1 GoF

Mediator The Mediator Design Pattern is used to encapsulate how a set of objects interact with each other. The different objects do not have to reference each other explicit.

Memento The Memento Design Pattern is used to store an internal state. This is often used for restoring the internal state of an object.
The object creates a memento (a copy of the state itself) and returns it to the world. The state of the object can be restored using such a memento. You can not change the memento and the inner live is unknown to the outer world.

Command The command pattern is used to encapsulate actions / request inside an object. For example, the design of a framework does not know which action the button should perform. Therefore, the button is created using a command as parameter. As soon as the user clicks on the button, the button executions the `command.Execute()` function.

Command Processor The Command Processor is an extension of the Command Pattern. The Command Processor is more or less just a Stack of Commands.
The Command Processor gets the command (`do_it`). If you want to undo more than one command you just call `undo_it` multiple times on the processor.

Visitor The Visitor Patter is used to "visit" class hierarchy. The visitor can perform action on the visited elements (for example spell checking in a document).
It is easy to implement a new Visitor (Just implement the Interface). However, extending the class hierarchy with a new subclass is very difficult. You have to extend the interface `Visitor` and therefore all implementations of this interface.

(External) Iterator The iterator design pattern is used to access the items in a Data structure (Aggregate in figure **??**). The benefit of iterator is that it hides the implementation details of the data structure.

Internal Iterator The internal Iterator is used to iterate over a data structure. In contrast to the Design Pattern - External Iterator you provide a function, which is called for each element.

```
List<T>.ForEach(Action<T>); // C#
Collection.forEach(); // Java
Stream.forEach(); // Java
Array.prototype.forEach(); // JS
```

Listing 1: Internal Iterator Examples

Class for States The state pattern is used allow an object to alter its internal behavior depending on its state. The object will appear to change its class.

Method for States

Collection for States

Collection for States Collection for States is a way to handle multiple objects in a particular state. For each state you have a list / collection, which contains the objects in this state.
If an event is triggered, then the event is executed on all objects in the list, which should handle this. The distinction between the states is made depending on which list / collection the work item is.

Factory Method The Factory Method is used to create an object, but the subclass decides which class is instantiated. With this pattern you can create concrete creators which can decide which concrete product should be created. The user of the creator only has to use the interface.

Template Method Using the Template Method Pattern you write the skeleton for an algorithm in the base class. The functions which do something with data can be overwritten in a subclass.

Strategy The Strategy Pattern is used to encapsulate an algorithm in an object. With this approach you can change the algorithm at runtime.

Flyweight The Flyweight pattern is used to support large number of objects efficiently. For example, you can store every character in its own object. In a document with 1000 characters you need more than 1000 objects. The Flyweight pattern helps here.
In the Flyweight class is only state indecent stuff stored. Therefore, the Flyweight is shareable. Instead, creating every time a new object with the character "a" you always reference to the same object.

2 Value Patterns

Value Object The Value Object Pattern is used to represent a class as values (immutable). In Java you don't have a Value Type `Year`. Year however, is just a number. With the Value Object Pattern you can simulate the same behavior for the Class `Year` as it was a type `int`. The Identity is **not** important for such a type / value.
In C# just use the `struct` keyword. In Java use the `record` keyword. However, you have to do some more stuff in Java.

Whole Value The Whole Value Pattern answers the following question:

> How can you represent primitive quantities from your problem domain without loss of meaning?
>
> «Express the type of the quantity as a Value Class.» [Henney]

The Whole Value can be immutable, but it is not required to be.

```
public final class Year {
    public Year(int year) { value = year; }
    public int getValue() { return value; }
    private final int value;
}
```

Enumeration Value An enumeration value is just an Enum in Java, C# or Rust.

Immutable Value An Immutable Value can not be changed. That means all internal fields are `const` or `final`.

```
struct Value {
    readonly int _value;
}
```

Simple Factory Intent:

- Provide an interface for creating objects without specifying the concrete implementation.

Motivation:

- You want to create concrete objects without knowing which concrete type you get.
- You are only interested in the interface.

Mutable Companion

> How can you simplify complex construction of an Immutable Value?

A Mutable Companion is a factory object for immutable values. `StringBuilder` is an example for such a Mutable Companion for `Strings`. For the final creation of the object (see Listing 2) you use the Design Pattern - Factory Method.

```
var yearBuilder = new YearCompanion(Year.of(2020));
yearBuilder.next();
var nextYear = yearBuilder.asValue();

public final class YearCompanion {
    private int value;

    public YearCompanion(Year toModify) {
        this.value = toModify.getValue();
    }
    public void next() {
        this.value++;
    }
    public Year asValue() { // factory method
        return Year.of(value);
    }
}
```

Listing 2: Example for Mutable Companion

3 CHECKS Patterns

Exceptional Behavior

> How can exceptional behavior caused by invalid input be handled without throwing errors?

If your function should indicate an error, but should not throw an error / exception then return an *exceptional value* in an error case. An *exceptional value* could be:

- `null` or `undefined` (not recommended)
- Enumeration Value to identify what went wrong

```
export enum CalculationError {
    DivByZero = "div/0",
    NumeratorIsNaN = "NaN(numerator)",
    DivisorIsNaN = "NaN(divisor)"
}

export class Calculator {
    public static divide(numerator: number, divisor
        : number): number | CalculationError {
        if (divisor === 0) {return CalculationError
            .DivByZero; }
        if (isNaN(numerator)) {return
            CalculationError.NumertorIsNaN; }
        if (isNaN(divisor)) {return
            CalculationError.DivisorIsNaN; }

        return numerator / divisor;
    }
}
```

Listing 3: Example for Exceptional Behavior

Meaningless Behavior

> Write methods (without) with minimalistic concern for possible failure.

You will return a meaningless value unless a condition has domain meaning

4 Framework

Framework A framework is a collection of classes, functions, ... that are working together. A framework provides *hook* for *extensions*. In contrast to a library is, that the framework own the control flow and calls **your** components (Inversion of Control).
Examples:

- Java
  - Hibernate
  - Velocity
- C#
  - .NET / Core
  - Entity Framework
- JS
  - Vue.js

Application Framework An Application Framework is a special kind of framework. In a framework you use the framework, and you start the framework. In an Application Framework the "main" method lives inside the application framework.
An example for an Application Framework is Eclipse. Based on the Eclipse you can create many other products (Java IDE, C++ IDE, …).
Examples:

- Java
  - Spring (Boot)
  - Application Server (J2EE)
- C#
  - ASP.NT
- JS
  - Angular

5 Reflection

Reflection Reflection consists of two aspects:

- *Introspection* (observe own state, e.g. query object properties)
- *Intercession* (modify own state, e.g. add another attribute)

> In computer science, reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behavior. [Wikipedia]

```
if (origin instanceof Cloneable) {
    cloned = ((Cloneable)origin).clone();
} else {
    cloned = origin.getClass().
          getDeclaredConstructor().newInstance();
    // ...
    BeanUtils.copyProperties(origin, cloned); //
          get data from getters, fill into setters.
}
```

Meta Level Programming The *Meta Level* provides a self-representaiton

- gives the software knowledge of its own structure
- consists of Meta Objects
- interface for manipulation meta objects is called Metaobject Protocol

The *base level* defines the application logic.

Meta Object If a programming language uses reflections it uses meta objects for:

- classes
- object attributes
- methods
- class relationships
- …

Meta Object Protocol The Metaobject protocol provides a vocabulary to access and manipulate the structure and behavior of a system of objects.

Pro / Con Reflection Benefits:

- Adapting a software system is easy
- support for many kinds of changes

Liabilities:

- Produces non-transparent APIs
  - control flow already hard to understand with polymorphism
- binding at runtime (late binding)
  - limited type safety (Rename class, but class name in string during reflection class is not renamed)
  - lower efficiency, no compiler-optimizaiton

Type Object A Rental System contains books and videos. For each book / video multiple copies may exist. The descriptive information (title, author, actors) is stored in the book / video class (shared among copies). Each media item type has own data and behavior.
Problem:

- We want to keep common behavior and data in only one place
  - object behavior depends on category (video / book)
  - categories should be objects themselves

> How can you categorize objects, eventually dynamically?
>
> Categorize objects by another object instead of a class (Book, Video) Thus, an object can change 'class' at runtime.

Solution:

- crate a category (*type*) object which describes multiple objects
- *objects* forward the calls to the underlying type

```
// Base level
public class Copy {
    protected MediaType type;
    protected int copyid; // invoentory no

    public int getId() { return copyid; }
    // example for delegation
    public String getTypeid() { return type.getId()
          ; }
    public String getTitle() { return type.getTitle
          () ;}
}

// Meta level
public class MediaType {
    protected String title;
    protected String typeid;

    public String getId() { return typeid; }
    public String getTitle() { return title; }
}
```

Listing 4: Example for Type Object

Property List **Problem**:

- attributes should be attachable / detachable after compilation
- objects share attributes / parameters across the class hierarchy

> How do you define properties (also arguments) in a flexible way, so they can be attached and detached at runtime?
>
> Provide objects with a 'property list'. That list allows to associate names with other values or objects.

Solution:

- property list maps attribute names to values
- each name (e.g. a string) defines a slot / property

```
public class Graphics {
    private Properties pl;

    public String get(String prop) {
        return pl.getProperty(prop, "");
    }
    public void set(String prop, Object value) {
        pl.setProperty(prop, value);
    }
}

package java.util;
public class Properties {
    public Set<String> stringPropertyNames() { /**/
          }
    public String getProperty(String prop, String
          defaultValue) { /**/ }
    public String setProperty(String prop, Object
          value) { /**/ }
}
```

Listing 5: Example for Anything

Benefits:

- you can extend the object with properties at runtime while keeping object identity

Liabilities:

- Type safety left to the programmer (see Design Pattern - Bridge Method)
- naming not checked by a compiler (see Design Pattern - Bridge Method)

Anything The Anything Pattern can be used to describe / create arbitrary data structure. The Anything Pattern is something like a recursively structured Property List (Design Pattern - Property List)
An object is a collection of properties (property list) where you access the different values by name / key. An array is a collection of properties (property list) where you access the different values by index.

6 Singleton

Singleton The Singleton pattern ensures that a class has only one (global) instance. Additionally, it provides easy access to this element.
However, you should **NOT** use it in your software. It is basically a global variable. Therefore, not testable and difficult to maintain.
If you want something like singleton, you should use a simple factory which always returns the same instance.

Registry The Singleton Design Pattern is evil and should not be used. An alternative could be the Registry pattern.
You will register multiple singletons in the registry. Using a `lookup(name)` function you will get your specific singleton. The `instance()` returns the default singleton from the registry.

Monostate If you have to use a framework, which provides a singleton you can kill this singleton using a *Monostate*. A monostate is a normal class, **but** all member variables are static members (see Listing 6). **Attention:** In parallel and async execution you have to sync the access to static members.

```
public interface Monostate {
    int getX();
    int getY();
}

public class MonostateImpl implements Monostate {
    public int getX() {
        return Singleton.getInstance().getX();
    }

    public int getY() {
        return Singleton.getInstance().getY();
    }
}
```

Listing 6: Kill a singleton

Service Locator The Service Locator is a singleton, which only returns services or finders to find the concrete service. The Service Locator is similar to a Registry. If you use the Service Locator you have only **one** singleton in your application.

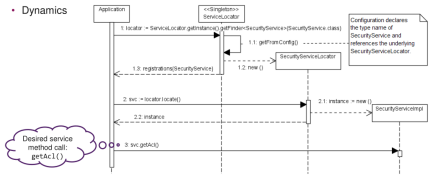

Figure 1: Service Locator Dynamics

Parameterize from Above Often you have functionality which should be shared over multiple or all layers (configuration, security). To solve this problem, without creating a singleton you can use *Parameterize from Above*. In the main method you will create all services. And you provide all services over the constructor.

Dependency Injection A Dependency Injection Container is used to store all services in a registry and injects the required services in the constructor while creating (Parameterize from Above).
You register all services and components in the container. The container will inject a compatible component according to your function signature.