Università degli Studi di Milano – Bicocca
DISCo – Dipartimento di Informatica, Sistemistica e Comunicazione

A Design Pattern Clues Catalogue

Contents

A Design Pattern Clues Catalogue

This catalogue aims at collecting the Design Pattern Clues identified so far inside all of the *Gang of Four* design pattern categories.

Up to now, we found 45 Design Pattern Clues, further subdivided into 9 categories, each of them collecting those elements that refer to a particular code structure. 20of the 45 Clues have been identified analyzing the creational design patterns category, 8 are related to the behavioural ones, and the remaining 17 have been defined upon the architectures of the structural design patterns.
The 9 categories of Design Pattern Clues we have defined to collect the 45 Clues are:

- *Class Declaration Information*: collects Clues that can be identified analyzing a class declaration;
- *Multiple Classes Information*: collects Clues that can be identified by the comparison of at least two classes and their contents;
- *Variables Information*: identifies useful information related to particular variables;
- *Instance Information*: collects information about particular instances of a class and specifies a peculiar instantiation technique;
- *Method Signature Information*: Clues belonging to this category are detected analyzing the method signatures;
- *Method Body Information*: contains those Clues that can be identified by only analyzing the body of any kind of methods;
- *Method Set Information*: collects Clues whose details can be deducted analyzing the whole set of methods the involved classes declare and implement;
- *Return Information*: specifies the information that can be obtained from a method return formalities;
- *Java Information*: collects Clues more strictly bound to some Java constructs.

For each Clue, its description will be subdivided in the following core points:

- *Meaning*: the meaning of the Clue, i.e. which particular code element the Clue identifies;
- *Belongs to*: the design pattern(s) inside which the Clue was detected;
- *Structure*: an UML class diagram showing an example class implementing the Clue and other possible related classes;
- *Expression*: an expression of the Clue, i.e. its name followed by all the information that is necessary for its complete identification; this expression is not intended to be a formal explanation of the Clue, but it only collects those fields that are useful to “link” a Clue to its localization in the source code;
- *Implementation*: how the Clue has to or should be implemented;
- *Dependent on*: specifies which elements (that can be EDPs, Clues or Micro-patterns) are necessary in order for the Clue to be found, that is without which the Clue cannot exist;
- *Sample code*: a code snippet showing a real implementation of the Clue (extracted from the two implementations we analyzed before).

We now introduce five categories of Clues following the description scheme we have just defined. The Clues are described into sections, each of them collecting those Clues that belong to a particular Clues category. For each Clue, we will also report which is the design pattern category it was identified in.
In order to obtain the complete catalogue, please send an e-mail to one of the following addresses:

stefano.maggioni@disco.unimib.it
arcelli@disco.unimib.it

Class Declaration Information

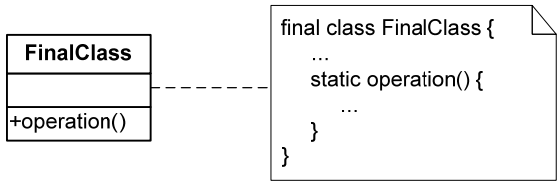
This category collects Clues that can be identified at a class declaration level. In this way, they can only be related to information about class modifiers, class inheritance or interface implementation. These last two kinds of information are already identified by an EDP; therefore, this category collects only information about class modifiers.
The *Class Declaration Information* category currently contains four Clues.

Final Class — Creational

Meaning
The class is declared `final`. The most important characteristic of a final class is that it cannot be extended.

Belongs to
Singleton

Structure



Expression
Final class (className)
className: the name of the class that is declared final.

Implementation
A class implementing this Clue is simply declared `final`. Most of the times the methods declared within it are all static. It is also possible that the class constructor is private, so that there cannot exist “uncontrolled” instances of the class.

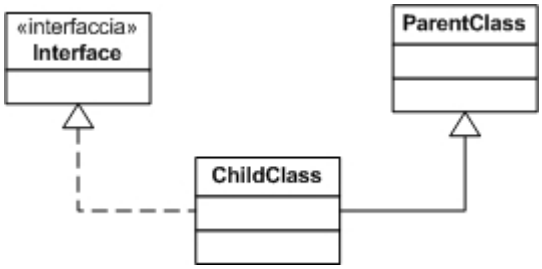
Sample code
```
final class FinalClass {
    …
}
```

Interface and Class Inherited — Structural

Meaning
The class implements an interface and extends a class, providing therefore the only mechanism to simulate multiple inheritance in the Java language.

Belongs to
Adapter (based on classes)

Structure



Expression
Interface and Class Inherited (className, interfaceName, parentName)
className: the name of the class implementing the Clue;
interfaceName: the name of the interface that is implemented by the class implementing the Clue;
parentName: the name of the class that is extended by the class implementing the Clue.

Implementation
No particular comments are to be made about the implementation of this clue, the class simply declares to be implementing an interface and extending another class. We can notice that this Clue is the union of two distinct Inheritance EDPs.

Dependent on
2 Inheritance EDP.

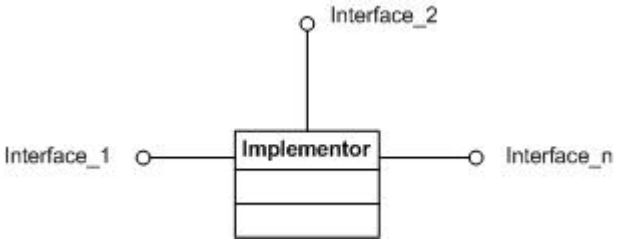
Sample code
```
class ChildClass implements Interface extends ParentClass {
        …
}
```

Multiple Interfaces Inherited — Structural

Meaning
The class implements *n* interfaces, with $n > 1$.

Belongs to
Adapter

Structure



Expression
Multiple Interfaces Inherited (className, interfaceNames)
classNames: the name of the class implementing the Clue;
interfaceNames: a list containing the *n* names of the *n* implemented interfaces.

Implementation
No particular comments are to be made about the implementation of this Clue, the class simply declares to be implementing more than one interface. We can notice that this Clue is the union of *n* distinct Inheritance EDPs.

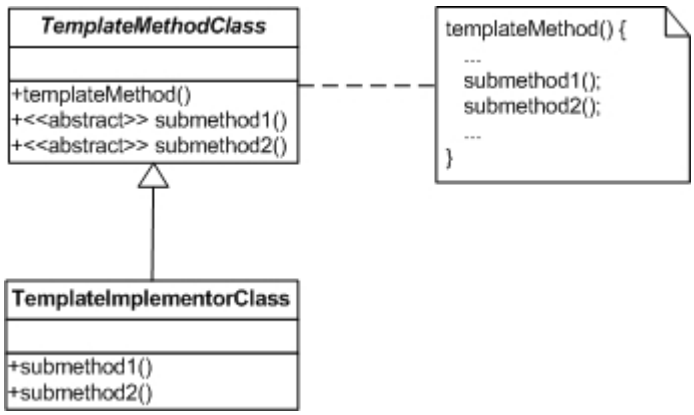
Dependent on
n Inheritance EDP.

Sample code
```
class Implementor implements Interface_1, Interface_2, …, Interface_n {
        …
}
```

Meaning
A class extends another class implementing a Template Method Clue.

Belongs to
Template Method

Structure



Expression
Template Implementor (className, parentClassName)
className: the name of the class implementing the Clue;
parentClassName: the parent class implementing a Template Method Clue.

Implementation
No particular comments are to be made about the implementation of this Clue. All classes that extend an abstract class which implements a Template Method Clue are all potential Template Implementors.

Dependent on
Template Method (Method Body Information category).

Sample code
```
abstract class TemplateMethodClass {
        public  void templateMethod() {
                …
                submethod1();
                submethod2();
                …
        }

        public abstract void submethod1();
        public abstract void submethod2();
}

public class TemplateImplementorClass extends TemplateMethodClass {
        …
        public void submethod1() { … }
        public void submethod2() { … }
        …
}
```

Multiple Classes Information

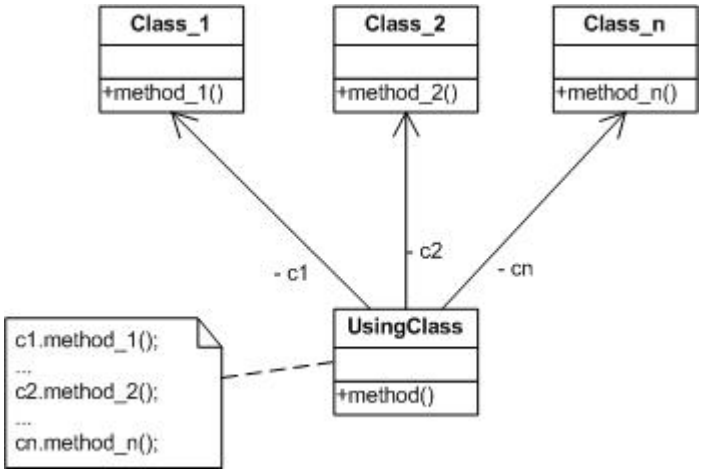
This category collects Clues that identify characteristics that can be identified by the comparison of at least two classes and their contents. This category currently contains two Clues.

Facade Method — Structural

Meaning
The body of a method consists uniquely of method calls to classes which are not related with it, i.e. which are not a superclass, an implemented interface or the class itself. A facade method could also contain some object creations, but no other statements besides object creations or method calls.

Belongs to
Facade.

Structure



Expression
Facade Method (className, facadeMethod)
className: the name of the class implementing the Clue;
facadeMethod: the name of the façade method.

Implementation
A class declares a method whose method calls are targeted to other unrelated classes, that is classes that do not appear in the inheritance hierarchy of the subject class.

Dependent on
N/A.

Sample code
```
public class UsingClass {
        Class_1 c1;
        Class_2 c2;
        …
        Classn cn;
        public void method() {
                …
```

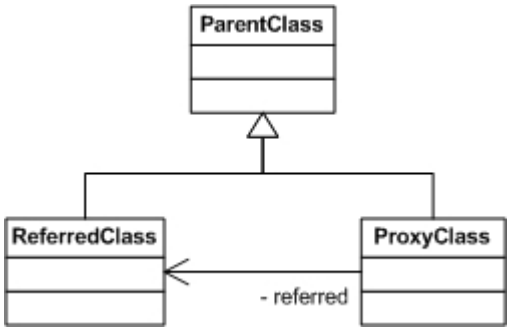
```
                c1.method_1();
                …
                c2.method_2();
                …
                cn.method_n();
        }
}

public class Class_1 {
        …
        public void method_1() { … }
}

public class Class_2 {
        …
        public void method_2() { … }
}

…

public class Class_n {
        …
        public void method_n() { … }
}
```

Proxy Class — Structural

Meaning
A class implements an interface or extends an (abstract) class, referring to a class that implements the same interface or extends the same (abstract) class.

Belongs to
Proxy.

Structure



Expression
Proxy Class (className, parentName, type, referenceName)
- className: the name of the class implementing the Clue;
- parentName: the name of the parent of the class implementing this Clue;
- type: interface or class, depending on the type of the parent;
- referenceName: the name of the reference which is declared by the class implementing the Clue and that has a common parent with it.

Implementation
As we can see from the picture regarding this Clue, we can see that it strictly satisfy the architecture of the Proxy design pattern. Its implementation is therefore highly related to the realization of the Proxy design pattern. This Clue, in conjunction with the Proxy Method Invoked Clue, is a really useful hint of the presence of a Proxy design pattern inside the code.

Dependent on
N/A.

Sample code
```
abstract class Parent {
                …
}

class ProxyClass extends Parent {
        …
        ReferredClass referred;
}

class ReferredClass extends Parent {
        …
}
```

Variables Information

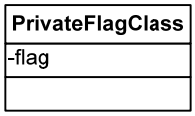
This category collects Clues identifying characteristics of some particular variables that can be found inside the code. There are only two Clues belonging to this group.

Private Flag — Creational

Meaning
The class maintains a control flag that is declared `private`. A flag belongs to a simple type (typically Boolean, but numerical fields are not to be excluded).

Belongs to
Singleton

Structure



Expression
Private flag (className, flagName, flagType)
- className: the class declaring the private flag;
- flagName: the flag name;
- flagType: the flag's data type, typically Boolean.

Implementation
The class declares a private variable.

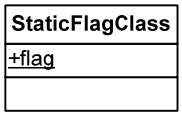
Sample code
```
class iSpooler {
        private static boolean instance_flag = false; //true if 1 instance

        private iSpooler() {}

        static public iSpooler Instance() { ... }
}
```

Static Flag — Creational

Meaning
The class maintains a control flag that is declared `static`. A flag belongs to a simple type (typically Boolean, but numerical fields are not to be excluded).

Belongs to
Singleton

Structure



Expression
Static flag (className, flagName, flagType)
- className: the class declaring the static flag;
- flagName: the flag name;
- flagType: the flag's data type, typically Boolean.

Implementation
The class declares a static variable. This field will be generally used inside the class to control the instantiation of objects belonging to the same class (see next section). The static variable is often declared `private` too, so that it cannot be modified by clients: therefore the class implements also the Private flag Clue, discussed next.

Sample code
```
class iSpooler {
        private static boolean instance_flag = false; //true if 1 instance

        private iSpooler() {}

        static public iSpooler Instance() { ... }
}
```

Instance Information

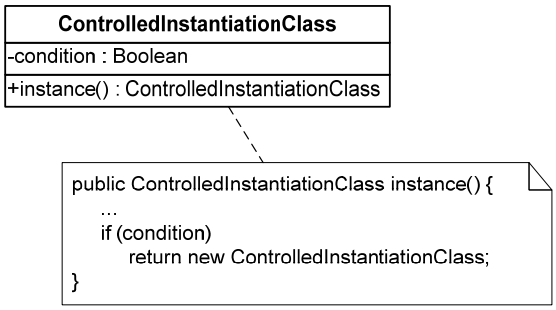
The Instance Information category collects information about particular created instances, and also suggests a controlled instantiation “pattern”. There are six Clues belonging to this category.

Controlled Self Instantiation — Creational

Meaning
The instantiation of an object occurs inside an `if` (or a `switch`) block, therefore under a condition.

Belongs to
Singleton

Structure



Expression
Controlled instantiation (className, methodName, derived)

> className: the class name which the created instance belongs to;
> methodName: the method which is responsible for the creation of the object;
> derived: a Boolean flag, which is true if the object belongs to a class that extends the current class, false otherwise.

Implementation
This Clue can be typically and conveniently found when a class declares a static method returning an object belonging to the same class. Inside this method we can find the object creation. This instantiation is under control of an `if` or `switch` block, that typically will work onto a flag or an instance that is declared static and/or private.

Sample code

```
public class Singleton {
        private static Singleton instance;

        protected Singleton() { ... }
        public static Singleton instance() {
                if (instance == null)
                        instance = new Singleton();
                return instance;
        }
}
```

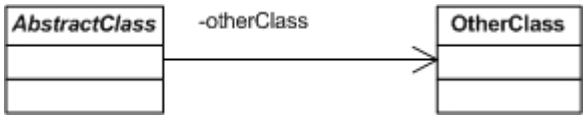
Instance in Abstract Class

Structural

Meaning
An abstract class has a reference to another class.

Belongs to
Bridge.

Structure



Expression
Instance in Abstract Class (className, instanceName, instanceType)
className: the name of the class implementing the Clue;
instanceName: the name of the instance declared in the abstract class;
instanceType: the data type of the declared instance.

Implementation
An abstract class simply maintains a reference to another class, that eventually could be concrete.

Dependent on
N/A.

Sample code

```
abstract class AbstractClass {
        OtherClass otherClass;
}

class OtherClass {
        …
}
```

Meaning
The class has a private instance of itself. Access to this instance can occur only from within the same class.

Belongs to
Singleton

Structure

Expression
Private self instance (className)
className: the class declaring the private instance;
No more information is needed, as the instance belongs to the same class by definition.

Implementation
The class declares a private instance of itself. If this Clue is coupled with a Static self instance Clue, we have a good probability that the class represents a possible instance of the Singleton design pattern.

Sample code

```
public class Singleton {
       private static Singleton instance;

       protected Singleton() {}

       public static Singleton instance() { ... }
}
```

Single Self Instance — Creational

Meaning
The class maintains a single instance of itself. Therefore this instance is unique inside the system.

Belongs to
Singleton

Structure

Expression
Single Self Instance (className)
className: the class declaring the single instance;
No more information is needed, as the instance belongs to the same class by definition.

Implementation
The class maintains a single instance of itself. This is a necessary condition for the class to be an instance of the Singleton design pattern. It is not a sufficient condition, as this instance must also be accessed only from within the class itself, and this property is not granted by this Clue.

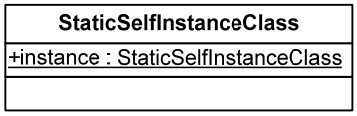
Sample code
```
public class Singleton {
        Singleton instance;
        …
}
```

Static Self Instance — Creational

Meaning
The class has a static instance of itself. Therefore this instance is unique inside the system.

Belongs to
Singleton

Structure



Expression
Static self instance (className)
className: the class declaring the static instance;
No more information is needed, as the instance belongs to the same class by definition.

Implementation
The class declares a static instance of itself. This Clue is often coupled with the Private self instance Clue, discussed next, so that all accesses to this instance can be allowed only by methods belonging to the same class.

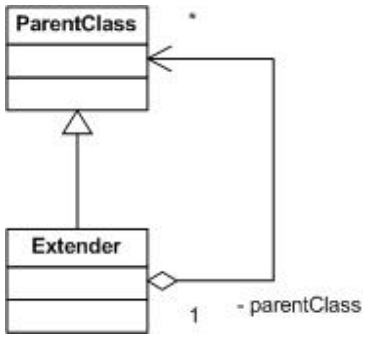
Sample code
```
public class Singleton {
        private static Singleton instance;

        protected Singleton() {}

        public static Singleton instance(){ ... }
}
```

Same Interface Container — Structural

Meaning
A class contains a set or a list of elements that are compatible with the same interface of the declaring class.

Belongs to
Composite (and maybe Decorator).

Structure



Expression
Same Interface Container (className, parentName, elementsList(elementType, elementName))
- className: the name of the class implementing the Clue;
- parentName: the name of the common interface;
- elementsList: the list of elements declared by the class, each one specified by its type and possibly by its name.

Implementation
The class declares some sort of structure (array, list, set, vector…) that will be used to store objects obeying to a common interface. In Java, starting from version 1.5, this objective can be simply achieved with the use of generics, but for older versions each element of the structure should be checked and compared to the others.

Dependent on
N/A.

Sample code

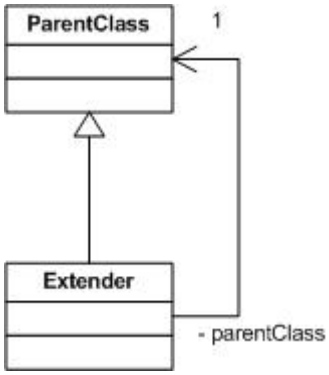
```
public class ParentClass {
        …
}

class SameInterfaceContainerClass {
        List<ParentClass> list;
}
```

Same Interface Instance — Structural

Meaning
A class contains a reference to an object whose type is compatible with the same interface of the declaring class.

Belongs to
Decorator.

Structure



Expression
Same Interface Instance (className, parentName, instanceType, instanceName)
- className: the name of the class implementing the Clue;
- parentName: the name of the common interface;
- instanceType: the specific type of the declared instance;
- instanceName: the name of the declared instance;

Implementation
The class simply declares some an object that is compatible to the same interface.

Dependent on
N/A.

Sample code

```
public class ParentClass {
        …
}

class SameInterfaceContainerClass {
    private ParentClass parentClass;
    }
```

Return Information

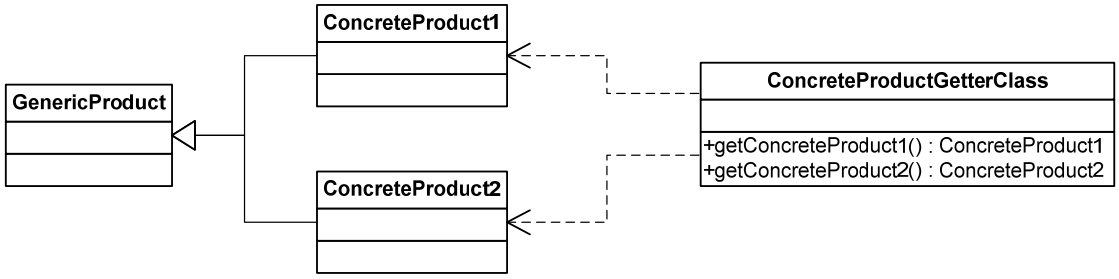
The Return Information category collects those Clues that give indications about various method return mechanisms. In particular, this group propose information about the number of return points that can be found inside a method, about the types of the returned objects and about possible empty (or default) implementations of methods that should return some sort of value. This category collects six Clues.

Concrete Product Getter	Structural

Meaning
A class declares one or more methods that return objects belonging to some other classes.

Belongs to
Abstract Factory, Prototype

Structure



Expression
Concrete product getter (className, methodName, returnType)
className: the name of the class implementing the Concrete product getter;
methodName: the name of the getter method;
returnType: the getter method's return type.

Implementation
A Concrete product getter generally limits its implementation to a single statement, corresponding to a certain object return, whose type corresponds to the getter return type. Therefore, a single return point can be found inside a Concrete product getter. We should consider Concrete product getters also those methods which don't limit their implementation to a simple object return, but that eventually execute some other kind of operations before returning.

Sample code
```
public class MazeFactory {
        public MazeFactory() {}
        // concrete product getters
        public final Maze makeMaze() { return new Maze(); }
        public final Wall makeWall() { return new Wall(); }
        public final Room makeRoom(int n) { return new Room(n); }
        public final Door makeDoor(Room r1, Room r2) {
                return new Door(r1, r2);
        }
}
```