

QUALITÉ DU SI

Sommaire [Télécharger en version PDF](#)

[Intro](#)

[PBT](#)

[Quête des machines états](#)

[Gestion d'erreurs & qualité du code](#)

[Frontends dataflow](#)

[Cloud & microservices](#)

[Communication Inter-process](#)

[Decentralized web](#)

[Final Boss](#)



QUALITÉ DU SI

MIAGE - INTRODUCTION

DISCLAIMER 1

VOTRE DIPLÔME EST GLOBAL, LES UE NE SONT PAS DES SILOS

Ce cours a pour pré-requis vos cours de L3 :

- Compréhension typage statique / dynamique
- Savoir lire une stack d'erreur
- Savoir lire une doc d'API

Ce cours a pour pré-requis vos cours de M1/M2 :

- SGBDR et NoSQL (NDD)
- Serialization / Deserialization (ALOM, ARI)
- Single Page Application (ARI)
- Programmation async monothreadée aka fiber aka green thread aka light thread aka event loop aka coroutine (ARI)
- CORS et CSP (CAR, ARI, ALOM, SSI)
- API REST/JSON (ALOM, ARI)
- Management de projet (MP)
- Anglais

DISCLAIMER 2

ON ATTEND DE DIPLÔMÉS M2 D'APPORTER DE NOUVELLES CONNAISSANCES DANS L'ENTREPRISE

Ce cours va vous dérouter par rapport à ce que vous avez vu dans vos cours de programmation

Ce n'est pas contradictoire

Ce cours vise à vous ouvrir à des connaissances de programmation moderne

Ce cours vous aidera à structurer vos futurs apprentissages

Ce cours doit vous amener à raisonner au niveau d'un SI, mais on va repartir des bases depuis le logiciel

DISCLAIMER 3

ÇA VA ÊTRE DENSE MAIS GUIDÉ

A condition que vous travaillez au fil de l'eau, vous n'aurez pas l'occasion de rattraper du retard pris

Cours en FR pour assurer le bonne compréhensions

TP en EN pour donner du vocabulaire et faciliter les recherches

7 Cours, incluant 4 TP, puis un projet de Système d'information

Vous avez accès à la première source d'information du monde : INTERNET

-> Vous ne trouverez pas la réponse copier-coller !

-> Mais vous avez accès au code source, docs d'API et communautés open source

Je réponds aux sollicitations entre 2 cours si vous posez des questions !

INTRO

POURQUOI LA QUALITÉ EST UN ENJEU



Le SI est un actif valorisable de l'entreprise



Le SI de qualité procure un avantage compétitif



Le SI de qualité améliore la satisfaction client



Le SI de qualité améliore la satisfaction au travail



Le SI doit répondre aux besoins fonctionnels et non fonctionnels

INTRO

LES DIMENSIONS DE LA QUALITÉ



Infrastructure : matériel, réseau, OS, ... *(du baremetal au cloud)*



Logiciel : applications construites et maintenues



Données : données du SI (SQL / NoSQL) ✓ *NDD*



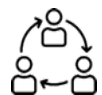
Information : communications inter-applicative



Administrative : qualité de la fonction SI, incluant les processus d'élaboration du budget et d'élaboration du planning










Service : valeur du service rendu « perçue » par le client



RH : organisation des équipes SI








INTRO

LES DIMENSIONS DE LA QUALITÉ

-  **Infrastructure** : matériel, réseau, OS, ... *(du baremetal au cloud)*
-  **Logiciel** : applications construites et maintenues
-  **Données** : données du SI (SQL / NoSQL) ✓ *NDD*
-  **Information** : communications inter-applicative
-  **Administrative** : qualité de la fonction SI, incluant les processus d'élaboration du budget et d'élaboration du planning ✓
-  **Service** : valeur du service rendu « perçue » par le client ✓
-  **RH** : organisation des équipes SI ✓ *Management de projets*

INTRO

LES DIMENSIONS DE LA QUALITÉ

-  **Infrastructure** : matériel, réseau, OS, ... *(du baremetal au cloud)*
 -  **Logiciel** : applications construites et maintenues
 -  **Données** : données du SI (SQL / NoSQL) ✓ *NDD*
 -  **Information** : communications inter-applicative
 -  **Administrative** : qualité de la fonction SI, incluant les processus d'élaboration du budget et d'élaboration du planning ✓
 -  **Service** : valeur du service rendu « perçue » par le client ✓
 -  **RH** : organisation des équipes SI ✓ *Management de projets*
- Architecture du SI*

INTRO

La qualité optimale

Attentes : besoins exprimés par le client final (utilisateur)

Spécification : traduction du besoin utilisateur (MOA)

Réalisation : mise en oeuvre du service (MOE)



LE TRILÈME

L'impact de la gestion de projet



LE TRILÈME

L'impact de la gestion de projet

Les approches orientées « cycle en V » mettent l'accent sur l'axe spécification, dont vont découler les 2 autres

Les approches agiles mettent l'accent à l'origine sur la attentes↔réalisation (XP)

... mais ont dévié dans leurs versions récentes sur l'axe attentes↔spécification (scrum, safe)

... ce qui fit émerger le mouvement software craftsmanship sur l'axe réalisation↔attentes

Les approches R&D mettent souvent l'accent sur l'axe réalisation seul (innovations de rupture)



INTRO

ASSURANCE « QUALITÉ »

La qualité est souvent présentée par une approche systémique orientée processus : l'assurance qualité

Vous manipulez peut être ces systèmes en entreprise : ISO, CMMi, ITIL, Safe, ...

Ces systèmes sont une partie de la dimension administrative, insuffisante prise seule

Dans un contexte de numérisation accélérée et totale de l'économie, les dimension Architecture et Management doivent être menées de concert

La dimension architecture est critique pour obtenir un avantage stratégique

CALENDRIER 2023

PRÉVISIONNEL

3/01: Intro & Property based Testing - Troll of Fame Kata (Typescript)

10/01: Quête des machines à états - Tennis Kata (Typescript)

17/01: Frontend dataflow - Front Kata (React)

24/01: Intro Blockchain & Smart contracts - Contract Kata (JsLigo)

31/01: Gestion des Erreurs & Présentation du projet

7/02 : Pas cours

21/02: Cloud et microservice

28/02: Communication Inter-process

QUALITÉ DU SI



COURS 1 - PBT

THE QUEST

PBT



THE QUEST

PBT

Quand on reprend un projet, qu'on accueille un nouveau développeur, qu'on revient sur un code d'il y a 6 mois, qu'on effectue une migration technique, ... : comment avoir une documentation à jour.

Les écrits ne sont JAMAIS à jours

Comment documenter les règles de gestion d'un projet?



CODE IS LAW

Code as specification ou spécification détaillée?

1. Vérifier les invariants : preuves > types > tests
2. Faciliter la compréhension du code : idiomes, compétences
3. Uniformiser les pratiques : design patterns, guidelines
4. Améliorer la communication tech/business : Ubiquitous language, spécifications, UML
5. Améliorer l'expérience utilisateur : UX design, user doc, vidéos

Moins bonne UX / Meilleure durabilité

Impact User Experience

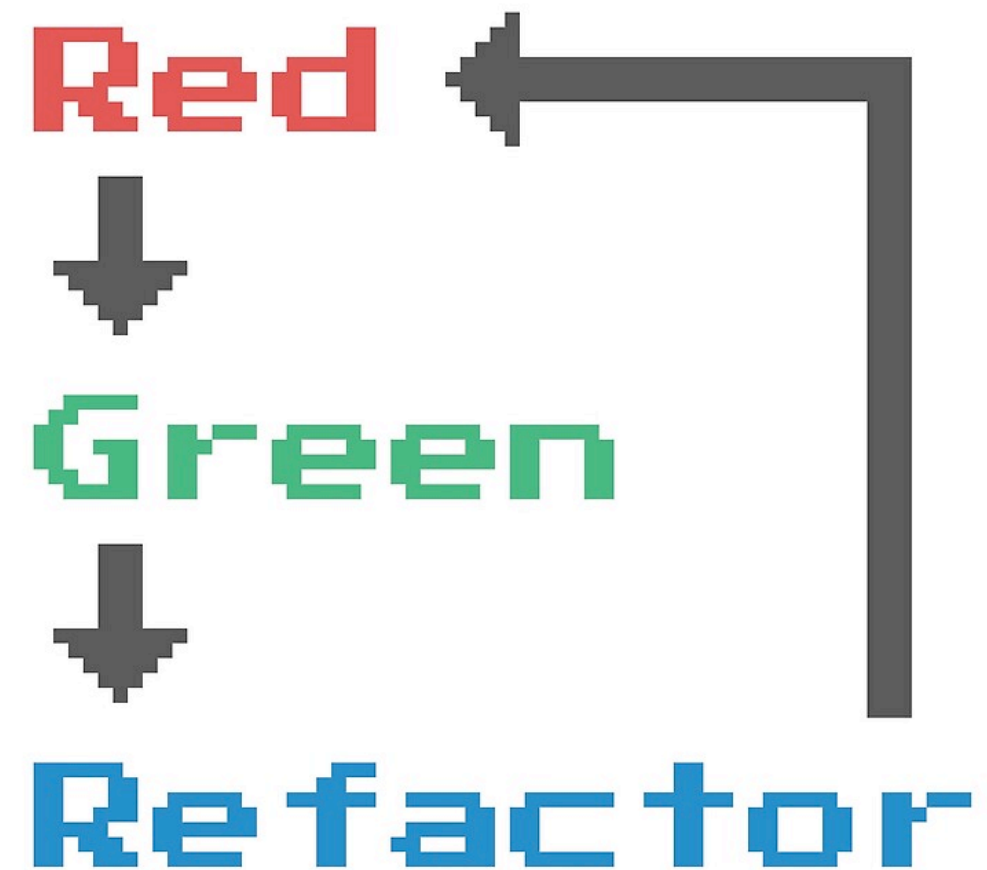
Impact Durabilité

Meilleures UX / Moins durabilité

TEST DRIVEN DEVELOPMENT

RED GREEN REFACTOR

1. Réception d'une demande de fonctionnalité
2. Écrire un test qui échoue
3. Écrire le code, jusqu'à ce que le test passe
4. Nettoyer le code
5. Itérer



CARACTÉRISTIQUE D'UN TEST

TEST UNITAIRES

Isolé: pas de dépendance à un composant externe (Base de données, système de fichier, ...)

Reproductible: retourne toujours le même résultat si vous ne modifiez rien entre les exécutions

Temps opportun: l'écriture d'un test doit prendre peu de temps au regard du code testé, si vous avez du mal à écrire un test, vous avez probablement un problème de couplage fort dans votre logiciel

Documentation exécutable: un test utile ne se contente pas de tester du code, il fournit une documentation à l'équipe technique. C'est ce qui doit aider à définir le bon niveau test, on évite les micro comme les méga tests

PROPRIÉTÉ ?

Remember last Christmas !

Quand a été fêté Noël en 2022 ?



PROPRIÉTÉ ?

Remember last Christmas !

Quand a été fêté Noël en 2022 ?

Quand a été fêté Noël en 2020 ?



PROPRIÉTÉ ?

Remember last Christmas !

Quand a été fêté Noël en 2022 ?

Quand a été fêté Noël en 2020 ?

Quand a été fêté Noël en 1990 ?



PROPRIÉTÉ ?

Remember last Christmas !

Quand a été fêté Noël en 2022 ?

Quand a été fêté Noël en 2020 ?

Quand a été fêté Noël en 1990 ?



Le 25 décembre est une PROPRIÉTÉ qui définit « Noël »

LES TESTS

DU TU AU PBT

Unit Tests

Fixed input

One execution

Assert result

Property Based Tests

Random input

Many executions

Assert result or behavior

TAKE AWAY

NOUS AVONS VU

Les tests sont une documentation à jour

Les TU testent des cas particulier et peuvent être généralisé par des tests d'invariants

Les tests de propriétés permettent de valider les règles business et de découvrir des bugs qui passeraient à la trappe des TU classiques



TROLL OF FAME KATA

LE ROI DES TROLLS,
GNONPOM, A CODÉ LE
TROLL OF FAME: UNE
APPLICATION
FABULEUSE QUI AIDE LES
TROLLS À APPRENDRE LES
NOMBRES QUAND ILS
CHASSENT.



GNONPOM ÉTAIT UN ROI
DÉVELOPPEUR, FÉRU DE
TEST DRIVEN
DÉVELOPPEMENT. IL A MIS
EN PRODUCTION TOF
QUAND TOUS LES TESTS
ÉTAIENT VERTS.

MALHEUREUSEMENT, IL A ÉTÉ ABATTU PAR UN HORRIBLE ELFE.

VIVE LE NOUVEAU ROI, VIVE LE TROLL AKLASS!

CETTE FOIS C'EST DÉCIDÉ LE TOURNOI DE CHASSE À L'ELFE EST LANCÉ !

À LA FIN DE CHAQUE BATAILLE, LES TROLLS VEULENT COMPARER LES NOMBRES ET
ATTRIBUTS DE CES ELFES DÉGOÛTANTS.

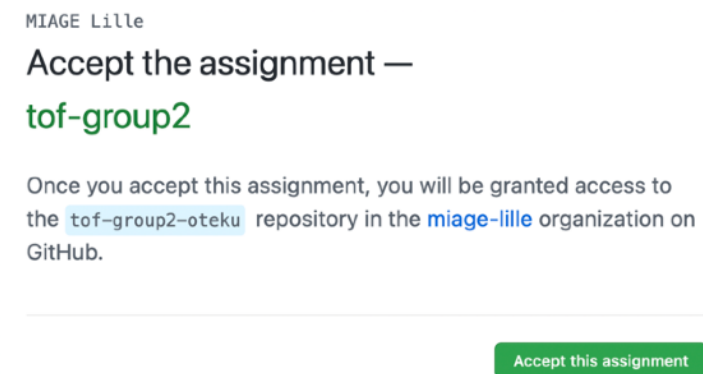
AVEC TOF ÇA DEVRAIT ÊTRE FACILE ... CA DEVRAIT.

TRAVAUX PRATIQUES

DÉBUTER UN TP

Les TP sont gérés avec Github Classroom

Cliquez sur le lien fourni et acceptez



Cela crée un repository privé personnel dans le groupe miage-lille

⚠ Vous avez jusqu'au mercredi suivant pour terminer

⚠ 1 étudiant = 1 TP = 1 repo

TRAVAUX PRATIQUES

ENVIRONNEMENT DE DÉVELOPPEMENT

Tous les TPs peuvent être réalisés:

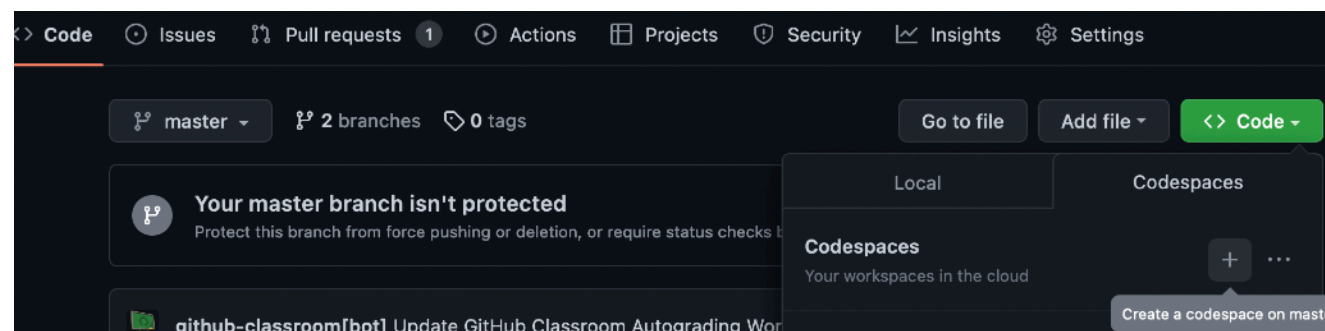
Via vscode + docker plugin (nécessite docker en local)

Via un webIDE:

Gitpod (possible de connecter un vscode local avec gitpod plugin)

ou

GitHub codespace



⚠ Tout autre choix est sous votre responsabilité!

TRAVAUX PRATIQUES

LIENS

GROUPE 1 (Thomas) <https://classroom.github.com/a/1F9SU84Y>

GROUPE 2 (Quentin) <https://classroom.github.com/a/ngKNSmhq>

THE BOSS

SI VOUS VOULEZ ALLER PLUS LOIN

Renforcement :

🎥 Types VS Tests

🎥 Much Ado About Testing



QUALITÉ DU SI



COURS 2 - LA QUÊTE DES MACHINES ÉTATS

THE QUEST

La plupart des bugs (i.e., dégâts financiers pour mon entreprise) rencontrés en prod (dans ma vie) sont liés à :

des MACHINES ÉTATS implicites

du code qui crash au RUNTIME

des EFFETS non maîtrisés

... et j'aime pas ça !!!

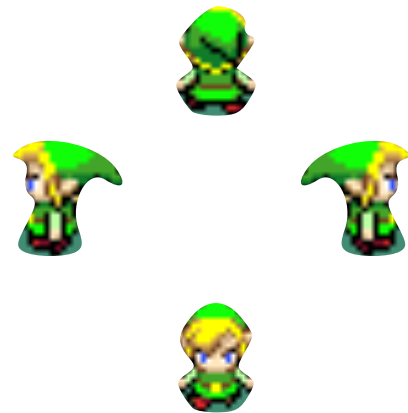
Peut-on améliorer la conception logicielle,
puis SI pour éviter cela ?

OUI c'est le but de ce cours



LINK TO THE PATH

IMPLÉMENTATION NAÏVE EN TYPESCRIPT



```
enum Direction {
  North,
  East,
  South,
  West
}

const label = (d : Direction) : string => {
  switch (d){
    case Direction.North: return "north";
    case Direction.South: return "south";
    case Direction.East: return "east";
    case Direction.West: return "west";
  }
}

console.log(label(1));
console.log(label(Direction.East));
console.log(label(Direction.South));
console.log(label(4));
```


LINK TO THE PATH

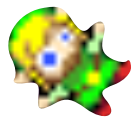
IMPLÉMENTATION NAÏVE EN TS



```
console.log(label(-35));
```

LINK TO THE PATH

IMPLÉMENTATION NAÏVE EN TS



```
console.log(label(-35));
```

No ERROR

Undefined

LINK TO THE PATH

IMPLÉMENTATION NAÏVE EN TS



```
console.log(label(-35));
```

No ERROR

Undefined

-35 n'est pas une Direction valide ... mais les valeurs d'un enum sont en réalité des numbers

LINK TO THE PATH

IMPLÉMENTATION NAÏVE EN TS



```
console.log(label(-35));
```

No ERROR

Undefined

-35 n'est pas une Direction valide ... mais les valeurs d'un enum sont en réalité des numbers
Direction.North = 0 ... Direction.West = 3

LINK TO THE PATH

IMPLÉMENTATION NAÏVE EN TS



```
console.log(label(-35));
```

No ERROR

Undefined

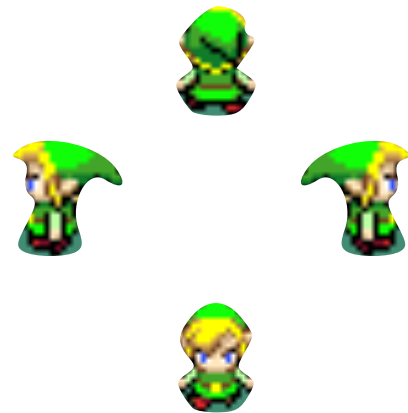
-35 n'est pas une Direction valide ... mais les valeurs d'un enum sont en réalité des numbers

Direction.North = 0 ... Direction.West = 3

Vu du système de type les enums sont des numbers 🍷

LINK TO THE PATH

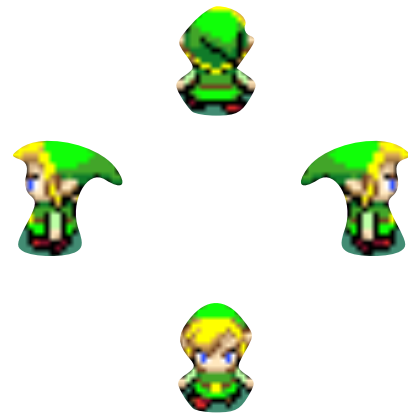
FAISONS CRASHER LE RUNTIME?



```
const label = (d : Direction) : string => {  
  switch (d){  
    case Direction.North: return "north";  
    case Direction.South: return "south";  
    case Direction.East: return "east";  
    case Direction.West: return "west";  
  }  
  throw new Error("Not a valid Direction")  
}
```


LINK TO THE PATH

FAISONS CRASHER LE RUNTIME?



```
const label = (d : Direction) : string => {  
  switch (d){  
    case Direction.North: return "north";  
    case Direction.South: return "south";  
    case Direction.East: return "east";  
    case Direction.West: return "west";  
  }  
  throw new Error("Not a valid Direction")  
}
```

Compilation ERROR

Unreachable code detected.

LINK TO THE PATH

FAISONS CRASHER LE RUNTIME?



```
const label = (d : Direction) : string => {  
  switch (d){  
    case Direction.North: return "north";  
    case Direction.South: return "south";  
    case Direction.East: return "east";  
    case Direction.West: return "west";  
  }  
  throw new Error("Not a valid Direction")  
}
```

Compilation ERROR

Unreachable code detected.

TS croit qu'on traite tous les cas ... Les enums sont dangereux!

LINK TO THE PATH

LES UNIONS



```
type Direction = "North" | "East" | "South" | "West";

const label = (d : Direction) : string => {
  switch (d ){
    case "North": return "north";
    case "East": return "east";
    case "South": return "south";
    case "West": return "west";
  }
}
```

LINK TO THE PATH

LES UNIONS



```
type Direction = "North" | "East" | "South" | "West";

const label = (d : Direction) : string => {
  switch (d ) {
    case "North": return "north";
    case "East": return "east";
    case "South": return "south";
    case "West": return "west";
  }
}
```

"North", "East", "South", "West" sont des types! (String literal type)

LINK TO THE PATH

LES UNIONS



```
console.log(label(-35));
```

LINK TO THE PATH

LES UNIONS



```
console.log(label(-35));
```

Compilation ERROR

Argument of type '-35' is not assignable to parameter of type 'Direction'.

LINK TO THE PATH

LES UNIONS



```
console.log(label(-35));
```

Compilation ERROR

Argument of type '-35' is not assignable to parameter of type 'Direction'.

On peut valider dès la compilation que l'on utilise des valeurs valides !!!

LINK TO THE PATH

LES UNIONS



```
const label = (d : Direction) : string => {  
  switch (d ){  
    case "North": return "north";  
    case "East": return "east";  
    case "South": return "south";  
    //case "West": return "west";  
  }  
}
```

LINK TO THE PATH

LES UNIONS



```
const label = (d : Direction) : string => {  
  switch (d ){  
    case "North": return "north";  
    case "East": return "east";  
    case "South": return "south";  
    //case "West": return "west";  
  }  
}
```

Compilation Error

Function lacks ending return statement and return type does not include 'undefined'.

LINK TO THE PATH

LES UNIONS



```
const label = (d : Direction) : string => {  
  switch (d ){  
    case "North": return "north";  
    case "East": return "east";  
    case "South": return "south";  
    //case "West": return "west";  
  }  
}
```

Compilation Error

Function lacks ending return statement and return type does not include 'undefined'.

On peut valider dès la compilation qu'on traite TOUTES LES valeurs valides !!!

LINK TO THE PATH

ET EN JAVA? JAVA 15 (SEALED), JAVA 14 (RECORD) ET JAVA 17 (EXHAUSTIVITÉ)

```
sealed interface Direction {  
    record North() implements Direction {}  
    record East() implements Direction {}  
    record South() implements Direction {}  
    record West() implements Direction {}  
}  
public static final String label(Direction d) {  
    return switch(d){  
        case Direction.North n -> "north";  
        case Direction.East e -> "east";  
        case Direction.South s -> "south";  
        case Direction.West w -> "west";  
    }  
}
```

LINK TO THE PATH

ET EN JAVA? JAVA 15 (SEALED), JAVA 14 (RECORD) ET JAVA 17 (EXHAUSTIVITÉ)

```
sealed interface Direction {  
    record North() implements Direction {}  
    record East() implements Direction {}  
    record South() implements Direction {}  
    record West() implements Direction {}  
}  
public static final String label(Direction d) {  
    return switch(d){  
        case Direction.North n -> "north";  
        case Direction.East e -> "east";  
        case Direction.South s -> "south";  
        case Direction.West w -> "west";  
    }  
}
```

Enfin des interfaces scellées + record permettent de décrire correctement un type « OU »

Finalement JEP 406 <http://openjdk.java.net/jeps/8213076> Java 17 (Septembre 2021) apporte le « switch/case » pour l'exhaustivité d'un pattern matching, ainsi que c'est un nouveau « switch/case » expressif (parce que l'instruction switch java est quand même bien pourri)

Puis JEP 420 Java18 pour une amélioration des pattern de déconstruction au sein d'un switch/case expressif

Modèle similaire à Kotlin avec 10 ans de retard ou Scala avec 15 ans de retard ...

Mettre en prod un projet JAVA < 18 est une faute professionnelle 🤔

TAKE AWAY

AVEC UN LANGAGE STATIQUEMENT (BIEN) TYPÉ

On peut valider qu'on traite uniquement les valeurs valides

On peut valider qu'on traite toutes les valeurs valides

Dès la compilation

Dans les langages ML : OCaml, Haskell, F#, ...

Mais aussi les langages modernes qui s'en inspirent : TS, Scala, Kotlin, Swift, Rust, C++17, ...



LA PLUPART DES APP DE GESTION SONT DES MACHINES ÉTATS

LES TRANSITIONS DE LINK



LA PLUPART DES APP DE GESTION SONT DES MACHINES ÉTATS

LES TRANSITIONS DE LINK



LA PLUPART DES APP DE GESTION SONT DES MACHINES ÉTATS

LES TRANSITIONS DE LINK



LA PLUPART DES APP DE GESTION SONT DES MACHINES ÉTATS

LES TRANSITIONS DE LINK



LA PLUPART DES APP DE GESTION SONT DES MACHINES ÉTATS

LES TRANSITIONS DE LINK



Link regarde dans une direction, avance, s'arrête, etc...

MODÉLISER LA COMMANDE

UN TYPE « ET »

```
type Direction = "North" | "East" | "South" | "West";

type Command = {
  order: string,
  direction: Direction
}

const turnEast : Command = {
  order: "face",
  direction: "East"
}
```


MODÉLISER LA COMMANDE

UN TYPE « ET »

```
type Direction = "North" | "East" | "South" | "West";

type Command = {
  order: string,
  direction: Direction
}

const turnEast : Command = {
  order: "face",
  direction: "East"
}
```

Le type command est un order de type string ET une direction de type Direction

MODÉLISER LA COMMANDE

UN TYPE « ET »



```
const yolo : Command = {  
  order: "triple_backflip",  
  direction: "East"  
}
```

MODÉLISER LA COMMANDE

UN TYPE « ET »



```
const yolo : Command = {  
  order: "triple_backflip",  
  direction: "East"  
}
```

MODÉLISER LA COMMANDE

UN TYPE « ET »



```
const yolo : Command = {  
  order: "triple_backflip",  
  direction: "East"  
}
```

Triple Backflip n'est pas un ordre valide

MODÉLISER LA COMMANDE

ON A DÉJÀ VU COMMENT RÉSOUDRE CELA => ORDER : UN TYPE « OU »

```
type Direction = "North" | "East" | "South" | "West";  
  
type Order = "Face" | "Start" | "Stop"  
  
type Command = {  
  order: Order,  
  direction: Direction  
}
```

MODÉLISER LA COMMANDE

ORDER : UN TYPE « OU »



```
const noSense : Command = {  
  order: "Start",  
  direction: "East"  
}
```

MODÉLISER LA COMMANDE

ORDER : UN TYPE « OU »



```
const noSense : Command = {  
  order: "Start",  
  direction: "East"  
}
```

Start East n'est pas une commande valide

MODÉLISER LA COMMANDE

ORDER : UN TYPE « OU » PARAMÉTRÉ = UNION DISCRIMINÉE EN TS

```
type Direction = "North" | "East" | "South" | "West";  
  
type Order = {kind: "Face", direction: Direction} | {kind: "Start"} | {kind: "Stop"}  
  
type Command = {  
  order: Order,  
}
```

Un discriminant est une propriété de type littéral commune à chaque type du type union

MODÉLISER LA COMMANDE

COMMAND : UN TYPE « OU » PARAMÉTRÉ = UNION DISCRIMINÉE EN TS

```
type Direction = "North" | "East" | "South" | "West";  
type Command = {kind: "Face", direction: Direction} | {kind: "Start"} | {kind: "Stop"}
```

Kind est le discriminante du type union Command

MODÉLISER LA COMMANDE

COMMAND : UN TYPE « OU »



MODÉLISER LA COMMANDE

COMMAND : UN TYPE « OU »



```
const faceEast : Command = {kind: "Face", direction: "East"}
```

MODÉLISER LA COMMANDE

COMMAND : UN TYPE « OU »



```
const start : Command = {kind: "Start"}
```

MODÉLISER LA COMMANDE

COMMAND : UN TYPE « OU »



```
const stop_ : Command = {kind: "Stop"}
```

TAKE AWAY

LES TYPES « OU » SONT TRÈS UTILES

Les types « ET » s'appellent aussi record ou types produit

Les classes sont des types « ET »

Les types « OU » s'appellent aussi variants ou types somme



SCRIPT DE COMMANDE

ENCHAINER LES COMMANDES AVEC UN TYPE OU RÉCURSIF



```
type Direction = "North" | "East" | "South" | "West"

type Command =
| {kind: "Face", direction: Direction}
| {kind: "Start"}
| {kind: "Stop"}
| {kind: "Chain", first: Command, second: Command}

const moveEast = {
  kind: "Chain",
  first: {kind: "Face", direction: "East"} ,
  second: {
    kind: "Chain",
    first: {kind: "Start"} ,
    second: {kind: "Stop"}
  }
}
```

SCRIPT DE COMMANDE

ENCHAINER LES COMMANDES AVEC UN TYPE OU RÉCURSIF



```
type Direction = "North" | "East" | "South" | "West"

type Command =
| {kind: "Face", direction: Direction}
| {kind: "Start"}
| {kind: "Stop"}
| {kind: "Chain", first: Command, second: Command}

const moveEast = {
  kind: "Chain",
  first: {kind: "Face", direction: "East"} ,
  second: {
    kind: "Chain",
    first: {kind: "Start"} ,
    second: {kind: "Stop"}
  }
}
```

SCRIPT DE COMMANDE

ENCHAINER LES COMMANDES AVEC UN TYPE OU RÉCURSIF



```
type Direction = "North" | "East" | "South" | "West"

type Command =
| {kind: "Face", direction: Direction}
| {kind: "Start"}
| {kind: "Stop"}
| {kind: "Chain", first: Command, second: Command}

const moveEast = {
  kind: "Chain",
  first: {kind: "Face", direction: "East"} ,
  second: {
    kind: "Chain",
    first: {kind: "Start"} ,
    second: {kind: "Stop"}
  }
}
```

SCRIPT DE COMMANDE

ENCHAINER LES COMMANDES AVEC UN TYPE OU RÉCURSIF



```
type Direction = "North" | "East" | "South" | "West"

type Command =
| {kind: "Face", direction: Direction}
| {kind: "Start"}
| {kind: "Stop"}
| {kind: "Chain", first: Command, second: Command}

const moveEast = {
  kind: "Chain",
  first: {kind: "Face", direction: "East"} ,
  second: {
    kind: "Chain",
    first: {kind: "Start"} ,
    second: {kind: "Stop"}
  }
}
```

SCRIPT DE COMMANDE

UNE COMMANDE PLUS COMPLEXE

```
const moveWestThenNorth = {
  "kind": "Chain",
  "first": {
    "kind": "Face",
    "direction": "West"
  },
  "second": {
    "kind": "Chain",
    "first": {
      "kind": "Start"
    },
    "second": {
      "kind": "Chain",
      "first": {
        "kind": "Stop"
      },
      "second": {
        "kind": "Chain",
        "first": {
          "kind": "Face",
          "direction": "North"
        },
        "second": {
          "kind": "Chain",
          "first": {
            "kind": "Start"
          },
          "second": {
            "kind": "Stop"
          }
        }
      }
    }
  }
}
```

SCRIPT DE COMMANDE

UNE COMMANDE PLUS COMPLEXE

Est complexe à lire

```
const moveWestThenNorth = {
  "kind": "Chain",
  "first": {
    "kind": "Face",
    "direction": "West"
  },
  "second": {
    "kind": "Chain",
    "first": {
      "kind": "Start"
    },
    "second": {
      "kind": "Chain",
      "first": {
        "kind": "Stop"
      },
      "second": {
        "kind": "Chain",
        "first": {
          "kind": "Face",
          "direction": "North"
        },
        "second": {
          "kind": "Chain",
          "first": {
            "kind": "Start"
          },
          "second": {
            "kind": "Stop"
          }
        }
      }
    }
  }
}
```

SCRIPT DE COMMANDE

KEEP CALM & USE FUNCTIONS

```
const face = (d: Direction) : Command => ({kind: "Face", direction: d})
const start = () : Command => ({kind: "Start"})
const stop_ = () : Command => ({kind: "Stop"})
const chain = (first: Command, second: Command) : Command => ({kind: "Chain", first, second})

const pipe = (first: Command, ...rest: Array<Command>) : Command =>
  rest.length > 0
  ? chain(first, pipe(rest[0], ...rest.slice(1)))
  : first
```

SCRIPT DE COMMANDE

KEEP CALM & USE FUNCTIONS 

```
const face = (d: Direction) : Command => ({kind: "Face", direction: d})
const start = () : Command => ({kind: "Start"})
const stop_ = () : Command => ({kind: "Stop"})
const chain = (first: Command, second: Command) : Command => ({kind: "Chain", first, second})

const pipe = (first: Command, ...rest: Array<Command>) : Command =>
  rest.length > 0
  ? chain(first, pipe(rest[0], ...rest.slice(1)))
  : first
```

```
const moveWestNorth = pipe(
  face("West"),
  start(),
  stop_(),
  face("North"),
  start(),
  stop_()
)
```


TAKE AWAY

NOUS AVONS VU

Le type `Command` est récursif : s'exprime en terme de `{kind: "Chain", first Command, second Command}`

Un système qui a des types « ET », des types « OU » et des types récursifs s'appelle un système de données algébriques (ADT)

Un ADT permet de représenter les valeurs autorisées d'un programme et leurs transitions



SUIVRE LES TRANSITION

NOUS POUVONS ÉCRIRE DES TRANSITIONS INVALIDES



```
const invalid = pipe(  
  face("East"),  
  stop_  
)
```

SUIVRE LES TRANSITION

NOUS POUVONS ÉCRIRE DES TRANSITIONS INVALIDES



```
const invalid = pipe(  
  face("East"),  
  stop_()  
)
```

SUIVRE LES TRANSITION

NOUS POUVONS ÉCRIRE DES TRANSITIONS INVALIDES



```
const invalid = pipe(  
  face("East"),  
  stop_  
)
```

SUIVRE LES TRANSITION

APPROXIMATION D'UN GADT

```
import { identity } from "fp-ts/lib/function";

type Direction = "North" | "East" | "South" | "West"

type Command<A, B, C> =
  | { kind: "Face", direction: Direction, proof: (_a: A) => A }
  | { kind: "Start", proof: (_a: A) => B }
  | { kind: "Stop", proof: (_a: A) => B }
  | { kind: "Chain", first: Command<A, B, any>, second: Command<B, C, any> }

const face = (d: Direction)
  : Command<"Idle", "Idle", void> => ({ kind: "Face", direction: d, proof: identity<"Idle"> })
const start = (): Command<"Idle", "Moving", void> => ({ kind: "Start", proof: (_: "Idle") => "Moving" })
const stop_ = (): Command<"Moving", "Idle", void> => ({ kind: "Stop", proof: (_: "Moving") => "Idle" })
const chain = <A, B, C>(first: Command<A, B, any>, second: Command<B, C, any>)
  : Command<A, C, any> => ({ kind: "Chain", first, second }) as Command<A, C, any>
```

SUIVRE LES TRANSITION

APPROXIMATION D'UN GADT

```
import { identity } from "fp-ts/lib/function";

type Direction = "North" | "East" | "South" | "West"

type Command<A, B, C> =
  | { kind: "Face", direction: Direction, proof: (_a: A) => A }
  | { kind: "Start", proof: (_a: A) => B }
  | { kind: "Stop", proof: (_a: A) => B }
  | { kind: "Chain", first: Command<A, B, any>, second: Command<B, C, any> }

const face = (d: Direction)
  : Command<"Idle", "Idle", void> => ({ kind: "Face", direction: d, proof: identity<"Idle"> })
const start = (): Command<"Idle", "Moving", void> => ({ kind: "Start", proof: (_: "Idle") => "Moving" })
const stop_ = (): Command<"Moving", "Idle", void> => ({ kind: "Stop", proof: (_: "Moving") => "Idle" })
const chain = <A, B, C>(first: Command<A, B, any>, second: Command<B, C, any>)
  : Command<A, C, any> => ({ kind: "Chain", first, second }) as Command<A, C, any>
```

On paramètre le type command avec 3 paramètres

Pour Face, B & C sont des types fantômes

Pour Start et Stop, C est un type fantôme

On ajoute des preuves :

Face conserve l'état de la machine: "Idle". La fonction Identity peut être utilisée comme preuve.

Start fait avancer la machine à états : "Idle" => "Moving"

Start fait avancer la machine à états : "Idle" => "Moving"

Chain fait avancer la machine à états en composant des commandes par associativité : la preuve est directement encodée dans leurs types

SUIVRE LES TRANSITION

ON PEUT CRÉER UNIQUEMENT DES COMMANDES VALIDES

```
/* COMPILER */
let validCmd =
  chain(
    chain(
      face("East"),
      start()
    ),
    stop_()
  );

/* ERREUR DE COMPILATION */
let invalidCmd = chain(
  face("East"),
  stop_()
);
```


TAKE AWAY

NOUS AVONS VU

Les GADT sont des types « OU » qui possèdent des témoins de type

ADT + GADT permettent de valider qu'on ne représente que des états autorisés et qu'on effectue que des transitions d'états autorisés

En Java ou TS, leur encodage et le messages d'erreurs sont complexe; leur utilisation est discutablement intéressante mais possible

Beaucoup plus aisé en Haskell ou OCaml





| CITIZEN | |
|------------|-------|
| J. SOKAL | 6 6 4 |
| M. DELIC | 4 3 3 |
| GAME SCORE | |
| 0 - 15 | |
| US OPEN | |

TENNIS KATA

TRAVAUX PRATIQUES

LIENS

GROUPE 1 (Thomas) <https://classroom.github.com/a/Nd8M0mh6>

GROUPE 2 (Quentin) https://classroom.github.com/a/CSHbQ_xq

THE BOSS

SI VOUS VOULEZ ALLER PLUS LOIN

Renforcement :

 Comprendre les ADT

 The power of composition

Diversification :

 Categories for the Working Hacker

 Writing Safer Code Using GADTs



QUALITÉ DU SI



COURS 3 - FRONTEND DATAFLOW

QUALITÉ DU SI



COURS 4 - BLOCKCHAINS, SMART CONTRACTS

QUALITÉ DU SI



COURS 5 - GESTION D'ERREURS & PRÉSENTATION PROJET

QUALITÉ DU SI



COURS 6 - CLOUD & MICROSERVICES

QUALITÉ DU SI



COURS 7 - COMMUNICATION INTERPROCESS

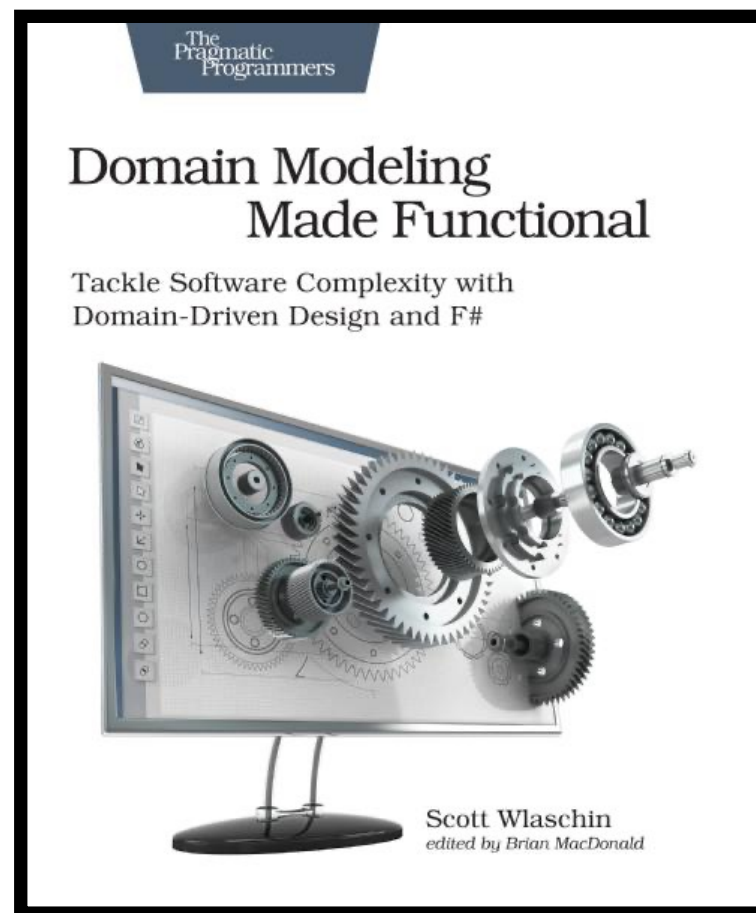
FINAL BOSS

LISEZ ! A METTRE ENTRE TOUTES LES MAINS

Product Owner :

Architectes :

Chefs de projets :



<https://pragprog.com/titles/swdddf/domain-modeling-made-functional/>

