♦ *Factory* (es: nome+cognome, tipi di media)

You create an abstraction which decides which of several possible classes to return and returns one. Then you call the methods of that class instance without ever knowing which derived class you are actually using.
This approach keeps the issues of data dependence separated from the classes' useful methods.

You should consider using a Factory pattern when:

- A class can't anticipate which kind of class of objects it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize.

- The base class is abstract and the pattern must return a complete working class.
- The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
- Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

♦ *Abstract Factory* (es: vehicle, giardino)

You can use this pattern when you want to return one of several related classes of objects, 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.

One classic application of the abstract factory is the case where your system needs to support multiple look-and-feel user interfaces, such as Windows-9x, Motif or Macintosh.
You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.
Then when you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

♦ *Builder* (es: builder personalizzato per ogni famiglia di vehicle)

A Builder pattern is somewhat like an Abstract Factory pattern in that both return classes made up of a number of methods and objects.
The main difference is that while the Abstract Factory returns a family of related classes, the Builder constructs a complex object *step by step* depending on the data presented to it.

Builder characteristics:

- A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
- Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
- Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.

♦ *Singleton* (es: vehicleComparators)

There are any number of cases in programming where you need to make sure that there can be one and *only one instance* of a class.

- a class that throws an Exception when it is instantiated more than once:

```
class SingletonException extends RuntimeException {
  public SingletonException () { super(); }
  public SingletonException (String s) { super(s); }
}//SingletonException

class PrintSpooler {
  static boolean instFlag=false; //true if 1 instance
```

```
  public PrintSpooler () throws SingletonException {
    if (instFlag)
      throw new SingletonException("Only one spooler allowed");
    else
      instFlag=true; //set flag for 1 instance
    System.out.println("spooler opened");
  }//PrintSpooler
  public void finalize () {
    instFlag=false; //clear if destroyed
  }//finalize
}//PrintSpooler
```

- making a final class; you can't create any instance of it, and can only call the static methods directly in the existing final class:

```
final class PrintSpooler {
  static public void print (String s) { System.out.println(s); }
}//PrintSpooler
```

- create Singletons using a static method to issue and keep track of instances; to prevent instantiating the class more than once, we make the constructor private so an instance can only be created from within the static method of the class:

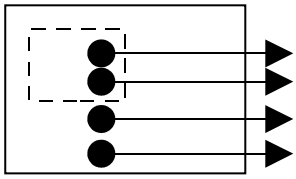
```
class iSpooler {
  static boolean instFlag=false; //true if 1 instance
  private iSpooler () { /* need no content */ }
  static public iSpooler Instance () {
    if (!instFlag) {
      instFlag=true;
      return new iSpooler(); //only callable from within
    }//if
    else
      return null; //return no further instances
  }//iSpooler
  public void finalize () {
    instFlag=false;
  }//finalize
}//iSpooler
```

♦ ***Adapter*** (es: Hashtable → HashtMap)

The Adapter pattern is used to *convert* the programming interface of one class into that of another.

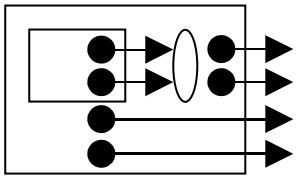
We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface:

- class adapter (by inheritance): we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface.



```
adapter extends obj
implements interface {
  ...
}//adapter
```

- object adapter (by object composition): include the original class inside the new one and create the methods to translate calls within the new class.



```
Class adapter {
  Obj o ;
  ...
}//adapter
```

♦ ***Composite*** (es: foglia o radice di un albero, organigramma di un'azienda, rpn calc)

A composite is a collection of objects, any one of which may be either a composite, or just a primitive object.

- creating methods for nodes and leaves, where a leaf could have the methods:

```
public String getName();
public String getValue();
```

- and a node could have the additional methods:

```
public Enumeration elements();
public Node getChild(String nodeName);
public void add(Object obj);
public void remove(Object obj);
```

♦ ***Command*** (es: azione di un comando di menu)

Separa l'azione dalla conoscenza delle operazioni che essa esegue.
Separa l'invocatore dall'esecutore:

Invocatore → Esecutore (gli viene passato lo stato dell'applicazione)

♦ ***Observer*** (es: impianto a sensori con eventi)

The Observer pattern assumes that the object containing the data is separate from the objects that display the data, and that these display objects observe changes in that data.