M1 MIAGE / MAPC
MODÉLISATION ARCHITECTURALE ET PATRONS DE CONCEPTION

Pascal POIZAT

OBJECTIFS

- savoir **concevoir** et **développer** des architectures logicielles de qualité
- connaître quelques uns des **principes** liés à la **qualité** du **logiciel**
  - DRY, SOLID, KISS, STUPID
  - patrons et anti-patrons
- savoir dans quel **contexte** et **comment** appliquer une technique

NON-OBJECTIFS

- un catalogue de patrons (ou anti-patrons) de conception
- une incitation à mettre le maximum de patrons, partout, tout le temps
- un cours d'UML ou de développement en Java (voir cours de L3)
- un cours complet de génie logiciel (voir autres cours de M1)
  par exemple : éléments techniques, conduite de projet, test, ...

ORGANISATION

- 9 séances de 4h (2h+2h)
- **évaluation**
  50% CC / 50% Examen
- **pré-requis**
  - bases de modélisation objet
  - bases de programmation objet
  - outil de modélisation UML
  - environnement de programmation Java

IMPORTANT

- il y a rarement (jamais)
  « **la** » bonne / meilleure solution
  à un problème

- c'est par **l'observation**
  et par **la pratique**
  que vos solutions gagneront en **qualité**

- cette matière demande un
  investissement personnel
  tant en séance qu'en phases de révision

attention !

BIBLIOGRAPHIE PRINCIPALE

- **M. P. Robillard, Introduction to Software Design with Java. Springer, 2017.**

recommandé

- C. Larman, *UML 2 et les Design Patterns (3ème ed)*. Pearson, 2005.
- E. Freeman and E. Robson, *Head First Design Patterns (2nd ed)*. O'Reilly, 2020.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

BIBLIOGRAPHIE SECONDAIRE

- J. Bloch, Effective Java (3rd ed). Addison Wesley, 2018.
- M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- R.-G. Urma, A. Mycroft, and M. Fusco, *Modern Java in Action: Lambdas, streams, functional and reactive programming (2nd ed)*. Manning, 2018.

WEBOGRAPHIE

recommandé

- **supports ACE (S. Mosser)**
  https://conception-objet.github.io/
  https://github.com/ace-lectures/pattern-repository

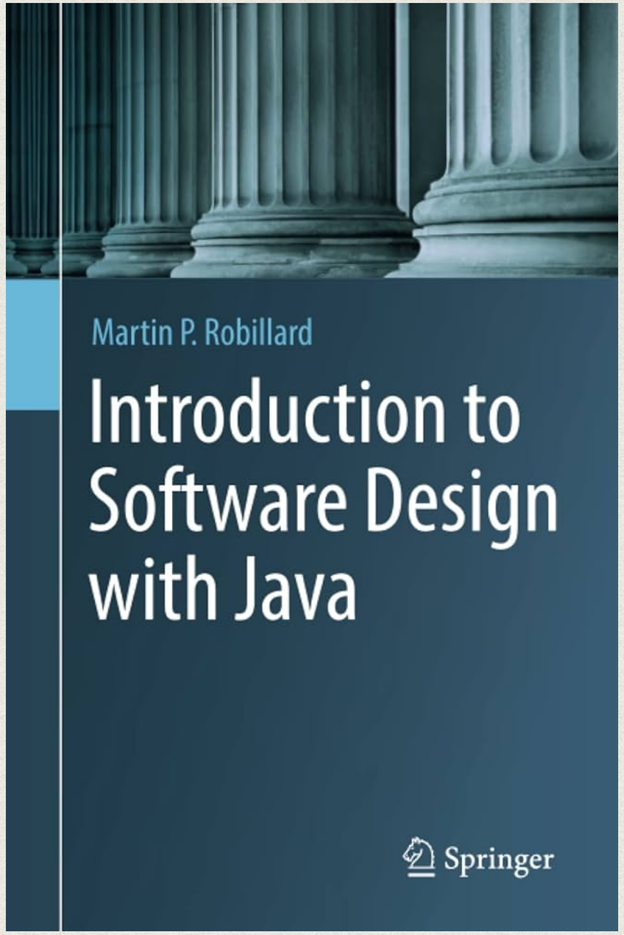
- listes de patrons
  https://java-design-patterns.com/
  https://refactoring.guru/design-patterns
  https://sourcemaking.com/design-patterns-and-tips

- autres
  blog de M. Fowler — https://martinfowler.com/
  blog de R. C. Martin — http://butunclebob.com/
  JEP café — https://www.youtube.com/playlist?list=PLX8CzqL3ArzV4BpOzLanxd4bZr46x5e87

M1 MIAGE / MAPC INTRODUCTION

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

LES 12 CADEAUX DE NOËL

- On the first day of Christmas my true love sent to me
  a partridge in a pear tree

  On the second day of Christmas my true love sent to me
  two turtle doves,
  and a partridge in a pear tree.

  On the third day of Christmas my true love sent to me
  three French hens,
  two turtle doves,
  and a partridge in a pear tree.

  …

LES 12 CADEAUX DE NOËL

- On the first day of Christmas my true love sent to me
  a partridge in a pear tree

  On the second day of Christmas my true love sent to me
  two turtle doves,
  and a partridge in a pear tree.

  On the third day of Christmas my true love sent to me
  three French hens,
  two turtle doves,
  and a partridge in a pear tree.

  …

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

code par I. Philipps, 1988

LES 12 CADEAUX DE NOËL

- On the first day of Christmas my true love sent to me
  a partridge in a pear tree

  On the second day of Christmas my true love sent to me
  two turtle doves,
  and a partridge in a pear tree.

  On the third day of Christmas my true love sent to me
  three French hens,
  two turtle doves,
  and a partridge in a pear tree.

  …

un problème ?

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

code par I. Philipps, 1988

LES 12 CADEAUX DE NOËL

- On the first day of Christmas my true love sent to me
  a partridge in a pear tree

  On the second day of Christmas my true love sent to me
  two turtle doves,
  and a partridge in a pear tree.

  On the third day of Christmas my true love sent to me
  three French hens,
  two turtle doves,
  and a partridge in a pear tree.

  ...

```java
static final int MAX_DAYS = 3;
static final String[] DAYS = { "first", "second", "third" };
static IntFunction<String> firstLine = d -> String
        .format("On the %s day of Christmas, my true love sent to me:%n", DAYS[d]);

static final String[] GIFTS = { "a partridge in a pear tree", "two turtle doves", "three French hens" };
static IntFunction<String> allGifts;
static {
    allGifts = d -> switch (d) {
        case 0 -> "and " + GIFTS[0];
        default -> GIFTS[d] + "\n" + allGifts.apply(d - 1);
    };
}

static final String SEPARATOR = "\n\n";

static String poem() {
    StringBuilder poem = new StringBuilder(firstLine.apply(0) + GIFTS[0] + SEPARATOR);
    for (int day = 1; day < MAX_DAYS; day++) {
        poem.append(firstLine.apply(day) + allGifts.apply(day) + SEPARATOR);
    }
    return poem.toString();
}
```

LES 12 CADEAUX DE NOËL

- On the first day of Christmas my true love sent to me
  a partridge in a pear tree

  On the second day of Christmas my true love sent to me
  two turtle doves,
  and a partridge in a pear tree.

  On the third day of Christmas my true love sent to me
  three French hens,
  two turtle doves,
  and a partridge in a pear tree.

  ...

```java
public class DayInfo {
    private DayInfo() {}

    public static final int MAX_DAYS = 3;
    private static final String[] DAYS = { "first", "second", "third" };
    private static final String[] GIFTS = { "a partridge in a pear tree", "two turtle doves", "three French hens" };

    public static String day(int day) { return (day ≥ 0 && day < MAX_DAYS) ? DAYS[day] : null; }
    public static String gift(int day) { return (day ≥ 0 && day < MAX_DAYS) ? GIFTS[day] : null; }
}

static IntFunction<String> firstLine = d -> String
        .format("On the %s day of Christmas, my true love sent to me:%n", DayInfo.day(d));

static IntFunction<String> allGifts;
static {
    allGifts = d -> switch (d) {
        case 0 -> "and " + DayInfo.gift(d);
        default -> DayInfo.gift(d) + "\n" + allGifts.apply(d - 1);
    };
}

static String poem() {
    final String SEPARATOR = "\n\n";
    StringBuilder poem = new StringBuilder(firstLine.apply(0) + DayInfo.gift(0) + SEPARATOR);
    for (int day = 1; day < DayInfo.MAX_DAYS; day++) {
        poem.append(firstLine.apply(day) + allGifts.apply(day) + SEPARATOR);
    }
    return poem.toString();
}
```

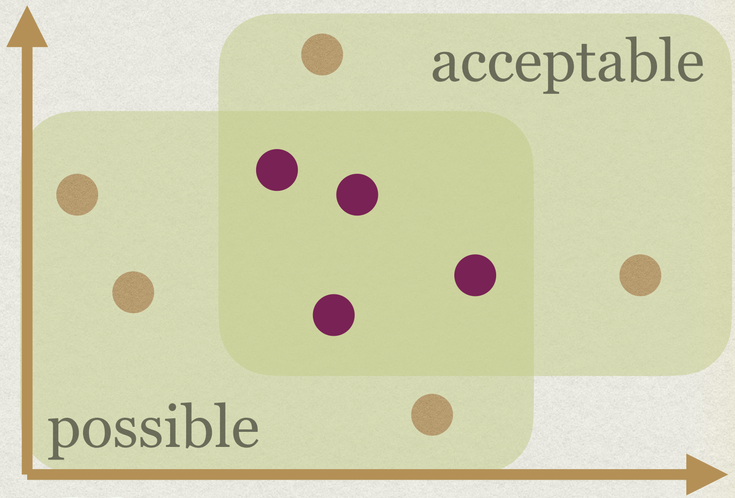
CONCEPTION (*DESIGN*)

- nombreuses définitions (dépend du focus)
  on suit ici [**Robillard, 2017**].

- construction d'**abstractions** (données et calculs)
  et **organisation** de ces abstractions
  en une application logicielle fonctionnelle

PROCESSUS DE CONCEPTION

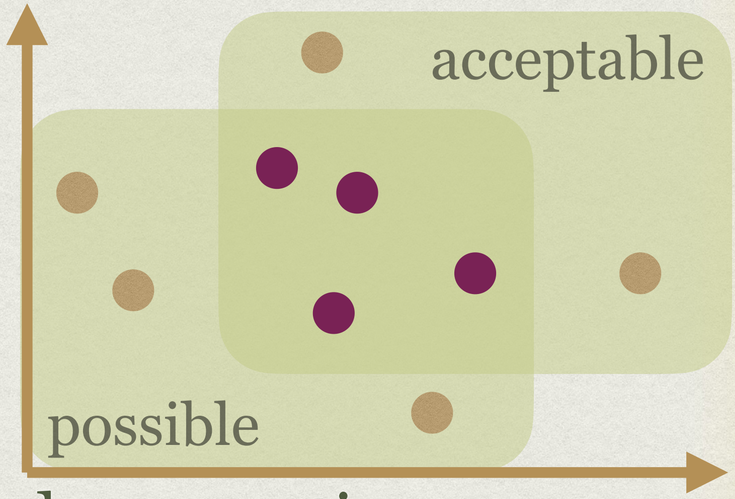
- le **processus de conception** est un processus de **prise de décisions** conduisant à l'obtention d'une **solution** (modèle plus ou moins abstrait)
- **espace de conception** à n dimensions (choix)
- **compromis** de conception
- existence de solutions **possibles** (ou non) et **acceptables** (ou non)



DIFFICULTÉ

- **incertitude** (critères de décision non encore connus, en grand nombre)

- caractère parfois **approximatif**
  de l'évaluation d'un critère de qualité d'une solution



- processus **heuristique** et **itératif**
  guidé par l'expérience

- poids des critères de qualité dépendant du **contexte** de conception
  par exemple : robustesse vs. durabilité d'une solution

CONCEPTION ET PROCESSUS DE DÉVELOPPEMENT

- la conception est seulement l'**une** des activités
  au sein des processus de développement logiciel

- impact de **bonnes pratiques** pour de bonnes solutions

  - **approche évolutive**
    itérations, refactoring, gestion de la dette technique

  - **conventions** et **principes**
    choix des noms, casse, indentation, annotations, commentaires
    DRY, KISS, SOLID, STUPID

PRINCIPES : DRY

- « **Don’t Repeat Yourself** » (Hunt & Thomas, 1999)
- **limiter la répétition**
  1 élément de connaissance = 1 représentation unique
- lien avec bogues (dupliquées) et maintenance (plus difficile)
- préférer passer par des (dépendre d’) **abstractions** moins changeantes
  mais attention à l’abstraction / optimisation prématurée

PRINCIPES : KISS

- « **Keep It Simple, Stupid!** » (origine ?)
- la **simplicité** comme l'un des objectifs de la conception
- éviter la complexité, à nouveau impact sur la maintenance

PRINCIPES : SOLID

- ensemble de principes (par B. Liskov, R. C. Martin, B. Meyer)
- **Single responsibility principle (SCP)**
  responsabilité unique, une unique raison de changer
- **Open/closed principle (OCP)**
  ouverture à l'extension, fermeture à la modification, permettre extension sans modification
- **Liskov substitution principle (LSP)**
  substituabilité, notion sémantique, S < T implique que ce qui est vrai pour tout T doit l'être pour tout S
- **Interface segregation principle (ISP)**
  ne pas dépendre de méthodes que l'on n'utilise pas
- **Dependency inversion principle (DIP)**
  les abstractions ne dépendent pas des détails - les détails dépendent des abstractions

PRINCIPES : STUPID

- ensemble de pratiques considérées comme mauvaises (W. Durand, 2013)
  https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/
  liées à la notion d'état global, de fort couplage, de perte de testabilité
- **Singleton**
- **Tight (strong) coupling**
- **Untestability**
- **Premature optimization,** « root of all evil » (Knuth, 1974)
- **Indescriptible naming**
- **Duplication**

CAPTURE DES CHOIX DE CONCEPTION

- **outils dédiés**
  plateformes de discussion, systèmes de gestion de versions

- **code source**
  choix encodés et potentiellement vérifiables par le compilateur
  mais pas la logique ayant conduit aux choix

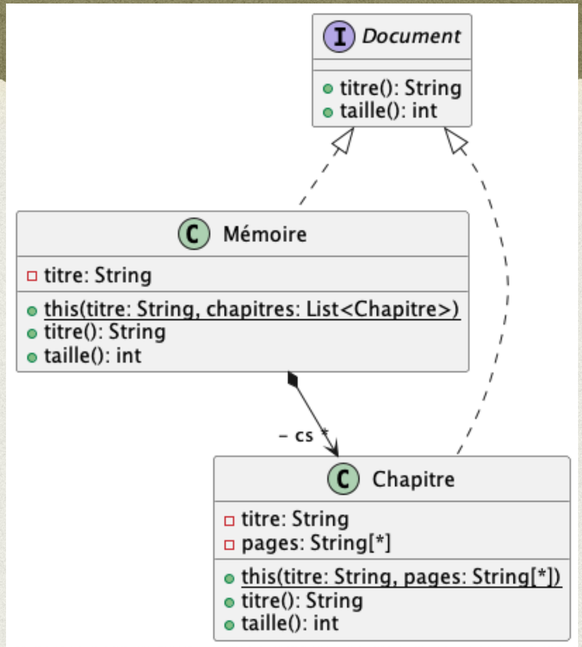
- **documents de conception** de différentes natures (GPML, DSML)
  y compris utilisation de modèles génératifs

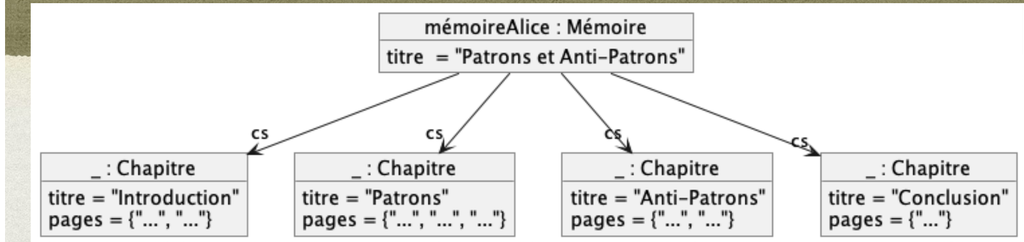
MODÉLISATION : UML

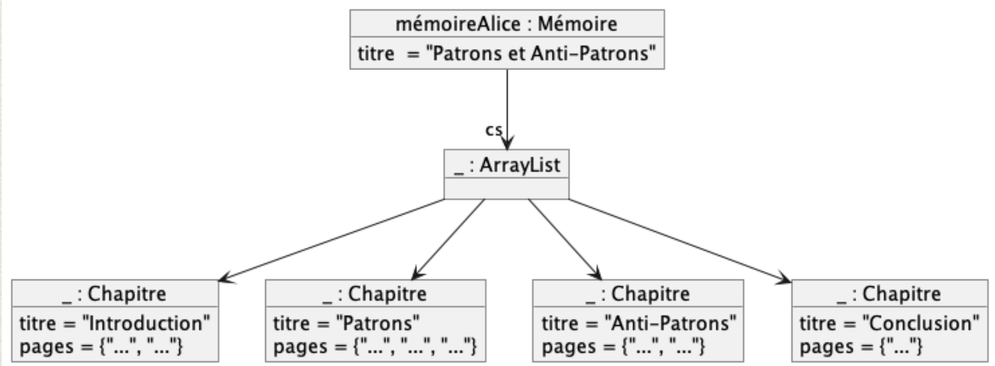
- choix de l'**outil**
  librairie texte vers modèle
  outil plus « lourd »

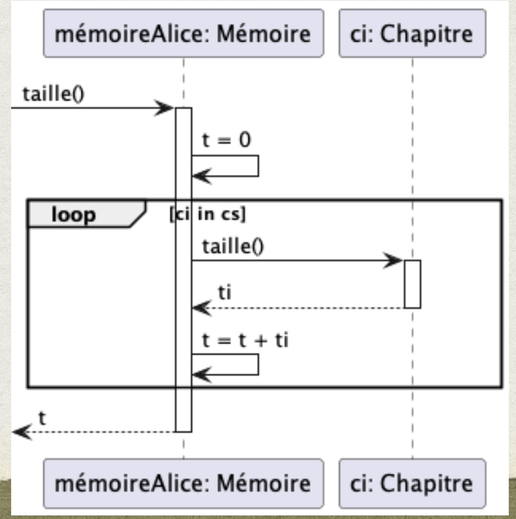
- choix des **diagrammes**
  diagrammes d'objets
  diagrammes de classe
  diagrammes d'états ou de séquence

- choix du **niveau de détail**
  on cible une partie du système
  on donne le détail nécessaire
  UML vs Java



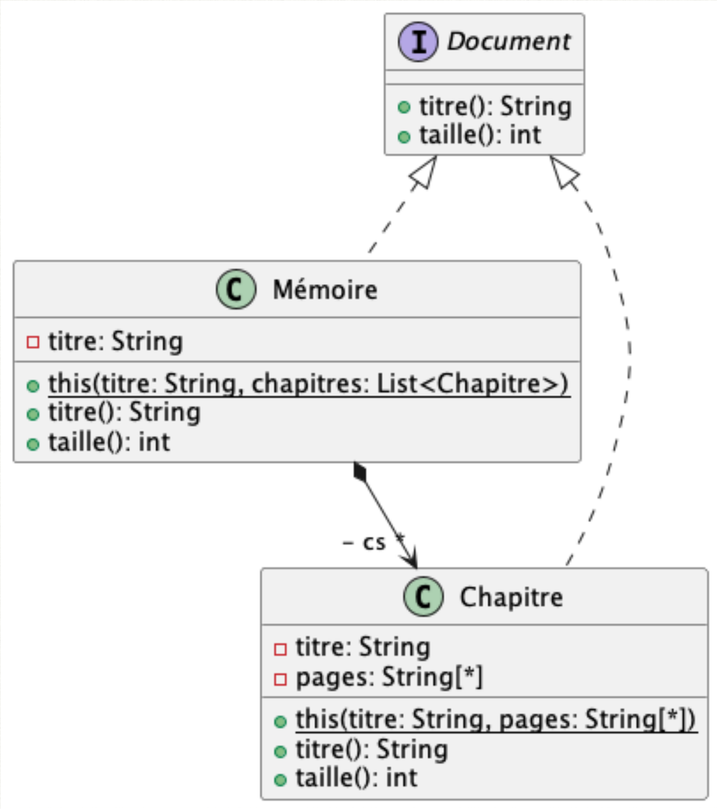






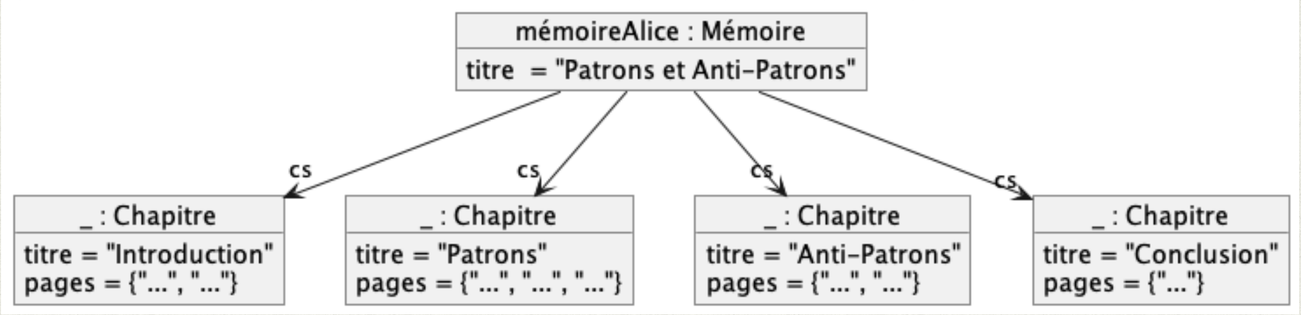
MODÉLISATION : UML

```
@!include ../commons.style
@startuml dc_example
interface Document {
    + titre(): String
    + taille(): int
}
class Mémoire implements Document {
    - titre: String
    + {static} this(titre: String, chapitres: List<Chapitre>)
    + titre(): String
    + taille(): int
}
class Chapitre implements Document {
    - titre: String
    - pages: String[*]
    + {static} this(titre: String, pages: String[*])
    + titre(): String
    + taille(): int
}
Mémoire *--> "- cs *" Chapitre
@enduml
```



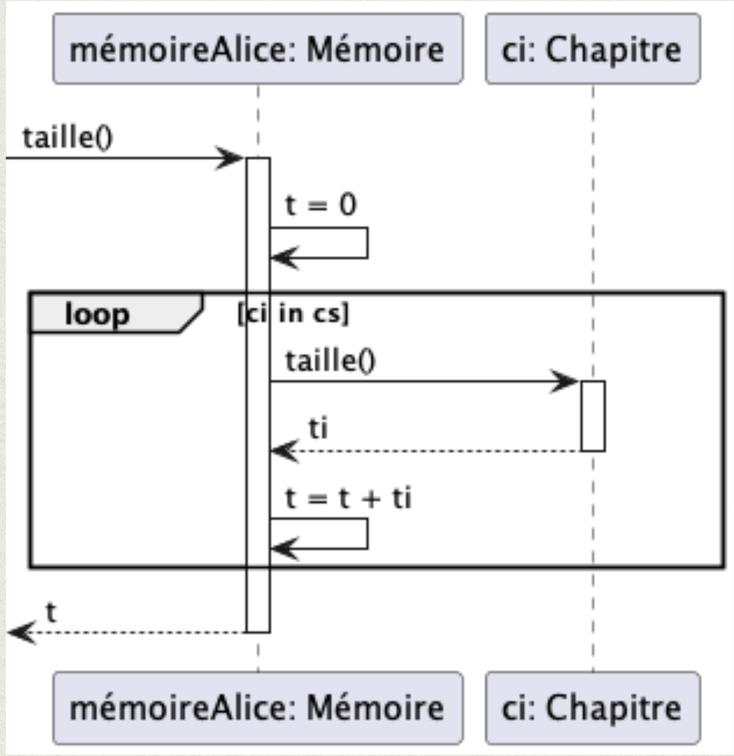
MODÉLISATION : UML

```
@!include ../commons.style
@startuml do_example2

object "mémoireAlice : Mémoire" as mA
mA : titre  = "Patrons et Anti-Patrons"

object "_ : Chapitre" as c1
c1 : titre = "Introduction"
c1 : pages = {"...", "..."}

object "_ : Chapitre" as c2
c2 : titre = "Patrons"
c2 : pages = {"...", "...", "..."}

object "_ : Chapitre" as c3
c3 : titre = "Anti-Patrons"
c3 : pages = {"...", "..."}

object "_ : Chapitre" as c4
c4 : titre = "Conclusion"
c4 : pages = {"..."}

mA ⟶ "cs" c1
mA ⟶ "cs" c2
mA ⟶ "cs" c3
mA ⟶ "cs" c4

@enduml
```



MODÉLISATION : UML

```
@!include ../commons.style
@startuml dseq_example
participant "mémoireAlice: Mémoire" as mA
participant "ci: Chapitre" as ci
[→ mA: taille()
activate mA
    mA → mA : t = 0
    loop ci in cs
        mA → ci: taille()
        activate ci
        mA ⟵ ci: ti
        deactivate ci
        mA → mA : t = t + ti
    end
[⟵ mA: t
deactivate mA
@enduml
```



RAPPELS

- **interface** = **contrat**
  un type abstrait

- **classe** = une **réalisation** (implémentation) possible **de contrat(s)**
  réalise un ou plusieurs types abstraits
  interface publique vs le reste

- **objet** = une **instance** de classe
  une classe vue comme un « moule à objets »
  accès à un service via appel de méthode

CAPTURE DU CORPUS DE L'EXPÉRIENCE

- **patrons de conception**, (très) nombreux
  on cite souvent le « **Gang of Four** », GoF (Gamma et al, 1994)
  proposition de 23 patrons regroupés en 3 grandes catégories

- patrons structurés en **4 parties** :
  nom (= vocabulaire commun)
  problème (= « quand ? »)
  solution (= « comment ? », éléments de conception abstraits)
  conséquences (= bénéfices, compromis à faire)

- même idée pour les défauts de conception : **anti-patrons** (aka odeurs de code)

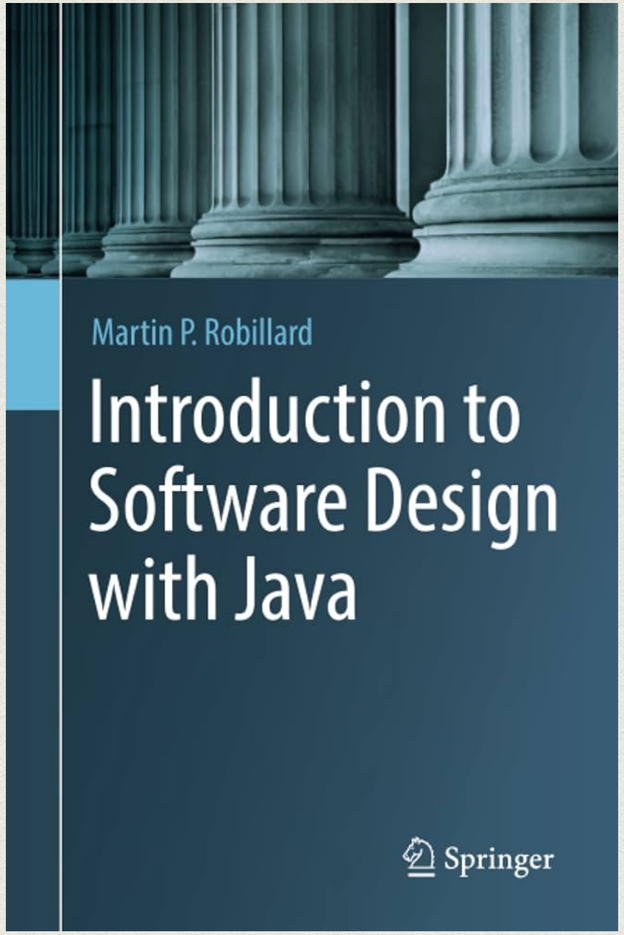
SYNTHÈSE

- processus et solution de **conception**
- rarement « **la** » bonne solution
  surtout si n dimensions de qualité
  résultat de **décisions de conception**
- développement logiciel >> conception
  intérêt de **bonnes pratiques**
  dont DRY, KISS, SOLID, STUPID
- **capture** de connaissances
  liées à un cas d'étude (dont **modèles** UML)
  ou général (**patrons** et **anti-patrons**)

M1 MIAGE / MAPC
ENCAPSULATION

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

ENCAPSULATION

- **décomposition** en abstractions plus faciles à gérer
- limiter le nombre de points de contact (**couplage**)
- principe de **masquage** de l'information

EXEMPLE : CARTES

```
int card = 13; // As de coeur
int suit = card / 13;
int rank = card % 13;
```

EXEMPLE : CARTES

```
int card = 13; // As de coeur
int suit = card / 13;
int rank = card % 13;
```

```
int[] card = {1, 0}; // As de coeur
```

EXEMPLE : CARTES

- pas de correspondance au **métier**
- **couplage fort**
  abstraction / implémentation
- modification **complexe**
  facilité de **corruption** des données
- `PrimitiveObsession+`

```
int card = 13; // As de coeur
int suit = card / 13;
int rank = card % 13;
```

```
int[] card = {1, 0}; // As de coeur
```

EXEMPLE : CARTES

- pas de correspondance au **métier**
- **couplage fort**
  abstraction / implémentation
- modification **complexe**
  facilité de **corruption** des données
- `PrimitiveObsession`+

```
int card = 13; // As de coeur
int suit = card / 13;
int rank = card % 13;
```

```
int[] card = {1, 0}; // As de coeur
```

énumérations

```
public enum Suit { CLUBS, DIAMONDS,
                   SPADES, HEARTS }
public enum Rank { ACE, TWO, …,
                   QUEEN, KING }
```

classe

```
public class Card {
    Suit aSuit;
    Rank aRank;
}
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

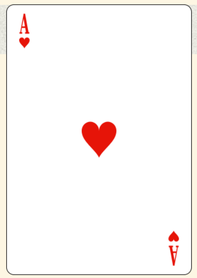
```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```

```
public class Card {
    public Suit aSuit;
    public Rank aRank;
}
public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```

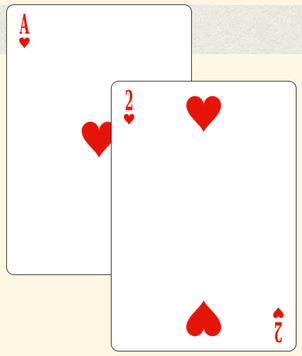


```
public class Card {
    public Suit aSuit;
    public Rank aRank;
}
public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```

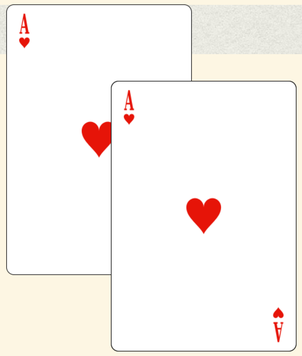


```
public class Card {
    public Suit aSuit;
    public Rank aRank;
}
public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

```
List<Card> deck = new ArrayList<>();
```

```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```



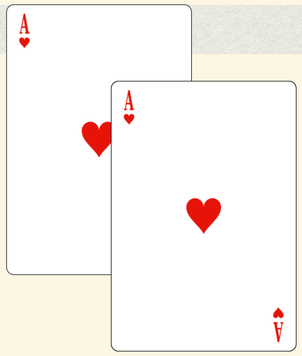
```
public class Card {
    public Suit aSuit;
    public Rank aRank;
}
public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

- encoder les concepts métier (abstractions / types) n'est que la **première** étape !
- attention à l'utilisation de l'information hors de tout principes
- nécessité de **masquage** d'information notion de **scopes** **visibilité** : public/privé/protégé/pack.
- interface (méthodes publiques typiquement) vs implémentation privée

```
List<Card> deck = new ArrayList<>();
```

```
public class Deck {
    List<Card> aCards = new ArrayList<>();
}
```



```
public class Card {
    public Suit aSuit;
    public Rank aRank;
}
public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

```
public class Card {
    private Suit aSuit;
    private Rank aRank;
    public Card(Rank pRank, Suit pSuit) {
        aRank = pRank;
        aSuit = pSuit;
    }
    public Rank getRank() { return aRank; }
    public Suit getSuit() { return aSuit; }
}

public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

EXEMPLE : DECK

- métier = classes / enums : OK
- encapsulation : OK
- masquage information : OK
- interaction contrôlée : OK

```
public class Card {
    private Suit aSuit;
    private Rank aRank;
    public Card(Rank pRank, Suit pSuit) {
        aRank = pRank;
        aSuit = pSuit;
    }
    public Rank getRank() { return aRank; }
    public Suit getSuit() { return aSuit; }
}

public class Deck {
    public List<Card> aCards
                        = new ArrayList<>();
}
```

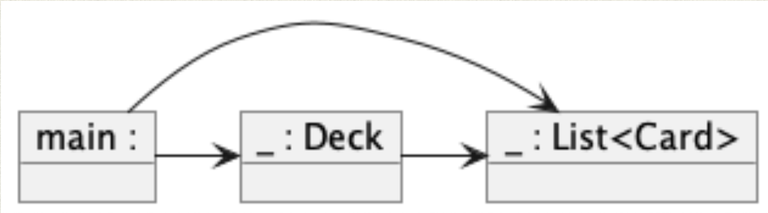
ECHAPPEMENT DE RÉFÉRENCES

```
public class Deck {
    private List<Card> aCards = new ArrayList<>();
    public Deck() { // ajouter 52 cartes + mélanger
    }
    public Card draw() { return aCards.remove(0); }
}
```

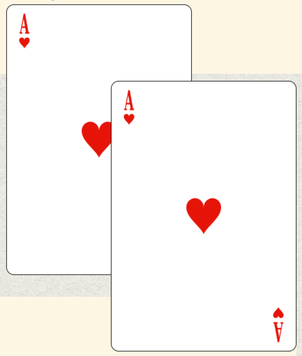
ECHAPPEMENT DE RÉFÉRENCES

```
public class Deck {
    private List<Card> aCards = new ArrayList<>();
    public Deck() { // ajouter 52 cartes + mélanger
    }
    public Card draw() { return aCards.remove(0); }
    public List<Card> getCards() { return aCards; }
}
```

ECHAPPEMENT DE RÉFÉRENCES

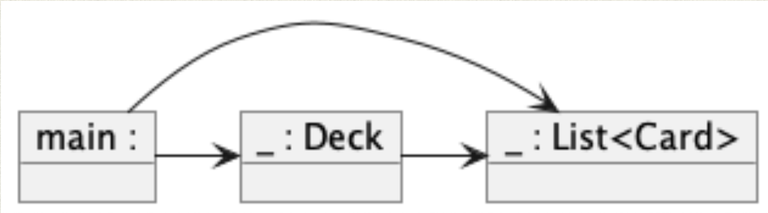



```
public class Deck {
    private List<Card> aCards = new ArrayList<>();
    public Deck() { // ajouter 52 cartes + mélanger
    }
    public Card draw() { return aCards.remove(0); }
    public List<Card> getCards() { return aCards; }
}
```



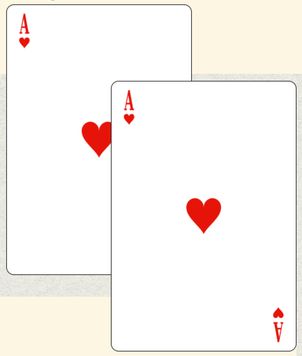
```
Deck deck = new Deck();
List<Card> cards = deck.getCards();
cards.add(new Card(Rank.ACE, Suit.HEARTS));
```

ECHAPPEMENT DE RÉFÉRENCES




```
public class Deck {
    private List<Card> aCards = new ArrayList<>();
    public Deck() { // ajouter 52 cartes + mélanger
    }
    public Card draw() { return aCards.remove(0); }
    public List<Card> getCards() { return aCards; }
}
```

- **échappement** de `aCards`
- on devrait passer par l'interface si on veut modifier les données



```
Deck deck = new Deck();
List<Card> cards = deck.getCards();
cards.add(new Card(Rank.ACE, Suit.HEARTS));
```

ECHAPPEMENT DE RÉFÉRENCES

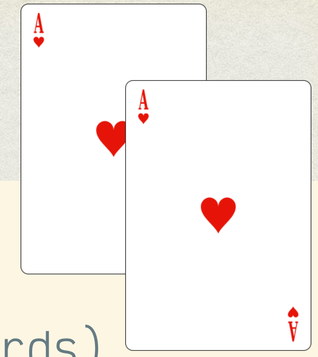
- éviter les getters / setters **automatiques**
  conception de mauvaise qualité
  abstraction non effective
  `InappropriateIntimacy+`

- même problème avec les **références externes**
  utilisées lors de la **création** d'objets
  (idem avec setter)

- et on peut avoir de nombreux niveaux à analyser

ECHAPPEMENT DE RÉFÉRENCES

- éviter les getters / setters **automatiques**
  conception de mauvaise qualité
  abstraction non effective
  `InappropriateIntimacy`+



```
public class Deck {
    private List<Card> aCards;
    public Deck(List<Card> pCards)
                                { aCards = pCards; }
}
```

- même problème avec les **références externes** utilisées lors de la **création** d'objets (idem avec setter)

```
List<Card> cards = new ArrayList<>();
Deck deck = new Deck(cards);
cards.add(new Card(Rank.ACE, Suit.HEARTS));
```

- et on peut avoir de nombreux niveaux à analyser

IMMUTABILITÉ

- on veut éviter de modifier les données sans passer par l'interface
- éviter la fuite de données ... sauf si **immutables**
- en pratique l'immutabilité est souvent **souhaitable**
- **attention** en Java String, (enums), (final)

```
public class Card {
    private Suit aSuit;
    private Rank aRank;
    public Card(Rank pRank, Suit pSuit) {
        aRank = pRank;
        aSuit = pSuit;
    }
    public Rank getRank() { return aRank; }
    public Suit getSuit() { return aSuit; }
}
```

EXPOSER DES DONNÉES

- pas l'implémentation !
- **différentes** solution (avec + et -)
- **extension** de l'interface
- retour de **copies** ou **wrappers** d'immutabilité, voire copies profondes

```
public class Deck {
    private List<Card> aCards = new ArrayList<>();
    public Deck() { ... }
    public Card draw() { return aCards.remove(0); }
    public int size() { return aCards.size(); }
    public Card getCard(int pIndex)
                        { return aCards.get(pIndex); }
}
```

```
public List<Card> getCards() {
    return new ArrayList<>(aCards);
    // ou :
    // return Collections.unmodifiableList(aCards);
}
```

CONTRATS

```
Card boum = new Card(null, Suit.HEARTS);
```

CONTRATS

- un **contrat** peut parfois être intégré à l'interface (typage)

```
Card boum = new Card(null, Suit.HEARTS);
```

- souvent besoin d'aller **au delà** de la simple interface
- programmation / conception par contrats pré-, post-, invariant (+ autres)
- annotations « maison », assert, JML, OCL

CONTRATS

- un **contrat** peut parfois être intégré à l'interface (typage)
- souvent besoin d'aller **au delà** de la simple interface
- programmation / conception par contrats pré-, post-, invariant (+ autres)
- annotations « maison », assert, JML, OCL

```
Card boum = new Card(null, Suit.HEARTS);
```

Java doc

```
/**
 * Constructor for {@link Card}s.
 * @param pSuit the {@link Suit} of …
 * @param pRank the {@link Rank} of …
 * @pre pSuit ≠ null && pRank ≠ null
 */
public Card(Suit pSuit, Rank pRank) {
    assert pSuit ≠ null;
    assert pRank ≠ null;
    aSuit = pSuit;
    aRank = pRank;
}
```

assertions

CONTRATS

- un **contrat** peut parfois être intégré à l'interface (typage)
- souvent besoin d'aller **au delà** de la simple interface
- programmation / conception par contrats pré-, post-, invariant (+ autres)
- annotations « maison », assert, JML, OCL

```
Card boum = new Card(null, Suit.HEARTS);
```

Java doc

```
/**
 * Construc
 * @param p
 * @param p
 * @pre pSu
 */
```

code.cartes.Card.Card(Suit pSuit, Rank pRank)

Constructor for Cards.

- **Parameters:**
  - **pSuit** the Suit of the Card
  - **pRank** the Rank of the Card
- **@pre**
  - pSuit != null && pRank != null

```
public Card(Suit pSuit, Rank pRank) {
    assert pSuit ≠ null;
    assert pRank ≠ null;
    aSuit = pSuit;
    aRank = pRank;
}
```

assertions

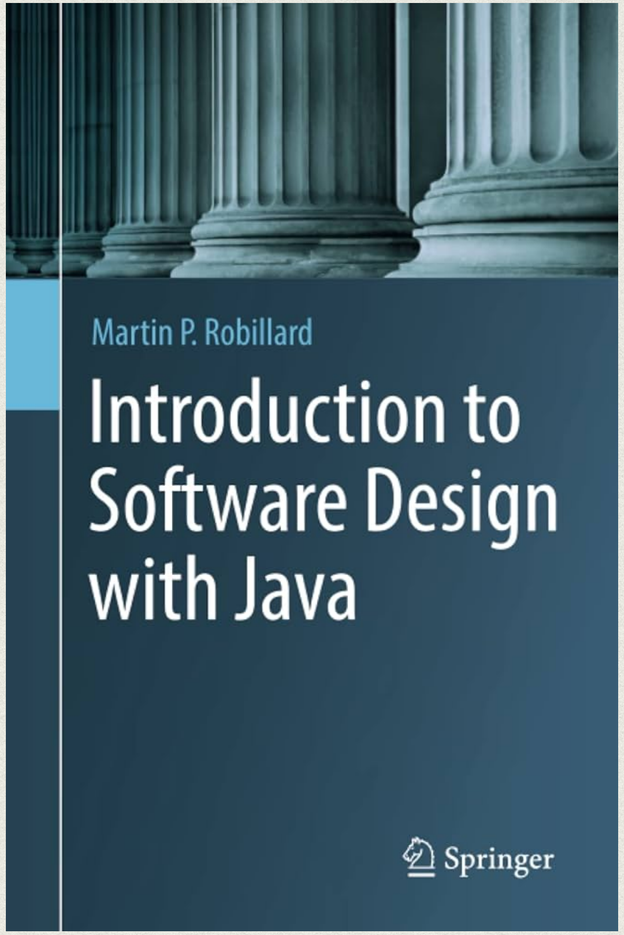
SYNTHÈSE

- concepts métier : **classes**
  éviter `PrimitiveObsession`[+]
- collection d'un petit nombre d'éléments :
  énumération
- **cacher** l'implémentation interne
  attention aux fuites de références
  éviter `InappropriateIntimacy`[+]
- exposer les données
  c'est différent d'exposer l'implémentation
- **immutabilité** si possible
- **contrats** si possible

M1 MIAGE / MAPC
TYPES ET INTERFACES

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

INTERFACE

- on a vu précédemment comment bien définir des classes
  bonne **encapsulation**

- mais comment **inter-agissent** ces classes (les objets de ces classes) ?
  → notion d'**interface**
  attention : le terme est surchargé en informatique

EXEMPLE

```
public class Deck {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    public Card draw() {…}
    public boolean isEmpty() {…}
}

public class Client { private Deck aDeck = new Deck(); … }

public static List<Card> drawCards(Deck pDeck, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pDeck.isEmpty(); i++) {
        result.add(pDeck.draw());
    }
    return result;
}
```

EXEMPLE

- **couplage fort**
  Client → Deck

- dépendance
  à une implémentation (Deck)
  plutôt qu'à une abstraction

- il peut y avoir
  plusieurs façons
  de gérer et tirer des cartes

```
public class Deck {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    public Card draw() {…}
    public boolean isEmpty() {…}
}

public class Client { private Deck aDeck = new Deck(); … }

public static List<Card> drawCards(Deck pDeck, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pDeck.isEmpty(); i++) {
        result.add(pDeck.draw());
    }
    return result;
}
```

EXEMPLE CORRIGÉ

```
public interface CardSource {
    Card draw(); // pre: !isEmpty(), return: next card
    boolean isEmpty(); // return: true if no card in the source
}
public class Deck implements CardSource {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    @Override public Card draw() {…}
    @Override public boolean isEmpty() {…}
}
public class Client { private CardSource aSource = new Deck(); …}

public static List<Card> drawCards(CardSource pSource, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pSource.isEmpty(); i++) {
        result.add(pSource.draw());
    }
    return result;
}
```

EXEMPLE CORRIGÉ

- définition d'une **interface**
  couplage faible
  Client → CardSource

- permet :
  vérification de typage
  (classe ⟷ interface ⟷ client)
  polymorphisme
  et extensibilité
  (CardSource prend ≠ formes)

- penser à :
  commentaires et contrats

```
public interface CardSource {
    Card draw(); // pre: !isEmpty(), return: next card
    boolean isEmpty(); // return: true if no card in the source
}
public class Deck implements CardSource {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    @Override public Card draw() {…}
    @Override public boolean isEmpty() {…}
}
public class Client { private CardSource aSource = new Deck(); …}

public static List<Card> drawCards(CardSource pSource, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pSource.isEmpty(); i++) {
        result.add(pSource.draw());
    }
    return result;
}
```

contrats

ITERATOR

ITERATOR

- comment donner **accès** à une collection
  en **respectant**
  encapsulation et masquage d'information ?

- on a vu des solutions
  (cf cours encapsulation)

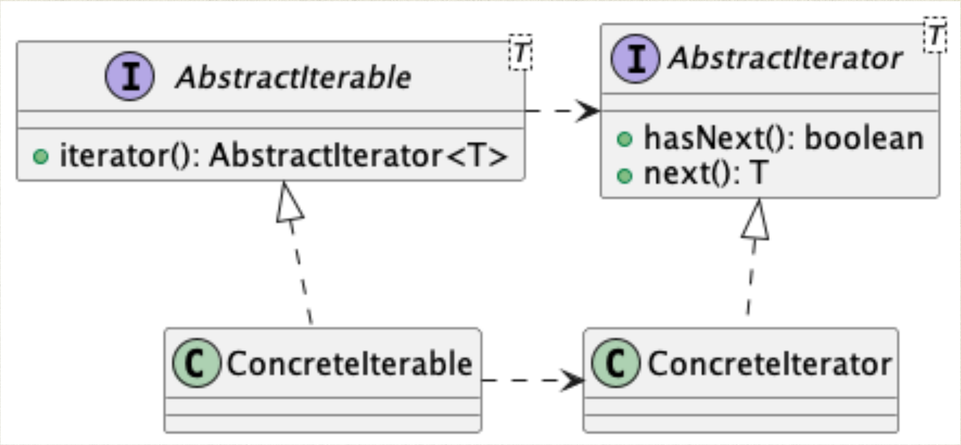
- autre solution, se demander quel est le **besoin**
  itérer sur les éléments ? patron **Iterator**

ITERATOR

- comment donner **accès** à une collection
  en **respectant**
  encapsulation et masquage d'information ?

- on a vu des solutions
  (cf cours encapsulation)

- autre solution, se demander quel est le **besoin**
  itérer sur les éléments ? patron **Iterator**



ITERATOR

- comment donner **accès** à une collection en **respectant** encapsulation et masquage d'information ?

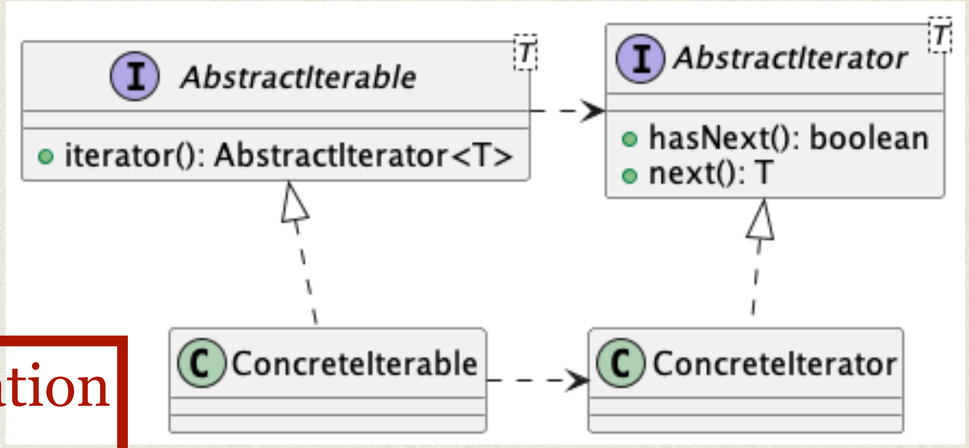
- on a vu des solutions
  (cf cours encapsulation)

- autre solution, se demander quel est le **besoin** itérer sur les éléments ? patron **Iterator**

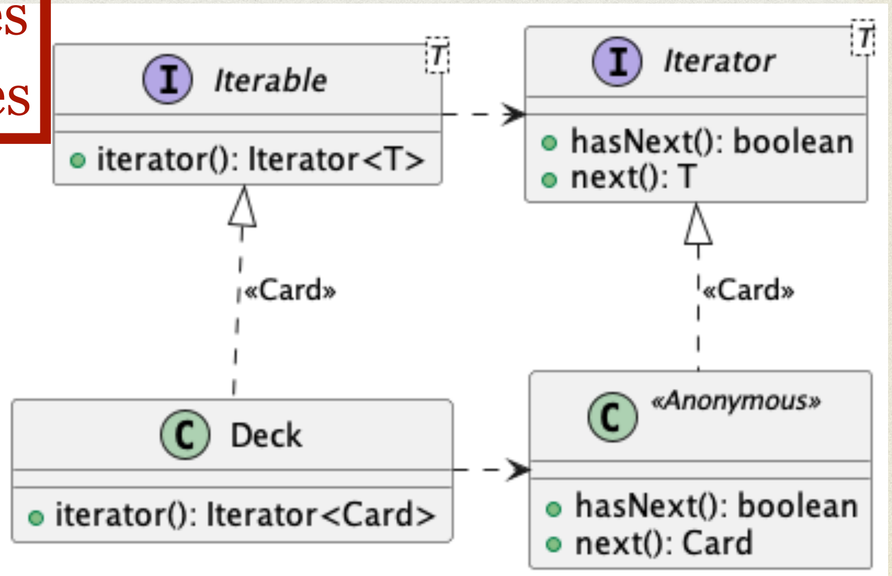
```
@Override
public Iterator<Card> iterator() { return aCards.iterator(); };
```

```
Iterable<Card> d = new Deck();
for(Card c: d) { System.out.println(c); }
```

bénéfice syntaxe dédiée



utilisation interfaces existantes



PÉRIMÈTRE D'INTERFACE

```
public interface CardSource {
    Card draw();        // pre: !isEmpty(), return: next card
    boolean isEmpty(); // return: true if no card in the source
}
public class Deck implements CardSource {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    public Iterator<Card> iterator() {…}
    @Override public Card draw() {…}
    @Override public boolean isEmpty() {…}
}
public class Client { private CardSource aSource = new Deck(); …}

public static List<Card> drawCards(CardSource pSource, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pSource.isEmpty(); i++) {
        result.add(pSource.draw()); }
    return result; }
```

PÉRIMÈTRE D'INTERFACE

- attention au **périmètre** lors de la définition des interfaces

- **cohérence** : `shuffle` et `iterator` vont-t-ils « **avec** » `draw` et `isEmpty` ?

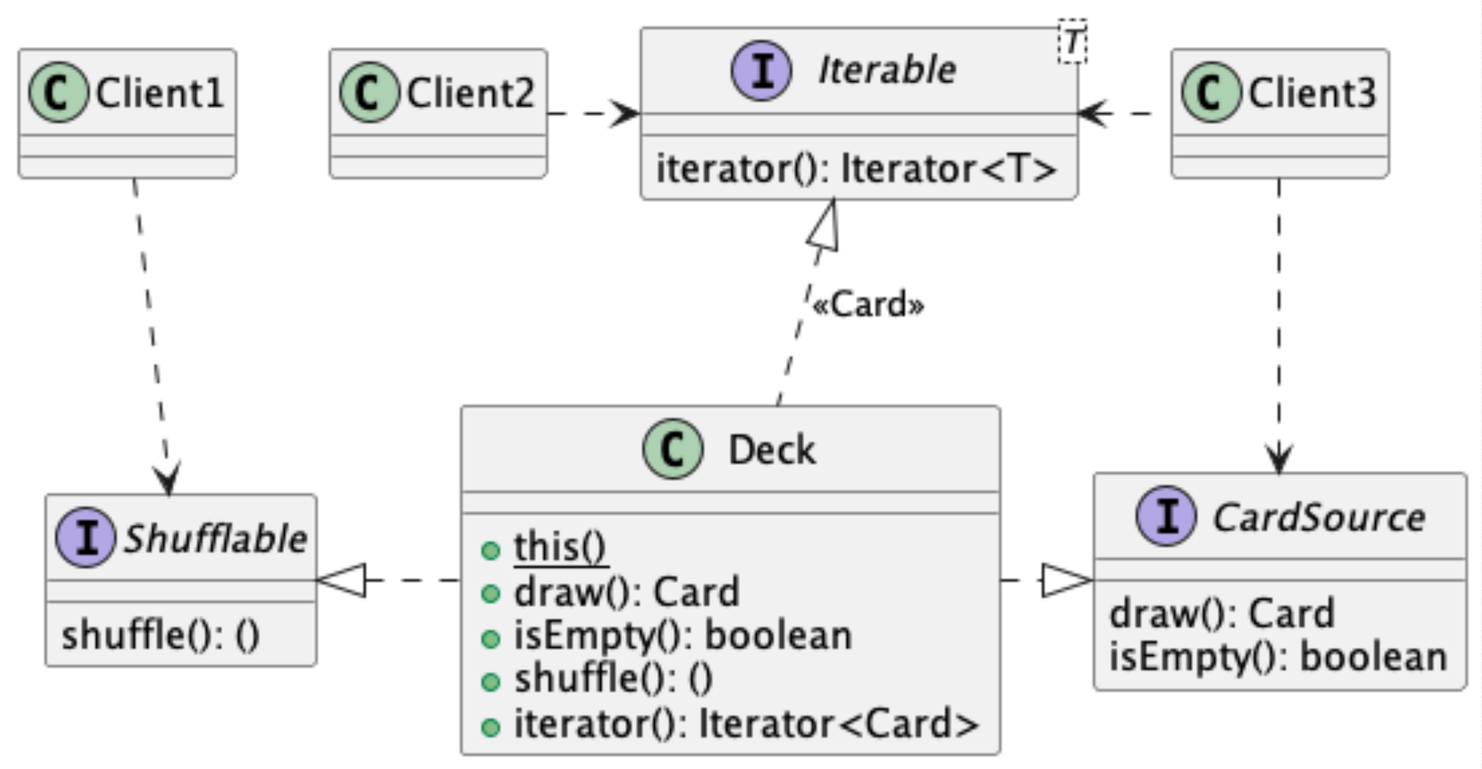
```
public interface CardSource {
    Card draw();        // pre: !isEmpty(), return: next card
    boolean isEmpty(); // return: true if no card in the source
}
public class Deck implements CardSource {
    private List<Card> aCards;
    public Deck() {…}
    public void shuffle() {…}
    public Iterator<Card> iterator() {…}
    @Override public Card draw() {…}
    @Override public boolean isEmpty() {…}
}
public class Client { private CardSource aSource = new Deck(); …}

public static List<Card> drawCards(CardSource pSource, int pNumber) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < pNumber && !pSource.isEmpty(); i++) {
        result.add(pSource.draw()); }
    return result; }
```

PÉRIMÈTRE D'INTERFACE

- principe **ISP** de SOLID
- dépendre **uniquement**
  des services
  dont on a **besoin**
- une implémentation
  peut réaliser
  plusieurs contrats
- promouvoir la **séparation des préoccupations**

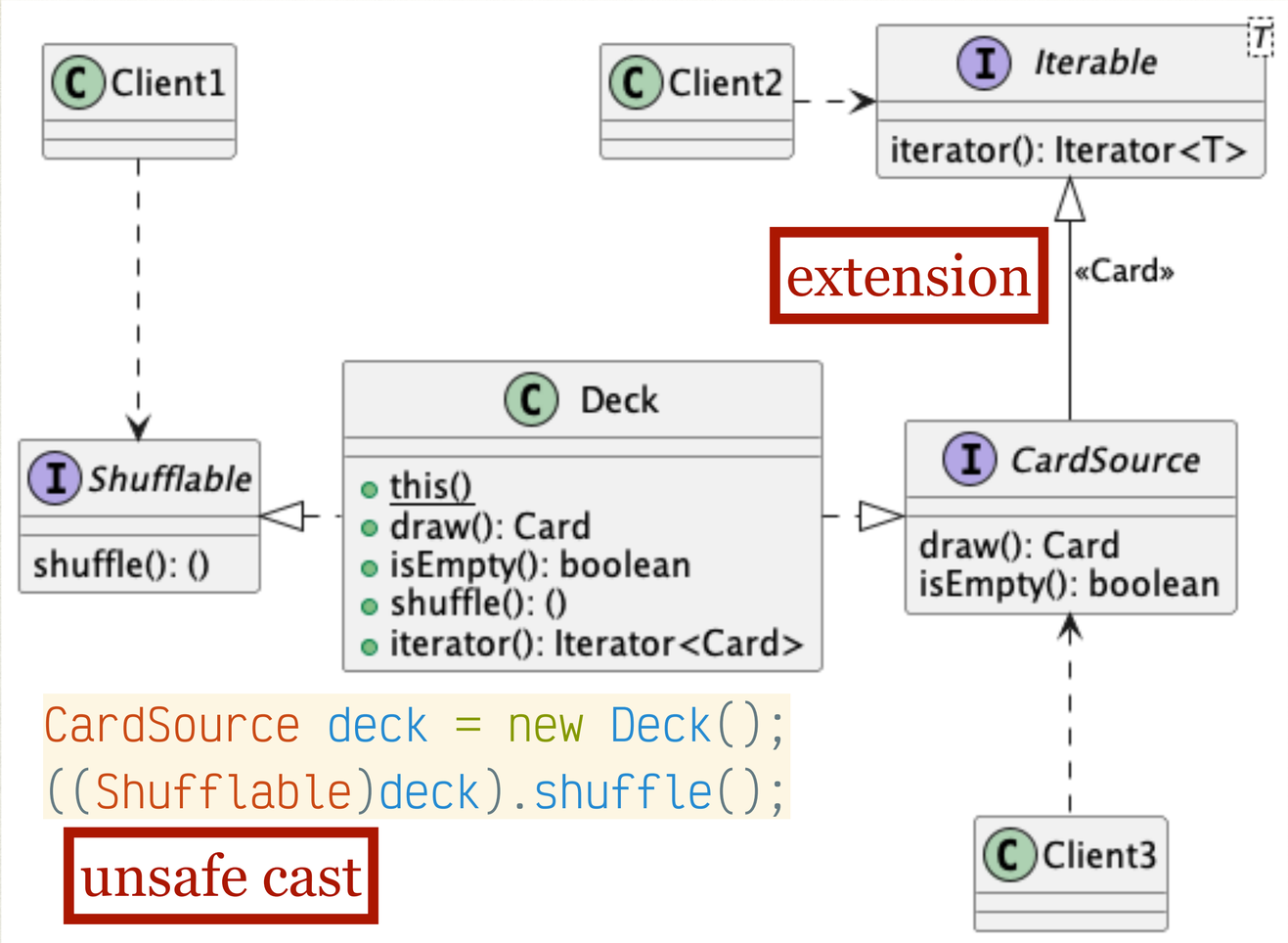
PÉRIMÈTRE D'INTERFACE

- principe **ISP** de SOLID
- dépendre **uniquement** des services dont on a **besoin**
- une implémentation peut réaliser **plusieurs** contrats
- promouvoir la **séparation des préoccupations**



EXTENSION D'INTERFACE

- on peut **étendre** les interfaces
- dépend de l'usage par les clients
  utilisation B ⇒ utilisation A ?
  alors **extension B-⊳A**
- **problème** :
  ici si on veut mélanger puis tirer
  on risque un **unsafe cast**
  → parfois revenir sur choix ISP



COMPARAISON ET TRI (V1)

- supposons que l'on veuille **trier** les cartes d'un deck

COMPARAISON ET TRI (V1)

- supposons que l'on veuille **trier** les cartes d'un deck

```
public void sort() {
    Collections.sort(aCards);
}
```

COMPARAISON ET TRI (V1)

- supposons que l'on veuille **trier** les cartes d'un deck

```
public void sort() {
    Collections.sort(aCards);
}
```

```
The method sort(List<T>) in the type Collections
is not applicable for the arguments
(List<Card>) Java(67108979)

<T> void java.util.Collections.sort(List<T> list)
```

COMPARAISON ET TRI (V1)

- supposons que l'on veuille **trier** les cartes d'un deck

```
public void sort() {
    Collections.sort(aCards);
}
```

```
The method sort(List<T>) in the type Collections
is not applicable for the arguments
(List<Card>) Java(67108979)

<T> void java.util.Collections.sort(List<T> list)
```

- **contrat** : les cartes doivent être **Comparables**

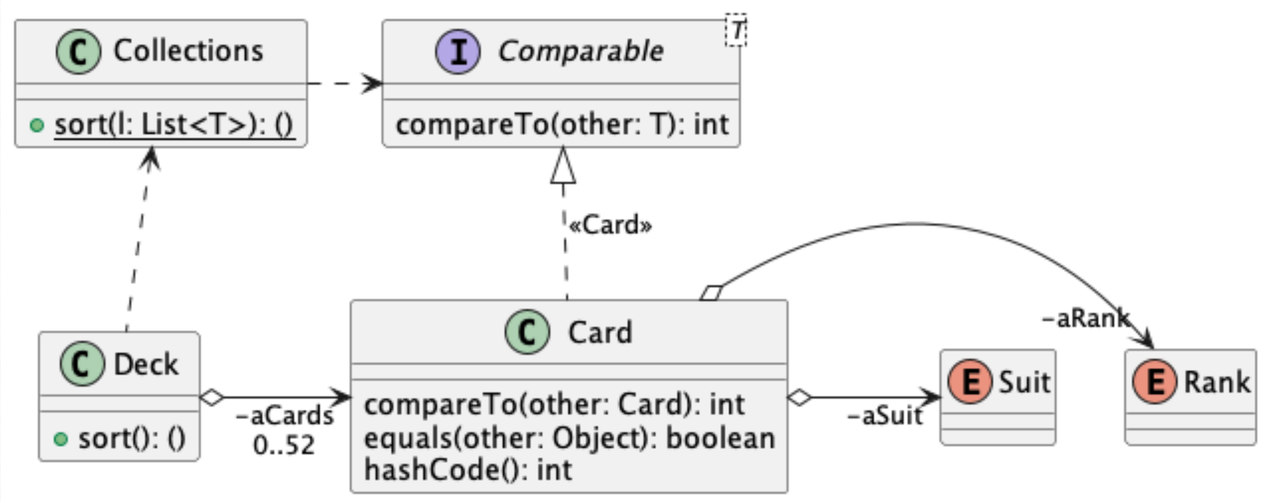
COMPARAISON ET TRI (V1)

- supposons que l'on veuille **trier** les cartes d'un deck

```
public void sort() {
    Collections.sort(aCards);
}
```

```
The method sort(List<T>) in the type Collections
is not applicable for the arguments
(List<Card>) Java(67108979)

<T> void java.util.Collections.sort(List<T> list)
```

- **contrat** : les cartes doivent être **Comparables**




```
@Override
public int compareTo(Card o) {
    return this.aRank.ordinal() - o.aRank.ordinal();
}
```

COMPARAISON ET TRI (V1)

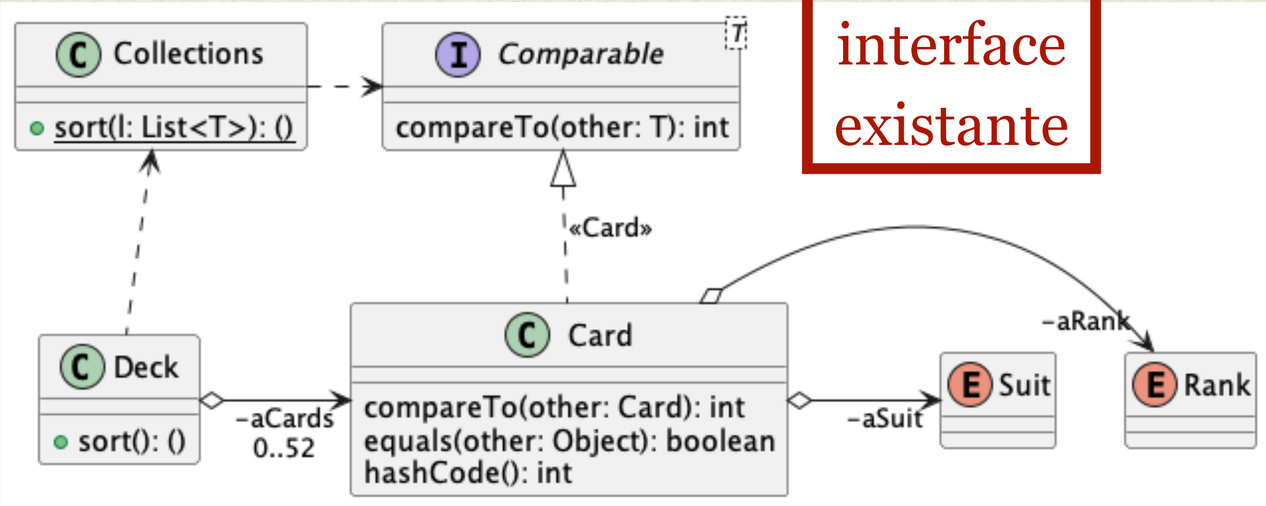
- supposons que l'on veuille **trier** les cartes d'un deck

```
public void sort() {
    Collections.sort(aCards);
}
```

```
The method sort(List<T>) in the type Collections
is not applicable for the arguments
(List<Card>) Java(67108979)

<T> void java.util.Collections.sort(List<T> list)
```

- **contrat** : les cartes doivent être **Comparables**

utilisation interface existante



attention : lien equals/hashCode/compareTo

```
@Override
public int compareTo(Card o) {
    return this.aRank.ordinal() - o.aRank.ordinal();
}
```

INTERLUDE : RÈGLES

- bien **lire la documentation des API que l'on utilise**
  notons
  `o1==o2` ⟷ `o1.equals(o2)`, `#o` ⟷ `o.hashCode()`,
  `o1<o2` ⟷ `o1.compareTo(o2)<0`, `o1>o2` ⟷ `o1.compareTo(o2)>0`, `o1~o2` ⟷ `o1.compareTo(o2)=0`

- **requis** (entre autres) :
  (=1) `o==o`
  (=2) `o1==o2` ⇒ `o2==o1`
  (=3) `o1==o2` ∧ `o2==o3` ⇒ `o1==o3`
  (#1) `o1==o2` ⇒ `#o1 = #o2`
  (<1) `o1<o2` ⟺ `o2>o1`
  (<2) `o1<o2` ∧ `o2<o3` ⇒ `o1<o3`

- **non requis** :
  `not(o1==o2)` ⇒ `not(#o1=#o2)`

  recommandé mais non requis (documenter) :
  `o1~o2` ⟺ `o1==o2`

COMPARAISON ET TRI (V2)

- et si on a plusieurs façons de trier les cartes ?

  par rang (A 2 3 … 10 V D R)
  par rang (2 3 … 10 V D R A)
  par couleur
  belote atout (7 8 D R 10 A 9 V)
  belote hors atout (7 8 9 V D R 10 A)

- utiliser un **flag** dans `Card` ?
  (paramètre le choix de tri)

COMPARAISON ET TRI (V2)

- et si on a plusieurs façons de trier les cartes ?

  par rang (A 2 3 … 10 V D R)
  par rang (2 3 … 10 V D R A)
  par couleur
  belote atout (7 8 D R 10 A 9 V)
  belote hors atout (7 8 9 V D R 10 A)

- utiliser un **flag** dans `Card` ? (paramètre le choix de tri)

```java
public class Card implements Comparable<Card> {
    private Suit aSuit;
    private Rank aRank;
    private CompareMode compareMode = CompareMode.RANK;

    public enum CompareMode { RANK, SUIT };

    public void setCompareMode(CompareMode compareMode) {
        this.compareMode = compareMode; }
    @Override
    public int compareTo(Card o) {
        return switch(this.compareMode) {
            case RANK → this.aRank.ordinal()-o.aRank.ordinal();
            case SUIT → this.aSuit.ordinal()-o.aSuit.ordinal();
        };
    }
}
```

COMPARAISON ET TRI (V2)

- et si on a plusieurs façons de trier les cartes ?

  par rang (A 2 3 … 10 V D R)
  par rang (2 3 … 10 V D R A)
  par couleur
  belote atout (7 8 D R 10 A 9 V)
  belote hors atout (7 8 9 V D R 10 A)

- utiliser un **flag** dans `Card` ? (paramètre le choix de tri)

```
public class Card implements Comparable<Card> {
    private Suit aSuit;
    private Rank aRank;
    private CompareMode compareMode = CompareMode.RANK;

    public enum CompareMode { RANK, SUIT };

    public void setCompareMode(CompareMode compareMode) {
        this.compareMode = compareMode; }
    @Override
    public int compareTo(Card o) {
        return switch(this.compareMode) {
            case RANK → this.aRank.ordinal()-o.aRank.ordinal();
            case SUIT → this.aSuit.ordinal()-o.aSuit.ordinal();
        };
    }
}
```

perte immutabilité

Switch Statement†

COMPARAISON ET TRI (V2)

- déplacer la **responsabilité** de la comparaison
- un **Comparator** par tri
- différents choix de placement ici choix dans `Deck`

```
public class Deck … {
    private List<Card> aCards;

    public Deck() { … }

    public static final Comparator<Card> c1 = (c1, c2) → …;
    public static final Comparator<Card> c2 = (c1, c2) → …;

    public void sort(int choix) {
        switch (choix) {
            case 0 → Collections.sort(aCards, c1);
            case 1 → Collections.sort(aCards, c2);
            default → Collections.sort(aCards, c2);
        }
    }
}
```

function objects
functional interfaces

Switch Statement†

COMPARAISON ET TRI (V2)

- est-ce le meilleur choix ?
  si `Card` a des attributs privés ?

- création de Comparators
  dans `Card` à l'aide de
  factory methods
  (= méthodes de fabrication
  d'objets)

COMPARAISON ET TRI (V2)

factory methods

- est-ce le meilleur choix ?
  si `Card` a des attributs privés ?

- création de Comparators dans `Card` à l'aide de **factory methods** (= méthodes de fabrication d'objets)

```
public class Card {
    …
    public static Comparator<Card> rankComparator()
        { return (card1, card2) → …; }
    public static Comparator<Card> suitComparator()
        { return (card1, card2) → …; }
}

public class Deck … {
    …
    public void sort(int choix) {
        switch (choix) {
            case 0 → Collections.sort(aCards,
                                      Card.rankComparator());
            case 1 → Collections.sort(aCards,
                                      Card.suitComparator());
            default → Collections.sort(aCards, …);
        }}
}
```

COMPARAISON ET TRI (V2)

factory methods

- est-ce le meilleur choix ?
  si `Card` a des attributs privés ?

- création de Comparators dans `Card` à l'aide de **factory methods** (= méthodes de fabrication d'objets)

```
public class Card {
    …
    public static Comparator<Card> rankComparator()
        { return (card1, card2) → …; }
    public static Comparator<Card> suitComparator()
        { return (card1, card2) → …; }
}
```

Switch Statement†

```
public class Deck … {
    …
    public void sort(int choix) {
        switch (choix) {
            case 0 → Collections.sort(aCards,
                                      Card.rankComparator());
            case 1 → Collections.sort(aCards,
                                      Card.suitComparator());
            default → Collections.sort(aCards, …);
        }}
}
```

TEMPLATE METHOD

- Comment éviter le Switch Statement ?

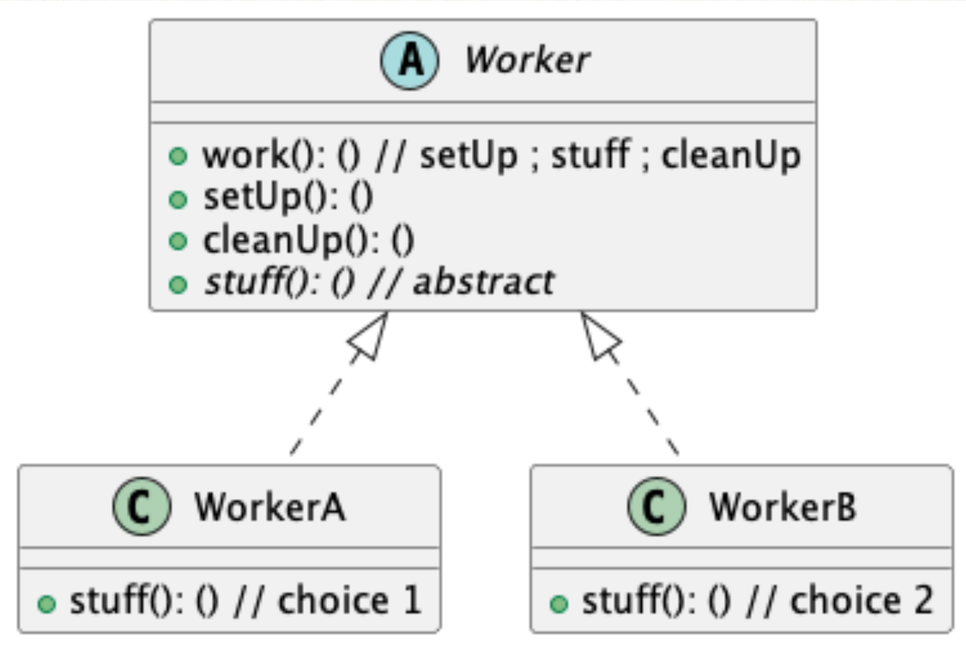
```
public class Worker {
    public void work(int choice) {
        setUp();
        switch (choice) {
            case 0: choice1(); break;
            case 1: choice2(); break;
            default: defaultChoice();
        }
        cleanUp();
    }
    public void setUp() {}
    public void cleanUp() {}
    public void defaultChoice() {}
    public void choice1() {}
    public void choice2() {}
}

Worker work1 = new Worker();
work1.work(0);
```

Switch Statement†

Magic Number†

TEMPLATE METHOD

- Comment éviter le Switch Statement ?




```
Worker work1 = new WorkerA();
work1.work();
```

```
public class Worker {
    public void work(int choice) {
        setUp();
        switch (choice) {
            case 0: choice1(); break;
            case 1: choice2(); break;
            default: defaultChoice();
        }
        cleanUp();
    }
    public void setUp() {}
    public void cleanUp() {}
    public void defaultChoice() {}
    public void choice1() {}
    public void choice2() {}
}
```

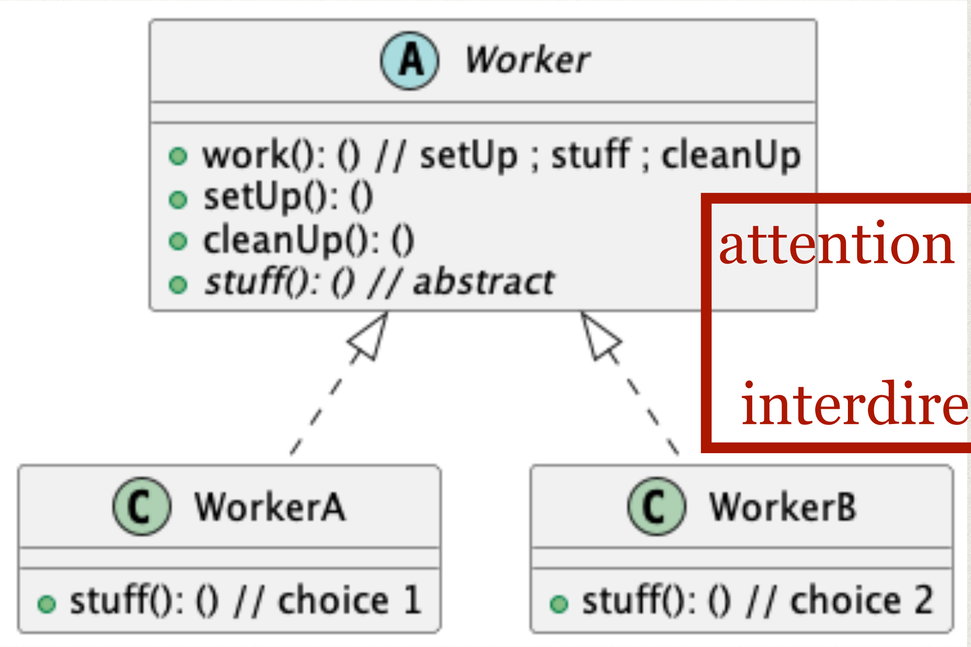
Switch Statement†

```
Worker work1 = new Worker();
work1.work(0);
```

Magic Number†

TEMPLATE METHOD

- Comment éviter le Switch Statement ?



attention à documenter
ou
interdire le masquage

```
Worker work1 = new WorkerA();
work1.work();
```

```
public class Worker {
    public void work(int choice) {
        setUp();
        switch (choice) {
            case 0: choice1(); break;
            case 1: choice2(); break;
            default: defaultChoice();
        }
        cleanUp();
    }
    public void setUp() {}
    public void cleanUp() {}
    public void defaultChoice() {}
    public void choice1() {}
    public void choice2() {}
}
```

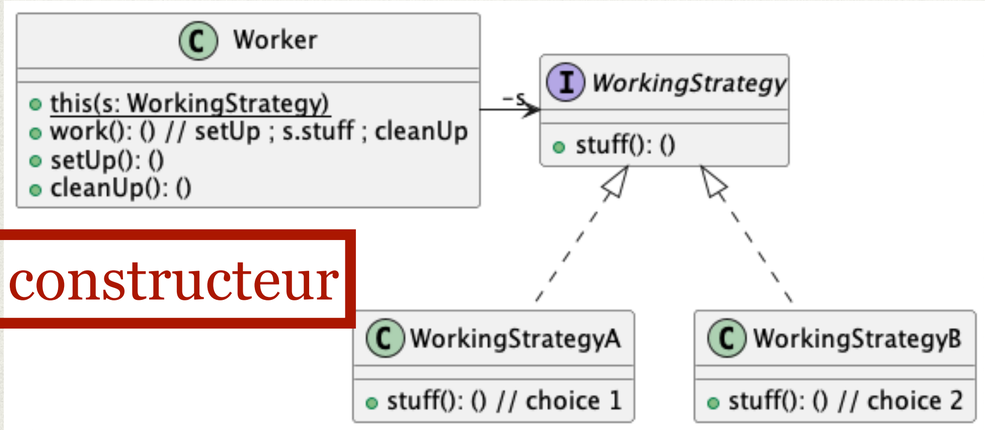
Switch Statement†

```
Worker work1 = new Worker();
work1.work(0);
```

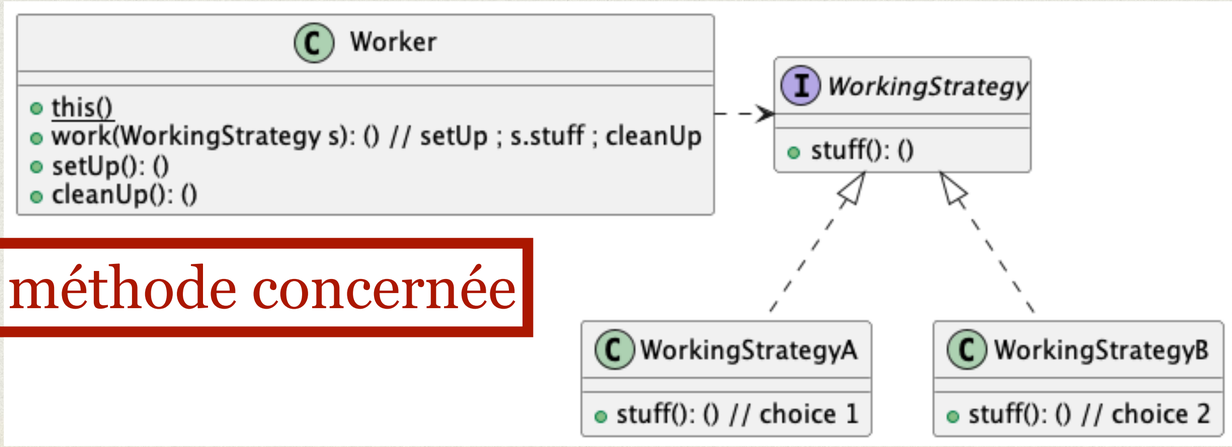
Magic Number†

STRATEGY

- Comment éviter le Switch Statement ?



stratégie et constructeur



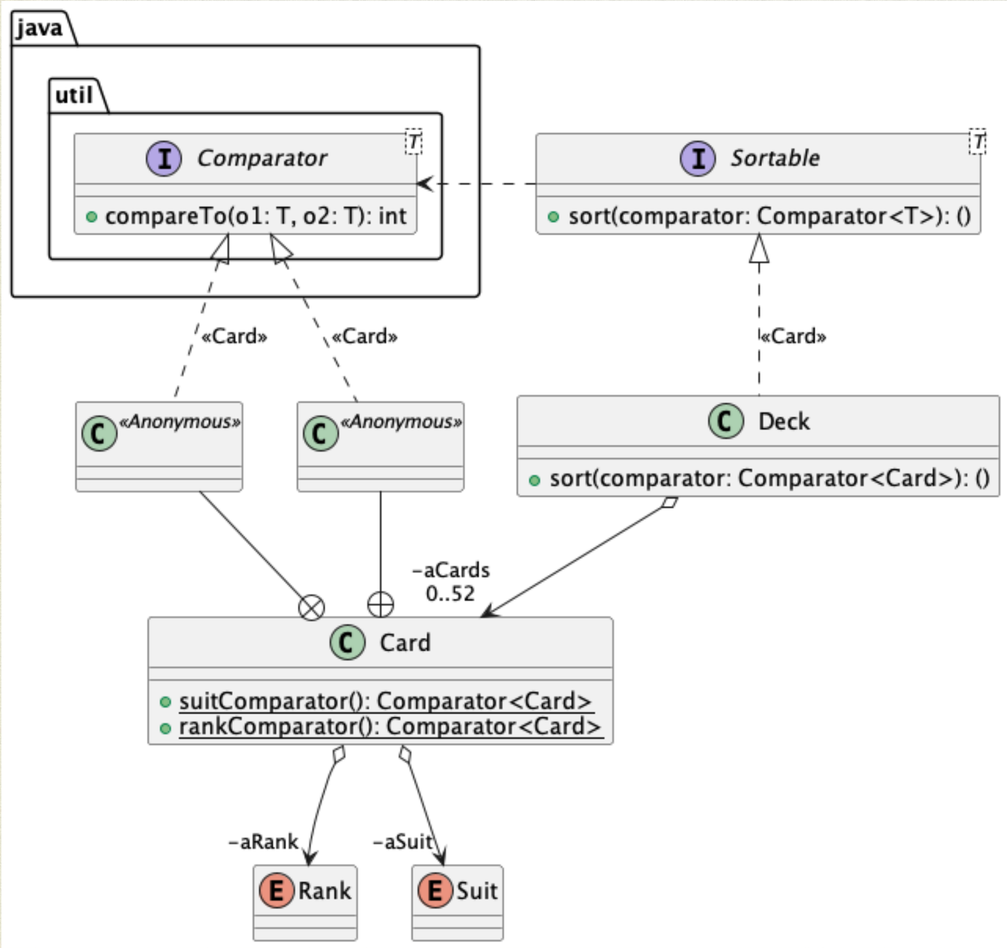
stratégie et méthode concernée

Switch Statement†

```
public class Worker {
    public void work(int choice) {
        setUp();
        switch (choice) {
            case 0: choice1(); break;
            case 1: choice2(); break;
            default: defaultChoice();
        }
        cleanUp();
    }
    public void setUp() {}
    public void cleanUp() {}
    public void defaultChoice() {}
    public void choice1() {}
    public void choice2() {}
}

Worker work1 = new Worker();
work1.work(0);
```

Magic Number†

STRATEGY (APPLICATION)

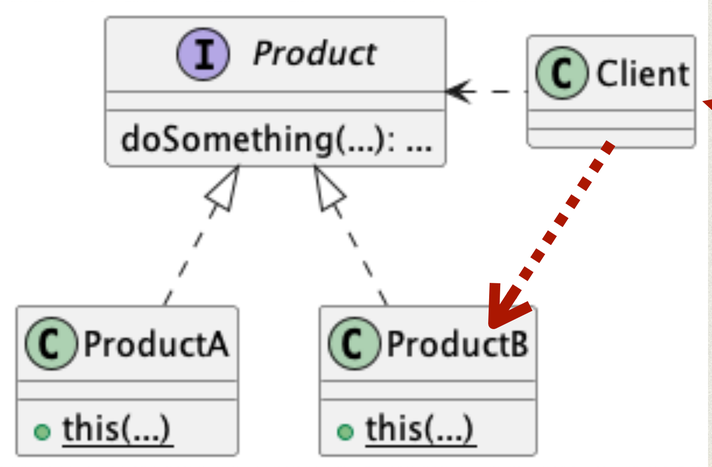



```
public interface Sortable<T> {
    void sort(Comparator<T> comparator);
}

public class Deck implements Sortable<Card>, … {
    private List<Card> aCards;
    public Deck() { … }
    …
    @Override public void sort(Comparator<Card> comparator) {
            Collections.sort(aCards, comparator);
    }
}

Sortable<Card> d = new Deck();
d.sort(Card.rankComparator());
```

CRÉATION D'OBJETS

- le client peut dépendre d'interfaces
  mais aussi (en même temps) d'**implémentations**

- comment améliorer ce **couplage** ?



constructeur à accès public

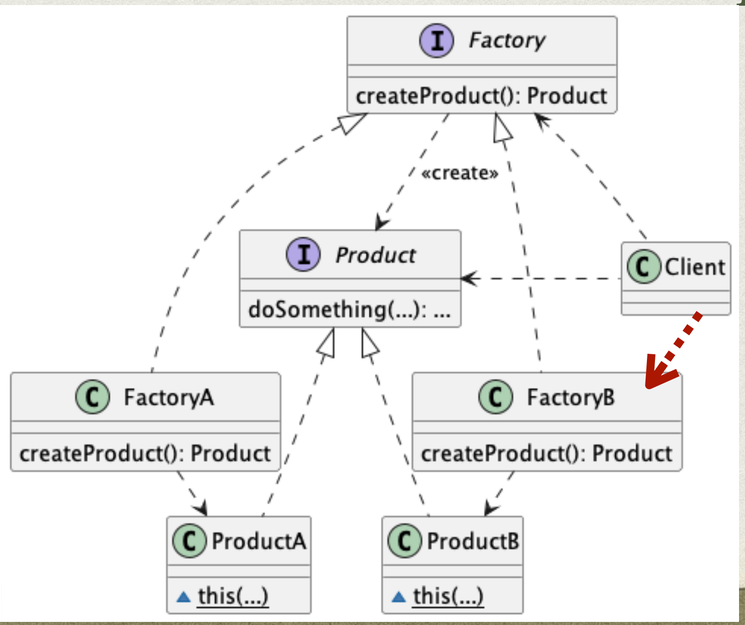
```
public class Deck implements CardSource {
    public Deck() {…}
}
```

couplage client / produit concret

```
// accès direct au constructeur
CardSource source = new Deck();
Card carte = source.draw();
```

FACTORY METHOD

- on a vu des factory methods
- il y a aussi un patron **Factory Method**
  interface de création d'objets
  + sous classes raffinant ce qui est créé




```
public interface CardSourceCreator {
    CardSource createCardSource();
}
```

interface factory

factory spécialisée

```
public class Croupier
                        implements CardSourceCreator {
    @Override public CardSource createCardSource() {
        return new Deck();
    }
}
```

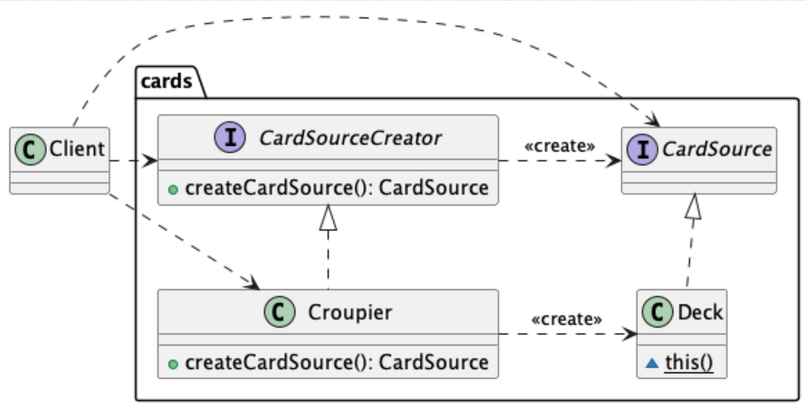
constructeur à accès limité

```
public class Deck implements CardSource {
   Deck() {…}
}
```

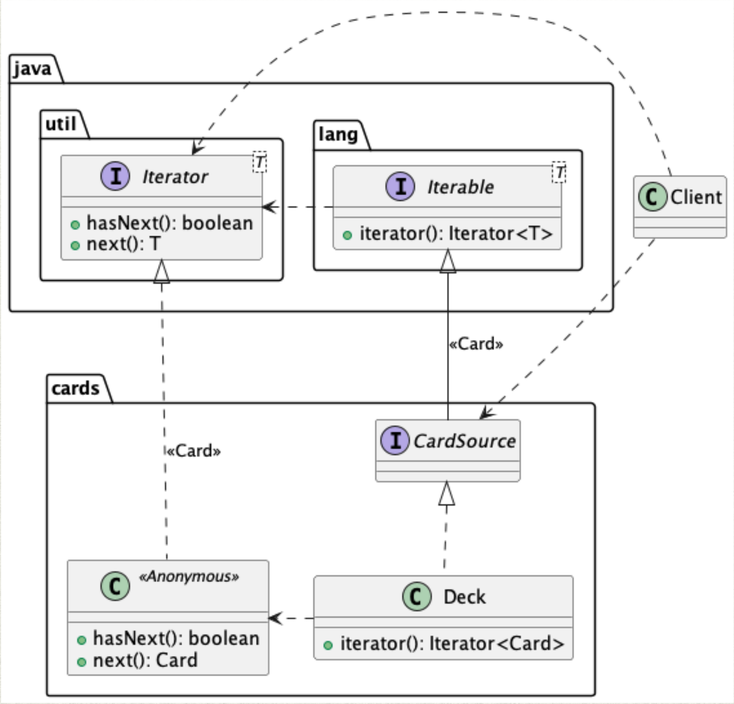
découplage client / produit concret

```
// illustration factory
CardSourceCreator creator = new Croupier();
CardSource source = creator.createCardSource();
Card carte = source.draw();
```

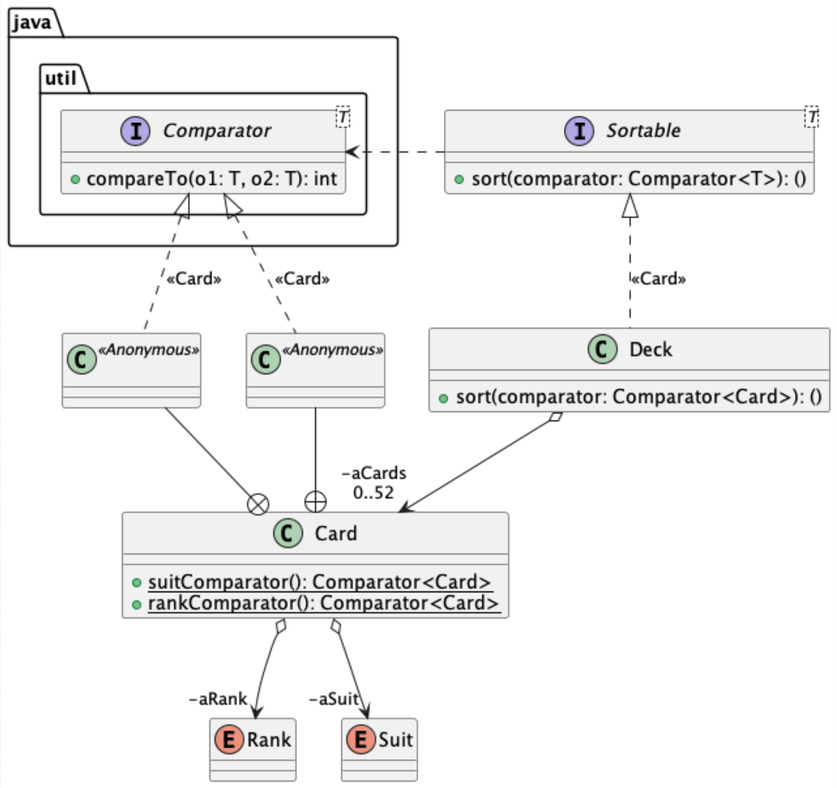
SYNTHÈSE



Factory Method



Iterator



Strategy

SYNTHÈSE

- **découpler** clients / implémentations avec des **interfaces**
  appliquer **ISP** (+ sous-typage interfaces)
  penser aux interfaces disponibles (*-able)

- patron **Iterator**
  exposer des collections en respectant l'encapsulation

- attention au **Switch Statement**[+]

- patron **Template Method**
  algorithme avec étapes particulières raffinées, choix statique

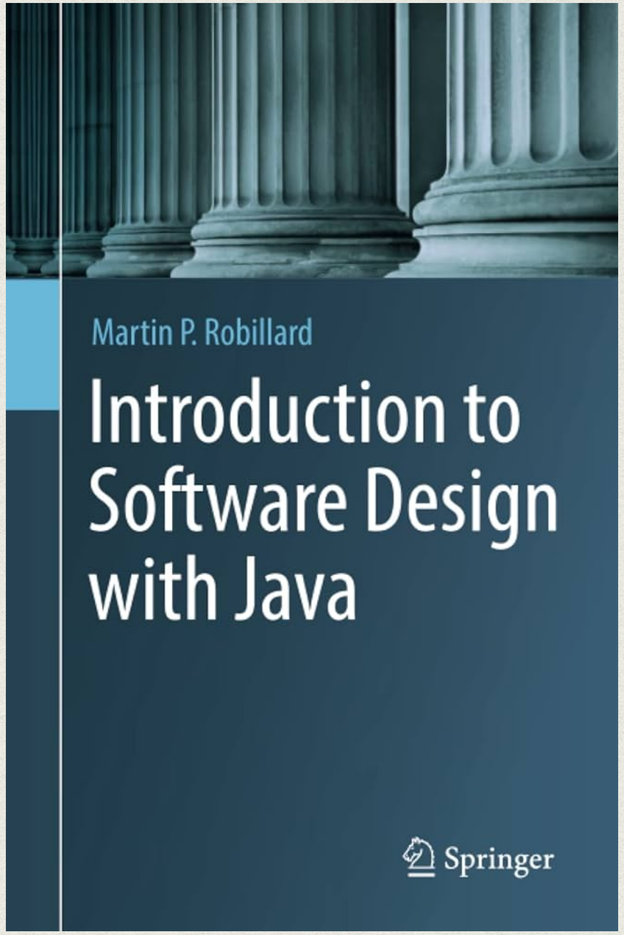
- patron **Strategy**
  dépendance à une famille d'algorithmes, choix dynamique

- patron **Factory Method**
  interface de création d'objets (+ factory methods en général)

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

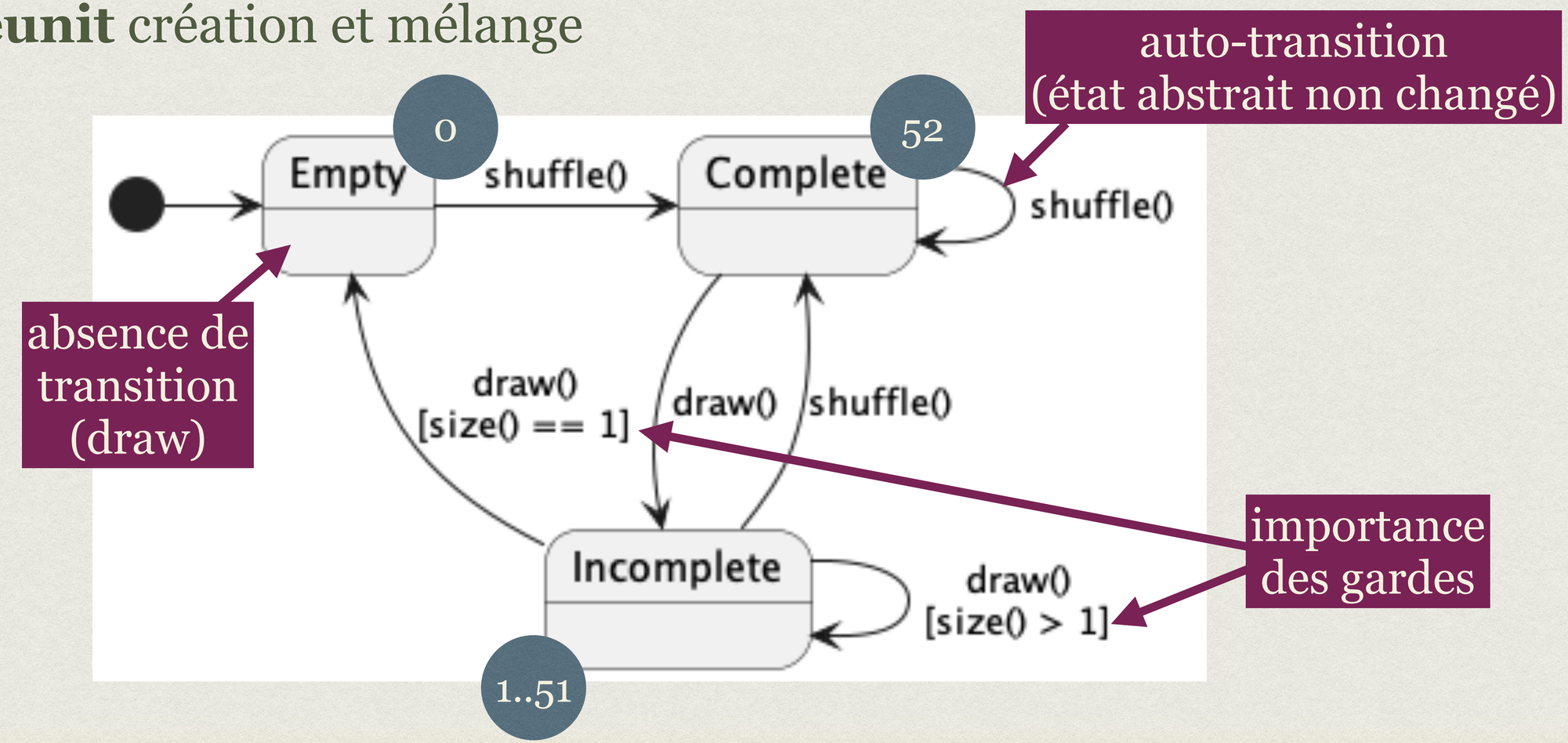
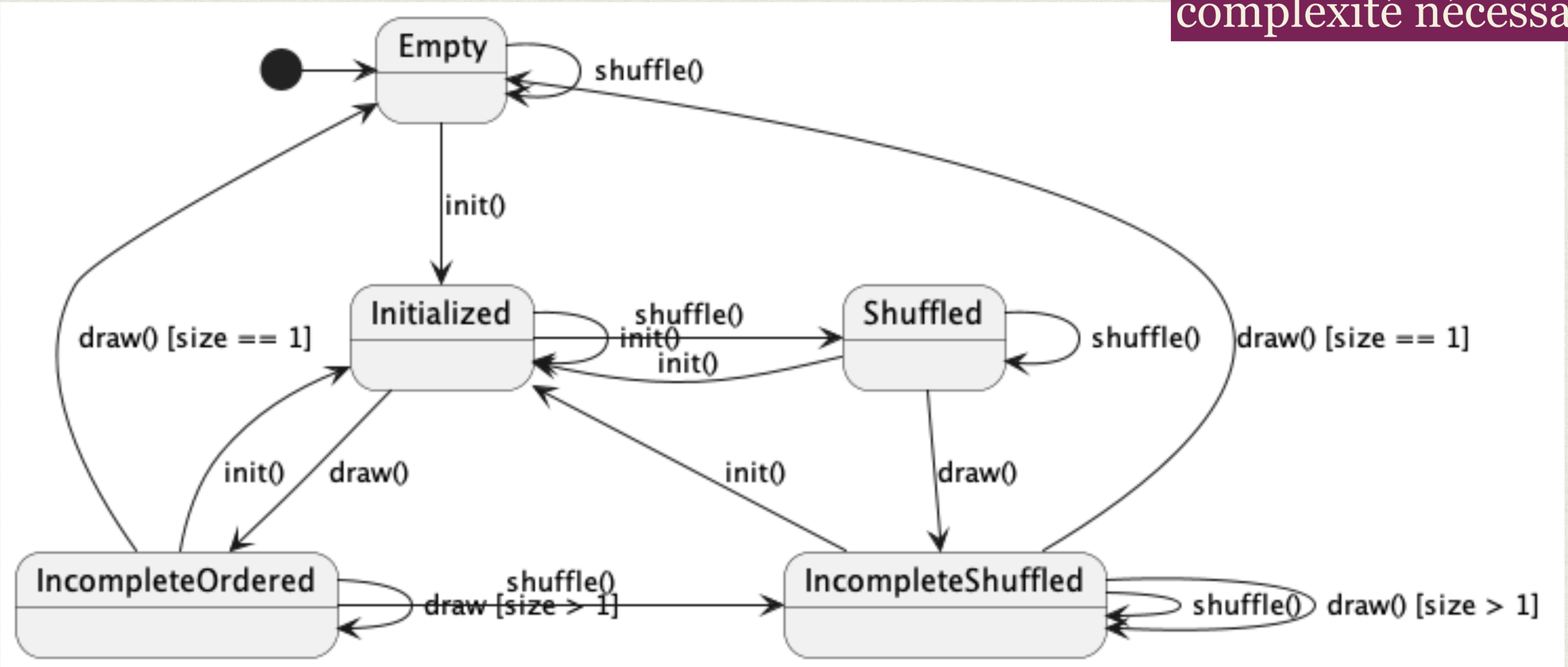


DIAGRAMME D'ÉTATS DU DECK

- 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

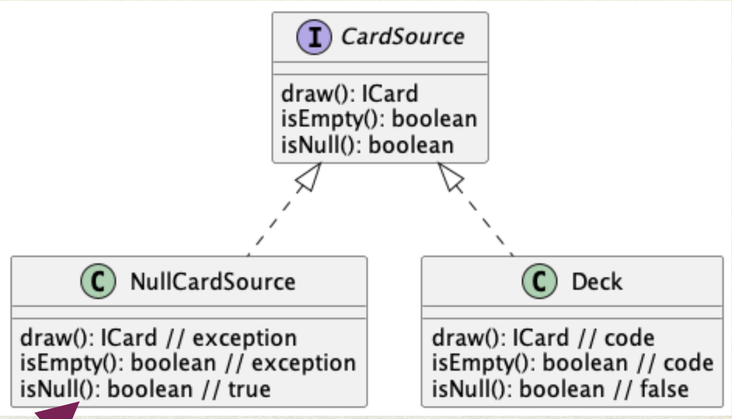
- type `Option[T]` et API associée (`empty`, `isPresent`, `get`, ...)
  ex : Optional<T> en Java, Option<T> en Java+vavr

- 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+`

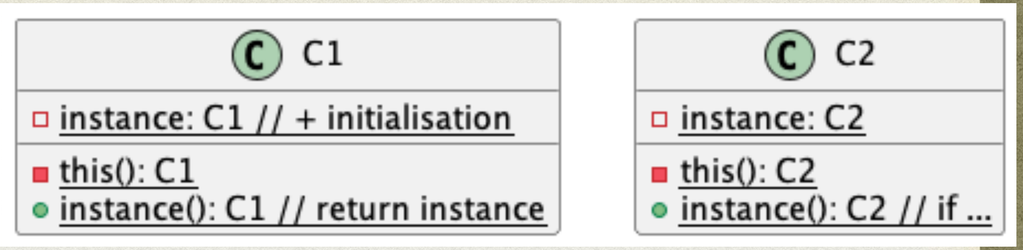
IDENTITÉ ET EGALITÉ

- 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 :

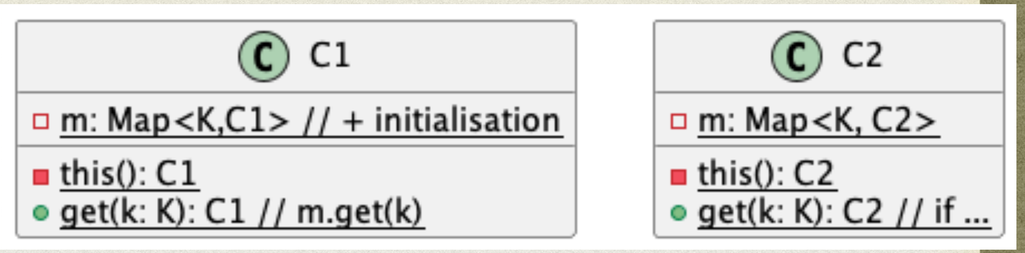
  [X] **getter statique**

  penser à la synchronisation :

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

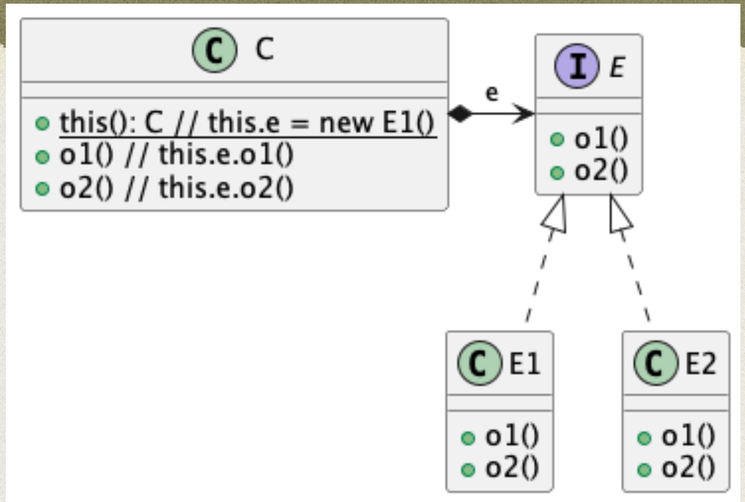
FLYWEIGHT

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



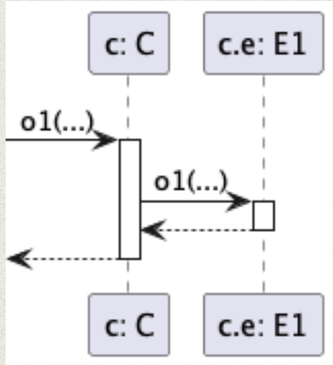
- **check-list pour une classe C FLYWEIGHT :**
  [X] **constructeur privé**
  [X] **variable statique privée de type collection** pour les instances
  typiquement `Map<K,C> m` où `K` ensemble attributs utilisés pour l'égalité sur `C`
  [X] **getter statique avec paramètre clé d'identification**
  typiquement `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`

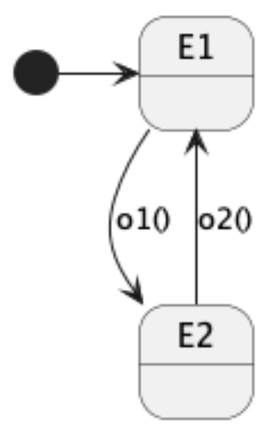
- **pour le contexte :**
  [X] **variable état** `e` de type `E`
  initialisée avec un objet de la **classe de l'état initial**
  [X] pour toute opération **`C::o`** de la signature de `C` **affectée par / affectant** l'état,
  une opération **`E::o`** et une **délégation**
  `C::o(…) { this.e.o(…) }`



- **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 :**

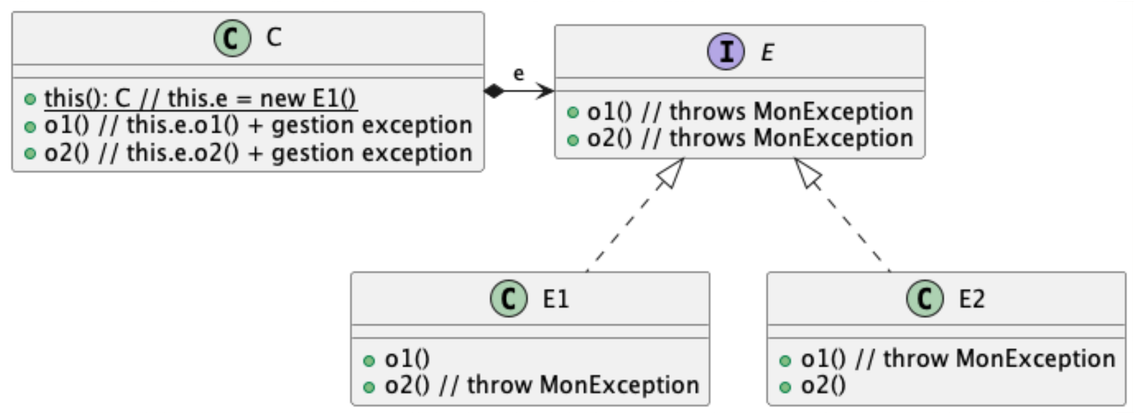
  [X] **levée d'exception** : `Ei::o(…) { throw new MonException(« … ») }`

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

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

  [X] **faire un choix :**

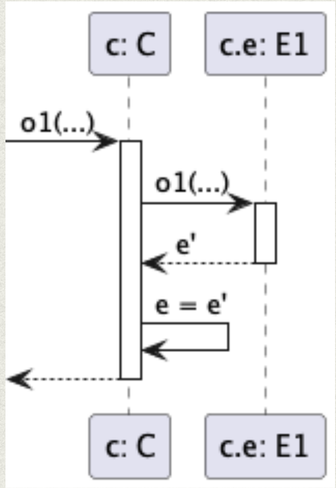
  - transfert exception (récupérable par le client)
  - récupération exception et traitement spécifique



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
```

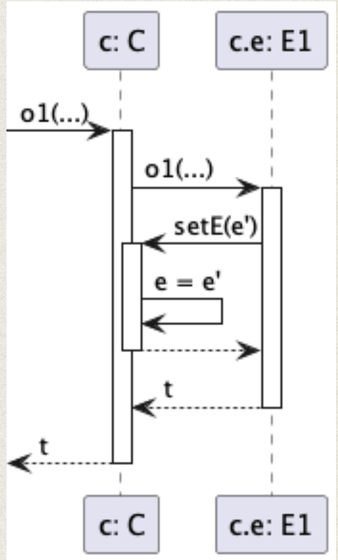
on peut aussi retourner des `Tuple2<T, E>` si besoin

STATE (TRANSITION CHOIX 2)

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

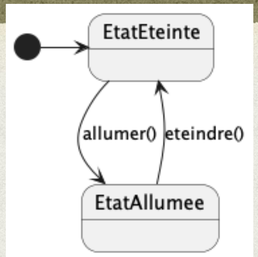
second choix : **contexte en paramètre et modificateur d'état**
permet aussi, si besoin, l'accès à des méthodes de travail dans C (visibilité paquetage)

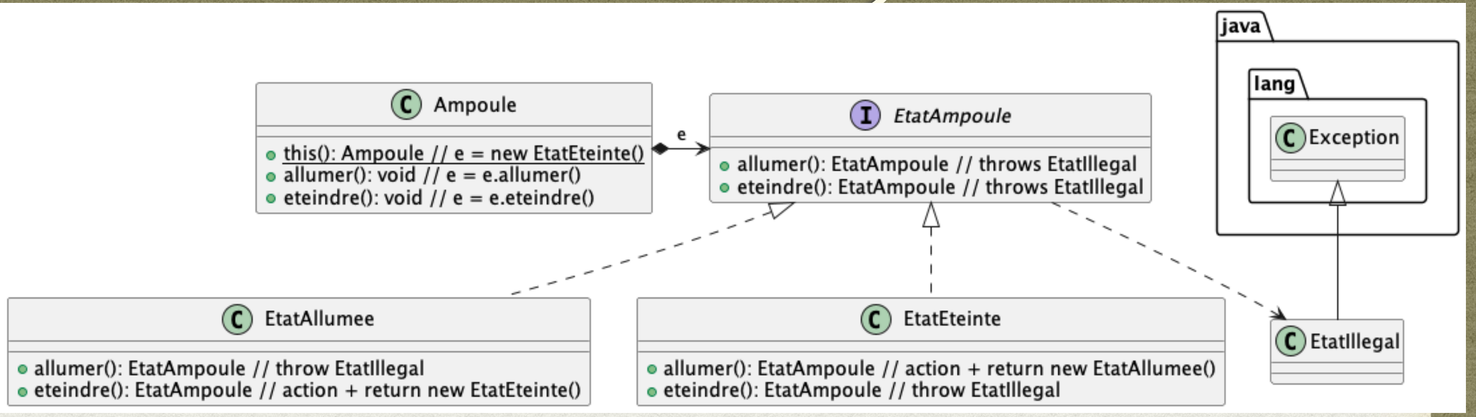
```
[X] E::o(c: C, …): T
[X] E1::o(c: C, …): T { … travail … ; c.setE(new E2()); return … }
[X] C::o(…): T { return e.o(this, …) }
[X] C::setE(e': E): () { this.e = e' }
```



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

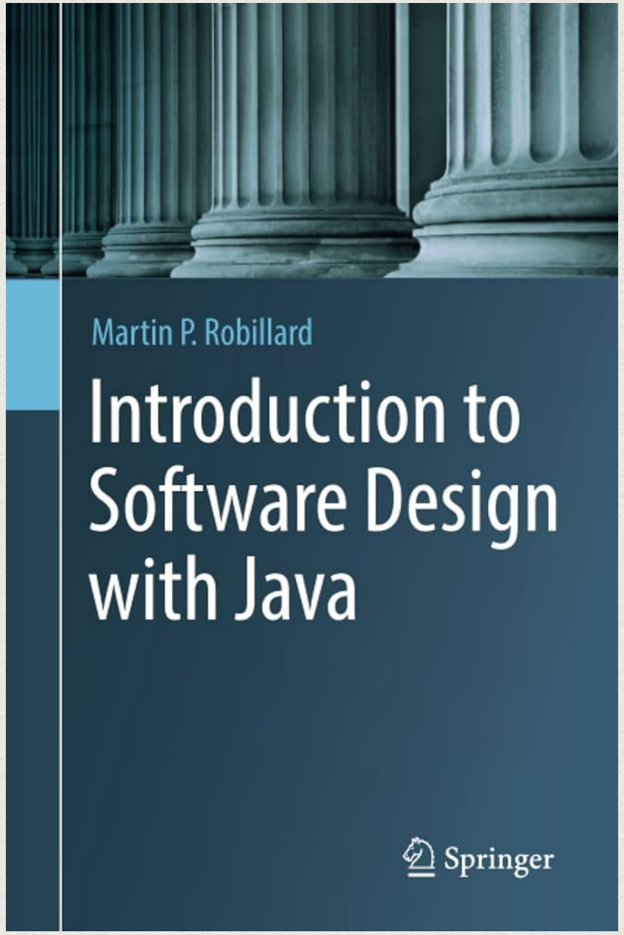
SYNTHÈSE

- 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)

M1 MIAGE / MAPC
COMPOSITION

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

STRUCTURATION

- système trop large → parties plus petites + assemblage
  « diviser pour régner » (*divide-and-conquer*)
- c'est classique, il y a tout un historique
  par exemple des modules de Parnas en 1972 aux (micro-)services
- ici via utilisation de relations « verticales » et « horizontales »
  spécialisation et composition

PRINCIPES DE BASE

- on part d'une classe qui a trop de responsabilités = **God Class**[+]

  ! responsabilité ≠ méthode

- on distribue les responsabilités
  - utilisation de l'héritage (! L de SOLID)
  - utilisation de la composition ET de la délégation

- exemple de GameModel et de ses « 13 piles de cartes »

PRINCIPES DE BASE

- attention à ne pas arriver à une solution problématique
- pour aider : Composite, Decorator, loi de Demeter

COMPOSITE

- illustration sur CardSource
  implémentations Deck, CardSequence et CompositeCardSource
  méthodes isEmpty et draw
  itération sur les éléments

- diagramme de classe et diagramme d'objets
  (problème de cycles, un patron n'est pas tout)

- diagramme de séquence et code
  (passer de l'un à l'autre + délégation)

- problématique de l'ajout dans le composite avec add(e) vs this()
  (avantages et inconvénients)

LOI DE DEMETER

- illustration sur deux diagrammes de séquence
  (générique et/ou GameModel)
- présentation du problème
- le principe de la loi
  (double point, le voisin de mon voisin)

DECORATOR

- illustration sur CardSource avec comportement « en plus »
  (log ou mémorisation)

- problématiques de certaines solutions
  - classes spécialisées (combinatoire, actions en + vs méthodes en +)
  - implémentation de (multi-)modalité (God Class et Switch Statement)

- solution et fonctionnement du décorateur

DECORATOR

- facilité de combiner les décorations
- importance du caractère additif de ces dernières
- suppression de comportement vs L de SOLID
- perte d’identité
- combinaison Composite + Décorateur

CREATION DYNAMIQUE : COPIE PAR CLONAGE

- que créer dynamiquement quand on a besoin ?
- illustration sur CardSourceManager et la copie de ses sources
- checklist pour utilisation de Clonable

CREATION DYNAMIQUE : CREATION PAR PROTOTYPES

- mauvaises solutions
- solution de Prototype

SYNTHÈSE

- attention aux **God Class**[+],
  respecter **SOLID**, diviser les choses
  utiliser la délégation au besoin
- loi de **Demeter**
- patrons **Composite**
  groupe (arbres) d'objets
  se comportant comme un objet « simple »
- patrons **Decorator**
  ajout dynamique de comportement à des objets
- clonage (`Clonable`) et patron **Prototype**
  création d'objets à partir d'objet existants
  sans forcément connaître le type de ces derniers
- **se rappeler :** les patrons ne font pas tout