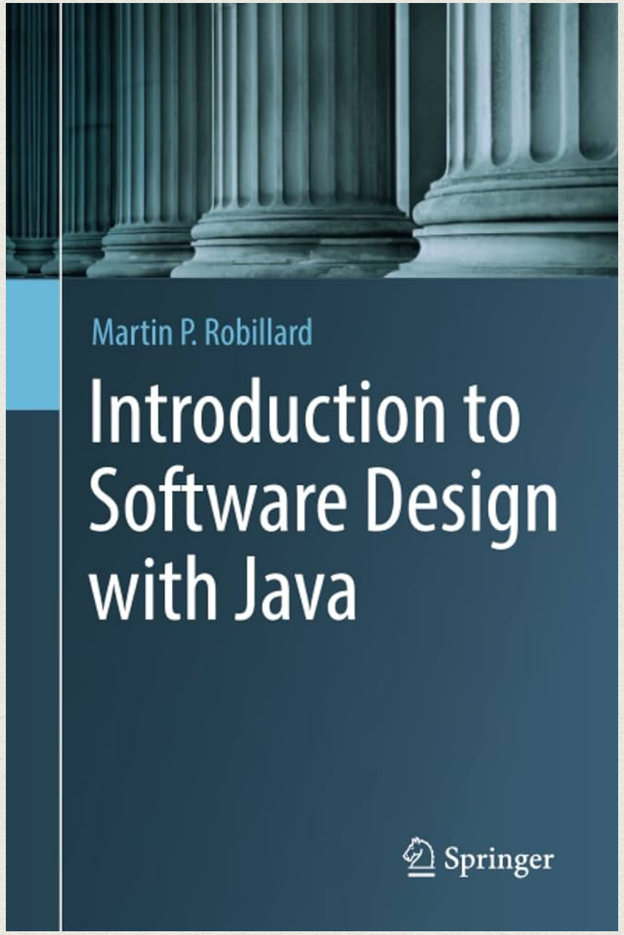
M1 MIAGE / MAPC
GESTION DE L'ÉTAT DES OBJETS

Pascal POIZAT

RÉFÉRENCES


Martin P. Robillard
Introduction to Software Design with Java
Springer

Conception Logicielle
Site web du cours INF5153 de l'UQAM
Retourner à la page d'accueil
Introduction
Notes de cours
Introduction au cours
Écosystème de développement
Schotten Totten
Capsules Vidéos
Introduction au cours
Introduction du cours de Conception Logicielle [IN...
Partager
Regarder sur YouTube
Conception
Sébastien Mosser - INF5153
Chapitre 0 - Capsule 1
Automne 2020

NOTION DE DYNAMIQUE

- **complémentarité** vision **statique** / vision **dynamique** d'un objet
  « *Deck a des cartes* » / « *on ne peut pas tirer de carte si Deck vide* »

- notions d'**état** d'objet
  notions associées : contraintes sur l'état, changement d'état

- **difficulté** de gérer changement d'état et effet sur les opérations
  opération possible ou non, modification de la sémantique de l'opération

NOTIONS D'ÉTAT

- état **exprimable** vs **atteignable** vs **souhaitable**
  état **concret** vs **abstrait**

- `Player { int score; }` donne $\mathbf{2^{32}}$ états exprimables
  tous atteignables ? tous souhaitables ? à la bataille avec 52 cartes non, maximum 26

- `Game { Player j1, j2; }` donne $2^{32} \times 2^{32}$ soit 2^{64} états !
  à comparer à $|\{(x, y) \mid x \geqslant 0, y \geqslant 0, x+y \leqslant 26\}|$ et à `{START,DURING,END}` ou `{WIN1,WIN2,TIE}`

- **exprimable** = représentable en fonction du choix de données
  atteignable = que l'on peut obtenir par appel d'opérations (à partir de l'état initial)
  souhaitable = que l'on souhaite atteindre

MODÉLISATION

- on utilise les **diagrammes d'états d'UML** pour modéliser
  - **états** (dont 1 état **initial** et 0+ état **finaux**)
  - **transitions**
    `source - événement(paramètres) [garde] / actions → cible`
    la garde porte sur variables de l'état + paramètres de l'événement
- quelle **sémantique de l'absence de transition** ?
  erreur … ou ne rien faire ?

DIAGRAMME D'ÉTATS DU DECK

- si on **réunit** création et mélange

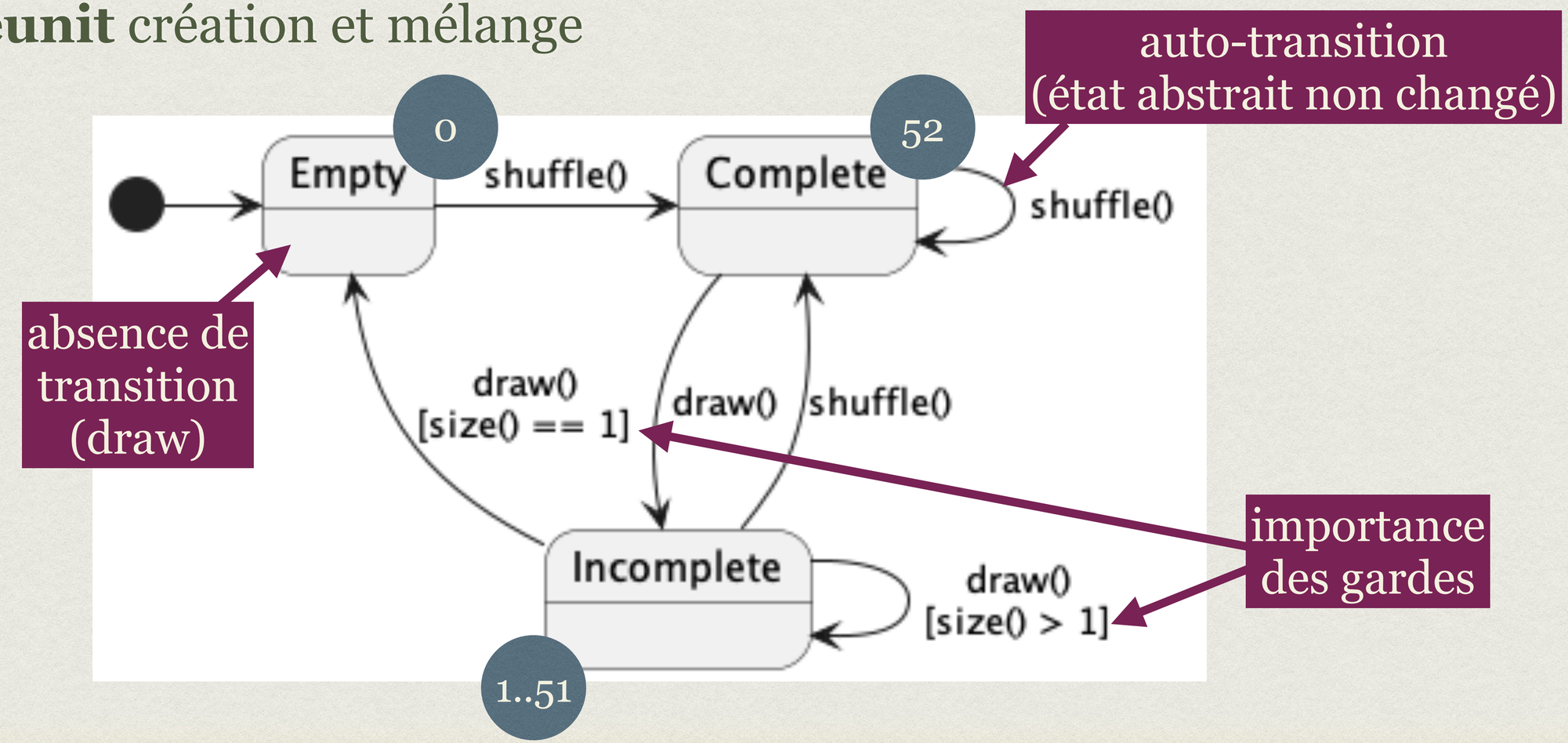
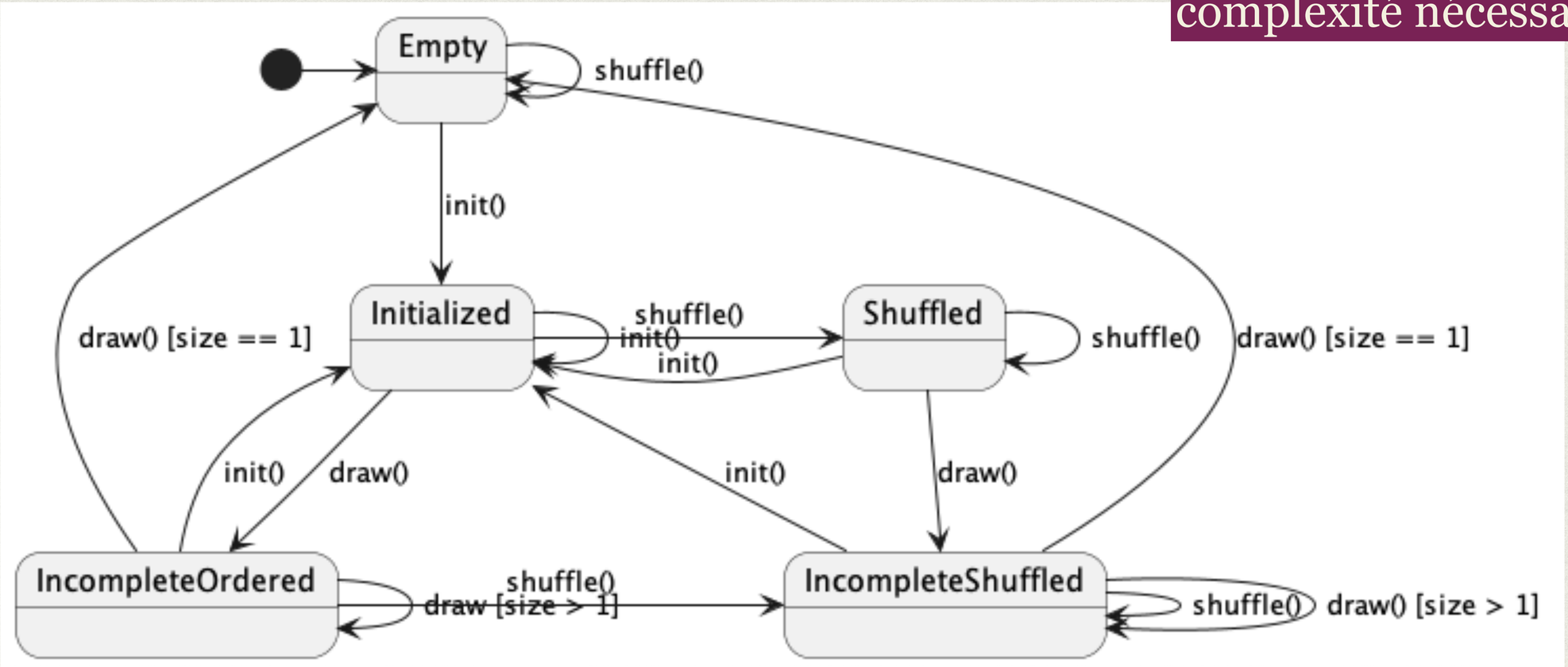


DIAGRAMME D'ÉTATS DU DECK

- si on **sépare** création et mélange

complexité nécessaire ?



BONS PRINCIPES

- **minimiser** l'espace des états
  supprimer les états **invalides** ou **inutiles**
  conserver uniquement ce qui est **nécessaire**

- **cas du Deck :** faut-il séparer création (`init`) et mélange (`shuffle`) ?
  pour un jeu de solitaire cela ne fait pas sens de séparer

- attention à la `SPECULATIVE GENERALITY`+
  « ***et si un jour*** *on a besoin de mélanger en cours de partie ?* »

BONS PRINCIPES

- **cas du Deck :** faut-il un attribut size pour « optimiser » ?
  car size() est souvent appelée

- attention aux `TEMPORARY FIELD`+s

- situation typique du **trade-off** entre temps de calcul et mémoire
  en général **non significatif** par rapport aux entrées-sorties

- ne pas stocker de valeur sauf si **contribue à la valeur** intrinsèque de l'objet

ABSENCE DE VALEUR

```
ICard card = null;
System.out.println(card.rank());

// Exception in thread "main" java.lang.NullPointerException:
// Cannot invoke "org.example.cards.cours5.metier.api.ICard.rank()" because "card" is null
```

- « *I call it my* ***billion-dollar mistake****. It was the invention of the null reference in 1965.* »
  — Sir Charles Antony Richard Hoare (Tony Hoare), 2009

- **interprétation** de `null` ?
  non initialisation temporaire, non initialisation incorrecte, absence de valeur, flag

ABSENCE DE VALEUR : PAS DE MODÉLISATION

- on modélise pour **éviter** les états (abstraits) avec variable non initialisée

validation des entrées

```java
public class Card implements ICard {
    private Rank rank; // ne doit pas être null
    private Suit suit; // ne doit pas être null

    /**
     * Creates a new card.
     * Throws IllegalArgumentException is a parameter is null.
     * @param rank
     * @param suit
    */
    public Card(Rank rank, Suit suit) {
        if (rank == null || suit == null)
            throw new IllegalArgumentException();
        this.rank = rank;
        this.suit = suit;
    }
```

contrats

```java
public class Card implements ICard {
    private Rank rank; // ne doit pas être null
    private Suit suit; // ne doit pas être null

    /**
     * Creates a new card.
     * @param rank
     * @param suit
     * @pre rank ≠ null && suit ≠ null
     */
    public Card(Rank rank, Suit suit) {
        assert rank ≠ null;
        assert suit ≠ null;
        this.rank = rank;
        this.suit = suit;
    }
```

ABSENCE DE VALEUR : MODÉLISATIONS PROBLÉMATIQUES

```
public class Card implements ICard {
    private Rank rank;
    private Suit suit;
    private boolean isJoker;

    public boolean isJoker() { return isJoker; }
    public Rank getRank() { return rank; } // et pour le Joker ?
    public Suit getSuit() { return suit; } // et pour le Joker ?
}
```

- références `null` : peut conduire à `NullPointerException`
- valeurs arbitraires (ex: as de coeur) : **prête à confusion**
- valeur spéciale d'un type énuméré (ex: `INEXISTANT`) : abus d'`enum` + **casse le code**

ABSENCE DE VALEUR : AD-HOC / TYPES OPTIONNELS

- **fonctionnalités ad-hoc du langage**
  ex : `c: Card` et `c.suit` vs `c: Card?` et `c?.suit` en Kotlin

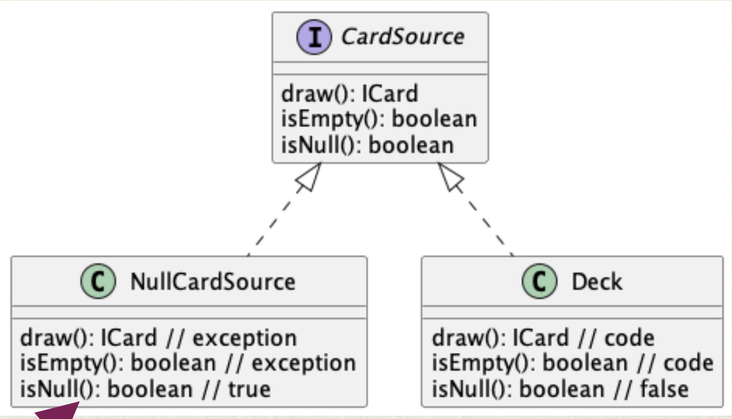
- type `Option[T]` et API associée (`empty`, `isPresent`, `get`, ...)
  ex : Optional<T> en Java, Option<T> en Java+vavr

- c'est parfois **lourd pour le client**

ABSENCE DE VALEUR : NULL OBJECT

```
public interface CardSource extends Iterable<ICard> {

    public static CardSource NULL = new CardSource() {
        @Override public ICard draw(){ assert !isEmpty(); return null; }
        @Override public boolean isEmpty() { return true; }
        @Override public boolean isNull() { return true; }
        @Override public Iterator<ICard> iterator() {
            return List.<ICard>of().iterator();
        }
    };

   ICard draw();

   boolean isEmpty();

   default boolean isNull() { return false; }
}
```

implantation
efficace grâce à Java 8+



patron
Null Object

- Optional<T> : encapsulation
  ici : extension du type T U { objet nul }

IMMUTABILITÉ ET « FINAL »

- comment **empêcher un changement** d'état **après initialisation** ?

- utilisation du mot-clé **final**

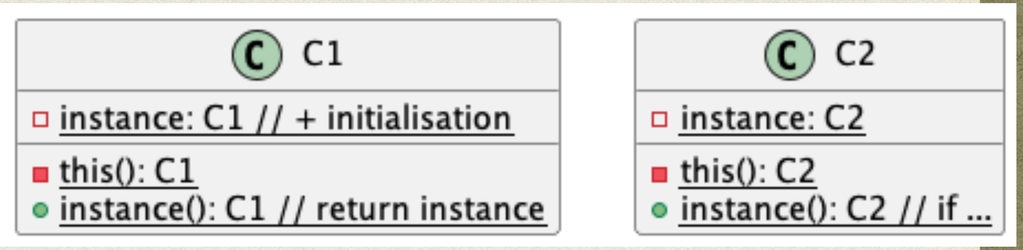
  attention : final gère la mise à jour de référence,
  sur une collection cela ne la rend pas immuable

- en général pas utile pour les variables locales aux méthodes
  sinon probablement présence de `LONG METHOD+`

IDENTITÉ ET EGALITÉ

- attention à la différence entre **identité** (==) et **égalité** (equals)
- on souhaite parfois tenir compte de critères d'**unicité**
  il n'y a pas deux objets distincts d'une classe qui soient égaux
- 2 **patrons** peuvent aider :
  - une seule valeur pour une classe : `SINGLETON` (attention à STUPID)
  - n valeurs uniques pour une classe (ex : cartes) : `FLYWEIGHT`

SINGLETON



- **check-list pour une classe `C` `SINGLETON` :**

  [X] **constructeur privé**

  pour stocker l'instance unique :

  [X] **variable statique privée**

  [X] décider si on veut initialiser cette variable **à la création <u>ou</u> au besoin** (coût)

  ```
  if (instance == null) { instance = new C(…) }; return instance
  ```

  pour accéder à l'instance :

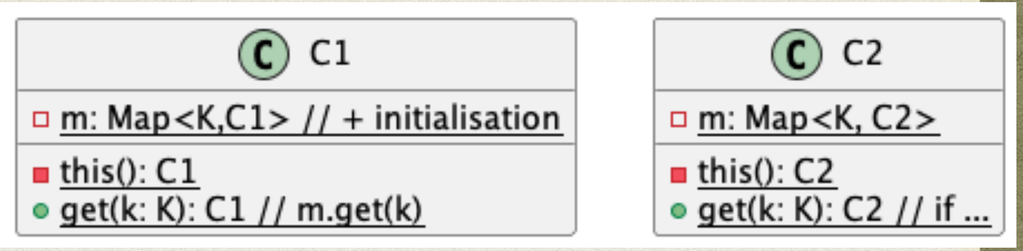
  [X] **getter statique**

  penser à la synchronisation :

  [X] nécessaire si multi-thread et initialisation au besoin

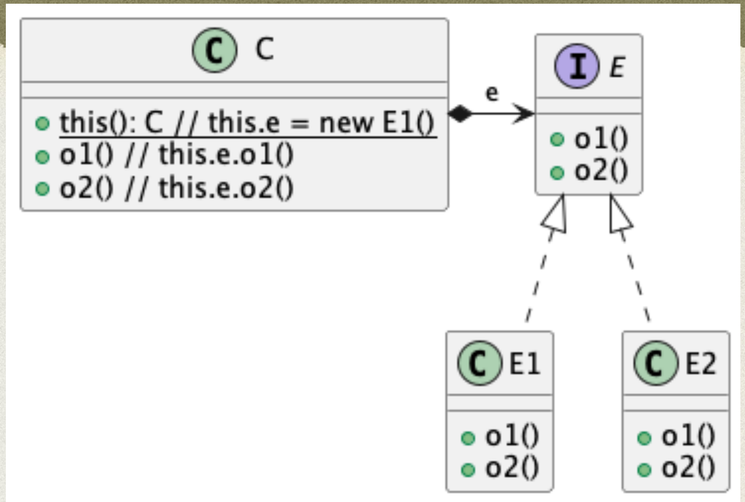
FLYWEIGHT

- forme de `SINGLETON` adaptée
  pour avoir **plusieurs valeurs uniques**



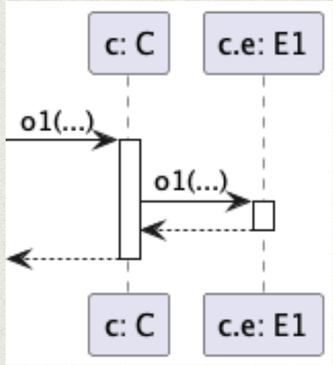
- **check-list pour une classe `C` `FLYWEIGHT` :**
  [X] **constructeur privé**
  [X] **variable statique privée de type collection** pour les instances
  typiquement `Map<K,C> m` où `K` ensemble attributs utilisés pour l'égalité sur `C`
  [X] **getter statique avec paramètre clé d'identification**
  typiquement `get(k: K):C`
  [X] mêmes questions sur initialisation à la création (e.g., via `static { … }`) ou au besoin
  et sur synchronisation

STATE (BASE)



checklists pour un contexte `C` qui a un état `E`

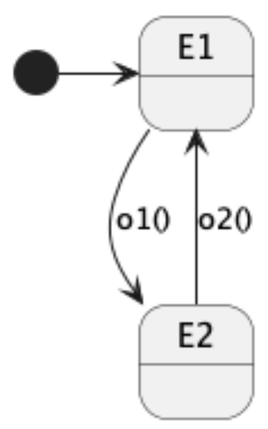
- **pour le contexte :**
  [X] **variable état** `e` de type `E`
  initialisée avec un objet de la **classe de l'état initial**
  [X] pour toute opération **`C::o`** de la signature de `C` **affectée par / affectant** l'état,
  une opération **`E::o`** et une **délégation**
  `C::o(…) { this.e.o(…) }`



- **pour les états :**
  [X] **interface `E`** reprenant les **opérations délégués** par `C`
  [X] **classes `Ei` implémentant** l'interface `E` (potentiellement `SINGLETON`s)

STATE (ERREURS)

comment traiter les erreurs (opération `o` non faisable dans l'état `Ei`) ?



- **dans chaque `Ei` concerné par `o` impossible :**

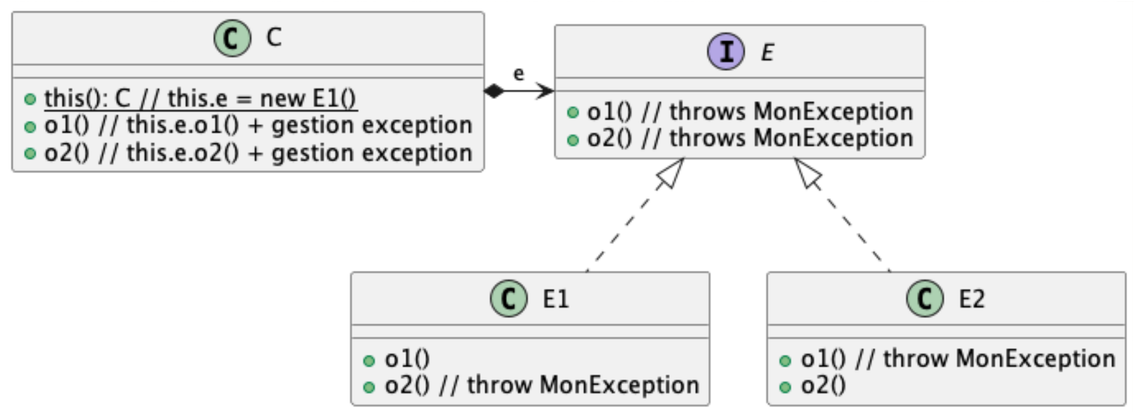
  [X] **levée d'exception** : `Ei::o(…) { throw new MonException(« … ») }`

  [X] **modification de la signature** de `E::o(o throws MonException)`

- **dans `C::o`**

  [X] **faire un choix :**

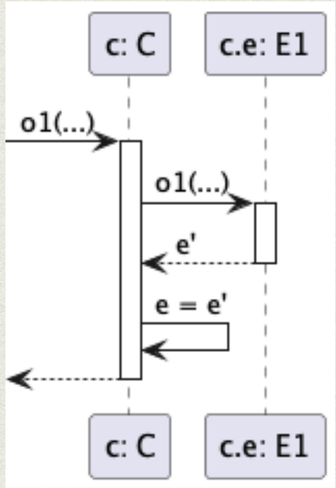
  - transfert exception (récupérable par le client)
  - récupération exception et traitement spécifique



l'utilisation de types comme `Try<T>` est une alternative

STATE (TRANSITION CHOIX 1)

comment traiter la transition entre états ? `E1 - o() → E2`



premier choix : **retour d'état**
possible si aucune opération concernée ne retourne de valeur

```
[X]E::o(…): E
[X]E1::o(…): E { … travail … ; return new E2() }
[X]C::o(…):() { e = e.o(…) } // délégation et modification d'état
```

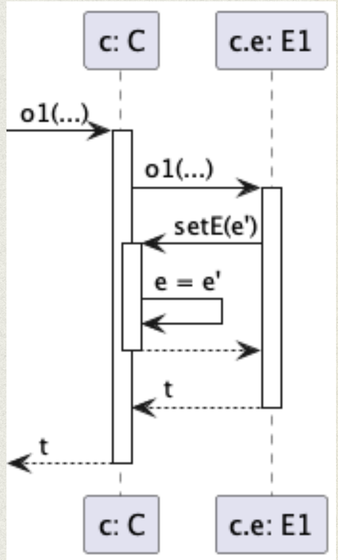
on peut aussi retourner des `Tuple2<T, E>` si besoin

STATE (TRANSITION CHOIX 2)

comment traiter la transition entre états ? `E1 - o() → E2`

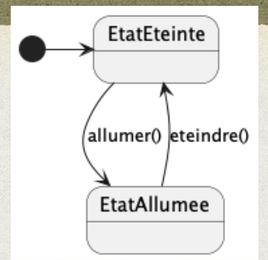
second choix : **contexte en paramètre et modificateur d'état**
permet aussi, si besoin, l'accès à des méthodes de travail dans C (visibilité paquetage)

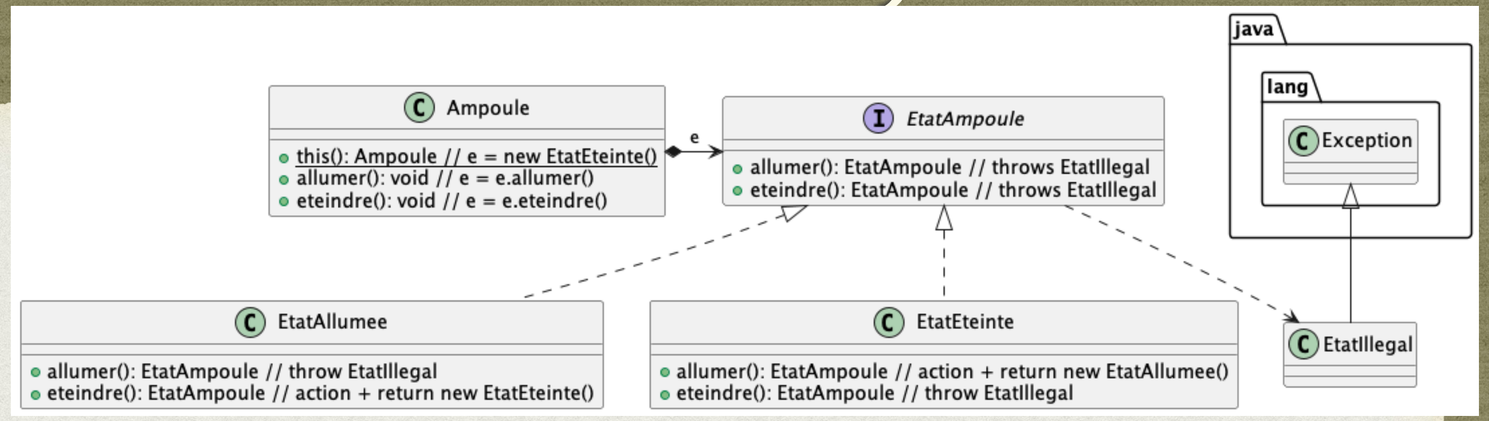
```
[X] E::o(c: C, …): T
[X] E1::o(c: C, …): T { … travail … ; c.setE(new E2()); return … }
[X] C::o(…): T { return e.o(this, …) }
[X] C::setE(e': E): () { this.e = e' }
```



important : visibilité paquetage pour `setE`

STATE (EXEMPLE V1)






```
public interface EtatAmpoule {
    EtatAmpoule allumer() throws EtatIllegal;
    EtatAmpoule eteindre() throws EtatIllegal;
}

public class EtatAllumee implements EtatAmpoule {
    @Override public EtatAmpoule allumer() throws EtatIllegal { throw new EtatIllegal("ampoule déjà allumée »); }
    @Override public EtatAmpoule eteindre() throws EtatIllegal {
        System.out.println("l'ampoule s'éteint");
        return new EtatEteinte();
    }
}

public class EtatEteinte implements EtatAmpoule {
    @Override public EtatAmpoule allumer() throws EtatIllegal {
        System.out.println("l'ampoule s'allume");
        return new EtatAllumee();
    }
    @Override public EtatAmpoule eteindre() throws EtatIllegal { throw new EtatIllegal("ampoule déjà éteinte »); }
}

public class Ampoule {
    private EtatAmpoule e;
    public Ampoule() { this.e = new EtatEteinte(); }
    public void allumer() throws EtatIllegal { this.e = this.e.allumer(); }
    public void eteindre() throws EtatIllegal { this.e = this.e.eteindre(); }
}
```

```
public class App {
    public static void main(String[] args) {
        // cas qui va bien
        Ampoule a = new Ampoule();
        try {
            a.allumer();
            a.eteindre();
        } catch (EtatIllegal e){ System.out.println(e.getMessage()); }
        // cas qui pose problème 1
        a = new Ampoule();
        try {
            a.allumer();
            a.allumer();
        } catch (EtatIllegal e){ System.out.println(e.getMessage()); }
        // cas qui pose problème 2
        a = new Ampoule();
        try {
            a.allumer();
            a.eteindre();
            a.eteindre();
        } catch (EtatIllegal e){ System.out.println(e.getMessage()); }
    }
}
```

STATE (EXEMPLE V2)

```java
public interface EtatAmpoule {
    Try<Integer> allumer(Ampoule a);
    Try<Void> eteindre(Ampoule a);
}

public class EtatAllumee implements EtatAmpoule {
    private EtatAllumee() {}
    public static final EtatAmpoule INSTANCE = new EtatAllumee();
    @Override public Try<Integer> allumer(Ampoule a) {
        return Try.failure(new EtatIllegal("ampoule déjà allumée »)); }
    @Override public Try<Void> eteindre(Ampoule a) {
        a._eteindre();
        a.changeEtat(EtatEteinte.INSTANCE);
        return Try.success(null);
    }
}

public class EtatEteinte implements EtatAmpoule {
    private EtatEteinte() {}
    public static final EtatAmpoule INSTANCE = new EtatEteinte();
    @Override public Try<Integer> allumer(Ampoule a) {
        a._allumer();
        a.changeEtat(EtatAllumee.INSTANCE);
        return Try.success(a.lumen());
    }
    @Override public Try<Void> eteindre(Ampoule a) {
        return Try.failure(new EtatIllegal("ampoule déjà éteinte »)); }
}
```

```java
public class Ampoule {
    private EtatAmpoule e;
    private int lumen;
    private Consumer<String> informateur;

    public Ampoule(Consumer<String> informateur, int lumen) {
        this.lumen = lumen;
        this.informateur = informateur;
        this.e = EtatEteinte.INSTANCE;
    }
    // visibilité package
    void _allumer() { informateur.accept("l'ampoule s'allume"); }
    void _eteindre() { informateur.accept("l'ampoule s'éteint"); }
    int lumen() { return lumen; }
    void changeEtat(EtatAmpoule e) { this.e = e; }

    private Consumer<? super Throwable> informateurErreur =
        exception -> informateur.accept(exception.getMessage());

    public Option<Integer> allumer() {
        return this.e.allumer(this).onFailure(informateurErreur).toOption();
    }

    public void eteindre() {
        this.e.eteindre(this).onFailure(informateurErreur);
    }
}
```

```java
public class App {
    private static final Consumer<String> afficheur = System.out::println;
    public static void main(String[] args) {
        final int LUMEN = 2500;
        Ampoule a;
        Option<Integer> lumens;
        // cas qui va bien
        a = new Ampoule(afficheur, LUMEN);
        lumens = a.allumer();
        System.out.println(lumens + " lumens");
        a.eteindre();
        // cas qui pose problème 1
        a = new Ampoule(afficheur, LUMEN);
        lumens = a.allumer();
        System.out.println(lumens + " lumens");
        a.allumer();
        // cas qui pose problème 2
        a = new Ampoule(afficheur, LUMEN);
        lumens = a.allumer();
        System.out.println(lumens + " lumens");
        a.eteindre();
        a.eteindre();
    }
}
```

exercice : analyser les différences entre v1 et v2

SYNTHÈSE

- importance de la notion d'**état**
  différencier états exprimables / atteignables / souhaitables
- **minimiser** l'espace d'état, pas de variable inutile,
  et s'assurer que **seuls les états souhaitables soient atteignables**
- attention à `Speculative Generality`†, `Temporary Fields`† et `Long Method`†
- patron `Null Object` (absence de valeur)
- patron `Singleton` (unicité, valeur unique d'une classe)
- patron `Flyweight` (unicité, valeurs uniques d'une classe)
- patron `State` (respect machine à états)