Adapter, Factory, and More

Making the Concrete Abstract

I can’t wait to see the changes I asked you to make on the interface.
Our last meeting was two months ago. You must be finished by now.
I haven’t started yet.
I had a few questions.
I figured I’d ask you about them the next time we talked.

In the meantime, I only did work for people who yelled at me every day.
Micromanagement has a bad reputation, but I’m not too proud to say I need it.
Okay... well, I’m optimistic that you can make those changes for me by next week.
I probably should have asked my questions.
Correctly **making objects is complex**

Especially making **collection of related objects**
- Parts of a car
- Look-and-feel of a window: canvas, scrollbar, etc.

The correct making of objects is **not easily centralized** in one place
- Often do it all over code wherever object is needed
- Violates SRP and DRY

DP angle is that “**new**” names and makes a **concrete class**; should be referring to abstract concrete class has to be named, but we can at least hide that from most of system
- encapsulating class will violate OCP, but no others will
Factory is the answer

- Actually a collection of patterns
  - Simply factory (idiom)
  - Factory Method
  - Abstract Factory (not today)

```java
class Event {
    protected String name;
    protected int firstAllowedDate = Integer.MAX_VALUE; // fail hard if no init

    public Event(int eventsFirstAllowedDate, String eventName) {
        firstAllowedDate = EventsFirstAllowedDate;
        name = eventName
    }

    protected boolean dateSupported(int dateNumber) {
        return dateNumber >= firstAllowedDate;
    }
}
```

Concrete class; Mystery how to make one
2 Factory for Dating Events

```java
class Event {
    protected String name;
    protected int firstAllowedDate = Integer.MAX_VALUE; // fail hard if no init

    // ...

    /*
     * static Factory methods, for convenience and correctness.
     * Note that Date can't even tell if Event has subclasses.
     *
     * public static Event makeSeeMovie() { return new Event(1, "SeeMovie"); }
     * public static Event makeGoToRestaurantEvent() {
     *    return new Event(1, "GoToRestaurant");
     * }
     * public static Event makeOrderFlowers() {
     *    return new Event(2, "OrderFlowers");
     * }
     */
```
What’s wrong with 2¢ Factory?

A. Doesn’t hide construction details adequately
B. Should pass an enum argument rather than defining multiple methods:
   \[\text{Event.makeEvent(SEEMOVIE)}\]
C. Violates the Open/Closed principle
D. “make” methods shouldn’t be static
E. Name prefix should be “create”, not “make”
Simple Factory (idiom)

```java
public enum DatingEvent { MOVIE, RESTAURANT, FLOWERS }

class SimpleEventFactory {
    public Event createEvent(DatingEvent eventEnum) {
        switch (eventEnum) {
            case MOVIE:
                return makeSeeMovie();
            case RESTAURANT:
                return makeGoToRestaurant();
            case FLOWERS:
                return makeOrderFlowers();
        }
    }

    // static Factory helper methods.
    protected static Event makeSeeMovie() { return new Event(1, "SeeMovie"); }
    protected static Event makeGoToRestaurantEvent() {
        return new Event(1, "GoToRestaurant");
    }
    protected static Event makeOrderFlowers() {
        return new Event(2, "OrderFlowers");
    }
}
```

How to make Events not necessarily part of Event
Comparison – increasingly abstract

class MyDatingClass {
...
    Event event =
        new Event(2, "OrderFlowers"); // magic constants

    // concrete Event class (majorly violates OCP)
    Event event = Event.makeOrderFlowers();

    // abstract class object (passed into constructor)
    Event event = factory.createEvent(FLOWERS);
Class Diagram for Simple Factory

This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes.

This is the product of the factory: pizza!

We’ve defined Pizza as an abstract class with some helpful implementations that can be overridden.

These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means “extend the abstract Pizza class”) and be concrete. As long as that’s the case, it can be created by the factory and handed back to the client.

This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.
What’s wrong with Simple Factory?

A. create method has to know about all the concrete classes it new’s

B. Client class composes concrete class

C. Factory is a concrete class
Simple Factory is Good, but not OCP

- SimpleEventFactory should implement EventFactory interface
- Clients of SEF should then compose EventFactory:

```java
class DateClass {
  private EventFactory factory;

  public DateClass(EventFactory factory) {
    this.factory = factory;
  }
  ...
```

- Someone can implement a new class with EventFactory interface, and DateClass can use it with **no modification**

- Means that when EventFactory is extended (new subclass), DateClass is extended with new EventFactory options.

- That’s **open for extension!**
Subclasses of Date are the factories!

In this example, no parallel Date/Event class hierarchy
- Event hierarchy is “flat” – class hierarchy is implicit in data of Event

What changes is time to first “flowers” event, what the flowers are, etc.
Factory Method Style: Separate Factory

- Separate Factory like SEF but in hierarchy with abstract super
- Super centralizes Event selector
- Subs implement making
Factory Motivation for Pizza Example: Franchising

Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Product classes

Pizza

NYStyleCheesePizza

NYStylePepperoniPizza

NYStyleClamPizza

NYStyleVeggiePizza

ChicagoStyleCheesePizza

ChicagoStylePepperoniPizza

ChicagoStyleClamPizza

ChicagoStyleVeggiePizza

The Creator classes

PizzaStore

createPizza()

orderPizza()

NYPizzaStore

createPizza()

The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas.

NYPizzaStore encapsulates all the knowledge about how to make NY style pizzas.

The factory method is the key to encapsulating this knowledge.
“composes” them all because their constructors are referenced.

```java
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
            if (item.equals("pepperoni")) {
                return new NYStylePepperoniPizza();
            } else {
                return null;
            }
        } else {
            return null;
        }
    }
}
```
Object Relationships

The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.

The abstract `factoryMethod()` is what all Creator subclasses must implement.

The ConcreteCreator implements the `factoryMethod()`, which is the method that actually produces products.

The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.
Adapter Pattern

Again? Composition and Delegation, again?
Adapter: Motivation

Real-world examples:
• Database connectors
• Chat systems

Adapter:
• Uses **Composition** and **Delegation**
• Adapter (wrapper) is **not subclass of wrapped object** (because we’re adapting)
• Naming: InputStreamToReaderAdapter
Adapter: Example

```java
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

First, you need to implement the interface of the type you’re adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts — they can’t do long-distance flying like ducks. To map between a Duck’s fly() method and a Turkey’s, we need to call the Turkey’s fly() method five times to make up for it.
**Adapter: Class Diagram**

- **Client**
- **Adapter**
- **Target**
- **Adaptee**

**All the classic DP elements:**
- Client is “open” because it **composes interface**
- Adapter **implements interface**, **delegates** to concrete class
Adapter: Exercise

- Java’s StringTokenizer class breaks a string into words using a specific algorithm.
- Oddly, it implements the Enumeration\(<E>\) interface instead of Iterator\(<E>\) interface.
- Need to pass the StringTokenizer to a class that requires (takes) an iterator.
- Write an adapter from Enumeration to Iterator.
- (Could do this for StringTokenizer, but that’s less reusable!)

```java
interface Enumeration\(<E>\) {
    boolean hasMoreElements();
    E nextElement();
}

interface Iterator\(<E>\) {
    boolean hasNext();
    E next();
}
```
Adapter Exercise (solution)

```java
public class EnumerationIterator implements Iterator {
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

USE:
Iterator<String> iter = new EnumerationIterator<String>(myStringTokenizer);
while(iter.hasNext()) {
    String str = iter.next();
    System.out.print(str + " ");
}
```