DESIGN PATTERN

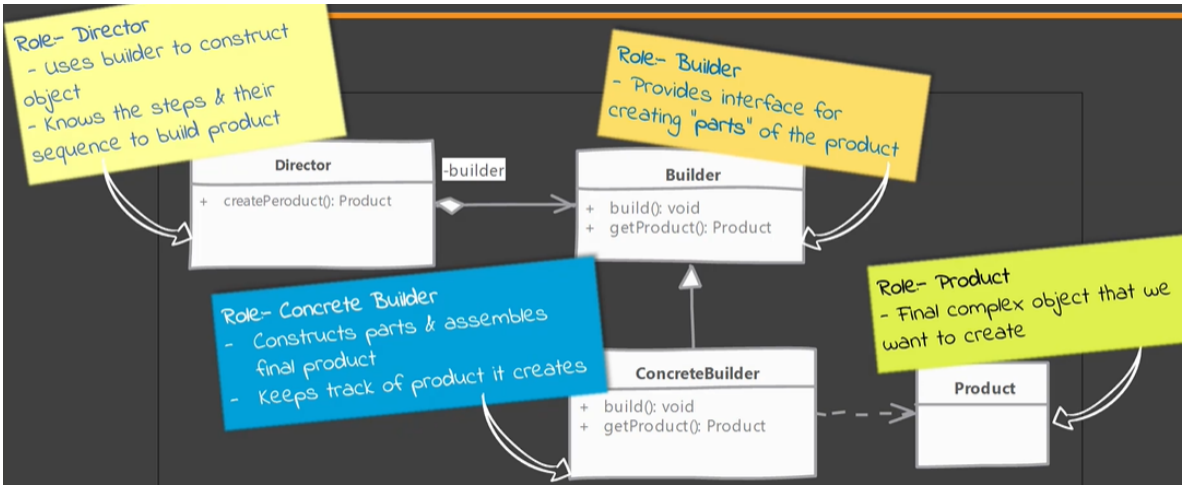
S single responsability Principle (une classe doit avoir une seule responsabilité)
O Open closed (ouvert à l'extension et fermé à la modification)
L liskov substitution (si on remplace une classe fils a une classe parente le comportement ne doit pas changé)
I Interface Segregation (un interface doit avoir juste les methodes communs aux classes qui l'implémentent)
D Dependency Inversion (ne pas dependre des algos, mais juste recevoir les interfaces)
3 types de Patern:

- Creational: creer objet: Builder, Simple Factory, factory, Prototype, Abstract factory, Singleton, Object Pool
- Structural: comment creer ou composer les objets: Adapter, Bridge, Decorator, Composite, Facade, Flyweight, Proxy
- Behavior: comment les objets vont communiquer entre eux: Chain of Responsaility, Command, Interpreter Mediator, Iterator, Memento, Obserrver, State, Strategy, Template method, Visitor, Null Object

Creational:

Builder:

A utiliser quand on doit creer un objet complex avec bcp d'etapes. (le client peut être le directeur)



Ex1: userDTO a name, adress et age en string (c lui qu'on construire) User a firsname, lastname, datenaiss..

```java
//Abstract builder
public interface UserDTOBuilder {
    //methods to build "parts" of product at a time
    UserDTOBuilder withFirstName(String fname) ;

    UserDTOBuilder withLastName(String lname);

    UserDTOBuilder withBirthday(LocalDate date);

    UserDTOBuilder withAddress(Address address);
    //method to "assemble" final product
    UserDTO build();
    //(optional) method to fetch already built object
    UserDTO getUserDTO();
```

son implementattion

```java
public class UserWebDTOBuilder implements UserDTOBuilder {
    private String firstName;private String lastName;
    private String age;private String address;
    private UserWebDTO dto;
    @Override
    public UserDTOBuilder withFirstName(String fname) {
        firstName = fname;
        return this;
    }
    @Override
    public UserDTOBuilder withLastName(String lname) {
        this.lastName = lname;
        return this;
    }
    @Override
    public UserDTOBuilder withBirthday(LocalDate date) {
        Period ageInYears = Period.between(date, LocalDate.now());
        age = Integer.toString(ageInYears.getYears());
        return this;
    }
    @Override
    public UserDTOBuilder withAddress(Address address) {
        this.address = address.getHouseNumber()+", "+address.getStreet()+"\n "
    +address.getCity()+"\n "+address.getState()+" "+address.getZipcode();
        return this;
    }
    @Override
    public UserDTO build() {
        dto = new UserWebDTO(firstName+" "+lastName, address, age);
        return dto;
    }
    @Override
    public UserDTO getUserDTO() {
        return dto;
    }
```

le client (Directeur)

```java
public static void main(String[] args) {
    User user = createUser();
    UserDTOBuilder builder = new UserWebDTOBuilder();
    UserDTO dto = directBuild(builder, user);
    System.out.println(dto);
}
private static UserDTO directBuild(UserDTOBuilder builder, User user) {
    return builder.withFirstName(user.getFirstName())
            .withLastName(user.getLastName())
            .withAddress(user.getAddress())
            .withBirthday(user.getBirthday()).build();
}
```

Ex 2 : meme cas mais UserDto n'a plus de construct avec arg mais des setter private (donc pas accesible en dehors de la classe) avec static Inner class UserDTOBuilder

```java
public class UserDTO {
    3 usages
    private String name;
    3 usages
    private String address;
    3 usages
    private String age;
    public String getName() { return name; }
    public String getAddress() { return address; }
    public String getAge() { return age; }
    1 usage
    private void setName(String name) { this.name = name; }
    1 usage
    private void setAddress(String address) { this.address = address; }
    1 usage
    private void setAge(String age) { this.age = age; }
    @Override
    public String toString() { return "name=" + name + "\nage=" + age + "\naddress=" + address ; }
    //Get builder instance
    public static UserDTOBuilder getBuilder() { return new UserDTOBuilder(); }
    //Builder
    7 usages
    public static class UserDTOBuilder {...}
```

les with qui sont dans le inner class ne changent pas

```
public static class UserDTOBuilder {
    2 usages
    private String firstName;private String lastName;
    2 usages
    private String age;private String address;
    6 usages
    private UserDTO userDTO;
    public UserDTOBuilder withFirstName(String fname) {
        this.firstName = fname;
        return this;
    }
    public UserDTOBuilder withLastName(String lname) {
        this.lastName = lname;
        return this;
    }
    public UserDTOBuilder withBirthday(LocalDate date) {
        age = Integer.toString(Period.between(date, Local
```

suite static inner class

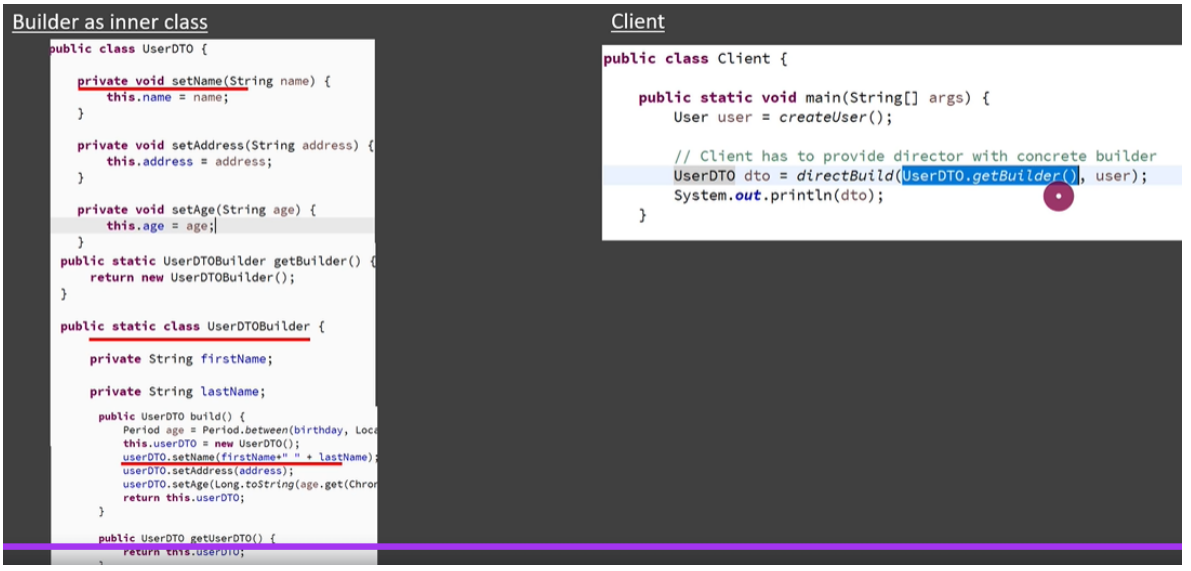
```
public UserDTO build() {
    this.userDTO = new UserDTO();
    userDTO.setName(firstName+" " + lastName);
    userDTO.setAddress(address);
    userDTO.setAge(age);
    return this.userDTO;
}

public UserDTO getUserDTO() { return this.userDTO; }
```

le client c presque la meme chose que pour l'ex 1

```java
public static void main(String[] args) {
    User user = createUser();
    // Client has to provide director with concrete builder
    UserDTO dto = directBuild(UserDTO.getBuilder(), user);
    System.out.println(dto);
```

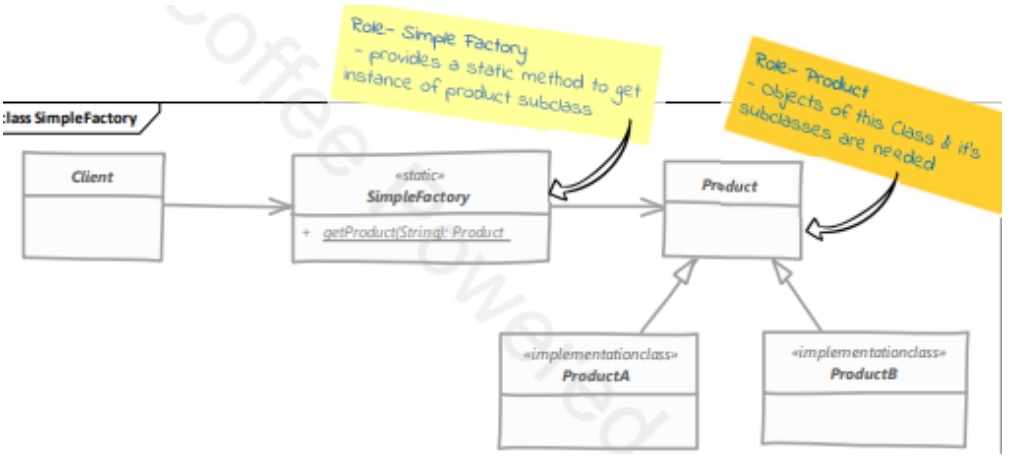
ex1 : résumé

ex2: résumé



Simple Factory:

Certains disent que ce n'est pas vraiment une pattern. C'est une classe qui instancie un objet selon le type (paramètre).

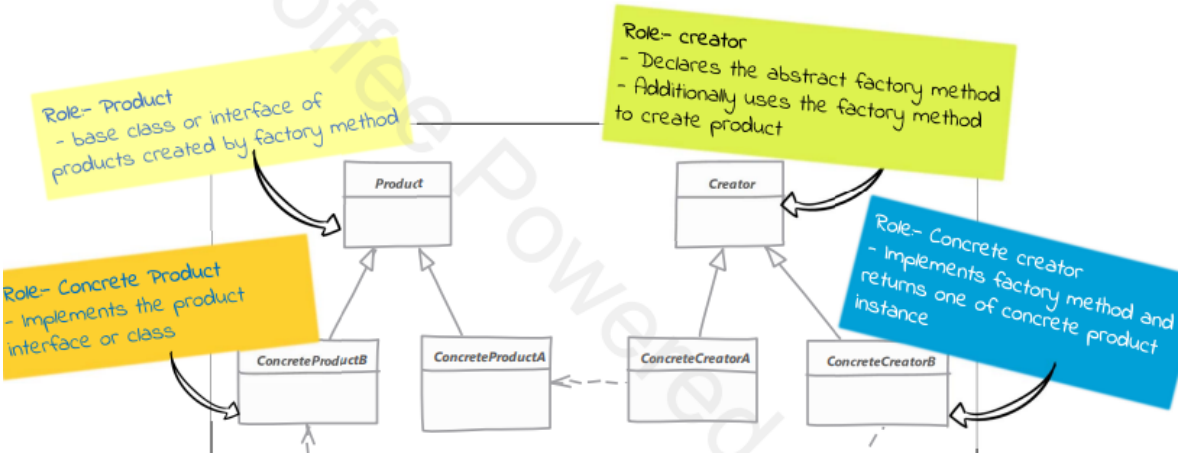



```php
public class PostFactory {
    public static Post createPost(String type) {
        switch(type) {
        case "blog":
                return new BlogPost();
        case "news":
                return new NewsPost();
        case "product":
                return new ProductPost();
        default:
                throw new IllegalArgumentException("Post type is unown");
        }
```

le client: Post est la classe de base et les BlogPost heritent de cette classe

```php
Post post = PostFactory.createPost("news");
System.out.println(post);
```

Factory Method:



ex:

la classe de base

```java
/**
 * This class represents interface for our "product" which is a message
 * Implementations will be specific to content type.
 *
 */
public abstract class Message {

    public abstract String getContent();

    public void addDefaultHeaders() {
        //Adds some default headers
    }

    public void encrypt() {
        //# Has some code to encrypt the content
```

ex de classe concréte

```java
public class JSONMessage extends Message {

    @Override
    public String getContent() {
        return "{\"JSON]\":[]}";
    }
```

le crétaor abstrait

```java
This is our abstract "creator".
The abstract method createMessage() has to be implemented by
its subclasses.
/
blic abstract class MessageCreator {

  public Message getMessage() {
      Message msg = createMessage();
      msg.addDefaultHeaders();
      msg.encrypt();
      return msg;
  }
  public abstract Message createMessage();
```

implementation creator

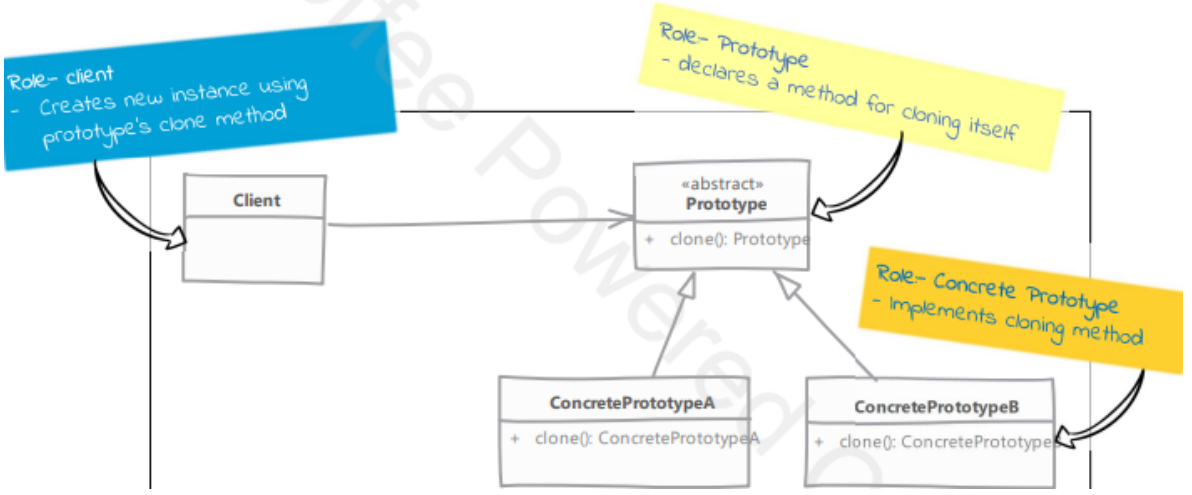
```java
* Provides implementation for creating JSON messages
*/
ublic class JSONMessageCreator extends MessageCreator {

   @Override
   public Message createMessage() {
       return new JSONMessage();
```

client

```java
public static void main(String[] args) {
    printMessage(new JSONMessageCreator());
    printMessage(new TextMessageCreator());
}
public static void printMessage(MessageCreator creator) {
    Message msg = creator.getMessage();
    System.out.println(msg.getContent());
```

Prototype:

Pour creer un objet qui est couteux (parceque par ex il demande a avoir acces à des ressources exterieurs)



le prototype

```java
 * This class represents an abstract prototype & defines the clone method
 */
public abstract class GameUnit implements Cloneable {

    private Point3D position;

    @Override
    public GameUnit clone() throws CloneNotSupportedException {
        GameUnit unit = (GameUnit)super.clone();
        unit.initialize();
        return unit;
    }
    protected void initialize() {
        this.position = Point3D.ZERO;
        reset();
    }
    protected abstract void reset();
    public GameUnit() {
        position = Point3D.ZERO;
    }

    public GameUnit(float x, float y, float z) {
        position = new Point3D(x, y, z);
    }

    public void move(Point3D direction, float distance) {
        Point3D finalMove = direction.normalize();
        finalMove = finalMove.multiply(distance);
        position = position.add(finalMove);
    }

    public Point3D getPosition() {
        return position;
    }
```

la classe clonable

```java
public class Swordsman extends GameUnit {

    private String state = "idle";

    public void attack() {
        this.state = "attacking";
    }

    @Override
    public String toString() {
        return "Swordsman "+state+" @ "+getPosition();
    }

    @Override
    protected void reset() {
        state = "idle";
    }
```

classe non clonable

```java
//Doesn't support cloning
public class General extends GameUnit{
    private String state = "idle";
    public void boostMorale() {
        this.state = "MoralBoost";
    }
    @Override
    public String toString() {
        return "General "+state+" @ "+getPosition();
    }
    @Override
    public GameUnit clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException("General are unique");
    }
    @Override
    protected void reset() {
        throw new UnsupportedOperationException("Reset not supported");
    }
```

client

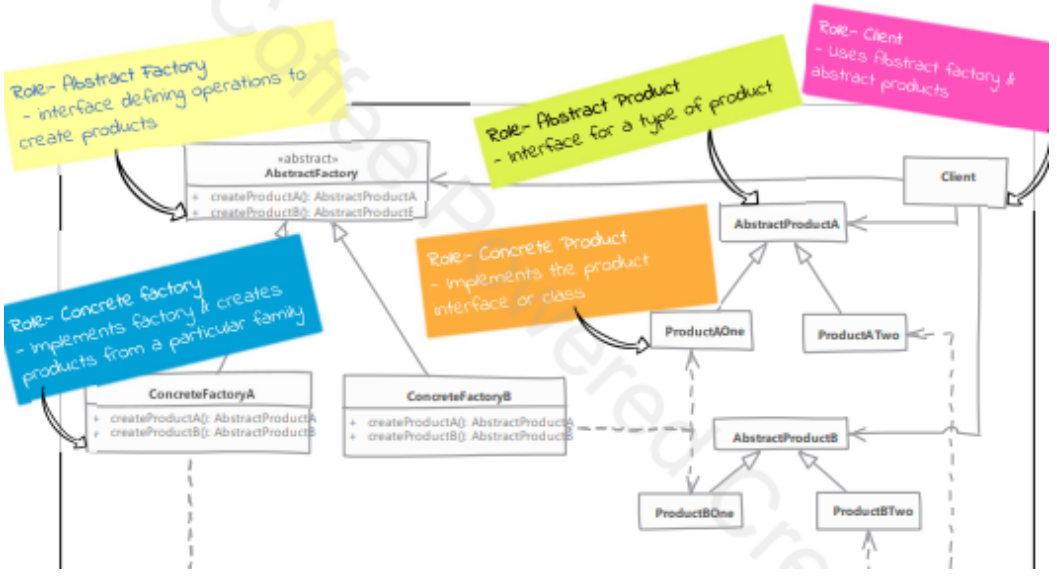
```java
Swordsman s1 = new Swordsman();
s1.move(new Point3D(-10, 0, 0),20);
s1.attack();
System.out.println(s1);

Swordsman s2 = (Swordsman) s1.clone()
```

deep copy: quand les champs sont immutables

shallow copy: c le clonable par defaut

Abstract Factory:

Il permet d'instancier un set d'objets (au moins 2) au lieu d'un autre set. Donc c pour gerer une famille d'objets



Il utilise le pattern factory design pattern

Ex: interface de base

```java
//Represents an abstract product
public interface Instance {
    enum Capacity{micro, small, large}

    void start();

    void attachStorage(Storage storage);

    void stop();
```

la famille aws

```java
//Represents a concrete product in a family "Amazon Web services"
public class Ec2Instance implements Instance {

    public Ec2Instance(Capacity capacity) {
        //Map capacity to ec2 instance types. Use aws API to provision
        System.out.println("Created Ec2Instance");
    }

    @Override
    public void start() {
        System.out.println("Ec2Instance started");
    }

    @Override
    public void attachStorage(Storage storage) {
        System.out.println("Attached "+storage+" to Ec2Instance");
    }

    @Override
    public void stop() {
        System.out.println("Ec2Instance stopped");
    }

    @Override
    public String toString() {
        return "EC2Instance";
    }
```

la famille google

```java
//Represents a concrete product in a family "Google Cloud Platform"
public class GoogleComputeEngineInstance implements Instance {

    public GoogleComputeEngineInstance(Capacity capacity) {
        //Map capacity to GCP compute instance types. Use GCP API to provision
        System.out.println("Created Google Compute Engine instance");
    }

    @Override
    public void start() {
        System.out.println("Compute engine instance started");
    }

    @Override
    public void attachStorage(Storage storage) {
        System.out.println("Attached "+storage+" to Compute engine instance");
    }

    @Override
    public void stop() {
        System.out.println("Compute engine instance stopped");
    }
```

un produit abstrait (storage)

```java
//Represents an abstract product
public interface Storage {
    String getId();
```

produit concret chez aws

```java

//Represents a concrete product in a family "Amazon Web Services"
public class S3Storage implements Storage {

    public S3Storage(int capacityInMib) {
        //Use aws s3 api
        System.out.println("Allocated "+capacityInMib+" on S3");
    }

    @Override
    public String getId() {
        return "S31";
    }

    @Override
    public String toString() {
        return "S3 Storage";
```

produit chez google

```java
//Represents a concrete product in a family "Google Cloud Platform"
public class GoogleCloudStorage implements Storage {

    public GoogleCloudStorage(int capacityInMib) {
        //Use gcp api
        System.out.println("Allocated "+capacityInMib+" on Google Cloud Storage");
    }

    @Override
    public String getId() {
        return "gcpcs1";
    }

    @Override
    public String toString() {
        return "Google cloud storage";
    }
```

abstract factory

```java
//Abstract factory with methods defined for each object type.
public interface ResourceFactory {
    Instance createInstance(Instance.Capacity capacity);
    Storage createStorage(int capMib);
```

factory aws

```java
//Factory implementation for Google cloud platform resources
public class AwsResourceFactory implements ResourceFactory {

    @Override
    public Instance createInstance(Capacity capacity) {
        return new Ec2Instance(capacity);
    }

    @Override
    public Storage createStorage(int capMib) {
        return new S3Storage(capMib);
```

factory google

```java
//Factory implementation for Google cloud platform resources
public class GoogleResourceFactory implements ResourceFactory {
    @Override
    public Instance createInstance(Capacity capacity) {
        return new GoogleComputeEngineInstance(capacity);
    }
    @Override
    public Storage createStorage(int capMib) {
        return new GoogleCloudStorage(capMib);
```

le client

```java
private ResourceFactory factory;
public Client(ResourceFactory factory) {
    this.factory = factory;
}
public Instance createServer(Instance.Capacity cap, int storageMib) {
    Instance instance = factory.createInstance(cap);
    Storage storage = factory.createStorage(storageMib);
    instance.attachStorage(storage);
    return instance;
}
public static void main(String[] args) {
    Client aws = new Client(new AwsResourceFactory());
    Instance i1 = aws.createServer(Capacity.micro, 20480);
    i1.start();
    i1.stop();

    Client google = new Client(new GoogleResourceFactory());
    Instance i2 = google.createServer(Capacity.micro, 20480);
    i2.start();
    i2.stop();
```

Singleton: (pour les fic de config ou params qui ne changent pas)

il s'assure qu'on a qu'une seule instance d'1 objet. Il peut être de type eager ou lazy

type eager:

```java
public class EagerRegistry {
    private EagerRegistry() {}
    private static final EagerRegistry INSTANCE = new EagerRegistry();
    public static EagerRegistry getInstance() {
        return INSTANCE;
```

use case

```java
EagerRegistry registry = EagerRegistry.getInstance();
EagerRegistry registry2 = EagerRegistry.getInstance();
System.out.println(registry == registry2);
```

lazy type: volatile pour les threads n'utilisent pas le cache mais tjs l'obj courant

```java
/**
 * This class demonstrates singleton pattern using Double Checked Locking or "classic" singleton.
 * This is also a lazy initializing singleton.
 * Although this implementation solves the multi-threading issue with lazy initialization using volatile
 * and double check locking, the volatile keyword is guaranteed to work only after JVMs starting with
 * version 1.5 and later.
 */
public class LazyRegistryWithDCL {
    private LazyRegistryWithDCL() {}
    //volatile pour ne pas utiliser la cache du main mais
    //la derniere valeur de INSTANCE
    private static volatile LazyRegistryWithDCL INSTANCE;
    public static LazyRegistryWithDCL getInstance() {
        if(INSTANCE == null) {
            synchronized (LazyRegistryWithDCL.class) {
                if(INSTANCE == null)
                INSTANCE = new LazyRegistryWithDCL();
            }
        }
        return INSTANCE;
    }
```

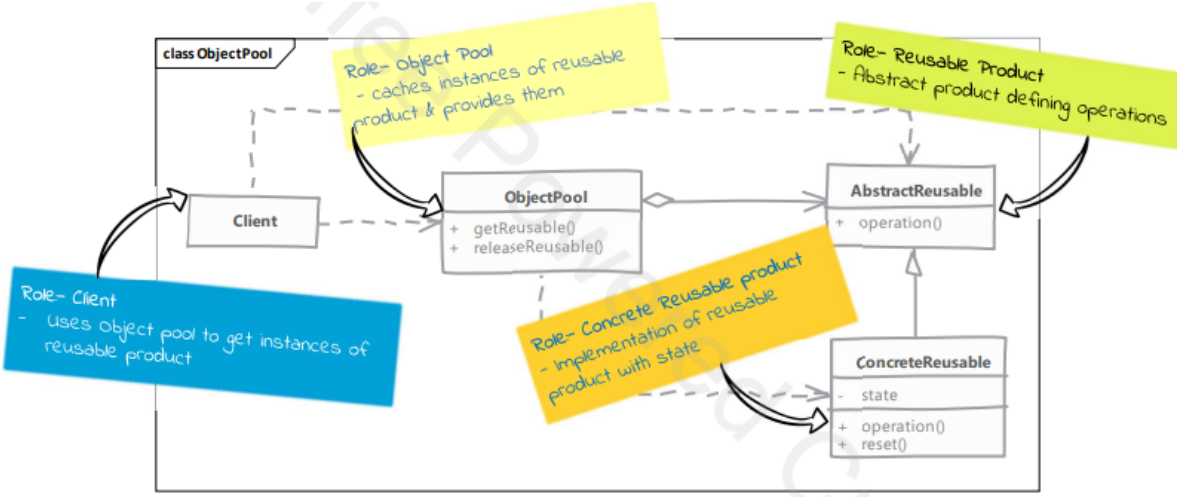
autre moyen de faire du lazy avec un inner private class (le top)

```java
 * Singleton pattern using lazy initialization holder class.
 * This ensures that, we have a lazy initialization
 * without worrying about synchronization.
 */
public class LazyRegistryIODH {
    private LazyRegistryIODH() {
        System.out.println("in LazyRegistryIODH singleton");
    }
    private static class RegistryHolder {
        static LazyRegistryIODH INSTANCE = new LazyRegistryIODH();
    }
    public static LazyRegistryIODH getInstance() {
        return RegistryHolder.INSTANCE;
    }
```

type enum; bon pour la serialization si nécessaire

```java
 * (Ref. Google I/O 2k8 Joshua Bloch)
 * Since Java 1.5 using enum we can create a singleton.
 * It handles serialization using java's in-built
 * mechanism and still ensure single instance
 */
public enum RegistryEnum {
    INSTANCE;
    public void someMethod() {
```

Object Pool: si la création d'un objet est couteux et qu'on a besoin de bcp dans un laps de temps court on peut utiliser ce pattern



ex:

```java
public interface Poolable {
    void reset();
```

```java
//Represents our abstract reusable
public interface Image extends Poolable{

    void draw();

    Point2D getLocation();

    void setLocation(Point2D location);
```

```java
//concrete reusable
public class Bitmap implements Image {
    private Point2D location;
    private String name;
    public Bitmap(String name) {
        this.name = name;
    }
    @Override
    public void draw() {
        System.out.println("Drawing "+name+" @ "+location);
    }
    @Override
    public Point2D getLocation() {
        return location;
    }
    @Override
    public void setLocation(Point2D location) {
        this.location = location;
    }
    @Override
    public void reset() {
        location = null;
        System.out.println("the location is reset");
    }
```

```java
public class ObjectPool<T extends Poolable> {
    private BlockingQueue<T> availablePool;
    public ObjectPool(Supplier<T> creator, int count) {
        availablePool = new LinkedBlockingQueue<>();
        for (int i = 0; i < count; i++) {
            availablePool.offer(creator.get());
        }
    }
    public T get() {
        try {
            return availablePool.take();
        } catch (InterruptedException e) {
            System.out.println("take() was interrupted");
        }
        return null;
    }
    public void release(T obj) {
        obj.reset();
        try {
            availablePool.put(obj);
        } catch (InterruptedException e) {
            System.out.println("put() was interrupted");
        }
    }
```

```java
private static final ObjectPool<Bitmap> bitmapPool = new ObjectPool<>(()-> new Bitmap("Logo.bmp"), 5);
public static void main(String[] args) {
    Bitmap b1 = bitmapPool.get();
    b1.setLocation(new Point2D(10, 10));
    Bitmap b2 = bitmapPool.get();
    b2.setLocation(new Point2D(-10, 0));

    b1.draw();
    b2.draw();
    bitmapPool.release(b1);
    bitmapPool.release(b2);
```

Ex: ThreadPoolExecutor, Datasource apps...

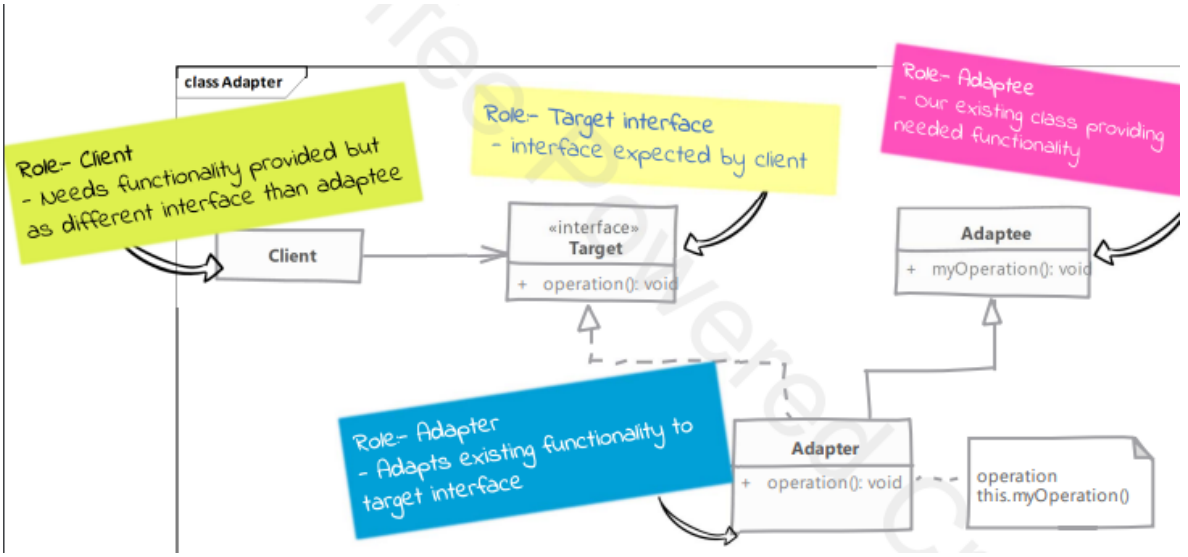
Structural:

il permet de definir comment les objets et/ou classes sont arrangés

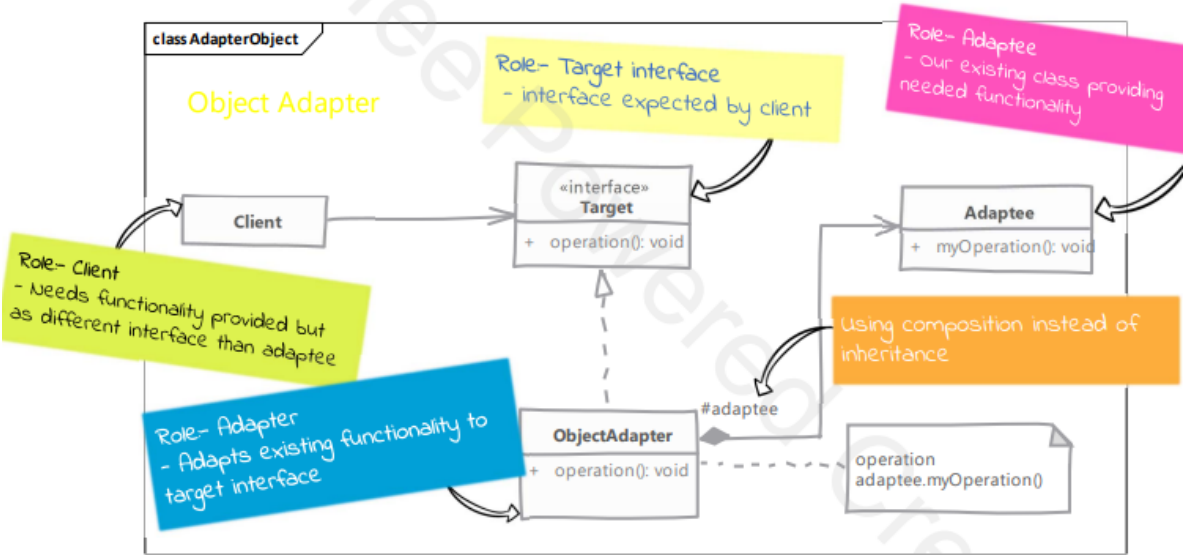
Adapter:

Il ya deux types: class et object

class



object (c le meilleur)



Dans cet ex on adapte l'employee au customer

```java
/**
 * Target interface required by new client code
 */
public interface Customer {
    String getName();
    String getDesignation();
    String getAddress();
```

la classe (methode) à adapter

```java
/*
 * Client code which requires Customer interface.
 */
public class BusinessCardDesigner {

    public String designCard(Customer customer) {
        String card = "";
        card += customer.getName();
        card += "\n" + customer.getDesignation();
        card += "\n" + customer.getAddress();
        return card;
```

la classe

```java
/**
 * An existing class used in our system
 */
public class Employee {

    private String fullName;

    private String jobTitle;

    private String officeLocation;

    public String getFullName() {
        return fullName;
```

l'adapter

```java
/*
 * A class adapter, works as Two-way adapter
 */
public class EmployeeClassAdapter extends Employee implements Customer{
    @Override
    public String getName() {
        return this.getFullName();
    }
    @Override
    public String getDesignation() {
        return this.getJobTitle();
    }
    @Override
    public String getAddress() {
        return this.getOfficeLocation();
    }
```

le client

```java
/** Using Class/Two-way adapter **/
EmployeeClassAdapter adapter = new EmployeeClassAdapter();
populateEmployeeData(adapter);
BusinessCardDesigner designer = new BusinessCardDesigner();
String card = designer.designCard(adapter);
System.out.println(card);
```

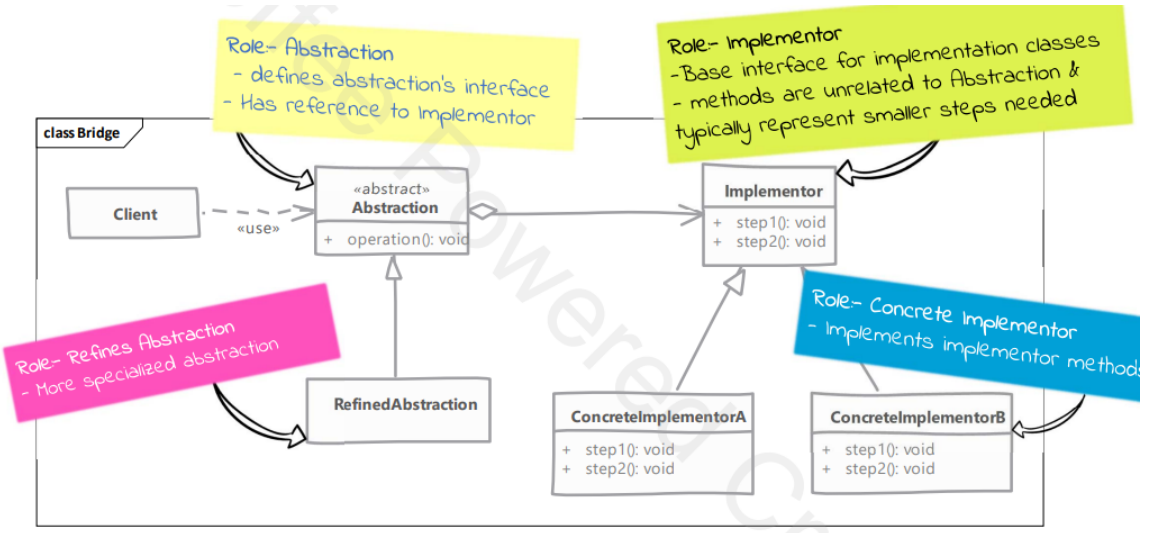
Pour l'object adapter (c le mieux comme dit plus haut): on va juste implements customer

```java
 * An object adapter. Using composition to translate interface
 */
public class EmployeeObjectAdapter implements Customer{
    private Employee adaptee;

    public EmployeeObjectAdapter(Employee adaptee) {
        this.adaptee = adaptee;
    }
    @Override
    public String getName() {
        return adaptee.getFullName();
    }
    @Override
    public String getDesignation() {
        return adaptee.getJobTitle();
    }
    @Override
    public String getAddress() {
        return adaptee.getOfficeLocation();
```

le client (le populate sert juste a faire des set)

```java
/** Using Object Adapter **/
Employee employee = new Employee();
populateEmployeeData(employee);
EmployeeObjectAdapter objectAdapter = new EmployeeObjectAdapter(employee);
card = designer.designCard(objectAdapter);
System.out.println(card);
```

Bridge:

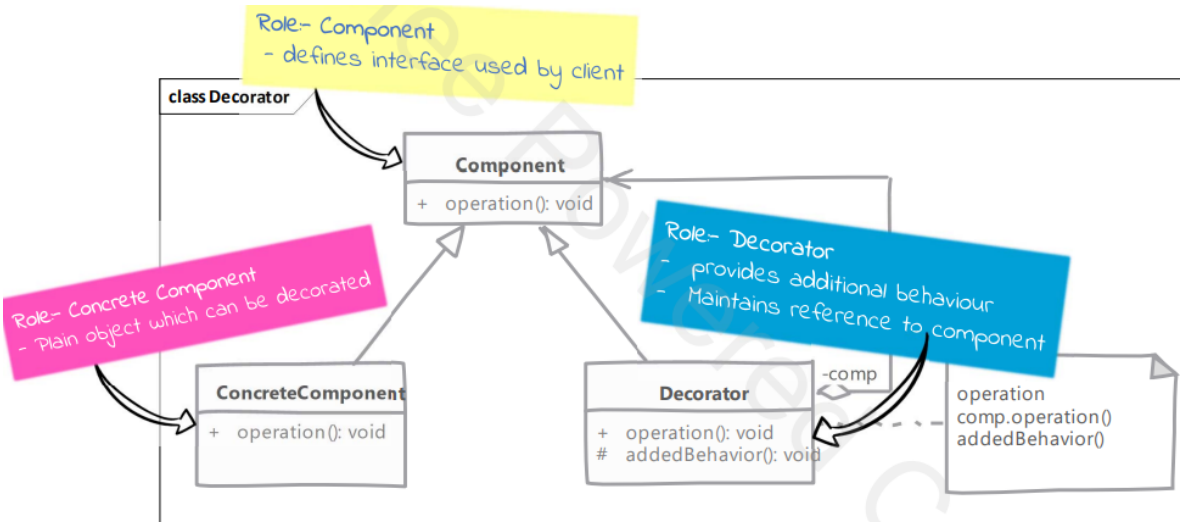



```java
//This is the abstraction.
//It represents a First in First Out collection
public interface FifoCollection<T> {

    //Adds element to collection
    void offer(T element);

    //Removes & returns first element from collection
    T poll();
```

```java
//This is the implementor.
//Note that this is also an interface
//As implementor is defining its own hierarchy which not related
//at all to the abstraction hierarchy.
public interface LinkedList<T> {

    void addFirst(T element);

    T removeFirst();

    void addLast(T element);

    T removeLast();

    int getSize();
```

```java
public static void main(String[] args) {
    FifoCollection<Integer> collection = new Queue<>(new SinglyLinkedList<>());
    collection.offer(10);
    collection.offer(40);
    collection.offer(99);
    System.out.println(collection.poll());
    System.out.println(collection.poll());
    System.out.println(collection.poll());
    System.out.println(collection.poll());
```

Decorator: Il permet d'ajouter des comportements a un objet aprés sa création (runtime)



ex: c le getContent qu'on affiche en html ou base 64 ...

```java
//Base interface or component
public interface Message {

    String getContent();
```

```java
//Concrete component. Object to be decorated
public class TextMessage implements Message {
    private String msg;
    public TextMessage(String msg) {
        this.msg = msg;
    }
    @Override
    public String getContent() {
        return msg;
    }
```

Base 64 decorator

```java
public class Base64EncodedMessage implements Message {
    private Message msg;
    public Base64EncodedMessage(Message msg) {
        this.msg = msg;
    }
    @Override
    public String getContent() {
        //Be wary of charset!! This is platform dependent..
        return Base64.getEncoder().encodeToString(msg.getContent().getBytes());
    }
```

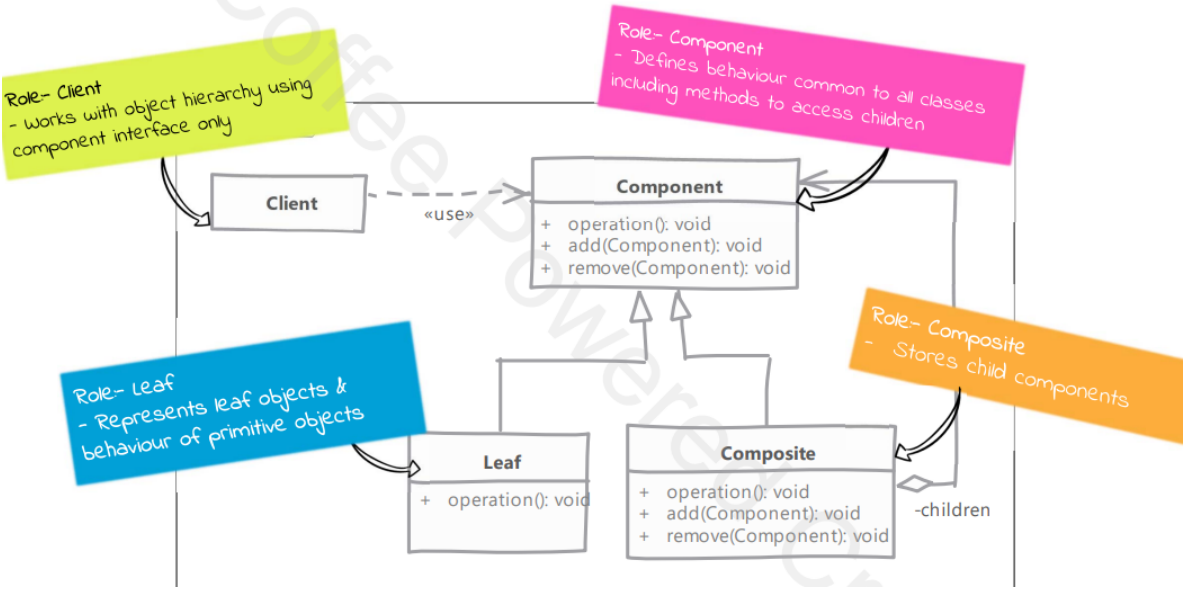
Html Decorator

```java
//Decorator. Implements component interface
public class HtmlEncodedMessage implements Message {

    private Message msg;
    public HtmlEncodedMessage(Message msg) {
        this.msg = msg;
    }
    @Override
    public String getContent() {
        return StringEscapeUtils.escapeHtml4(msg.getContent());
    }
```

client

```java
public static void main(String[] args) {
    Message m = new TextMessage("The <FORCE> is strong with this one!");
    System.out.println(m.getContent());

    Message decorator = new HtmlEncodedMessage(m);
    System.out.println(decorator.getContent());
    decorator = new Base64EncodedMessage(m);
    System.out.println(decorator.getContent());
```

Composite: Pour creer un objet sans se soucier de savoir si c une feuille (leaf) ou tronc(composite)



ex:

```java
//The component base class for composite pattern
//defines operations applicable both leaf & composite
public abstract class File {

    private String name;

    public File(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public abstract void ls();

    public abstract void addFile(File file);

    public abstract File[] getFiles();

    public abstract boolean removeFile(File file);
```

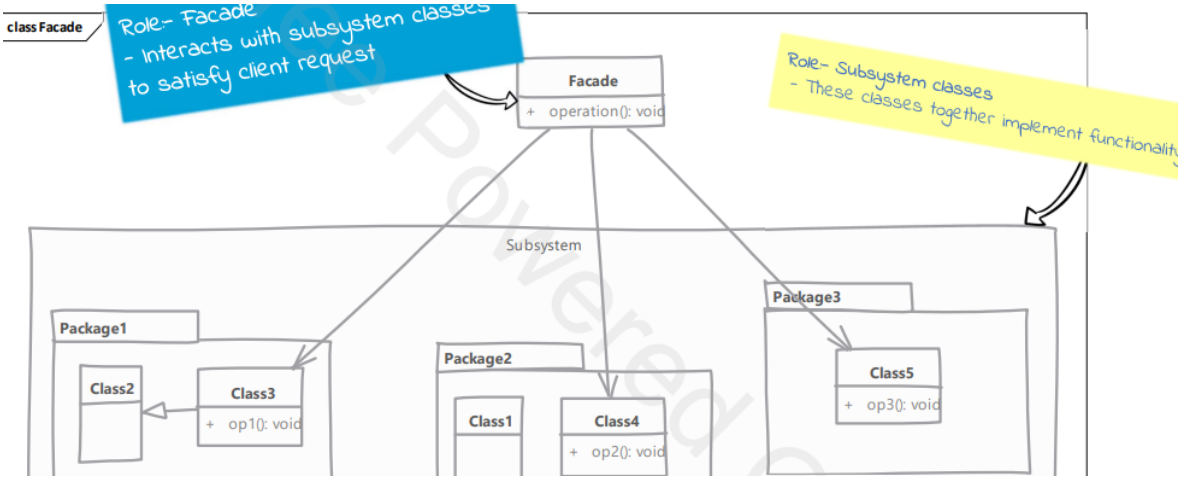
```java
//Leaf node in composite pattern
public class BinaryFile extends File {
    private long size;
    public BinaryFile(String name, long size) {
        super(name);
        this.size = size;
    }
    @Override
    public void ls() {
        System.out.println(getName()+"\t"+size);
    }

    @Override
    public void addFile(File file) {
        throw new UnsupportedOperationException(
                "Leaf node doesn't support add operation");
    }

    @Override
    public File[] getFiles() {
        throw new UnsupportedOperationException(
                "Leaf node doesn't support get operation");
    }

    @Override
    public boolean removeFile(File file) {
        throw new UnsupportedOperationException(
                "Leaf node doesn't support remove operation");
    }
```

```java
//Composite in the composite pattern
public class Directory extends File {
    private List<File> children = new ArrayList<>();
    public Directory(String name) {
        super(name);
    }
    @Override
    public void ls() {
        System.out.println(getName());
        children.forEach(File::ls);
    }
    @Override
    public void addFile(File file) {
        children.add(file);
    }
    @Override
    public File[] getFiles() {
        return children.toArray(new File[children.size()]);
    }
    @Override
    public boolean removeFile(File file) {
        return children.remove(file);
    }
```

le client

```java
public static void main(String[] args) {
    File root1 = createTreeOne();
    root1.ls();
    File leaf = createTreeTwo();
    leaf.ls();
}

//Client builds tree using leaf and composites
private static File createTreeOne() {
    File file1 = new BinaryFile("File1", 1000);
    Directory dir1 = new Directory("dir1");
    dir1.addFile(file1);
    File file2 = new BinaryFile("file2", 2000);
    File file3 = new BinaryFile("file3", 150);
    Directory dir2 = new Directory("dir2");
    dir2.addFile(file2);
    dir2.addFile(file3);
    dir2.addFile(dir1);
    return dir2;
}

private static File createTreeTwo() {
    return new BinaryFile("Another file", 200);
}
```

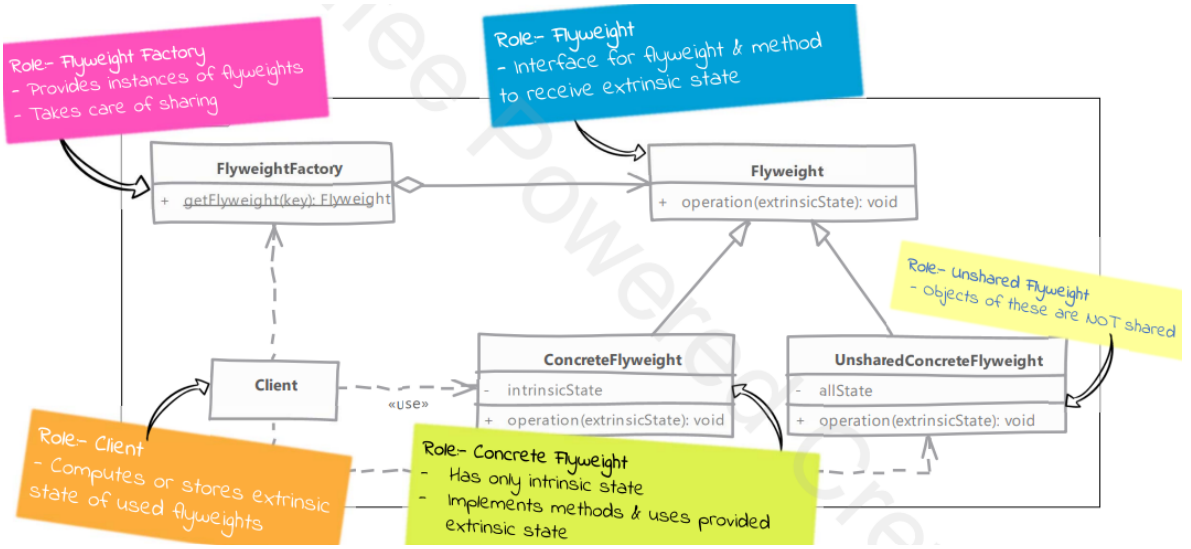
Facade: Pour que le client ne soit pas trés lié aux interfaces (méthodes) qu'il utilise.



deplacer la partie commenter dans une classe différente pour ne pas lier le client et l'implémentation

```java
public static void main(String[] args) {
    Order order = new Order("101", 99.99);
    EmailFacade facade = new EmailFacade();
    boolean result = facade.sendOrder(order);

    System.out.println("Order Email "+ (result?"sent!":"NOT sent..."));

}

/*private static boolean sendOrderEmailWithoutFacade(Order order) {
    Template template = TemplateFactory.createTemplateFor(TemplateType.Email);
    Stationary stationary = StationaryFactory.createStationary();
    Email email = Email.getBuilder()
                .withTemplate(template)
                .withStationary(stationary)
```

Flyweight: pour optimiser les ressources mémoires



ex

```java
//Interface implemented by Flyweights
public interface ErrorMessage {
    //Get error message
    String getText(String code);
```

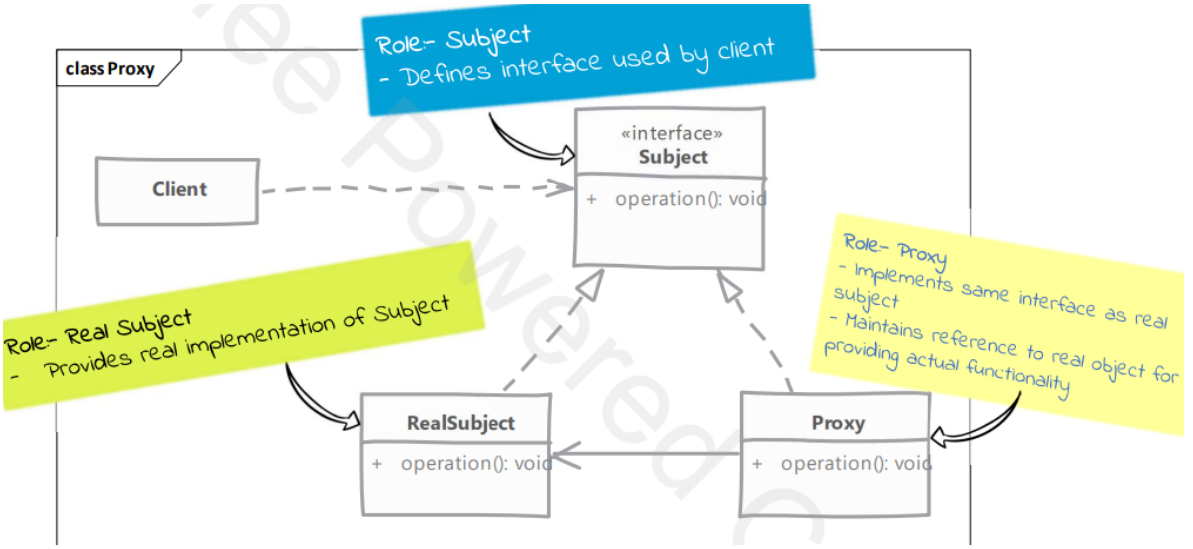
```java
//A concrete Flyweight. Instance is shared
public class SystemErrorMessage implements ErrorMessage {
    //some error msg $errorCode
    private String messageTemplate;
    //http://somedomain.com/help?error
    private String helpurlBase;
    public SystemErrorMessage(String messageTemplate, String helpurlBase) {
        this.messageTemplate = messageTemplate;
        this.helpurlBase = helpurlBase;
    }
    @Override
    public String getText(String code) {
        return messageTemplate.replace("$errorCode", code)+helpurlBase;
    }
```

```java
//Flyweight factory. Returns shared flyweight based on key
public class ErrorMessageFactory {

    //This serves as key for getting flyweight instance
    public enum ErrorType {GenericSystemError, PageNotFoundError, ServerError}

    private static final ErrorMessageFactory FACTORY = new ErrorMessageFactory();

    public static ErrorMessageFactory getInstance() {
        return FACTORY;
    }
    private Map<ErrorType, SystemErrorMessage> errorMessages = new HashMap<>();
    private ErrorMessageFactory() {
        errorMessages.put(ErrorType.GenericSystemError, new SystemErrorMessage(
                "A generic error of type $errorCode. Please refer to:\n",
                "http://google.com/q="));
        errorMessages.put(ErrorType.PageNotFoundError, new SystemErrorMessage(
                "Page not found. An error of type $errorCode. Please refer to:\n",
                "http://google.com/q="));
    }
    public SystemErrorMessage getError(ErrorType type) {
        return errorMessages.get(type);
    }

    public UserBannedErrorMessage getUserBannedMessage(String caseId) {
        return new UserBannedErrorMessage(caseId);
    }
```

client

```java
public static void main(String[] args) {
    SystemErrorMessage msg1 = ErrorMessageFactory.getInstance().getError(ErrorType.GenericSystemError);
    System.out.println(msg1.getText("4056"));
    UserBannedErrorMessage msg2 = ErrorMessageFactory.getInstance().getUserBannedMessage("1202");
    System.out.println(msg2.getText(null));
}
```

Proxy: Pour des besoins de performances (on ne crée l'objet reel que si nécessaire- lazy loading) ou de protection



Ex of static proxy:

```java
//Interface implemented by proxy and concrete objects
public interface Image {
    void setLocation(Point2D point2d);
    Point2D getLocation();
    void render();
```

obj reel

```java
//Our concrete class providing actual functionality
public class BitmapImage implements Image {
    private Point2D location;
    private String name;

    public BitmapImage(String filename) {
        //Loads image from file on disk
        System.out.println("Loaded from disk:"+filename);
        name = filename;
    }
    @Override
    public void setLocation(Point2D point2d) {
        location = point2d;
    }
    @Override
    public Point2D getLocation() {
        return location;
    }
    @Override
    public void render() {
        //renders to screen
        System.out.println("Rendered "+this.name);
    }
}
```

le proxy

```
//Proxy class.
public class ImageProxy implements Image {
    private String name;private BitmapImage imageReel;
    private Point2D location;
    public ImageProxy(String name) {
        this.name = name;
    }
    @Override
    public void setLocation(Point2D point2d) {
        if(imageReel != null) {
            imageReel.setLocation(point2d);
        }else {
            location = point2d;
        }
    }
    @Override
    public Point2D getLocation() {
        if(imageReel != null) {
            return imageReel.getLocation();
        }
        return location;
    }
    @Override
    public void render() {
        if(imageReel == null) {
            imageReel = new BitmapImage(name);
            if(location != null) {
                imageReel.setLocation(location);
            }
        }
        imageReel.render();
    }
```

factory

```
//Factory to get image objects.
public class ImageFactory {
    //We'll provide proxy to caller instead of real object
    public static Image getImage(String name) {
        return new ImageProxy(name);
    }
```

le client

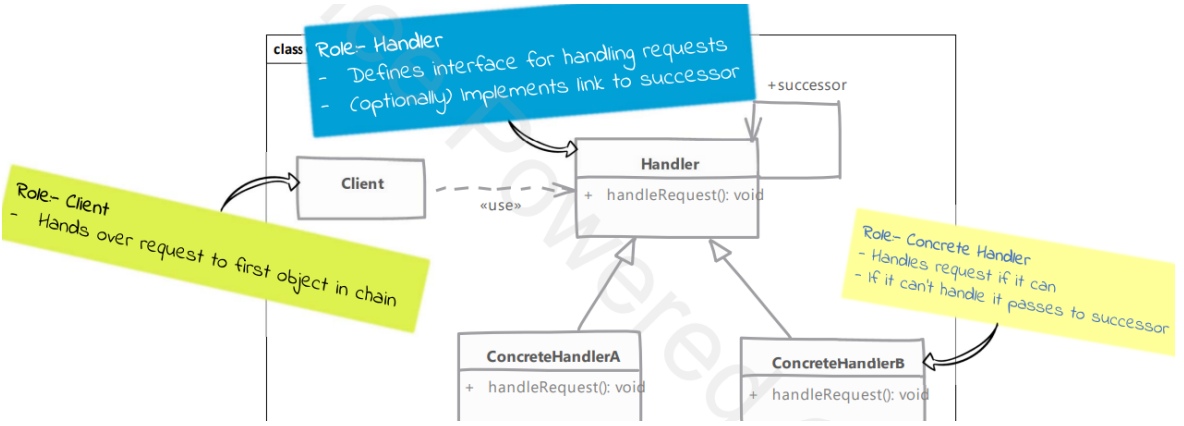
```
lic static void main(String[] args) {
Image img = ImageFactory.getImage("A1.bmp");

img.setLocation(new Point2D(10, 10));
System.out.println("Image location:"+img.getLocation());
System.out.println("Rendering image now...");
img.render();
```

for dynamic exple see code

Behavior:

il permet de definir comment les objets et/ou classes sont arrangés

Chain of Responsability: c'est utilisé par les servmets par ex (lun aprés l'autre jusqu'a passé par ts)



obj req avec gett sett et builder

```
//Represents a request in our chain of responsibility
public class LeaveApplication {
    public enum Type {Sick, PTO, LOP};
    public enum Status {Pending, Approved, Rejecetd };
    private Type type;
    private LocalDate from;
    private LocalDate to;
    private String processedBy;
    private Status status;
    public LeaveApplication(Type type, LocalDate from, LocalDate to) {
        this.type = type;
        this.from = from;
        this.to = to;
        this.status = Status.Pending;
    }

    public Type getType() {
        return type;
    }

    public LocalDate getFrom() {
        return from;
    }
```

l'interface

```java
//This represents a handler in chain of responsibility
public interface LeaveApprover {
    void processLeaveApplication(LeaveApplication application);
    String getApproverRole();
}
```

abstract

```java
//Abstract handler
public abstract class Employee implements LeaveApprover {
    private String role;
    private LeaveApprover successor;
    public Employee(String role, LeaveApprover successor) {
        this.role = role;
        this.successor = successor;
    }
    @Override
    public void processLeaveApplication(LeaveApplication application) {
        if(!processRequest(application) && successor!= null) {
            successor.processLeaveApplication(application);
        }
    }
    protected abstract boolean processRequest(LeaveApplication application);
    @Override
    public String getApproverRole() {
        return role;
    }
```

concret 1

```java
//A concrete handler
public class ProjectLead extends Employee {
    public ProjectLead(LeaveApprover successor) {
        super("Project Lead", successor);
    }
    @Override
    protected boolean processRequest(LeaveApplication application) {
        //type sick leave & duration is less or eq to 2 days
        if(application.getType() == LeaveApplication.Type.Sick) {
            if(application.getNoOfDays() <= 2) {
                application.approve(getApproverRole());
                return true;
            }
        }
        return false;
    }
```

concret 2

```java
//A concrete handler
public class Manager extends Employee {

    public Manager(LeaveApprover nextApprover) {
        super("Manager", nextApprover);
    }

    @Override
    protected boolean processRequest(LeaveApplication application) {
        switch (application.getType()) {
        case Sick:
            application.approve(getApproverRole());
            return true;
        case PTO:
            if(application.getNoOfDays() <= 5) {
                application.approve(getApproverRole());
                return true;
            }
        }
        return false;
    }
```

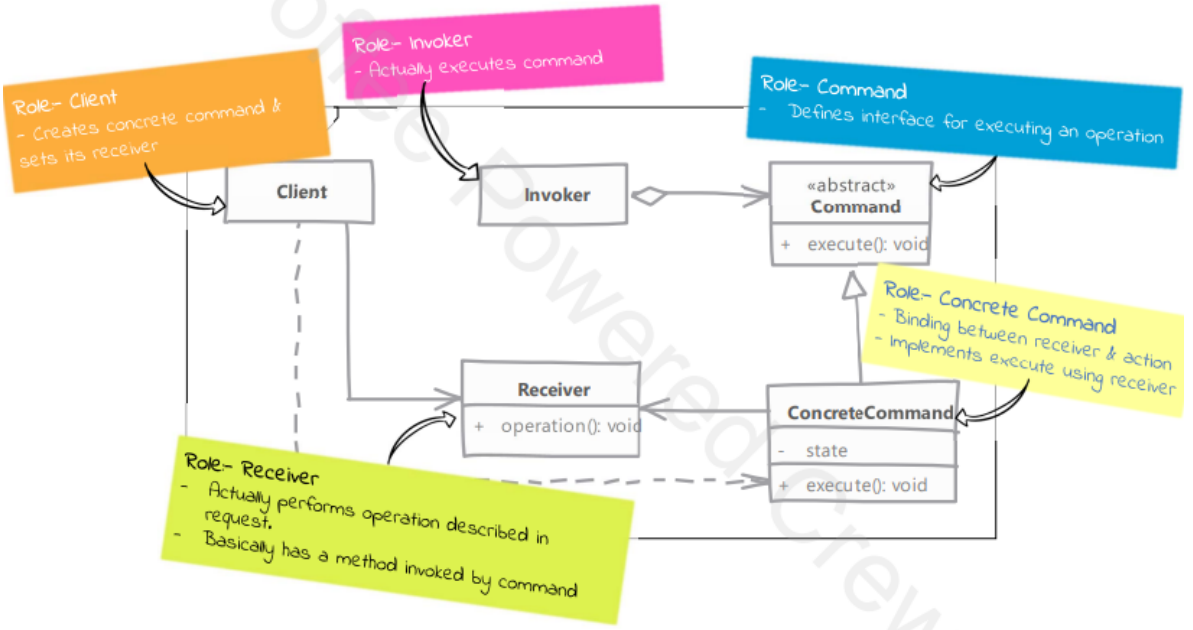
concret 3

```java
//A concrete handler
public class Director extends Employee {
    public Director(LeaveApprover nextApprover) {
        super("Director", nextApprover);
    }
    @Override
    protected boolean processRequest(LeaveApplication application) {
        if(application.getType() == Type.PTO) {
            application.approve(getApproverRole());
            return true;
        }
        return false;
    }
```

client

```java
public static void main(String[] args) {
   LeaveApplication application = LeaveApplication.getBuilder().withType(Type.Sick)
           .from(LocalDate.now())
           .to(LocalDate.of(2023, 2,15)).build();
   System.out.println(application);
   System.out.println("****************************");
   LeaveApprover approver = createChain();
   approver.processLeaveApplication(application);
   System.out.println(application);
}
private static LeaveApprover createChain() {
    Director director = new Director(null);
    Manager manager = new Manager(director);
    ProjectLead lead = new ProjectLead(manager);
    return lead;
}
```

Command:




```java
//This class is the receiver.
public class EWSService {
    //Add a new member to mailing list
    public void addMember(String contact, String contactGroup) {
        //contact exchange
        System.out.println("Added "+contact +" to "+contactGroup);
    }
    //Remove member from mailing list
    public void removeMember(String contact, String contactGroup) {
        //contact exchange
        System.out.println("Removed "+contact +" from "+contactGroup);
    }
```

```java
//Interface implemented by all concrete
//command classes
public interface Command {
    void execute();
```

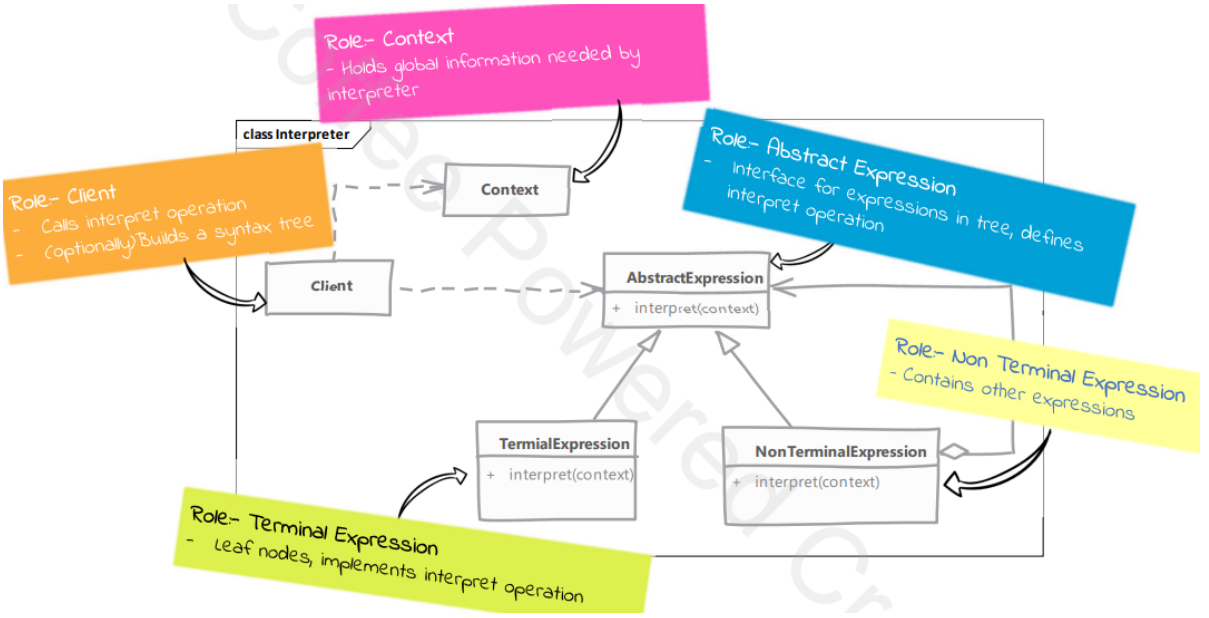
commd concret

```java
//A Concrete implementation of Command.
public class AddMemberCommand implements Command {
    private String emailAddress;
    private String listName;
    private EWSService receiver;
    public AddMemberCommand(String email, String listName, EWSService service) {
        this.emailAddress = email;
        this.listName = listName;
        this.receiver = service;
    }
    @Override
    public void execute() {
        receiver.addMember(emailAddress, listName);
    }
```

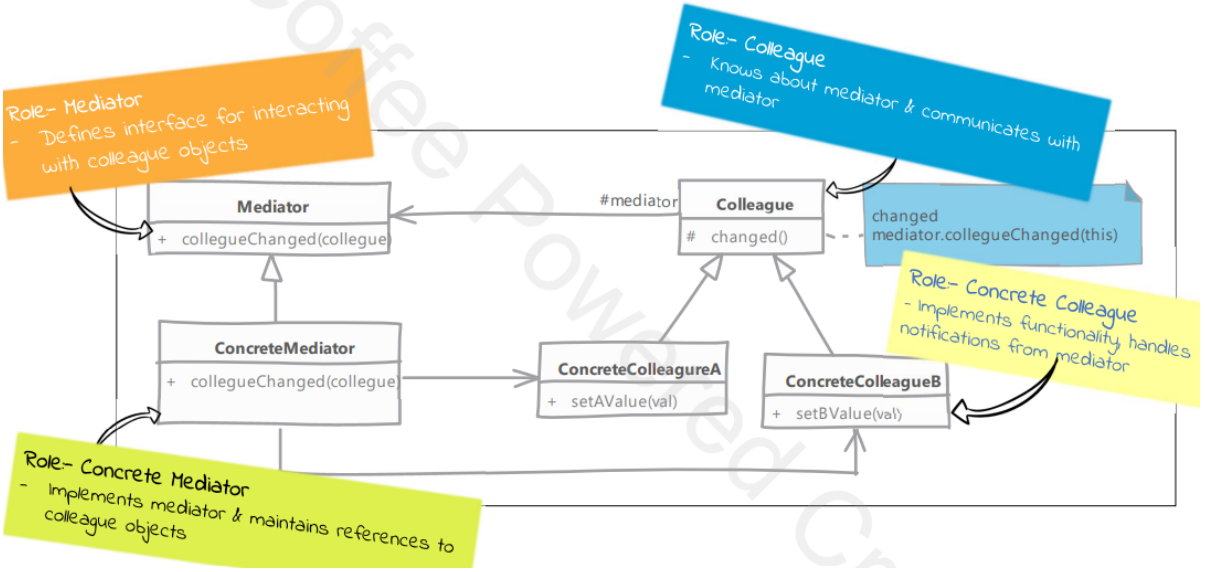
le client

```java
public static void main(String[] args) throws InterruptedException {
    EWSService service = new EWSService();
    Command c1 = new AddMemberCommand("sd@sd.cd", "spam", service);
    MailTasksRunner.getInstance().addCommand(c1);

    Command c2 = new AddMemberCommand("b@b", "spam", service);
    MailTasksRunner.getInstance().addCommand(c2);
    Thread.sleep(3000);
    MailTasksRunner.getInstance().shutdown();
```

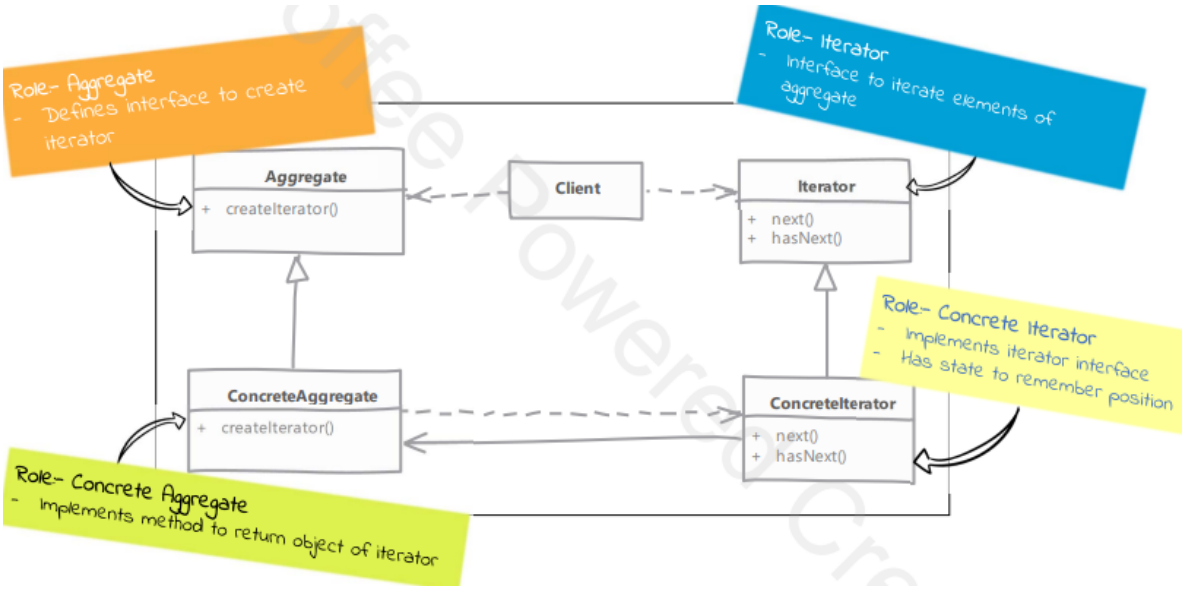
Interpreter: Interprêté les régles: ex lire un fichier exige d'avoir un user et admin role



Mediator: simplifie la communication entre composants



Iterator:



ex:

```
//Iterator interface allowing to iterate over
//values of an aggregate
public interface Iterator<T> {
    boolean hasNext();
    T next();
```

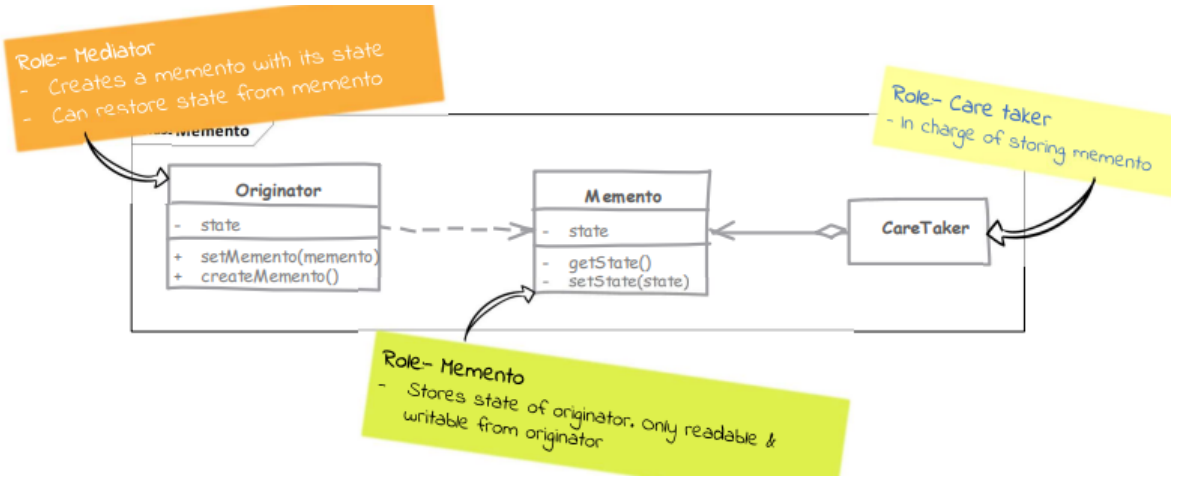
la liste d'obj

```
//This enum represents the aggregate from iterator pattern
public enum ThemeColor {
    RED,
    ORANGE,
    BLACK,
    WHITE;
    public static Iterator<ThemeColor> getIterator() {
        return new ThemeColorIterator();
    }
    private static class ThemeColorIterator implements Iterator<ThemeColor> {
        private int position;
        @Override
        public boolean hasNext() {
            return position < ThemeColor.values().length;
        }
        @Override
        public ThemeColor next() {
            return ThemeColor.values()[position++];
        }
    }
```

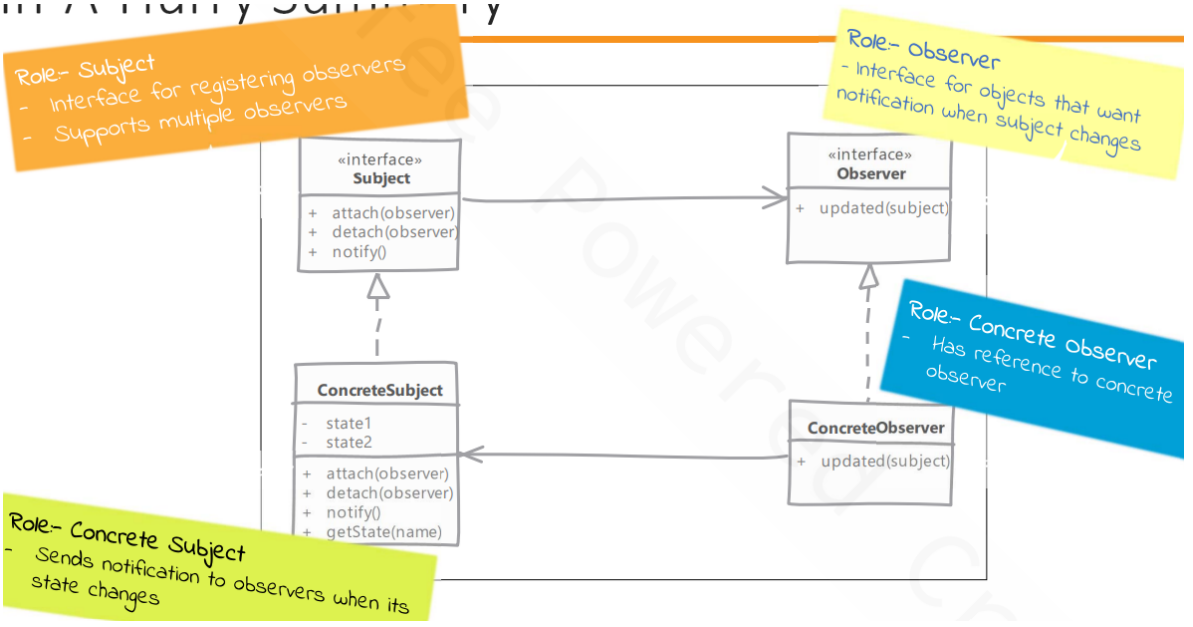
use case

```
Iterator<ThemeColor> iter = ThemeColor.getIterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Memento: Enreg l'etat d'un obj sans exposer le detail



Observer:



Subject

```
//A concrete subject
public class Order {


    private List<OrderObserver> observers = new LinkedList<>();

    public void attach(OrderObserver observer){
        observers.add(observer);
    }

    public void detach(OrderObserver observer) {
        observers.remove(observer);
    }

    public void addItem(double price) {
        itemCost += price;
        count ++;
        observers.stream().forEach(e->e.updated(this));
    }
```

Attaching Observers to Subject

```
Order order = new Order("100");
PriceObserver price = new PriceObserver();
order.attach(price);
QuantityObserver quant = new QuantityObserver();
order.attach(quant);
```

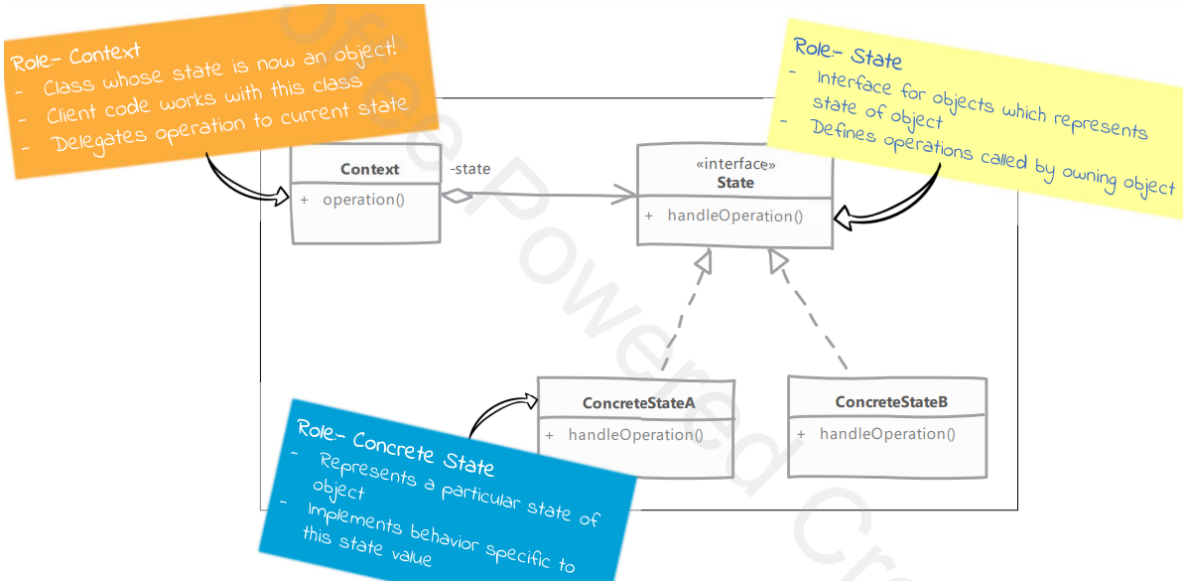
Observer

```
//Abstract observer
public interface OrderObserver {

    void updated(Order order);
}
```

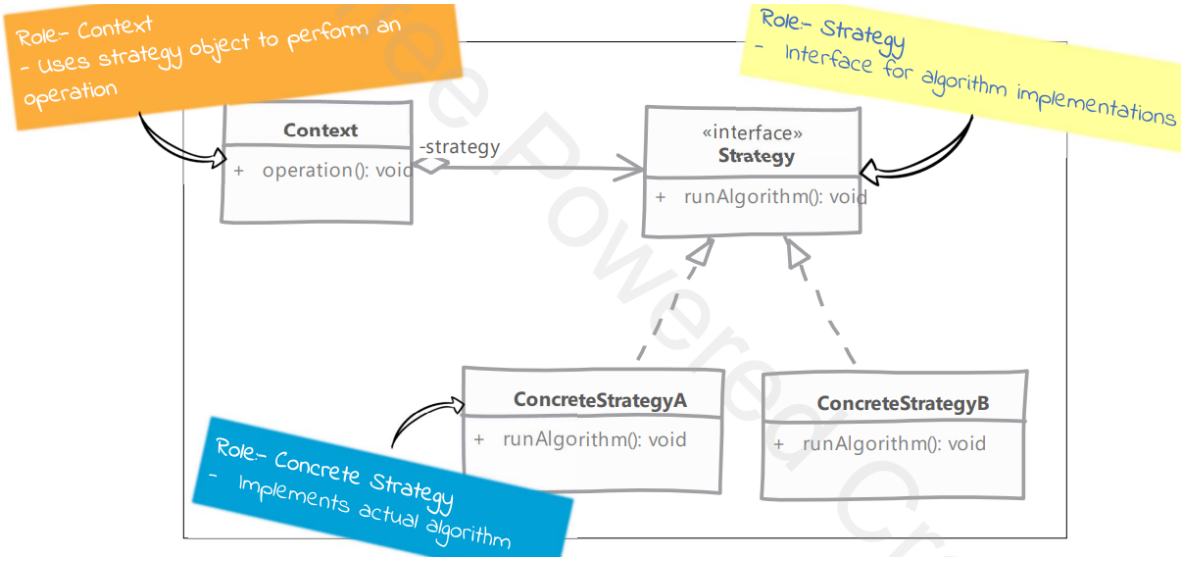
Concrete Observer

```
//Concrete observer
public class PriceObserver implements OrderObserver {

    @Override
    public void updated(Order order) {
        double total = order.getItemCost();

        if(total >= 500) {
            order.setDiscount(20);
        } else if(total >= 200) {
            order.setDiscount(10);
        }
    }

}
```

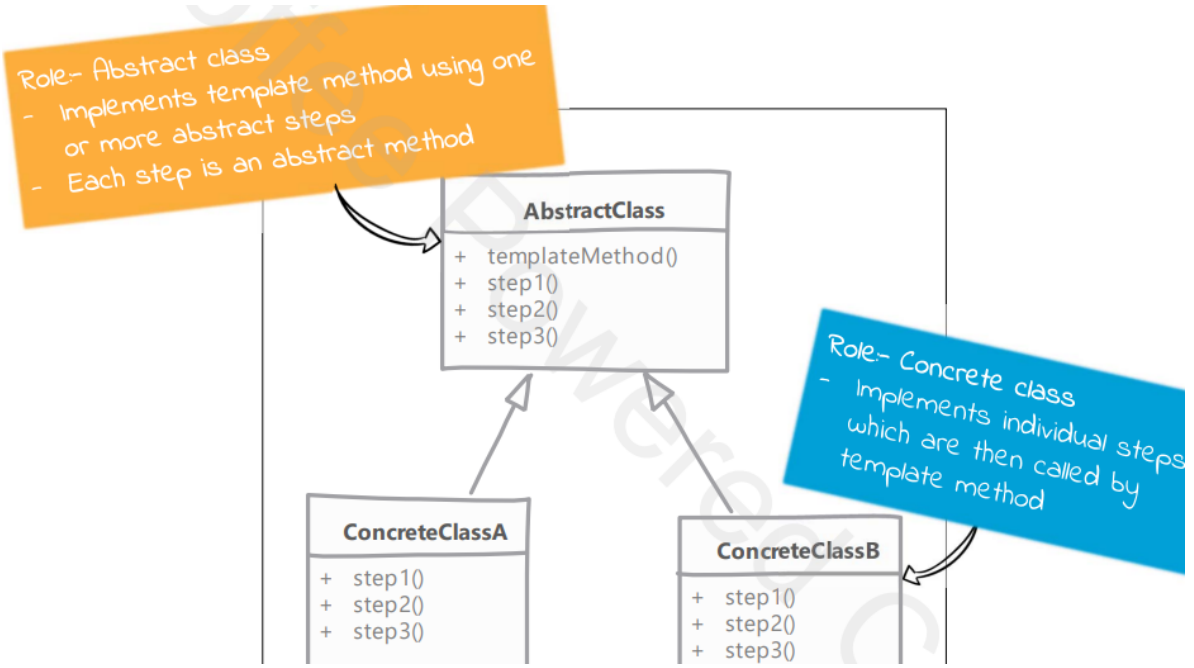
State:



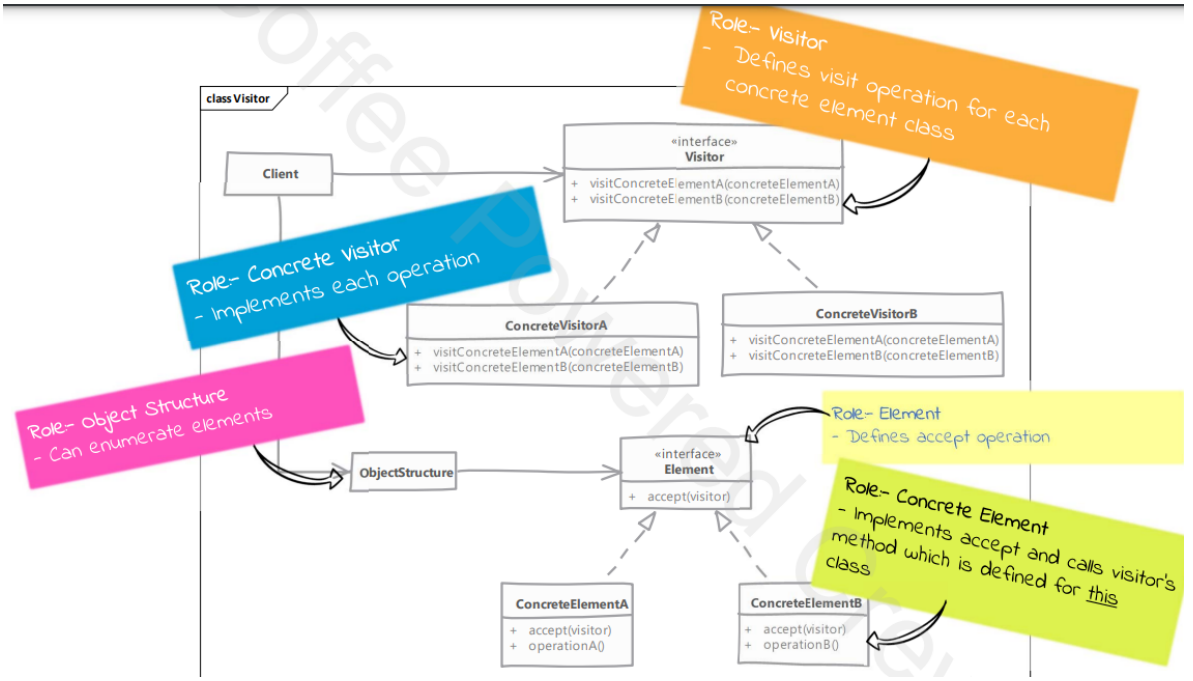
Strategy:



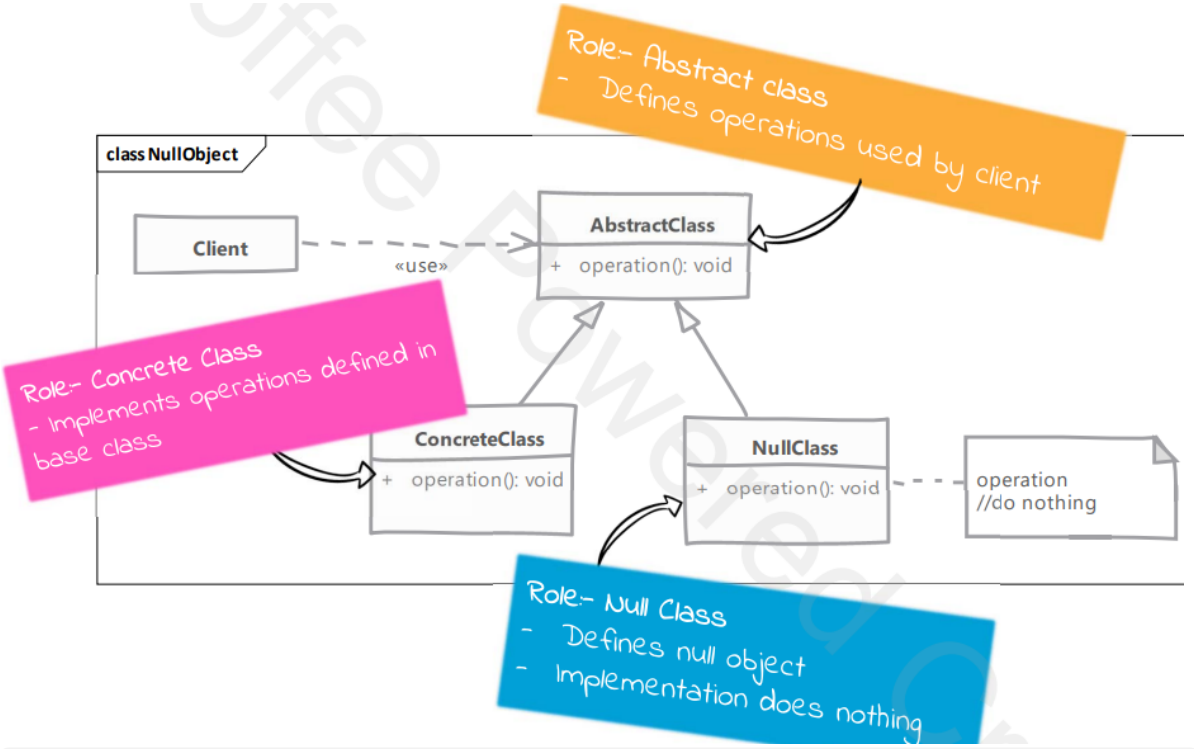
Template method:



Visitor:



Null Object:



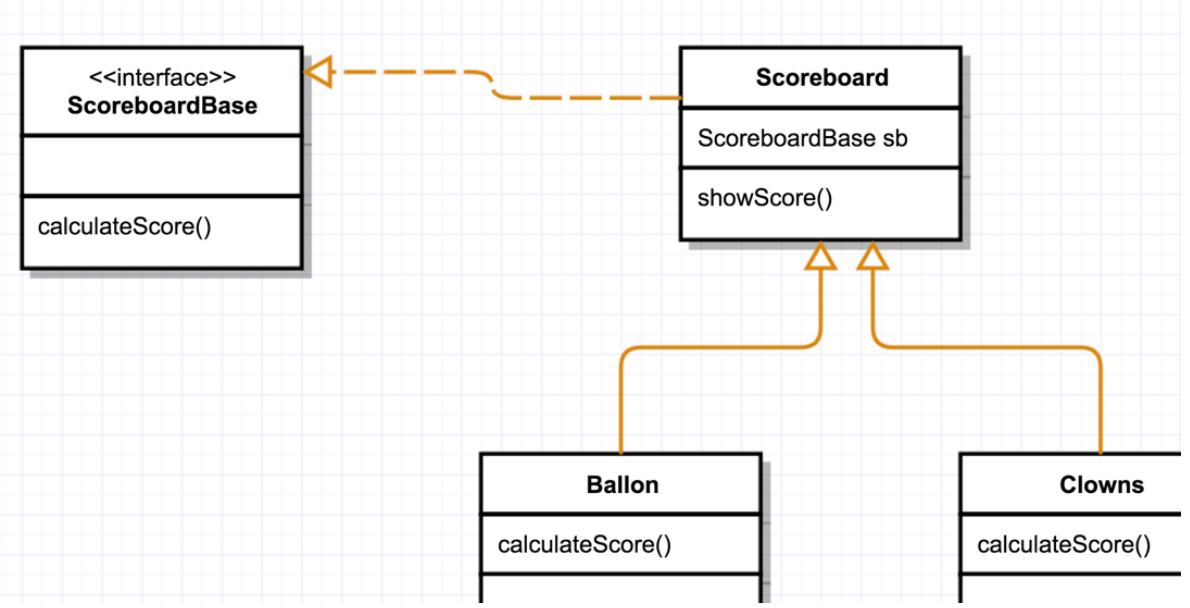
Untitled Attachment

Untitled Attachment

Master Java Design Patterns

Strategy:

ex en cours (ici un abstract classe ou on declare une method de calcul abstraite)



classe abstraite de base

```java
public abstract class ScoreAlgorithmBase {
    1 usage   3 implementations
    public abstract int calculateScore(int taps, int multiplier);
```

une classe qui a comme composition la classe abstraite

```java
public class ScoreBoard {
    4 usages
    public ScoreAlgorithmBase algorithmBase;
    3 usages
    public void showScore(int taps, int multiplier) {
        System.out.println(algorithmBase.calculateScore(taps, multiplier));
    }
```

classe réelle ou on fait le calcul

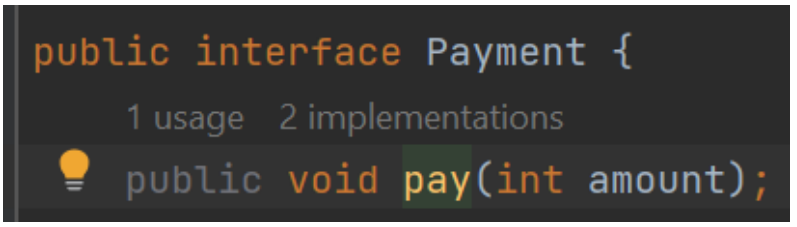
```java
public class Balloon extends ScoreAlgorithmBase {
    1 usage
    @Override
    public int calculateScore(int taps, int multiplier) {
        return (taps * multiplier) - 20;
    }
```

use case avec 2 classes réelles

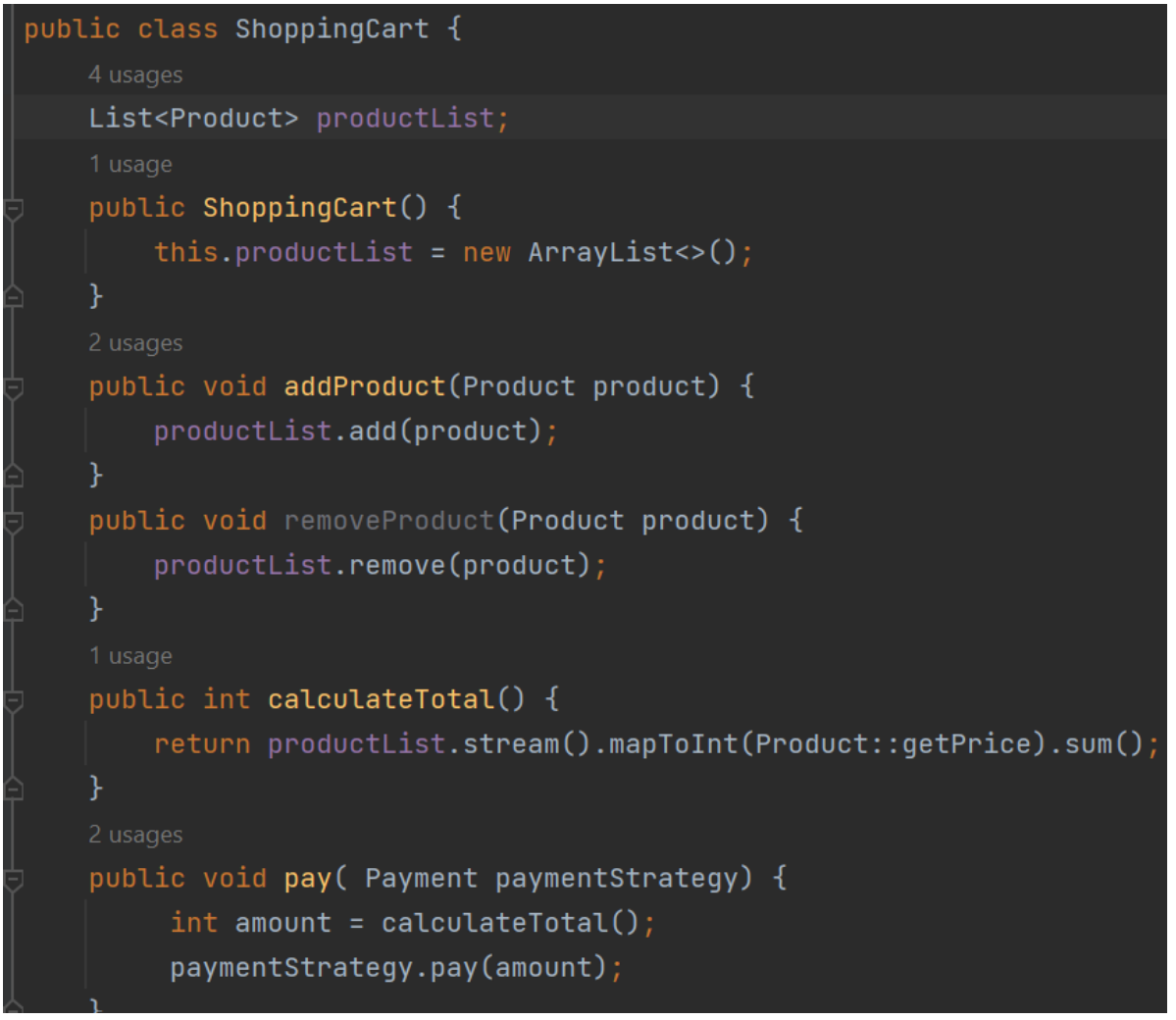
```java
public static void main(String[] args) {
    ScoreBoard scoreBoard = new ScoreBoard();
    System.out.println("Balloon Tap Score");
    scoreBoard.algorithmBase = new Balloon();
    scoreBoard.showScore( taps: 10,  multiplier: 5);

    System.out.println("Clown Tap Score");
    scoreBoard.algorithmBase = new Clown();
    scoreBoard.showScore( taps: 10,  multiplier: 5);
```

On peut use un interface au lieu d'une classe abstraite et l'implémenter. le use case ne change pas

Autre ex: ici avec interface



classe directrice qui a une composition de l'interface



classe reelle qui implemente l'interface

```java
public class PaypalAlgorithm implements Payment{
    1 usage
    private String email;
    1 usage
    private String password;
    1 usage
    public PaypalAlgorithm(String email, String password) {
        this.email = email;
        this.password = password;
    }
    1 usage
    @Override
    public void pay(int amount) {
        System.out.println( amount + " paid with Paypal");
```

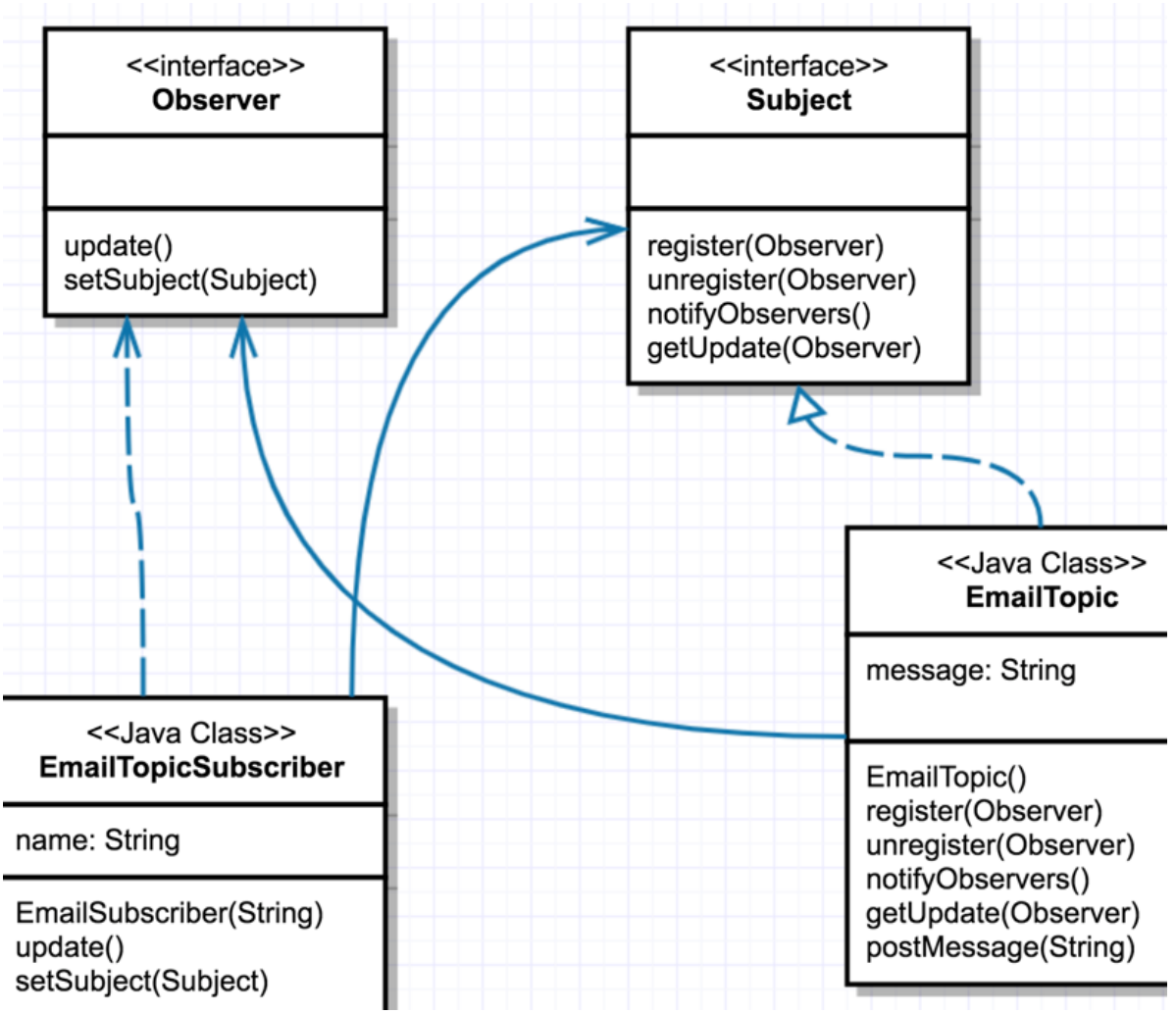
use case

```java
ShoppingCart cart = new ShoppingCart();
Product pants = new Product( upcCode: "234",  price: 25);
Product shirt = new Product( upcCode: "987",  price: 15);
cart.addProduct(pants);
cart.addProduct(shirt);
//payment decisions
cart.pay(new PaypalAlgorithm( email: "paulo@buildappswithpaulo.com",  password: "nowayman"));

cart.pay(new CreditCardAlgorithm( name: "Paulo",  cardNumber: "238756464"));
```

Observer:

On a des objects qui sont sujet d'un server. Si ce serveur change d'état tous ces obets sont notifiés et change d'état

ex ci dessous



l'interface observer

```
public interface Observer {
    2 usages   1 implementation
    void update();
    3 usages   1 implementation
    void setSubject(Subject subject);
}
```

interface subject (le getUpdate peut être de type String)

```java
public interface Subject {
    3 usages   1 implementation
    void register(Observer observer);
    1 implementation
    void unregister(Observer observer);
    1 usage   1 implementation
    void notifyObservers();
    1 usage   1 implementation
    Object getUpdate(Observer observer);
```

la classe qui implémente subject (notifyObservers permet de diffuser le postMsg)

```java
public class EmailTopic implements Subject {
    5 usages
    private List<Observer> observers;private String message;
    1 usage
    public EmailTopic(){this.observers = new ArrayList<>();}
    3 usages
    @Override
    public void register(Observer observer) {
        if(observer == null)
            throw new NullPointerException("Null object/Observer");
        if(!observers.contains(observer))
            observers.add(observer);
    }
    @Override
    public void unregister(Observer observer) {observers.remove(observer);}
    1 usage
    @Override
    public void notifyObservers() {observers.stream().forEach(Observer::update);}
    1 usage
    @Override
    public Object getUpdate(Observer observer) {return this.message;}
    1 usage
    public void postMessage(String msg){
        System.out.println("Message posted to my topic: "+msg);
        this.message = msg;notifyObservers();}
```

la classe qui implémente Observer

```java
public class EmailTopicSubscriber implements Observer {
    3 usages
    private String name;
    //Reference to our subject
    2 usages
    private Subject topic;
    3 usages
    public EmailTopicSubscriber(String name) {
        this.name = name;
    }
    2 usages
    @Override
    public void update() {
        String msg = (String) topic.getUpdate( observer: this);
        if(msg == null)
            System.out.println(name + " No new message on this topic!");
        else System.out.println(name + " Retrieving message "+msg);
    }
    3 usages
    @Override
    public void setSubject(Subject subject) {
        this.topic = subject;
    }
```

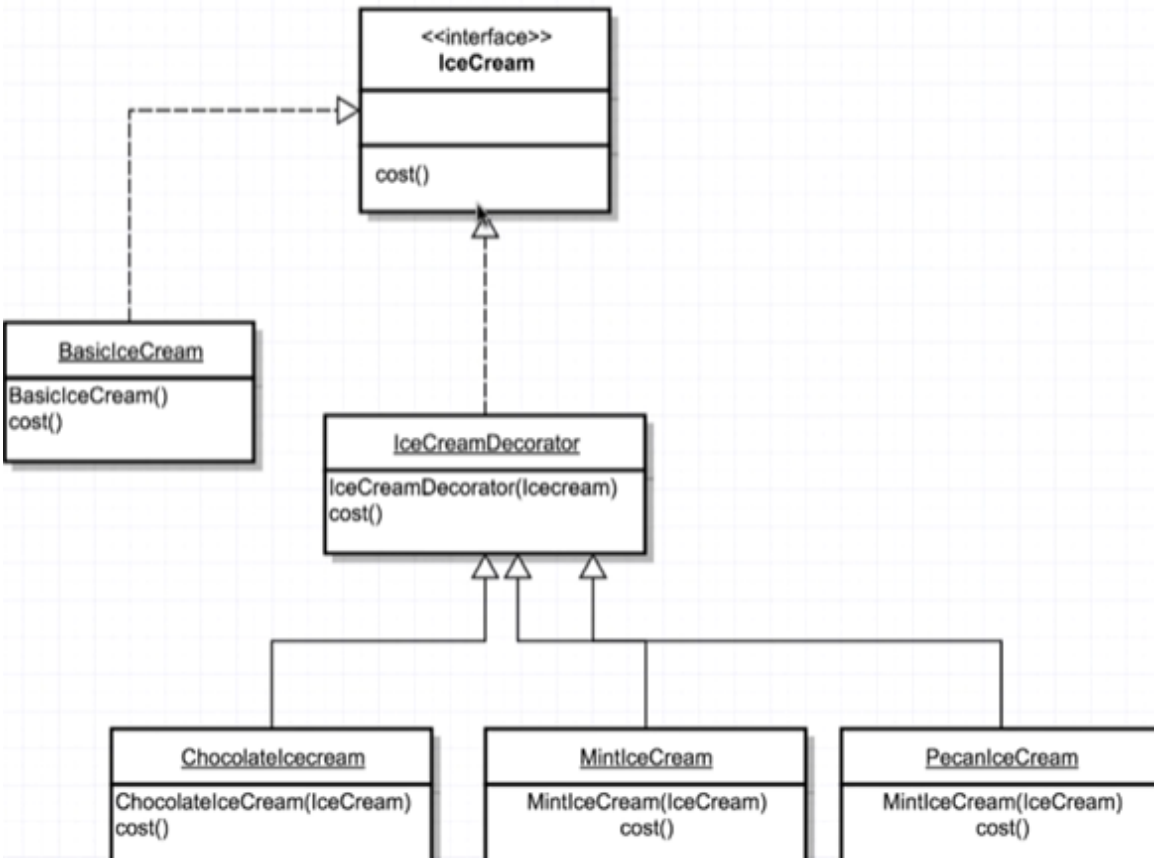
le main

```java
public static void main(String[] args) {
    EmailTopic topic = new EmailTopic();
    //create observers
    Observer firstObserver = new EmailTopicSubscriber( name: "FirstObserver");
    Observer secondObserver = new EmailTopicSubscriber( name: "SecondObserver");
    Observer thirdObserver = new EmailTopicSubscriber( name: "ThirdObserver");
    //register them
    topic.register(firstObserver);
    topic.register(secondObserver);
    topic.register(thirdObserver);
    //attaching observer to subject
    firstObserver.setSubject(topic);
    secondObserver.setSubject(topic);
    thirdObserver.setSubject(topic);
    //check for update
    firstObserver.update();//no msg becaususe no post yet
    //privuder - bloodcast - all 3 receives the message
    topic.postMessage( msg: "Hello subscribers");
```

result

```
FirstObserver No new message on this topic!
Message posted to my topic: Hello subscribers
FirstObserver Retrieving message Hello subscribers
SecondObserver Retrieving message Hello subscribers
ThirdObserver Retrieving message Hello subscribers
```

Decorator:



interface de base

```
public interface IceCream {
    7 usages   5 implementations
    double cost();
```

la classe decorator

```java
public class IceCreamDecorator implements IceCream {
    2 usages
    private IceCream iceCream;
    3 usages
    public IceCreamDecorator(IceCream iceCream){
        this.iceCream = iceCream;
    }
    7 usages  3 overrides
    @Override
    public double cost() {
        return iceCream.cost();
    }
```

la classe de base

```java
public class BasicIceCream implements IceCream {
    1 usage
    public BasicIceCream() {
        System.out.println("Creating a basic Ice-Cream");
    }
    7 usages
    @Override
    public double cost() {
        return 0.50;
    }
```

une des classe concréte

```java
public class MintIceCream extends IceCreamDecorator {
    1 usage
    public MintIceCream(IceCream iceCream) {
        super(iceCream);
    }
    7 usages
    @Override
    public double cost() {
        System.out.println("Adding Mint Ice-Cream");
        return 1.50 + super.cost();
    }
```

le main

```java
public static void main(String[] args) {
    IceCream basicIceCream = new BasicIceCream();
    System.out.println(String.format("basic Ice-Cream cost $%s", basicIceCream.cost()));
    //Add vanilla to the order
    IceCream vanilla = new VanillaIceCream(basicIceCream);
    System.out.println(String.format("Adding Vanilla cost $%s", vanilla.cost()));
    //Add Mint
    IceCream mint = new MintIceCream(vanilla);
    System.out.println(String.format("Adding mint cost $%s", mint.cost()));
```