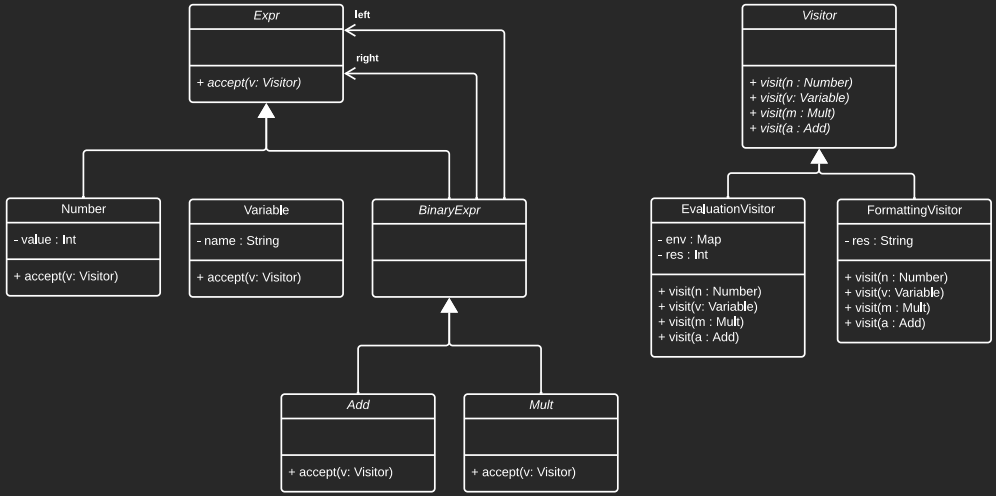
Idaho State University | Computer Science

Design Issues:

- Impossible to reuse part of library functionality
- Changing one feature can destabilize other features (SDP)
- New features cannot be incrementally added (OCP)



Visitor Based Design



Advantages of the Visitor

Idaho State University | Computer Science

- **New operations are easy to add** without changing element classes (add a new concrete visitor).
  - Different concrete elements do not have to implement their part of a particular algorithm
- Related behavior focused in a single concrete visitor
- **Visiting across hierarchies:** Visited classes are not forced to share a common base class
- **Accumulating state:** Visitors can accumulate state as they visit each element, thus, encapsulating the algorithm and all its data.

Issues of the Visitor

- If we add a new type to a hierarchy
  - we need to modify all the visitors to handle the type.
- Many visitors will have empty methods to comply to the interface
- Sometimes data structures are extended, but it's optional to process extensions

Summary

- Visitor brings functional-style decomposition to OO designs
- Use Visitor for Stable element hierarchies
  - Visitor works well in data hierarchies where new elements are never or at least not very often added.
- Do not use it, if new elements are a likely change
- Visitor only makes sense if we have to add new operations often! In this case Visitor closes our design against these changes.

Factory Method Pattern

CS 2263

The Problem With “New”

- Each time we invoke the “new” command to create a new object, we violate the “Code to an Interface” design principle.
- Example:
  - `Duck duck = new DecoyDuck()`
- Even though our variable’s type is set to an “interface”, in this case “Duck”, the class that contains this statement depends on “DecoyDuck”
- In addition, if you have code that checks a few variables and instantiates a particular type of class based on the state of those variables, then the containing class depends on each referenced concrete class

```
if (hunting) {
   return new DecoyDuck();
} else {
   return new RubberDuck();
}
```

Obvious Problem: needs to be recompiled each time a dependency changes

- add new classes –> change this code
- remove existing classes –> change this code

PizzaStore Example

Idaho State University | Computer Science

- We have a pizza store program that wants to separate the process of creating a pizza with the process of preparing/ordering a pizza
- Initial code: mixed the two processes

```
public class PizzaStore {
    Pizza orderPizza(String type) {
        Pizza pizza;
        // Creation code
        if (type == "cheese") {
           pizza = new CheesePizza();
        } else if (type == "greek") {
           pizza = new GreekPizza();
        } else if (type == "pepperoni") {
           pizza = new PepperoniPizza();
        }
        // Coding to Interface
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

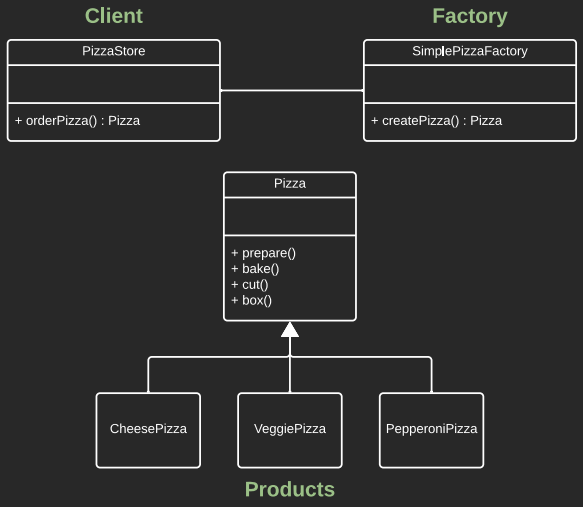
Encapsulate Creation Code

- A simply way to encapsulate this code is to put it in a separate class
  - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code.

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        if (type == "cheese") {
            return new CheesePizza();
        } else if (type == "greek") {
            return new GreekPizza();
        } else if (type == "pepperoni") {
            return new PepperoniPizza();
        }
    }
}
```

Class Diagram of New Solution



Factory Method

- To demonstrate the factory method pattern, the pizza store example evolves
  - to include the notion of different franchises
  - that exist in different parts of the country (California, New York, and Chicago)

- Each franchise will need its own factory to create pizzas that match the proclivities of the locals
  - However, we want to retain the preparation process that has made `PizzaStore` such a great success

Factory Method

- The Factory Method Design Pattern allows you to do this by
  - placing abstract, “code to an interface” code in a superclass
  - placing object creation code in a subclass
- `PizzaStore` becomes an abstract class with an abstract create Pizza() method
- We then create subclasses that override `createPizza()` for each region.

New PizzaStore Class

```
public abstract class PizzaStore {

    protected abstract createPizza(String type);

    public Pizza orderPizza(String type) {

        Pizza pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Beautiful Abstract Base Class!

Factory Method

- This class is a (very simple) OO framework. The framework provides one service “prepare pizza”.
- The framework invokes the `createPizza()` factory method to create a pizza that it can prepare using a well-defined, consistent process.
- A “client” of the framework will subclass this class and provide an implementation of the `createPizza()` method.
- Any dependencies on concrete “product” classes are encapsulated in the subclass.

New York Pizza Store

Idaho State University | Computer Science

```
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
        if (type == "cheese") {
            return new NYCheesePizza();
        } else if (type == "greek") {
            return new NYGreekPizza();
        } else if (type == "pepperoni") {
            return new NYPepperoniPizza();
        }
        return null;
    }
}
```

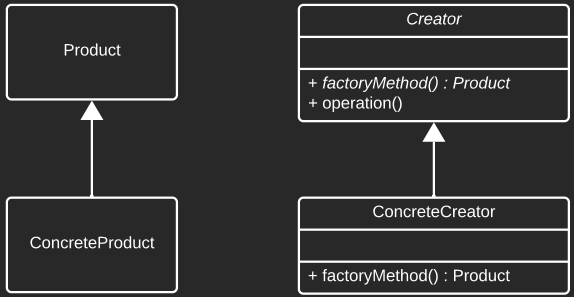
Nice and Simple

- If you want a NY-Style Pizza, you create an instance of this class and call `orderPizza()` passing in the type.
- The subclass makes sure that the pizza is created using the correct style.
- If you need a different style, create a new subclass.

Factory Method: Definition and Structure

Idaho State University | Computer Science

- The factory method design pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Factory Method leads to the creation of parallel class hierarchies; `ConcreteCreators` produce instances of `ConcreteProducts` that are operated on by Creator's via the Product interface

Abstract Factory Pattern

CS 2263

Dependency Inversion Principle

- Factory Method is one way of following the dependency inversion principle
  - "Depend upon abstractions. Do not depend upon concrete classes."
- Normally "high-level" classes depend on "low-level" classes;
  - Instead, they BOTH should depend upon an abstract interface
- `DependentPizzaStore` depends on eight concrete `Pizza` subclasses
  - `PizzaStore`, however, depends on the `Pizza` interface, as do the Pizza subclasses
- In this design, `PizzaStore` (the high-level class) no longer depends on the `Pizza` subclasses (the low level classes); they both depend on the abstraction "Pizza". Nice.

Dependency Inversion Principle: How To?

- To achieve the dependency inversion principle in your own designs, follow these GUIDELINES:
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of its base class

- These are guidelines because if you were to blindly follow these instructions you would never produce a system that could be compiled or executed
  - Instead use them as instructions to help optimize your design

- And remember, not only should low-level classes depend on abstractions, but high-level classes should to… this is the very embodiment of “code to an interface”

Demonstration

- Lets look at some code
  - The `FactoryMethod` directory of src.zip file contains an implementation of the pizza store using the factory method design pattern
    - It even includes a file called `DependentPizzaStore.java` that shows how the code would be implemented without using this pattern
  - `DependentPizzaStore` is dependent on 8 different concrete classes and 1 abstract interface (Pizza)
  - `PizzaStore` is dependent on just the Pizza abstract interface (nice!)
    - Each of its subclasses is only dependent on 4 concrete classes.
    - furthermore, they shield the superclass from these dependencies

Moving On

- The factory method approach to the pizza store is a big success allowing our company to create multiple franchises across the country quickly and easily
  - But, bad news, we have learned that some of the franchises
    - while following our procedures (the abstract code in `PizzaStore` forces them to)
    - are skimping on ingredients in order to lower costs and increase margins
  - Our company's success has always been dependent on the use of fresh quality ingredients
    - so "Something Must Be Done!"

Abstract Factory to the Rescue!

- We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process
  - Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used
  - But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises
    - They'll have to come up with some other way to lower costs.

First, We need a Factory Interface

Idaho State University | Computer Science

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

Note the introduction of more abstract classes:

`Dough`, `Sauce`, `Cheese`, etc.

A Region-Specific Factory

Idaho State University | Computer Science

```
public class ChicagoPizzaIngredientFactory
        implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThickCurstDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
           new Spinach(), new Eggplant() };
        return veggies;
    }
    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```

This factory ensures that quality ingredients are used during the pizza creation process…

… while also taking into account the tastes of people who live in Chicago

But how (or where) is this factory used?

Within Pizza Subclasses…


```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void back() {
        System.out.println("Back for 25 min at 350");
    }

    void cut() {...}
}
```

First, alter the Pizza abstract base class to make the prepare method abstract…

Within Pizza Subclasses...

Idaho State University | Computer Science

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFacotry) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

Then, update `Pizza` subclasses to make use of the factory! Note: we no longer need subclasses like `NYCheesePizza` and `ChicagoCheesePizza` because the ingredient factory no handles regional differences

One Last Step…


```
public class ChicagoPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory = new ChicagoPizzaIngredientFacotry();

        if (item == "cheese") {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Chicago Style Cheese Pizza");
        } else if (item == "veggie") {
            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("Chicago Style Veggie Pizza");
        }
        ...
    }
}
```

We need to update our `PizzaStore` subclasses to create the appropriate ingredient factory and pass it to each `Pizza` subclass in the `createPizza` factory method

Summary: What Did We Just Do?

1. We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza

2. This abstract factory gives us an interface for creating a family of products
   2.1 The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products

3. Our client code (`PizzaStore`) can then pick the factory appropriate to its region, plug it in, and get the correct style of pizza (Factory Method) with the correct set of ingredients (Abstract Factory)

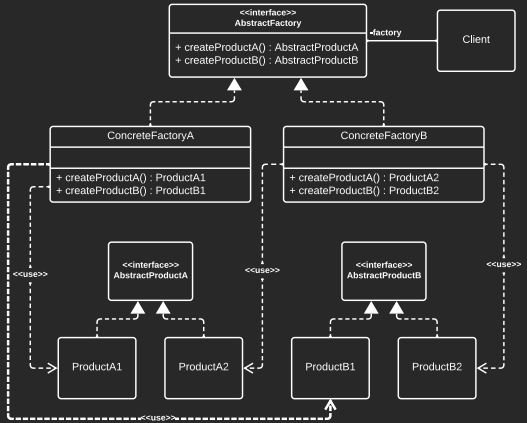
Demonstration

- Examine and execute the code located in the Abstract Factory directory of the src.zip file associated with this lecture

Abstract Factory: Definition and Structure

Idaho State University | Computer Science

- The abstract factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Factory Wrapup

- All factories encapsulate object creation
  - Simple Factory is not a bona fide design pattern but it is simple to understand and implement
  - Factory Method relies on inheritance: object creation occurs in subclasses
  - Abstract Factory relies on composition: object creation occurs in concrete factories

- Both can be used to shield your applications from concrete classes
  - And both can aid you in applying the dependency inversion principle in your designs

For Next Time

- Review this Lecture
- Watch Lecture 24

Are there any questions?