Factory Patterns & STL Containers

CS 247

University of Waterloo

cs247@uwaterloo.ca

July 14th, 2017
Overview

1 Factory
   - Factory Idiom
   - Factory Method Pattern
   - Abstract Factory

2 STL Containers
   - Random Access Containers
   - Lists
   - Associative Collections
A Factory is an object that makes other objects. With a factory, you can use:

- **Encapsulation**: client code is not directly tied to specific classes, so classes can be changed, added, or refactored without changing client code.

- **Polymorphism**: delegating the creation of an object to the right factory means that we can decide what kind of object to create at runtime.
Example:
1. two types of pizzas: CheesePizza and PepperoniPizza
2. three steps to make a pizza: get pizza essentials, bake, cut
Is the following code robust?

Pizza * orderPizza() {
    Pizza * pizza = new Pizza();
    pizza->bake();
    pizza->cut();
    return pizza;
}
Consider the scenario where you have more than one type of pizza. Creation of pizzas depends on pizza types.

Pizza * orderPizza(String type) {
    Pizza * pizza;
    if (type == "cheese") {
        pizza = new CheesePizza();
    } else if (type == "pepperoni") {
        pizza = new PepperoniPizza();
    }
    pizza->bake();
pizza->cut();
    return pizza;
}

Which OOP principle does the code above break?
Factory Idiom: A Simple Factory

You may want to encapsulate the creation of pizza.

class SimplePizzaFactory {
public:
    Pizza * createPizza(string type) {
        Pizza * pizza;
        if (type == "cheese") {
            pizza = new CheesePizza();
        } else if (type == "pepperoni") {
            pizza = new PepperoniPizza();
        }
        return pizza;
    }
};
class PizzaStore {
    SimplePizzaFactory factory;

public:
    PizzaStore(SimplePizzaFactory factory):
        factory{factory} {}

    Pizza * orderPizza(string type) {
        Pizza * pizza = factory.createPizza(type);
        pizza->bake();
        pizza->cut();
        return pizza;
    }
};
The previous example is simple, practical, and commonly used, but does not take advantage of Polymorphism. The Factory Method 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 Pattern

Example:
1. two types of pizza stores: ChicagoPizzaStore and NYPizzaStore
3. NYPizzaStore produces NYCheesePizza and NYPepperoniPizza.
class PizzaStore {
public:
    Pizza * orderPizza(string type) {
        Pizza * pizza = createPizza(type);
        pizza->bake();
        pizza->cut();
        return pizza;
    }

private:
    virtual Pizza * createPizza(string type) = 0;
};
class NYPizzaStore : public PizzaStore {
    Pizza * createPizza(string type) override {
        if (type == "cheese") {
            return new NYCheesePizza();
        }
        if (type == "pepperoni") {
            return new NYPepperoniPizza();
        }
    }
};
Factory Method Pattern

class ChicagoPizzaStore : public PizzaStore {
    Pizza * createPizza(string type) override {
        if (type == "cheese") {
            return new ChicagoCheesePizza();
        }
        if (type == "pepperoni") {
            return new ChicagoPepperoniPizza();
        }
    }
};
The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
What’s the difference between Abstract Factory and Factory Design Pattern?
Abstract Factory vs. Factory Design Pattern

Factory Method is used to create one product only but Abstract Factory is about creating families of related or dependent products.
Abstract Factory

Example:
1. two types of PizzaIngredientFactories: ChicagoPizzaIngredientFactory and NYPizzaIngredientFactory
2. two types of products: sauce and cheese
4. NYPizzaIngredientFactory produces NYSauce and NYCheese.
class PizzaIngredientFactory {
public:
    virtual Sauce * createSauce() = 0;
    virtual Cheese * createCheese() = 0;
};
Abstract Factory

class NYPizzaIngredientFactory: public PizzaIngredientFactory
{
public:
    Sauce * createSauce() override;
    Cheese * createCheese() override;
};

class ChicagoPizzaIngredientFactory: public PizzaIngredientFactory
{
public:
    Sauce * createSauce() override;
    Cheese * createCheese() override;
};
class NYSauce: public Sauce {...};
class ChicagoSauce: public Sauce {...};

class NYCheese: public Cheese {...};
class ChicageCheese: public Cheese {...};
The standard template library has concrete implementations of many different containers. A good reference is available at http://www.cplusplus.com/reference/stl/.
Random Access Containers

You’ll want to choose a container based on the situation you’re in.

- **Array**: if you know in advance how many elements you’ll have.
- **Vector**: if you’ll be adding items to the end.
- **Deque** (*double-ended queue*): if you’ll be adding items to both ends.

All of the above have the benefit of random-access, which is handy.
Lists don’t have the ability to do random access, but they have the added benefit of being able to insert into the *middle* without much work. A forward list (new to C++11) is a list without the double-ended pointers, so you can only iterate through them in one direction (on the plus side, they take up a little less space).
An Associative Collection is a container that doesn’t keep track of the order in which elements were inserted (e.g. a Set, or a Map). These are typically implemented as Binary Search Trees, and can be iterated over in order (i.e. alphabetically by key).

Unordered Associative Collections are similar, but are implemented as hash functions instead of BSTs. This makes lookup faster (O(1) instead of O(log n)) but does not keep the keys in meaningful order.
End