ENCART CAMÉRA

Patrons de conception
Comportementaux

Sébastien Mosser - INF5153
Chapitre 7 - Capsule 3
Automne 2020

ace

ENCART CAMÉRA

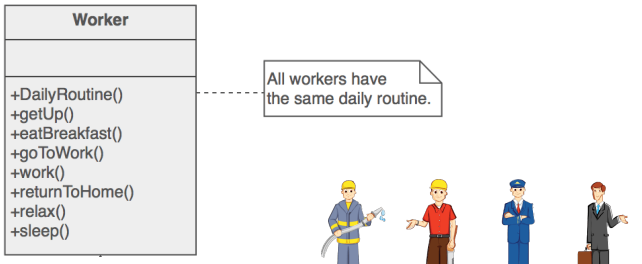
Template Method

ENCART CAMÉRA

Problème

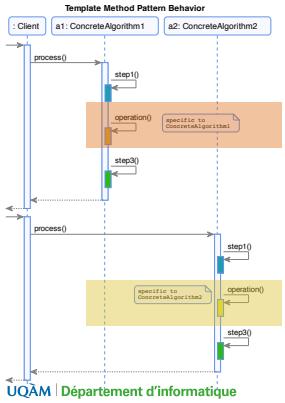
Comment **gérer un algorithme général** mais dont on pourra **spécialiser certaines étapes** ?

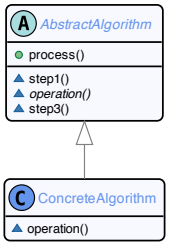
Exemple



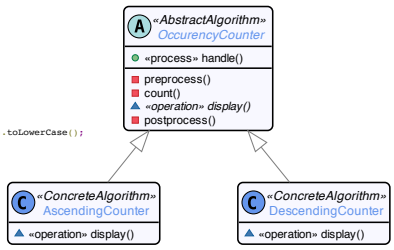
Intention

- Définir un **squelette d'algorithme** dans une opération
  - L'algorithme référence des **étapes bien définies**
- Permettre par héritage de **redéfinir certaines étapes**
  - **Modifier l'algorithme** sans modifier sa **structure générale**





```java
public abstract class OccurencyCounter {

  private String data;

  protected Map<String, Integer> counter;

  public OccurencyCounter(String input) {
    this.data = input;
    this.counter = new HashMap<>();
  }

  public void handle() {
    preprocess();
    count();
    display();
    postprocess();
  }

  private void preprocess() {
    System.out.println("Preprocessing data");
    this.data = this.data.replaceAll("[^a-zA-Z ]", " ").toLowerCase();
  }

  private void count() {
    System.out.println("Counting word occurencies");
    for(String w: data.split("\\s+")) {
      counter.put(w,counter.getOrDefault(w,0)+1);
    }
  }

  protected abstract void display();

  private void postprocess() {
    System.out.println("Found ["+counter.size()+"] distinct words");
```




```java
public class AscendingCounter extends OccurencyCounter {
    public AscendingCounter(String input) { super(input); }

    @Override protected void display() {
        this.counter.entrySet().stream()
                .sorted(comparingByValue())
                .forEach((e) -> System.out.println("   " + e));
    }
}
```

```java
System.out.println("\n# Counting word occurencies using an ASC counter");
OccurencyCounter asc = new AscendingCounter(text);
asc.handle();

System.out.println("\n# Counting word occurencies using a DSC counter");
OccurencyCounter dsc = new DescendingCounter(text);
dsc.handle();
```


Conséquences

- **Réutilisation** du code
- Structure de **contrôle inversée**
- La **visibilité** permet de jouer sur les **capacités d'extension**
- Mais …
  - **Repose sur des sous-classes** (cf stratégie)

Où le rencontrer dans la "vraie-vie"(c) ?

ENCART CAMÉRA

- Canevas de tests unitaire
- Dans l'API Java :
  - Entrées / Sorties (InputStream, Writer, …)
  - Structure de données abstraites (AbstractList, …)
  - Gestion du protocole HTTP (HTTPServlet.doXXX())

ENCART CAMÉRA

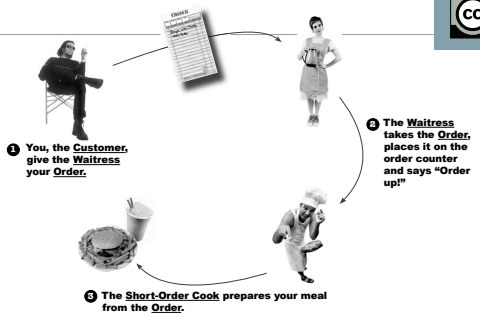
Command

Problème

ENCART CAMÉRA

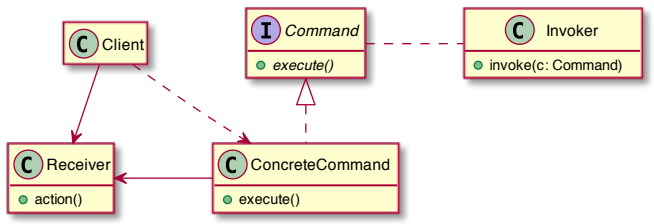
Comment **réaliser** un **traitement**,
sans forcément savoir qui va l'effectuer ?

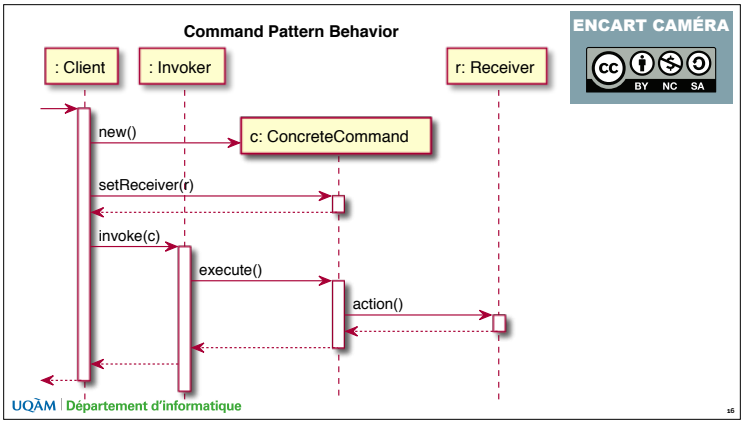
Exemple



Intention

- **Encapsuler une requête** comme un objet
- **Découpler émission** de la requête et **exécution**
- Permettre de **défaire** les traitements effectués
- Gérer des **files d'attentes** ou de priorités





Utilisation dans le code


```
public class Light{
  private boolean on;
  public void switchOn(){ on = true; }
  public void switchOff(){ on = false; }
}
```

```
public interface Command{
  public void execute();
}
```

```
public class LightOnCommand implements Command{
  Light light;

  public LightOnCommand(Light light){
    this.light = light;
  }

  public void execute(){
    light.switchOn();
  }
}
```



```
public class Client{

  public static void main(String[] args)    {

    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightsOnCommand(light);
    Command lightsOff = new LightsOffCommand(light);

    //switch on
    control.setCommand(lightsOn);
    control.pressButton();

    //switch off
    control.setCommand(lightsOff);
    control.pressButton();
  }
}
```

```
public class RemoteControl{

  private Command command;

  public void setCommand(Command command){
    this.command = command;
  }

  public void pressButton(){
    command.execute();
  }

}
```

Conséquences

ENCART CAMÉRA

- **Découplage** entre invocation et réalisation
- **Une action est un objet** comme les autres
- Possibilité de faire des "**Macros-commandes**"
  - *En couplant avec le patron Composite*
- **Ouvert/Fermé** sur l'évolution des commandes disponibles

Où le rencontrer dans la "vraie-vie"(c) ?

ENCART CAMÉRA

- Protocole de communication (e.g., réseau, inter-objets)
- Architectures événementielles
  - Micro-services, bus de messages répartis
- Interfaces graphique
  - Do / Undo
- Système de gestion de versions

ENCART CAMÉRA

Observer

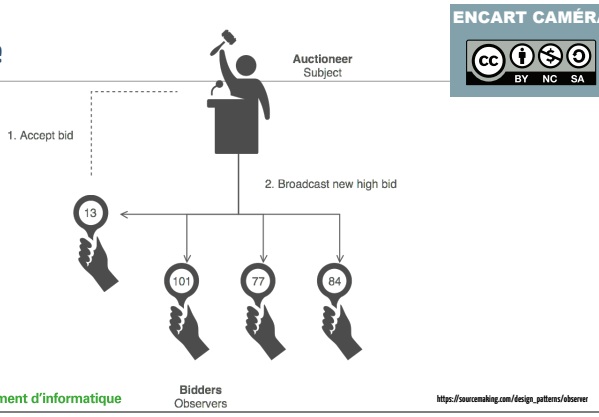
Problème

ENCART CAMÉRA

- Une application a deux **aspects co-dépendant**
- Un **changement sur un objet** impose de **modifier les autres**
- On ne sait pas à l'avance **combien d'objets** sont impactés
  - *Les objets en question ne peuvent être fortement couplés*

Exemple

ENCART CAMÉRA

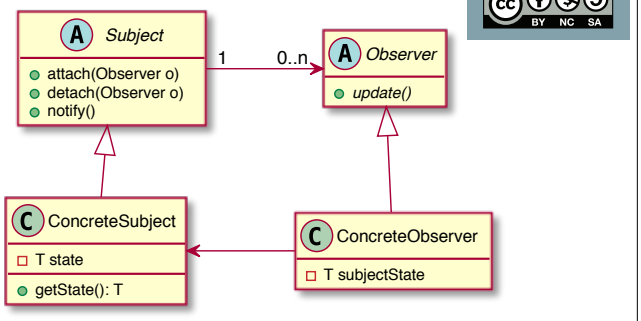


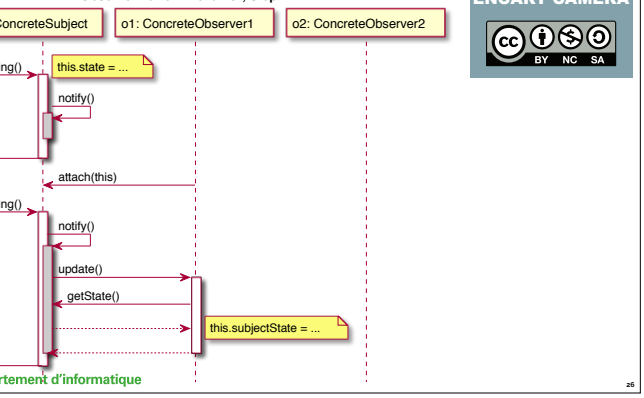
https://sourcemaking.com/design_patterns/observer

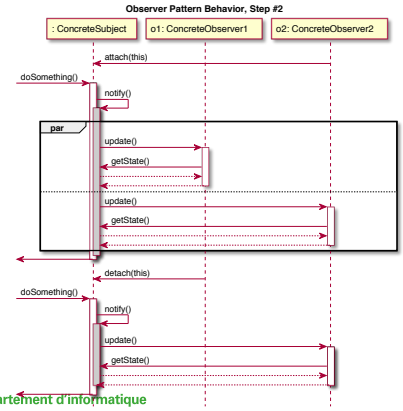
Intention

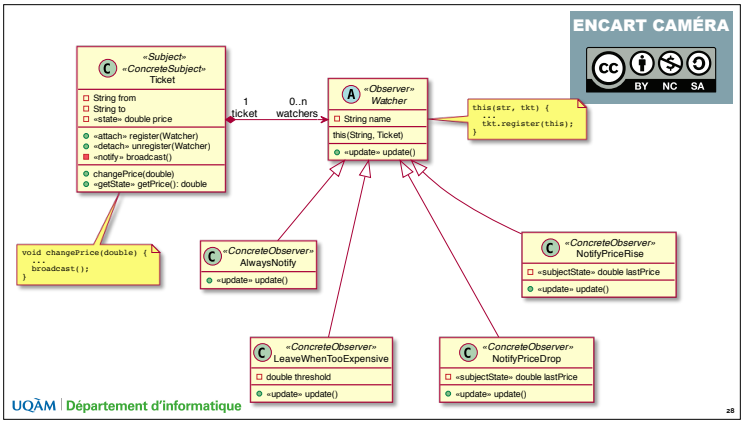
ENCART CAMÉRA

Définir une relation "1-n" de telle façon que si on change l'objet unique d'état, tout ses dépendants sont prévenus et mis à jour automatiquement.









Utilisation dans le code

```
public class Ticket {

  private String from;
  private String to;
  private double price;

  public Ticket(String from, String to, double price) {
    // ...
    this.watchers = new HashSet<>();
  }

  public void changePrice(double percentage) {
    // ...
    broadcast();
  }

  public double getPrice() { }

  private Set<Watcher> watchers;
  public void register(Watcher w) {
    this.watchers.add(w);
  }

  public void unregister(Watcher w) {
    this.watchers.remove(w);
  }

  private void broadcast() {
    Set<Watcher> targets = new HashSet<>(this.watchers);
    targets.parallelStream().forEach(Watcher::update);
  }
}
```

La mise en oeuvre de ce patron dans le code est très invasive !


```
public class LeaveWhenTooExpensive extends Watcher {

  private double threshold;

  public LeaveWhenTooExpensive(String n, Ticket t, double percentage) {
    super(n,t);
    this.threshold = t.getPrice() * percentage;
  }

  @Override
  public void update() {
    super.update();
    if(this.ticket.getPrice() < threshold)
      System.out.println("  ->> Sending email notification, price is still correct");
    else {
      System.out.println("  ->> too expensive, not interested anymore");
      this.ticket.unregister(this);
    }
  }
}
```

Conséquences

ENCART CAMÉRA

- **Variations abstraites** entre sujets et Observateurs
- **Absence de couplage** (*géré au niveau méta*)
- "**Broadcast**" des communications avec les observateurs

Où le rencontrer dans la "vraie-vie"(c) ?

ENCART CAMÉRA

- API Java (depuis JDK 1.0)
  - Interface **Observer**, Class **Observable**
- Principe classique en Interaction Personne-Machine
  - L'activation d'un bouton déclenche des comportements
- C'est une simplification "objet" des architectures Pub/Sub
  - Micro-services et chorégraphies d'événements

ENCART CAMÉRA

State

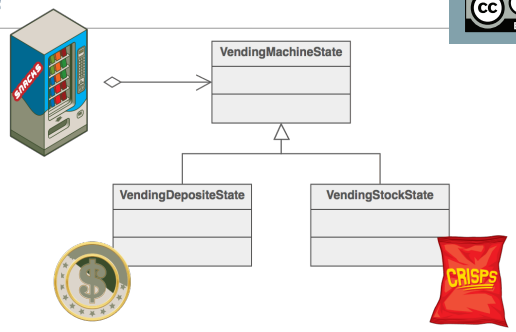
Problème

ENCART CAMÉRA

Comment **modifier le comportement** d'un objet quand son **état interne change**, et ainsi **obtenir des traitements différents** ?

Exemple

ENCART CAMÉRA

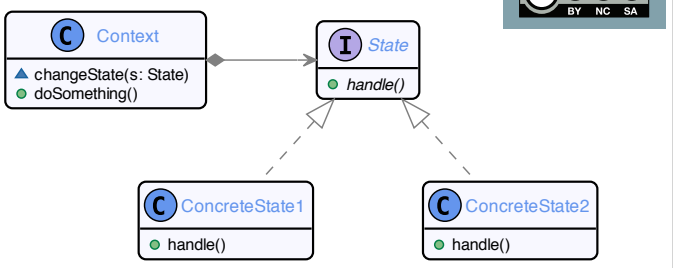


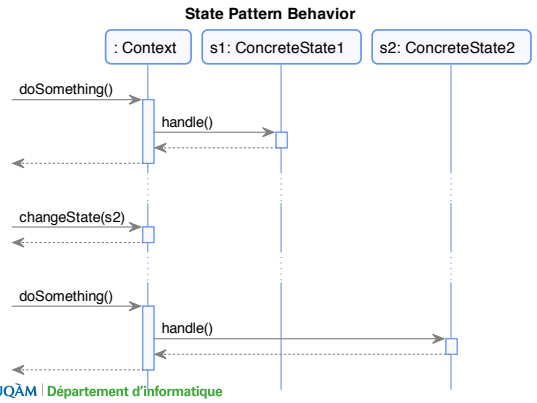
[sourcemaking]

Intention

ENCART CAMÉRA

- **Eviter les instructions conditionnelles** en masse
- **Simplifier l'ajout** ou la suppression de nouveaux états
- Donner l'**impression** que l'objet a été **modifié**
  - alors que c'est **uniquement sont état** qui a varié





Conséquences

- **Séparation des comportements** relatifs à chaque état
- **Transitions explicites** par action sur l'objet
- **Transparence** pour l'appelant

Où le rencontrer dans la "vraie-vie"(c) ?

ENCART CAMÉRA

- Gestion des connexion réseaux
- Mise en oeuvre d'une machine à état
- Javax Lifecycle

ENCART CAMÉRA

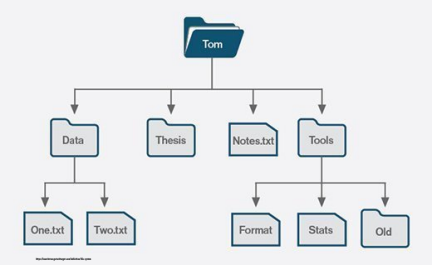
Visitor

Problème

ENCART CAMÉRA

Comment **rendre externe** à **une hiérarchie d'objets** la **mise en oeuvre d'un comportement**, pour pouvoir **changer celui-ci sans avoir à changer les objets** ?

Exemple



Rechercher un fichier ?

Supprimer un répertoire ?

Copier un répertoire ?

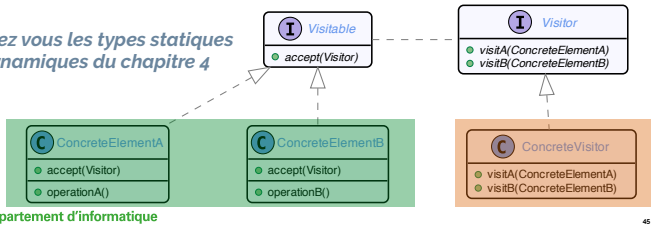
Intention

- Définir une **abstraction pour les opérations** à effectuer
- **Définir UNE FOIS POUR TOUTES le parcours** de la hiérarchie
  - *Il existe des variantes où on peut laisser ça au développeur*
- "**Lancer**" une opération sur une hiérarchie,
  - **Sans avoir a modifier la hiérarchie** pour créer de nouvelles opérations

Principe du "double dispatch"

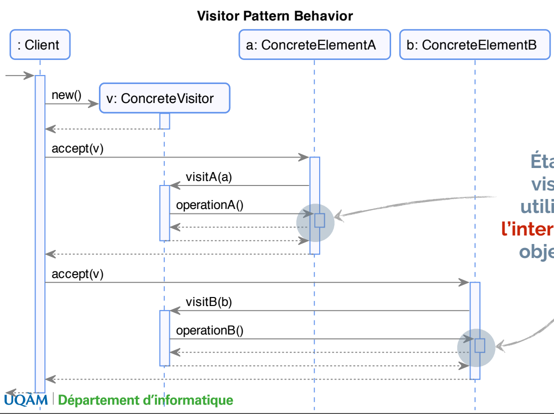
- La **hiérarchie** "accepte" l'opération
- Elle délègue au **visiteur** l'exécution de l'opération

Rappelez vous les types statiques et dynamiques du chapitre 4

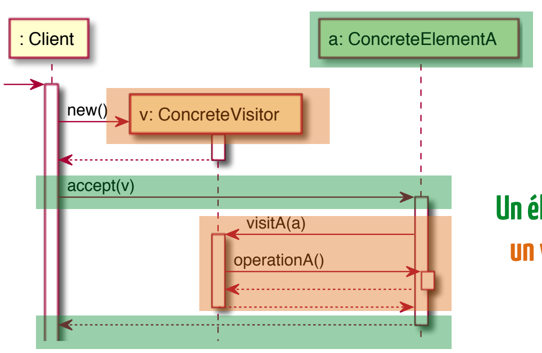


Définition du “Double Dispatch”

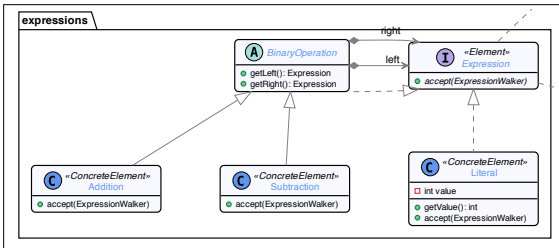
> *Double Dispatch is the **selection of method** based on the **runtime types** of two objects — the **receiver** and the **argument**.*



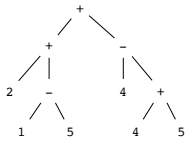
Étant externe, les visiteurs peuvent utiliser **uniquement l'interface publique** des objets qu'ils visitent



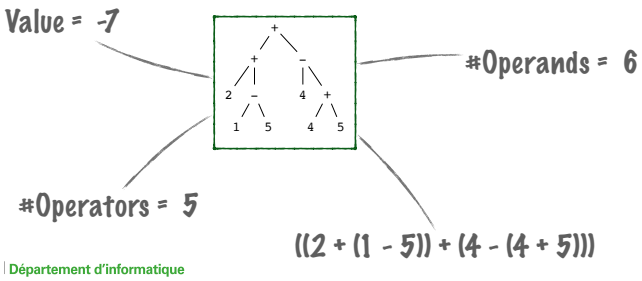
Un élément accepte, un visiteur visite.




```
Expression e =
        new Addition(
                new Addition(
                        new Literal(2),
                        new Subtraction(
                                new Literal(1),
                                new Literal(5))),
                new Subtraction(
                        new Literal(4),
                        new Addition(
                                new Literal(4),
                                new Literal(5))));
```

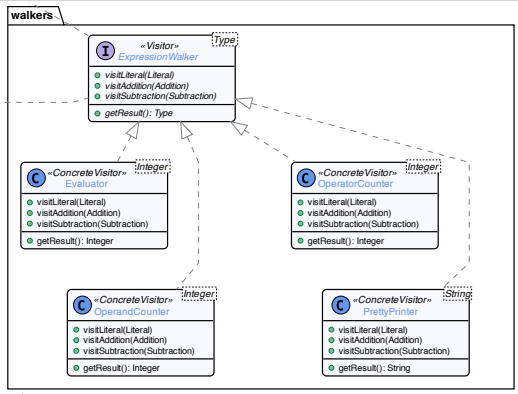


A objets égaux, traitement différents



Utilisation dans le code


```
PrettyPrinter printer = new PrettyPrinter();
e.accept(printer);
System.out.println("Expression: " + printer.getResult());

OperandCounter opc = new OperandCounter();
e.accept(opc);
System.out.println("  #Operands =  " + opc.getResult());

OperatorCounter ops = new OperatorCounter();
e.accept(ops);
System.out.println("  #Operators =  " + ops.getResult());

Evaluator eval = new Evaluator();
e.accept(eval);
System.out.println("Value =  " + eval.getResult());
```




```java
public class PrettyPrinter implements ExpressionWalker<String> {

    private StringBuffer buffer = new StringBuffer();

    @Override
    public void visitLiteral(Literal literal) {
        buffer.append(literal.getValue());
    }

    @Override
    public void visitAddition(Addition addition) {
        buffer.append("(");
        addition.getLeft().accept(this);
        buffer.append(" + ");
        addition.getRight().accept(this);
        buffer.append(")");
    }

    @Override
    public void visitSubtraction(Subtraction subtraction) {
        buffer.append("(");
        subtraction.getLeft().accept(this);
        buffer.append(" - ");
        subtraction.getRight().accept(this);
        buffer.append(")");
    }

    @Override
    public String getResult() { return buffer.toString(); }
}
```

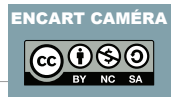
On repasse par le "double dispatch" lorsqu'un élément visité doit en visiter d'autres

Conséquences

- On peut définir **autant de visiteurs différents** qu'on le souhaite
  - Et on peut le faire avec une **approche fonctionnelle**
    - (Donc un peu moins 🐷 que cet exemple, sans effets de bord)
- L'**introduction du visiteur est invasive** dans la hiérarchie
- Le double-dispatch a un **coût non négligeable à l'exécution**
  - *Et c'est même considéré par certains comme une mauvaise pratique*
  - *Mais on peut faire mieux avec du pre-processing (p.-ex. pré-cabler les visites)*

Où le rencontrer dans la "vraie-vie"(c) ?

ENCART CAMÉRA

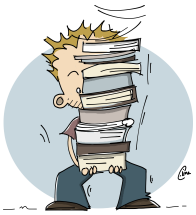


- Compilation
  - On visite des arbres de syntaxes abstraits
- Transformation de modèles
  - Norme QVT, langage ATL, …
- Langage intégrant le "pattern-matching"

UQÀM | Département d'informatique

FACULTÉ DES SCIENCES
Université du Québec à Montréal

ENCART CAMÉRA



https://mosser.github.io/

https://ace-design.github.io/

***Abonne toi** à la chaine,*
et met un pouce bleu !