Cours 6 - C++ pour les mathématiques appliquées

Conception Objet

La dernière fois . . .

Analyse

- Diagramme des cas d'utilisations
- Diagramme de séquence
- Diagramme de transitions
- Diagramme de classes d'analyse

Analyse et conception

Vue globale

> - Analyse : spécification (quoi)
> - Conception architecturale (structure globale)
> - Conception détaillée (comment)

Objectif de la conception objet détaillée

> Décrire précisément
>
> - classes logicielles qui composent le système
> - interactions entre les objets
> - comportement des objets

Affectation des responsabilités aux objets

Connaissances

- données encapsulées
- objets connexes
- données dérivées (pouvant être calculée)

Comportements

- action (calcul, création)
- délégation (action sur un autre objet)
- coordination des actions entre objets

Principes de conception

Propriétés

- Faible couplage entre classes
- Cohésion forte au sein d'une classe
- Principe de l'expert
- Principe du créateur
- Principe du contrôleur

Principe de l'expert

Qui RÉALISE une tâche T ? Au sein de quelle classe ?

> T est réalisée par un objet expert qui a accès à l’information nécessaire pour calculer T

Principe du créateur

Qui CRÉE des objets de la classe A ?

> une classe créateur B tel qu'il existe une relation entre A et B (par exemple, B contient des objets de A).

Principe du contrôleur

Où effectuer le TRAITEMENT des événements du système ?

> Dans une ou plusieurs classes de contrôle (des contrôleurs).

Utilisation des diagrammes UML (1)

Diagramme des classes logicielles

- inspiré du diagramme de classes conceptuelles de la modélisation objet
- contient des classes logicielles précisant les structures de données et les communications entre objets
- description complete des classes (attributs, méthodes)
- peut être instancié par des diagrammes d'objets types

Utilisation des diagrammes UML (2)

Diagrammes de séquence ou de collaboration : collaborations entre objets

Diagrammes d'états-transitions : comportement des objets

Diagrammes d'activité : comportement d'une opération

Cohérence entre les différents diagrammes

Vigilance

> - l'ensemble des diagrammes forme un tout !
> - correspond à très proche échéance à la structure du code

Exemples

> - les objets utilisés dans les diagrammes sont des instances de classes
> - méthodes et attributs nécessaires à la réalisation de scénario
> - relation entre objets

Patrons de conception

Principe des patrons de conception

La conception de logiciels est difficile!

$\longrightarrow$ conception objet (extensibilité, reutilisabilité) mais il faut des développeurs *expérimentés*

Objectifs des patrons de conception -- Design Patterns

- recueillir l'expérience et l'expertise
- transmettre pour une réutilisation

Historique

- concept introduit en 1977 par C. Alexander (architecte en bâtiment)
- patrons en conception objets introduits en 1987 par Beck et Cunningham
- thèse de Erich Gamma en 1991 (junit, eclipse jdt, maintenant à ibm)

Patrons de conception : bibliographie

- La programmation orientée objet. Hugues Bersini. 2013.
- Design Patterns. Eric Freeman, Elisabeth Freeman, Kathy sierra et Bert Bates. 2011.
- Object - Oriented Modeling And Design With Uml, Michael R. Blaha, James R Rumbaugh. 2007.
- Design Patterns. Elements of Reusable Object-Oriented Software. E. Gamma, R. Helm, R. Johnson, J. Vlissides. 1995.

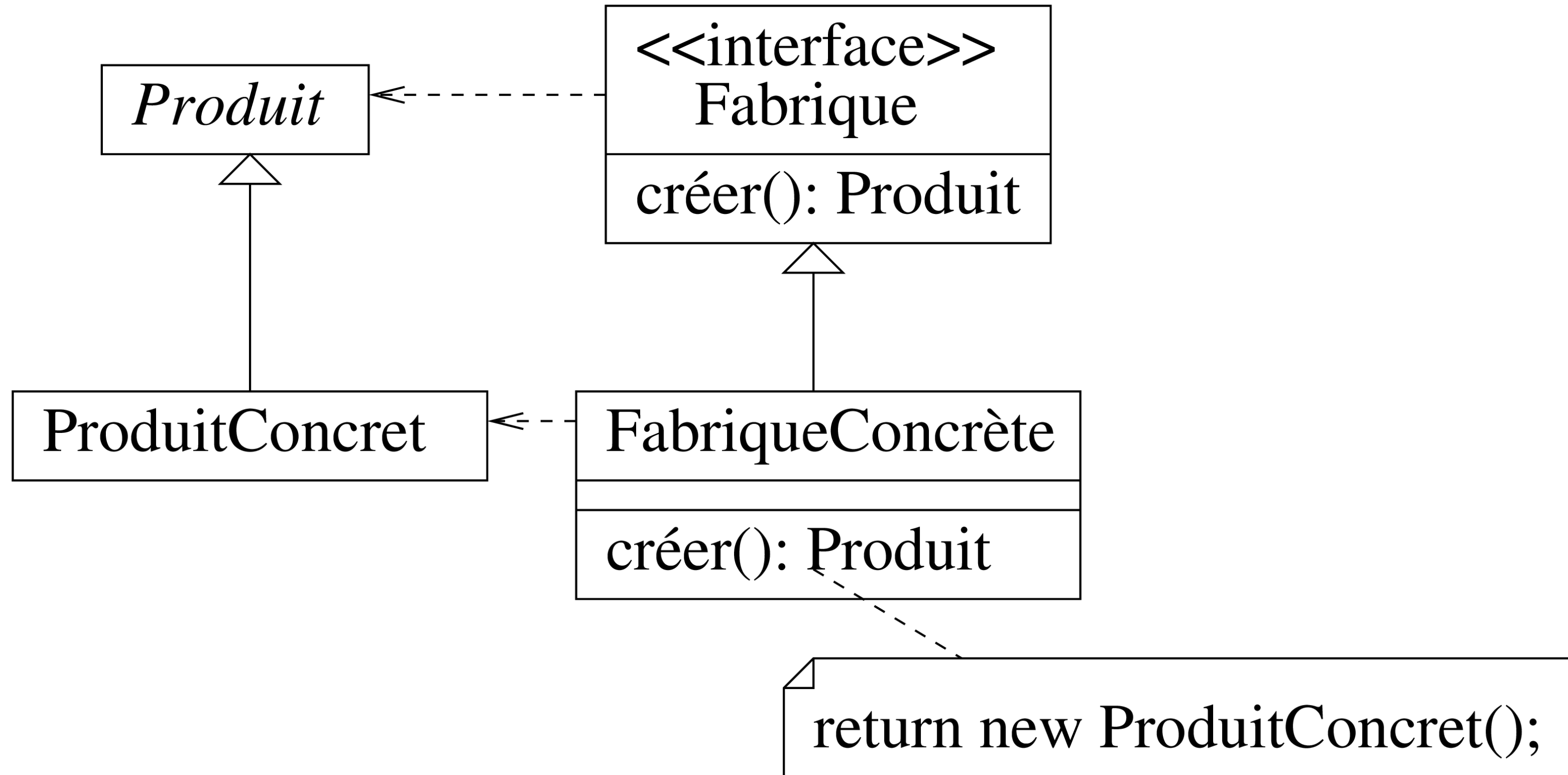
Patrons créationnistes

Singleton

Singleton
–final static instanceUnique : Singleton
instance(): Singleton

```
class S {
    S() { /*...*/ };
    S(S const& other) = delete;
    S(S&& other) = delete;
public:
    static S& GetInstance() {
    static S instance;
    return instance;
    }
};
```

Méthode Fabrique



Méthode Fabrique : idée

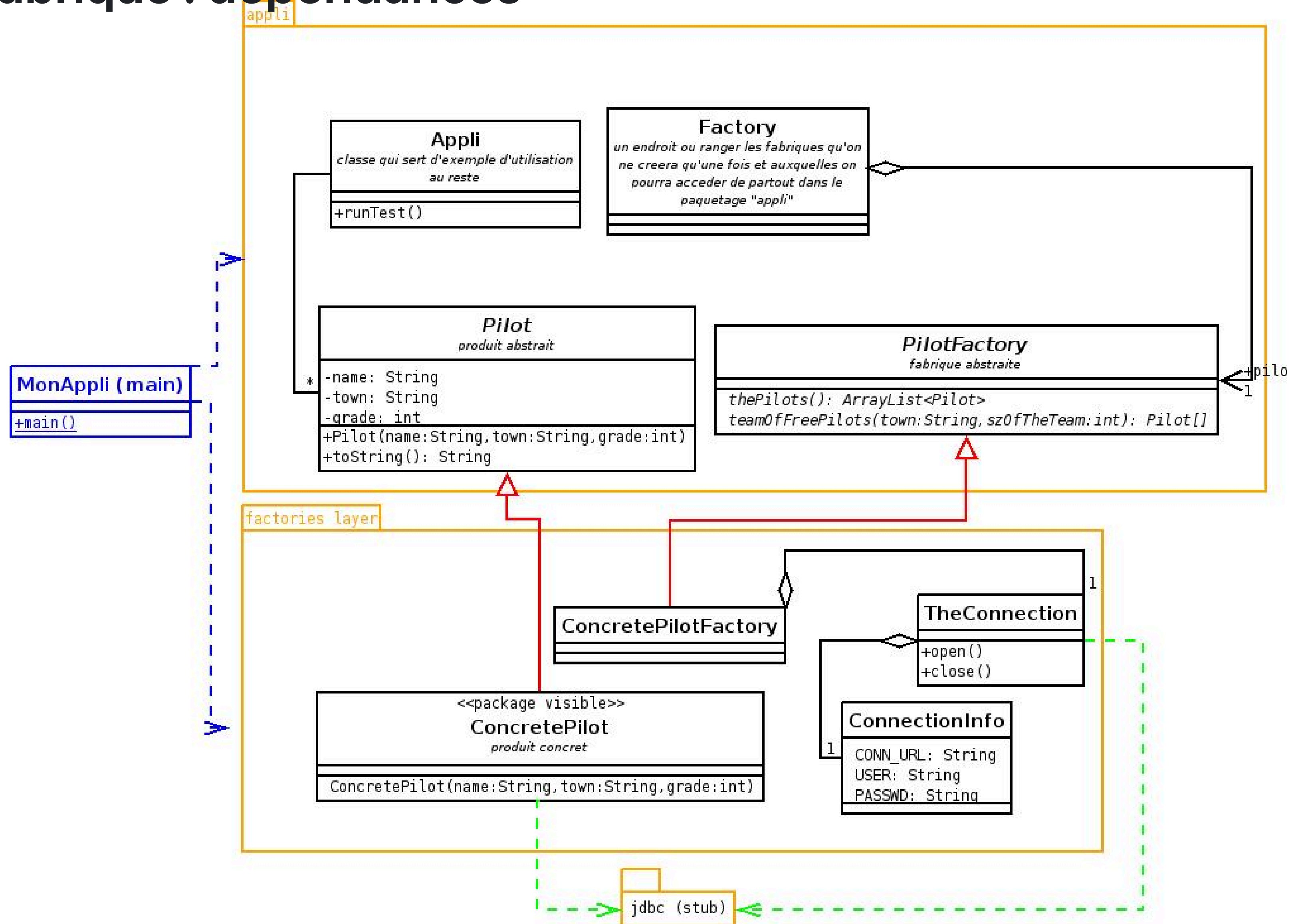
Objectif

Création d'objets sans en connaître la classe exacte

Principes

- des objets de classe à créer
- c'est l'interface qui en est chargée
- via la "méthode fabrique"\ *(pas d'appel direct au constructeur de)*
- les classes `ProduitConcret` et `FabriqueConcrète` contiennent le code relatif à la création

Méthode Fabrique : dépendances



Fabrique : exemple

```cpp
enum class PointType { cartesian, polar };
class Point {
    float         m_x, my;
    PointType     m_type;
    Point(const float x, const float y, PointType t) : m_x{x}, m_y{y}, m_type{t} {}
  public:
    friend ostream &operator<<(ostream &os, const Point &obj) {
        return os << "x: " << obj.m_x << " y: " << obj.m_y;
    }
    static Point NewCartesian(float x, float y) {
        return {x, y, PointType::cartesian};
    }
    static Point NewPolar(float a, float b) {
        return {a * cos(b), a * sin(b), PointType::polar};
    }
};
```

Fabrique Abstraite : principe

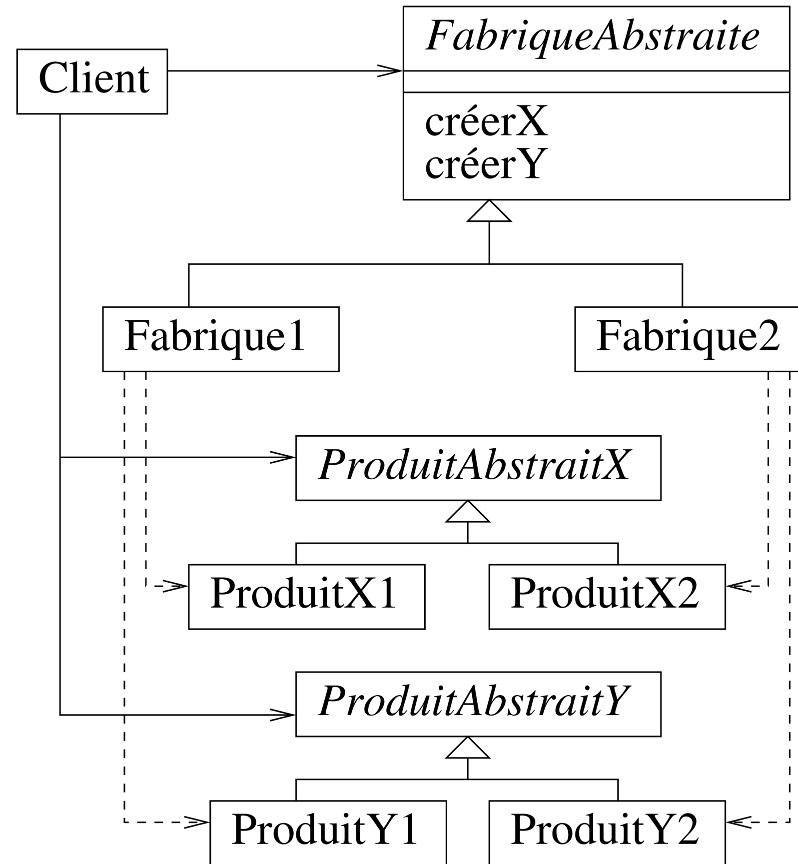
Objectif

- Création de *familles d'objets interdépendants*;
- Le client ne connaît pas les classes exactes;

Principe

- le client veut produire des et (interfaces)
- il existe plusieurs implémentations de et : ce sont des *familles* : `XImplem1` fonctionne uniquement avec `YImplem1` `...`
- le client reçoit une `Fabrique` de `X` et `Y` (interface) qu'il utilise pour produire des `X` et `Y` des d'une même famille
- la fabrique peut être `Fabrique1` par exemple

Fabrique Abstraite



Fabrique abstraite : exemple

```cpp
struct Point {
  virtual ~Point() = default;
  virtual unique_ptr<Point> create() = 0;
  virtual unique_ptr<Point> clone()  = 0; };
struct Point2D : Point {
  unique_ptr<Point> create() { return make_unique<Point2D>(); }
  unique_ptr<Point> clone() { return make_unique<Point2D>(*this); }
};
struct Point3D : Point {
  unique_ptr<Point> create() { return make_unique<Point3D>(); }
  unique_ptr<Point> clone() { return make_unique<Point3D>(*this); }
};
void who_am_i(Point *who) {
  auto new_who       = who->create();
  auto duplicate_who = who->clone();
  delete who;
}
```

Fabrique Abstraite : évaluation

Avantages

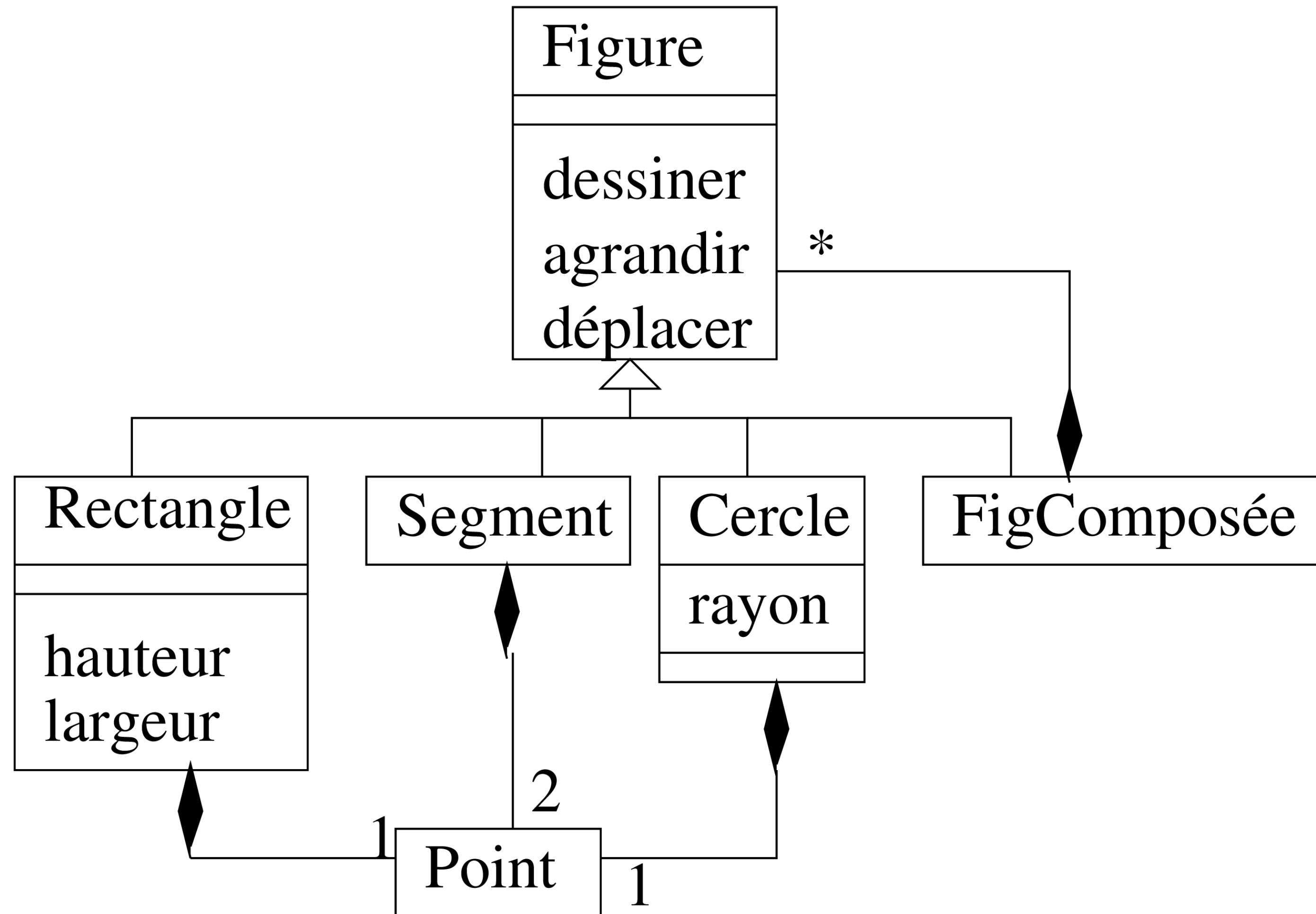
- changement de famille de produits
- ajout d'une famille de produits
- utilisation cohérente des produits

Inconvénients

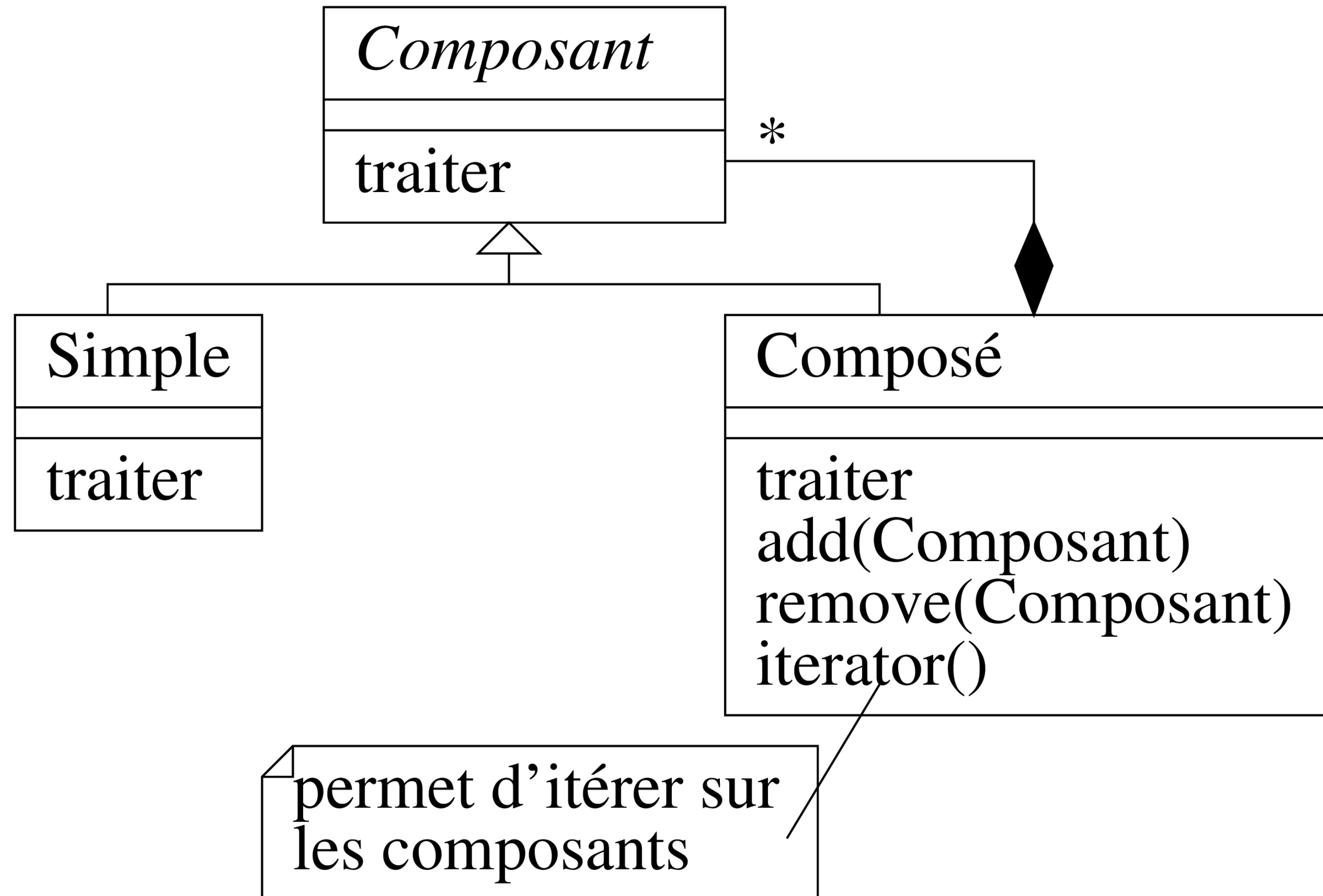
difficulté d'ajouter des produits :
$\Longrightarrow$ modifier toutes les fabriques existantes

Patrons structurels

Exemple de Composite



Objet Composite



Objet Composite : Principe

But

- créer des objets simples ou composés
- ayant des méthodes de traitement uniformes

Motivation

Manipuler un objet indépendamment du fait qu'il soit composé ou simple

Solution

- on crée une interface `Composant`.
- le client manipule les objets via cette interface.

Composite : exemple

```cpp
struct Shape {
    virtual void draw() = 0;
};
struct Circle : Shape {
    void draw() { cout << "Cercle" << endl; }
};
struct Group : Shape {
    string              m_name;
    vector<Shape*>      m_objects;
    Group(const string &n) : m_name{n} {}
    void draw() {
      cout << "Groupe " << m_name.c_str() << " contiens:" << endl;
      for (auto &&o : m_objects)
        o->draw();
    }
};
```

Composite : instanciation

```cpp
int main() {
  Group root("root");
  root.m_objects.push_back(new Circle);
  Group subgroup("sub");
  subgroup.m_objects.push_back(new Circle);
  root.m_objects.push_back(&subgroup);
  root.draw();
  return EXIT_SUCCESS;
}
```

Adaptateur

Besoin d'utiliser un service rendu par une classe qui

- soit n'est pas encore développée (et dont *l'interface n'est pas fixée*)
- soit se trouve dans une bibliothèque existante mais *dont l'interface ne correspond* pas exactement à celle de l'application

exemples

- Adapter le changement de mode uniquement pour l'effet sur la pompe mais pas sur l'alarme
- action par interface graphique sur application mobile au lieu de panneau de contrôle.

Adaptateur : objectif (1)

Objectif

Adapter la classe à l'interface de la `Cible`

Solution 1 : modifier `Source`

mais elle peut être utilisée ailleurs ...

Adaptateur : objectif (2)

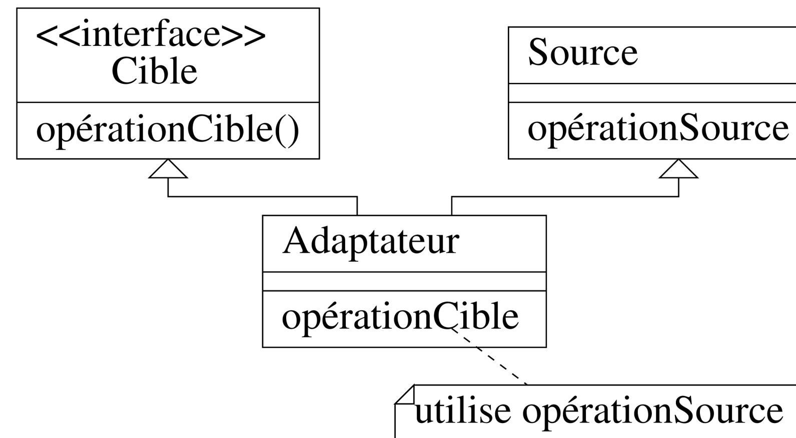
Objectif

Adapter la classe à l'interface de la `Cible`

`

Solution 2 : par héritage multiple

attention à la confusion des noms, `opérationSource` peut être redéfinie !



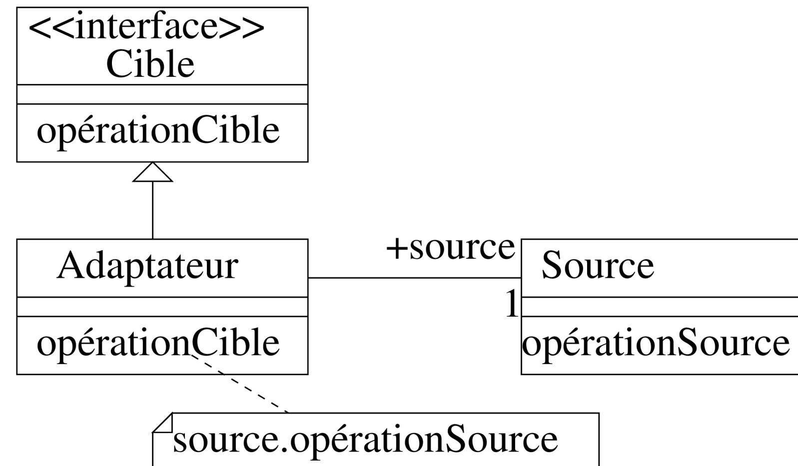
Adaptateur : objectif (3)

Objectif

Adapter la classe à l'interface de la `Cible`

`

Solution 3 : par délégation



Décorateur

Proposer des fonctionnalités nouvelles aux objets, sans réécrire ou modifier le code existant.

Décorateur : exemple

```
struct Shape {
  virtual operator string() = 0; };
struct Circle : Shape {
  float   m_radius;
  Circle(const float radius = 0) : m_radius{radius} {}
  void resize(float factor) { m_radius *= factor; }
  operator string() {
    ostringstream oss;
    oss << "Un cercle de rayon" << m_radius;
    return oss.str(); }
};
struct Square : Shape {
  float   m_side;
  Square(const float side = 0) : m_side{side} {}
  operator string() {
    ostringstream oss;
    oss << "Un carré de côté " << m_side;
    return oss.str(); }
};
```

Décorateur : exemple (2)

```
struct TransparentShape : Shape {
  const Shape&    m_shape;
  uint8_t         m_transparency;
  TransparentShape(const Shape& s, const uint8_t t) : m_shape{s}, m_transparency{t} {}
  operator string() {
      ostringstream oss;
      oss << string(const_cast<Shape&>(m_shape)) << " has "
          << static_cast<float>(m_transparency) / 255.f * 100.f
          << "% transparency";
      return oss.str();
  }
};
```

Décorateur : instanciation

```cpp
int main() {
  TransparentShape TransparentShape{ColoredShape{Square{5}, "green"}, 51};
  cout << string(TransparentShape) << endl;
  return EXIT_SUCCESS;
}
```

Bilan

Avantages

- Facilité l'ajout de fonctionnalités pour un objet existant à l'exécution et à la compilation;
- Améliore la flexibilité : pas de limite sur le nombre;
- Permet la rétrocompatibilité sans modifier les API.

Inconvénients

- Peut potentiellement compliqué l'instanciation d'un objet car il doit être envelopper dans des décorateurs pour avoir un sens.
- La sur-utilisation conduit à des difficultés d'apprentissage et de maintenance

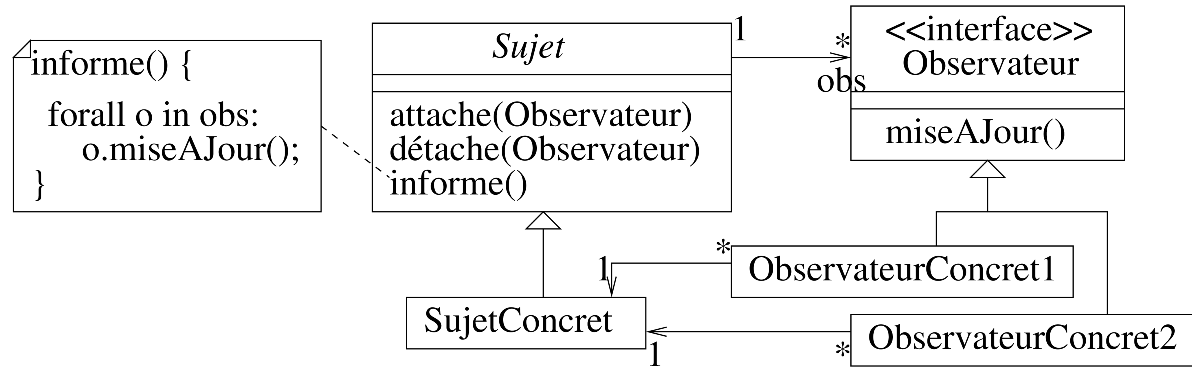
Patrons comportementaux

Observateur - l'intuition

Définir une dépendance entre un *sujet (observé)* et des *observateurs*

- Tout changement d'état du sujet est notifié aux observateurs

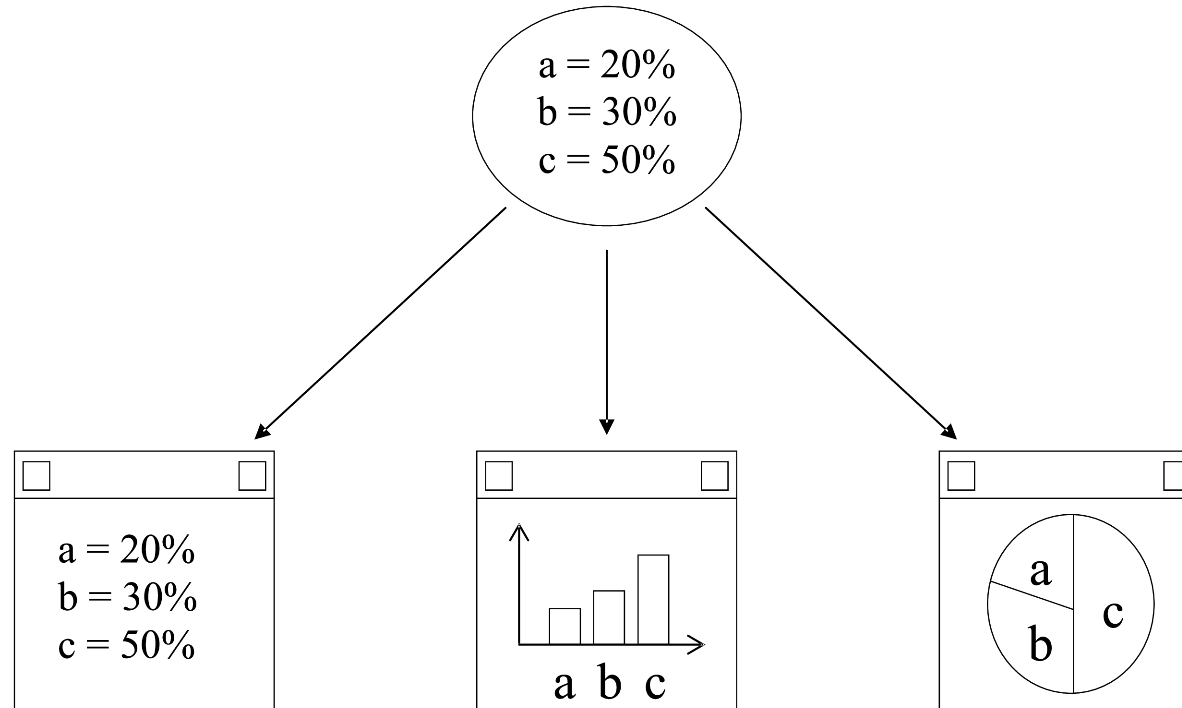
Observateur



Principe

- un ensemble d' associés à un
- quand le change d'état, il exécute
- un *ObservateurConcret* doit connaître son sujet
- la classe abstraite est un utilitaire

Observateur : idée



Observateur : exemple

```
template<typename T>
  struct Observer {
  virtual void field_changed(T& source, const string& field_name) = 0;
  };
template<typename T>
  struct Observable {
  void notify(T& source, const string& field_name) {
    for (auto observer: m_observers)
    observer->field_changed(source, field_name);
    }
  void subscribe(Observer<T>& observer) { m_observers.push_back(&observer); }
  void unsubscribe(Observer<T>& observer) {
    m_observers.erase(remove(m_observers.begin(), m_observers.end(), &observer), m_observers.end());
  }
  private:
  vector<Observer<T>*>  m_observers;
};
```

Observateur : exemple (2)

```cpp
struct Person : Observable<Person>{
  void set_age(uint8_t age) {
    auto old_can_vote = get_can_vote();
    this->m_age = age;
    notify(*this, "age");
    if (old_can_vote != get_can_vote()) notify(*this, "can_vote");
  }
  uint8_t get_age() const { return m_age; }
  bool get_can_vote() const { return m_age >= 16; }
private:
  uint8_t m_age{0};
};
```

Observateur : exemple (3)

```
struct TrafficAdministration : Observer<Person> {
  void field_changed(Person &source, const string &field_name) {
    if (field_name == "age") {
      if (source.get_age() < 17)
        cout << "Not old enough to drive!\n";
      else {
        cout << "Mature enough to drive!\n";
        source.unsubscribe(*this); }
    }
  }
};
int main() {
  Person p;
  TrafficAdministration ta;
  p.subscribe(ta);
  p.set_age(16);
  p.set_age(17);
  return EXIT_SUCCESS;
}
```

Stratégie, Commande, État

Trois patrons comportementaux aux caractéristiques communes :

- associer des objets à certains traitements
- définir une hiérarchie de classe des traitements
- accéder aux traitements de manière uniforme
- découpler données et traitement
- séparer hiérarchie des traitement et classes utilisatrices

Stratégie/Commande/État

Stratégie

Implémenter une operation de plusieurs façons Choisir à l'exécution laquelle utiliser

Commande

Associer une même commande à plusieurs objets Gérer des séquences et historiques de commandes

État

Plusieurs opérations associées à un état qui évolue

Stratégie

Objectif

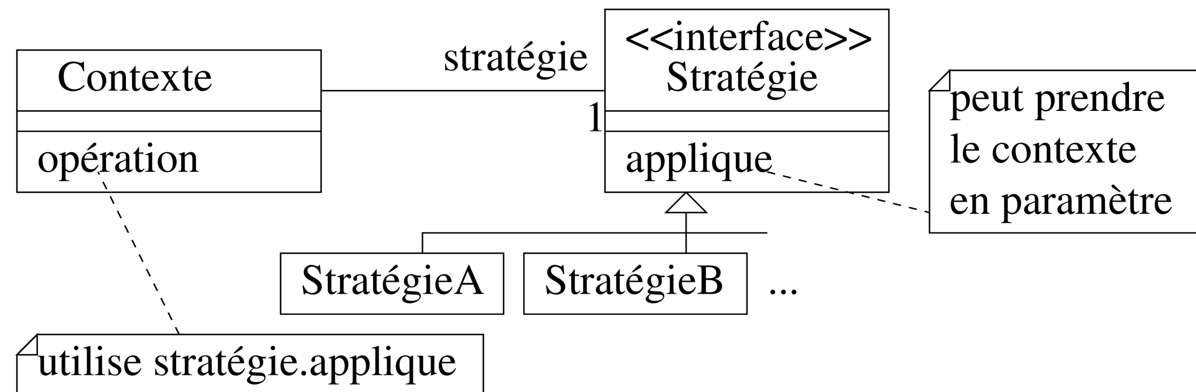
- Encapsuler des *algorithmes* dans des objets
- Pour les rendre *dynamiquement interchangeables*

Motivation

Constat : il existe souvent plusieurs algorithmes pour résoudre un problème.

On voudrait pouvoir choisir *à l'exécution* lequel utiliser.

Stratégie (2)



Stratégie : exemple (1)

```
enum class Format { Markdown, Html };
struct ListStrategy {
  virtual ~ListStrategy() = default;
  virtual void add_list_item(ostringstream& oss, string& item) {};
  virtual void start(ostringstream& oss) {};
  virtual void end(ostringstream& oss) {};
};
struct MarkdownListStrategy: ListStrategy {
  void add_list_item(ostringstream& oss, string& item) override
    { oss << " - " << item << endl; }
};
struct HtmlListStrategy: ListStrategy {
  void start(ostringstream& oss) override { oss << "<ul>" << endl; }
  void end(ostringstream& oss) override { oss << "</ul>" << endl; }
  void add_list_item(ostringstream& oss, string& item) override
    { oss << "\t<li>" << item << "</li>" << endl; }
};
```

Strategie : exemple (2)

```cpp
struct TextProcessor {
  void clear() {
    m_oss.str("");
    m_oss.clear(); }
  void append_list(vector<string>& items) {
    m_list_strategy->start(m_oss);
    for (auto& item: items)
      m_list_strategy->add_list_item(m_oss, item);
    m_list_strategy->end(m_oss); }
  void set_output_format(Format& format) {
    switch (format) {
      case Format::Markdown: m_list_strategy =
        make_unique<MarkdownListStrategy>(); break;
      case Format::Html: m_list_strategy =
        make_unique<HtmlListStrategy>(); break;
    } }
  string str() { return m_oss.str(); }
  private:
    ostringstream                m_oss;
    unique_ptr<ListStrategy>     m_list_strategy;
};
```

Strategie : exemple (3)

```cpp
int main() {
  // markdown
  TextProcessor tp;
  tp.set_output_format(Format::Markdown);
  tp.append_list({ "foo", "bar", "baz" });
  cout << tp.str() << endl;
  // html
  tp.clear();
  tp.set_output_format(Format::Html);
  tp.append_list({ "foo", "bar", "baz" });
  cout << tp.str() << endl;
  return EXIT_SUCCESS;
}
```

Stratégie : synthèse

Avantages

- dérivation de possible (indépendamment de la choisie)
- changement dynamique de stratégie

Inconvénients

- mémoire (objet stratégie)
- coût d'indirection (si pas d'optimisation dynamique)

Commande

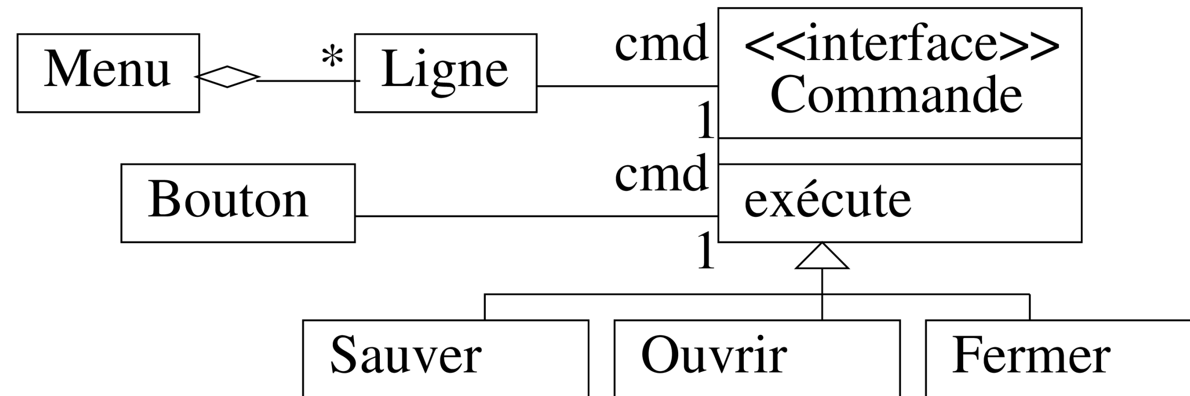
Objectif

Encapsuler des *commandes* dans des objets

Motivation

- partager des actions (e.g. IHM)
- même action dans un menu et sur un bouton
- stocker un historique (log, undo)

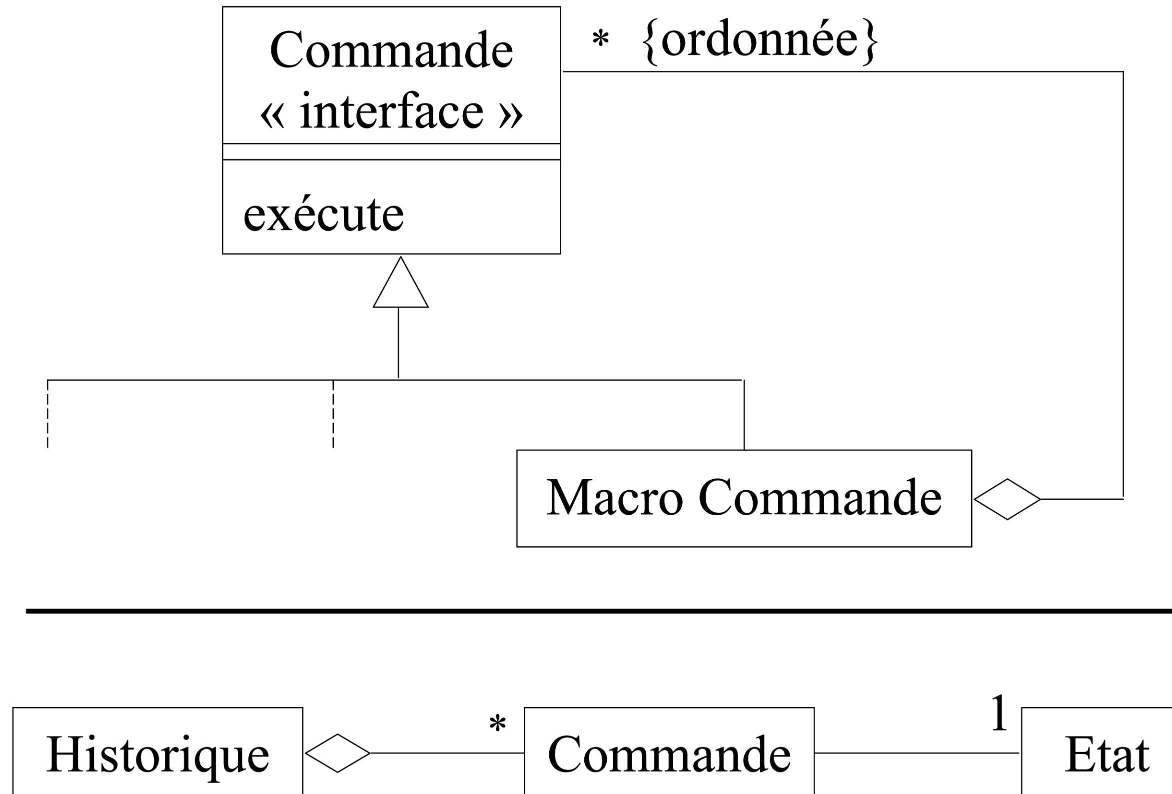
Exemple de patron commande (1)



Avantages

- Modification à l'exécution du menu
- Partage (menu, bouton)

Exemple de patron commande (2)



Exemple (1) : Commande

```cpp
class Command {
  public:
    virtual void execute() = 0;
};

// Receiver Class
class Light {
public:
  void on() { cout << "The light is on\n"; }
  void off() { cout << "The light is off\n"; }
};
```

Exemple (1) : Dérivation

```cpp
// Command for turning on the light
class LightOnCommand : public Command {
public:
  LightOnCommand(Light *light) : mLight(light) {}
  void execute() { mLight->on(); }

private:
  Light *mLight;
};

// Command for turning off the light
class LightOffCommand : public Command {
public:
  LightOffCommand(Light *light) : mLight(light) {}
  void execute() { mLight->off(); }

private:
  Light *mLight;
};
```

Exemple (1) : Invoquer

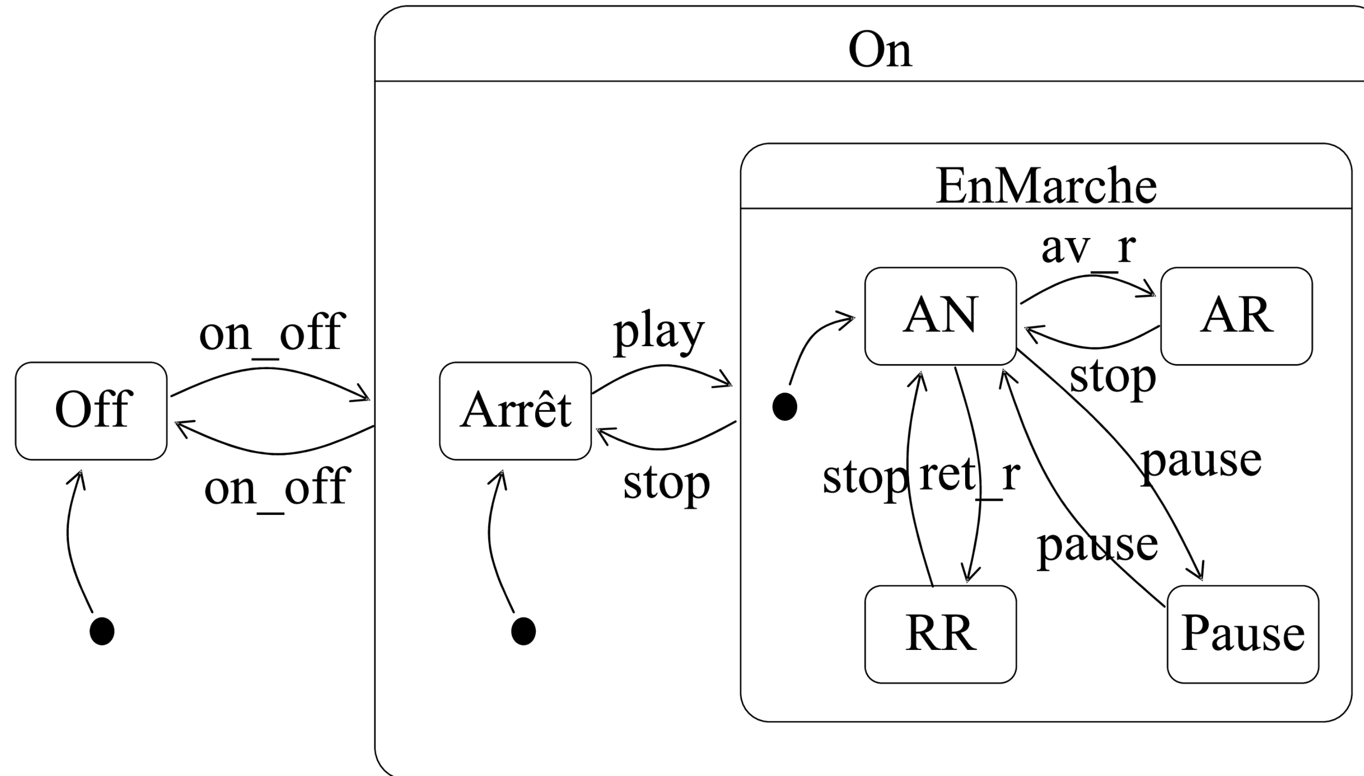
```cpp
class RemoteControl {
public:
  void setCommand(Command *cmd) { mCmd = cmd; }
  void buttonPressed() { mCmd->execute(); }

private:
  Command *mCmd;
};
```

Exemple (1) : Main

```
// The client
int main() {
  // Receiver
  Light *light = new Light;
  // concrete Command objects
  LightOnCommand *lightOn = new LightOnCommand(light);
  LightOffCommand *lightOff = new LightOffCommand(light);
  // invoker objects
  RemoteControl *control = new RemoteControl;
  // execute
  control->setCommand(lightOn);
  control->buttonPressed();
  control->setCommand(lightOff);
  control->buttonPressed();
  delete light, lightOn, lightOff, control;
  return 0;
}
```

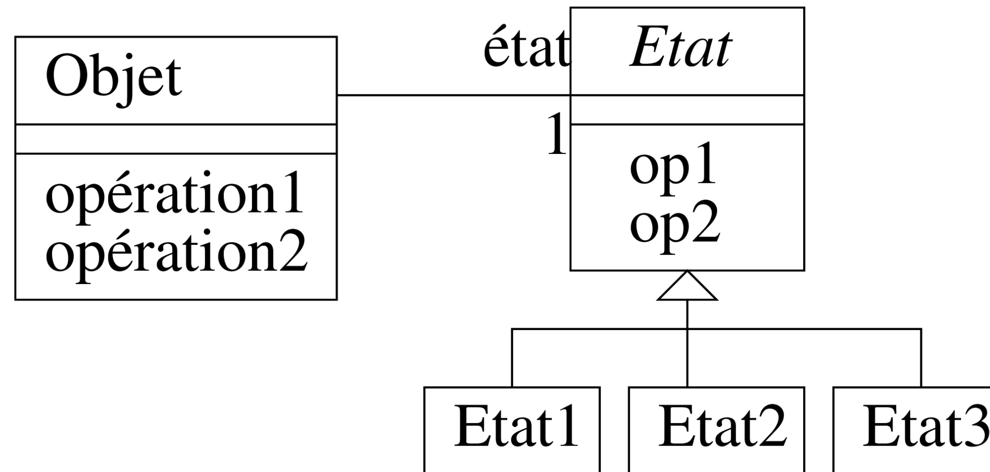
Patron État : l'intuition (platine cassette)



État

Objectif

Réaliser des objets dont le comportement change lorsque leur état interne est modifié.



État : Fonctionnement

- l'objet délègue une partie de son comportement à son état
- la relation `etat` change dans le temps
- `op1` et `op2` peuvent recevoir l'objet en paramètre au besoin

Patron à état : exemple

```
class Etat {
public:
  virtual void parite() = 0;
  virtual Etat *next() = 0;
};
```

- la méthode renvoie, pour un état élémentaire, une instance de cet état, et pour un état composite, une instance de l'état initial du sous-automate correspondant.
- A chaque événement qui peut être reçu par la platine est associée une méthode qui retourne l'état de la platine après les actions correspondantes.

Interprète

Objectif

Étant donné un langage, il permet de définir une représentation pour sa grammaire abstraite (autrement dit, une structure d'arbre abstrait), ainsi qu'un interprète utilisant cette représentation.

Interprète : exemple (1)

```
struct Token {
  enum Type { integer, plus, minus, lparen, rparen };
  Type m_type;
  string m_text;
  Token(Type typ, const string &txt) : m_type(typ), m_text(txt) {}
  friend ostream &operator<<(ostream &os, const Token &o) {
    return os << "`" << o.m_text << "`";
  }
};
```

Interprète : exemple (2)

```
vector<Token> lex(const string& input) {
  vector<Token>   result;
  for (auto curr = begin(input); curr != end(input); ++curr) {
    switch (*curr) {
      case '+': result.emplace_back(Token::plus, "+"); break;
      case '-': result.emplace_back(Token::minus, "-"); break;
      case '(': result.emplace_back(Token::lparen, "("); break;
      case ')': result.emplace_back(Token::rparen, ")"); break;
      default: // number
        auto first_not_digit = find_if(curr, end(input), [](auto c) {
          return !isdigit(c);
        });
        string integer = string(curr, first_not_digit);
        result.emplace_back(Token::integer, integer);
        curr = --first_not_digit;
    }
  }
  return result;
}
```

Interprète : exemple (3)

```cpp
int main() {
  auto tokens = lex("(13-4)-(12+1)");
  for (auto &t : tokens)
    cout << t << " ";
  // Output: `(` `13` `-` `4` `)` `-` `(` `12` `+` `1` `)`
  return EXIT_SUCCESS;
}
```

Interprète (1)

Avantages

- On peut facilement modifier et étendre la grammaire. Par exemple, on peut facilement ajouter de nouvelles expressions en définissant de nouvelles classes.
- L'implémentation de la grammaire est simple, et peut être réalisée automatiquement à l'aide d'outils de génération.

Interprète (2)

Incovénients

- Lorsque la grammaire est complexe, on a une multiplication des classes.
- Il devient alors délicat d'ajouter de nouvelles opérations, car celles-ci doivent être ajoutées dans toutes les classes de la hiérarchie.

Visiteur

Objectifs (1)

- Représenter une opération définie en fonction de la structure d'un objet. Il permet alors de définir de nouvelles opérations sans modifier les classes qui définissent la structure des objets auxquelles elles s'appliquent.

Visiteur (2)

Principe

- On considère une hiérarchie de classes définissant la structure d'objets. On suppose qu'on a une classe abstraite `Element` , racine de cette hiérarchie.
- On définit une interface Visiteur, qui contient une méthode `void visite(ElementX e)` pour chaque sous-classe concrète `ElementX` de `Element` .

Visiteur : exemple (1)

```
struct Document {
  virtual void add_to_list(const string &line) = 0;
};
struct Markdown : Document {
  void add_to_list(const string &line) { m_content.push_back(line); }
  string m_start = "* ";
  list<string> m_content;
};
struct HTML : Document {
  void add_to_list(const string &line) { m_content.push_back(line); }
  string m_start = "<li>";
  string m_end = "</li>";
  list<string> m_content;
};
```

Visiteur : exemple (2)

```
struct DocumentPrinter {
  void operator()(Markdown &md) {
    for (auto &&item : md.m_content)
      cout << md.m_start << item << endl;
  }
  void operator()(HTML &hd) {
    cout << "<ul>" << endl;
    for (auto &&item : hd.m_content)
      cout << "\t" << hd.m_start << item << hd.m_end << endl;
    cout << "</ul>" << endl;
  }
};
```

Visiteur : exemple (3)

```cpp
using document = std::variant<Markdown, HTML>;
int main() {
  HTML hd;
  hd.add_to_list("This is line");
  document d = hd;
  DocumentPrinter dp;
  std::visit(dp, d);
  return EXIT_SUCCESS;
}
```

Visiteur (1)

- oblige à traiter tous les cas; est vérifié à la compilation
- une programmation *purement objet* ;
- évite l'utilisation de `dynamic_cast`, de devoir déclarer une variable initialisée à l'aide d'une conversion, et évite le risque d'erreur de conversion à l'exécution ;
- évite d'effectuer plusieurs tests pour trouver le code à exécuter;
- permet l'ajout de nouvelles opérations sans modifier la hiérarchie de classes, et l'ajout de nouvelles classes (en modifiant l'interface).

Visiteur (2)

- Ce patron est lourd à mettre en oeuvre : il faut prévoir une méthode par classe;
- Le traitement de méthodes comportant des paramètres et un résultat est également lourd ;
- on a un surcoût à chaque indirection ;
- le code est peu lisible lorsqu'on ne connaît pas le patron.

Conclusion

Les patrons que l'on a vu

- Patrons de création
  - singleton
  - méthode fabrique
  - fabrique abstraite
- Patrons de structure
  - composite
  - adaptateur
  - décorateur (exercice)
- Patrons de comportement
  - stratégie, commande, état
  - observateur
  - interprète
  - visiteur
  - ...