Program to Interface, Factories and Injection

Program to an Interface, Not an Implementation

Early in the **WeatherStation** constructor:

```
Barometer bar = new Barometer() ;
```

Why is this problematic:

- **`WeatherStation`** depends on a specific *implementation* of a **`Barometer`**.
- In particular, here a simulated **`Barometer`**!
- Each different type of **`Barometer`** makes us change the **`WeatherStation`**.
- Or jury-rig some sort of way of loading the "right" **`Barometer`**.
- Reduces the reusability of **`WeatherStation`** in different contexts.

Program to an Interface, Not an Implementation

Somewhere in the **WeatherStation** :

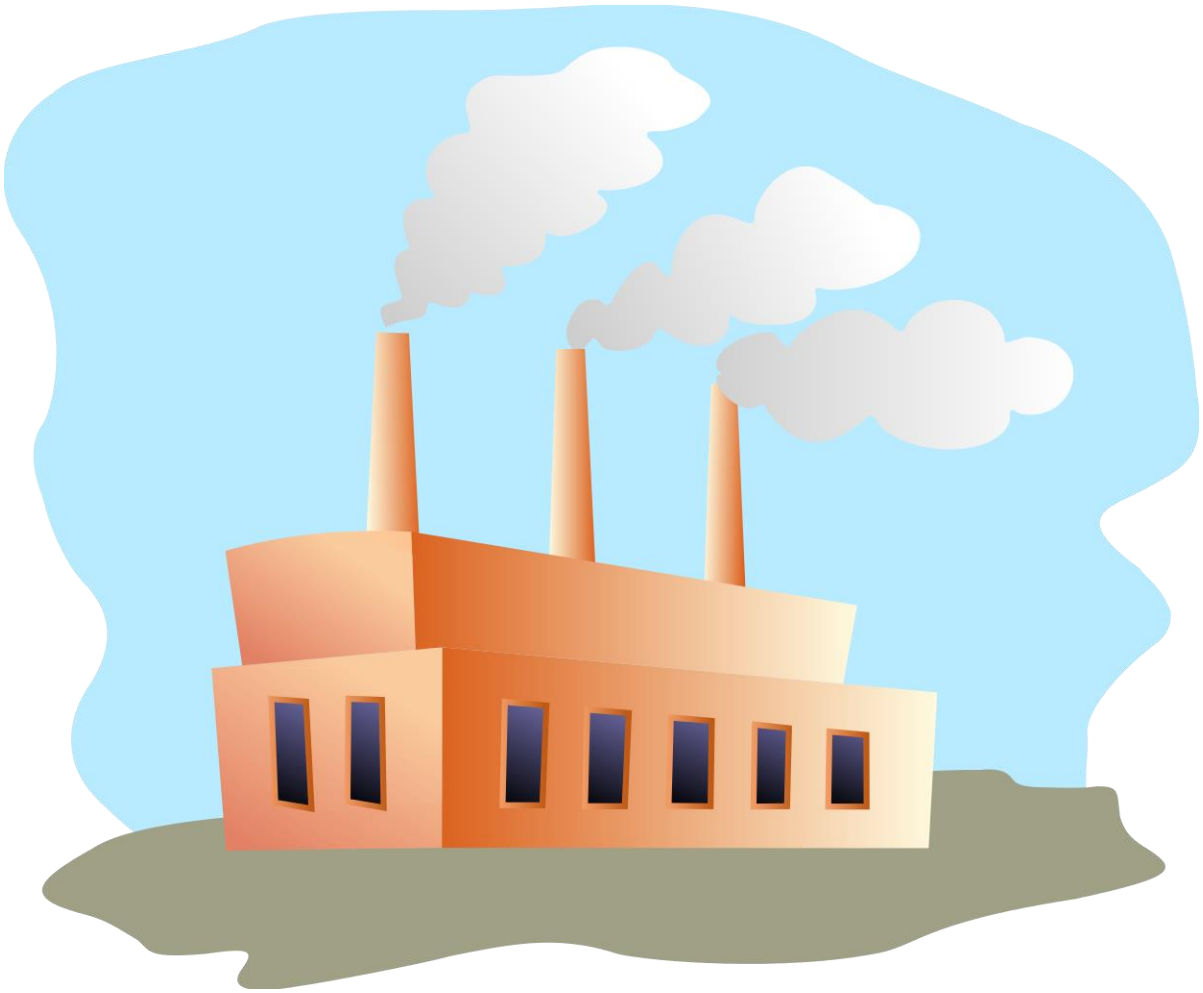
```
If (intBarometerType == 1) {
    barObj = new Barometer1() ;
}

If (intBarometerType == 2) {
    barObj = new Barometer2() ;
}
```

Issues:

***new* keyword gets used/abused**

Client is aware of all Barometer types

(need to recompile when you add a new type!)



P2I: Factories

Go ask someone else for what we need:

- Some other *known* class (static factories):

  java.util.System:

  `public Console console() ;`

  - Get object to read and write to the application's console.
  - How this is done is inherently system dependent.
  - Different on Windows, Linux, Mac

P2I: Factories

Go ask someone else for what we need:

- Some other *known* class (static factories)
- Some other *known* object (dynamic factories)

java.util.Runtime:

```
Runtime rt = Runtime.getRuntime() ;

. . .

Process p = rt.exec("some system command") ;
```

- Processes are inherently system dependent.
- As is how to create one to execute a command.

Factory Method Pattern

Often we have related, parallel hierarchies.

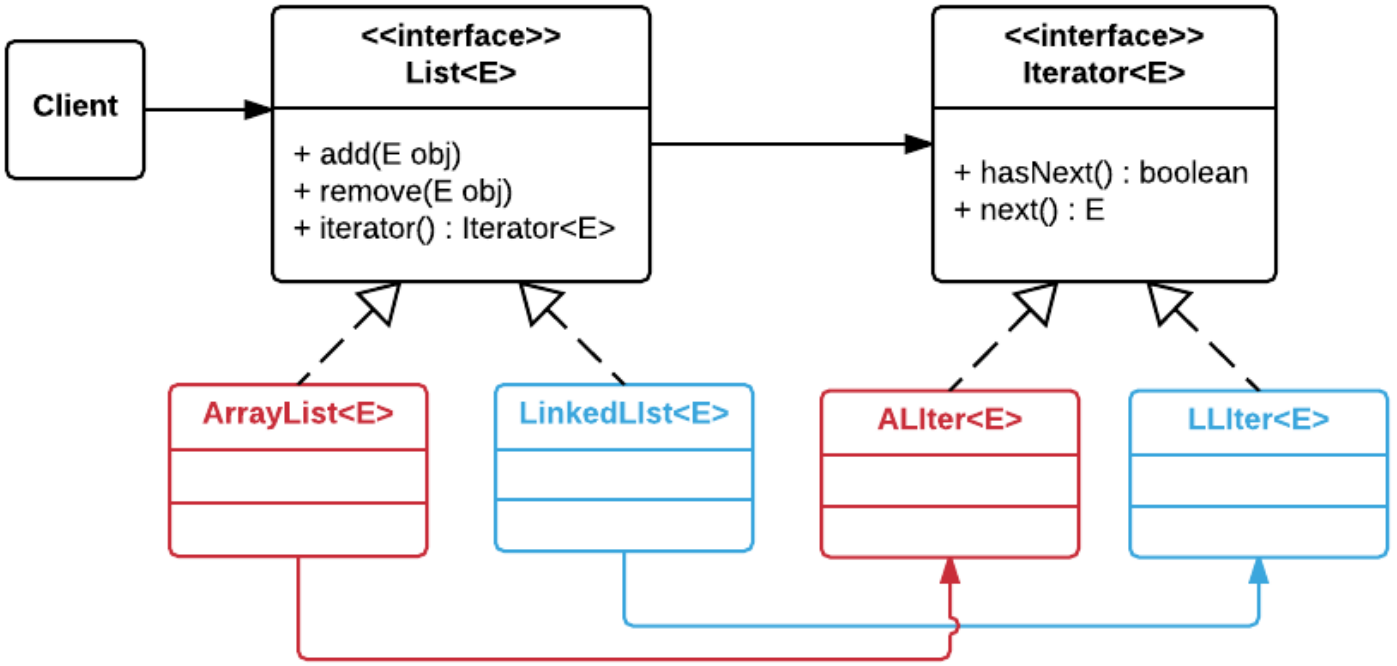
Factory method uses an object in one hierarchy to retrieve an appropriate object from the other hierarchy.

Factory Method Pattern

Often we have related, parallel hierarchies.

Factory method uses an object in one hierarchy to retrieve an appropriate object from the other hierarchy.

Example: Java Collections and their Iterators.



Abstract Factory Method

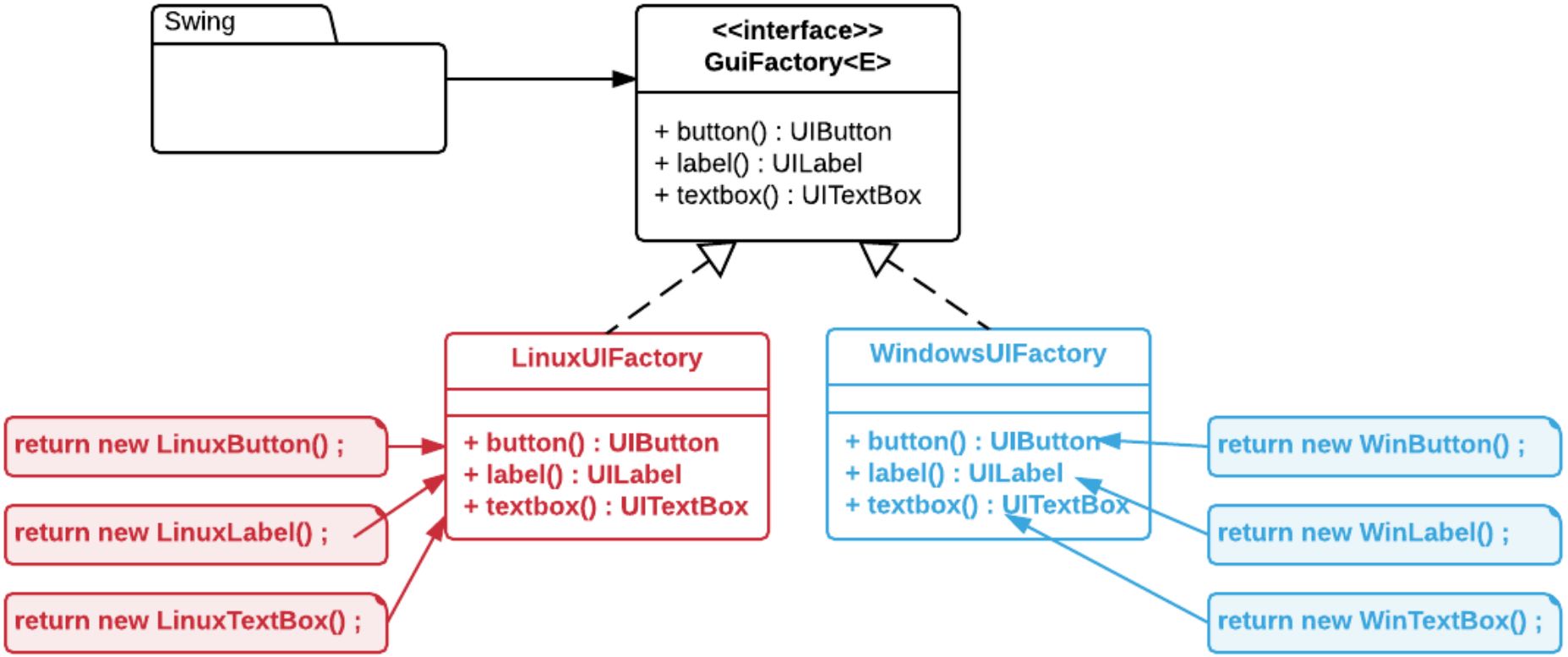
Sometimes we have different implementations of a set of related objects.

Implementations are not interchangeable.

Example: Swing

- Swing provides system independent classes for system dependent UI elements.
- That is, JButtons look like X-Windows buttons on Linux and Windows buttons on Windows.
- Same from frames, text boxes, etc.
- Configure runtime with an object that returns the correct type of button, frame, etc. for the system we run on.

Swing Windowing Abstract Factory



Barometer & Factory - 1

First, use an abstract barometer class:

```
public abstract class Barometer {
    public static Barometer createBarometer() { ... }

    abstract public double pressure() ;
}
```

Next, change the simulated barometer class and all other barometer classes to

1.Have a different name, and
2.Extend the abstract class above

```
public class SimBarometer extends Barometer { ... }

public class RealBarometer extends Barometer { ... }
```

Barometer & Factory - 2

Change the factory **`createBarometer()`** method to return the desired specific, concrete barometer object.

```
public static Barometer createBarometer() {
    return new SimBarometer() ; // or something else
}
```

Inside the **`WeatherStation`**, use the factory to get the **Barometer** to use:

```
public WeatherStation {

    private final Barometer bar ;

    public WeatherStation() {
        bar = Barometer.createBarometer() ;
    }

    . . .
```

P2I: Factory Pattern Points

Factory Approach Advantages

- Decouple the class needing an object from the creation of a concrete object.
- Always get compatible objects (esp. with Abstract Factory).

Factory Approach Disadvantages

- Still have to know where the factory is.
- Coupled to the factory class or object.

Can we decouple even more? Yes - *dependency injection*.

Dependency Inversion Principle (DIP)

- Low-level components should depend on high-level components, not the other way around.

- OR -

- High-level components should not depend on low-level components. **Both should depend on abstractions**.

- Abstractions should not depend on details (of low level entities). Details should depend on abstractions.

- OR -

- High-level components control the interface to low-level components.

One use of DIP is through dependency **injection** pattern

P2I: Dependency Injection Pattern

Dependency Injection - target object (e.g., **`WeatherStation`**) is given the objects it needs (e.g., barometer and temperature sensor) from the surrounding environment:

- Via **constructor** arguments.
- Via specific **injection methods**.

In fact, we've already seen this:

P2I: Dependency Injection Pattern

Dependency Injection - target object (here WeatherStation) is given the object it needs from the surrounding environment:

- Via constructor arguments.
- Via specific injection methods.

In fact, we've already seen this:

Anything compatible with **WeatherStation** can be substituted.

```
public class TextUI implements Observer {

    private final WeatherStation station ;

    public TextUI(WeatherStation station) {
        this.station = station ;
        this.station.addObserver(this) ;
    }
```

Dependency Injection & Barometer - 1

First, define a barometer *interface*:

```
public interface Barometer {
    public double pressure() ;
}
```

Next, change the simulated barometer class and all other barometer classes to

1.Have a different name, and
2.Implement the interface above

```
public class SimBarometer implements Barometer { ... }
public class RealBarometer implements Barometer { ... }
```

Dependency Injection & Barometer - 2

Change the **`public static void main()`** method to inject whichever concrete barometer we want:

```
public static void main(String[] args) {
    . . .
    WeatherStation ws =
        new WeatherStation( new SimBarometer() );
}
```

In the **`WeatherStation`** constructor, save the injected **`Barometer`**:

```
public WeatherStation {
    private final Barometer bar ;
    public WeatherStation(Barometer bar) {
        this.bar = bar ;
    }
    . . .
```

Now To The Temperature Sensor

Early in the **`WeatherStation`** constructor

```
KelvinTempSensor sensor = new KelvinTempSensor() ;
```

On the surface this looks exactly like **`Barometer`**:

- We create a concrete sensor object in the weather station.
- This limits weather station reusability with different sensors.
- So:
  - Define an interface, say **`IKelvinTempSensor`**.
  - Implement the interface for all real & simulated sensor classes.
  - Create the desired concrete sensor in main or other driver method.
  - Inject this object into the **`WeatherStation`** constructor.

But there is more here than meets the eye!

Problems With The Temperature Sensor

- The interface represents an "odd" notion of what temperature looks like:
  - Scaled integer from 0 to 65535?
  - Measures up to 655.35 °K?
  - That's a weird upper bound - why is it there?
- The designers thought the problem was selecting an integrating the best sensor.

Problems With The Temperature Sensor

- The interface represents an "odd" notion of what temperature looks like:
  - Scaled integer from 0 to 65535?
  - Measures up to 655.35 °K?
  - That's a weird upper bound - why is it there?
- The designers thought the problem was selecting an integrating the best sensor.
- The designers were **WRONG**!
- The **real problem** is how to hide the details of specific sensor used from the weather station.
- All the weather station needs is a general value representing the temperature in some reasonable form.

Design Caveats (Uncle Bob Martin)

- Woe is the designer who prematurely decides on a database, and then finds that flat files would have been sufficient.
- Woe is the designer who prematurely decides upon a web-server, only to find that all the team really needed was a simple socket interface.
- Woe is the team whose designers prematurely impose a framework upon them, only to find that the framework provides powers they don't need and adds constraints they can't live with.
- Blessed is the team whose designers have provided the means by which all these decisions can be deferred until there is enough information to make them.
- Blessed is the team whose designers have so isolated them from slow and resource hungry IO devices and frameworks that they can create fast and lightweight test environments.
- Blessed is the team whose designers care about what really matters and defer those things that don't.