Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Design pattern

Factory

Some motivations

- Consider a user interface toolkit to support multiple look-and-feel standards:
  - for portability an application must not hard code its widgets for one look and feel.
- Use of the factory pattern allows:
  - generation of different instances of a class, using same parameter types
  - increase of system flexibility – code can use an object of an interface (type) w/o knowing which class (implementation) it belongs to

Factory pattern

- Problem
  - You want a class to create a related class polymorphically
- Context
  - Each class knows which version of the related class it should create
- Solution
  - Declare abstract method that derived classes override
- Consequences
  - Type created matches type(s) it's used with

Factory pattern

- Factory: a class whose sole job is to easily create and return instances of other classes:
  - it's a creational pattern; makes it easier to construct complex objects, create individual objects in situations where the constructor alone is inadequate.
  - instead of calling a constructor, use a static method in a "factory" class to set up the object

Pattern intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Lets a class defer instantiation to subclasses
- We'll see some variations on the theme of Factory

The problem with new

- In some cases there's need to instantiate closely related classes (e.g. derived from a common base) depending on some criteria, e.g.:

```
Duck duck;
if ( picnic ) {
   duck = new MallardDuck();
} else if( decorating ) {
   duck = new DecoyDuck();
} else if( inBathTub ) {
   duck = new RubberDuck();
}
```

The problem with new

- In some cases there's need to instantiate closely related classes (e.g. derived from a common base) depending on some criteria, e.g.:

What happens if we have to add another duck ?

-
  ```
  Duck duck;
  if ( picnic ) {
     duck = new MallardDuck();
  } else if( decorating ) {
     duck = new DecoyDuck();
  } else if( inBathTub ) {
     duck = new RubberDuck();
  }
  ```

Simple Factory

Goal

- Encapsulate the creation of related classes into one class: we'll have to modify only that class when the implementation changes
- The factory will handle the details of object creation
- The Simple Factory is not a real Design Pattern, it's more a programming idiom

Design Patterns and Programming Idioms

- According to Alexander, a pattern:
  - Describes a recurring problem
  - Describes the core of a solution
  - Is capable of generating many distinct designs
- An Idiom is more restricted
  - Still describes a recurring problem
  - Provides a more specific solution, with fewer variations
  - Applies only to a narrow context
    - e.g., the C++ language

Simple Factory example

```
Pizza* orderPizza(string type) {
  Pizza* pizza = 0;

  if ( type.compare(“4cheeses”) == 0 )
    pizza = new FourCheesesPizza();
  else if ( type.compare(“zucchini”) == 0 )
    pizza = new ZucchiniPizza();
  else if ( type.compare(“ham_mushrooms”) == 0 )
    pizza = new HamMushroomsPizza();

  pizza->prepare();
  pizza->bake();
  pizza->box();
  return pizza;
}
```

Simple Factory example

```
Pizza* orderPizza(string type) {
  Pizza* pizza = 0;

  if ( type.compare(“4cheeses”) == 0 )
    pizza = new FourCheesesPizza();
  else if ( type.compare(“zucchini”) == 0 )
    pizza = new ZucchiniPizza();
  else if ( type.compare(“ham_mushrooms”) == 0 )
    pizza = new HamMushroomsPizza();

  pizza->prepare();
  pizza->bake();
  pizza->box();
  return pizza;
}
```

Adding new types of pizzas will require to change this code

Simple Factory example

```
Pizza* orderPizza(string type) {
  Pizza* pizza = 0;

  if ( type.compare("4cheeses") == 0 )
    pizza = new FourCheesesPizza();
  else if ( type.compare("zucchini") == 0 )
    pizza = new ZucchiniPizza();
  else if ( type.compare("ham_mushrooms") == 0 )
    pizza = new HamMushroomsPizza();

  pizza->prepare();
  pizza->bake();
  pizza->box();
  return pizza;
}
```

Adding new types of pizzas will require to change this code

This part of code will remain the same

Encapsulating object creation

```
class SimplePizzaFactory {
  public: Pizza* createPizza( string type ) const {

    Pizza* pizza = 0;

    if ( type.compare(“4cheeses”) == 0 )
      pizza = new FourCheesesPizza();
    else if ( type.compare(“zucchini”) == 0 )
      pizza = new ZucchiniPizza();
    else if ( type.compare(“ham_mushrooms”) == 0 )
      pizza = new HamMushroomsPizza();

    return pizza;
  }
};
```

Using the Simple Factory

```
class PizzaStore {
  private: SimplePizzaFactory* factory;

  public: PizzaStore( SimplePizzaFactory* factory ) :
     this->factory( factory ) { }

  public: Pizza* orderPizza( string type ) {
     Pizza* pizza;
     pizza = factory->createPizza( type );
     pizza->prepare();
     pizza->bake();
     pizza->box();

     return pizza;
  }
};
```

Using the Simple Factory

Hold a reference to a Simple Factory

```
class PizzaStore {
  private: SimplePizzaFactory* factory;

  public: PizzaStore( SimplePizzaFactory* factory ) :
     this->factory( factory ) { }

  public: Pizza* orderPizza( string type ) {
     Pizza* pizza;
     pizza = factory->createPizza( type );
     pizza->prepare();
     pizza->bake();
     pizza->box();

     return pizza;
  }
};
```

Using the Simple Factory

Hold a reference to a Simple Factory

Get the factory passed in the constructor

```
class PizzaStore {
  private: SimplePizzaFactory* factory;

  public: PizzaStore( SimplePizzaFactory* factory )
     this->factory( factory ) { }

  public: Pizza* orderPizza( string type ) {
     Pizza* pizza;
     pizza = factory->createPizza( type );
     pizza->prepare();
     pizza->bake();
     pizza->box();

     return pizza;
  }
};
```

Using the Simple Factory

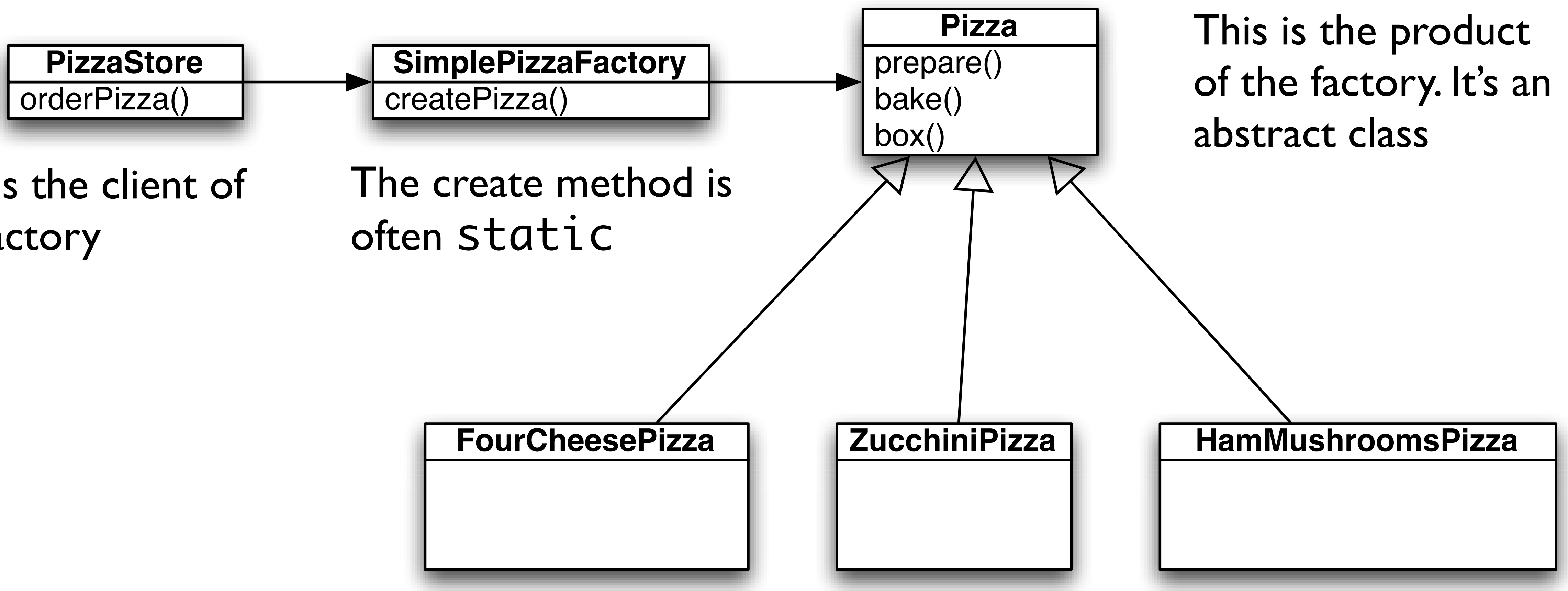
```
class PizzaStore {
  private: SimplePizzaFactory* factory;

  public: PizzaStore( SimplePizzaFactory* factory )
     this->factory( factory ) { }

  public: Pizza* orderPizza( string type ) {
     Pizza* pizza;
     pizza = factory->createPizza( type );
     pizza->prepare();
     pizza->bake();
     pizza->box();

     return pizza;
  }
};
```

Hold a reference to a Simple Factory

Get the factory passed in the constructor

Use the factory with the create() method instead of using a `new`

Simple Factory UML class diagram



This is the product of the factory. It's an abstract class

This is the client of the factory

The create method is often `static`

Concrete products of the factory.

Factory Method

Class creational

Some motivations

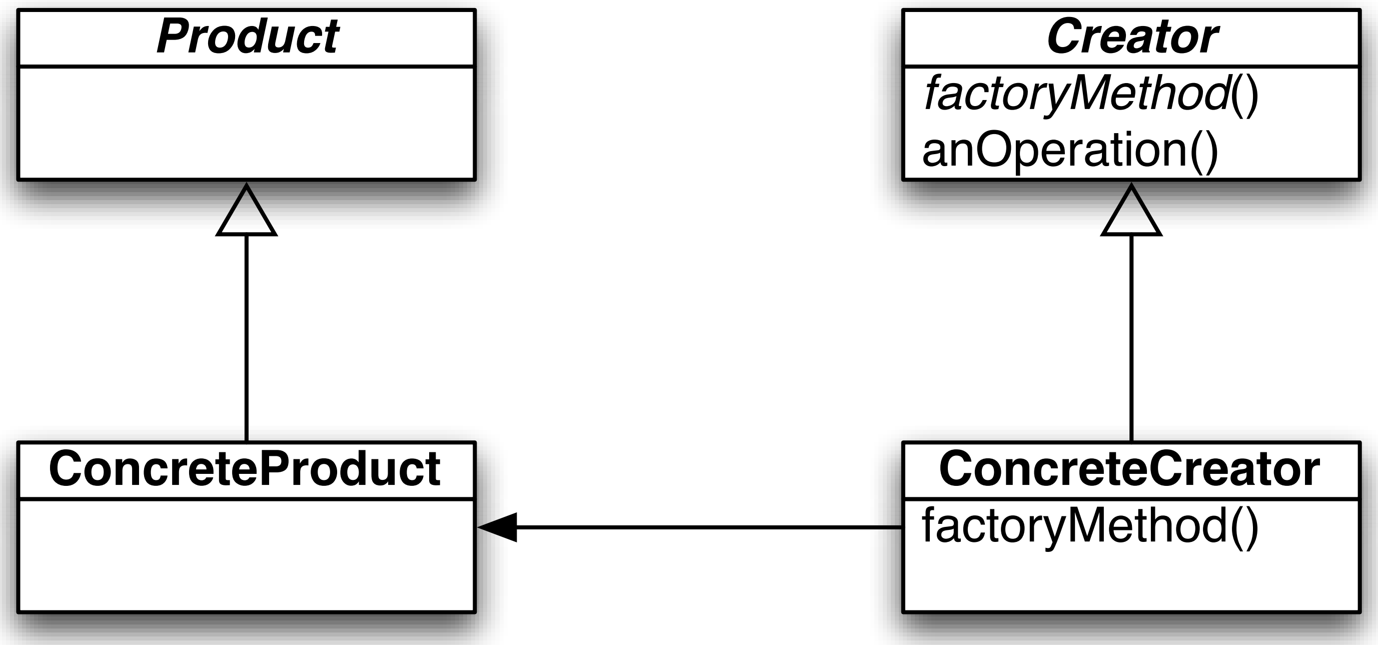
- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create
  - a class wants its subclasses to specify the object it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Factory Method

- Problem
  - You want a class to create a related class polymorphically
- Context
  - Each class knows which version of the related class it should create
- Solution
  - Declare abstract method that derived classes override
- Consequences
  - Type created matches type(s) it's used with

Factory method UML class diagram

All products must implement the same interface so that the classes which use the products can refer to the interface and not to the concrete class



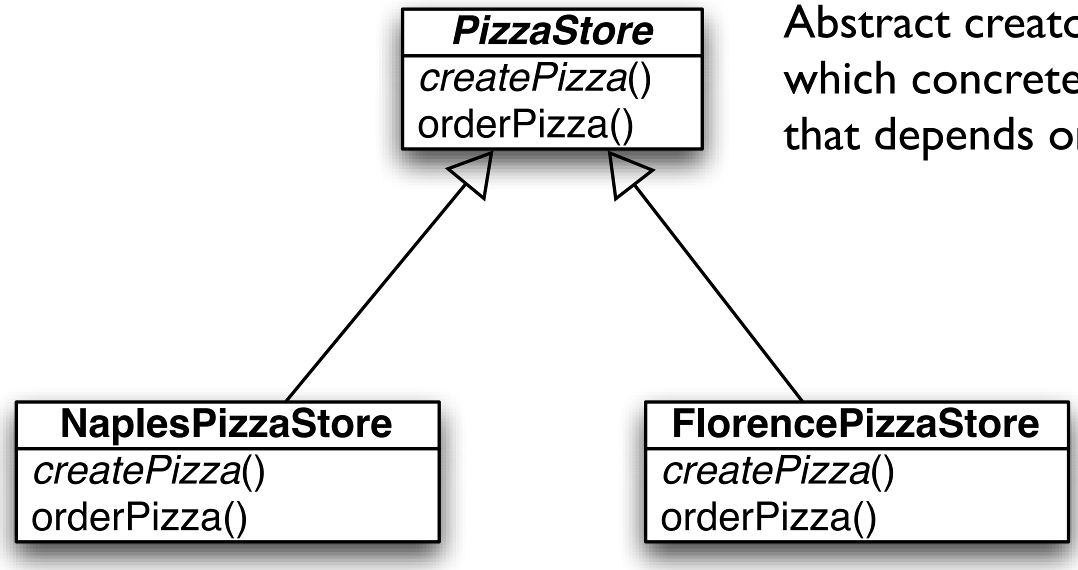
The abstract `factoryMethod()` must be implemented by all the subclasses. The other methods are there to operate on products produced by the factory method.

The implementation of `factoryMethod()` actually produces products

The concrete creator is the only responsible for creating one or more concrete products, and is the only class that knows how to create these products

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory Method example UML class diagram

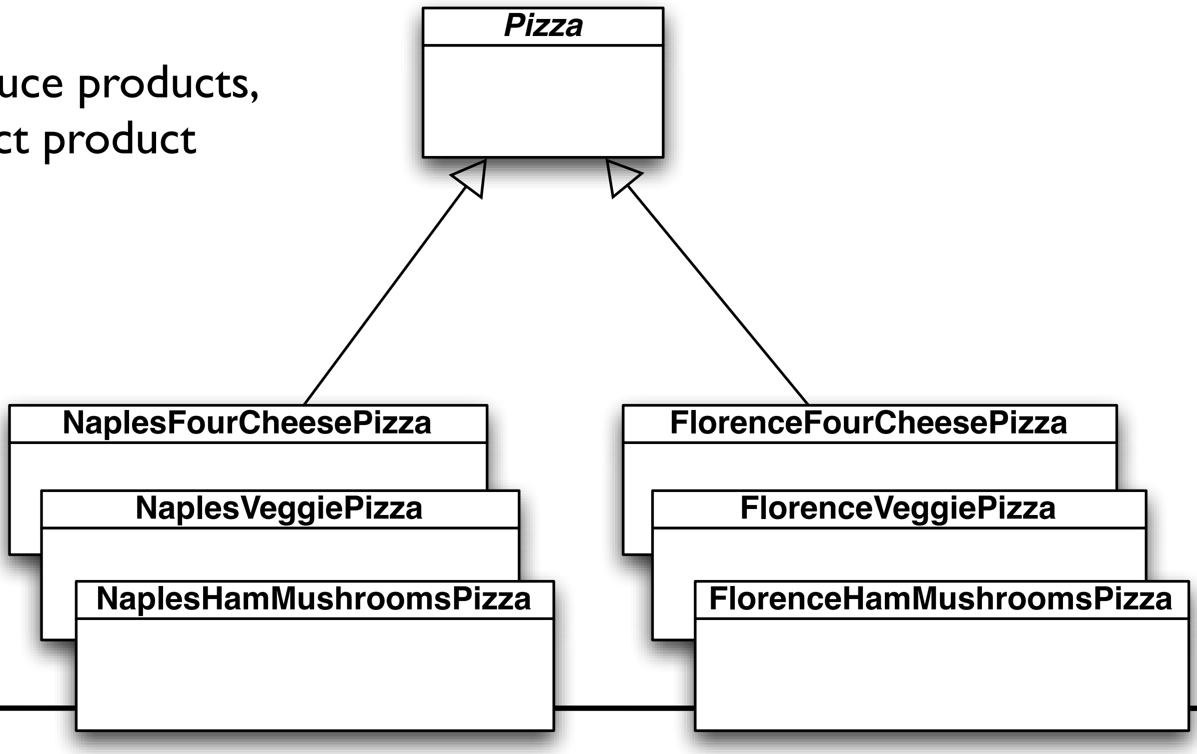


Abstract creator class. The creator does not know which concrete product is produced, but may have code that depends on an abstract product

Creator classes

Concrete creator classes. They create different styles of pizza implementing the abstract factory method

Factories produce products, like this abstract product



Product classes

The concrete products produced by the different concrete factories

Participants

- Product: defines the interface of objects the factory method creates
- ConcreteProduct: implements the Product interface
- Creator: declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object. May call the factory method to create a Product object
- ConcreteCreator: overrides the factory method to return an instance of a ConcreteProduct

Factory Method example

```
class PizzaStore {
  protected: PizzaStore() {  }
  public: virtual ~PizzaStore() = 0 { }

  public: Pizza* orderPizza( string type) const {
    Pizza* pizza;

    pizza = createPizza(type);

    cout << "- Making a " << pizza->getName() << " -" << endl;
    pizza->prepare();
    pizza->bake();
    pizza->cut();
    pizza->box();
    return pizza;
}

  public: virtual Pizza* createPizza(string type) const = 0;
};
```

Factory Method example

```
class PizzaStore {
  protected: PizzaStore() {  }
  public: virtual ~PizzaStore() = 0 { }

  public: Pizza* orderPizza( strin
    Pizza* pizza;

    pizza = createPizza(type);

    cout << "- Making a " << pizza->getName() << " -" << endl;
    pizza->prepare();
    pizza->bake();
    pizza->cut();
    pizza->box();
    return pizza;
}

  public: virtual Pizza* createPizza(string type) const = 0;
};
```

The `createPizza()` is back into the PizzaStore object rather than in a factory object

Factory Method example

```
class PizzaStore {
  protected: PizzaStore() {  }
  public: virtual ~PizzaStore() = 0 { }

  public: Pizza* orderPizza( strin
    Pizza* pizza;

    pizza = createPizza(type);

    cout << "- Making a " << pizza->getName() << " -" << endl;
    pizza->prepare();
    pizza->bake();
    pizza->
    pizza->
    return
}

  public: virtual Pizza* createPizza(string type) const = 0;
};
```

The `createPizza()` is back into the PizzaStore object rather than in a factory object

The factory object has been moved to this method

Factory Method example

```
class PizzaStore {
  protected: PizzaStore() {  }
  public: virtual ~PizzaStore() = 0 { }

  public: Pizza* orderPizza( strin
    Pizza* pizza;

    pizza = createPizza(type);

    cout << "- Making a " << pizza->getName() << " -" << endl;
    pizza->prepare();
    pizza->bake();
    pizza->
    pizza->
    return
}

  public: virtual Pizza* createPizza(string type) const = 0;
};
```

The `createPizza()` is back into the PizzaStore object rather than in a factory object

The factory object has been moved to this method

The factory method is abstract in the PizzaStore

Factory Method example - cont

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( string type ) const {
    if( type.compare( "fourcheese" ) == 0 ) {
        return new NaplesStyleFourCheesePizza();
    } else if( type.compare( "veggie" ) == 0 ) {
        return new NaplesStyleVeggiePizza();
    } else if( type.compare( "clam" ) == 0 ) {
        return new NaplesStyleClamPizza();
    } else if( type.compare( "hammushrooms" ) == 0 ) {
        return new NaplesStyleHamMushroomsPizza();
    } else return 0;
}
};
```

Factory Method example - cont

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( s
   if( type.compare( "fourchee
       return new NaplesStyleFo
   } else if( type.compare( "v
       return new NaplesStyleVe
   } else if( type.compare( "c
       return new NaplesStyleClamPizza();
   } else if( type.compare( "hammushrooms" ) == 0 ) {
       return new NaplesStyleHamMushroomsPizza();
   } else return 0;
}
};
```

The `createPizza()` of the Naples pizza store ensures that pizzas are created as in Naples: thick, large crust and using only buffalo mozzarella cheese

Factory Method example - cont

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( s
    if( type.compare( "fourchee
        return new NaplesStyleFo
    } else if( type.compare( "v
        return new NaplesStyleVe
    } else if( type.compare( "c
        return new NaplesStyleClamPizza();
    } else if( type.compare( "hammushrooms" ) == 0 ) {
        return new NaplesStyleHamMushroomsPizza();
    } else return 0;
}
};
```

The `createPizza()` of the Naples pizza store ensures that pizzas are created as in Naples: thick, large crust and using only buffalo mozzarella cheese

Each subclass of PizzaStore overrides the abstract `createPizza()` method, while all subclasses use the `orderPizza()` method defined in PizzaStore.

Decoupling

- The `PizzaStore::orderPizza()` is defined in the abstract PizzaStore class, not in the subclasses: the method does not know which subclass is running the code and making the pizzas
    - it's decoupled from that code
- When `orderPizza()` calls `createPizza()` one of the subclasses is called in action, depending on the PizzaStore subclass
    - it's NOT a run-time decision by the subclass

The factory method

- The factory method handles the object creation and encapsulates it in a subclass. This decouples the client code in the superclass (e.g. code like `orderPizza()`) from the object creation in the subclass.
  - the factory method has to be virtual and possibly also pure virtual (but a default implementation may be provided, to obtain flexibility: subclasses can override how they are created)
  - the factory method may be parameterized (or not) to select among variations of the product (e.g. useful for de-serialization)

How to get a pizza

- Get a pizza store:
`PizzaStore* mergellinaStore = new NaplesPizzaStore();`
- Take an order:
`mergellinaStore->orderPizza(“veggie”);`
- The `orderPizza()` method calls the `createPizza()` method implemented in the subclass:
`Pizza* pizza = createPizza(“veggie”);`
- The `orderPizza()` finished preparing it:

```
pizza->prepare();
pizza->bake();
...
```

Implementing pizzas

```
class Pizza {
  protected: string name;
  protected: string dough;
  protected: string sauce;
  protected: list< string > toppings;
  protected: Pizza() {    }
  public: virtual ~Pizza() = 0 {   }
  public: virtual void prepare() const {
    cout << "Preparing " << _name.c_str() << endl;
    cout << "Tossing dough..." << endl;
    cout << "Adding sauce..." << endl;
    cout << "Adding toppings: " << endl;
    for( list< string >::iterator itr = toppings.begin();
         toppings.end() != itr; ++itr ) {
      cout << "   " << itr->c_str() << endl;
    }
  }
  public: virtual void bake() const {
    cout << "Bake for 25 minutes at 350" << endl;
  }
  // void bake(); void cut(); void box(); string getName(); ...
```

Implementing pizzas

Abstract class (it has abstract methods)

```
class Pizza {
  protected: string name;
  protected: string dough;
  protected: string sauce;
  protected: list< string > toppings;
  protected: Pizza() {    }
  public: virtual ~Pizza() = 0 {   }
  public: virtual void prepare() const {
    cout << "Preparing " << _name.c_str() << endl;
    cout << "Tossing dough..." << endl;
    cout << "Adding sauce..." << endl;
    cout << "Adding toppings: " << endl;
    for( list< string >::iterator itr = toppings.begin();
         toppings.end() != itr; ++itr ) {
      cout << "   " << itr->c_str() << endl;
    }
  }
  public: virtual void bake() const {
    cout << "Bake for 25 minutes at 350" << endl;
  }
  // void bake(); void cut(); void box(); string getName(); ...
```

Implementing pizzas

Abstract class (it has abstract methods)

```
class Pizza {
  protected: string name;
  protected: string dough;
  protected: string sauce;
  protected: list< string > toppings;
  protected: Pizza() {    }
  public: virtual ~Pizza() = 0 {   }
  public: virtual voi
    cout << "Preparin
    cout << "Tossing
    cout << "Adding s
    cout << "Adding t
    for( list< string
         toppings.end(
      cout << "    " << itr->c_str() << endl;
    }
  }
  public: virtual void bake() const {
     cout << "Bake for 25 minutes at 350" << endl;
  }
  // void bake(); void cut(); void box(); string getName(); ...
```

The class provides some basic default methods for preparing, baking, cutting,... They are virtual and can be overridden by the subclasses

Implementing pizzas - cont.

```
class NaplesStyleVeggiePizza : public Pizza {

  public: NaplesStyleVeggiePizza() {

    name = "Naples Style Veggie Pizza";
    dough = "Thick Crust Dough";
    sauce = "Marinara Sauce";

    toppings.push_back( "Buffalo Mozzarella Cheese" );

    toppings.push_back( "Garlic" );
    toppings.push_back( "Onion" );
    toppings.push_back( "Mushrooms" );
    toppings.push_back( "Friarelli" );

  }

  public: virtual void bake() const {
    cout << "Bake for 20 minutes at 350" << endl;
  }
};
```

Implementing pizzas - cont.

```
class NaplesStyleVeggiePizza : public Pizza {

  public: NaplesStyleVeggiePizza() {

    name = "Naples Style Veggie Pizza";
    dough = "Thick Crust Dough";
    sauce = "Marinarc

    toppings.push_bac

    toppings.push_bac
    toppings.push_back( "Onion" );
    toppings.push_back( "Mushrooms" );
    toppings.push_back( "Friarelli" );

  }

  public: virtual void bake() const {
     cout << "Bake for 20 minutes at 350" << endl;
  }
};
```

The Naples style pizza has its thick crust, marinara sauce, *friarelli* veggie and uses buffalo mozzarella cheese

Implementing pizzas - cont.

```
class NaplesStyleVeggiePizza : public Pizza {

  public: NaplesStyleVeggiePizza() {

    name = "Naples Style Veggie Pizza";
    dough = "Thick Crust Dough";
    sauce = "Marinarc

    toppings.push_bac

    toppings.push_bac
    toppings.push_back( "Onion" );
    toppings.push_back( "Mushrooms" );
    toppings.push_back( "Friarelli" );

  }

  public: virtual void bake() const {
    cout << "Bake for 20 minutes at 350" << endl;
  }
};
```

The Naples style pizza has its thick crust, marinara sauce, *friarelli* veggie and uses buffalo mozzarella cheese

The Naples style pizza is baked less time, to make a soft crust

Putting everything together

```
PizzaStore* mergellinaStore = new NaplesPizzaStore();

Pizza* pizza = mergellinaStore->orderPizza("veggie");
```

- This approach is useful also if there's only one concrete creator since the Factory Method decouples product implementation from its use
- The factory method and creator do not need to be abstract, they may provide some basic implementation
- The implementation of each concrete store looks like the Simple Factory, but in this previous approach the factory is another object composed with the PizzaStore, here it is a subclass extending an abstract class
  - it's not a one-shot solution, we are using a framework that let's subclasses decide which implementation will be used
  - the factory method can also change the products created: it's more flexible

Lazy initialization

- The constructor simply initializes the product to 0, the creation is delegated to the accessor method (check also the Singleton pattern!):

```
class Creator {
public: Creator() { product = 0; };
public:  Product* getProduct();
protected:  virtual Product* createProduct();
private:  Product* product;
};
Product* Creator::getProduct() {
  if ( product == 0 ) {
    product = createProduct();
  }
  return product;
}
```

Abstract Factory

Object creational

Motivation

- Consider a user interface toolkit to support multiple look-and-feel standards.
- For portability an application must not hard code its widgets for one look and feel.
  - How to design the application so that incorporating new look and feel requirements will be easy?

Solution

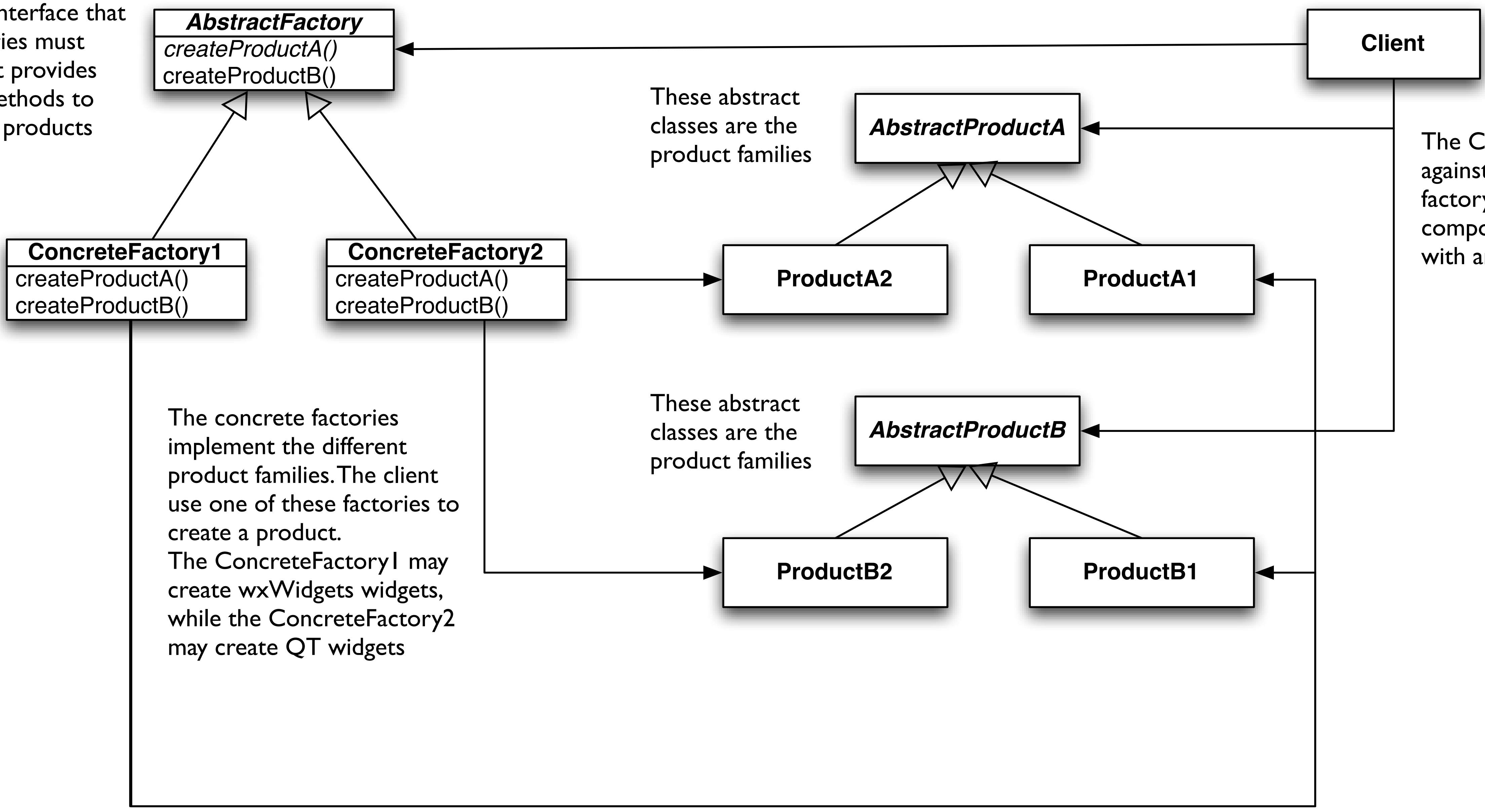
- Define an abstract **WidgetFactory** class.
    - This class declares an interface to create different kinds of widgets.
- There is one abstract class for each kind of widget and concrete subclasses implement widgets for different standards.
- **WidgetFactory** offers an operation to return a new widget object for each abstract widget class. Clients call these operations to obtain instances of widgets without being aware of the concrete classes they use.

Intent and applicability

- Provide an interface for creating families of related or dependent objects w/o specifying their concrete classes
- This pattern can be applied when:
    - a system should be independent of how its products are created, composed or represented
    - a system should be configured with one or multiple families of products
    - a family of related product objects is designed to be used together (and there’s need to enforce this constraint)
    - there is need to provide a class library of products revealing their interfaces and not their implementations

- The Abstract Factory pattern is one level of abstraction higher than the factory pattern.
- This pattern returns one of several related classes, each of which can return several different objects on request.
    - In other words, the Abstract Factory is a factory object that returns one of several factories.

Abstract Factory UML class diagram



The Abstract Factory defines the interface that all the factories must implement. It provides (abstract) methods to produce the products

These abstract classes are the product families

The Client is written against the abstract factory and composed at runtime with an actual factory

These abstract classes are the product families

The concrete factories implement the different product families. The client use one of these factories to create a product.
The ConcreteFactory1 may create wxWidgets widgets, while the ConcreteFactory2 may create QT widgets

Participants

- AbstractFactory: declares an interface for operations that create abstract product objects
- ConcreteFactory: implements the operations to create concrete product objects
- AbstractProduct: declares an interface for a type of product object
- ConcreteProduct: defines a product to be object created by the corresponding concrete factory, implementing the AbstractProduct interface
- Client: uses only the interfaces create by the AbstractXXX classes

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This factory creates objects having a particular implementation, to create different objects use a different factory. This promotes consistency among products: products of a whole family are created.
- AbstractFactory defers creation to the ConcreteFactory classes. It insulates the client from implementation classes.

Implementation

- An application typically needs only one instance of a factory: these are implemented using the Singleton pattern
- Often the concrete factories are built using the Factory Method pattern for each product
- The AbstractFactory usually defines a different operation for each kind of product; these products are encoded in the operation signatures, thus adding a new kind of product requires changing the interface.

Abstract Factory: example

```
// Abstract Factory
class PizzaIngredientFactory {
public:
  virtual Dough* createDough() const = 0;
  virtual Sauce* createSauce() const = 0;
  virtual Cheese* createCheese() const =
0;
  virtual std::vector< Veggies* >
    createVeggies() const = 0;
  virtual Clams* createClam() const = 0;
  virtual ~PizzaIngredientFactory() = 0 {}
};
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
  public: Dough* createDough() const {
    return new ThickCrustDough();
  }
  public: Sauce* createSauce() const {
    return new MarinaraSauce();
  }
  public: Cheese* createCheese() const {
    return new BuffaloMozzarellaCheese();
  }
  public: std::vector< Veggies* >
  createVeggies() const {
    std::vector< Veggies* > veggies;
    veggies.push_back( new Friarelli() );
    veggies.push_back( new Onion() );
    veggies.push_back( new Mushroom() );
    veggies.push_back( new RedPepper() );
    return veggies;
  }
  public: Clams* createClam() const {
    return new FreshClams();
  }
};
```

Abstract Factory: example

```
// Abstract Factory
class PizzaIngredientFactory {
public:
  virtual Dough* createDough() const = 0;
  virtual Sauce* createSauce() const = 0;
  virtual Cheese* createCheese() const =
0;
  virtual std::vector< Veggies* >
    createVeggies() const = 0;
  virtual Clams* createClam() const = 0;
  virtual ~PizzaIngredientFactory() = 0 {}
};
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
  public: Dough* createDough() const {
    return new ThickCrustDough();
  }
  public: Clams* createClam() const {
    return new FreshClams();
  }
};
```

We have many classes: one for each ingredient. If there’s need for a common functionality in all the factories implement a method here.

Abstract Factory: example

```
// Abstract Factory
class PizzaIngredientFactory {
public:
  virtual Dough* createDough() const = 0;
  virtual Sauce* createSauce() const = 0;
  virtual Cheese* createCheese() const =
0;
  virtual std::vector< Veggies* >
    createVeggies() const = 0;
  virtual Clams* createClam() const = 0;
  virtual ~PizzaIngredientFactory() = 0 {}
};
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
  public: Dough* createDough() const {
    return new ThickCrustDough();
  }
  public: Sauce* createSauce() const {
    return new MarinaraSauce();
  }
  public: Cheese* createCheese() const {
    return new BuffaloMozzarellaCheese();
  }
  public: std::vector< Veggies* >
  createVeggies() const {
    std::vector< Veggies* > veggies;
    veggies.push_back( new Friarelli() );
    veggies.push_back( new Onion() );
    veggies.push_back( new Mushroom() );
    veggies.push_back( new RedPepper() );
    return veggies;
  }
  public: Clams* createClam() const {
    return new FreshClams();
  }
};
```

Abstract Factory: example

```
//
cl
pu

0;

};
```

We are creating a specific version of ingredient for each factory.
Some ingredients may be shared by different factories, though.

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
  public: Dough* createDough() const {
    return new ThickCrustDough();
  }
  public: Sauce* createSauce() const {
    return new MarinaraSauce();
  }
  public: Cheese* createCheese() const {
    return new BuffaloMozzarellaCheese();
  }
  public: std::vector< Veggies* >
  createVeggies() const {
    std::vector< Veggies* > veggies;
    veggies.push_back( new Friarelli() );
    veggies.push_back( new Onion() );
    veggies.push_back( new Mushroom() );
    veggies.push_back( new RedPepper() );
    return veggies;
  }
  public: Clams* createClam() const {
    return new FreshClams();
  }
};
```

Abstract Factory: example

```
class Pizza {
private: std::string name;
protected:
  Dough* dough;
  Sauce* sauce;
  std::vector< Veggies* > veggies;
  Cheese* cheese;
  Clams* clam;
  Pizza() { }
public: virtual void prepare() const = 0;
  virtual ~Pizza() {
    for( auto itr = begin(veggies); its != end(veggies); ++itr ) {
      delete *itr;
    }
    veggies.clear();
  }
  virtual void bake() const {
    std::cout << "Bake for 25 minutes at 350"
    << std::endl;
  }
  virtual void box() const {
     std::cout << "Place pizza in official
     PizzaStore box" << std::endl;
  } //...all the other methods...
```

Abstract Factory: example

```
class Pizza {
private: std::string name;
protected:
  Dough* dough;
  Sauce* sauce;
  std::vector< Veggies* > veggies;
  Cheese* cheese;
  Clams* clam;
  Pizza() { }
public: virtual void prepare() const = 0;
  virtual ~Pizza() {
    for( auto itr = begin(veggies); its != end(veggies); ++itr ) {
      delete *itr;
    }
    veggies.clear();
  }
  virtual void bake() const {
    std::cout << "Bake for 25 minutes at 350"
    << std::endl;
  }
  virtual void box() const {
     std::cout << "Place pizza in official
     PizzaStore box" << std::endl;
  } //...all the other methods...
```

The pure virtual `prepare` method will collect all the ingredients from the ingredient factory

Abstract Factory: example

- The concrete product classes get their ingredients from the ingredient factories: there’s no more need for specific classes for the regional versions

```
class ClamPizza : public Pizza {
  private: PizzaIngredientFactory* ingredientFactory;
  public: ClamPizza(PizzaIngredientFactory* ingredientFactory) :
    ingredientFactory( ingredientFactory ) {
  }
  void prepare() const {
    std::cout << "Preparing " << getName().c_str() << std::endl;
    dough = ingredientFactory->createDough();
    sauce = ingredientFactory->createSauce();
    cheese = ingredientFactory->createCheese();
    clam = ingredientFactory->createClam();
  }
};
```

Abstract Factory: example

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( std::string item ) const {
    Pizza* pizza = 0;

    PizzaIngredientFactory* ingredientFactory =
        new NaplesPizzaIngredientFactory();

    if( item.compare( "cheese" ) == 0 ) {
      pizza = new CheesePizza( ingredientFactory );
      pizza->setName( "Naples Style Cheese Pizza" );
    } else if( item.compare( "veggie" ) == 0 ) {
      pizza = new VeggiePizza( ingredientFactory );
      pizza->setName( "Naples Style Veggie Pizza" );
    } else if( item.compare( "clam" ) == 0 ) {
      pizza = new ClamPizza( ingredientFactory );
      pizza->setName( "Naples Style Clam Pizza" );
    } else if( item.compare( "pepperoni" ) == 0 ) {
      pizza = new PepperoniPizza( ingredientFactory );
      pizza->setName( "Naples Style Pepperoni Pizza" );
    }
    return pizza;
  }
};
```

Abstract Factory: example

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( std::string item ) const {
    Pizza* pizza = 0;

    PizzaIngredientFactory* ingredientFactory =
        new NaplesPizzaIngredientFactory();

    if( item.compare( "cheese" ) == 0 ) {
      pizza = new CheesePizza( ingredientFactory );
      pizza->setName( "Naples Style Cheese Pizza" );
    } else if( item.compare( "veggie" ) == 0 ) {
      pizza = new VeggiePizza( ingredientFactory );
      pizza->setName( "Naples Style Veggie Pizza" );
    } else if( item.compare( "clam" ) == 0 ) {
      pizza = new ClamPizza( ingredientFactory );
      pizza->setName( "Naples Style Clam Pizza" );
    } else if( item.compare( "pepperoni" ) == 0 ) {
      pizza = new PepperoniPizza( ingredientFactory );
      pizza->setName( "Naples Style Pepperoni Pizza" );
    }
    return pizza;
  }
};
```

The store is composed with the regional ingredient factory.

Abstract Factory: example

```
class NaplesPizzaStore : public PizzaStore {

  public: Pizza* createPizza( std::string item ) const {
    Pizza* pizza = 0;

    PizzaIngredientFactory* ingredientFactory =
        new NaplesPizzaIngredientFactory();

    if( item.compare( "cheese" ) == 0 ) {
      pizza = new CheesePizza( ingredientFactory );
      pizza->setName( "Naples Style Cheese Pizza" );
    } else if( item.compare( "veggie" ) == 0 ) {
      pizza = new VeggiePizza( ingredientFactory );
      pizza->setName( "Naples Style Veggie Pizza" );
    } else if( item.compare( "clam" ) == 0 ) {
      pizza = new ClamPizza( ingredientFactory );
      pizza->setName( "Naples Style Clam Pizza" );
    } else if( item.compare( "pepperoni" ) == 0 ) {
      pizza = new PepperoniPizza( ingredientFactory );
      pizza->setName( "Naples Style Pepperoni Pizza" );
    }
    return pizza;
  }
};
```

The store is composed with the regional ingredient factory.

For each type of product we pass the factory it needs, to get the ingredients from it.
The factory (built according to Abstract Factory pattern) creates a <u>family</u> of products

Putting everything together

```
PizzaStore* nStore = new NaplesPizzaStore();

Pizza* pizza = nStore->orderPizza( "cheese" );

std::cout << "Just ordered a " << pizza->toString() << std::endl;

pizza = nStore->orderPizza( "clam" );

std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

Putting everything together

```
PizzaStore* nStore = new NaplesPizzaStore();

Pizza* pizza = nStore->orderPizza( "cheese" );

std::cout << "Just ordered a " << pizza->toString() << std::endl;

pizza = nStore->orderPizza( "clam" );

std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

The `orderPizza()` method calls the `createPizza()` method

Putting everything together

```
PizzaStore* nStore = new NaplesPizzaStore();

Pizza* pizza = nStore->orderPizza( "cheese" );

std::cout << '                                 ring() << std::endl;

pizza = nStore->orderPizza( "clam" );

std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

The `orderPizza()` method calls the `createPizza()` method

When the `createPizza()` method is called the factory gets involved

Putting everything together

```
PizzaStore* nStore = new NaplesPizzaStore();

Pizza* pizza = nStore->orderPizza( "cheese" );
```

The `orderPizza()` method calls the `createPizza()` method

```
std::cout << '                                   ring() << std::endl;
```

When the `createPizza()` method is called the factory gets involved

```
pizza = nStore
```

When `prepare()` method is called the factory creates the ingredients

```
std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

Factories and smart pointers

Instead of returning raw pointers we can use C++11 smart pointers, like `unique_ptr` or `shared_ptr`

```
#include <iostream>
#include <string>
#include <memory>Remind: using
namespace std;

class Song {
public:
Song(string name, string t) :
  artist(name), title(t) {}

  string artist, title;
};
```

```
unique_ptr<Song> SongFactory(string
artist, string title) {

    return unique_ptr<Song>(
         new Song(artist, title));
}

int main() {
// Obtain unique_ptr from function
// that returns rvalue reference.
  auto pSong = SongFactory("Michael
         Jackson", "Beat It");
}
```

Remind: `unique_ptr<T>` does not allow copy construction, instead it supports move semantics. Yet, you can return a `unique_ptr<T>` from a function and assign the returned value to a variable. since the return value is a temporary object that will be destroyed as soon as the function exits, thus guaranteeing the uniqueness of the returned pointer.

```
#include <iostream>
#include <string>
#include <memory>Remind: using
namespace std;

class Song {
public:
Song(string name, string t) :
  artist(name), title(t) {}

  string artist, title;
};
```

```
unique_ptr<Song> SongFactory(string
artist, string title) {

    return unique_ptr<Song>(
         new Song(artist, title));
}

int main() {
// Obtain unique_ptr from function
// that returns rvalue reference.
  auto pSong = SongFactory("Michael
         Jackson", "Beat It");
}
```

Factories and smart pointers

Let’s see an example with polymorphism and `unique_ptr`:

```
class Document {
public:
    virtual void draw() {
        std::cout << "Document::draw()" << std::endl;
    }
};

class MultimediaDocument : public Document {
    virtual void draw() override {
        std::cout << "Document::draw()" << std::endl;
    }
};

std::unique_ptr<Document> documentFactory(bool multimediaType) {
    std::unique_ptr<Document> result;
    if (multimediaType)
        result = std::unique_ptr<MultimediaDocument>(new MultimediaDocument);
    else {
        result = std::unique_ptr<Document>(new Document);
    }
    return result;
}
```

Factories and smart pointers

- unique_ptr is the best choice for a factory: if you need a shared_ptr you can construct one from the unique_ptr (there's a specific constructor in shared_ptr):

```
unique_ptr<Widget> createWidget(int id);

auto sp = shared_ptr<Widget>(createWidget(i));
```

- The only reason to return a `shared_ptr` if the ownership of the object must be shared with the factory

Singleton

Motivations

- Sometimes it is appropriate to have exactly one instance of a class: e.g., window managers, print spoolers, filesystems, program configurations.
- Typically, those types of objects known as **singletons**, are accessed by disparate objects throughout a software system, and therefore require a global point of access.
- The Singleton pattern addresses all the concerns above. With the Singleton design pattern you can:
    - Ensure that only one instance of a class is created.
    - Provide a global point of access to the object.
    - Allow multiple instances in the future without affecting a singleton class' clients.

Intent and applicability

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.
- Singletons maintain a static reference to the sole singleton instance and return a reference to that instance from a static method.

Implementation

- The Singleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the static method used to get it is called for the first time. This technique ensures that singleton instances are created only when needed.
- The Singleton class implements a protected constructor so clients cannot instantiate Singleton instances.
- To avoid that the subclasses call the protected constructors, we can make the Singleton constructor private, so that only Singleton's methods call it.

Singleton: an example

```
class Singleton {
public:
  static Singleton*
     getInstance();
protected:
  Singleton();
private:
  static Singleton* instance;
};
```

```
Singleton*
Singleton::instance = 0;

Singleton*
Singleton::getInstance() {
  if ( instance == 0 )
    instance =
      new Singleton();
  return instance;
}
```

Singleton: another example

```
class OtherSingleton {

private:
    static OtherSingleton* pInstance;
    OtherSingleton ();

    OtherSingleton(const OtherSingleton& rs) {
              pInstance = rs.pInstance;
    }

    OtherSingleton& operator = (const
       OtherSingleton& rs) {
           if (this != &rs) {
              pInstance = rs.pInstance;
           }

           return *this;
    }

    ~OtherSingleton ();
```

```
  public:

    static OtherSingleton& getInstance() {
        static OtherSingleton theInstance;
        pInstance = &theInstance;
        return *pInstance;

    }

};

    OtherSingleton* OtherSingleton::pInstance =
    nullptr;
```

Singleton: anc

Private copy constructor and assignment avoid that they can be called by users
Private destructor means that users can not erroneously destroy the object

getInstance returns a reference.
This approach is founded on C++'s guarantee that local static objects are initialized when the object's definition is first encountered during a call to that function.

```
class OtherSingleton {

private:
    static OtherSingleton* pInstance;
    OtherSingleton ();

    OtherSingleton(const OtherSingleton& rs) {
            pInstance = rs.pInstance;
    }

    OtherSingleton& operator = (const
       OtherSingleton& rs) {
           if (this != &rs) {
               pInstance = rs.pInstance;
           }

           return *this;
    }

    ~OtherSingleton ();

         static OtherSingleton& getInstance() {
             static OtherSingleton theInstance;
             pInstance = &theInstance;
             return *pInstance;

         }

     };

         OtherSingleton* OtherSingleton::pInstance =
         nullptr;
```

Consequences

- It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
- We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- We can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- What happens in a multi-threaded program when the method to get instances is called concurrently ? There may be need to synchronize/protect it in this case.

Reading material

- M. Bertini, “Programmazione Object-Oriented in C++”, parte II, cap. 4

Credits

- These slides are based on the material of:
  - Glenn Puchtel
  - Fred Kuhns, Washington University
  - Aditya P. Matur, Purdue University
  - Aaron Bloomfield, University of Virginia
  - Joey Paquet, Concordia University