

ARCHITECTURE LOGICIELLE

David Auber
david.auber@u-bordeaux.fr

DÉROULEMENT DE L'ENSEIGNEMENT

Le cours se déroulera sur 12 semaines

Cours magistral le mardi matin 10h15 ~12h15

Travaux dirigés en demi groupe.

- Le mardi de 14h00 ~ 16h00 : Avec M. Auber Groupe 1
- Le mardi de 14h00 ~ 16h00 : Avec M. Celerier Groupe 2

Site Web : www.labri.fr/~auber/ALM1GL

Moodle : moodle1.u-bordeaux.fr/enrol/index.php?id=3349

OBJECTIFS DU COURS ET DES TRAVAUX DIRIGÉS

Connaitre la modélisation UML:

- diagramme de classe/objet
- Diagramme de séquences

Connaitre l'ensemble des modèles de conceptions.

- Création
- Structure
- Comportement

Construire des architectures logicielles:

- LISIBLE, maintenables, réutilisables et extensibles

SOURCES BIBLIOGRAPHIQUES

Ce cours est basé sur deux ouvrages élémentaires pour tout programmeur dans un langage Objet.

Modélisation et conception orientées objet avec UML 2.0,

Michael Blaha & James Rumbaugh

Design patterns. Catalogue des modèles de conception réutilisables.

Eric Gamma, Richard Helm, Ralph Johnson, John Vissides.

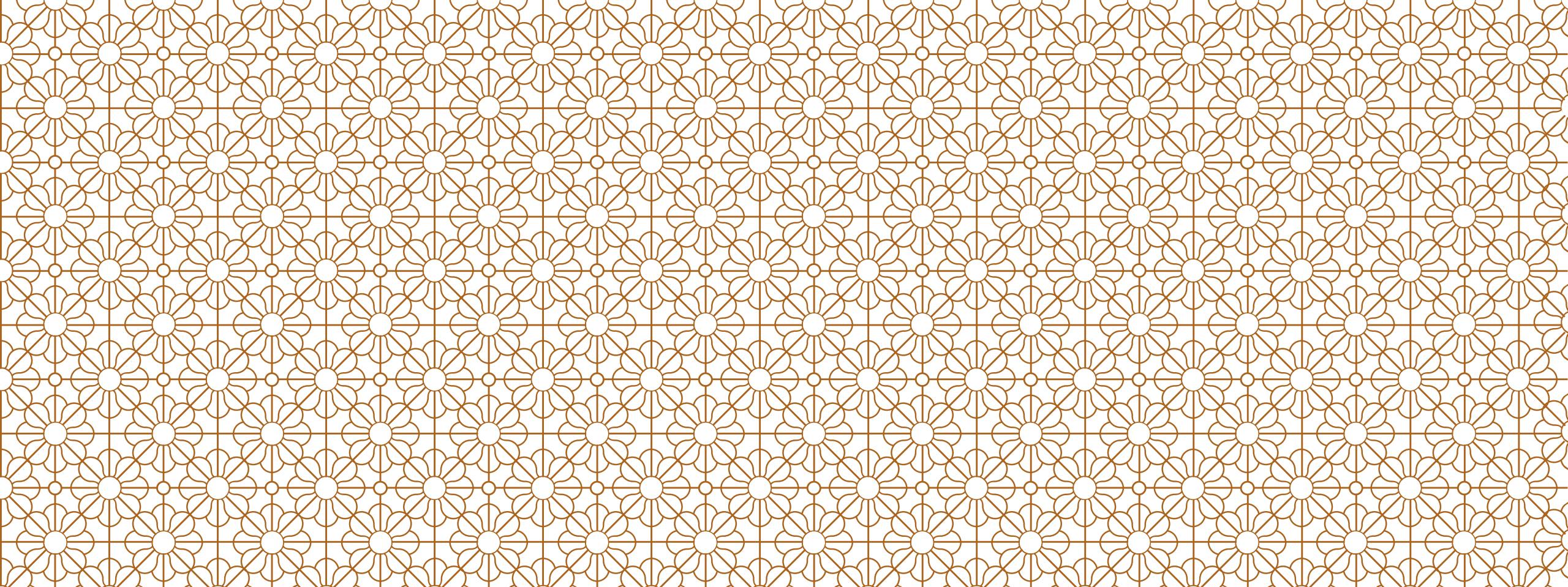
EVALUATION DE ENSEIGNEMENT

Examen:

- Epreuve sur table de trois heures portant sur tout le cours et sur tous les travaux dirigés.

Contrôle continu :

- Compte rendu de TD par groupe de 2 déposés sur moodle :
 - Un seul fichier zip nommé TPX_NOM1_NOM2.zip contenant :
 - Un fichier pdf entre 2 et 8 pages de compte rendu
 - Un répertoire contenant les sources du projet
 - Un fichier jar exécutable de votre programme
- Un mini projet à la fin des travaux dirigés.
 - Par groupe de deux, contenu à définir ultérieurement.



RAPPEL ET INTRODUCTION UML

QU'EST-CE QUE L'ORIENTÉ OBJET ?

Dire que vous développez en orienté objet signifie que vous organisez votre logiciel sous la forme d'une collection d'objets indépendants qui incorporent à la fois une structure de données et un comportement.

LES 4 CARACTÉRISTIQUES DE L'APPROCHE OBJET

L'identité : signifie que les données sont organisées en entités discrètes et distinguables nommées objets.

La classification : signifie que deux objets possédant la même structure de données et le même comportement sont des représentants d'une même classe.

L'héritage : est le partage des attributs et des opérations entre les classes sur la base d'une relation hiérarchique. Une super classe possède des informations générales que les sous classes spécialisent et décrivent en détail.

Le Polymorphisme : Signifie que les mêmes opérations peuvent se comporter différemment dans des classes différentes.

MODÉLISATION VERSUS IMPLÉMENTATION

L'objectif de la modélisation est de se détacher des langages de programmation afin d'élaborer le logiciel à un niveau plus abstrait.

Le développement d'une application complète prend bien plus de temps que la modélisation de celle-ci. Une erreur de conception détectée pendant l'implémentation peut obliger à tout recommencer.

LES TROIS MODÈLES DE LA MODÉLISATION OBJET

- 1. Le modèle de classe** : décrit la structure statique des objets d'un système et de leurs relations. Un diagramme de classe est un graphe dont les nœuds sont des classes et les arcs des relations entre ces classes.
- 2. Le modèle d'états** : décrit les états successifs d'un objet au cours du temps. Un diagramme d'état est un graphe dont les sommets sont des états et les arcs des transitions entre les états.
- 3. Le modèle d'interaction** : décrit la façon dont les objets d'un système coopèrent pour obtenir un résultat. Il commence par les cas d'utilisation qui sont détaillés par les diagrammes de séquence et des diagrammes d'activités. Un cas d'utilisation est axé sur une fonctionnalité

MODÉLISATION DES CLASSES ET OBJETS EN UML 2.0

Un modèle de classe/objet capture la structure statique d'un système en caractérisant les objets de ce système, leurs relations, les attributs et les opérations de chaque classe d'objet. C'est le plus important des trois modèles.

Objet : Un objet est un concept, une abstraction, une entité ou une instance qui a une signification pour une application.

Classe : Une classe décrit un groupe d'objets qui possèdent les mêmes propriétés (attributs/champs/données membres), le même comportement (opérations/méthodes/services), les mêmes types de relations et la même sémantique.

DIAGRAMME DE CLASSES

Nom de classe
Nom attribut1 : type de donnée1 = valeur par défaut Nom attribut2 : type de donnée2 = valeur par défaut
Nom d'opération (liste arguments) : type du résultat Nom d'opération (liste arguments) : type du résultat

DIAGRAMME DE OBJETS

Nom de l'instance : Type de l'objet

Nom attribut1 : type de donnée1 = valeur

Nom attribut2 : type de donnée2 = valeur

EXEMPLE DE CLASSES ET D'OBJETS

Card
-color : unsigned int = SPADE
-value : unsigned int = ACE
+display()

Kd : Card
color : unsigned int = DIAMON
value : unsigned int = KING

Ac : Card
color : unsigned int = CLUB
value : unsigned int = ACE

DIAGRAMME, PORTÉE

Les attributs et les méthodes statiques sont en soulignés:

Nom de classe
<u>Nom attribut1 :type de donnée1 = valeur par défaut</u> Nom attribut2 :type de donnée2 = valeur par défaut
<u>Nom opération (liste arguments) :type du résultat</u>
Nom opération (liste arguments) :type du résultat

DIAGRAMME, VISIBILITÉ

On indique la visibilité d'un attribut/méthode en plaçant un des signes suivants devant son nom :

+ : public

: protected

- : private

~ : package

Nom de classe
+ <u>Nom attribut1 :type de donnée1 = valeur par défaut</u>
<u>Nom attribut2 :type de donnée2 = valeur par défaut</u>
- <u>Nom opération (liste arguments) :type du résultat</u>
~ <u>Nom opération (liste arguments) :type du résultat</u>

DIAGRAMME, ÉNUMÉRATION

On modélise une énumération comme une classe en ajoutant le mot clé « enumeration » au dessus. Les valeurs de l'énumération sont données les unes à la suites des autres.

« enumeration »
Couleur
Piques
Trèfles
Cœurs
Carreaux

GÉNÉRALISATION ET HÉRITAGE

La généralisation permet :

- De prendre en charge le polymorphisme : On peut appeler une opération sur la super-classe le compilateur se charge d'appeler le code de la bonne fonction. Permet l'ajout de nouveau type Sans perturber le code existant.
- De structurer la description des objets : la généralisation permet de construire une taxonomie des types. Permet de modéliser les similarités et les différences entre les types.
- De réutiliser du code : Grace à la généralisation, vous pouvez hériter du code d'un autre projet. Réutiliser du code est souvent plus productif. Elle permet aussi d'adapter du code existant pour un problème spécifique.

GÉNÉRALISATION ET HÉRITAGE

On indique une généralisation en reliant deux classes avec par une flèche ayant pour extrémité un triangle isocèle vide.

Généralisation



Spécialisation

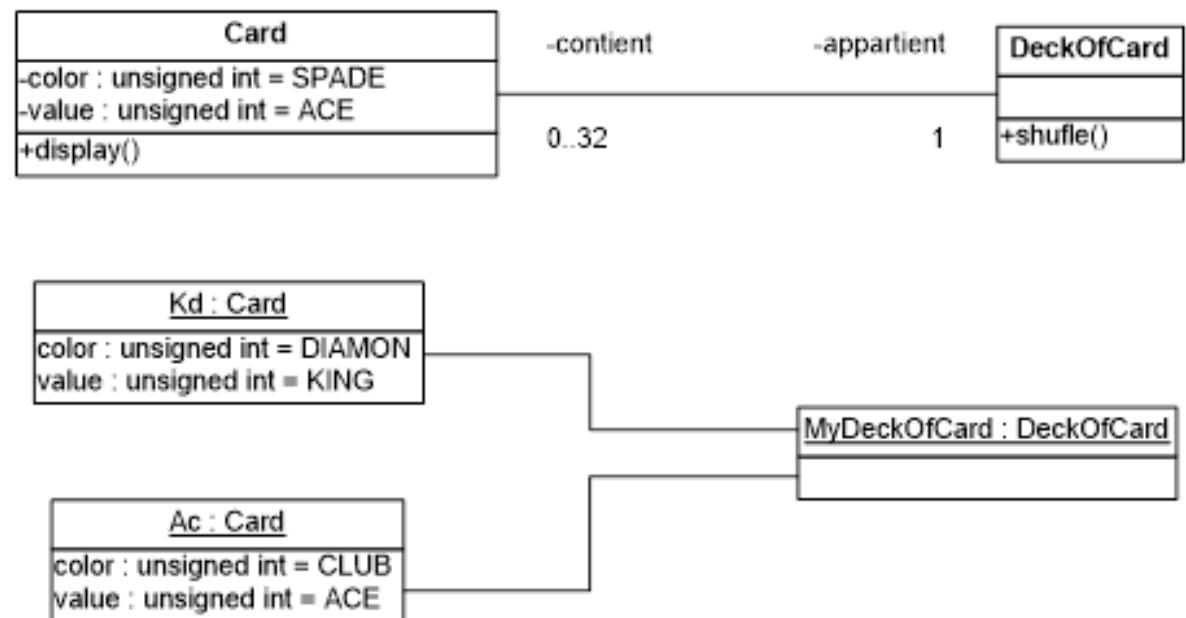
Nom de classe
Nom attribut1 : type de donnée1 = valeur par défaut
Nom attribut2 : type de donnée2 = valeur par défaut
Nom d'opération (liste arguments) : type du résultat
Nom d'opération (liste arguments) : type du résultat

Nom de classe
Nom attribut1 : type de donnée1 = valeur par défaut
Nom attribut2 : type de donnée2 = valeur par défaut
Nom d'opération (liste arguments) : type du résultat
Nom d'opération (liste arguments) : type du résultat

CONCEPT DE LIENS ET D'ASSOCIATIONS

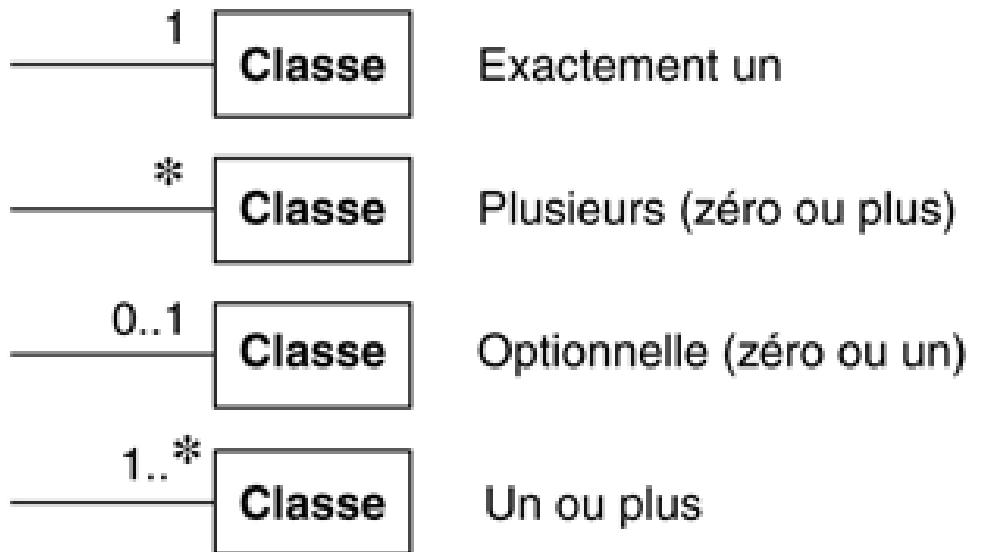
Un lien est une connexion physique ou conceptuelle entre deux objets.

Une association est une description d'un groupe de liens qui partagent une structure et une sémantique commune.



MULTIPLICITÉ DES ASSOCIATIONS

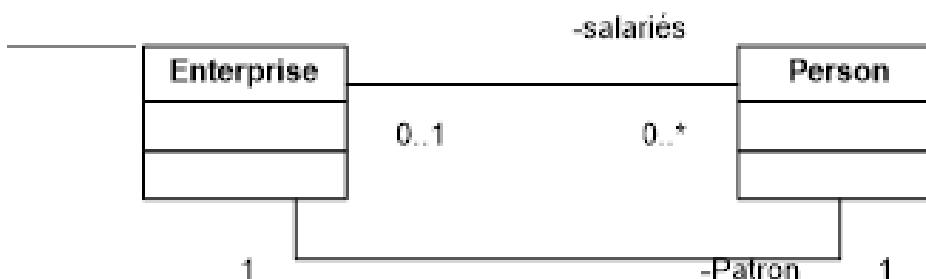
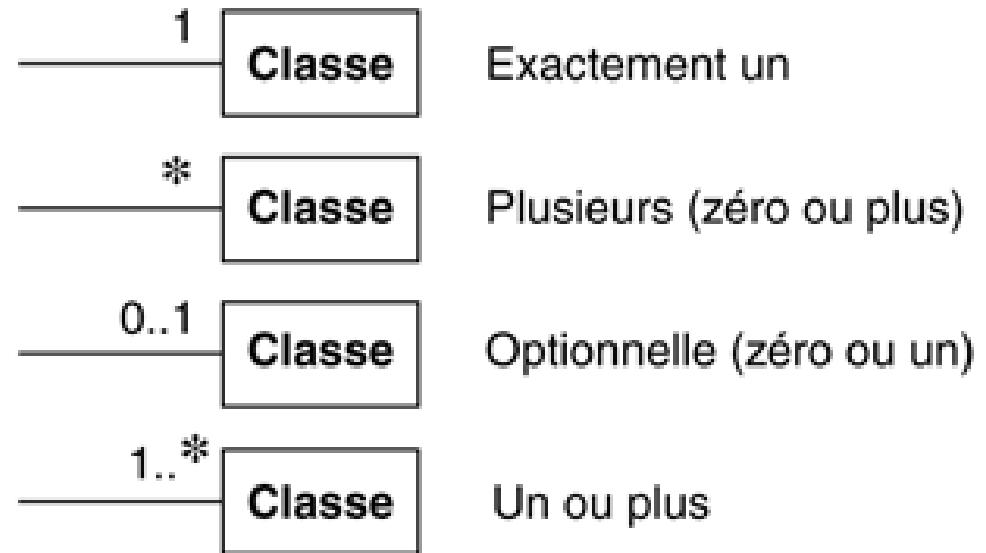
On indique le nombre d'association possible entre deux classes en ajoutant un nombre ou un intervalle à l'extrémité de l'association.



MULTIPLICITÉ DES ASSOCIATIONS

On indique le nombre d'association possible entre deux classes en ajoutant un nombre ou un intervalle à l'extrémité de l'association.

Attention il faut utiliser plusieurs associations si les liens n'ont pas la même sémantique.



ORDONNANCEMENT, BAGS ET SÉQUENCES

Par défaut une association représente un ensemble de liens. Deux objets ne peuvent donc être relié deux fois via la même association.

On peut indiquer trois autres types d'association

Rien : ensemble de liens

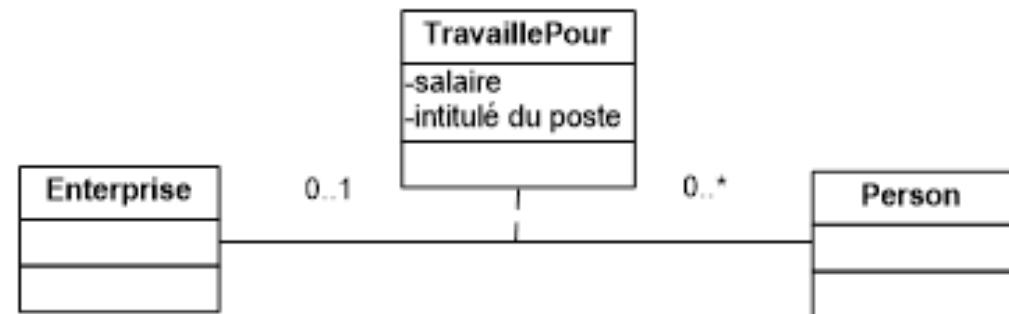
Ordered : ensemble ordonné

Bags : tableau/list plusieurs liens possibles

Sequence : Plusieurs liens ordonnées.

CLASSE D'ASSOCIATION

Une classe d'association est une classe qui modélise une association entre deux classes.



AGRÉGATION

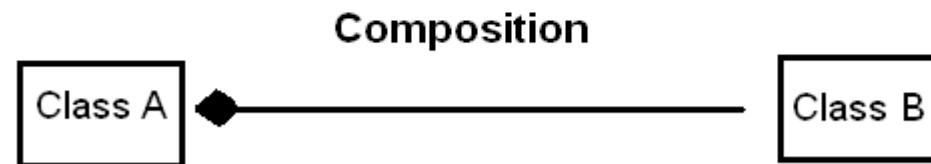
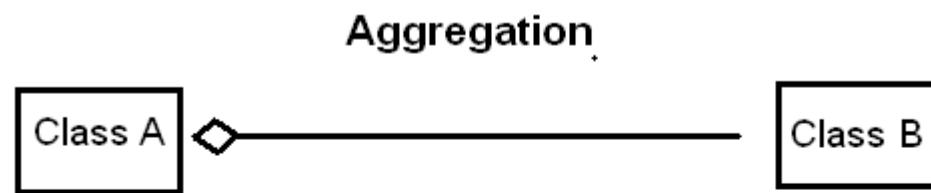
L'agrégation est une sorte d'association dans laquelle un objet agrégat est constitué de constituants. Elle permet de modéliser la relation « Tout ou partie », il y a une agrégation si on des points suivants est vrai:

- Peut-on utiliser l'expression fait partie de .
- Les opérations appliquées à l'objet constitué s'appliquent-elles automatiquement à ses constituants.
- Les valeurs des attributs constitué se propagent-elles à tous ses constituants ou à certains d'entre eux ?
- L'association présente-t-elle une asymétrie intrinsèque, dans laquelle une classe est subordonnée à une autre ?

COMPOSITION

La composition est une forme restrictive de l'agrégation :

- Une partie constituante ne peut pas appartenir à plus d'un assemblage
- La durée de vie d'une partie constituante est celle du constituant.



MODÉLISATION DES INTERACTIONS : CAS D'UTILISATION

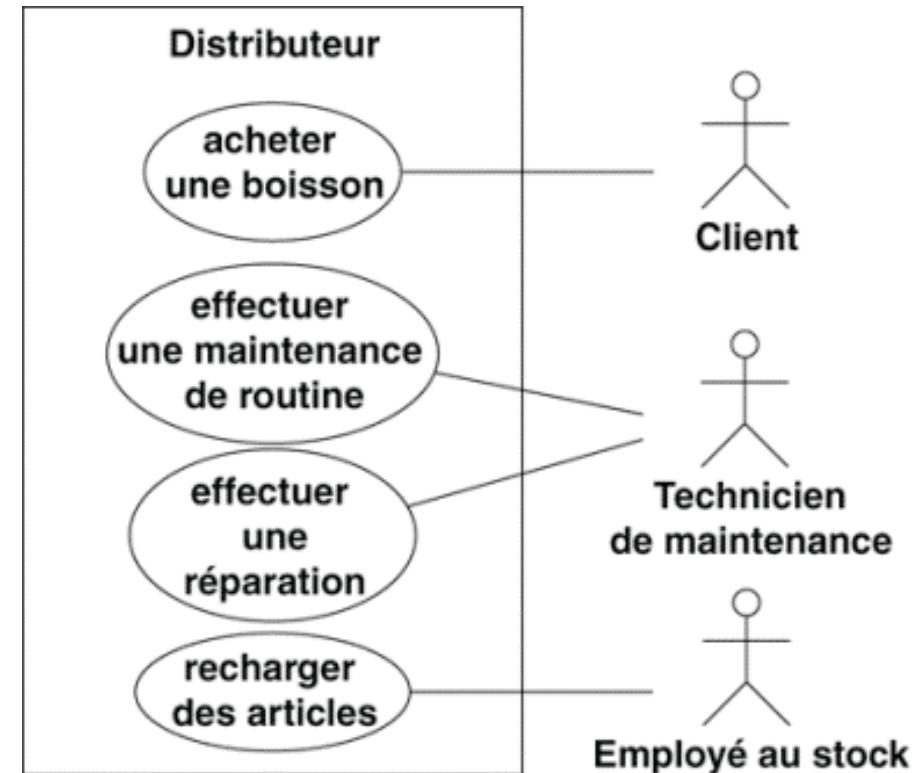
Au niveau le plus élevé, les cas d'utilisation décrivent comment un système interagit avec les acteurs extérieurs. Chaque cas d'utilisation représente une partie des fonctionnalités que le système fournit à ces utilisateurs.

Acteurs : est un utilisateur externe direct du système. Un objet ou un ensemble d'objet qui communique directement avec le système.

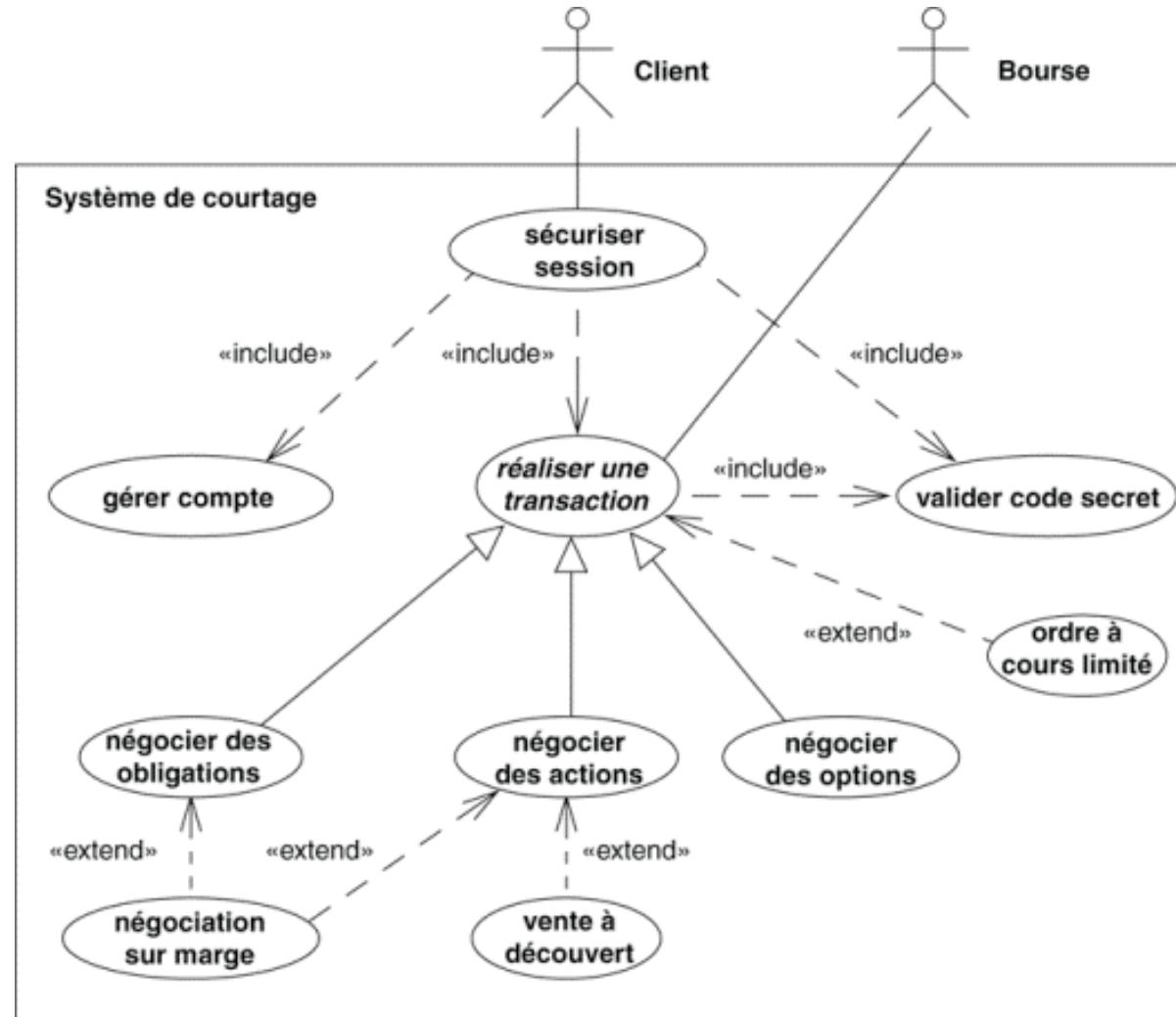
Cas d'utilisation : est une partie cohérente des fonctionnalités qu'un système peut fournir en interagissant avec les acteurs.

MODÉLISATION DES INTERACTIONS : DIAGRAMME DE CAS D'UTILISATION

- Un rectangle contient les cas d'utilisation du système.
- Un nom dans une ellipse représente un cas d'utilisation. L'icône d'un bonhomme représente un acteur
- Des lignes connectent les cas d'utilisation aux acteurs qui y participent



MODÉLISATION DES INTERACTIONS : RELATION ENTRE CAS D'UTILISATION



MODÉLISATION DES INTERACTIONS : EXEMPLE DE CAS D'UTILISATION

Cas d'utilisation : Jouer une partie.

Résumé : Le joueur lance une partie

Acteurs : Joueur :

Précondition : Il n'y a pas de partie en cours.

Description : Le jeu est dans l'état « Menu », dans lequel il affiche la liste des options possibles. Le joueur sélectionne l'item Jouer. Le programme initialise le jeu et lance la partie jusqu'à ce que le joueur est fini. Puis si le joueur a fini un bon score enregistre son score dans la liste des meilleurs scores.

Exceptions : Le joueur appuie sur la touche « Escape », le programme affiche le menu avec la possibilité de changer les options, sauvegarder, afficher les meilleurs scores, ou reprendre la partie.

Post Conditions : Le programme est dans l'état « Menu » et il n'y a pas de partie en cours.

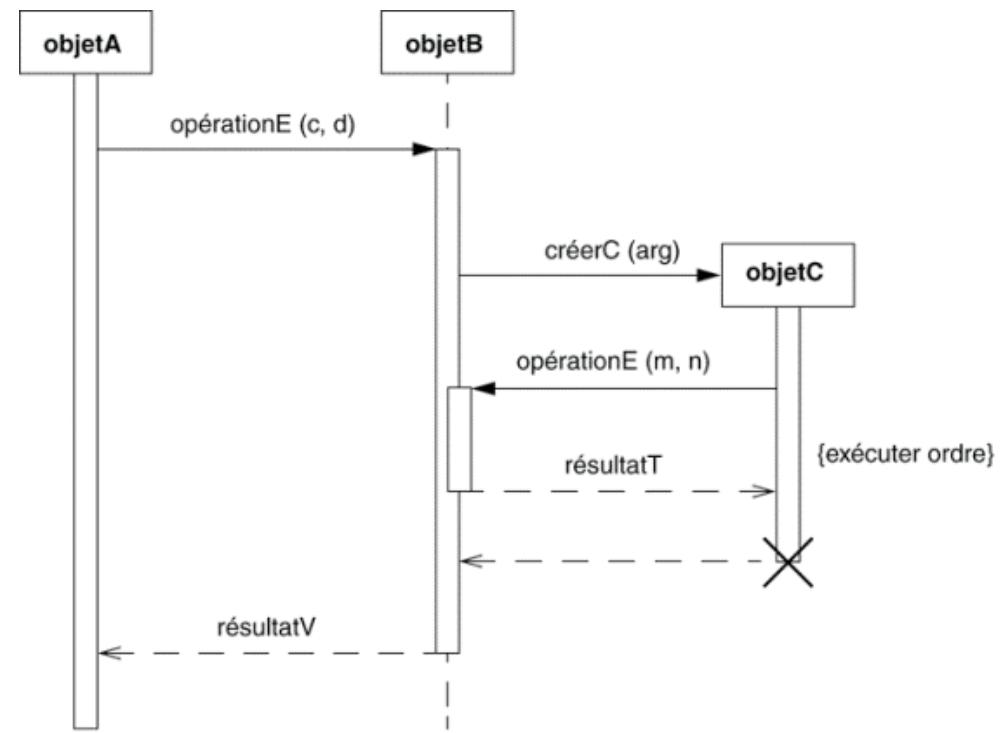
MODÉLISATION DES INTERACTIONS : DIAGRAMME DE SÉQUENCE

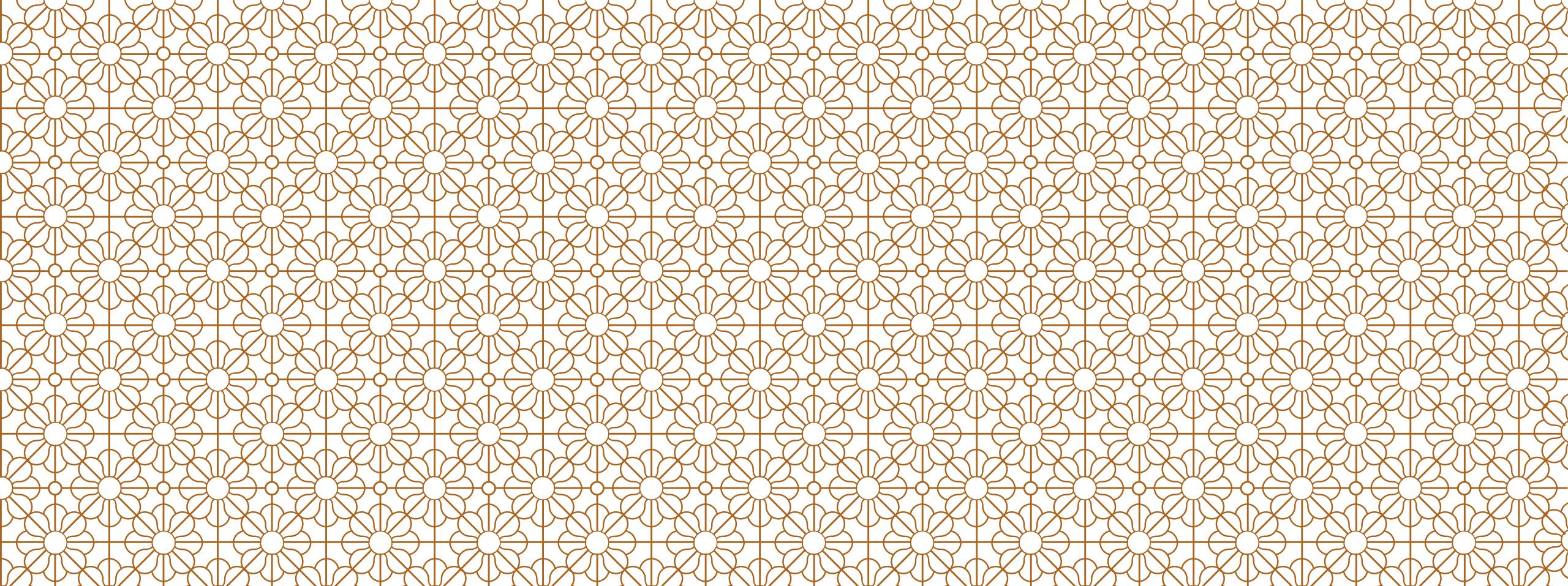
Fournissent plus de détail et représentent les messages que s'échange un ensemble d'objets au fil du temps. Les messages comprennent à la fois les signaux asynchrones et les appels de procédures.

- Objets actifs
- Objets passifs
- Objets temporaires

MODÉLISATION DES INTERACTIONS : DIAGRAMME DE SÉQUENCE

- Les objets sont représentés par des lignes verticales appelées lignes de vie et chaque message est symbolisé par une flèche horizontale allant de l'émetteur au récepteur.
- Le temps s'écoule sur l'axe vertical, du haut vers le bas, sans échelle de temps.
- Les diagrammes de séquence peuvent contenir des signaux concurrents.
- La description du comportement de chaque cas d'utilisation nécessite plusieurs diagrammes de séquences





DESIGN PATTERN

PROBLÈMES DE CONCEPTION CLASSIQUES

- Créer un objet en spécifiant explicitement une classe :
 - Le programme est assujetti à une implémentation spécifique.
- Assujettissement à une opération particulière :
 - Astreint à une forme unique de réponse à une méthode.
- Dépendance vis-à-vis de la plateforme matérielle et logicielles :
 - Difficile à maintenir et à porter.
- Assujettissement de la représentation d'un objet ou à son code :
 - Le programme est assujetti à une implémentation spécifique.
- Assujettissement à un algorithme:
 - Difficile de faire évoluer le programme
- Couplage fort
 - Réutilisation difficile d'une sous partie de l'architecture
- Extension des fonctionnalités pas sous-classes
 - Multiplication des classes dans l'architecture

LES PATTERNS À LA RESCOUSSE

Un modèle décrit un problème que l'on rencontre régulièrement lors de l'élaboration d'un logiciel. Il apporte à ce problème une solution élégante, c'est-à-dire:

- Flexible
- Extensible
- Réutilisable
- Maintenable

Il existe trois groupes de pattern:

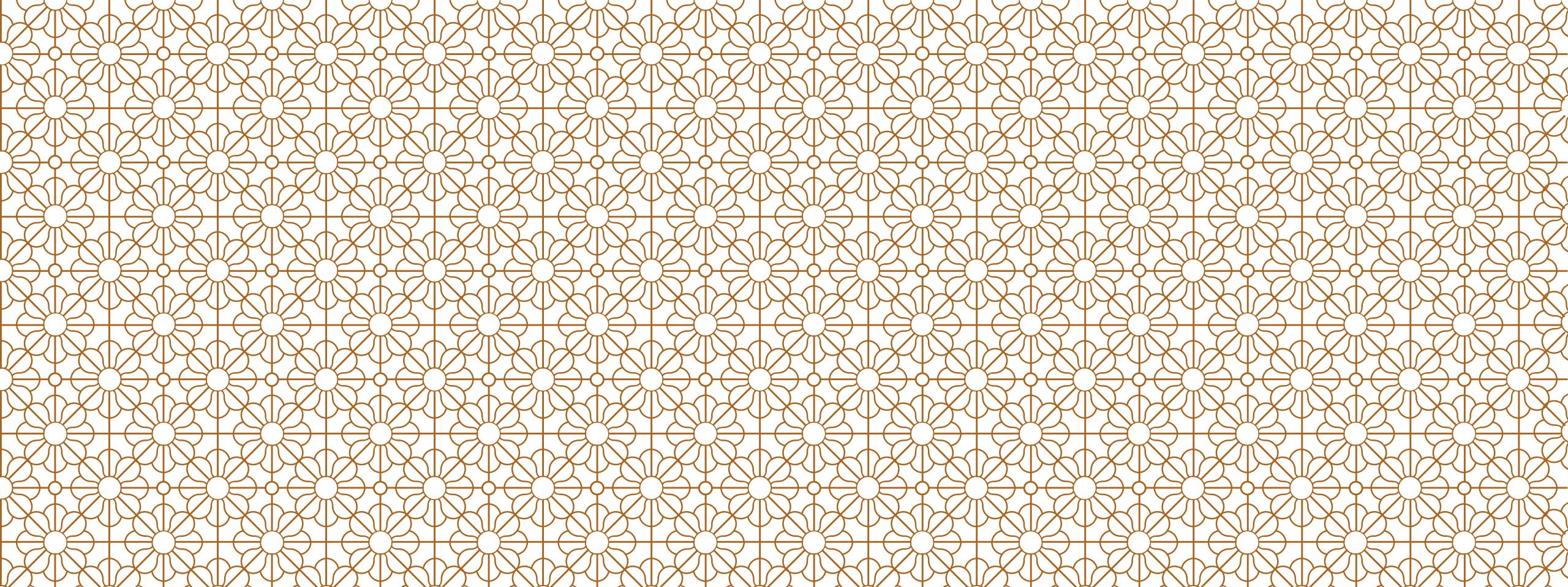
- Les modèles de création
- Les modèles structuraux
- Les modèles de comportement

MODÈLES DE CRÉATION

Les modèles de création permettent de modéliser le processus d'instanciation. Ils sont particulièrement intéressants dans les architectures basées sur la composition d'objet.

« Dans le cas de la composition d'objet, plein de petits objets peuvent être combinés ensemble, le processus d'instanciation devient alors plus compliqué qu'un simple new. »

- Fabrique abstraite (Abstract factory)
- Monteur (Buldeur)
- Fabrication (Method Factory)
- Prototype (Prototype)
- Singleton (Singleton)



FABRIQUE ABSTRAITE

FABRIQUE ABSTRAITE

EXEMPLE: JEU DE POKER

Nous voulons créer une architecture qui permet de modéliser un jeu de poker. Pour cela, nous devons être capable de créer une table, des cartes et de jetons spécifique à chaque Casino.

Comment faire pour que notre code ne soit pas assujetti au Casino que l'on utilise ?

Biarritz



Bordeaux



Las Vegas



FABRIQUE ABSTRAITE

Intention :

- La fabrique abstraite fournit une interface pour la création de famille d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes

Indication d'utilisation :

- Un système doit être indépendant de la façon dont ses produits ont été créés, combinés et représentés.
- Un système doit être constitué à partir d'une famille de produits, parmi plusieurs.
- On souhaite renforcer le caractère de communauté d'une famille d'objet produit pour être utilisés ensemble.

FABRIQUE ABSTRAITE

Constituant

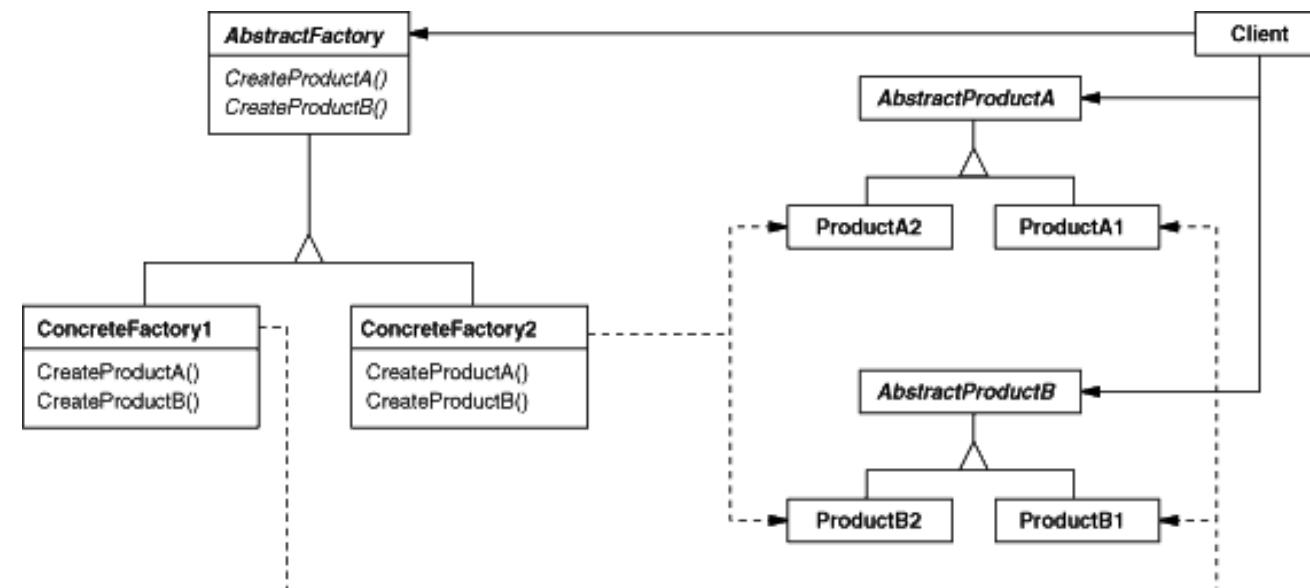
Fabrique abstraite (Casino) : déclare une interface contenant les opérations de création d'objets produits abstrait.

Fabrique concrète (CasinoBordeaux, CasinoBiarritz) : Implémente les opérations de création d'objet produit concrets

Produit abstrait : Déclare une interface pour un type d'objet produit.

Produit concret (Jeton Paris, Table Biarritz) : Définit un objet produit qui doit être créé par la fabrique concrète correspondante. Implémente l'interface produit abstrait.

Client : Il n'utilisera que les interfaces déclarées par les classes FabriqueAbstraite et ProduitAbstrait.



FABRIQUE ABSTRAITE

Conséquence d'utilisation :

- Il isole les classes concrètes.
- Il facilite la substitution de familles de produits
- Il favorise le maintien de la cohérence entre les objets
- Gérer de nouveaux produit est difficile (modification de toute l'architecture)

IMPLEMENTATION

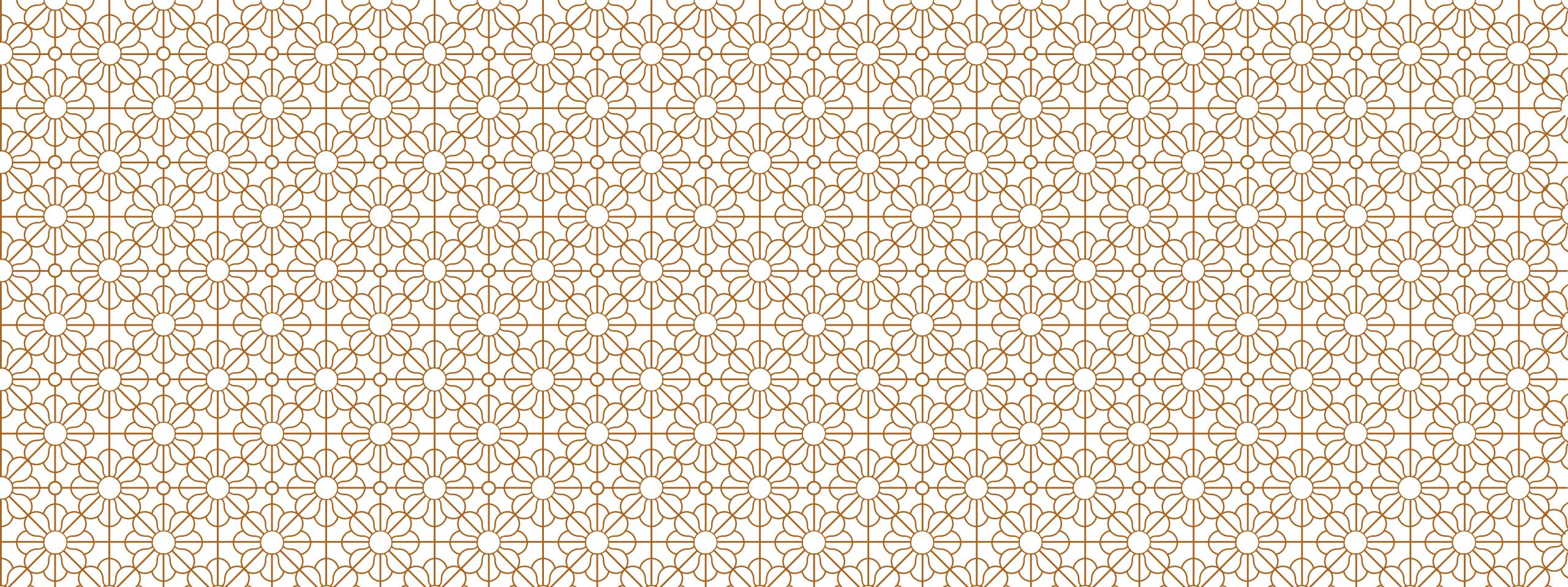
```
public interface Casino {  
    Table createTable();  
    DeckOfCard createDeckOfCard();  
    Chips createChips();  
}
```

```
class CasinoBordeaux implements Casino {  
    @Override  
    public Table createTable() {  
        return new TableBordeaux();  
    }  
    @Override  
    public DeckOfCard createDeckOfCard() {  
        return new DeckOfCardBordeaux();  
    }  
    @Override  
    public Chips createChips() {  
        return new ChipsBordeaux();  
    }  
}
```

IMPLEMENTATION

```
public static void main(String argv) {  
    Casino cas = new  
    CasinoBordeaux();  
    //Casino cas = new CasinoBiarritz();  
    startGame(cas);  
}
```

```
public static void startGame(Casino c) {  
    Table t = c.createTable();  
    DeckOfCard deck =  
    c.createDeckOfCard();  
    deck.shuffle();  
    t.draw();  
    for (Player p: _players)  
        p.addChips(c.createChips());  
}
```



MONTEUR (BUILDER)

MONTEUR

EXEMPLE: CRÉATION DE TABLEAU

Nous avons un ensemble de données et nous voulons créer une architecture qui permet de fabriquer des tableaux dans différents formats. Par exemple un tableau au format Excel, un tableau au format CSV ou encore une interface graphique qui affiche ce tableau.

Comment faire pour que notre code ne soit pas assujetti au type d'objet qu'il construit ?

Month	Target	Actual
January	6	
February	3	
March	2	
April	3	

January, 6
February, 3
March, 2
April, 3

```
<Worksheet ss:Name="Feuil1">
  <Table ss:ExpandedColumnCount="2" ss:ExpandedRowCount="4"
    x:FullColumns="1"
    x:FullRows="1" ss:DefaultColumnWidth="61.714285714285708"
    ss:DefaultRowHeight="14.571428571428571">
    <Row ss:Height="22.714285714285715">
      <Cell ss:StyleID="s64"><Data ss:Type="String">January</Data></Cell>
      <Cell><Data ss:Type="Number">6</Data></Cell>
    </Row>
```

MONTEUR

Intention :

- Dissocie la construction d'un objet complexe de sa représentation de sorte que le même processus de construction permette des représentations différentes.

Indication d'utilisation :

- L'algorithme de création d'un objet complexe doit être indépendant des parties qui composent l'objet et de la manière dont ces parties sont agencées
- Le processus de construction doit autoriser des représentations différentes de l'objet en construction.

BUILDER

Constituant

Monteur,

Le monteur spécifie une interface abstraite pour la création de parties d'un objet produit.

Monteur Concret,

Le monteur construit et assemble des parties du produit par l'implémentation de l'interface monteur.

Il définit la représentation qu'il crée et en conserve la trace.

Il fournit une interface pour la récupération du produit final.

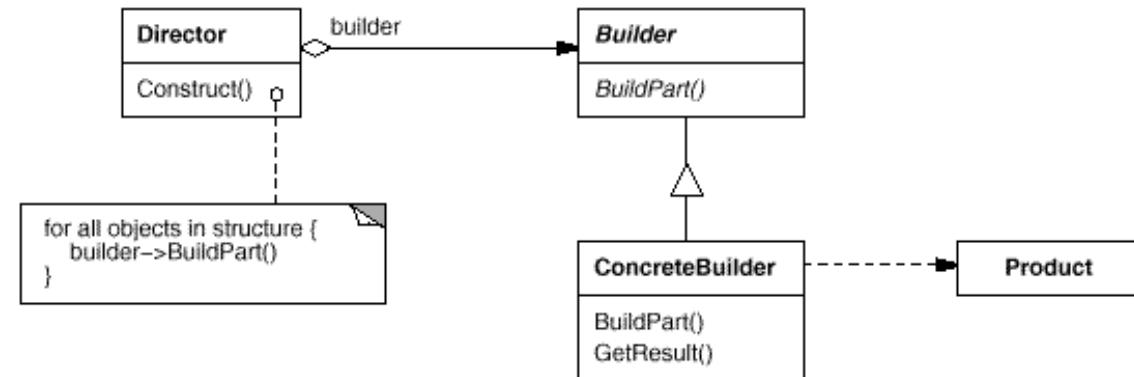
Directeur,

Le Directeur construit un objet en utilisant l'interface de Monteur.

Produit,

Le produit représente l'objet complexe en cours de construction. MonteurConcret construit la représentation interne du produit et définit le processus par lequel il est assemblé.

Il comporte les classes qui définissent les parties constitutives, y compris les interfaces nécessaires à l'assemblage des parties pour donner le résultat final.



BUILDER

Conséquence d'utilisation :

- Il permet de modifier la représentation interne d'un produit
- Il isole le code de construction et de représentation.
- Il permet un meilleur contrôle du processus de construction.

IMPLEMENTATION

```
1 #ifndef _RESULBUIDER_H
2 #define _RESULBUIDER_H
3
4 #include <string>
5 #include <Qt/qdatetime.h>
6
7 class ResultBuilder {
8 public:
9     virtual ~ResultBuilder() {}
10    virtual void beginDocument() = 0;
11    virtual void beginWorkSheet(const std::string&) = 0;
12    virtual void endWorkSheet() = 0;
13    virtual void beginRow() = 0;
14    virtual void endRow() = 0;
15    virtual void addCell(const double&, const unsigned int color) = 0;
16    virtual void addCell(const std::string&, const unsigned int color) = 0;
17    virtual void addCell(const QDateTime&, const unsigned int color) = 0;
18    virtual void endDocument() = 0;
19 };
20
21 #endif
22
```

IMPLEMENTATION

```
// void PKRPlayerStats::playerBankroll(std::string userName, ResultBuilder *document) {
    document->beginWorkSheet(string("bankroll"));
    document->beginRow();
    unsigned int style = 0xFFFFFFFF;
    document->addCell("hand_id", style);
    document->addCell("date", style);
    document->addCell("value", style);
    document->addCell("bets", style);
    document->endRow();

    vector<const HandInfo *> listH;
    const HandInfo *h;
    // forEach(h, data.getHands(startingTime, userName, position)) {
    //     const HandPlayerInfo &playInfo = h->getPlayerInfo(userName);
    //     //if (fabs(playInfo.gain) < 0.001) continue; //5. * h->blind ) continue;
    //     listH.push_back(h);
    // }
    CmpDate cmp;
    sort(listH.begin(), listH.end(), cmp);
    vector<const HandInfo *>::const_iterator it = listH.begin();

    double sumGain = 0;
    int discret = listH.size() / 200;
    size_t count = 0;
    double sumBets = 0;

    for(;it!=listH.end();++it) {
        const HandInfo *h = *it;
        const HandPlayerInfo &playInfo = h->getPlayerInfo(userName);
        double gain = playInfo.gain;
        if (count%discret == 0 && count != listH.size()-1) {
            document->beginRow();
            stringstream tmp;
            tmp << h->id;
            document->addCell(tmp.str(), style);
            document->addCell(h->startDate, style);
            document->addCell(sumGain, style);
            document->addCell(sumBets, style);
            document->endRow();
        }
        ++count;
        sumGain += gain;
    }
}
```

IMPLEMENTATION

```
#include <sstream>
using namespace std;

class CSVBuilder : public ResultBuilder {
    stringstream result;

public:
    virtual void beginDocument() {
    }

    virtual void beginWorkSheet(const string &) {
    }

    virtual void endWorkSheet() {
    }

    virtual void beginRow() {
    }

    virtual void endRow() {
        result << endl;
    }

    virtual void addCell(const QDateTime& d, const unsigned int ) {
        result << d.toString("yyyy-'MM'-'dd'T'hh':mm':ss'.0").toStdString() << " ";
    }

    virtual void addCell(const double& d, const unsigned int ) {
        result.precision(4);
        result << showpoint << d << " ";
    }

    virtual void addCell(const string& str, const unsigned int ) {
        result << str << " ";
    }

    virtual void endDocument() {
    }

    string getDocument() {
        return result.str();
    }
};

#endif // CSVBUILDER_H
```

CSVBuilder.h

IMPLEMENTATION

```
#ifndef _QtTableBuilder
#define _QtTableBuilder

#include <Qt/qtablewidget.h>
#include <Qt/qbrush.h>
#include <iomanip>
#include "ResultBuilder.h"

using namespace std;

class QDoubleItem : public QTableWidgetItem {
public:
    QDoubleItem(const char *str) :
        QTableWidgetItem(str) {
    }
    bool operator<(const QTableWidgetItem &other) const {
        double thisVal = atof(this->text().toStdString().c_str());
        double otherVal = atof(other.text().toStdString().c_str());
        return thisVal < otherVal;
    }
};

template<typename TYPE> void addCell(int i, int j, TYPE val, QTableWidget *table, const QBrush &style) {
    stringstream tmp;
    tmp << setw(4) << val;
    QTableWidgetItem *newItem = new QTableWidgetItem(tmp.str().c_str());
    newItem->setBackground(style);
    table->setItem(i, j, newItem);
}

template<> void addCell<double>(int i, int j, double val, QTableWidget *table, const QBrush &style) {
    stringstream tmp;
    tmp << setw(4) << floor(10000. * val) / 10000. ;
    QTableWidgetItem *newItem = new QDoubleItem(tmp.str().c_str());
    newItem->setBackground(style);
    table->setItem(i, j, newItem);
}

class QtTableBuilder : public ResultBuilder {
    static const unsigned int MAX_SIZE = 10000;
    QTableWidget *table;
    int nbRows, nbCols;
    int i, j;
    map<string, QBrush> style;
public:
    QtTableBuilder(QTableWidget * table) :
        table(table) {
```

QtTableBuilder.h

IMPLEMENTATION

```
ifndef _XmlBuilder
#define _XmlBuilder

#include <fstream>
#include "ResultBuilder.h"
#include <iomanip>
#include <vector>

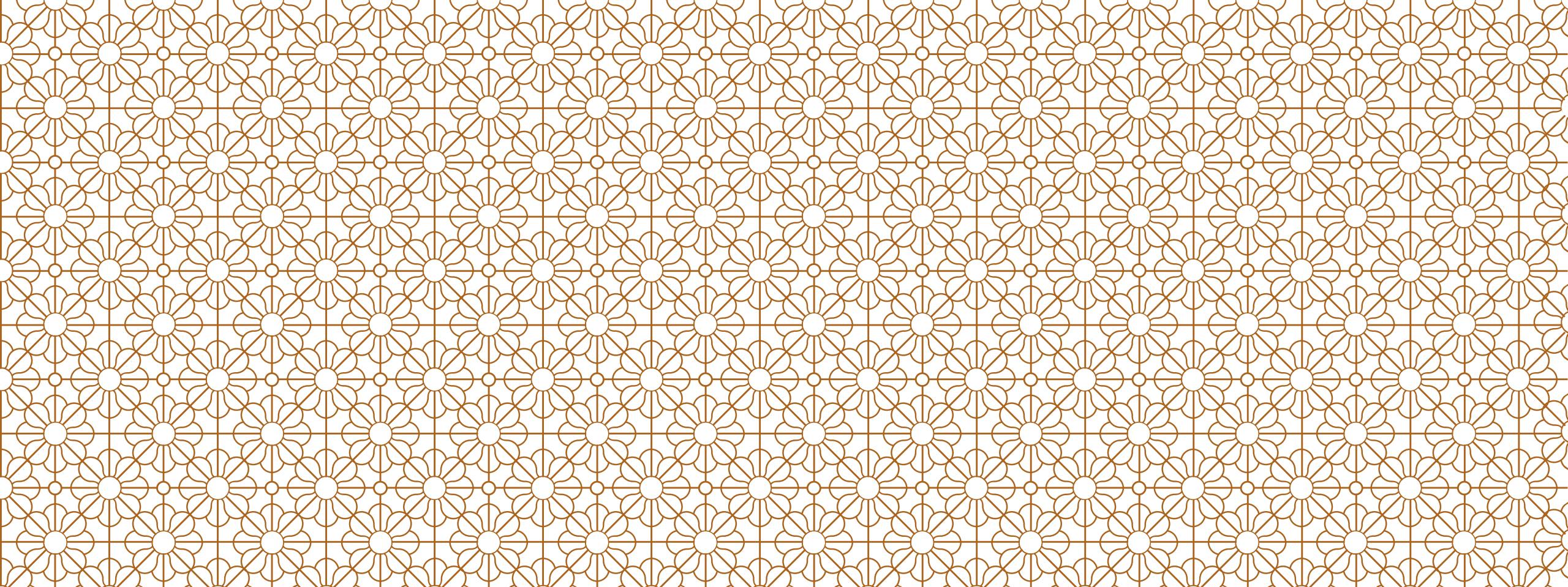
using namespace std;

class XmlBuilder : public ResultBuilder {
    static const unsigned int MAX_SIZE = 10000;
    stringstream result;
    stringstream table;
    stringstream header;
    stringstream footer;

    set<unsigned int> used_colors;
    //style
    void generateStyle(ostream &os) {
        os << "<Styles>" << endl;
        set<unsigned int>::const_iterator it = used_colors.begin();
        for (; it!=used_colors.end(); ++it) {
            os << std::hex;
            os << "<Style ss:ID=" << *it << std::hex << "\">" << endl;
            os << "<Interior ss:Color="#" << *it << std::hex << "\\" ss:Pattern="Solid\\"/>" << endl;
            os << "<NumberFormat ss:Format="#" />" << endl;
            os << "</Style>" << endl;
            os << "<Style ss:ID=" << *it << std::hex << "\">" << endl;
            os << "<Interior ss:Color="#" << *it << std::hex << "\\" ss:Pattern="Solid\\"/>" << endl;
            // os << "<NumberFormat ss:Format="Fixed\\"/>" << endl;
            os << "</Style>" << endl;
            os << "<Style ss:ID=" << *it << std::hex << "\">" << endl;
            os << "<Interior ss:Color="#" << *it << std::hex << "\\" ss:Pattern="Solid\\"/>" << endl;
            os << "<NumberFormat ss:Format="Short Date\\"/>" << endl;
            os << "</Style>" << endl;
        }
        os << "</Styles>" << endl;
    }

public:
    virtual void beginDocument() {
        std::ifstream in("header.xml");
        while (!in.eof()) {
            char line[MAX_SIZE];
            in.getline(line, MAX_SIZE);
            string lines(line);
            //cout << lines << endl << flush;
            header << lines << endl;
        }
        in.close();
    }
}
```

XmlBuilder.h



SINGLETON

SINGLETON

EXEMPLE: TEST BUFFERISÉ ALGORITHME DE GRAPHE

Nous avons un algorithme qui s'applique sur une structure de donnée et nous voulons bufferiser son résultat. Cependant, lorsque la structure de donnée change nous ne devons dévalider le buffer.

SINGLETON

Intention :

- Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe.

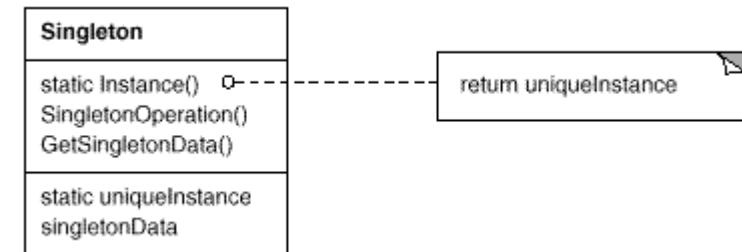
Indication d'utilisation :

- S'il doit y avoir qu'une seule instance d'une classe, qui de plus doit être accessible aux clients en un point particulier.
- Si l'instance unique doit être extensible par dérivation en sous-classe et si l'utilisation d'une instance étendue doit être permise aux clients sans qu'ils aient besoin de modifier leur code.

SINGLETON

Constituant

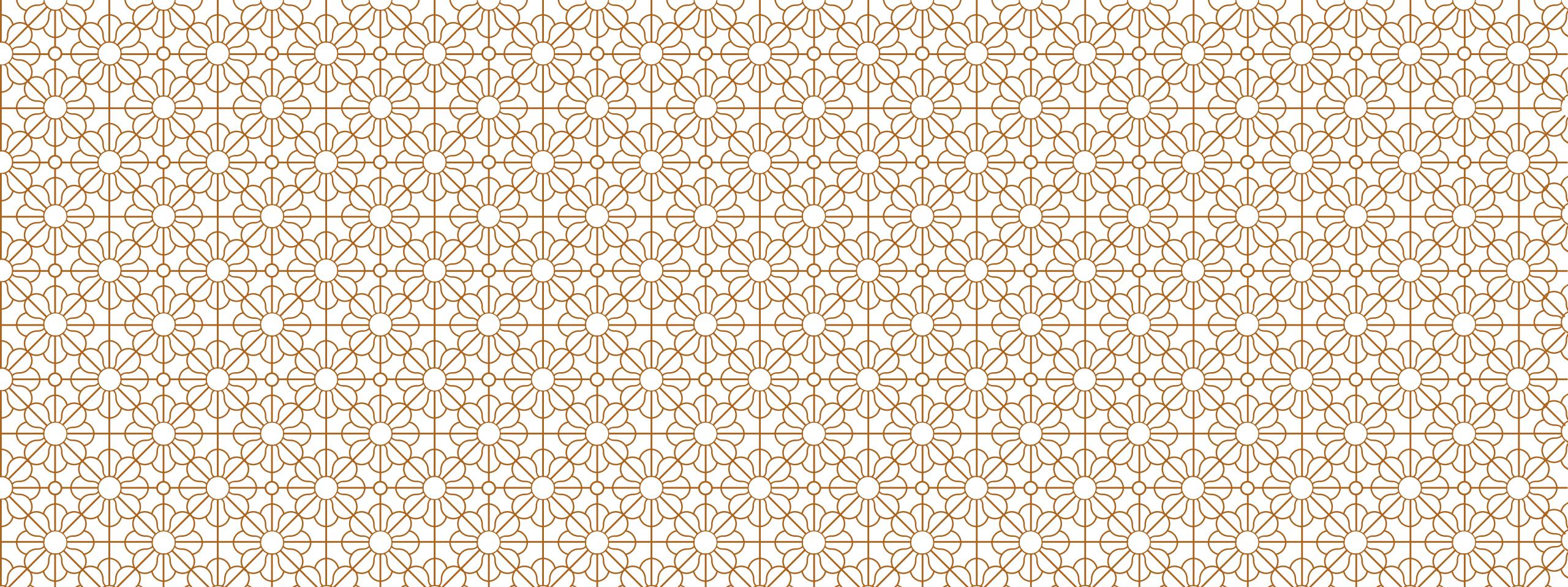
Singleton, définit une opération Instance qui donne au client l'accès à son unique instance.



SINGLETON

Conséquence d'utilisation :

- Accès contrôlé à une instance unique
- Réduction de l'espace de nom
- Raffinement des opérations et de la représentation
- Autorise un nombre variable d'instance
- Souplesse amélioré par rapport aux opérations de classes.



PROTOTYPE

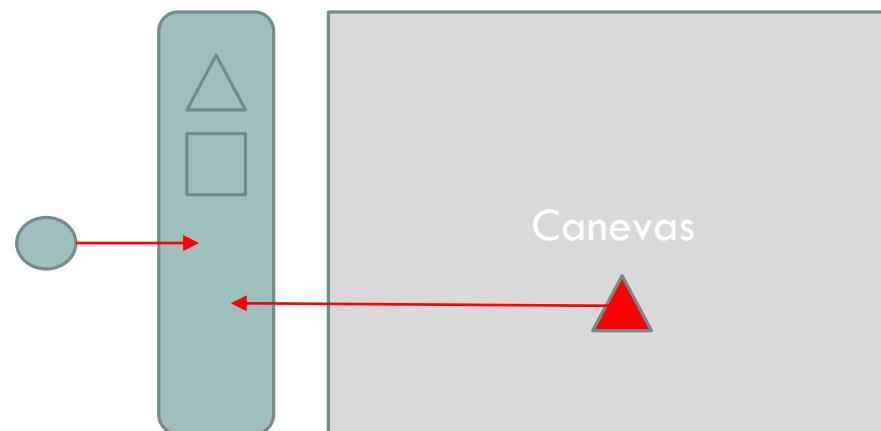
PROTOTYPE

EXEMPLE: BARRE D'OUTILS DE DESSIN

Nous voulons créer une logiciel de dessin qui fonctionne en déplaçant des forme de la barre d'outils vers le « canevas ». Une fois placer, nous voulons changer les attributs des formes.

Comment faire pour ne pas modifier le code de la barre d'outils à chaque fois que l'on veut y ajouter une forme.

Comment faire pour ajouter dans la barre d'outils des formes dont on a changé les propriétés?



PROTOTYPE

Intention :

- Spécifie le type des objets à créer à partir d'une instance de prototype et crée de nouveaux objets en copiant ce prototype.

Indication d'utilisation :

- Le système doit être indépendant de la manière dont ses produits sont créés, composés et représentés.
- Si les classes à instancier sont spécifiée à l'exécution, (chargement dynamique).
- Pour éviter de créer une hiérarchie de classe de fabriques, qui réplique la hiérarchie de classes de produits.
- Si les instances d'une classe peuvent prendre un état parmi un petit nombre.

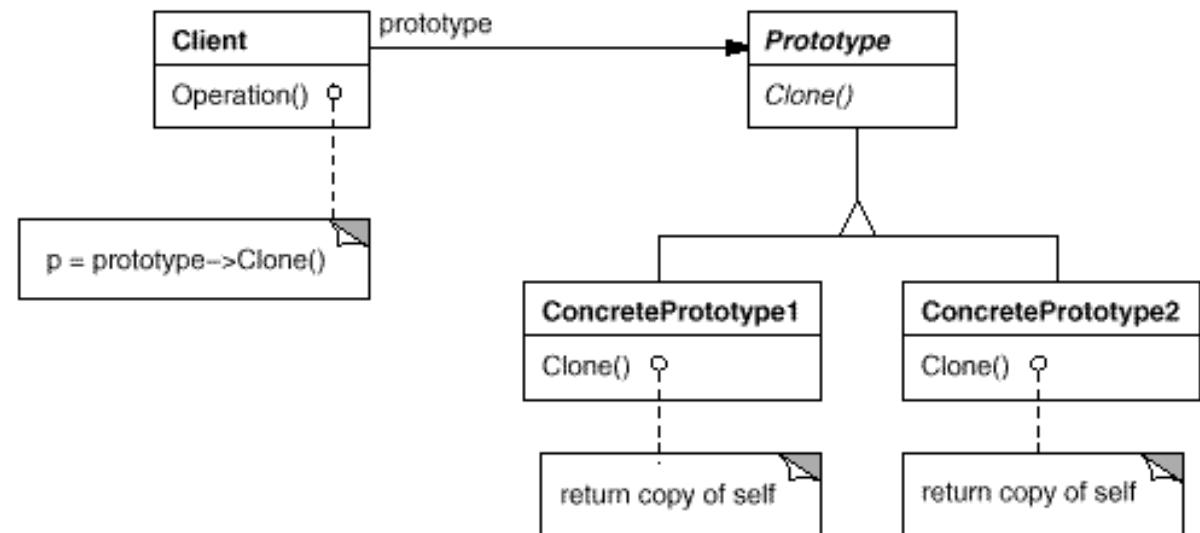
PROTOTYPE

Constituant

Prototype : déclare une interface pour se cloner lui-même

Prototype Concret: Le prototype concret implémente une opération capable de se cloner elle-même

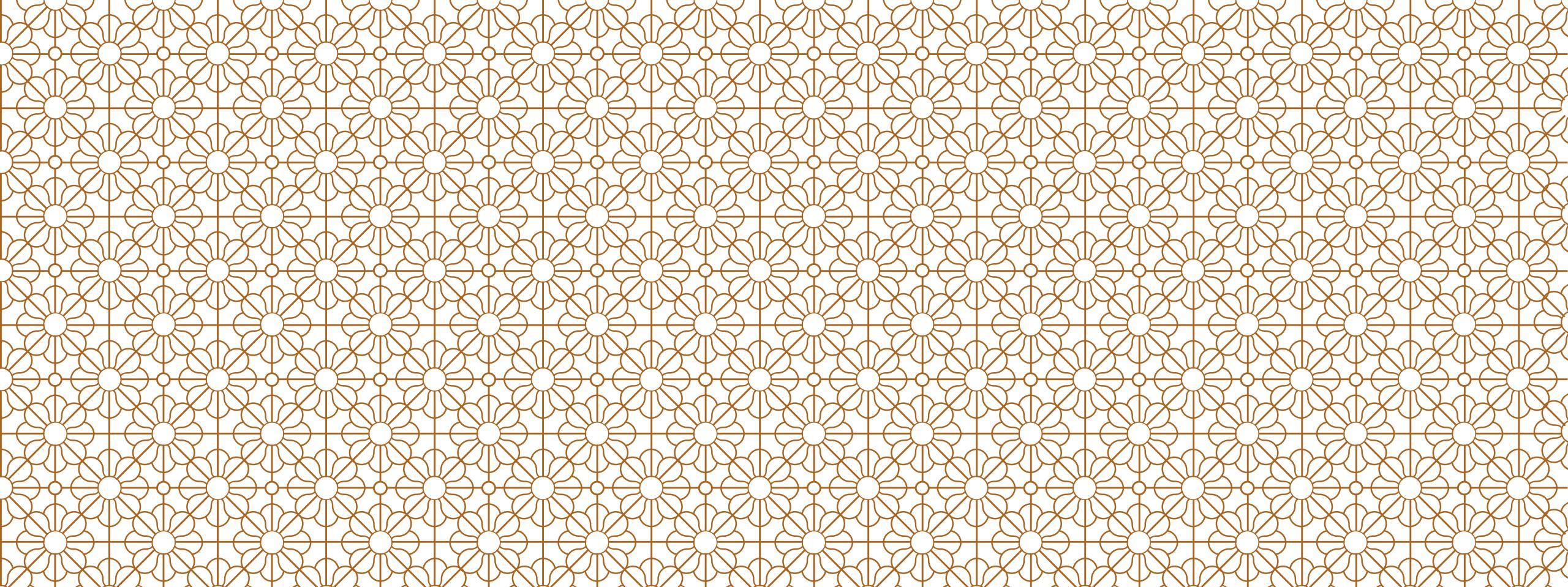
Client : Le client crée un nouvel objet en demandant au prototype de se cloner lui-même



PROTOTYPE

Conséquence d'utilisation :

- Addition et suppression de produit à l'exécution.
- Spécification de nouveaux objets par valeur modifiables.
- Spécification de nouveaux objets par structure modifiable.
- Limitation du nombre de dérivation de classes.
- Configuration dynamique d'une application avec des classes.

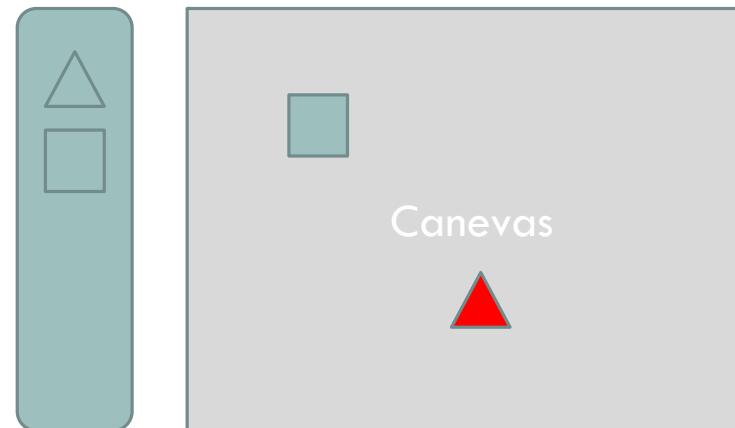


FABRICATION

METHOD FACTORY

EXEMPLE: EDITEUR DE PROPRIÉTÉS

Nous voulons ajouter à notre implémentation de forme précédente un mécanisme d'édition de propriété. Lorsque l'on clique bouton droit sur une figure dans le canevas un fenêtre spécifique à chaque forme doit s'ouvrir



FABRICATION

Intention :

- Définit une interface pour la création d'un objet, mais en laissant les sous-classes le choix des classes à instancier.
La fabrication permet à une classe de déléguer l'instanciation à une sous classe.

Indication d'utilisation :

- Une classe ne peut prévoir la classe des objets qu'elle aura à créer.
- Une classe attend des ses sous-classes qu'elles spécifient les objets qu'elles créent.
- Les classes déléguent des responsabilités à une des nombreuses sous-classes assistantes et l'on veut disposer localement de l'information permettant de connaître la sous-classe assistante qui a reçu cette délégation.

FABRICATION

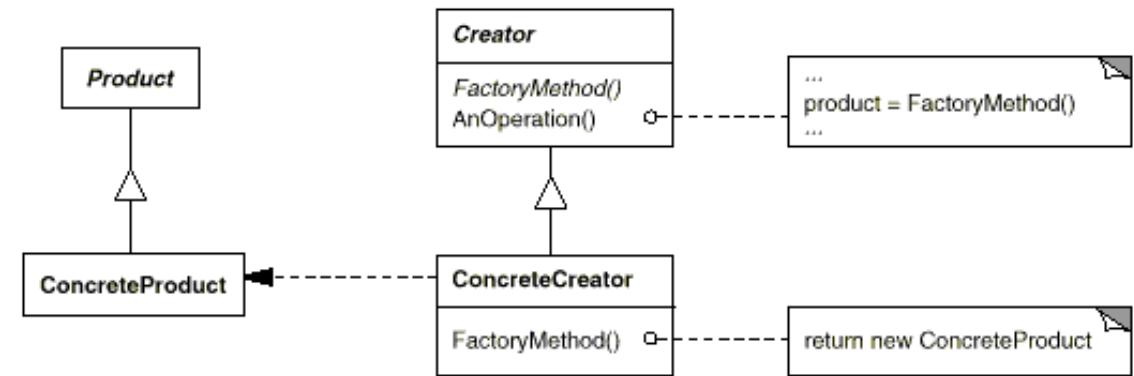
Constituant

Produit : Le produit définit l'interface des objets créés par la fabrication.

Produit Concret : Le produit concret implémente l'interface produit.

Facteur : Le Facteur déclare la fabrication ; celle-ci renvoie un objet de type Produit. Le facteur peut également définir une implémentation par défaut de la fabrication, qui renvoie un objet ProduitConcret par défaut.

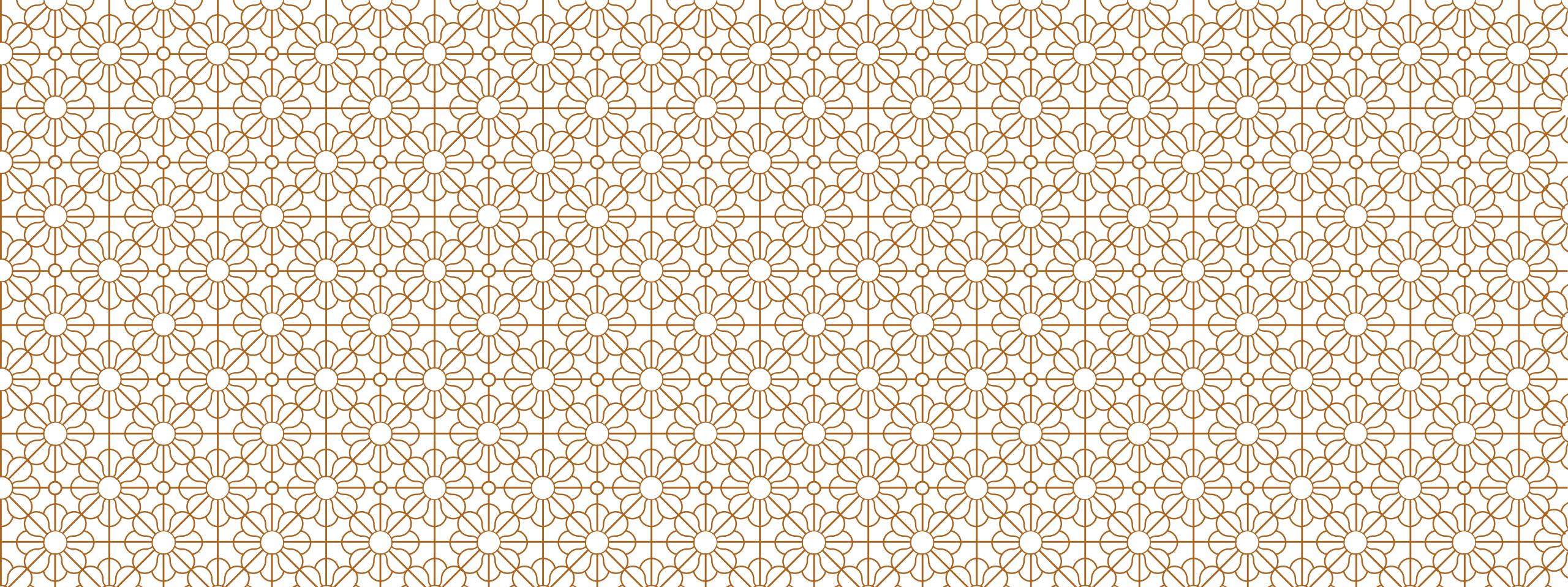
FacteurConcret : Le FacteurConcret surcharge la fabrication pour renvoyer une instance du produit concret.



FABRICATION

Conséquence d'utilisation :

- Indépendance du code vis-à-vis des classes spécifique de l'application.
- Besoin de dériver la classe facteur à chaque fois que l'on veut faire un produit concret.
- Il procure un gîte aux sous-classes.
- Il interconnecte des hiérarchies parallèles.



DECORATOR

DECORATEUR

EXEMPLE: CRYPTAGE/DECRYPTAGE TRANSPARENT

On veut créer une architecture permettant de travailler avec des fichiers ou des chaînes de caractères de manière transparente.

Ces fichiers ou chaînes de caractères pourront ensuite être cryptés / décryptés avec différents algorithmes et ces algorithmes pourront être composés les uns avec les autres.

Si nous avons 2 types de flux (fichier F et string S) et 2 algorithmes de cryptage (C1, C2) les possibilités sont les suivantes:

F (F non crypté)

S (S non crypté)

F C1 (F crypté avec C1)

S C1 (S crypté avec C1)

F C1 C2 (F crypté avec C1 puis C2)

S C1 C2 (S crypté avec C1 puis C2)

F C2 C1 (F crypté avec C2 puis C1)

S C2 C1 (S crypté avec C2 puis C1)

F C2 (F crypté avec C2)

S C2 (S crypté avec C2)

Comment éviter l'explosion combinatoire du nombre de classes à implémenter ?

DECORATOR

Intention :

- Attache dynamiquement des responsabilités à un objet. Les décorateurs fournissent une alternative souple à la dérivation, pour étendre les fonctionnalités.

Indication d'utilisation :

- Pour ajouter dynamiquement des responsabilités à des objets individuels, ceci d'une façon transparente, c.à.d sans affecter les autres objets.
- Pour des responsabilités qui doivent être retirées.
- Quand l'extension par dérivation est impraticable. Trop de combinaisons possibles.

DECORATOR

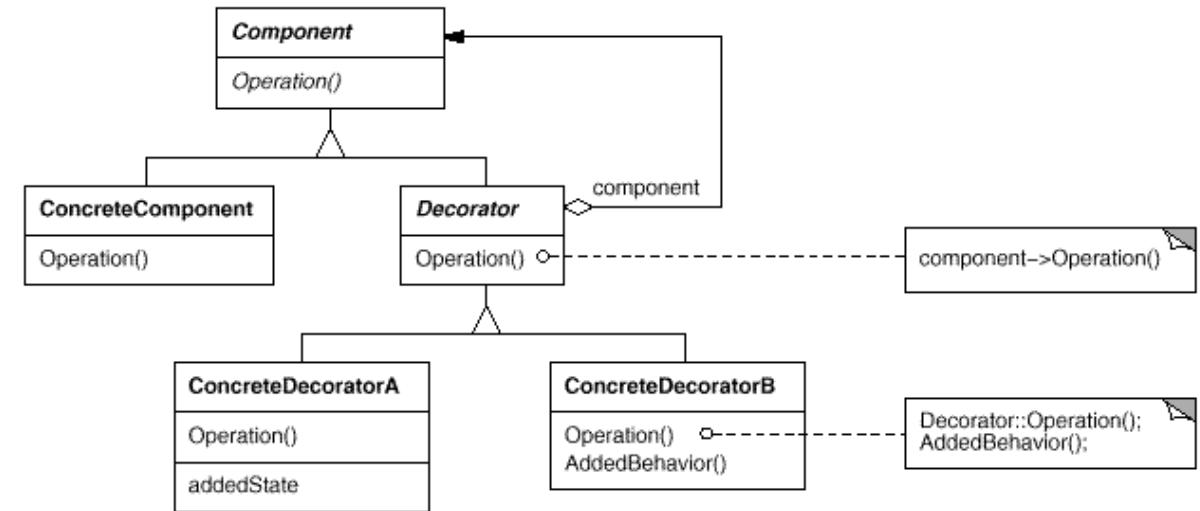
Constituant

Composant : définit l'interface des objets qui peuvent recevoir dynamiquement des responsabilités supplémentaires.

Composant concret : Définit un objet auquel des responsabilités peuvent être ajoutées.

Décorateur : Gère une référence à un objet composant et définit une interface conforme à celle du composant.

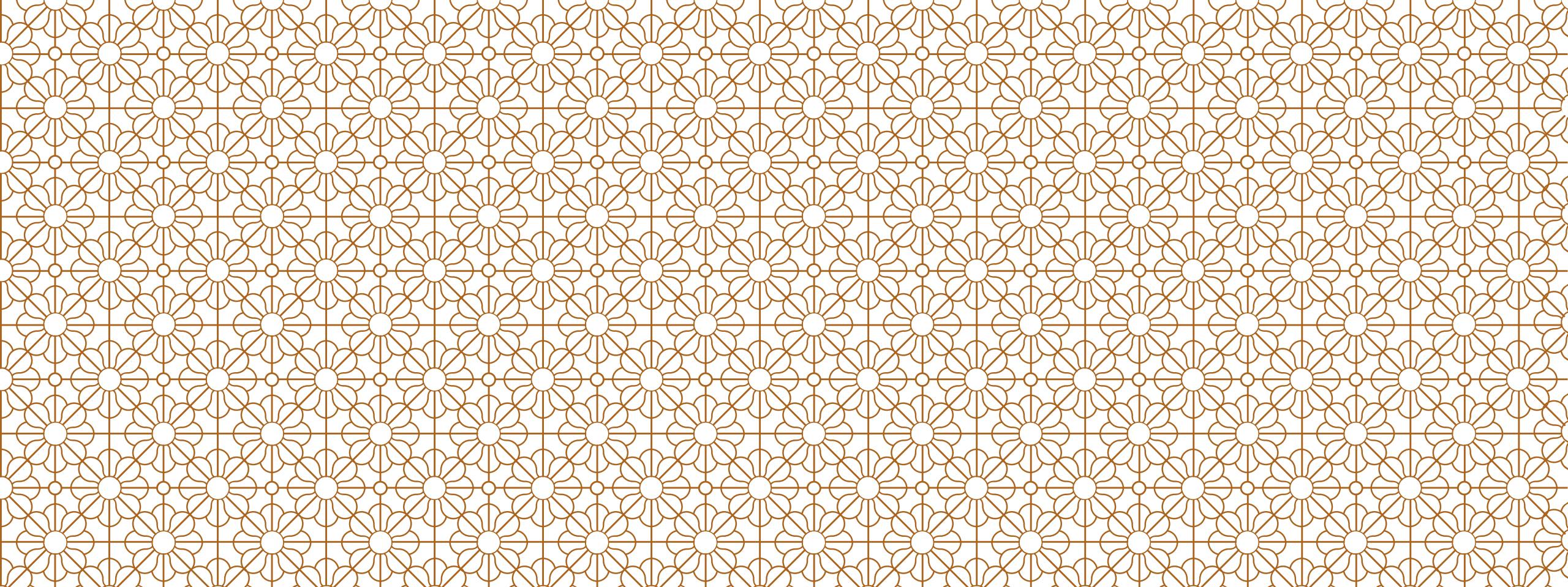
Décorateur concret : ajoute des responsabilités à un composant.



DECORATOR

Conséquence d'utilisation :

- Offre plus de souplesse que l'héritage statique
- Evite de surcharger des fonctionnalités les classes situées en haut de la hiérarchie.
- Un composant n'est pas identique
- Multitude petits objets, difficile à comprendre.



COMPOSITE

COMPOSITE

EXEMPLE: GESTION DE GROUPES D'ÉTUDIANTS

On veut créer une architecture permettant de stocker nos étudiants, les groupes d'étudiants ainsi que les promotions. Par exemple pour le Master nous aurons les étudiants, les groupe 1 et le groupe 2 et les parcours et enfin le master dans son entier.

Nous voulons chacun de nos objets être capable d'obtenir la note moyenne, l'effectif.

COMPOSITE

Intention :

- Le modèle composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même manière les objets individuels et les combinaisons de ceux-ci.

Indication d'utilisation :

- On souhaite représenter des hiérarchies de l'individu à l'ensemble
- On souhaite que le client n'ait pas à se préoccuper de la différence entre combinaisons d'objets et objets individuels.

COMPOSITE

Constituant

Composant :

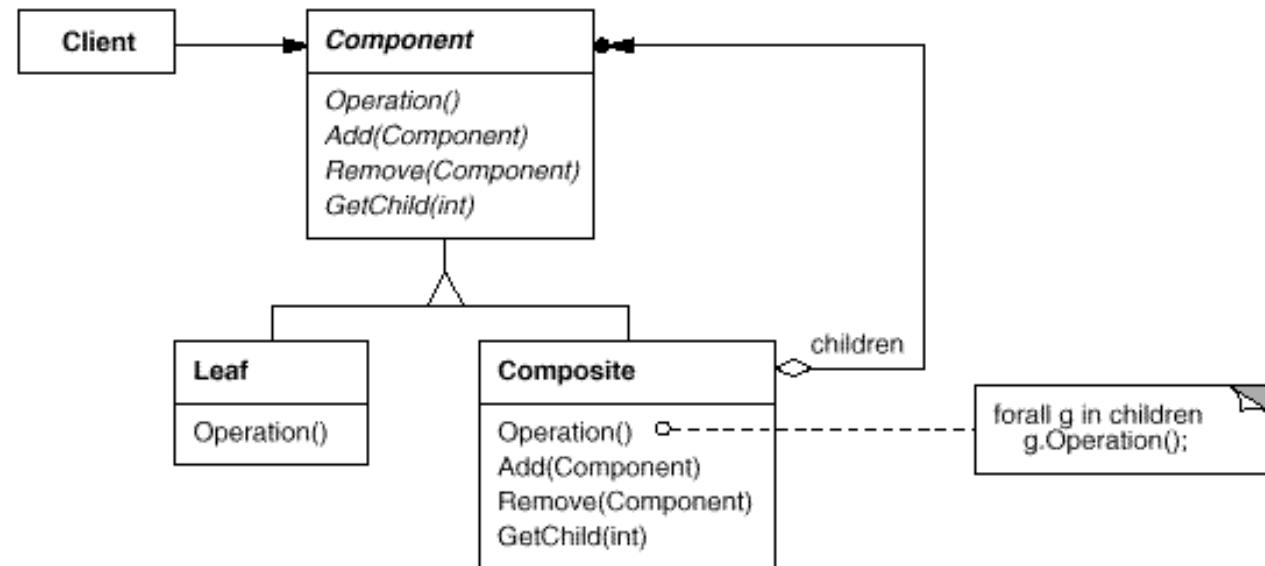
- Il déclare l'interface des objets entrant dans la composition.
- Il implémente le comportement par défaut qui convient pour l'interface commune à toutes les classes.
- Il déclare une interface pour accéder aux enfants
- Peut définir une interface pour accéder au parent.

Feuille : Définit le comportement d'objet primitives dans la composition.

Composite :

- Définit le comportement d'objet composant dotée d'enfants.
- Il stocke les composants enfants.
- Il implémente la gestion des enfants.

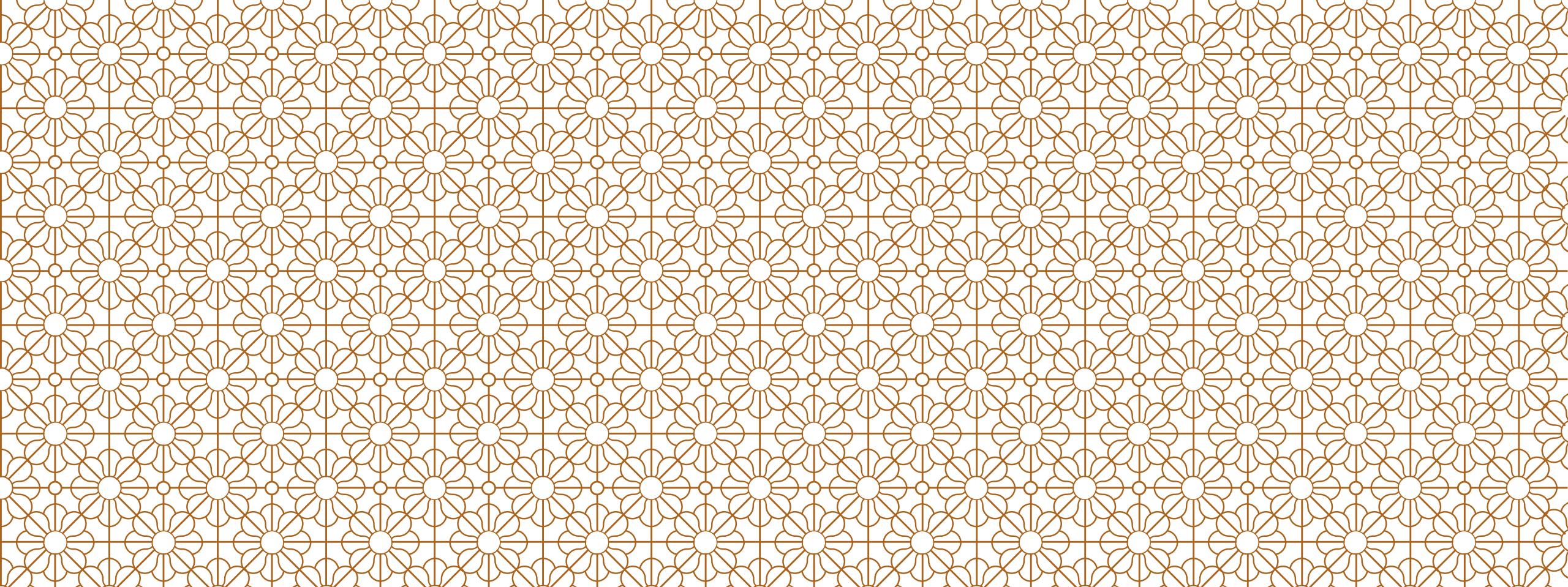
Client : Manipule les objets de la composition à l'aide de l'interface composant.



COMPOSITE

Conséquence d'utilisation :

- Définit des hiérarchies de classes consistant en des objets primitives et des objets composites.
- Simplifie le niveau client.
- Ajout de nouveau composant simplifié.
- - Difficile de contrôler ce qu'il y a dedans.



PROCURATION

PROCURATION

EXEMPLE: ALGORITHME AVEC MISE EN MÉMOIRE TAMPON

Nous voulons construire un ensemble de classes permettant de modéliser des algorithmes de calcul et de permettre à une application de les utiliser de manière transparente.

De plus nous voulons ajouter un mécanisme de mise en mémoire tampon des résultats de notre algorithme de tel sorte que le code d'un client ne soit pas impacté par le fait que l'on utilise un algorithme bufférisé ou non.

PROCURATION

Intention :

- Fournit à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

Indication d'utilisation :

- Procuration à distance : représentant local d'un objet situé à distance
- Procuration virtuelle : crée des objets lourds à la demande
- Procuration de protection : contrôle l'accès à l'objet original
- Référence intelligente : remplaçant d'un pointeur brut

PROCURATION

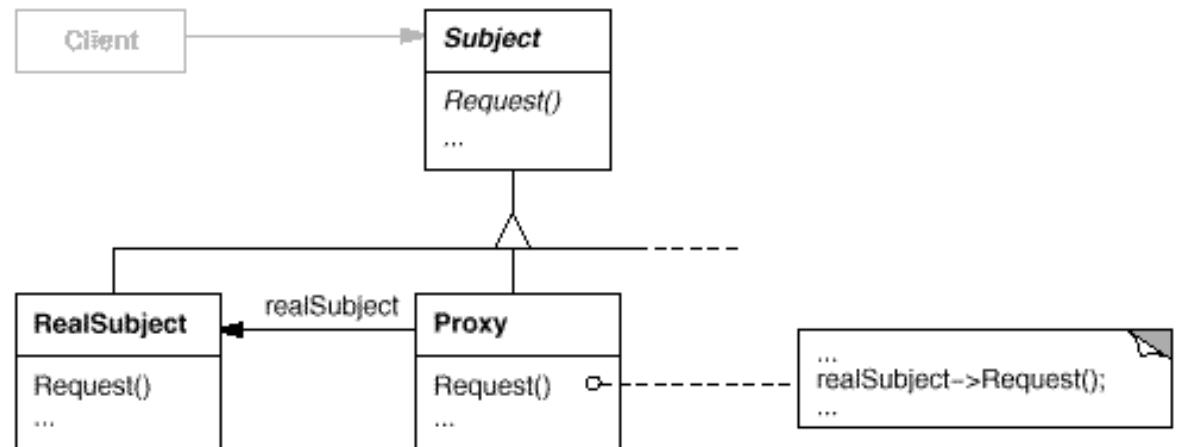
Constituant

Procuration :

- gère une référence qui le permet d'accéder au sujet réel.
- Procure une interface identique à celle du sujet
- Contrôle l'accès au sujet réel

Sujet :

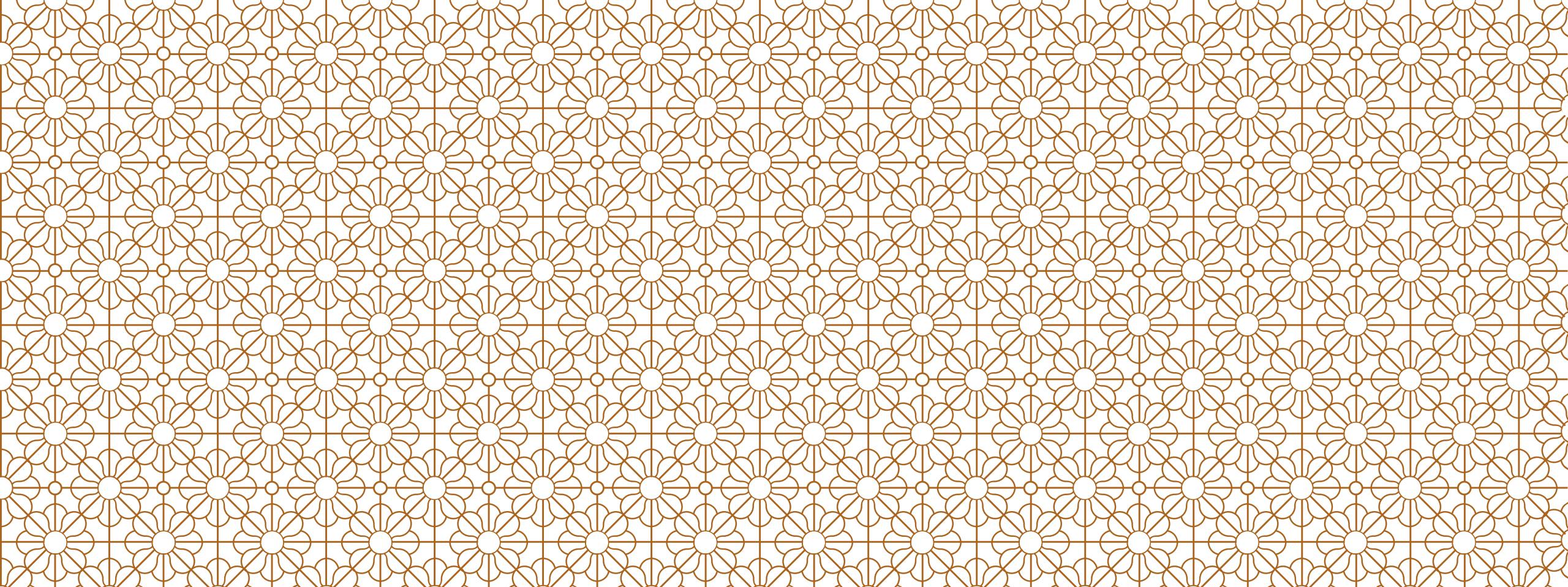
- définit une interface commune pour sujetRéel et procuration



PROCURATION

Conséquence d'utilisation :

- peut cacher le fait qu'un objet réside dans autre espace d'adresse
- optimisation
- sécurité.



ADAPTATEUR



ADAPTATEUR

EXEMPLE: ADAPTATION DE FORME EN SHAPE

Nous disposons d'une architecture permettant de modéliser des forme géométrique. Cette architecture est composée d'une interface Forme proposant les méthodes affiches(), déplace(double x, double y), et tourne(double degree).

Nous voudrions réutiliser cette hiérarchie de classes dans un autre projet qui doit offrir l'interface d'accès en anglais suivante : Shape, display(), move(double x, double y), rotate(double rad).

ADAPTATEUR

Intention :

- Convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adaptateur permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles.
- Concrètement, ce pattern nous permet de créer une interface pour un objet effectuant les actions dont nous avons besoin, mais n'utilisant pas l'interface qui nous convient.

ADAPTATEUR

Indication d'utilisation :

- On veut utiliser une classe existante dont l'interface ne coïncide pas avec celle escomptée.
- On souhaite créer une classe réutilisable qui collabore avec des classes sans relations avec elle et encore inconnues, c'est-à-dire avec des classes qui n'auront pas nécessairement des interfaces compatibles.
- Pour les adaptateurs d'objets, on a besoin d'utiliser plusieurs sous-classes existantes, mais l'adaptation de leur interface par dérivation de chacune d'entre elles est impraticable. Un adaptateur objet peut adapter l'interface de sa classe parente.

ADAPTATEUR DE CLASSE

Constituant

Adaptee:

- Classe que l'on veut adapter à l'interface Target.

Target :

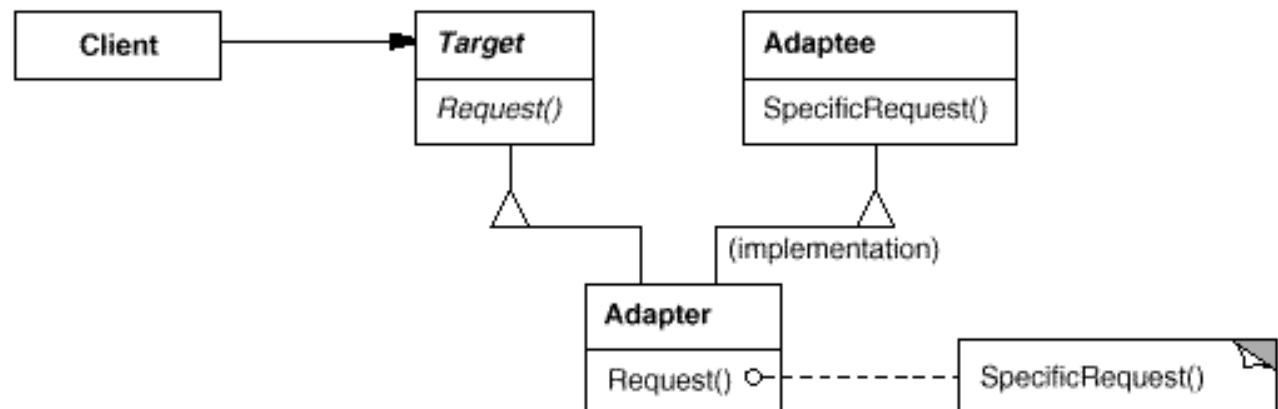
- Définit l'interface que le client doit utiliser.

Adapter :

- Implémente l'interface de l'interface Target en utilisant le code de Adaptee.

Client :

- Accède à l'implémentation de Adaptee via l'interface Target.



ADAPTATEUR D'OBJET

Constituant

Adaptee:

- Classe que l'on veut adapter à l'interface Target.

Target :

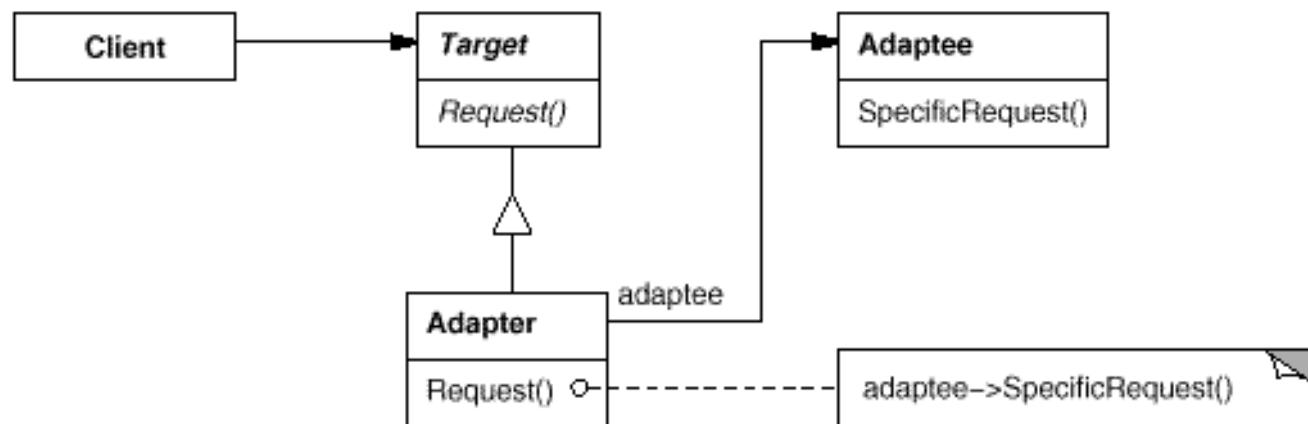
- Définit l'interface que le client doit utiliser.

Adapter :

- Implémente l'interface de l'interface Target en utilisant le code de Adaptee.

Client :

- Accède à l'implémentation de Adaptee via l'interface Target.



ADAPTATEUR DE CLASSE

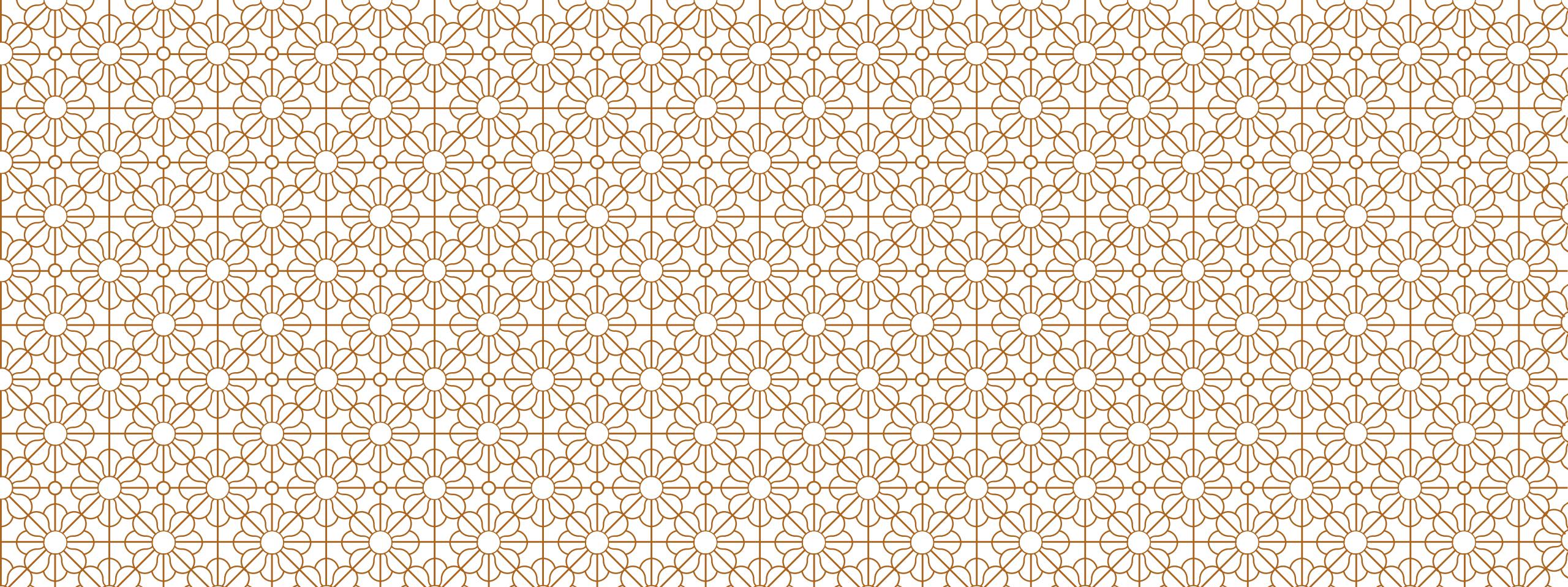
Conséquence d'utilisation :

- Adapte l'adapté en s'en remettant à une classe adaptateur concrète. Il en résulte qu'un adaptateur de classe ne marche pas si on veut adapter une classe et toutes ses sous-classes.
- Permet à un adaptateur de redéfinir certain des comportements de l'adapté, du fait que Adaptateur est une sous classe de Adapte.
- Introduit un seul objet et aucun pointeur additionnel n'est nécessaire pour atteindre adapte.

ADAPTATEUR D'OBJET

Conséquence d'utilisation :

- Permet à un simple adaptateur de travailler avec plusieurs adaptés ; c'est-à-dire, adapte lui-même et tout ses sous classes. L'adaptateur peut aussi ajouter des fonctionnalités à tous les adaptés en une seule fois.
- Rend plus difficile la surcharge du comportement de Adapte. Cela nécessite la dérivation de cette dernière, et impose que l'adaptateur fasse référence à la sous classe plutôt qu'à adapte lui-même.



PONT

PONT

EXEMPLE: SHAPE AVEC DIFFÉRENT MOTEUR DE RENDU

Nous disposons de hiérarchies de Shape qui s'affiche avec Swing. Nous voulons maintenant utiliser JavaFX.

Comment faire pour ne pas avoir à modifier le code existant si nous devons changer le moteur de rendu dans le futur. (Exemple rendu OpenGL).

PONT

Intention :

Découpe une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.

PONT

Indication d'utilisation :

- On souhaite éviter un lien définitif entre l'abstraction et son implémentation.
- Il faut que les abstractions et les implémentations soient toutes deux étendues par dérivation.
- Il ne faut pas que les modifications apportées à l'implémentation d'une abstraction aient un impact sur le code client, en particulier le code ne doit pas être recompilé.
- On souhaite cacher au client l'implémentation d'une abstraction.
- On est confronté à une prolifération de classes.
- On veut faire partager une même implémentation à plusieurs objets et cela doit être caché au client.

PONT

Constituant

Abastraction:

- Définit l'interface d'abstraction souhaitée
- Maintient un pointeur vers l'implémentation

RedefinedAbstraction :

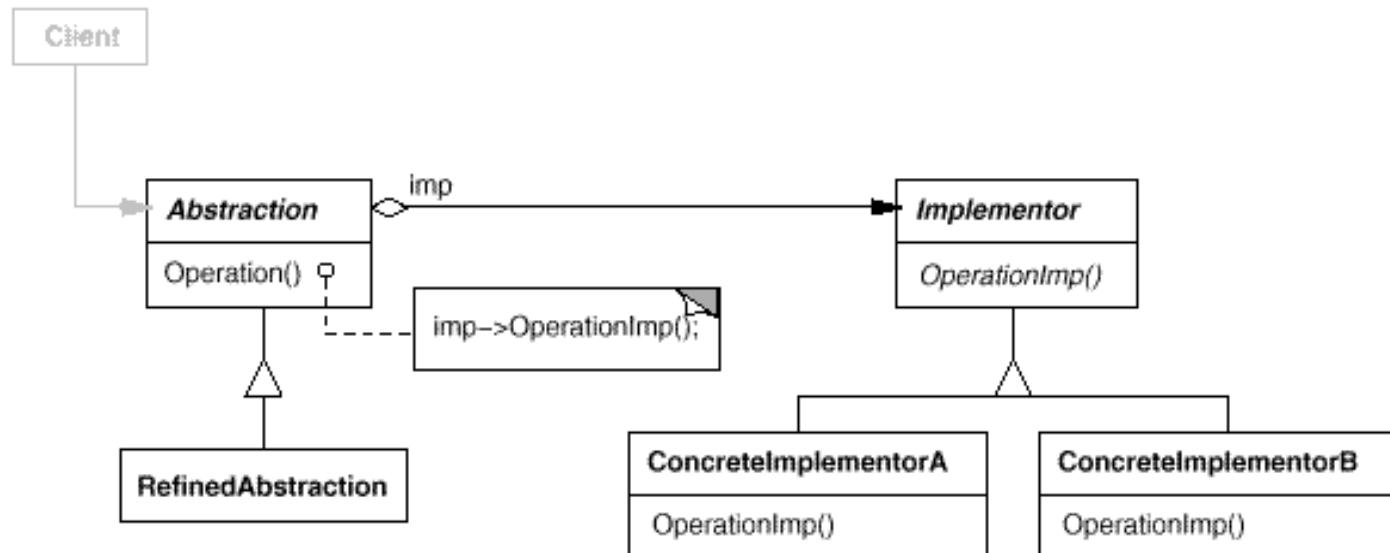
- Etend abstraction par héritage

Implementor :

- Définit l'interface de la classe d'implémentation.

Concretelimplementor :

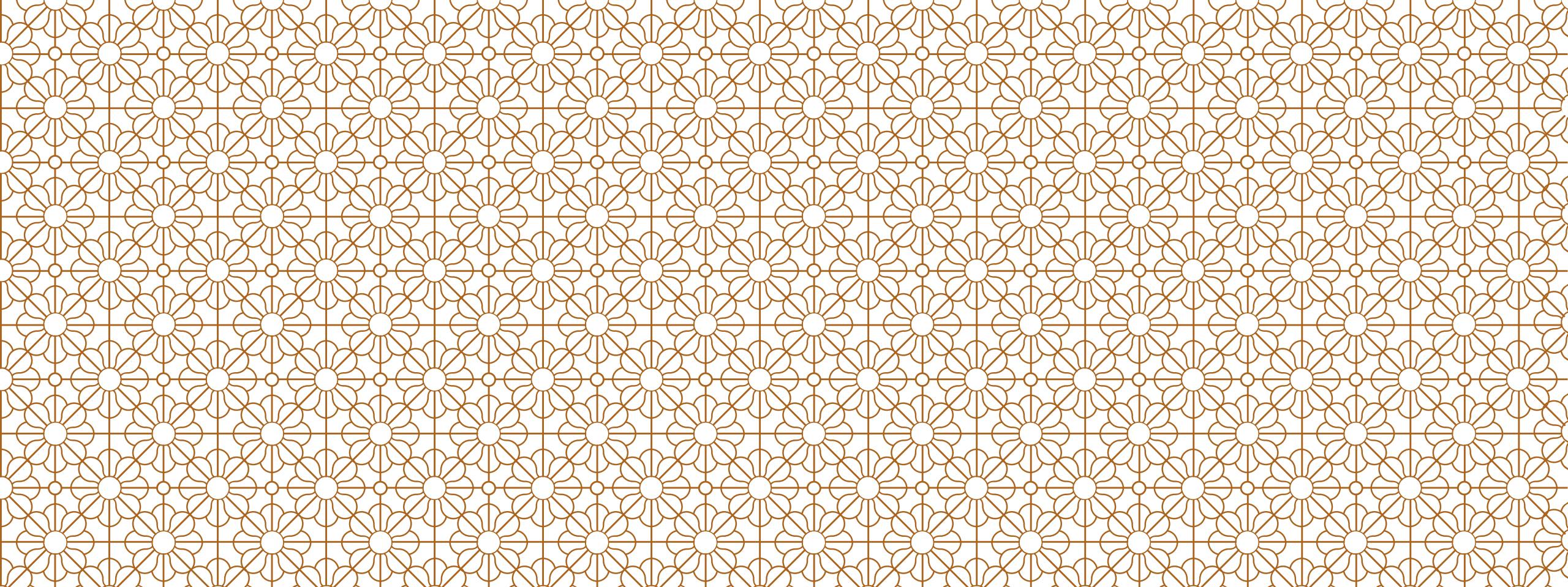
- Implémente l'interface implémentation



PONT

Conséquence d'utilisation :

- Découplage de l'interface et de l'implémentation. Une implémentation n'est pas liée de façon permanente à une interface. L'implémentation d'une abstraction peut être composée à l'exécution. Il est même possible qu'un objet change d'implémentation lors de l'exécution.
- Capacité d'extension accrue. Les hiérarchies d'Abstraction et d'implémentation peuvent être étendues indépendamment l'une de l'autre.
- Dissimulation des détails d'implémentation aux clients. On peut masquer au client les détails d'implémentations.



FACADE

FACADE

EXEMPLE: PARALLÉLISASSIONS DES DÉVELOPPEMENTS

Nous devons réaliser une architecture complexe pour la réalisation d'un logiciel. Pour paralléliser le développement, nous voulons être capable d'implémenter les tests de recette avant même que l'architecture soit réaliser. Nous voulons aussi être capable d'implémenter plusieurs sous-parties de l'architecture globale de manière indépendante.

FACADE

Intention :

Fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.

FACADE

Indication d'utilisation :

- On souhaite disposer d'une interface simple pour un sous-système complexe.
- Il y a beaucoup de dépendance entre les clients et les classes d'implémentations d'une abstraction.
- On cherche à structurer en niveau un sous-système. On utilise la façade pour définir un point d'entrée à chaque niveau du sous-système.

FACADE

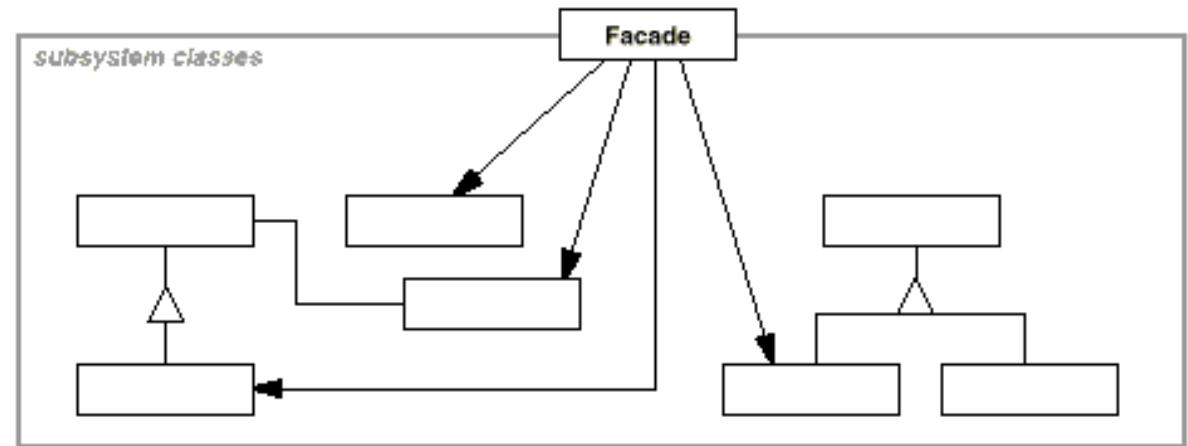
Constituant

Facade:

- Connait les classes du sous-système qui permettent d'effectuer une requête.
- Délègue les requêtes d'un client aux objets du systèmes.

Subsystem:

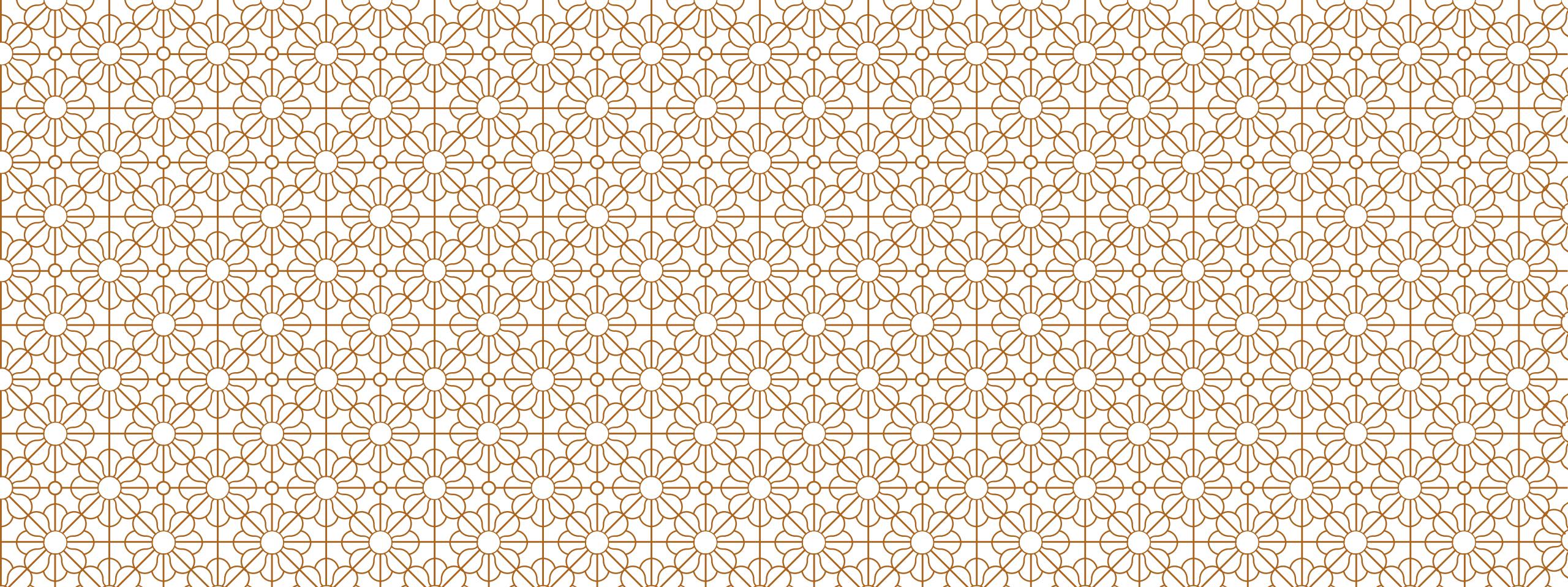
- Implémente les classes du sous-systèmes
- Effectue les opération ordonnées par la façade
- Ne connaît pas la façade



FACADE

Conséquence d'utilisation :

- Masque au client les composants du sous-système
- Favorise le couplage faible entre le sous-système et ses clients.
- Il n'empêche pas les applications d'utiliser les classes du sous-système si nécessaire.



OBSERVATEUR

OBSERVATEUR

EXEMPLE: MISE À JOURS DE L'AFFICHAGE ET BESOIN DE SAUVEGARDE

Nous disposons d'un composite de formes géométrique et nous voulons mettre à jours l'affichage graphique de nos formes quand elles sont modifiée. De plus, nous sommes capable de sauvegarder dans un fichier notre composite et nous voulons savoir à tout moment si il est nécessaire de le sauvegarder ou pas.

OBSERVATEUR

Intention :

Définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent soient notifiés et automatiquement mis à jour.

OBSERVATEUR

Indication d'utilisation :

- Quand un concept a deux représentations, l'une dépendant de l'autre.
- Quand la modification d'un objet nécessite de modifier les autres.
- Quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèse sur la nature de ces objets.

OBSERVATEUR

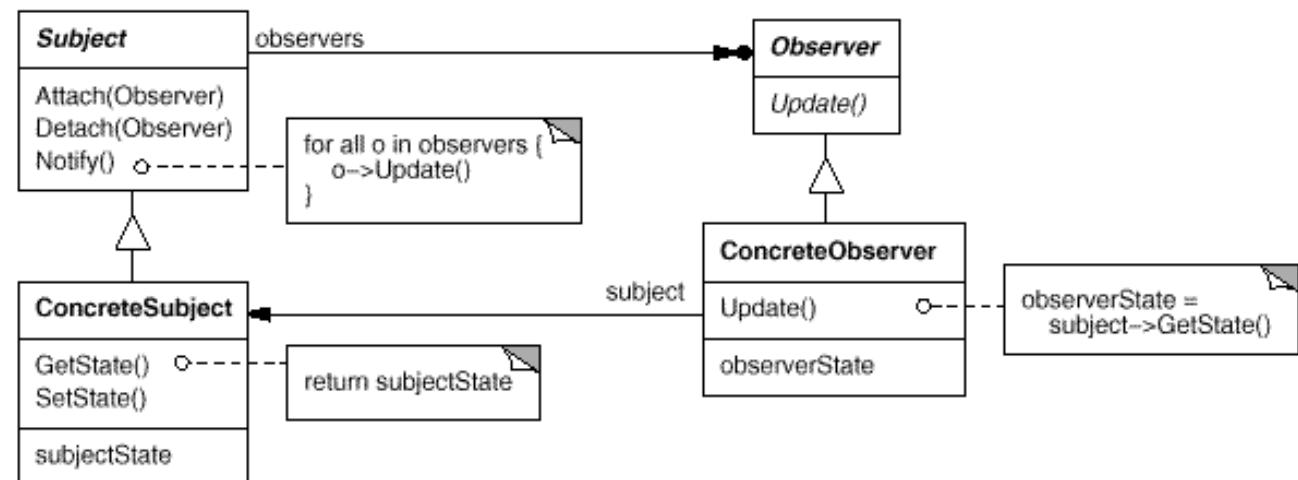
Constituant

Subject: Connait ses observateurs, il peut y avoir un nombre quelconque d'observateurs. Fournit une interface pour attacher/détacher des observateurs.

Observer : Fournit une interface pour les objet qui doivent être notifié quand il y a une modification sur le sujet.

ConcreteSubject: Stocke l'état qui intéresse les observateurs concrets. Notifie les observateur quand cet état change.

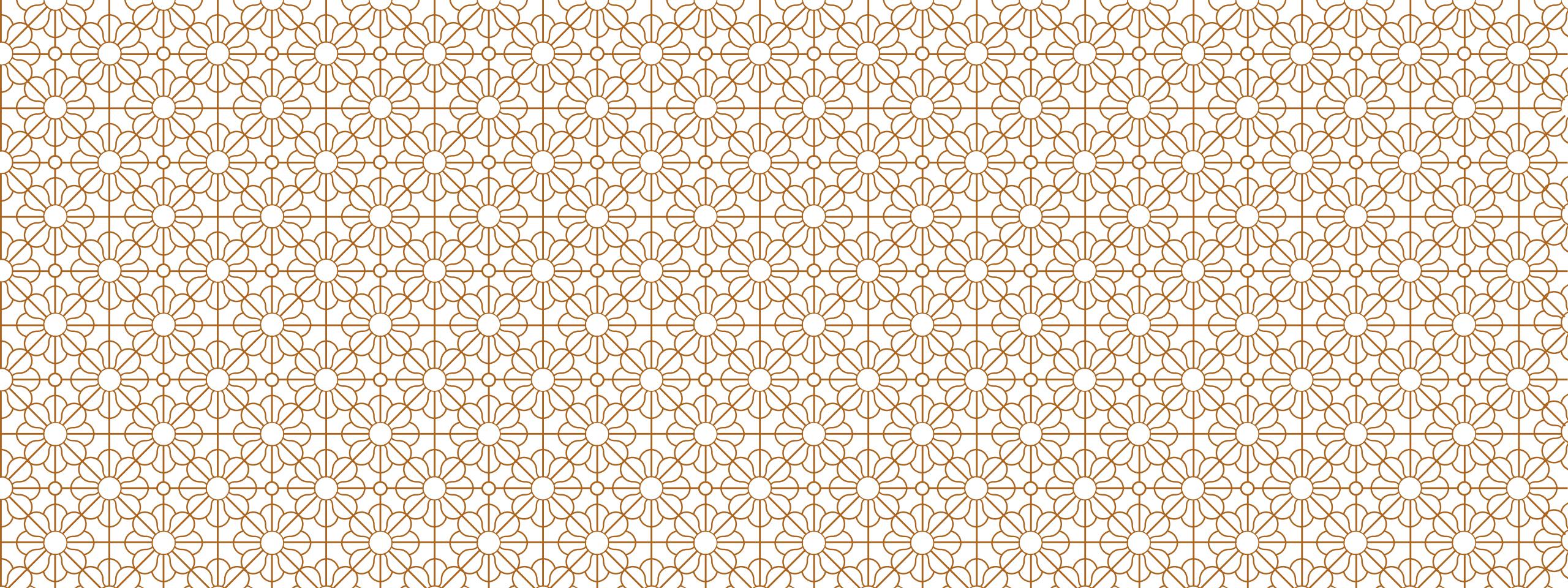
ConcreteObserver: Conserve une référence vers le sujet concret. Stocke l'état qui doit être synchroniser/mis à jours en fonction de l'état du sujet. Implémente l'interface de mise à jours.



OBSERVATEUR

Conséquence d'utilisation :

- Isoler le couplage entre Sujet et Observateur
- Support de la diffusion.
- Mises à jour inopinées



ITÉRATEURS

ITERATEUR

Intention :

Fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans dévoiler la représentation interne de celui-ci.

ITÉRATEUR

Indication d'utilisation :

- Accéder au contenu d'un agrégat sans dévoiler son implémentation
- Gérer simultanément plusieurs parcours dans des agrégats.
- Permettre une itération polymorphe.

ITERATEUR

Constituant

Aggregate:

- Définit une interface pour accéder à un itérateur concrète.

ConcreteAggregate:

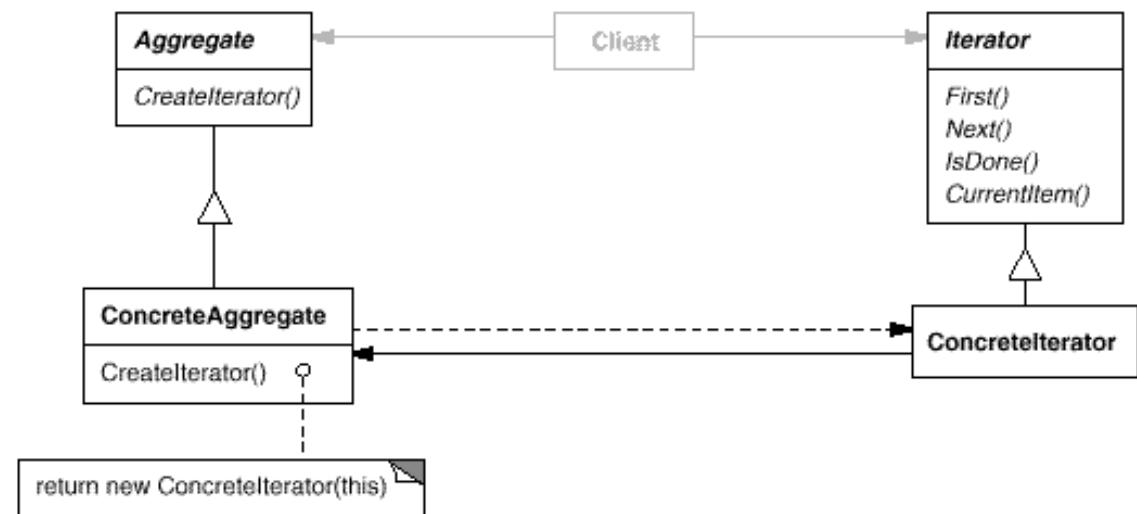
- Implémente la méthode de création de l'itérateur concrète.

Iterator:

- Définit une interface pour parcourir les éléments de l'aggrégat

ConcretIterator:

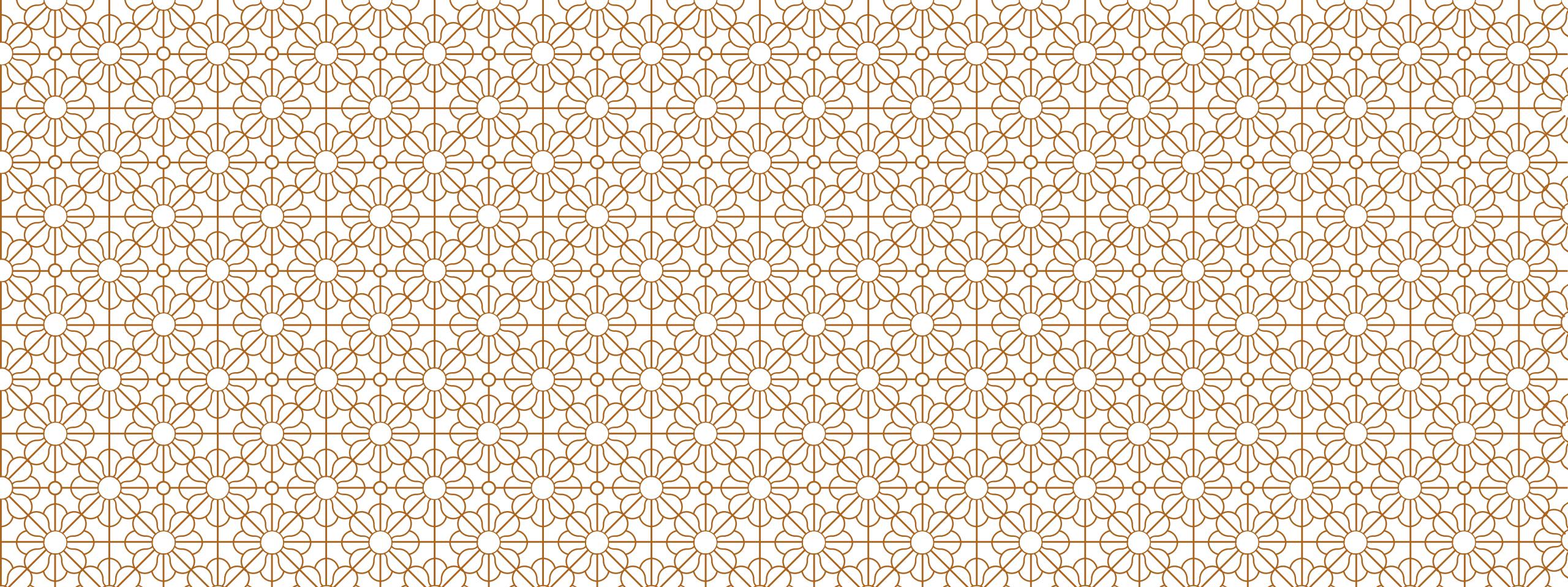
- Implémente l'interface itérateur



ITERATEUR

Conséquence d'utilisation :

- Il permet des modifications dans le parcours des agrégats
- Les itérateurs simplifient l'interface des agrégats.
- Il peut y avoirs plusieurs parcours simultanément.



VISITEUR

VISITEUR

Intention :

Le visiteur fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

VISITEUR

Indication d'utilisation :

- Une structure d'objet contient beaucoup de classes différentes d'interface distinctes et l'on veut réaliser des opérations sur ces objets qui dépendent de leurs types réels « dynamique ».
- Il s'agit d'effectuer plusieurs opérations distinctes et sans relation entre elles, sur les objets d'une structure, et ceci sans polluer leurs classes avec ces opérations
- Les classes qui définissent la structure objet changent rarement, mais on doit souvent définir de nouvelles opérations sur cette structure.

VISITEUR

Constituant

Visiteur:

- Déclare une opération visit pour chaque classe d'élément concrets dans la structure d'objet.

ConcreteVisitor:

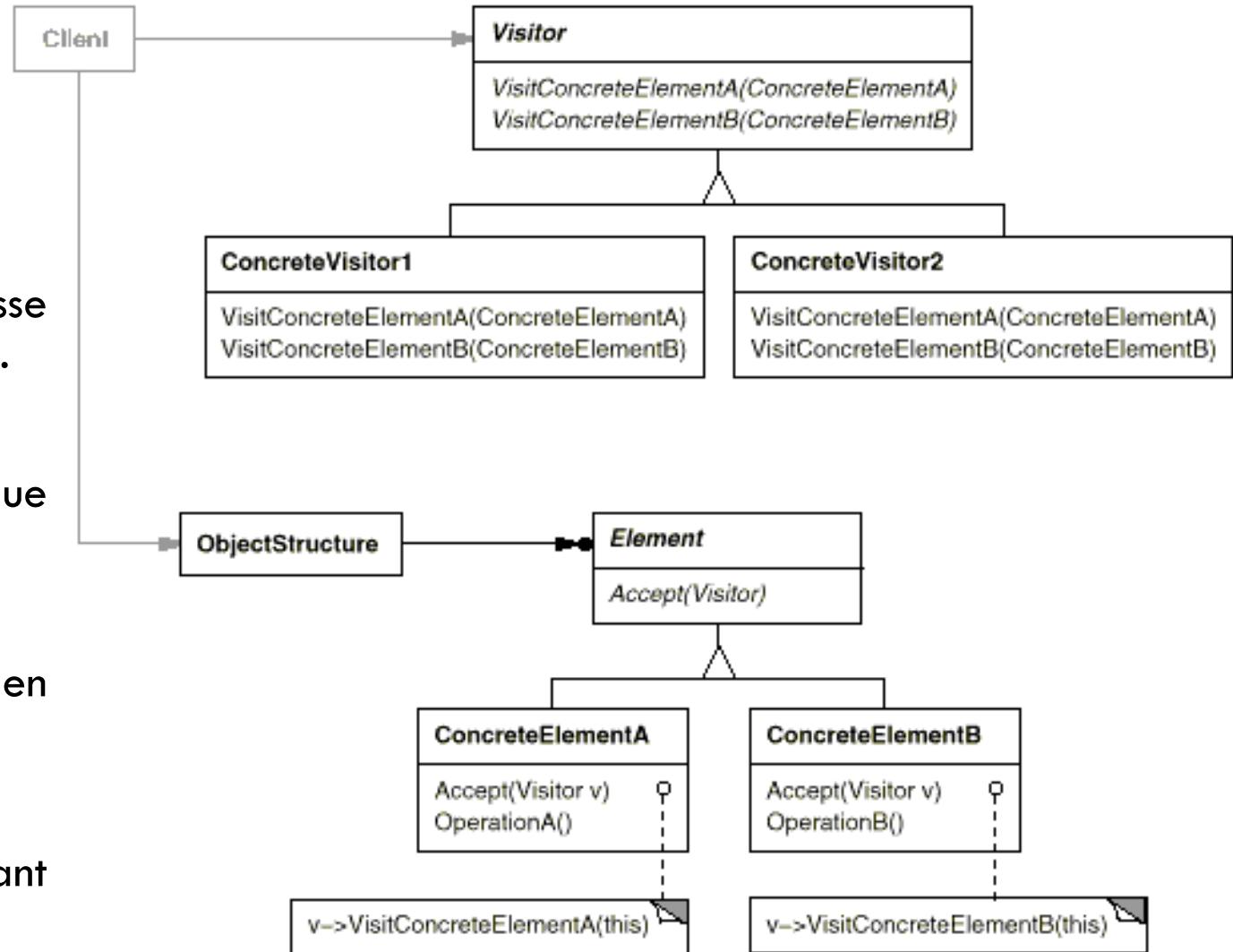
- Implémente les méthodes visit pour chaque classe d'élément concret.

Element:

- Définit une méthode accepte qui prend en paramètre un visitor.

ConcreteElement:

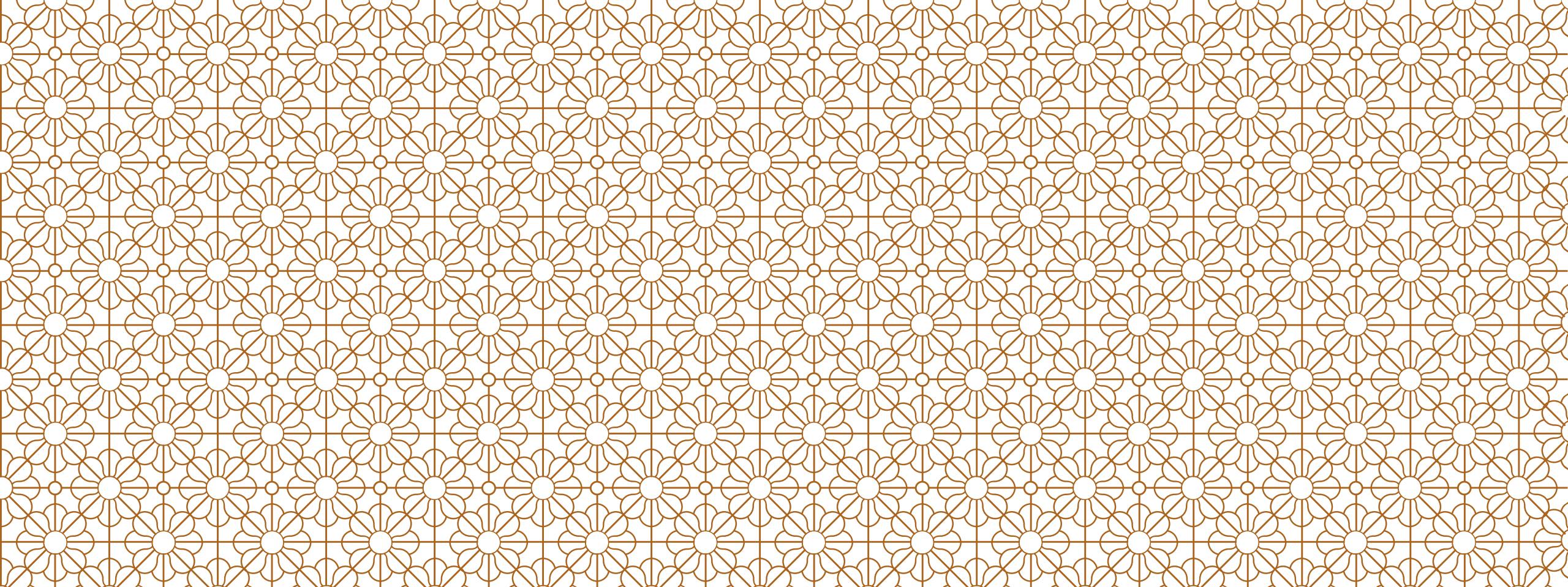
- Implémente la méthode accept en se passant comme paramètre de la méthode visit.



VISITEUR

Conséquence d'utilisation :

- Le visiteur facilite l'addition de nouvelles opérations
- Le visiteur rassemble des opérations de même type et isole celle non apparentées.
- L'addition de nouvelle classe élément concret est difficile.



MEMENTO

MEMENTO

Intention :

Sans violation de l'encapsulation saisir et transmettre l'état interne d'un objet, dans le but de pouvoir le restaurer ultérieurement.

MEMENTO

Indication d'utilisation :

- Un instantané de tout ou partie d'un objet doit être sauvegardé
- L'utilisation d'une interface directe pour atteindre l'état conduirait à révéler des détails d'implémentation.

MÉMENTO

Constituant

Originator:

- Construit un memento contenant un instantané de son état interne.
- Utilise un memento pour restaurer son état interne.

Memento:

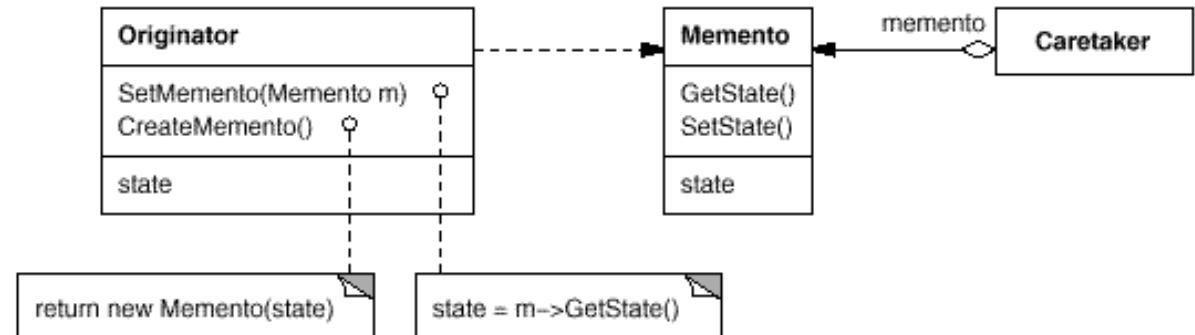
- Stocke l'état interne de l'objet original.

Element:

- Définit une méthode accepte qui prend en paramètre un visitor.

Caretaker:

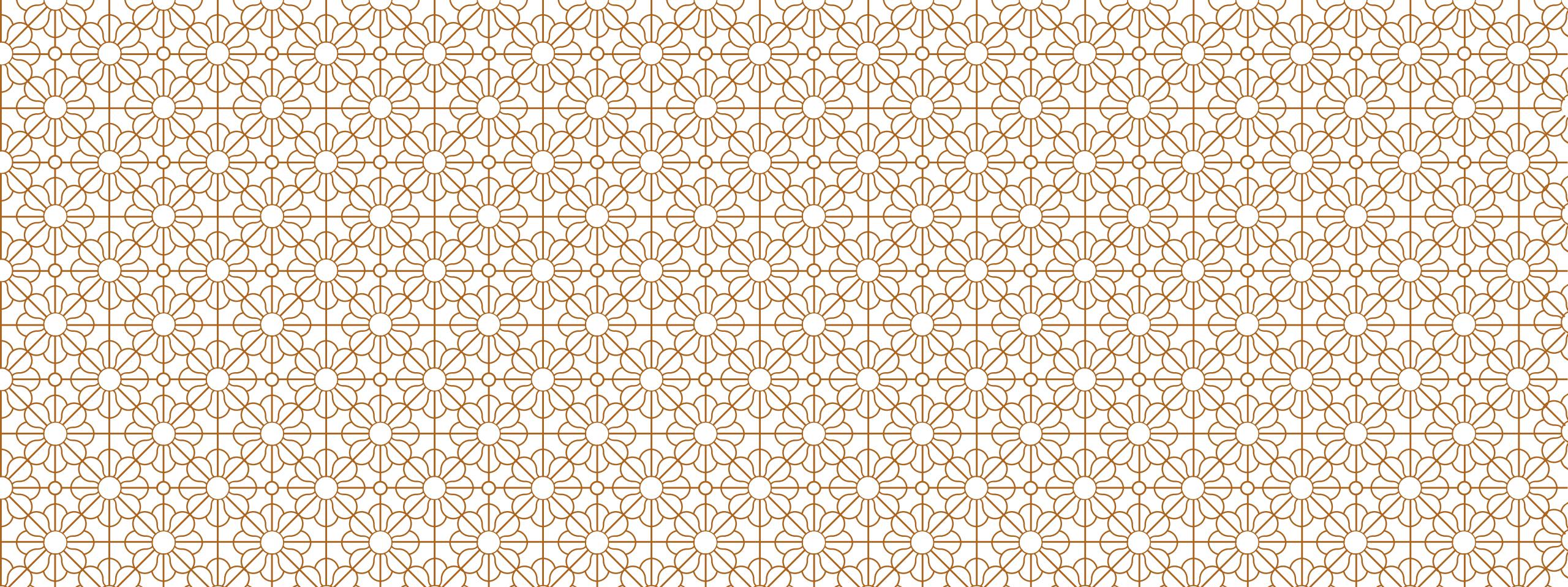
- L'utilisateur des memento les stocke pour les restaurer ultérieurement mais ne peut pas accéder à leur contenu.



MÉMENTO

Conséquence d'utilisation :

- Préservation des frontières d'encapsulation
- Il simplifie l'objet (sur lequel s'applique le memento)
- Peut être couteuse et du coup non approprié



COMMAND

COMMAND

Intention :

Encapsuler une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attentes et récapitulatifs des requêtes. De plus, permet la réversion des opérations.

COMMAND

Indication d'utilisation

Introduire dans des objets sous la forme de paramètre des actions à effectuer.

Spécifier mettre en file d'attente et exécuter les requêtes à différent instants.

Assurer le service « défaire »(UNDO).

Permettre une mémorisation de modification

Structurer un système autour d'opération de haut niveau.

COMMAND

Constituant

Command:

- Déclare une interface pour exécuter une opération.

ConcretCommand:

- Définit le lien entre l'objet sur lequel s'applique l'opération.

Client:

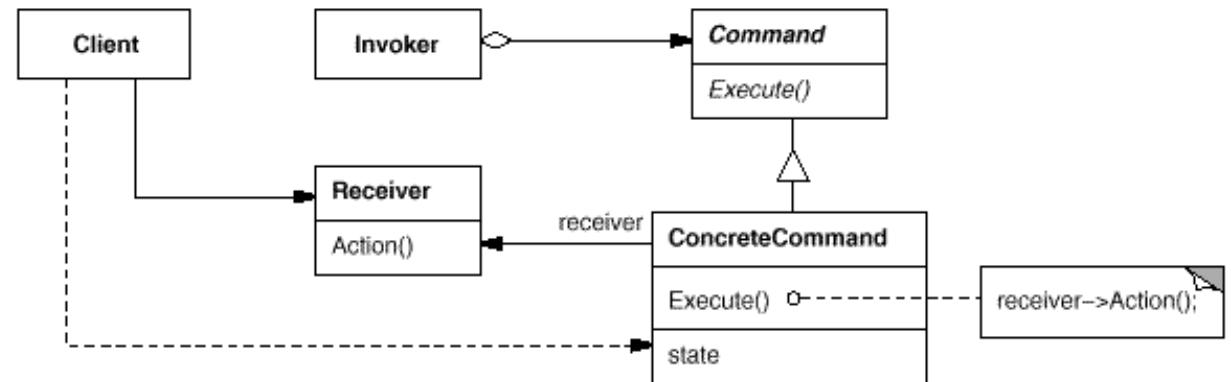
- Instancie la commande concrète et la lie à son récepteur.

Invoker:

- Demande à la commande de s'exécuter.

Receiver:

Objet sur lequel s'applique l'opération



COMMAND

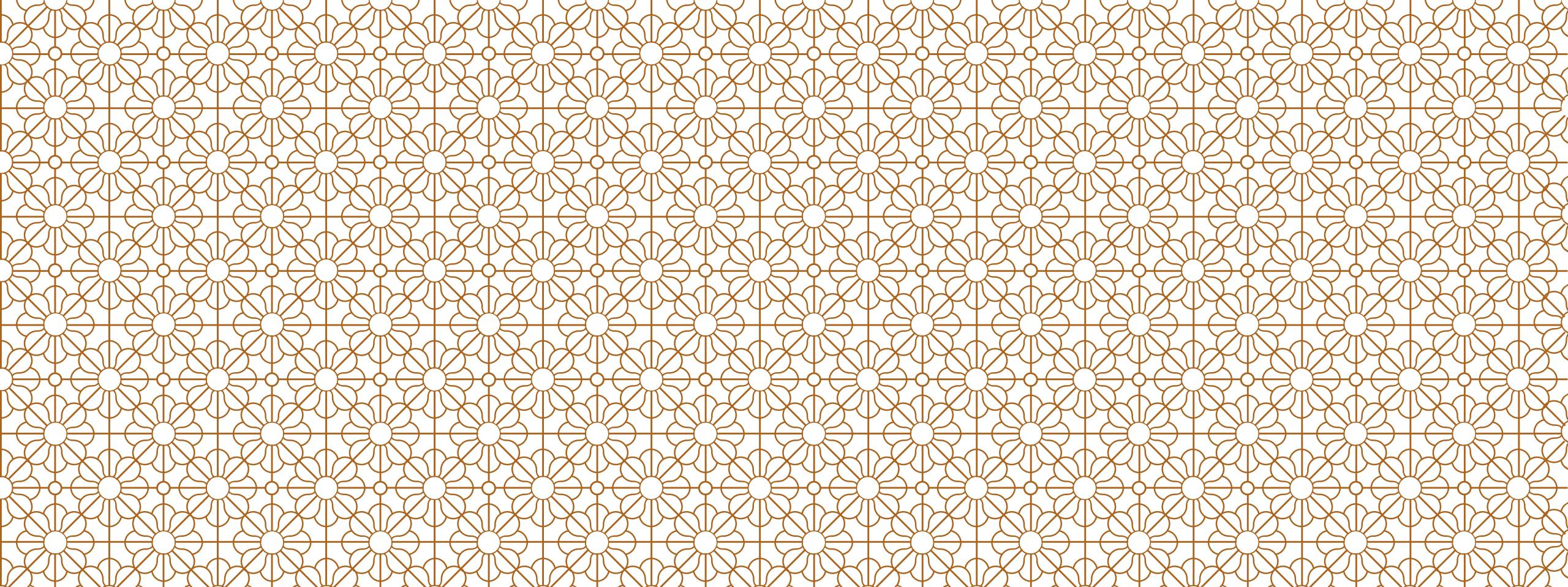
Conséquence

Commande supprime tout couplage entre l'objet qui invoque une opération et celui qui sait comment réaliser cette opération.

Les commandes sont des objets à part entière. Ils peuvent être manipuler et étendus comme n'importe quel autre objet.

On peut assembler des commandes dans une commande composite.

Il est facile d'ajouter des nouveaux objets commande car cela n'implique pas de modifier le code existant.



CHAIN OF RESPONSABILITY

CHAIN OF RESPONSABILITY

Intention

Eviter le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet d'entreprendre la requête. Chainer les objets récepteurs et faire passer la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite.

CHAIN OF RESPONSABILITY

Indication d'utilisation

Une requête peut être gérée par plus d'un objet à la fois et le gestionnaire n'est pas connu à priori. Ce dernier doit être déterminé automatiquement.

On souhaite adresser une requête à un ou plusieurs objets sans spécifier explicitement le récepteur

L'ensemble des objets qui peuvent traiter une requête doit être défini dynamiquement.

CHAIN OF RESPONSABILITY

Constituant

Handler:

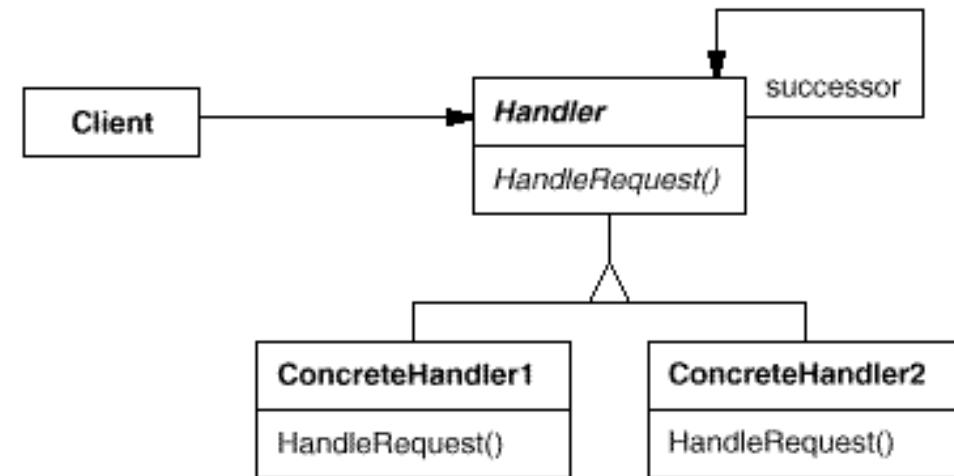
- Définit une interface pour gérer une requête.

ConcreteHandler:

- Implémente la requête dont il est responsable.
- Il peut accéder à son successeur
- Si il ne peut pas effectuer l'opération, il la transmet à son successeur.

Client:

- Initialise la requête sur un handler de la chaîne.



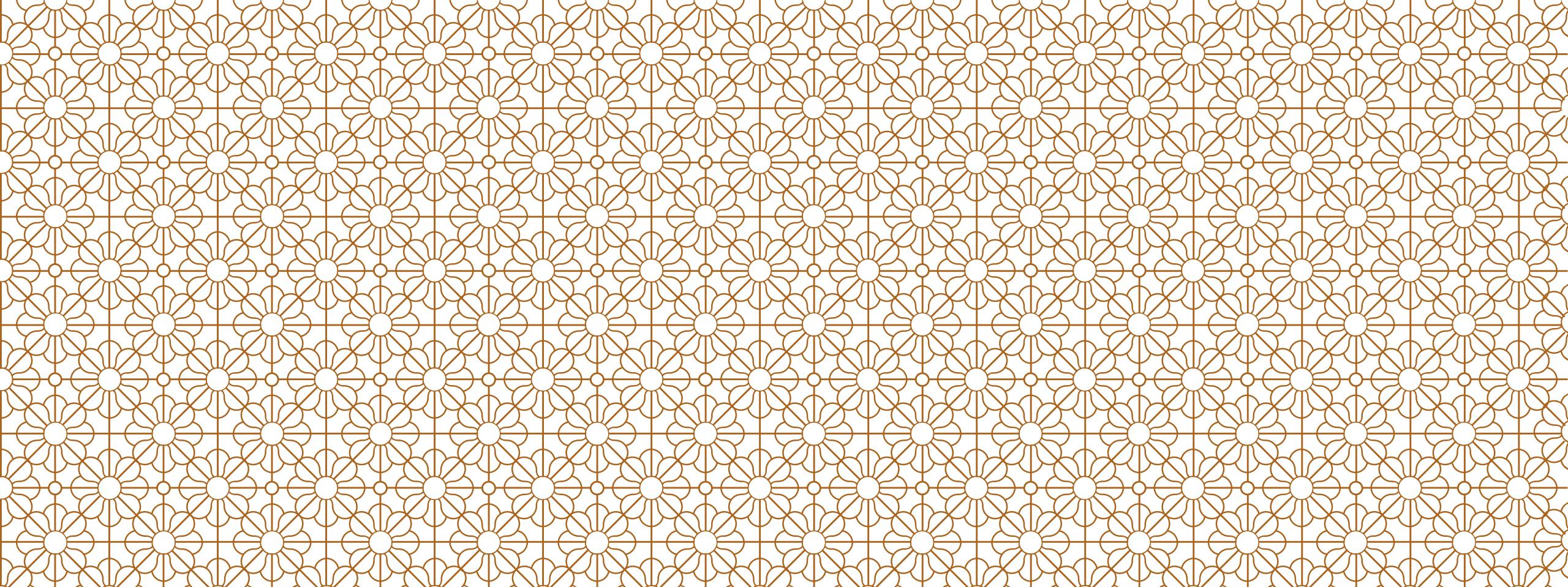
CHAIN OF RESPONSABILITY

Conséquence

Réduction du couplage,

Souplesse accrue dans l'attribution de responsabilités aux objets.

La réponse n'est pas garantie.



MEDIATOR



MEDIATOR

Intention :

Définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets. Les médiateurs favorise le couplage faible en dispensant les objets de se faire explicitement référence. Il permet de faire varier indépendamment les relations d'interaction.

MEDIATOR

Indication d'utilisation

Les objets d'un ensemble communiquent d'une façon bien définie mais très complexe. Le résultat des interdépendances est non structuré et difficile à appréhender.

La réutilisation d'un objet est difficile du fait qu'il fait référence à beaucoup d'autres objets et communique avec eux.

Un comportement distribué entre plusieurs classes doit pouvoir être spécialisé sans une pléthore de dérivation.

MEDIATOR

Constituant

Mediator:

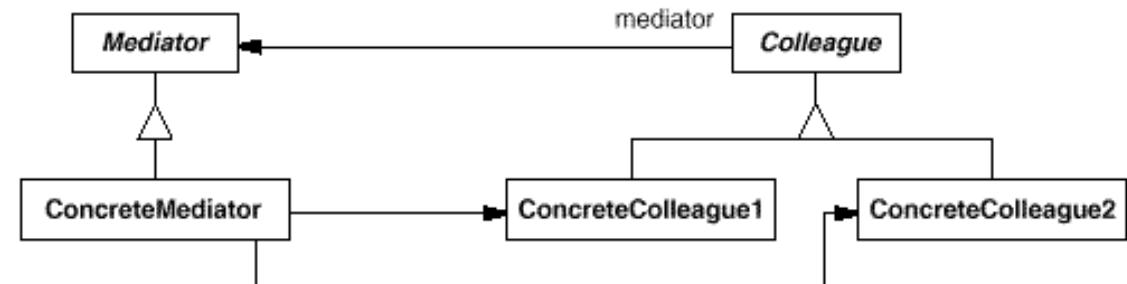
- Définit une interface pour communiquer avec les objets « collègues ».

ConcreteMediator:

- Implémente le comportement coopératif entre les collègues.
- Connait les collègues.

Colleague:

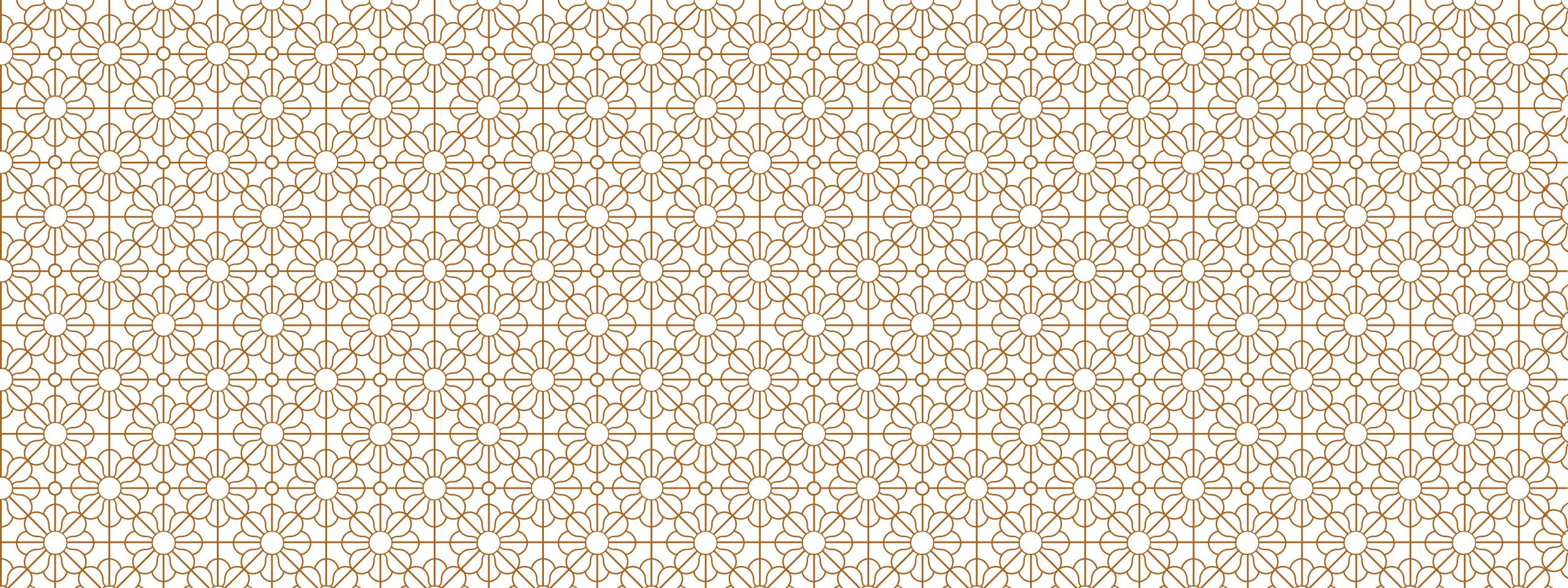
- Chaque collègue connaît le médiator.
- Chaque collègue communique avec le mediator au lieu de communiquer directement avec un autre collègue.



MEDIATOR

Conséquence

- Il limite la création de sous classes
- Il réduit le couplage entre les collègues
- Il simplifie les protocoles objet
- Il formalise les coopérations d'objet.
- Il centralise le contrôle



STATE

STATE

Intention :

Permet à un objet de modifier son comportement quand son état interne change. Tout se passe comme si l'objet changeait de classe.

STATE

Indication d'utilisation :

- Le comportement d'un objet dépend de son état et ce changement de comportement doit intervenir dynamiquement en fonction de cet état.
- Les opérations comportent de grands pans de déclarations conditionnelles fonctions de l'état d'un objet.

STATE

Constituant

Context:

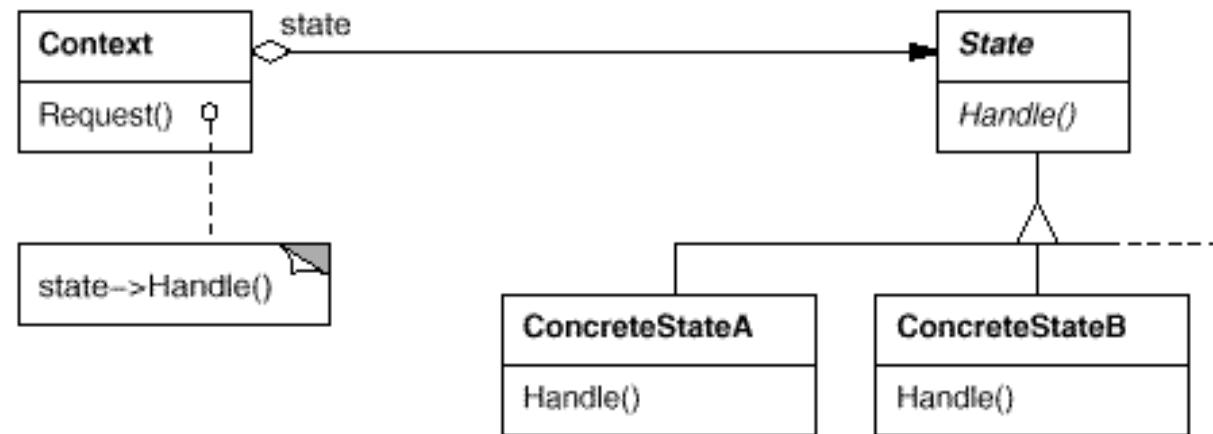
- Définit une interface d'intérêt pour le client.
- Maintient une instance d'état concrète représentant l'état courant.

State :

- Définit l'interface pour abstraire le comportement des différents état possibles

ConcretState:

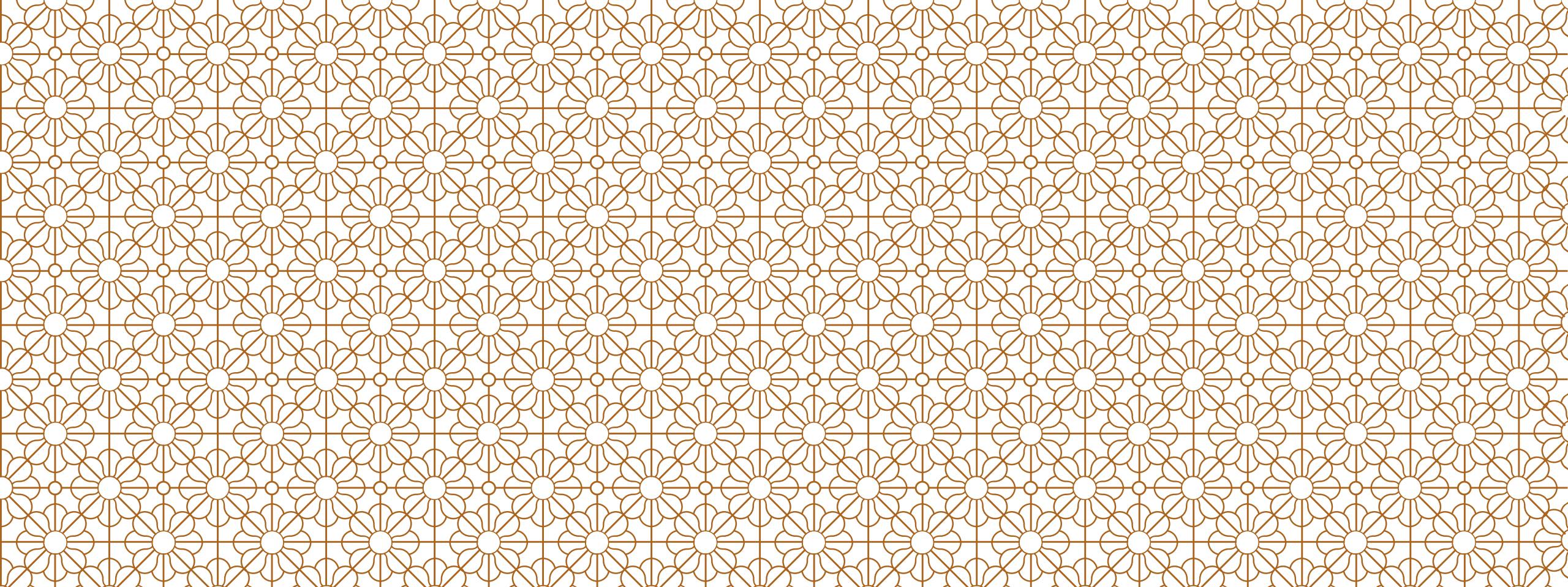
- Chaque sous classe d'état concrète implémente l'état le comportement de context dans un état donné.



STATE

Conséquence

- Il isole les comportements spécifiques d'états et fait un partitionnement des différents comportements état par état.
- Il rend les transitions d'état plus explicites.
- Les objets état peuvent être partagés.



STRATEGY

STRATEGY

Intention :

Définit une famille d'algorithme, encapsule chacun d'entre eux et les rend interchangeables. Le modèle stratégie permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent.

STRATEGY

Indication d'utilisation :

- Plusieurs classes apparentées ne diffèrent que par leur comportement ; Les stratégies donnent le moyen d'appareiller une classe avec un comportement parmi plusieurs autres.
- On a besoin de plusieurs variantes d'un algorithme.
- Un algorithme utilise des données que les clients n'ont pas à connaître.
- Une classe définit de nombreux comportements qui figurent dans ses opérations sous la forme de déclarations conditionnelle multiple.

STATE

Constituant

Context:

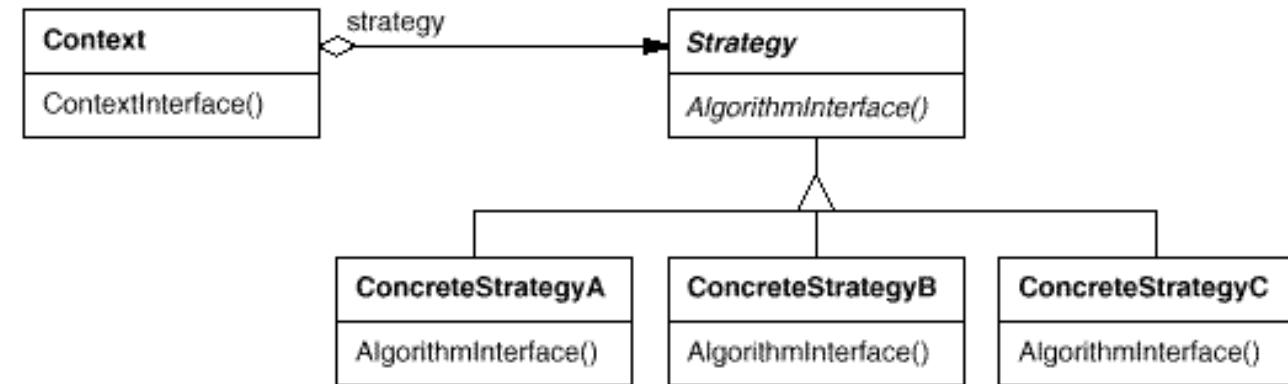
- Est configuré avec un objet strategy.

Strategy :

- Définit une interface commune à tous les algorithmes utilisés par context.

ConcretStrategy:

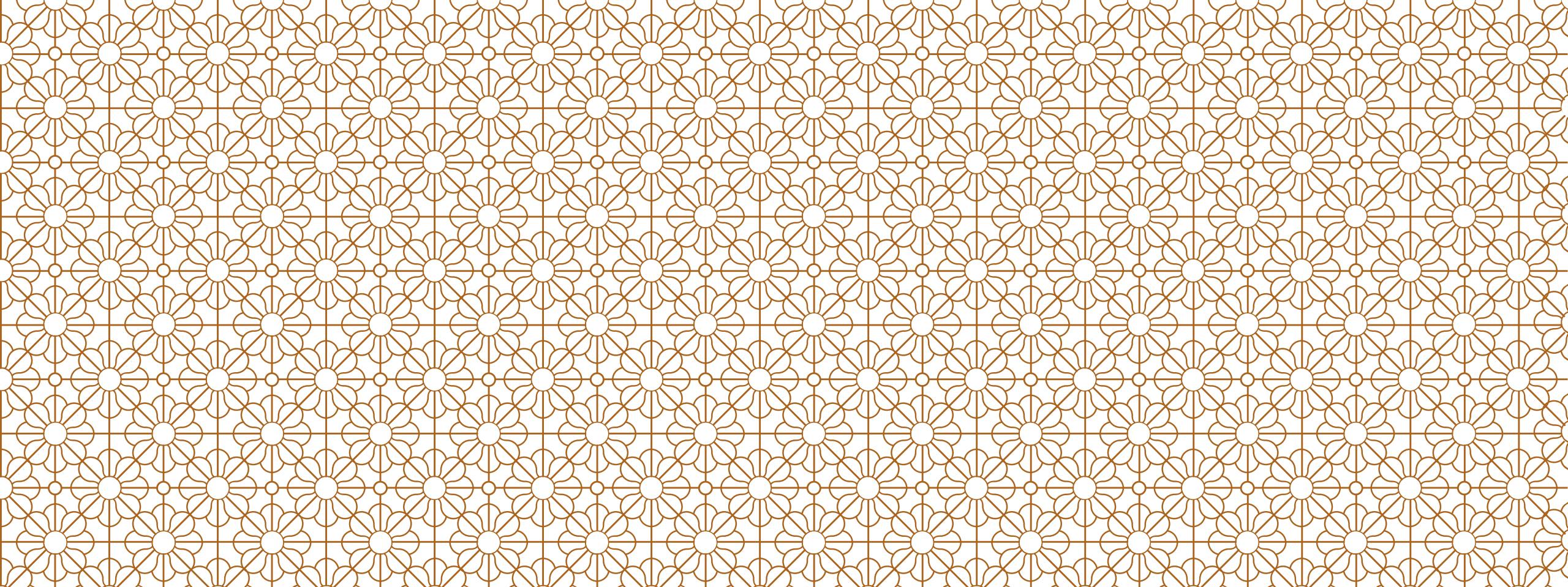
- Implémente un algorithme en implémentant l'interface strategy.



STRATEGY

Conséquence

- Famille d'algorithmes apparentés
- Solution alternative à la dérivation en sous classes
- Les stratégies dispensent de l'utilisation de déclarations conditionnelles.



FLYWEIGHT

FLYWEIGHT

Intention :

Le modèle Poids Mouche utilise une technique de partage qui permet la mise en œuvre efficace d'un grand nombre d'objets de fine granularité.

FLYWEIGHT

Indication d'utilisation :

- L'application utilise un grand nombre d'objets
- Les coûts de stockage sont élevés du fait d'une réelle quantité d'objets
- La plupart des états de l'objet peuvent être considérés comme extrinsèques
- Plusieurs groupes d'objets peuvent être remplacés par un nombre relativement faible d'objet partagés.
- L'application ne dépend pas de l'identité des objets. Du fait que les objets poids mouche peuvent être partagés, des objets de conceptions distinctes peuvent passer pour identique lors des tests comparatifs.

FLYWEIGHT

Constituant

Flyweight:

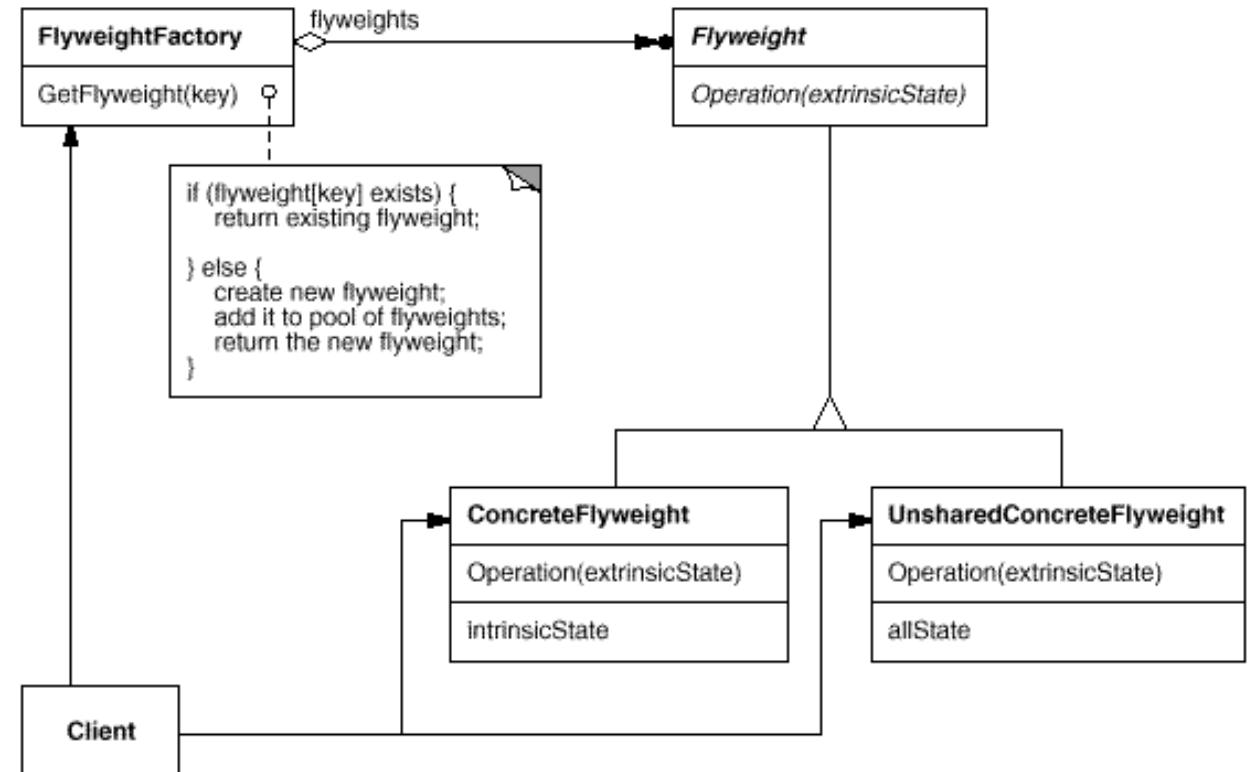
Déclare l'interface des opérations applicables sur un l'état extrinsèque.

Concrete flyweight :

- Implémente l'opération sur l'état extrinsèque et stocke l'état intrinsèque de l'objet.

Flyweight Factory:

- Permet de récupérer une instance de flyweight.
- Garantie le partage des objets en s'assurant de retourner la même instance pour une clé identique.



FLYWEIGHT

Conséquence

- L'utilisation du poids mouche peut introduire un coût à l'exécution résultant du transfert, de la recherche ou du calcul des états extrinsèques.

- L'économie est fonction de plusieurs facteurs :

Diminution du nombre total d'instances qui résulte du partage

Nombre total d'états intrinsèques par objets

Le fait que l'état extrinsèque soit stocké ou calculé.