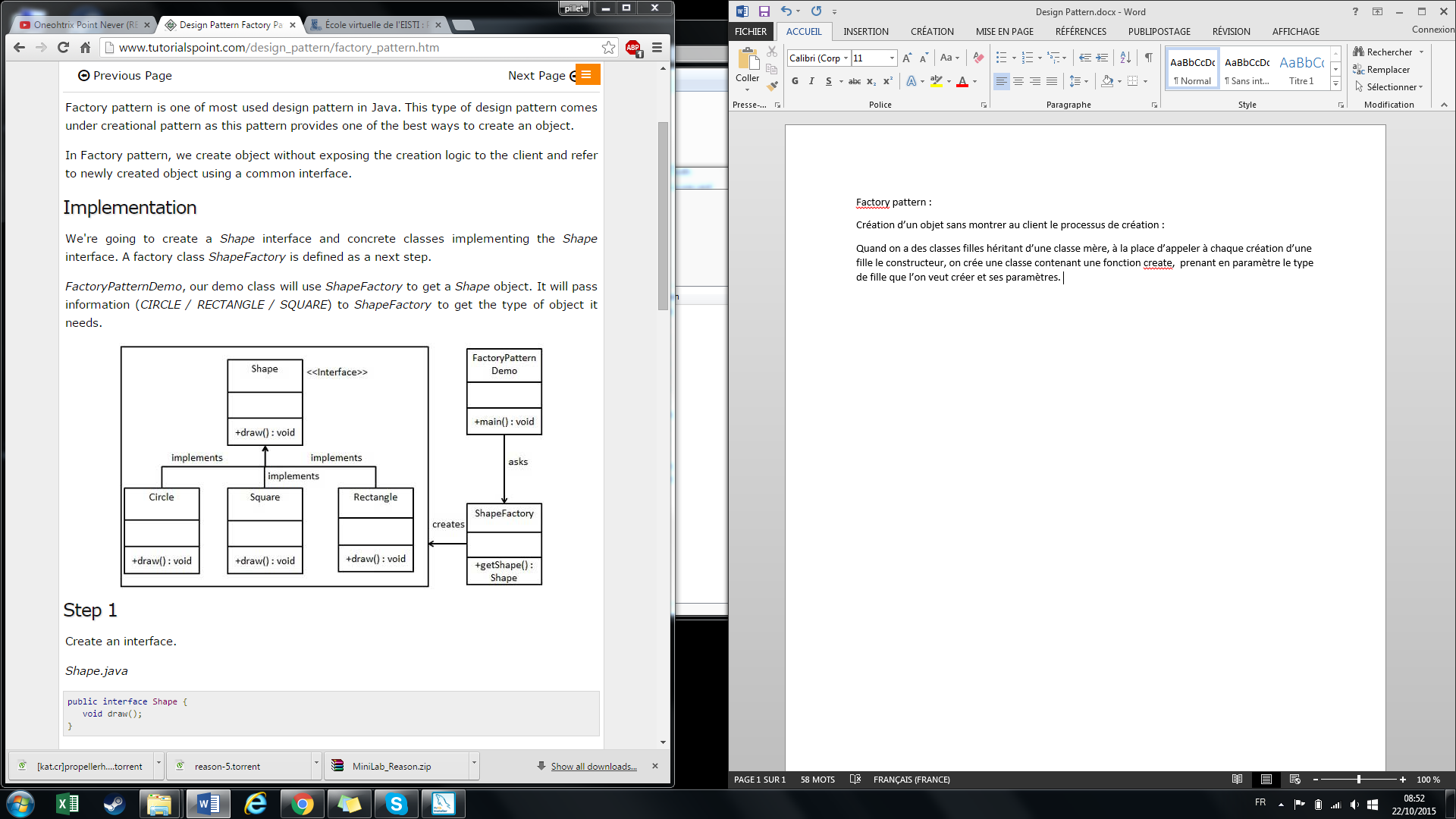
# Factory pattern :

Création d’un objet sans montrer au client le processus de création :

Quand on a des classes filles héritant d’une classe mère, à la place d’appeler à chaque création d’une fille le constructeur, on crée une classe contenant une fonction create, prenant en paramètre le type de fille que l’on veut créer et ses paramètres.



Ici, on a 3 formes : cercle, carré, rectangle. Les 3 formes héritent d’une interface Shape. Chaque forme a sa fonction dessiner.

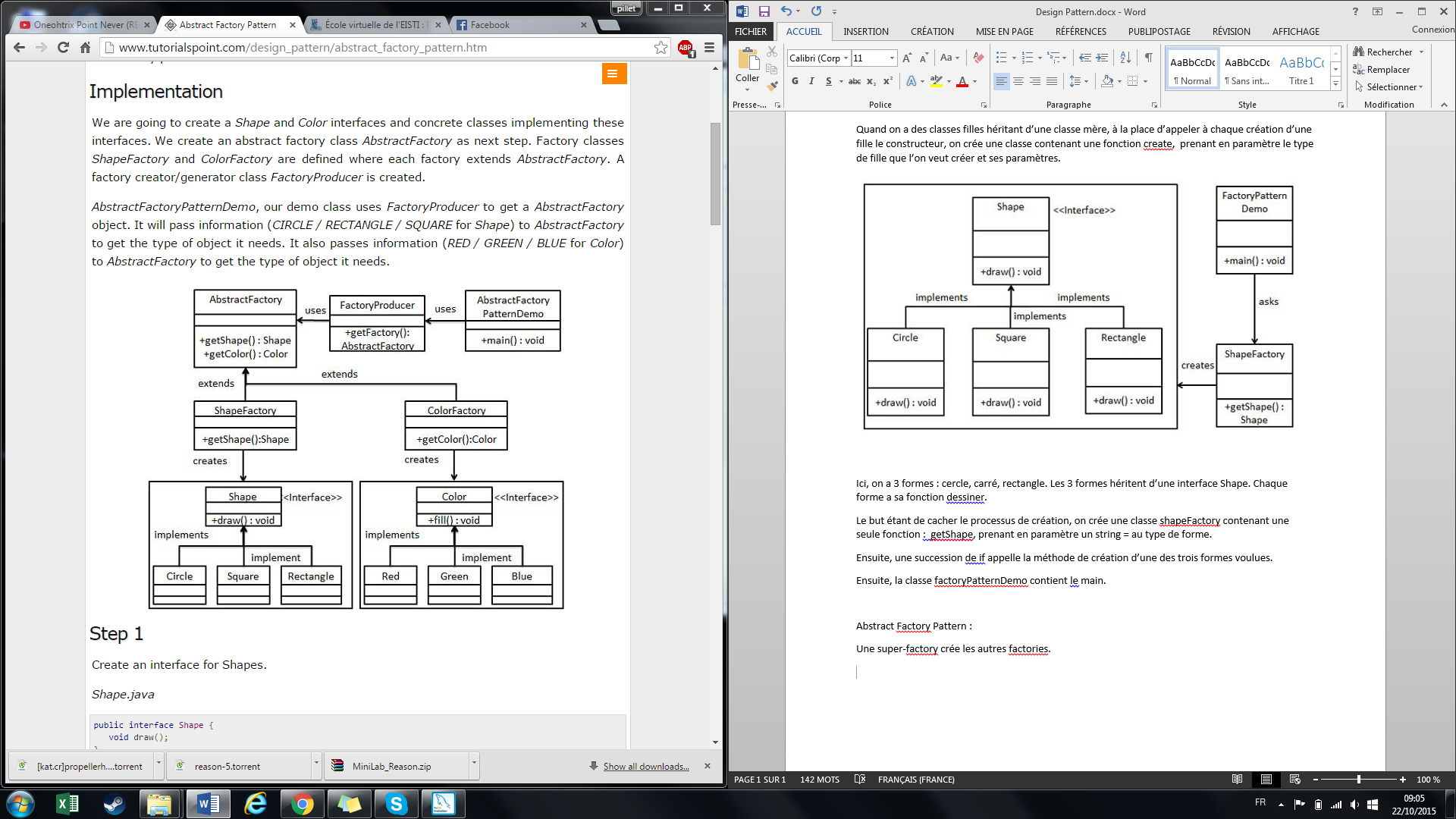
Le but étant de cacher le processus de création, on crée une classe shapeFactory contenant une seule fonction : getShape, prenant en paramètre un string = au type de forme.

Ensuite, une succession de if appelle la méthode de création d’une des trois formes voulues.

Ensuite, la classe factoryPatternDemo contient le main.

# Abstract Factory Pattern :

Une super-factory crée les autres factories.



On crée d’abord une interface shape. On crée 3 classes implémentant cette interface : un cercle, un carré, un rectangle, chacune implémentant la méthode draw.

On crée d’un autre coté une interface color, implémentée par 3 classes : Red, Green, Blue, implémentant la méthode fill().

On crée ensuite une classe abstraite de Factory pour les shapes et pour les colors. Elle a deux méthodes : getShape et getColor Cette classe hérite de deux filles :

Comme dans les Factory normales, ici, la classe shapeFactory prend en paramètre une shape et la crée, et la Color Factory prend en paramètre une couleur et la crée. Ces deux classes héritent de l’AbstractFactory.

On crée ensuite un Factory Producer pour avoir des factories des classes concrètes, utilisant des informations comme la forme ou la couleur :

Une seule méthode : getFactory(string) : prend en paramètre soit Shape, dans ce cas on instancie la classe ShapeFactory, soit color, dans ce cas on instancie la classe ColorFactory.

Pour finir, on crée la dernière classe : le main pour tester.

D’abord, on appelle la méthode de FactoryProducer pour créer une forme :

AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

Ceci va uniquement créer un objet qui s’appelera shapeFactory et qui nous permettra de créer n’importe quelle forme. Maintenant que l’on a la shapeFactory, on peut créer une forme :

Shape shape1 = shapeFactory.getShape("CIRCLE");

Maintenant, on a une forme1, on peut donc la dessiner.

shape1.draw();

# MVC :

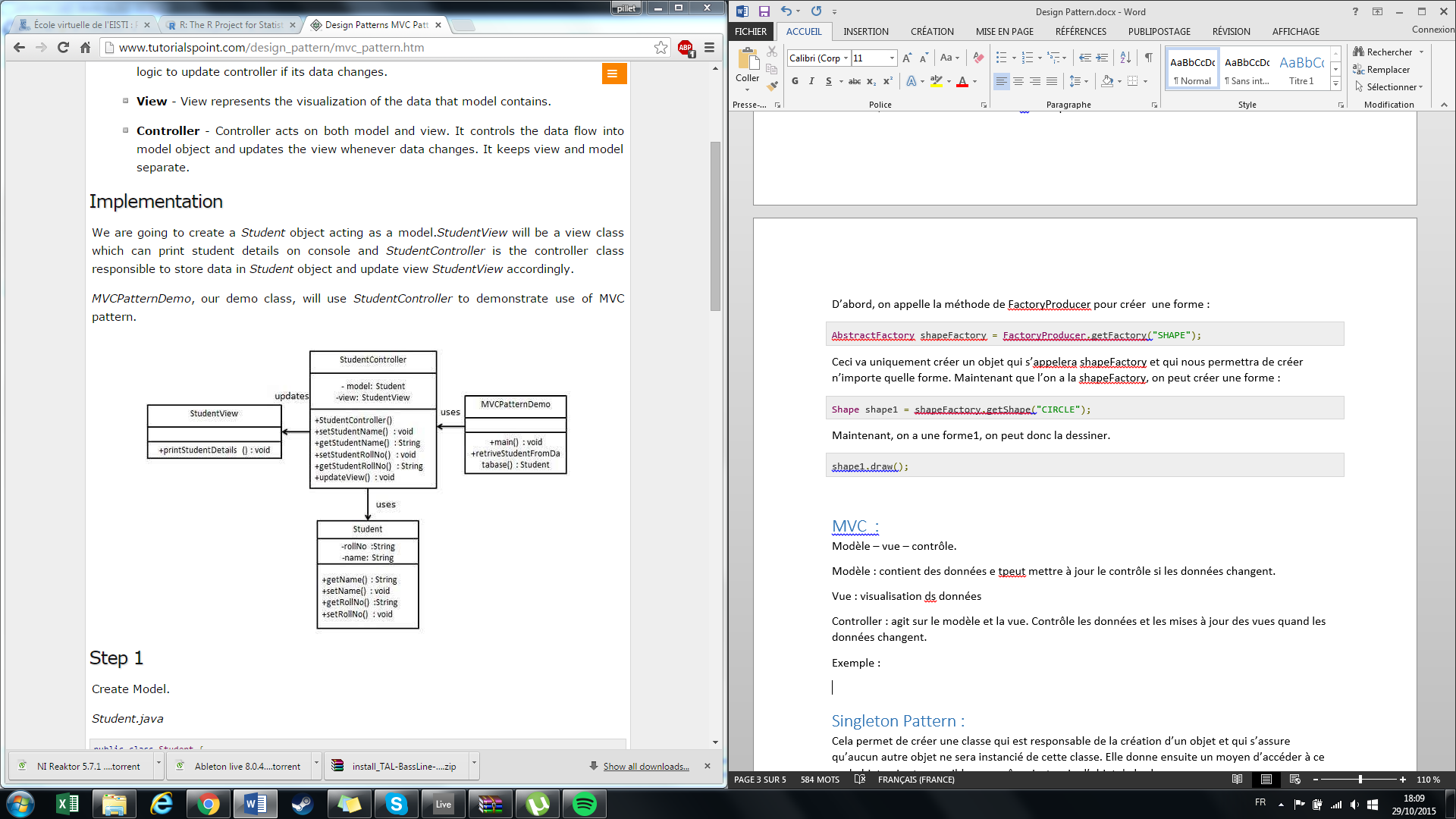
Modèle – vue – contrôle.

Modèle : contient des données e tpeut mettre à jour le contrôle si les données changent.

Vue : visualisation ds données

Controller : agit sur le modèle et la vue. Contrôle les données et les mises à jour des vues quand les données changent.

Exemple :



Ici, on a une classe student qui est le modèle. Une classe vue qui affichera les détails du student sur la console, et une classe studentcontroller qui stockera les données dans l’objet student et les mettra a jour.

On crée d’abord une classe student. On lui donne un nom et nu ID. on lui donne également des geters et des seters.

Ensuite, on crée la classe studentView : on lui donne simplement une méthode qui va afficher tous les détails de l’étudiant, et qui prendra en paramètre le nom de l’étudiant et son ID.

Pour finir, on crée la classe student controller :

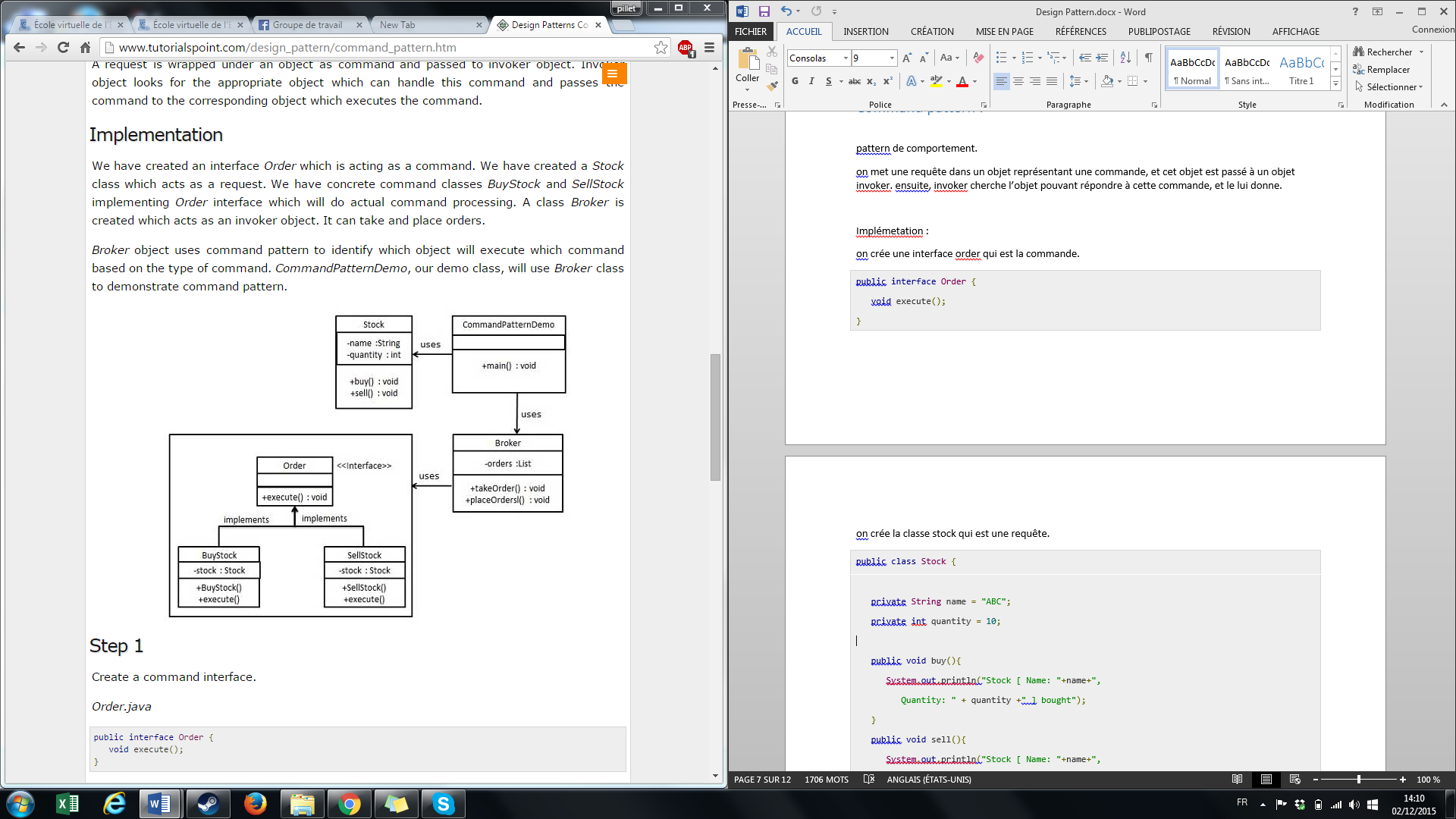
Elle prend en paramètre un student et une view. Le controleur prend toujours en paramètre les 2 objets de vue et de modèle. Ici, on donne donc à un objet controller un attribut student et un attribut view. C’est le controller qui appelera les méthodes set, get pour l’ID et le nom du student, et qui se chargera d’afficher les détails student.

**On utilise jamais les méthodes contenues dans les classes view et model. elles seront automatiquement appelées dans la classe control. Elle prendra en attribut les instances sur lesquelles on veut agir.**

# Command pattern :

pattern de comportement.

on met une requête dans un objet représentant une commande, et cet objet est passé à un objet invoker. ensuite, invoker cherche l’objet pouvant répondre à cette commande, et le lui donne.



Implémetation :

on crée une interface order qui est la commande.

public interface Order {

void execute();

}

on crée la classe stock qui est une requête qui sera appelée directement dans le main.

public class Stock {

private String name = "ABC";

private int quantity = 10;

public void buy(){

System.out.println("Stock [ Name: "+name+",

Quantity: " + quantity +" ] bought");

}

public void sell(){

System.out.println("Stock [ Name: "+name+",

Quantity: " + quantity +" ] sold");

}

}

on crée une implémentation de commande qui est buyStock et sellStock : ces classes vont se charger de répondre à la commande. on ne les appelera jamais directement.

public class BuyStock implements Order {

private Stock abcStock;

public BuyStock(Stock abcStock){

this.abcStock = abcStock;

}

public void execute() {

abcStock.buy();

}

}

public class SellStock implements Order {

private Stock abcStock;

public SellStock(Stock abcStock){

this.abcStock = abcStock;

}

public void execute() {

abcStock.sell();

}

}

on crée une classe broker qui fait l’intermédiaire entre la requête et la commande.

on crée cet objet, qui a une méthode takeOrder, prenant en paramètre un ordre, et placeOrder, qui exécute la liste d’ordres avec execute().

import java.util.ArrayList;

import java.util.List;

public class Broker {

private List<Order> orderList = new ArrayList<Order>();

public void takeOrder(Order order){

orderList.add(order);

}

public void placeOrders(){

for (Order order : orderList) {

order.execute();

}

orderList.clear();

}

}

on a alors dans le main :

public class CommandPatternDemo {

public static void main(String[] args) {

Stock abcStock = new Stock();

BuyStock buyStockOrder = new BuyStock(abcStock);

SellStock sellStockOrder = new SellStock(abcStock);

Broker broker = new Broker();

broker.takeOrder(buyStockOrder);

broker.takeOrder(sellStockOrder);

broker.placeOrders();

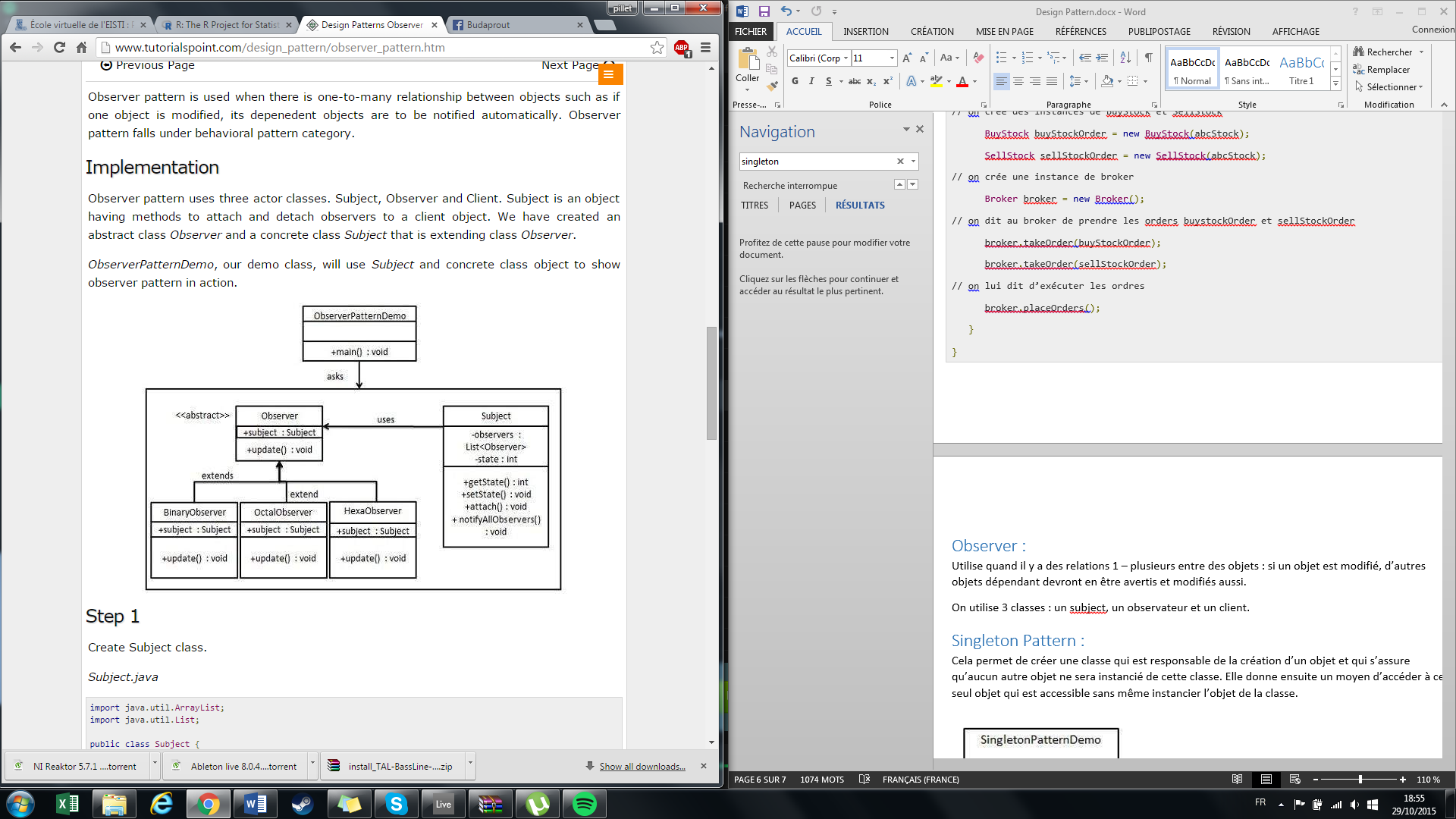
}

}

on crée tout d’abord un stock, un objet. quelqconque qu’on va placer dans une requête order.

une fois qu’on a placé ces objets dans des requêtes, on les donne à la méthode takeOrder de broker, qui exécutera automatiquement les tâches demandées selon le nom de l’ordre.

# Observer :



quand il y a des relations entre des objets : quand un objet est modifié, les autres objets dépendants ne sont pas automatiquement modifiés. Pattern de comportement.

il faut donc un observer lié à un objet, et quand cet objet est modifié, l’observer va update toutes les autres classes dépendantes.

3 classes : sujet, observer, et client.

Subject : objet ayant des méthodes attachant et détachant les observers à un client.

import java.util.ArrayList;

import java.util.List;

public class Subject {

private List<Observer> observers = new ArrayList<Observer>();

private int state;

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

notifyAllObservers();

}

public void attach(Observer observer){

observers.add(observer);

}

public void notifyAllObservers(){

for (Observer observer : observers) {

observer.update();

}

}

}

on crée une classe abstraite observer et une classe concrete subject qui hérite de observer.

on crée l’observer abstrait :

public abstract class Observer {

protected Subject subject;

public abstract void update();

}

puis les objets hérités :

public class BinaryObserver extends Observer{

public BinaryObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );

}

}

et

public class OctalObserver extends Observer{

public OctalObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );

}

}

et

public class HexaObserver extends Observer{

public HexaObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );

}

}

et un main :

public class ObserverPatternDemo {

public static void main(String[] args) {

Subject subject = new Subject();

new HexaObserver(subject);

new OctalObserver(subject);

new BinaryObserver(subject);

System.out.println("First state change: 15");

subject.setState(15);

System.out.println("Second state change: 10");

subject.setState(10);

}

}

on crée d’abord le subject.

ensuite, on crée 3 objets observers, auxquels on attribue notre subject. ces objets affichent l’état de leur subject quand ils sont mis à jours.

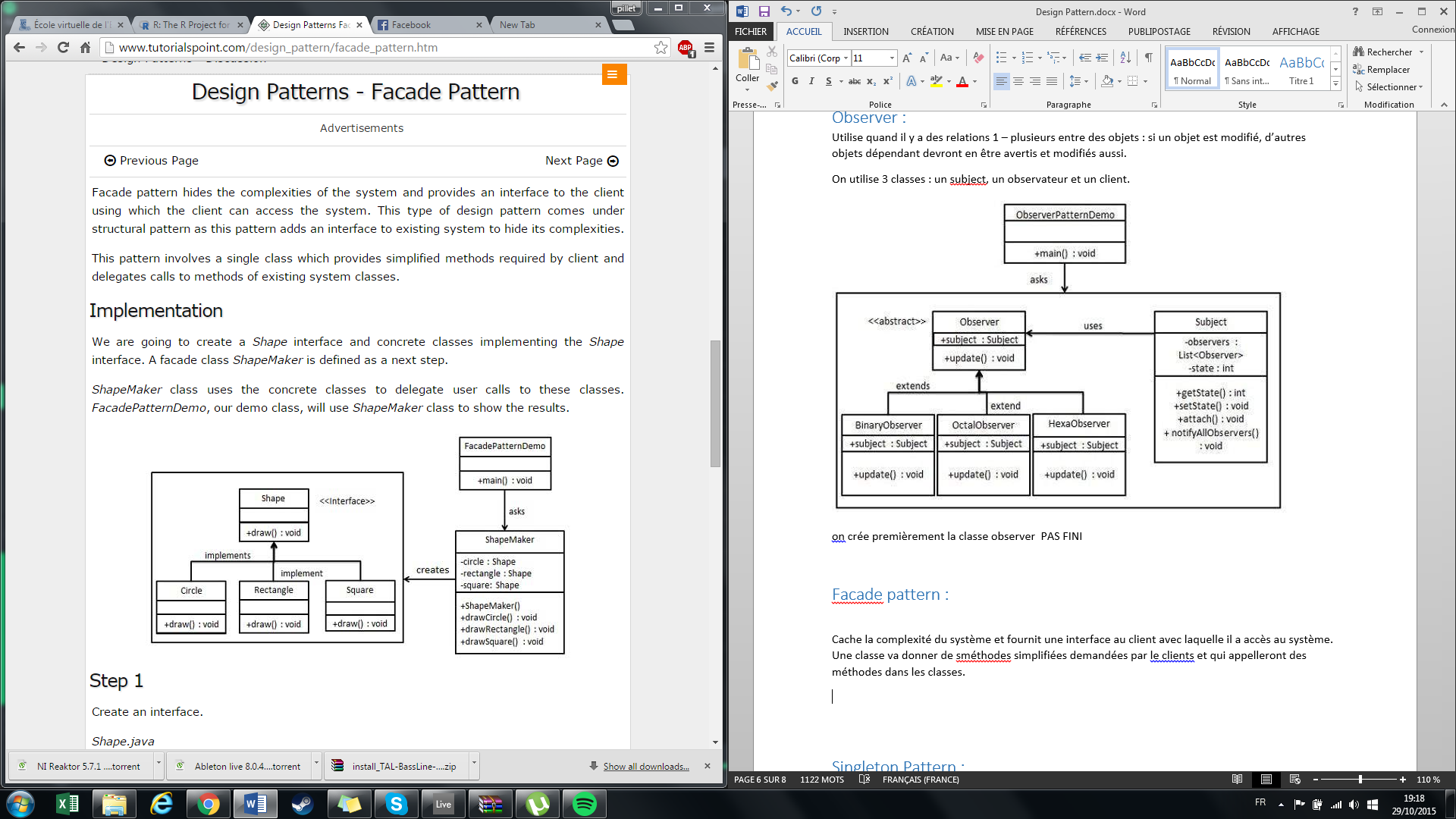
ainsi, quand on crée un objet, automatiquement, le subject passé en paramètre devient un attribut de l’observer, et on rajoute chaque subject dans un tableau de subject.

si on change subject :

subject setState : on change le state du subject, et automatiquement, tous les subjects du tableau créé plus haut sont changés aussi avec update. automatiquement, le update va faire écrire chaque objet son nouveau state.

# Facade pattern :

Cache la complexité du système et fournit une interface au client avec laquelle il a accès au système. Une classe va donner de sméthodes simplifiées demandées par le clients et qui appelleront des méthodes dans les classes.



on crée une classe shape qui est une interface, et 3 classes filles implémentant la classe shape. Un cercle, un rectangle et un carré.

Chacune ayant une méthode draw.

on crée ensuite une classe facade shapemaker. Elle utilise les méthodes des classes concretes pour déléguer.

On crée donc la classe interface :

public interface Shape {

void draw();

}

Ensuite on définit les 3 classes concrètes implémentant la méthode draw.

Ensuite, on définit la classe shapemaker. Elle prend en paramètre 3 shapes : rectangle, triangle, square.

Le constructeur va donc créer des des instances et les attribuer aux attributs.

Elle implémentera des méthodes dessinant chaque forme.

Ensuite, on pourra faire :

public class FacadePatternDemo {

public static void main(String[] args) {

ShapeMaker shapeMaker = new ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

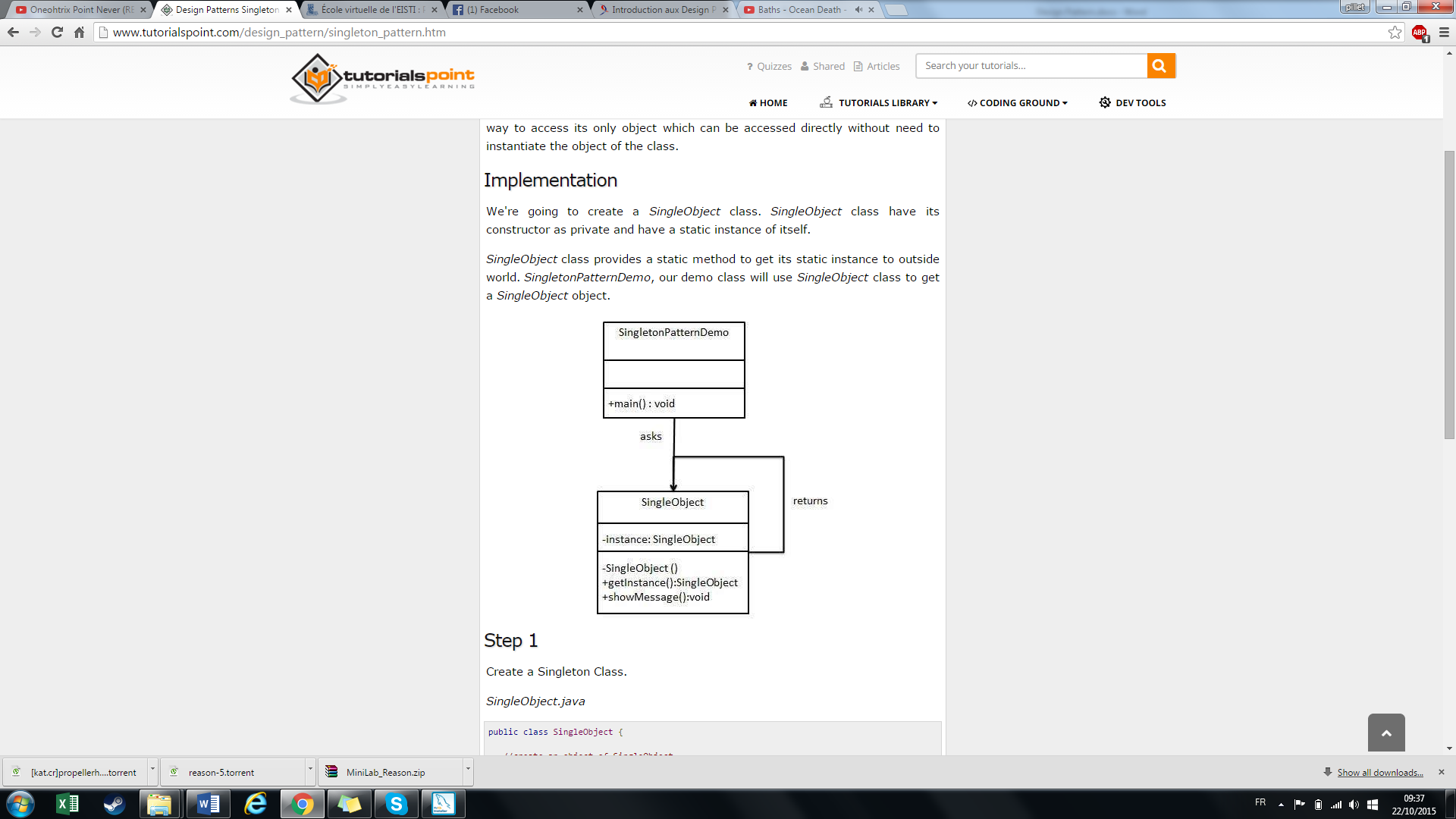
shapeMaker.drawSquare();

}

}

# Singleton Pattern :

Cela permet de créer une classe qui est responsable de la création d’un objet et qui s’assure qu’aucun autre objet ne sera instancié de cette classe. Elle donne ensuite un moyen d’accéder à ce seul objet qui est accessible sans même instancier l’objet de la classe.



On crée une classe SingleObject.

On lui donne une instance statique, ainsi elle instancie un seul objet qui est déjà créé lorsque la classe est créée :

private static SingleObject instance = new SingleObject();

Du coup, son constructeur est privé pour que l’on ne puisse pas créer d’autre d’objet depuis l’exterieur :

private SingleObject(){}

On lui donne une méthode retournant l’objet créé statiquement

Et une méthode affichant hello world.

Ensuite, on crée un main :

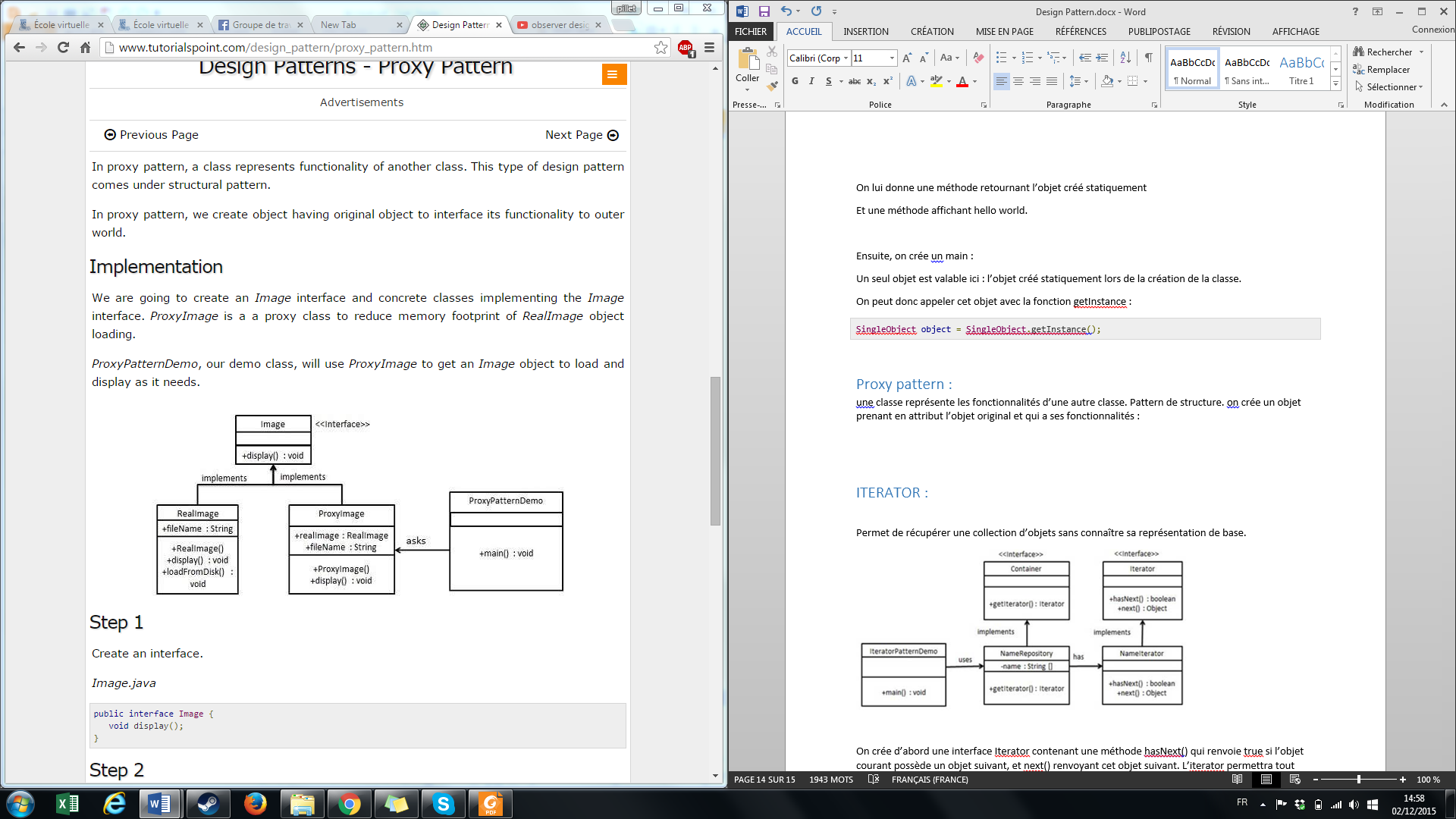
Un seul objet est valable ici : l’objet créé statiquement lors de la création de la classe.

On peut donc appeler cet objet avec la fonction getInstance :

SingleObject object = SingleObject.getInstance();

# Proxy pattern :

une classe représente les fonctionnalités d’une autre classe. Pattern de structure. on crée un objet prenant en attribut l’objet original et qui a ses fonctionnalités :



on crée une interface image, qui a une méthode display :

public interface Image {

void display();

}

on crée deux classes concrètes implémentant la même interface :

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName){

this.fileName = fileName;

loadFromDisk(fileName);

}

@Override

public void display() {

System.out.println("Displaying " + fileName);

}

private void loadFromDisk(String fileName){

System.out.println("Loading " + fileName);

}

}

et le proxy :

le proxy prend en paramètre une realImage et un fileName.

public class ProxyImage implements Image{

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName){

this.fileName = fileName;

}

@Override

public void display() {

if(realImage == null){

realImage = new RealImage(fileName);

}

realImage.display();

}

}

ensuite, dans le main :

public class ProxyPatternDemo {

public static void main(String[] args) {

Image image = new ProxyImage("test\_10mb.jpg");

//image will be loaded from disk

image.display();

System.out.println("");

//image will not be loaded from disk

image.display();

}

}

on crée une image proxy, et on l’affiche avec sa méthode.

au début, l’image ne sera pas chargée par défaut.

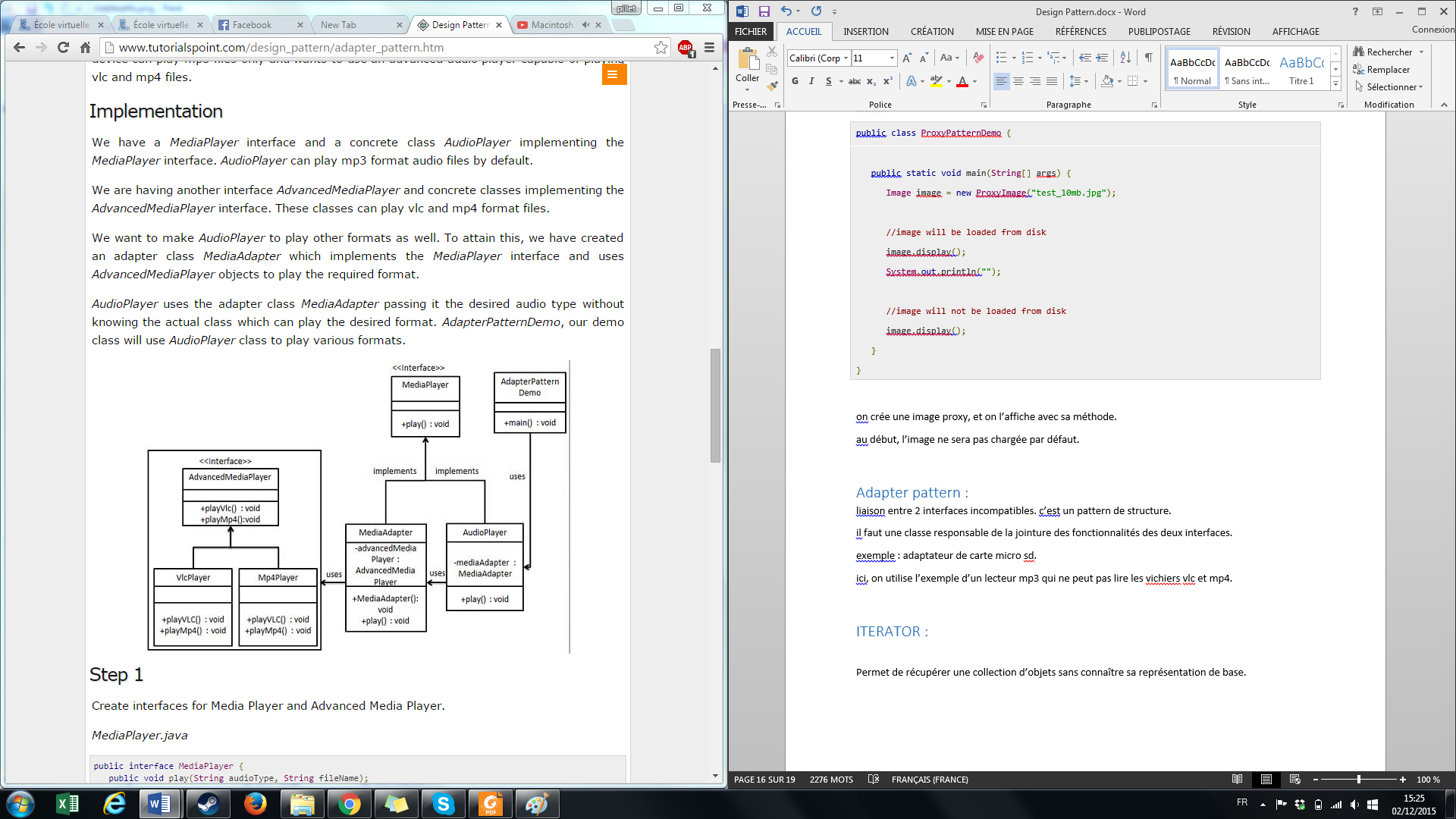
# Adapter pattern :

liaison entre 2 interfaces incompatibles. c’est un pattern de structure.

il faut une classe responsable de la jointure des fonctionnalités des deux interfaces.

exemple : adaptateur de carte micro sd.

ici, on utilise l’exemple d’un lecteur mp3 qui ne peut pas lire les vichiers vlc et mp4.



on crée une interface MediaPlayer, ayant juste une méthode play, prenant en compte le type et le nom du fichier.

public interface MediaPlayer {

public void play(String audioType, String fileName);

}

on crée ensuite le médiaPlayer avancé, capable de lire les fichiers vlc et mp4 :

d’abord sous forme d’interface :

public interface AdvancedMediaPlayer {

public void playVlc(String fileName);

public void playMp4(String fileName);

}

puis sous forme de classe concrè-te :

public class VlcPlayer implements AdvancedMediaPlayer{

@Override

public void playVlc(String fileName) {

System.out.println("Playing vlc file. Name: "+ fileName);

}

@Override

public void playMp4(String fileName) {

//do nothing

}

}

public class Mp4Player implements AdvancedMediaPlayer{

@Override

public void playVlc(String fileName) {

//do nothing

}

@Override

public void playMp4(String fileName) {

System.out.println("Playing mp4 file. Name: "+ fileName);

}

}

on crée la classes adapter implémentant l’interface MediaPlayer :

public class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter(String audioType){

if(audioType.equalsIgnoreCase("vlc") ){

advancedMusicPlayer = new VlcPlayer();

}else if (audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer = new Mp4Player();

}

}

@Override

public void play(String audioType, String fileName) {

if(audioType.equalsIgnoreCase("vlc")){

advancedMusicPlayer.playVlc(fileName);

}

else if(audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer.playMp4(fileName);

}

}

}

la classe prend en paramètre le mediaPlayer avancé.

dans son constructeur, si le type est vlc ou mp4, son attribut mediaplayer avancé sera un player mp4 ou vlc.

ensuite, la méthode play appelera un fichier à jouer et utilisera soit le mediaplayer vlc soit le mp4 qu’il a en attribut.

on crée une classe concrete implémentant médiaPlayer :

public class AudioPlayer implements MediaPlayer {

MediaAdapter mediaAdapter;

@Override

public void play(String audioType, String fileName) {

//inbuilt support to play mp3 music files

if(audioType.equalsIgnoreCase("mp3")){

System.out.println("Playing mp3 file. Name: " + fileName);

}

//mediaAdapter is providing support to play other file formats

else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){

mediaAdapter = new MediaAdapter(audioType);

mediaAdapter.play(audioType, fileName);

}

else{

System.out.println("Invalid media. " + audioType + " format not supported");

}

}

}

il prend en paramètre un mediaAdapter.

dans sa méthode play, si c’est un mp3, on peut jouer le mp3 directement  
sinon, si c’est un mp3 ou un mp4, on lui donne en attribut un mediaAdapter, prenant en paramètre l’audiotype demandé, et on utilise sa méthode play.

dans le main :

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp3", "beyond the horizon.mp3");

audioPlayer.play("mp4", "alone.mp4");

audioPlayer.play("vlc", "far far away.vlc");

audioPlayer.play("avi", "mind me.avi");

}

}

# Composite pattern :

utilisé quand on doit utiliser un groupe d’objets de la même façon.

pattern de structure :

on crée une classe contenant un groupe de ses propres objets, donnant des moyens de modifier ce groupe d’objets.

implémentation :

on crée une classe employé :

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private String dept;

private int salary;

private List<Employee> subordinates;

// constructor

public Employee(String name,String dept, int sal) {

this.name = name;

this.dept = dept;

this.salary = sal;

subordinates = new ArrayList<Employee>();

}

public void add(Employee e) {

subordinates.add(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

public List<Employee> getSubordinates(){

return subordinates;

}

public String toString(){

return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");

}

}

la classe possède des attributs classiques, et une liste d’employés.

on a donc nu constructeur classique,

une méthode pour ajouter un employé à la liste

une pour en supprimer

une pour retournier la liste

une pour afficher des infos

ensuite, dans le main :

public class CompositePatternDemo {

public static void main(String[] args) {

Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);

Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);

CEO.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

//print all employees of the organization

System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {

System.out.println(headEmployee);

for (Employee employee : headEmployee.getSubordinates()) {

System.out.println(employee);

}

}

}

}

on crée plusieurs employés, et on peut les grouper dans les listes de certains employés :

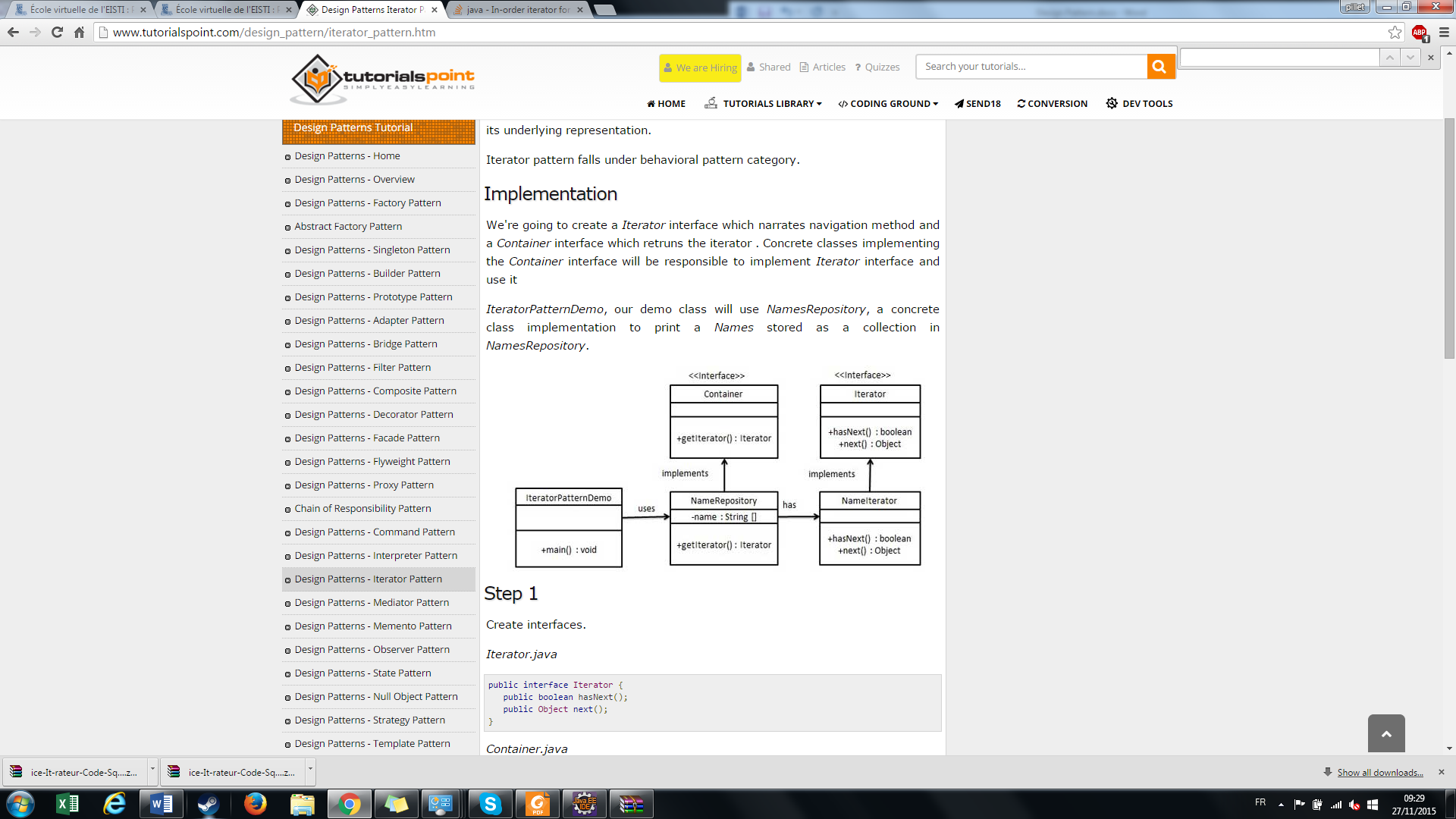
ici, dans la liste du CEO, on aura le head sales et le head marketing, qui eux meme contiendront les salesexecutives 1 et 2 pour le head sales, et les clerk1 et 2 pour le head marketing.

ainsi, à la fin, non peut afficher la liste des employés de façon hierarchique.

# Decorator pattern :

# ITERATOR :

Permet de récupérer une collection d’objets sans connaître sa représentation de base.



On crée d’abord une interface Iterator contenant une méthode hasNext() qui renvoie true si l’objet courant possède un objet suivant, et next() renvoyant cet objet suivant. L’iterator permettra tout simplement de savoir ou l’on se trouve dans la collection, et de se déplacer dans la collection.

public interface Iterator {

public boolean hasNext();

public Object next();

}

On crée ensuite une interface Container : elle sera implémentée par la classe possédant les données. D’où container.

public interface Container {

public Iterator getIterator();

}

La classe possédant les données est NameRepository : elle a un attribut string [] contenant la collection d’objets. (Ici un tableau).

public class NameRepository implements Container {

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

Elle prend également en attribut l’implémentation de l’iterator.

Elle a donc 2 attributs : une collection et un iterateur qui dira ou on est dans la collection.

Ensuite, dans le main, on appelera une boucle for pour afficher tous les éléments de la collection :

public class IteratorPatternDemo {

public static void main(String[] args) {

NameRepository namesRepository = new NameRepository();

for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){

String name = (String)iter.next();

System.out.println("Name : " + name);

}

}

}

For(iterator iter = namesRepository.getIterator() ; iter.hasnext(); ){

// la première partie de la boucle for : on veut bel et bien avoir un iterateur parcourant la liste.

Iterator est ici une interface. On crée une instance de cette interface, et on lui attribue l’iterateur propre à namesRepository que l’on a créé dans cette classe, mais qui est privé donc que l’on ne peut pas appeler directement. On avance dans la boucle for avec hasNext().

String name = (String)iter.next();

System.out.println(“Name : “ + name);