Design Patterns and Idioms in JAVA

Martina Umlauft
umlauft@ftw.at

24. Juli 2001

ftw. Forschungszentrum Telekommunikation Wien

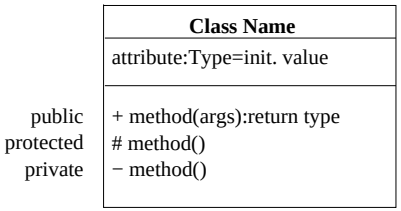
Overview

- Introduction to UML
- Review JAVA / OO
- **Design Patterns**
  discussion after each
- Summary
- References

→

- **Typesafe Constant Idiom**
- **Callback Idiom / Observer Pattern**
- **Singleton Pattern**
- **Factory Pattern**

Introduction to UML / 1



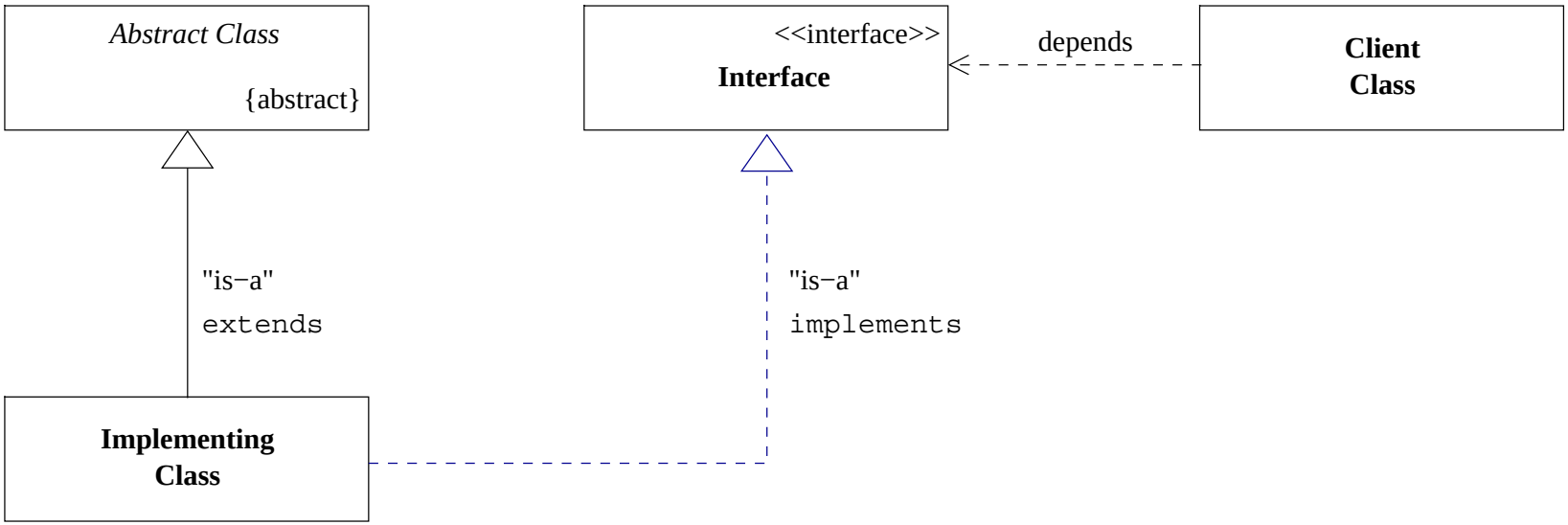
Class

- **Type** used to construct classes
- **Loaded** by classloader
- **Class Members** `static`

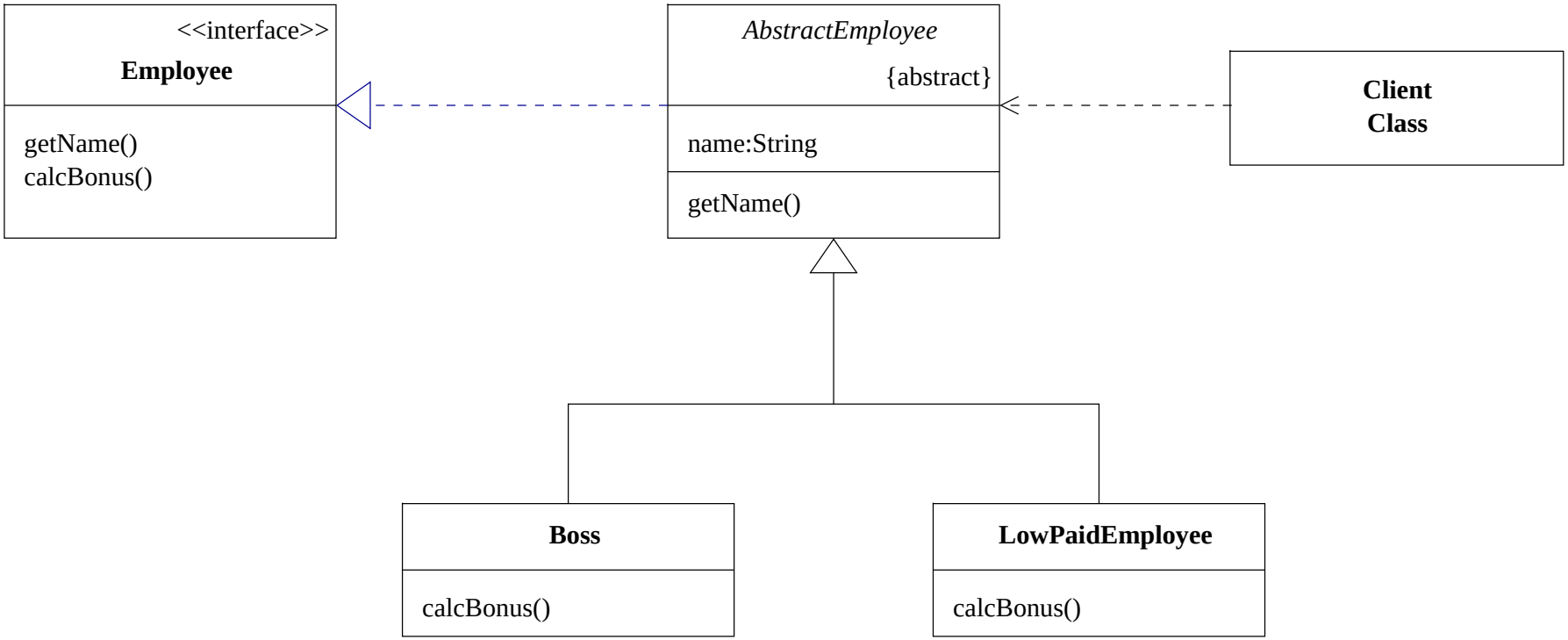
Object

- **Constructed** with `new`
- **Members** one copy per object
- **Variables** hold *references* to objects

Introduction to UML / 2



Review JAVA / OO / 1



Review JAVA / OO / 2

```
public interface Employee {

    public String getName();
    public float calcBonus();
}
```

Review JAVA / OO / 3

```
public abstract class AbstractEmployee implements Employee {

    private String name;

    public AbstractEmployee(String name) {

        this.name=name;
    }

    public String getName() {
        return name;
    }
}
```

Review JAVA / OO / 4

```
public class Boss extends AbstractEmployee {

    public Boss(String name) { super(name); }

    public float calcBonus() {

        // return ridiculously high value
    }
}

public class LowPaidEmployee extends AbstractEmployee {

    public LowPaidEmployee(String name) { super(name); }

    public float calcBonus() { return 0.0; }
}
```

Review JAVA / OO / 5

```
public class ClientClass {

    public void calcBonii(Employee emp) {

        System.out.println(emp.getName() + " gets:" + emp.calcBonus());
    }
}

...
ClientClass backoffice = new ClientClass();
backoffice.calcBonii(new Boss("Alex"));
backoffice.calcBonii(new LowPaidEmployee("StudentX"));
```

Typesafe Constant Idiom / 1

Example for traditional constants

```
package java.awt;

public class Font {

    public static final int PLAIN  = 0;
    public static final int BOLD   = 1;
    public static final int ITALIC = 2;

    public Font(String name, int style, int size) { ...
    }
}
```

What happens if someone mixes up style and size?

Typesafe Constant Idiom / 2

Typesafe constants

```
public final class AlienRace {

    public static final AlienRace BORG    = new AlienRace();
    public static final AlienRace VULCAN  = new AlienRace();
    public static final AlienRace FERENGI = new AlienRace();

    private AlienRace();
}
```

- class `final` to prevent subclassing
- constructor `private` to prevent fake constants

Typesafe Constant Idiom / 3

```
public void constTest(AlienRace alien) {

    if(alien == AlienRace.BORG)
        System.out.println("Resistance is futile!");

    if(alien == AlienRace.VULCAN)
        System.out.println("Fascinating.");
}
```

- wrong constants impossible
- no range-checking necessary

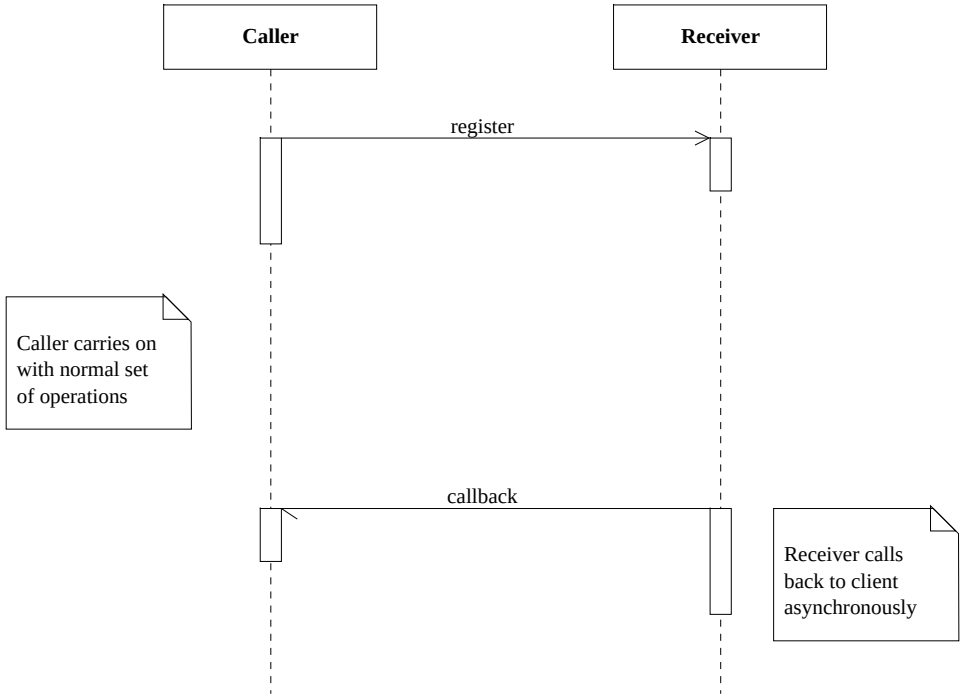
Callback Idiom / Observer Pattern / 1

Also known as “Listener”.

Application

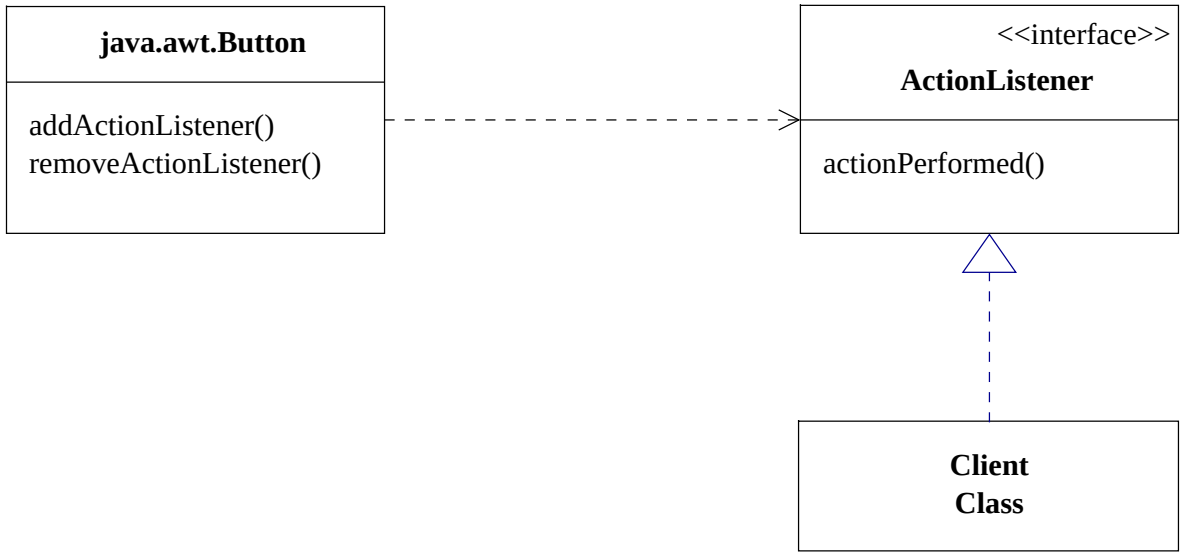
- abstraction with 2 aspects, one dependent on the other (MVC)
- change to one object requires change to others (number unknown)
- object should be able to notify others without knowing them
- when you would use a function pointer in C

Callback Idiom / Observer Pattern / 2



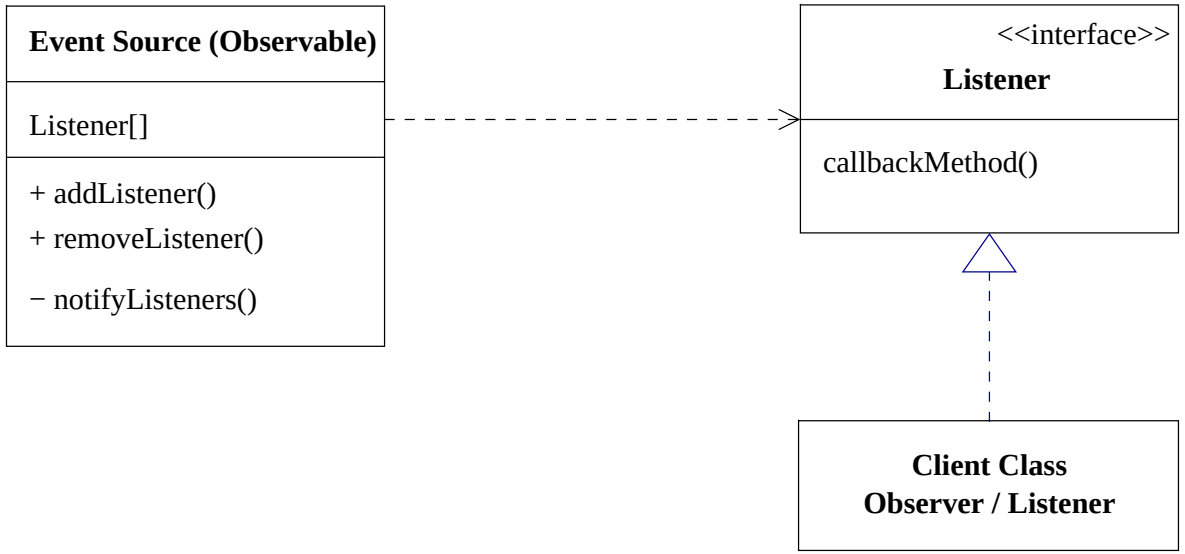
Callback Idiom / Observer Pattern / 3

Example: java.awt.Button



Callback Idiom / Observer Pattern / 4

Inner workings



Callback Idiom / Observer Pattern / 5

```
public class EventSource {

    private Vector listeners = new Vector();  // from java.util

    public void addListener(Listener l) {

        listeners.add(l);
    }

    public void removeListener(Listener l) {

        listeners.remove(l);
    }

    private void notifyListeners() {          // explained later
    }
}
```

Callback Idiom / Observer Pattern / 6

```
public interface Listener {

    public void callbackMethod( /* possible parameters */ );
}
```

- classes who want to listen must implement this
- hides actual type of listeners

Callback Idiom / Observer Pattern / 7

```
public class Client implements Listener {

    public Client(EventSource source) {

        source.addListener(this);
    }

    public void callbackMethod() {

        // act; e.g. update window
    }
}
```

Callback Idiom / Observer Pattern / 8

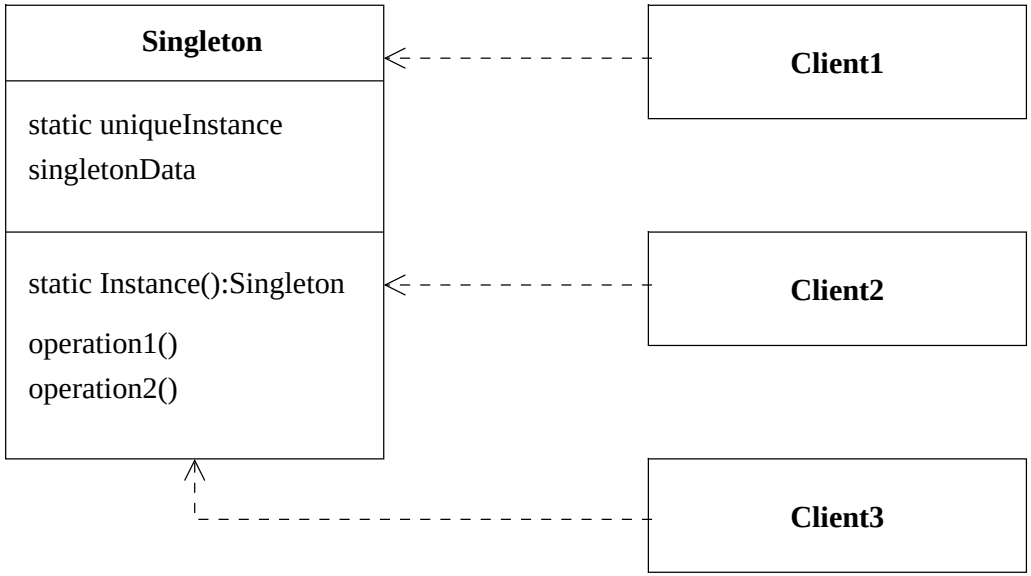
```
public class EventSource {

    Vector listeners = new Vector();

    private void notifyListeners() {

        Iterator iter = listeners.iterator();

        while(iter.hasNext()) {

            Listener currentListener = (Listener) iter.next();

            currentListener.callbackMethod(); // we never know the real
                                              // type of the Listeners (!)
        }
    }
}
```

Singleton Pattern / 1

Application

- ensure some class has only 1 instance
- provide global point of access (avoid handing references around)
- possible to change to more than 1 instance (e.g. a pool)

Singleton Pattern / 2



Singleton Pattern / 3

```
public class Singleton {

    static Singleton uniqueInstance = new Singleton();
    private String someData = "....";          // Example

    protected Singleton() { }                  // "private" Constructor

    public static Singleton Instance() {       // also known as getInstance()
                                               // or sharedInstance()
        return uniqueInstance;
    }

    public void operation() {                  // some service the singleton provides
    }
}
```

Singleton Pattern / 4

```
public class Client {

    public void doSomething() {

        Singleton s = Singleton.Instance();  // wherever we are we can
                                             // immediately get a
                                             // reference to the server

        s.operation();                       // now use it
    }
}
```

Factory Pattern / 1

Also known as "Kit". Group of patterns: Factory method, Abstract Factory.

Application

- system should be independent of object creation
- system configurable for multiple families of objects
- enforce that a family of related objects is used together

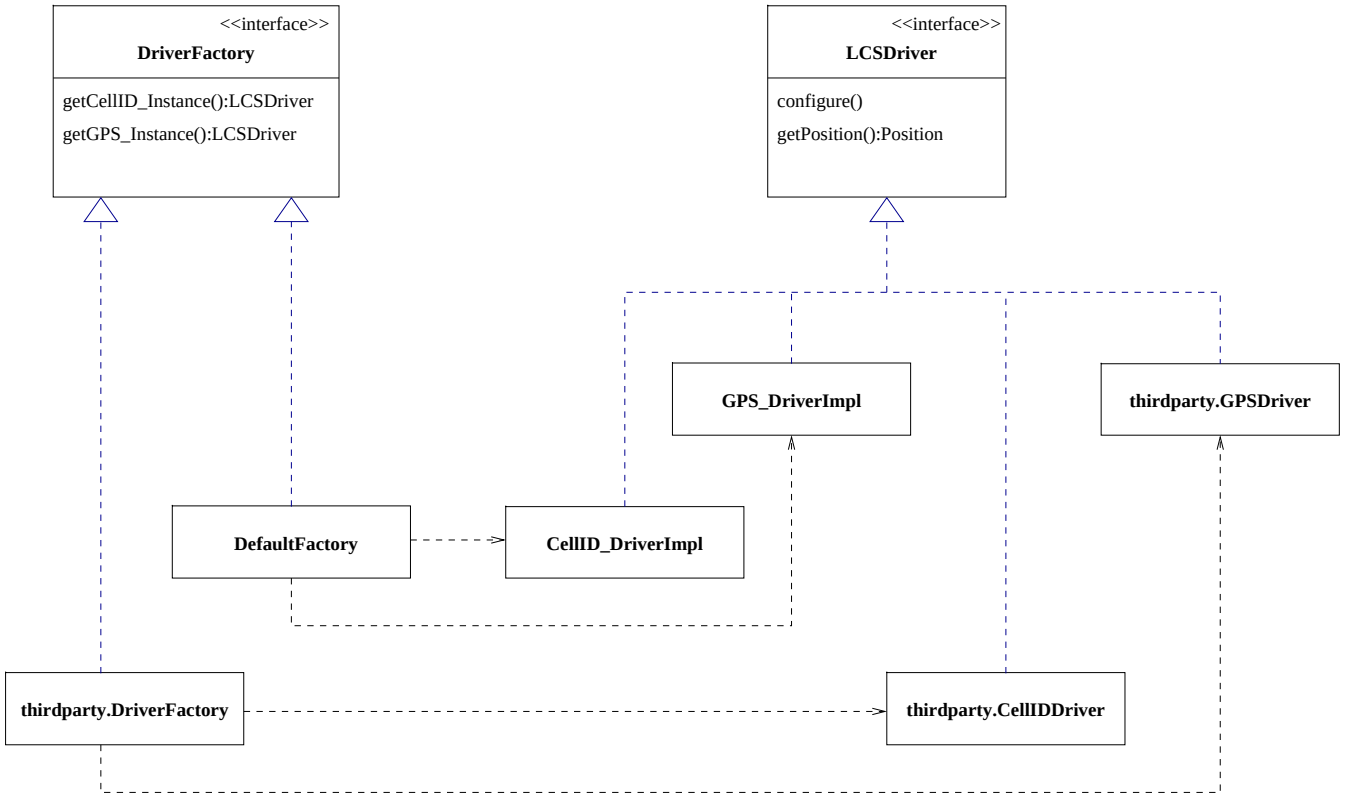
Example: windowing toolkits with different look & feels

Factory Pattern / 2

Problem: can't specify constructor in interface.

```
public interface LCSDriver {

    public LCSDriver();                          // not legal !!

    public void configure();
    public Position getPosition();
    ...
}

...
LCSDriver lcs = new GPS(impl specific param); // -> would need this in client code
lcs.configure();
...
```

Factory Pattern / 3



Factory Pattern / 4

```
public interface LCSDriver {

    public void configure();
    public Position getPosition();

}

public interface DriverFactory {

    public LCSDriver getCellID_Instance();
    public LCSDriver getGPS_Instance();

}
```

Factory Pattern / 5

```
public class DefaultFactory implements DriverFactory {

    public LCSDriver getCellID_Instance() {

        return new CellID_DriverImpl();
    }

    public LCSDriver getGPS_Instance() {

        return new GPS_DriverImpl( /* special params */ );
    }
}
```

Factory Pattern / 6

Traditional solution for client code

```
...
LCSDriver driver;

if(driver_type == CELLID)
    driver = new CellID_DriverImpl();

if(driver_type == GPS)
    driver = new GPS_DriverImpl( /* special params */ );
...
```

$\longrightarrow$ **Client code would depend on implementation details!**

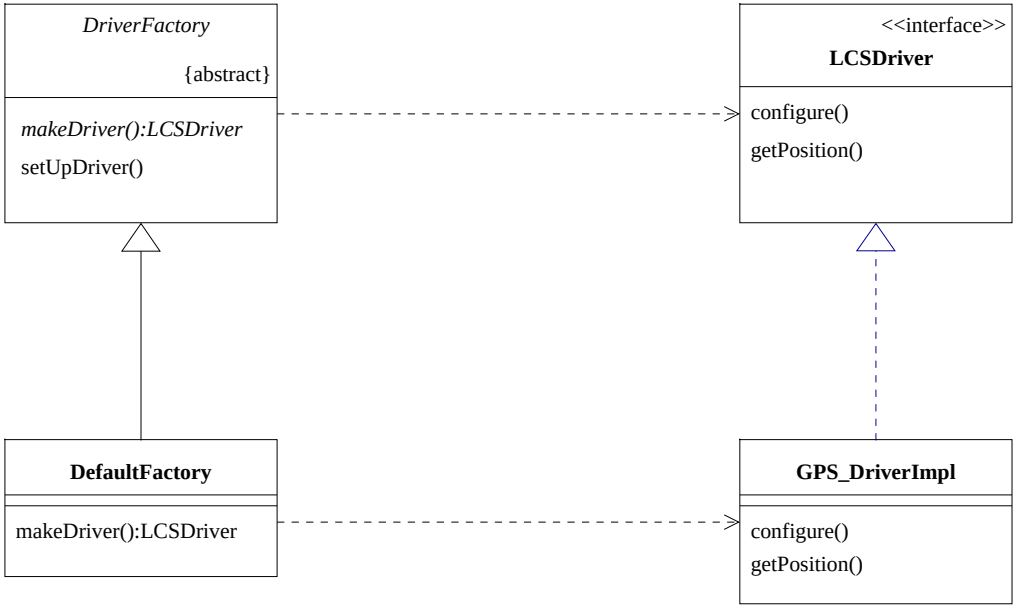
Factory Pattern / 7

Solution with Abstract Factory Pattern

```
public class LCS_User {

    public static void useDrivers(DriverFactory df) { // method not coupled to concrete
                                                       // factory or product instance
        LCSDriver gps = df.getGPS_Instance();
        Position pos = gps.getPosition();
    }
}

...                                                    // main
DriverFactory c1_Factory = new DefaultFactory();
DriverFactory commercial_Factory = new thirdparty.DriverFactory();

LCS_User.useDriver(c1_Factory);
LCS_User.useDriver(commercial_Factory);
```

Factory Pattern / 8

Factory method



Summary / 1

Typesafe Constant Idiom

- better, unforgeable constants
- no range-checking necessary
- compiler-checkable(!)

Callback Idiom / Observer Pattern

- used widely in AWT and Swing
- use instead of C function pointers
- decouples Observer(Listener) implementation from event source.

Summary / 2

Singleton Pattern

- guarantees exactly one instance in the system
- avoids having to hand around references to this central instance
- possible to change to more than one instance later

Factory Pattern

- decouples object creation details
- allows for system configuration with multiple families of objects
- enforces that families of objects are used together

References

UML Distilled Second Edition - *A Brief Guide to the Standard Object Modeling Language*, Fowler, Scott, Addison-Wesley, ISBN 0-201-65783-X

Design Patterns - *Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, ISBN 0-201-63361-2

Java in Practice - *Design Styles and Idioms for Effective Java*, Warren, Bishop, Addison-Wesley, ISBN 0-201-36065-9

Essential JAVA Style - *Patterns for Implementation*, Langer, Prentice Hall, ISBN 0-13-085086-1