Design Patterns

Matthias Colin

Programme

- Concepts POO
- Design Patterns
- Application en C++ et C#

History of Design Patterns

- 1966, Patterns of Streets, Christopher Alexander
- 1987, OOPSLA, patterns on programming, Kent Beck and Ward Cunningham
- 1994, Design Patterns: Elements of Reusable Object-Oriented Software
  - Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides
  - 23 classic software design patterns, pre-UML, languages smalltalk and C++
- 1996, Pattern-Oriented Software Architecture, vol 1, Buschman, Meunier, Rohnerts, Sommerlad
- 2000, Pattern-Oriented Software Architecture, vol 2, Schmidt, Stal, Rohnert, Buschmann
- 2002, Patterns of Enterprise Application Architecture, Fowler
- 2004, Head First Design Patterns, Freeman, Robson, Bates, Sierra

Historique

- ***Design Patterns: Elements of Reusable Object-Oriented Software*** par le gang of 4 (gof)

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (1994)

- 23 Problématiques avec solution
  - 1 version générique et 1 exemple concret
  - Modélisation en “UML”
  - Solution en C++ et/ou smalltalk
- Adaptation en Java, C#, Python, …

UML

- statique:
  - diagramme de classes
- dynamique
  - diagramme de séquence ou collaboration
  - diagramme d’états transitions

Classification des Design Patterns (GoF)

- construction (creational patterns)
  - singleton
  - fabrique abstraite, méthode de fabrication
  - monteur(builder)
  - prototype
- structures de données (structural patterns)
  - composite
  - décorateur
  - adaptateur, façade,
  - bridge
  - proxy
  - flyweight
- comportements (behavioral patterns)
  - itérateur
  - état
  - observateur
  - commande
  - template method, strategy, visiteur
  - chaîne de responsabilité
  - memento
  - mediator
  - interpreter

Iterator: C++

pair: begin,end

operations:

- ++it: advance
- *it: read [write (opt)]
- ==, !=: compare iterators to decide end of iteration

optional operations:

- --it: backward
- +=, +, -=, -: random access

Iterator C#

Interfaces:

- IEnurable<T>: iterable object
  - GetEnumerator()
- IEnumerator<T> : iterator
  - MoveNext()
  - Current

Iterator C++

header <iterator>

- next, prev: eq +, -
- advance: eq += or -=
- distance

header <algorithm>

- for_each, find_if, fill, generate, transform…

Composite: tree

Problématique:

- donnée arborescente avec plusieurs types de noeuds
- propagation d’une opération

Exemples:

- Système de fichiers: fichiers réguliers, **dossiers**, liens, jonctions, …
- DOM+XML, JSON

Composite + Visitor

Problématique: comment ajouter plusieurs opérations dans un composite

=> maintenance minimale

principe Separation of Concerns

Construction d’objets

IArithmeticExpression expr = new Number(){Value = 12};

// code after depends on interface API

Number expr = new Number(){Value = 12};

// code after using Number type: high dependency

How to decrease dependency to constructor ?

Abstract Factory + Factory Method + Singleton

https://en.wikipedia.org/wiki/Abstract_factory_pattern

https://en.wikipedia.org/wiki/Factory_method_pattern

https://en.wikipedia.org/wiki/Singleton_pattern

Builder

https://en.wikipedia.org/wiki/Builder_pattern

Exemples:

- stringstream, stringbuf (C++)
- StringBuilder (C#, Java)
- lombok @Builder (Java)

Interfaces: focus

- couplage faible
- injection de dépendance (DI)

Patterns de fabrication

Problématique: appel à un constructeur => code peu maintenable

Airbus380 plane = new Airbus380(couleur, nombreSiege)

comment passer à un Airbus320 ?

Principe de substitution de Liskov **LSP** mit en échec par l'appel au constructeur

Exemples en Java

- List
- String
- objet custom Plane avec plusieurs attributs

Fabriques

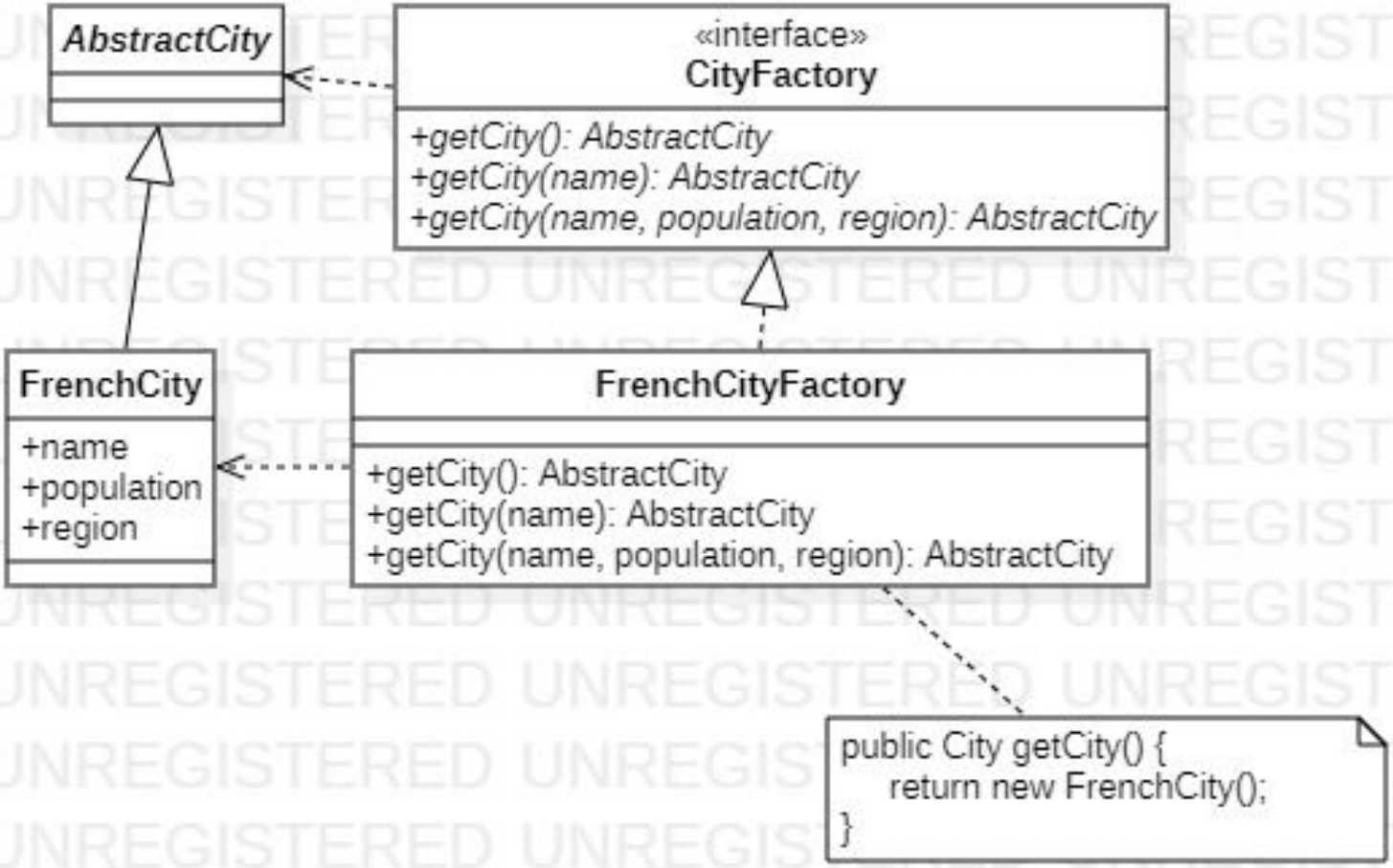
Objectif: encapsuler l'appel d'un constructeur (new) dans une méthode

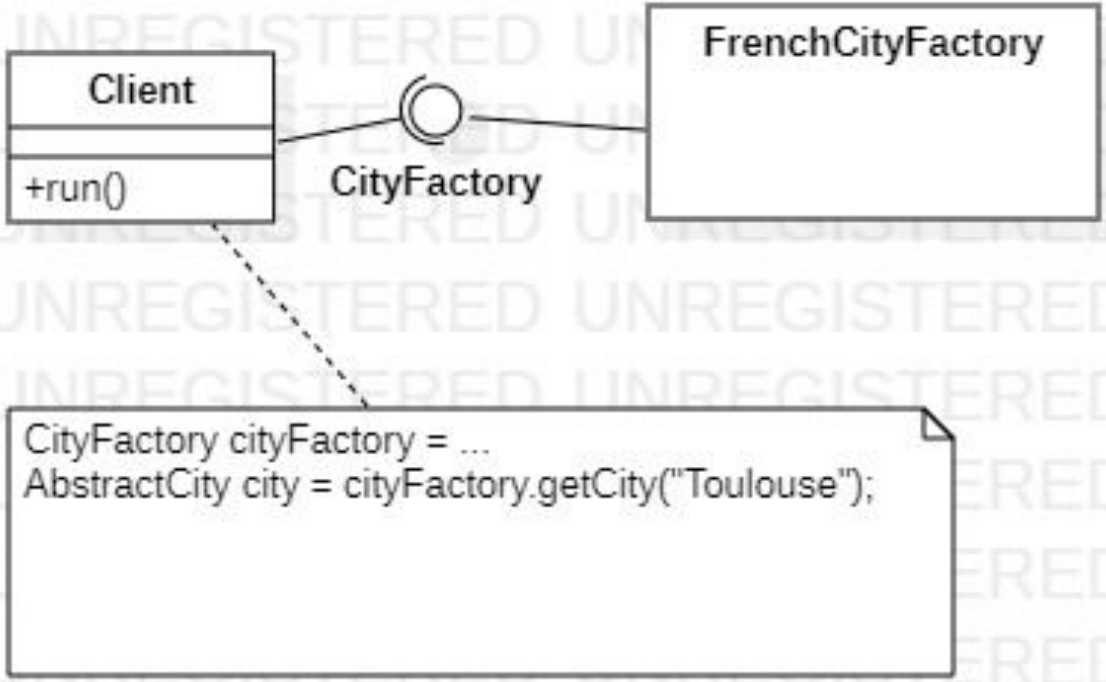
2 patterns:

- Factory method (Fabrique, Fabrication, Méthode de fabrication)
  - 1 seul type d'objet (Ex: Plane)
- Abstract Factory (Fabrique abstraite)
  - 1 famille d'objets cohérents (Ex: Mobilier => Table, Chaise, Lampe, ..)

Eléments de solution:

- classe de l'objet à fabriquer => niveau d'abstraction Interface ou class abstraite


AbstractCity
«interface»
CityFactory
+getCity(): AbstractCity
+getCity(name): AbstractCity
+getCity(name, population, region): AbstractCity
FrenchCity
+name
+population
+region
FrenchCityFactory
+getCity(): AbstractCity
+getCity(name): AbstractCity
+getCity(name, population, region): AbstractCity
public City getCity() {
return new FrenchCity();
}



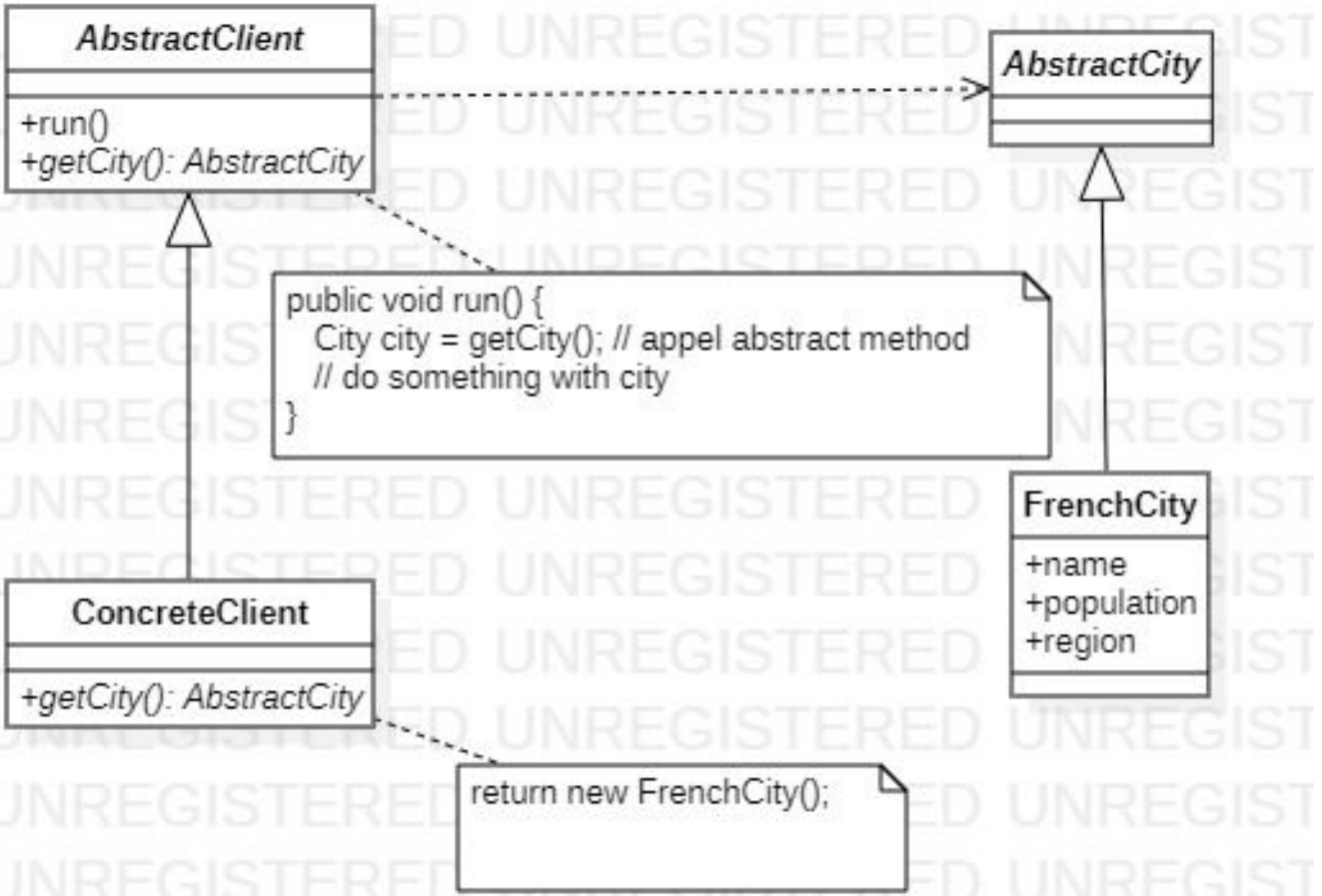
Problématique: comment obtenir l'instance de la fabrique

- fabrique par défaut (singleton)
- configuration externe (Injection dépendance)

Simplifications

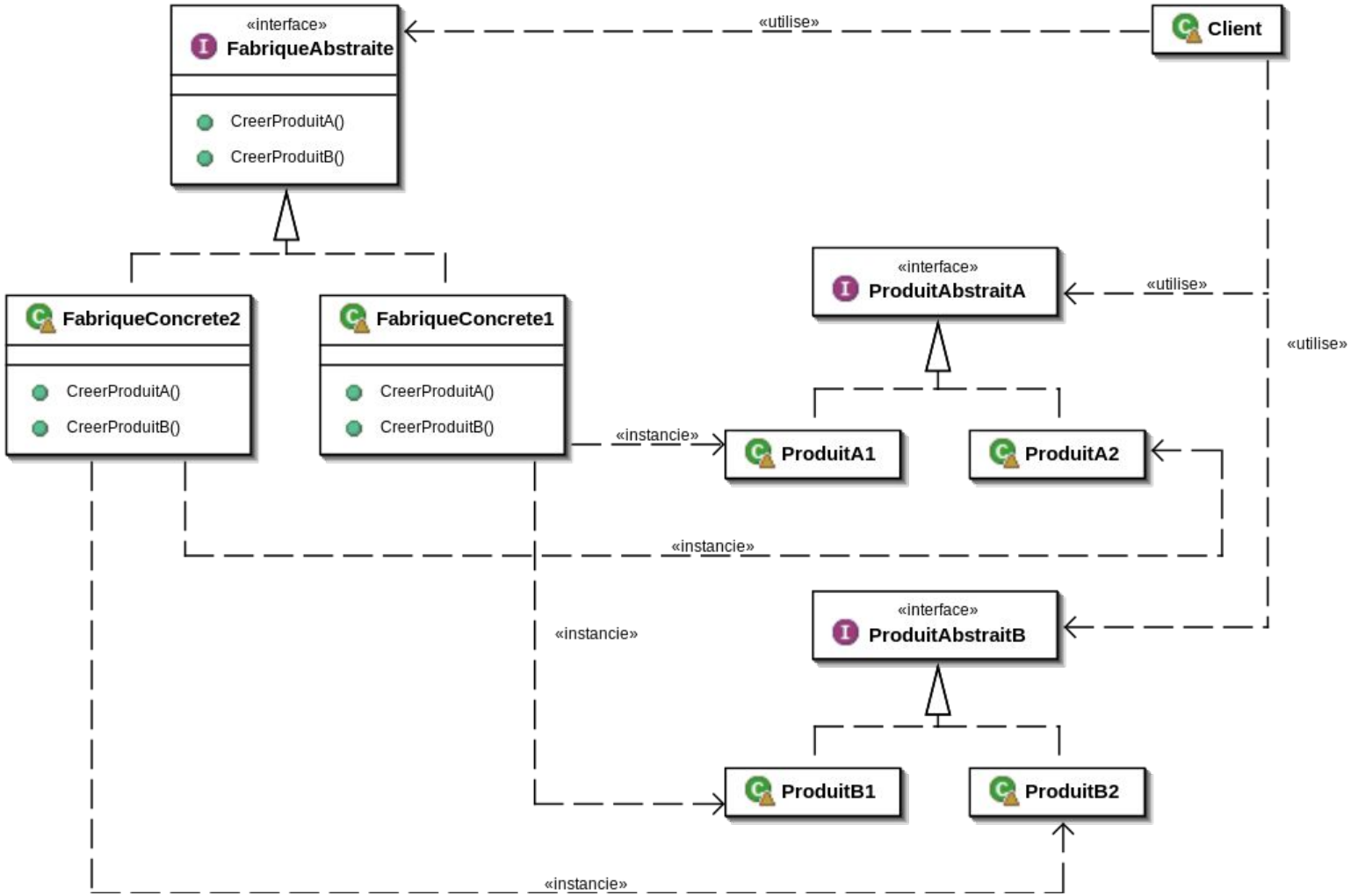
- Classe abstraite du produit est fusionnée avec la fabrique
  - List.of(...)
  - Stream.of(...)
  - LocalDatetime.of(year, month, day)
  - LocalDatetime.of(year, month, day, hour)
  - LocalDatetime.of(year, month, day, hour, minute)
  - Optional.of(city), Optional.empty()
- Classe client et classe Fabrique fusionnées

  (cf slide suivant)


AbstractClient
+run()
+getCity(): AbstractCity
AbstractCity
public void run() {
City city = getCity(); // appel abstract method
// do something with city
}
FrenchCity
+name
+population
+region
ConcreteClient
+getCity(): AbstractCity
return new FrenchCity();

Extensions

- Abstract Factory (Fabrique Abstraite)
  - Plusieurs type de produits => une méthode de fabrication par type de produit
  - Exemple: en XML, on peut avoir des éléments, textes, commentaires, attributs, ….
    - interface Document est une fabrique abstraite avec les methodes
      - Element createElement(...)
      - Text createNodeText(...)
      - Attr createAttribute(...)


«interface»
FabriqueAbstraite
CreerProduitA()
CreerProduitB()
«utilise»
Client
«interface»
ProduitAbstraitA
«utilise»
«utilise»
FabriqueConcrete2
CreerProduitA()
CreerProduitB()
FabriqueConcrete1
CreerProduitA()
CreerProduitB()
«instancie»
ProduitA1
ProduitA2
«instancie»
«instancie»
«interface»
ProduitAbstraitB
ProduitB1
ProduitB2
«instancie»

Singleton

- Problématique: unicité d’une instance d’une classe particulière
- Exemples:
  - Factory
  - Connection ou DataSource
- Solution
  - constructeur privé remplacé par une méthode de classe publique
  - méthode de classe contrôle unicité
  - attribut de classe privé pour mémoriser l’instance unique
- Discussion:
  - passage de paramètre ?

Adaptateur / Adapter

Problématique: adapter un besoin nouveau à un existant avec une interface légèrement différente (nom de méthode, paramètres)

Intérêt: réutilisabilité

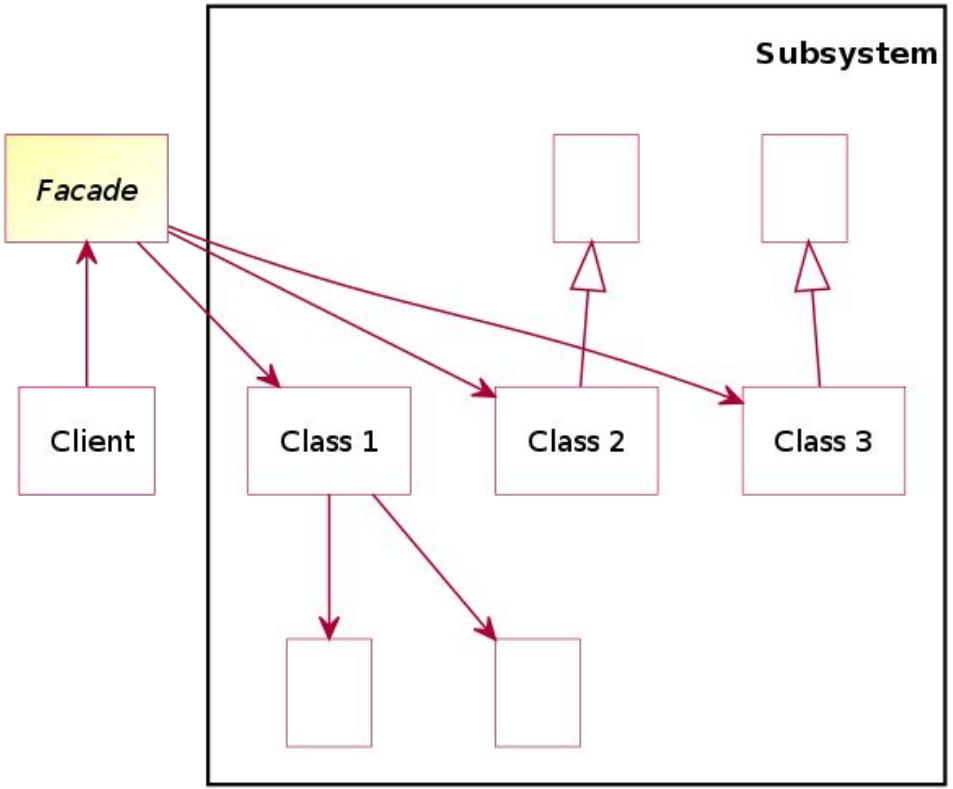
Solutions: héritage vs **composition** (+ souple, adaptable)

Extensions: Façade

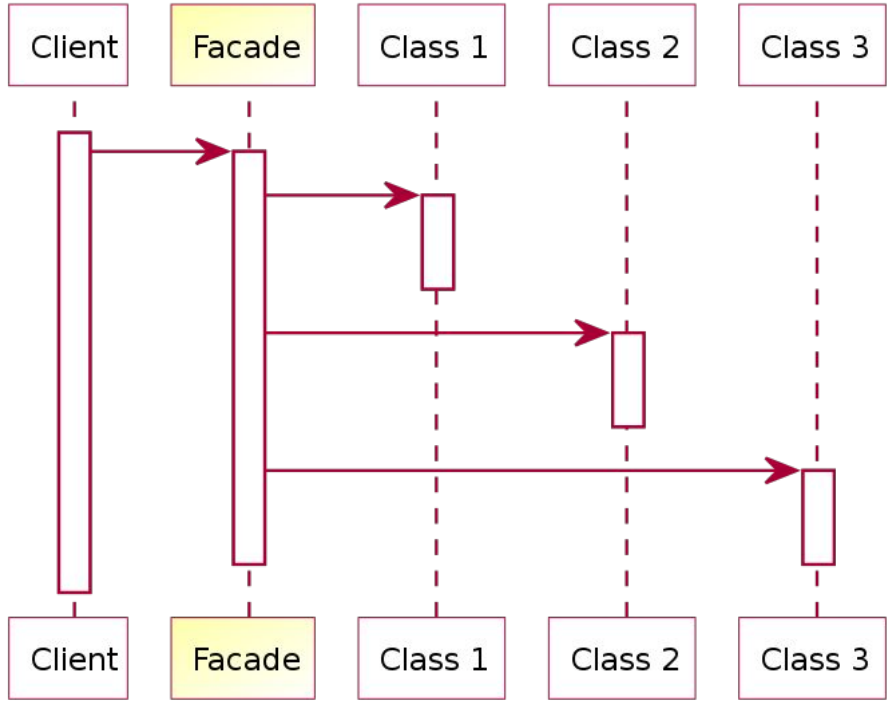
Façade (fr) / Facade (en)

Avantages:

- maintenabilité
- indépendance du code client vis à vis des classes derrière la façade
- simplifier des api complexes en une api correspondant au besoin



Sample class diagram



Sample sequence diagram

source Wikipedia: https://en.wikipedia.org/wiki/Facade_pattern

Composite

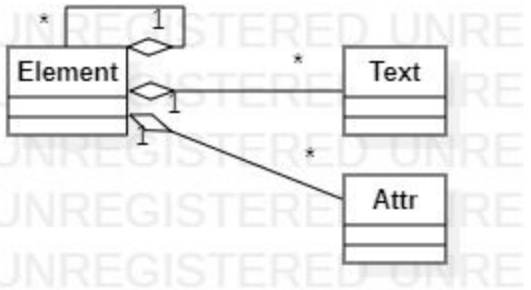
Problématique:

- structure arborescente: xml, json, filesystem, ihm
- propager une opération dans l'arborescence
  - Exemple: taille totale du filesystem, pretty print xml, …

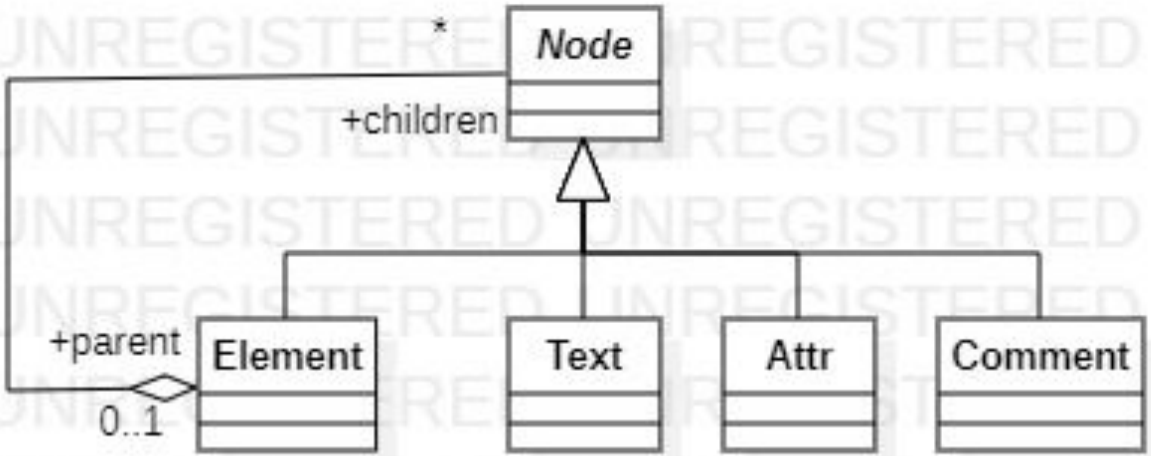
Solution:

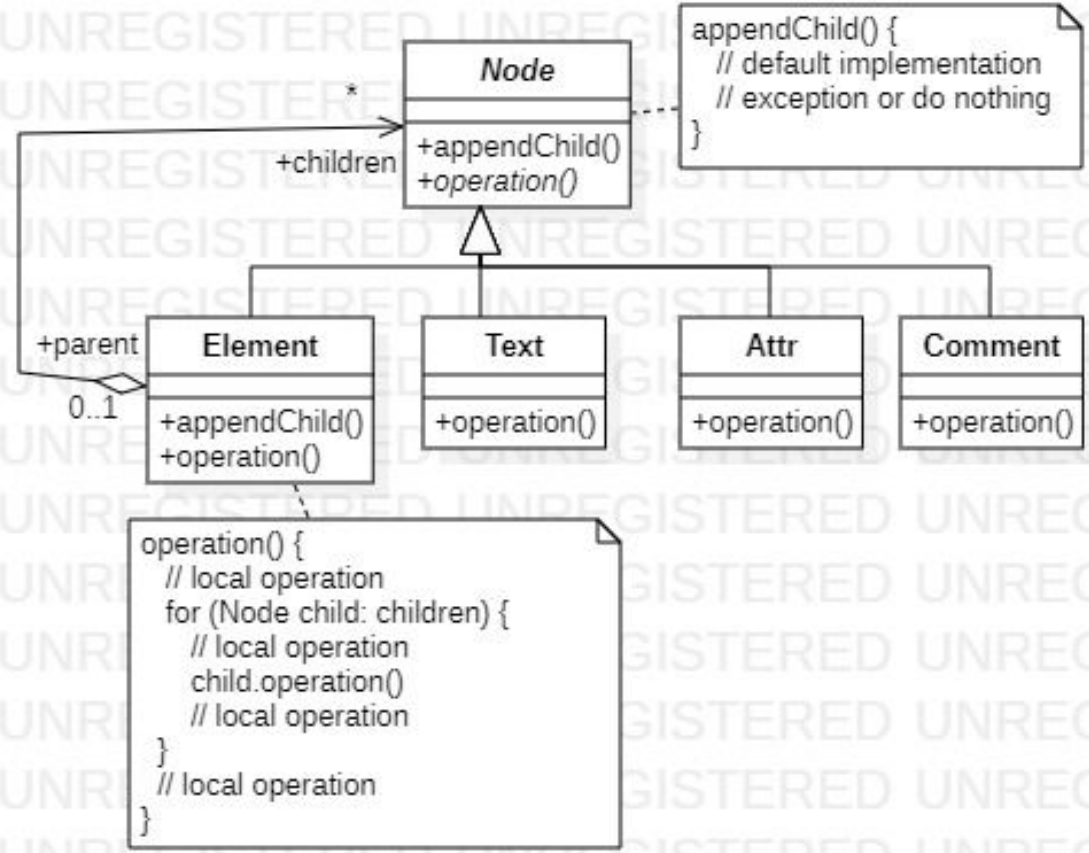
- classes:
  - Composite: conteneur
  - Leaf: feuille(s)
  - *Component* : classe abstraite ou interface de généralisation
- méthodes de gestion des enfants au niveau de Component (+ souple)

Solution bancale



Solution du pattern

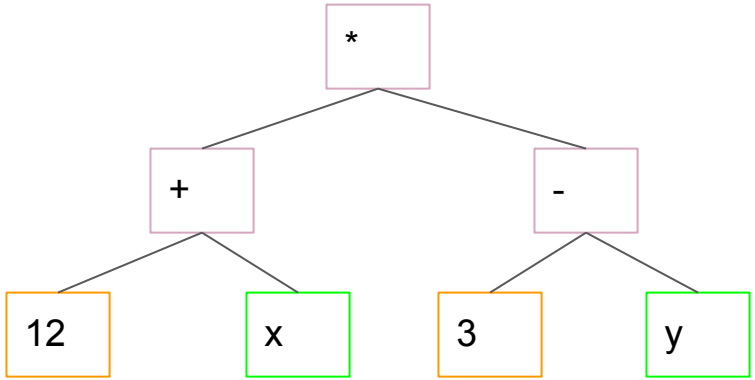



Node
+appendChild()
+operation()
appendChild() {
// default implementation
// exception or do nothing
}
*
+children
+parent
0..1
Element
+appendChild()
+operation()
Text
+operation()
Attr
+operation()
Comment
+operation()
operation() {
// local operation
for (Node child: children) {
// local operation
child.operation()
// local operation
}
// local operation
}

Atelier Composite

- Expression arithmétique
- Opérateur: +, -, *, …
- Valeur numérique (float)
- Variable (x, x1, y)
- Opération: toString

Exemple: (12 + x) * (3 - y)



Itérateur / Iterator

Problématique: parcours d'une structure de données (tableaux, listes, ensembles, listes d'associations, graphes, …)

Solution:

- externaliser le parcours
- chaque itérateur est spécifique à la structure qu'il parcoure
- la structure délivre son itérateur

Implémentation dans les langages

- Java: interfaces Iterable (iterator()) et Iterator (hasNext(), next())
- Python: objet iterable (__iter__) et iterator (__next__)
- C++: structure fournit 2 itérateurs (begin(), end()), itérateur(++, *, ==)

Bonus: boucle “foreach” utilisant implicitement l’itérateur

for (Plane plane: planes) { …} // Java, C#, C++

for plane in planes // python, delphi

for plane of planes //javascript

Strategy

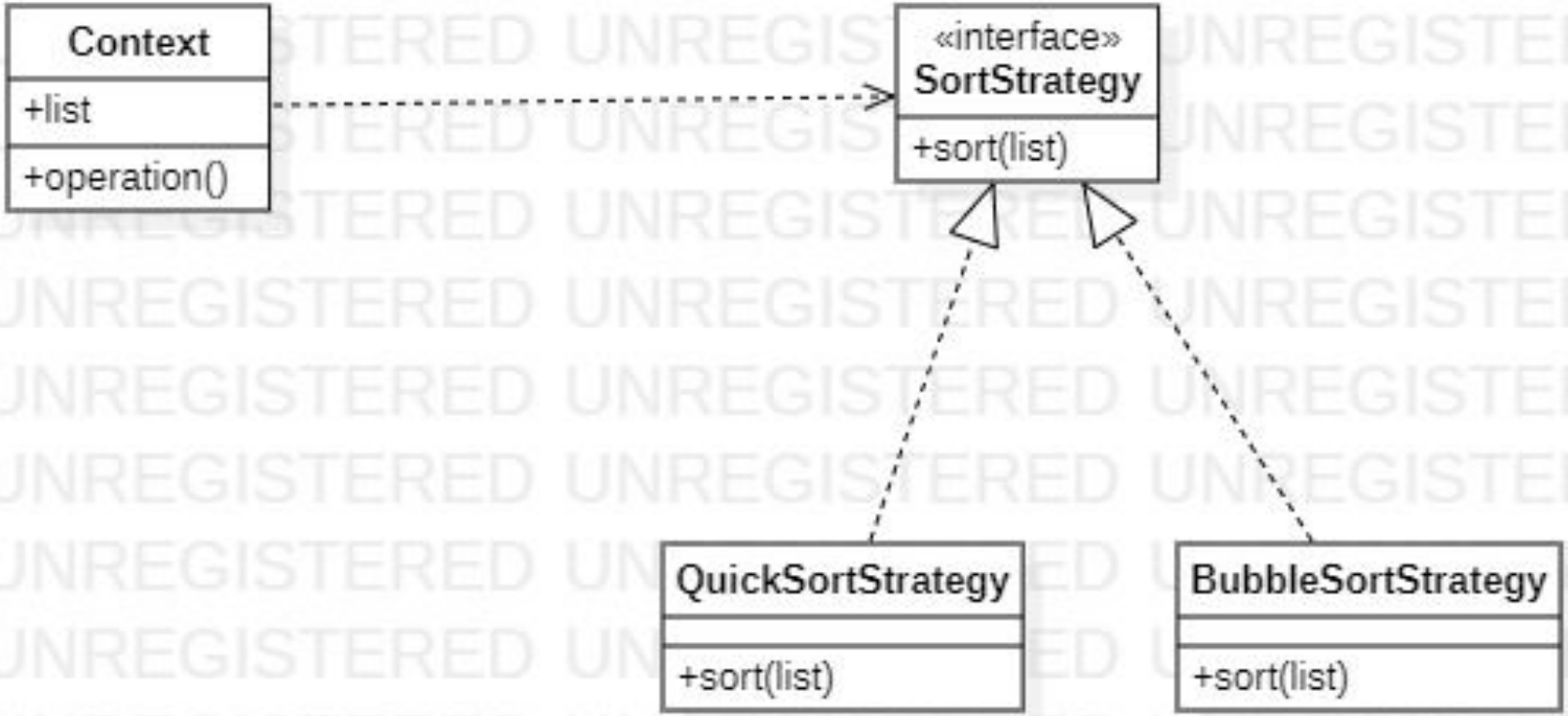
Problématique: algorithme avec **coût** important disponible en plusieurs variantes

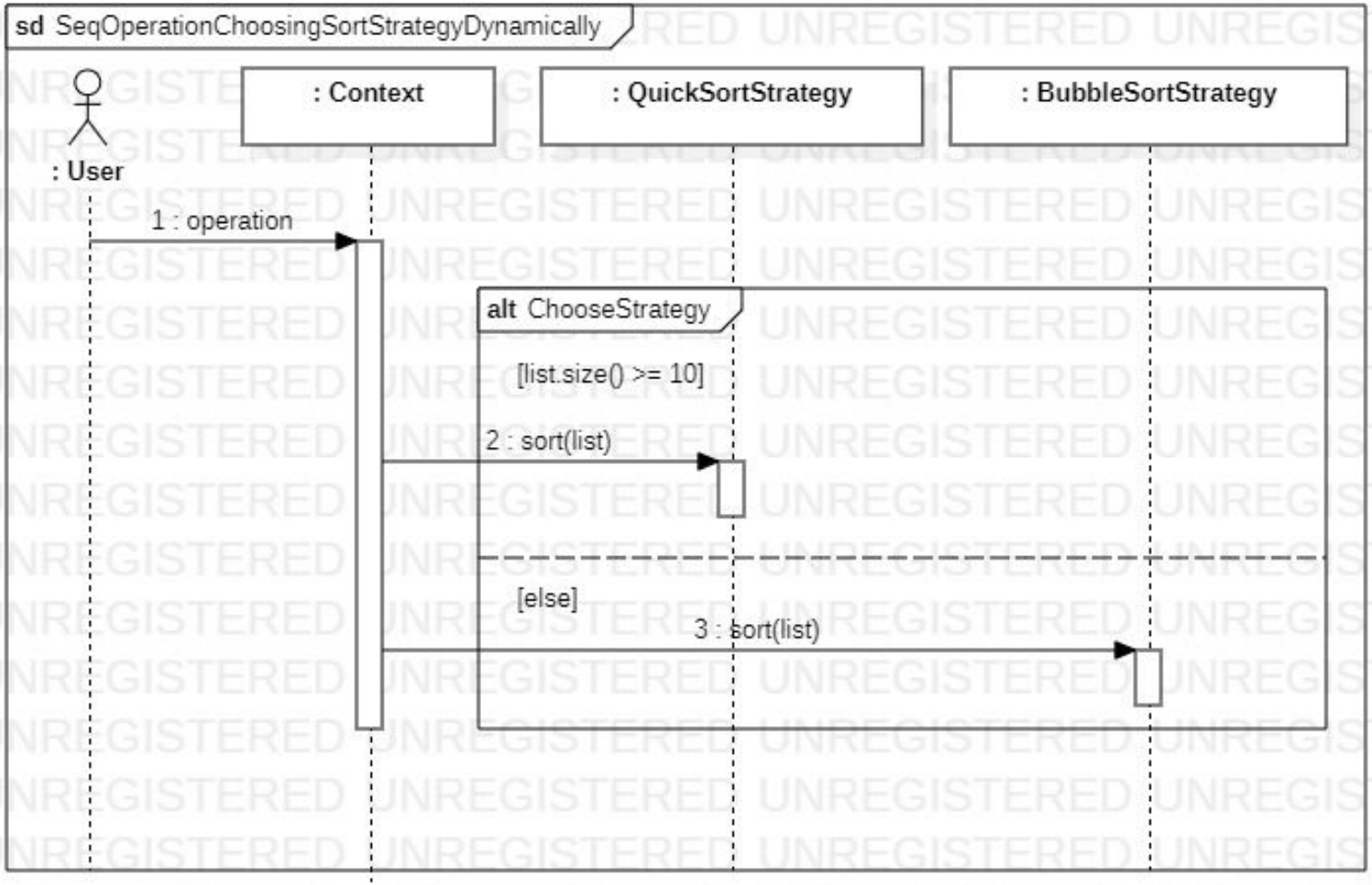
Exemple: tri => à bulle, insertion, rapide, fusion
Exemple: base de données, SELECT

- … where id = 125 => stratégie: parcours par l'index associé à la PK
- … where region is null; => stratégie: balayer toute la table

Choix de la stratégie:

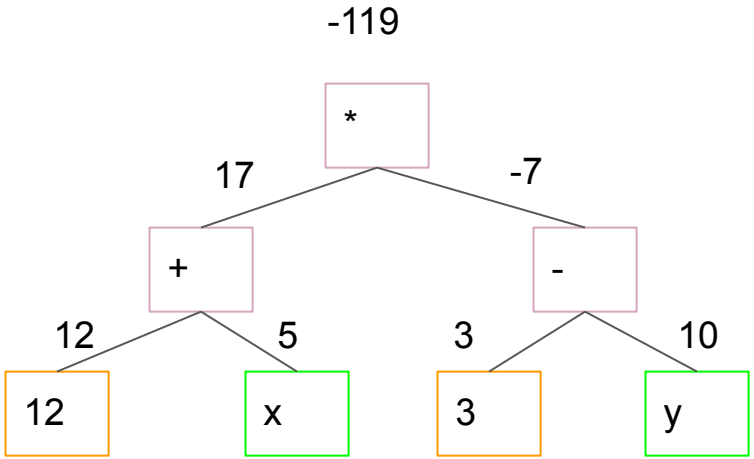
- constructeur du contexte
- setter dans le contexte
- paramètre de l'opération
- choix dynamique dans le contexte


Context
+list
+operation()
«interface»
SortStrategy
+sort(list)
QuickSortStrategy
+sort(list)
BubbleSortStrategy
+sort(list)


sd SeqOperationChoosingSortStrategyDynamically
: User
: Context
: QuickSortStrategy
: BubbleSortStrategy
1 : operation
alt ChooseStrategy
[list.size() >= 10]
2 : sort(list)
[else]
3 : sort(list)

Atelier Composite + Visitor

Valeurs des variables pour évaluation
x = 5
y = 10

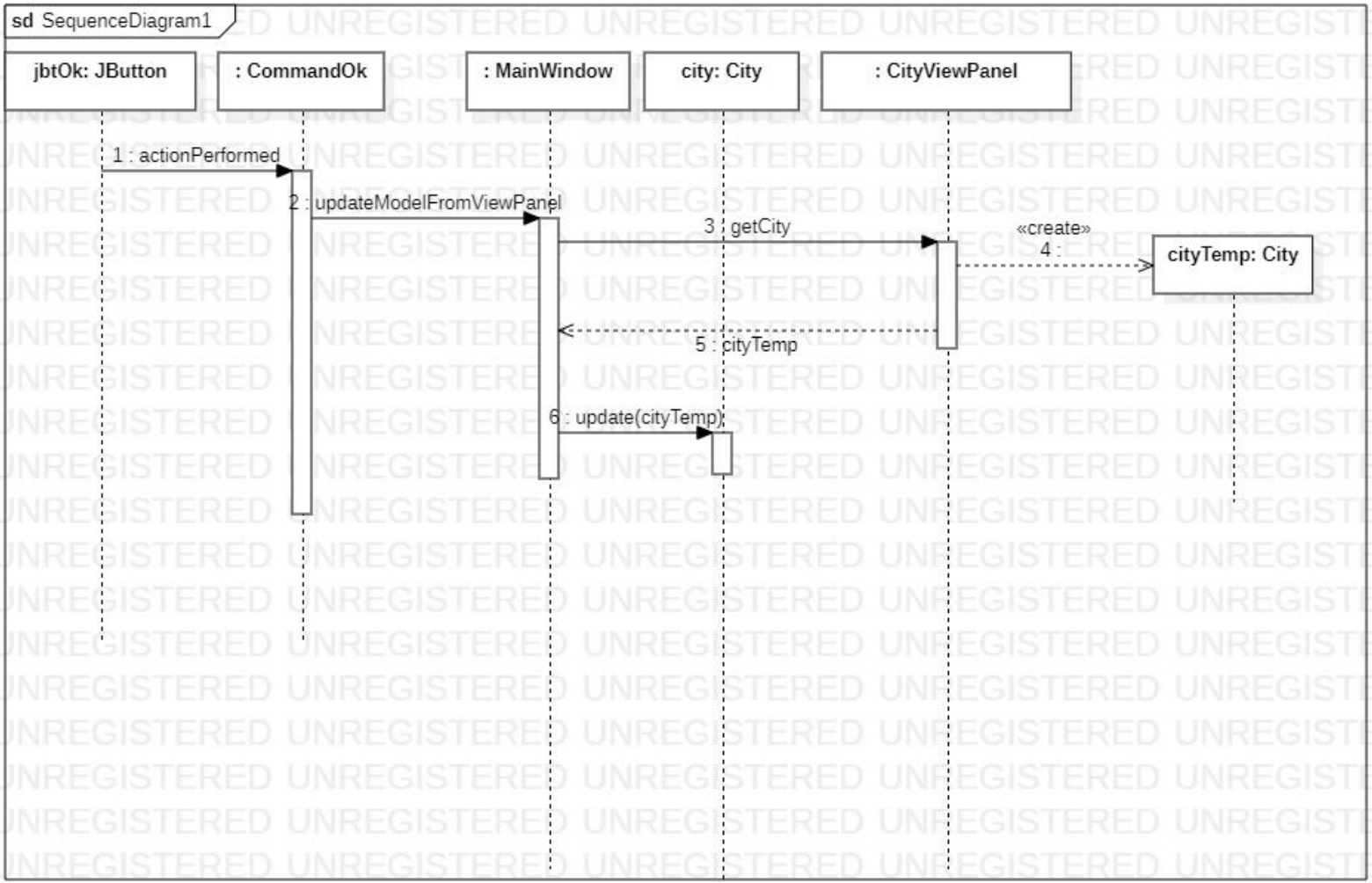


Observateur / Observer

Problématique: un objet observé change d'état et un certain nombre d'observateurs veulent être au courant de ce changement pour réagir (rafraîchissement, synchro base de données, …)

Commande / Command

Problématique: couplage faible entre un donneur d'ordre et un executeur (receiver)

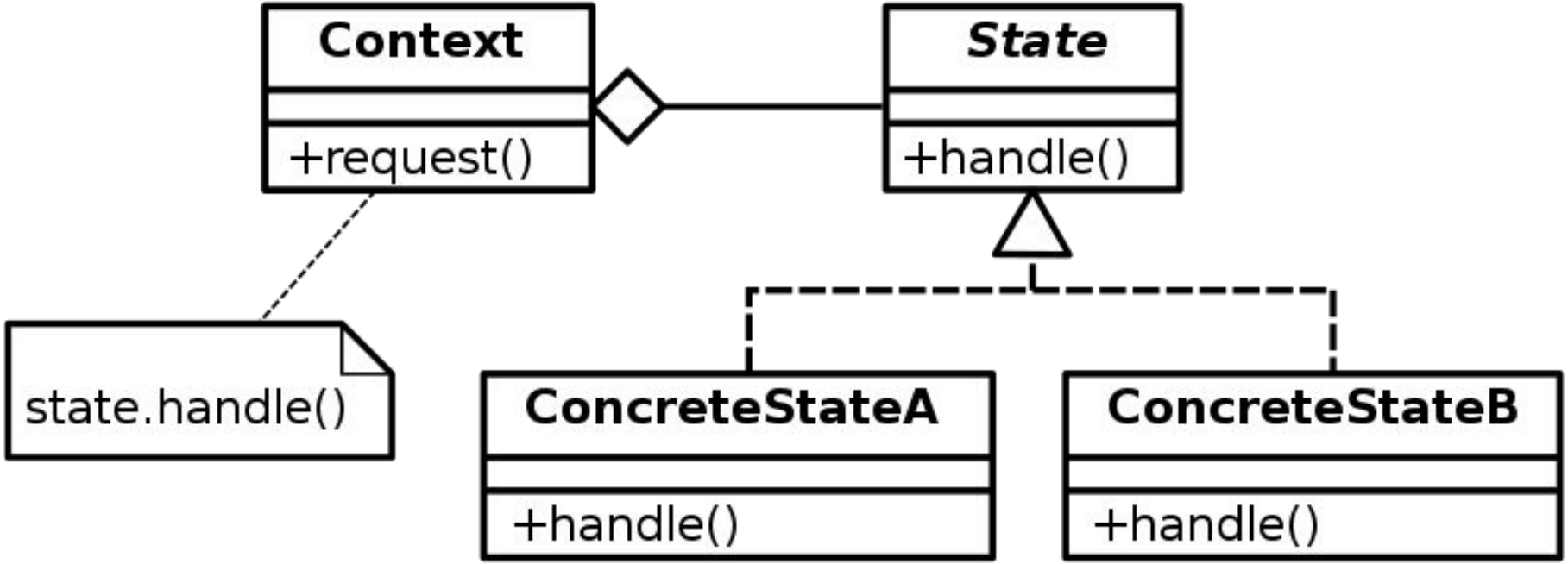

sd SequenceDiagram1
jbtOk: JButton
: CommandOk
: MainWindow
city: City
: CityViewPanel
1 : actionPerformed
2 : updateModelFromViewPanel
3 : getCity
«create»
4 :
cityTemp: City
5 : cityTemp
6 : update(cityTemp);

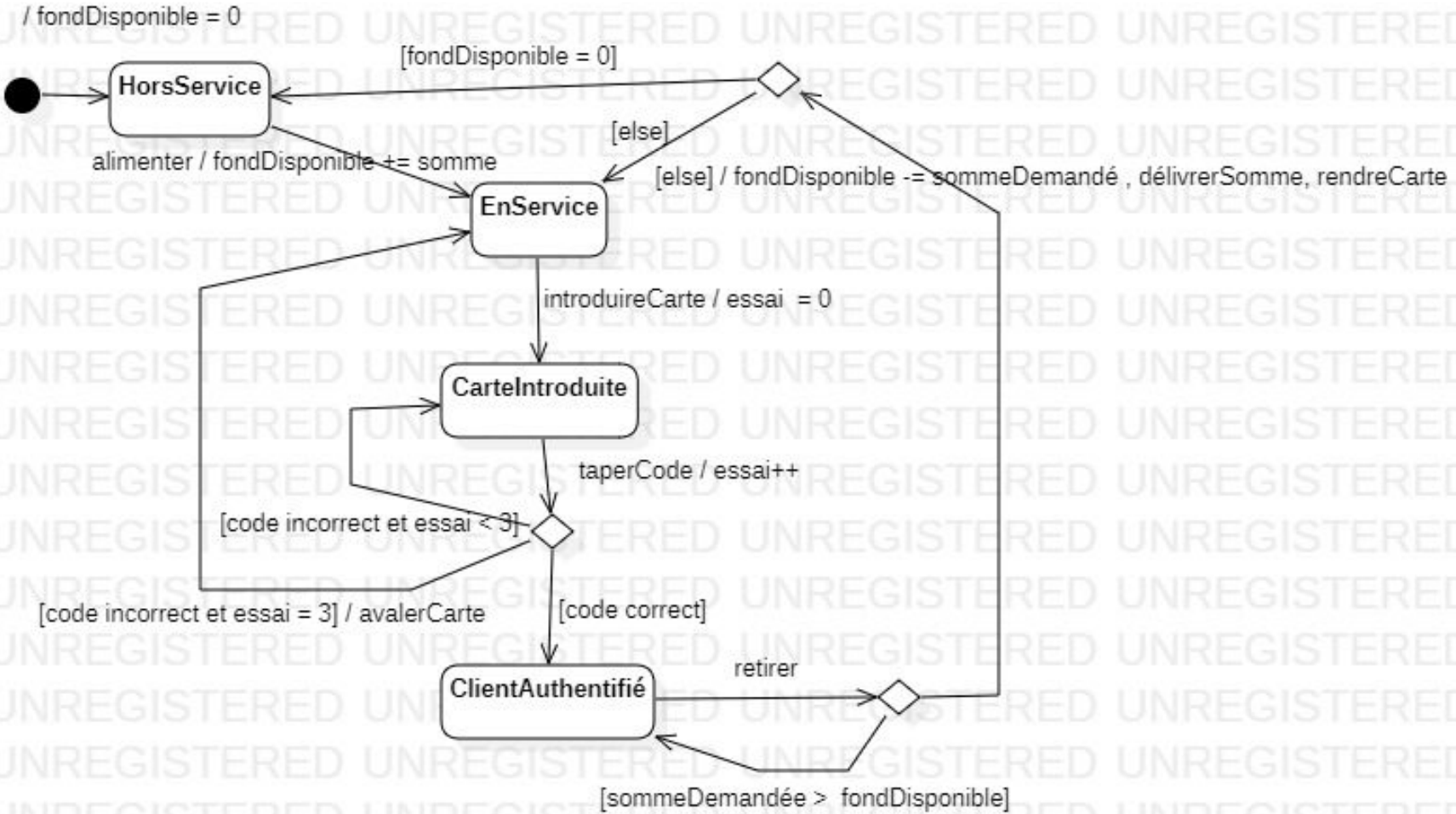
Etats Transitions / State

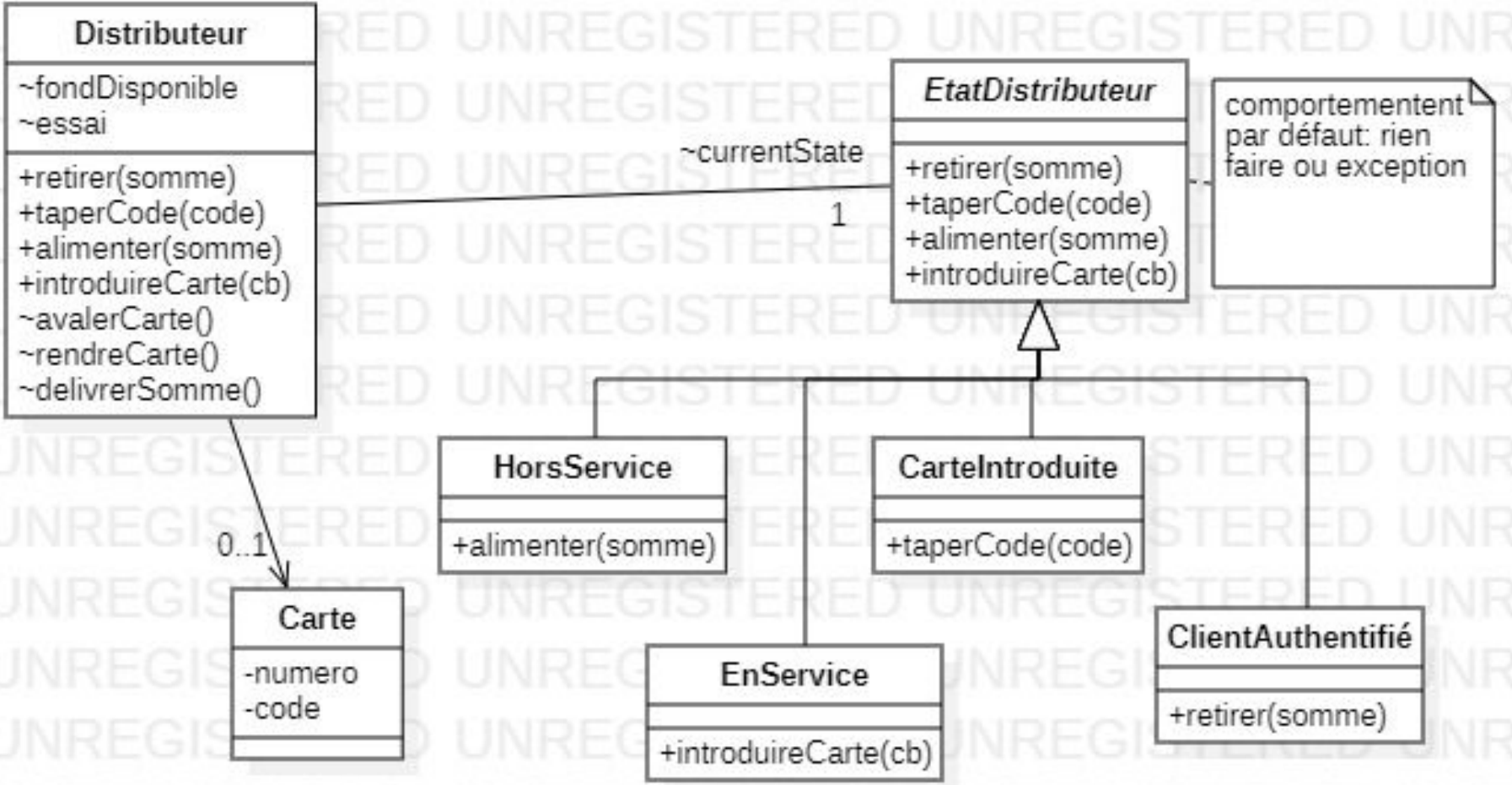
Problématique: un objet à plusieurs états, i.e ayant un comportement différent en fonction de son état.

Exemple: distributeur bancaire, barrière de péage

Solution: externaliser le traitement des fonctions dans des classes Etats


Context
+request()
State
+handle()
state.handle()
ConcreteStateA
+handle()
ConcreteStateB
+handle()


/ fondDisponible = 0
HorsService
[fondDisponible = 0]
[else]
alimenter / fondDisponible += somme
[else] / fondDisponible -= sommeDemandé , délivrerSomme, rendreCarte
EnService
introduireCarte / essai = 0
CarteIntroduite
taperCode / essai++
[code incorrect et essai < 3]
[code incorrect et essai = 3] / avalerCarte
[code correct]
ClientAuthentifié
retirer
[sommeDemandée > fondDisponible]


Distributeur
~fondDisponible
~essai
+retirer(somme)
+taperCode(code)
+alimenter(somme)
+introduireCarte(cb)
~avalerCarte()
~rendreCarte()
~delivrerSomme()
~currentState
1
EtatDistributeur
+retirer(somme)
+taperCode(code)
+alimenter(somme)
+introduireCarte(cb)
comportementent par défaut: rien faire ou exception
0..1
Carte
-numero
-code
HorsService
+alimenter(somme)
CarteIntroduite
+taperCode(code)
EnService
+introduireCarte(cb)
ClientAuthentifié
+retirer(somme)

Conception

- GRASP: General Responsability Assignment Software Patterns
- **SOLID principles:**
  - SRP: Single Responsability Principle
  - OCP: Open-Close Principle
  - LSP: Liskov Substitution Principle
  - ISP: Interface Segregation Principle
  - DIP: Dependency Inversion Principle

GRASP: General Responsibility Assignment Software Patterns|Principles

- 9 principles first published by Craig Larman in *Applying UML and Patterns*
  - controller
  - creator
  - indirection
  - information expert
  - **low coupling**
  - high cohesion
  - **polymorphism**
  - protected variations
  - pure fabrication

https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)

Other principles

Separation of Concerns

DIP: Program to an interface, not an implementation. (Gang of Four 1995:18)

Composition over inheritance: Favor 'object composition' over 'class inheritance'. (Gang of Four 1995:20)

Autre classification des patterns du Gof

Interface: adapter, façade, composite, bridge

Responsability: singleton, observer, mediator, proxy, chain of responsability, flyweight

Construction: builder, factory method, abstract factory, prototype, memento

Operation: template, state, strategy, command, interpreter

Extension: decorator, iterator

Autres patterns

Liste: POSA 1 à 5

- https://en.wikipedia.org/wiki/Pattern-Oriented_Software_Architecture
- https://en.wikipedia.org/wiki/Software_design_pattern

Quelques uns:

- MVC, MVP, MVVP
- RAII (Resource acquisition is initialization)
- Pool

Creational Patterns

- Abstract Factory, Factory Method
- Singleton, Builder
- Dependency Injection
- Lazy Initialization
- Multiton
- Object Pool
- Prototype
- RAII

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Marker
- Proxy
- Module
- Twin
- Front Controller
- Extension object

Anti patterns

- Mauvaise pratiques
  - God object
  - too much YAGNI
- Code Smells
- Qualité de code
  - SonarQube
  - Linter
  - Tests + Couverture
  - Quality Gate
- Méthodes agiles

Bibliographie

- Design Patterns: Elements of Reusable Object-Oriented Software (1994)
  The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Design patterns pour Java - Les 23 modèles de conception: description et solutions illustrées en UML 2 et Java (2009, 5ème édition 2022)
  ENI Editions
  Laurent Debrauwer
- Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software
  2004, O'Reilly
  Elisabeth Robson, Eric Freeman
  Traduction française: Design Patterns - Tête la première
- Design Patterns in Java (2006)
  Pearson EducationInc.
  Steven John Metsker, William C. Wake
- Pattern-Oriented Software Architecture "POSA" vol 1 à 5 (1996-2007)