Design Patterns

E. RAMAT

Université du Littoral - Côte d'Opale

18 janvier 2007

Plan

Objectifs

- Comprendre les bases de la philosophie des “formes de conception” ;
- Connaître le vocabulaire spécifique ;
- Connaître quelques “patterns”
- Concevoir objet différemment ;
- Traduire les patterns en C++.

Définition

- Description d'une solution classique à un problème récurrent ;
- Décrit une partie de la solution . . . avec des relations avec le système et les autres parties . . . ;
- C'est une technique d'architecture logicielle.

Ce que ce n'est pas . . .

- Une brique : un pattern dépend de son environnement ;
- Une règle : un pattern ne peut pas s'appliquer mécaniquement ;
- Une méthode : ne guide pas une prise de décision ; un pattern est la décision prise.

Avantages

- Un vocabulaire commun ;
- Capitalisation de l'expérience ;
- Un niveau d'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité ;
- Réduire la complexité ;
- Guide/catalogue de solutions.

Inconvénients

- Effort de synthèse ; reconnaître, abstraire . . . ;
- Apprentissage, expérience ;
- Les patterns “se dissolvent” en étant utilisés.

Description

- nom : augmente le vocabulaire, réifie une idée de solution, permet de mieux communiquer ;
- problème : quand appliquer la forme, le contexte . . . ;
- solution : les éléments de la solution, leurs relations, responsabilités, collaborations. Pas de manière précise, mais suggestives. . . ;
- conséquences : résultats et compromis issus de l'application de la forme.

Bibliographie

- La Bible : Design pattern: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson et J. Vlissides ;
- Head First : Design Patterns, Eric Freeman et Elisabeth Freeman, édition o'reilly ;
- Modern C++ Design, Generic Programming and Design Patterns Applied, Andrei Alexandrescu, C++ In-Depth Series, Bjarne Stroustrup, 323 p., 2001 ;
- Advanced C++ Programming Styles and Idioms, James O. Coplien, édition Addison-Wesley, 544 p., 1991 ;
- http://home.earthlink.net/ huston2/dp/patterns.html ;
- http://www.vico.org/pages/PatronsDisseny.html

Types de design pattern

- Création
  - abstraire le processus d'instanciation ;
  - rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés ;
  - encapsuler la connaissance de la classe concrète qui instancie ;
  - cacher ce qui est créé, qui crée, comment et quand.
- Structure :
  - comment les objets sont assemblés ;
- Comportement, pour décrire :
  - des algorithmes ;
  - des comportements entre objets ;
  - des formes de communication entre objets.

Plan

Introduction

- singleton : instance unique ;
- factory method : constructeur virtuel ;
- abstract factory : fabrique d'objets ;
- prototype : construction à partir de prototypes ;

Plan

Singleton

Utilisation

- on utilise le **Singleton** lorsque :
  - il n'y a qu'une unique instance d'une classe et qu'elle doit être accessible de manière connue ;
  - une instance unique peut être sous-classée et que les clients peuvent référencer cette extension sans avoir à modifier leur code ;
  - l'instance doit disparaître à la fin du programme.
- des exemples : l'horloge système, le manager d'imprimantes, le clavier, le générateur aléatoire, ...

Singleton

Première traduction C++

Singleton

```
class Singleton
{
public:
  static Singleton* instance()
  {
    if (!pInstance) pInstance = new Singleton;
    return pInstance;
  }
// operations
private:
  Singleton();
  Singleton(const Singleton&);

  static Singleton* pInstance;
};

Singleton* Singleton::pInstance = 0;
```

Singleton

Commentaires

- un attribut statique (attribut de classe) pInstance ;
- allocation dynamique du Singleton s'il n'existe pas ;
- initialisation à nul du Singleton ;
- les constructeurs sont privés pour éviter la création d'instance ;
- inconvénients :
  - le Singleton n'est jamais détruit ;
  - cette implémentation supporte-t-elle l'héritage ? ;
  - que se passe-t-il en multi-threading ?

Singleton

```
class Singleton
{
public:
  static Singleton& instance()
  {
    static Singleton pInstance;
    return pInstance;
  }
// operations
private:
  Singleton();
  Singleton(const Singleton&);
};
```

Singleton

Commentaires

- une instance locale et statique pInstance ;
- pInstance est créée au premier appel de la fonction instance() ;
- pInstance est détruite à la fin de l'application ;
- problème :
  - s'il existe plusieurs classes de type singleton, l'ordre des destructions n'est pas garanti ;
  - des dépendances entre singletons peuvent exister.

Singleton

La fonction instance()

intance()

```
static Singleton& instance()
{
  extern void __ConstructSingleton(void* memory);
  extern void __DestroySingleton();
  static bool __initialized = false;
  static char __buffer[sizeof(Singleton)];
  if (!__initialized)
  {
    __ConstructSingleton(__buffer);
    atexit(__DestroySingleton);
    __initialized = true;
  }
  return *reinterpret_cast<Singleton*>(__buffer);
}
```

Singleton

Commentaires

- les fonctions __ConstructSingleton et __DestroySingleton sont les traductions C des appels au constructeur et au destructeur ;
- la fonction C atexit enregistre le fait que la fonction passée en paramètre doit être appelé à la fin du programme ;
- `reinterpret_cast < Singleton* >` est un cast C sans contrôle de type (contrairement à static_cast ou dynamic_cast).
- problème : s'il existe des dépendances entre singletons.

TD

La suite en TD !

Singleton

Deuxième traduction C++ (version Meyer)

Singleton

```
template<typename T> class Singleton
{
public:
  static T& instance()
  {
    static T pInstance;
    return pInstance;
  }
// operations
private:
  T();
  T(const T&);
};
```

Plan

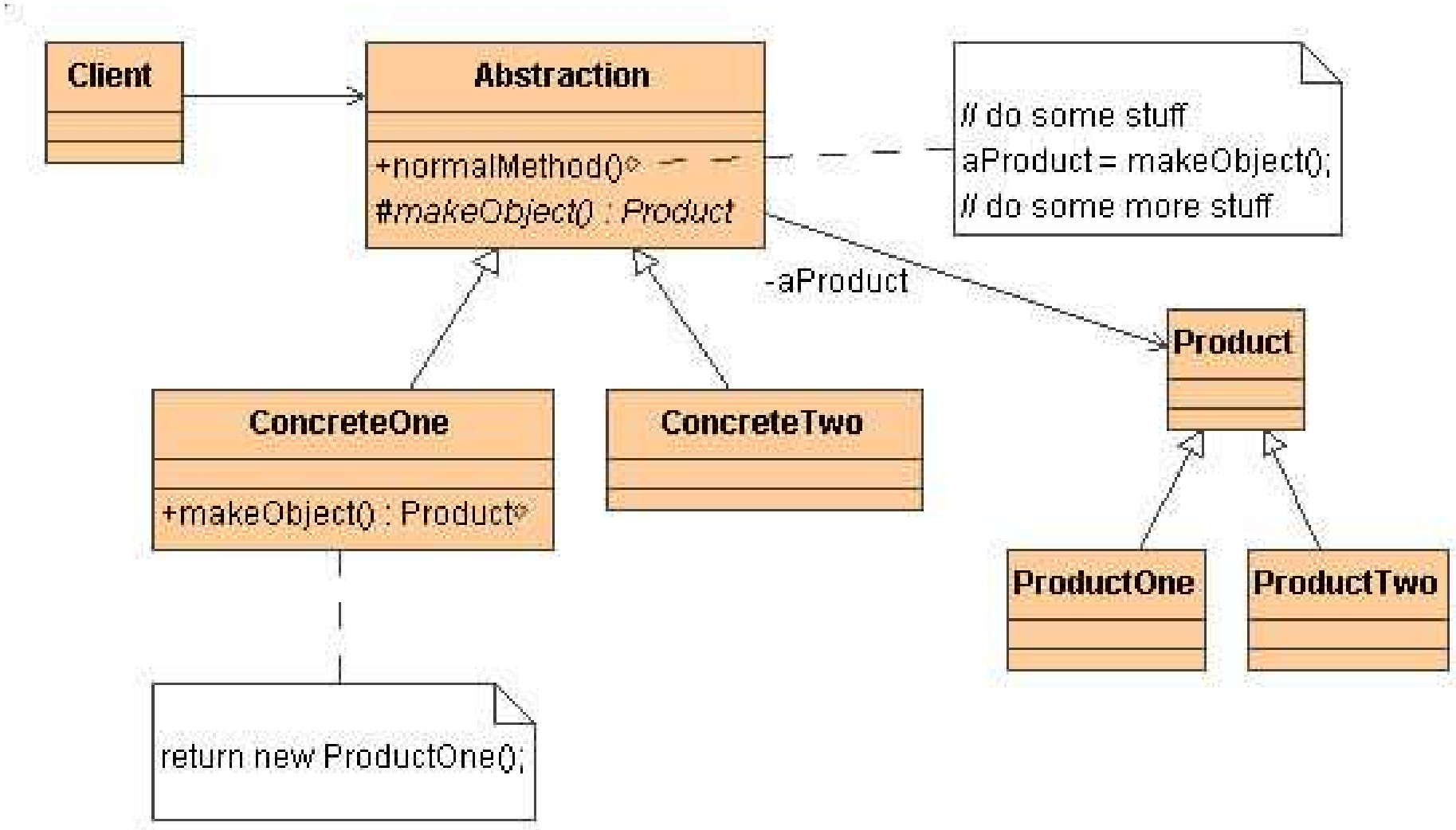
Factory Method

Utilisation

- on utilise le **Factory Method** lorsque :
  - l'on définit une interface pour créer un objet, mais on laisse les sous-classes décident quelle classe doit être instantiée ;
  - une classe ne peut anticiper la classe de l'objet qu'elle doit construire ;
  - une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique ;
- l'opérateur `new` est considéré comme inutilisable ;
- aussi appelé `virtual constructor` ;
- des exemples : un éditeur multi-documents où chaque type de document a besoin d'un éditeur spécifique . . .

Factory Method

Diagramme UML



Factory Method

```
class DocumentManager
{
public:
  Document* newDocument();
  virtual Document* createDocument() = 0;
private:
  std::list<Document*> listOfDocs;
};
Document* DocumentManager::newDocument()
{
  Document* pDoc = createDocument();

  listOfDocs.push_back(pDoc);
  return pDoc;
}
Document* GraphicDocumentManager::createDocument()
{
  return new GraphicDocument;
}
```

Factory Method

Commentaires

- la fonction createDocument remplace l'opérateur `new` ;
- la fonction newDocument ne peut pas utiliser l'opérateur `new` car elle ne connaît pas la classe concrète de l'objet à créer ;
- la fonction createDocument est alors abstraite et déléguée aux sous-classes.

Plan

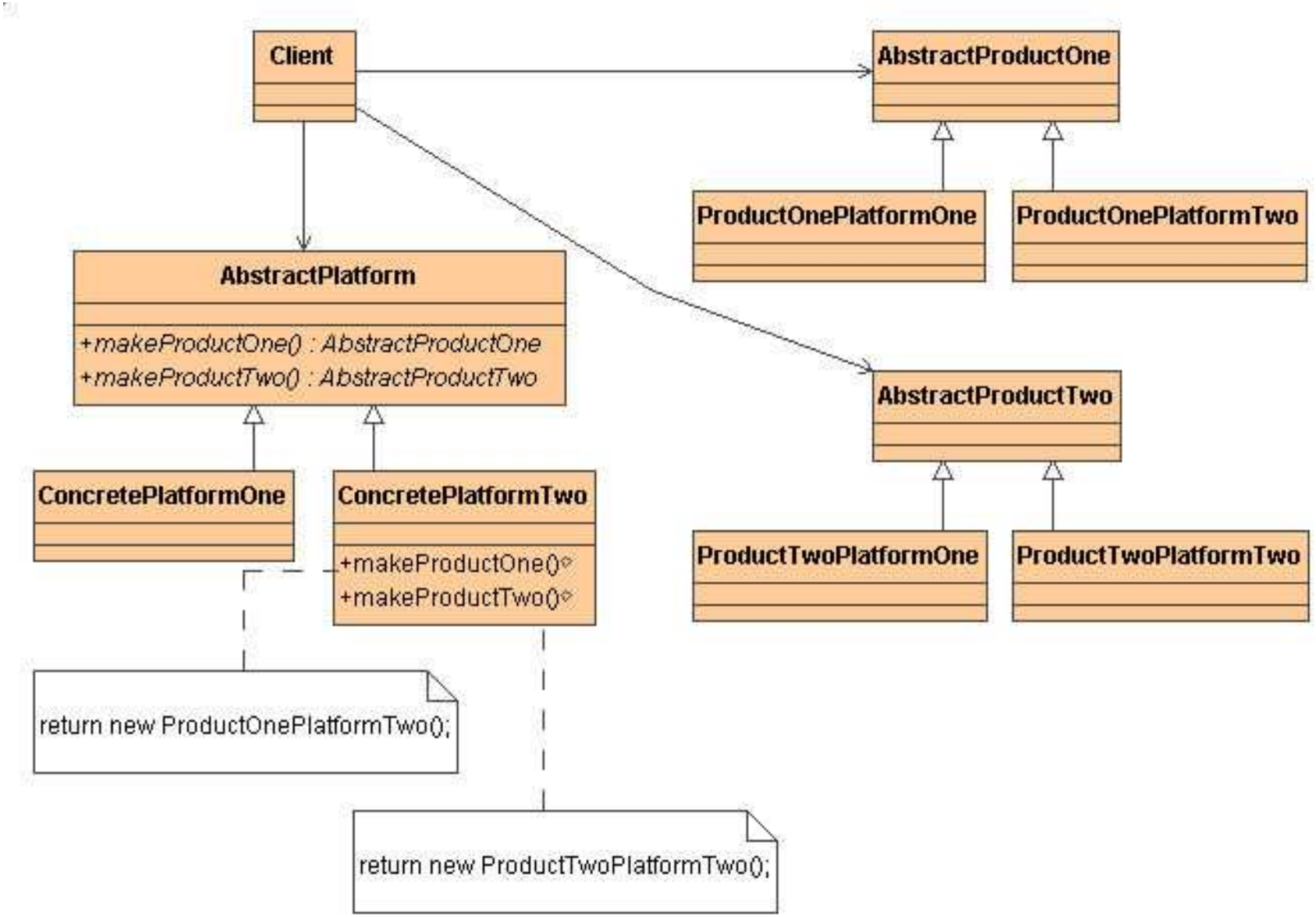
Abstract Factory

Utilisation

- on utilise le **Abstract Factory** lorsque :
  - l'on doit fournir une interface pour créer une famille de composants interdépendants ou non sans spécifier leur classe d'appartenance concrète ;
  - un système doit être indépendant de la façon dont ses composants sont créés, assemblés, représentés ;
  - un système repose sur un composant d'une famille de composants ;
  - on veut définir une interface unique à une famille de composants concrets ;
  - l'operateur `new` est considéré comme inutilisable ;
  - Factory Method est similaire à Abstract Factory mais sans l'hypothèse sur les familles.
- des exemples : un éditeur multi-documents où chaque type de document a besoin d'un éditeur spécifique . . .

Abstract Factory

Diagramme UML



Plan

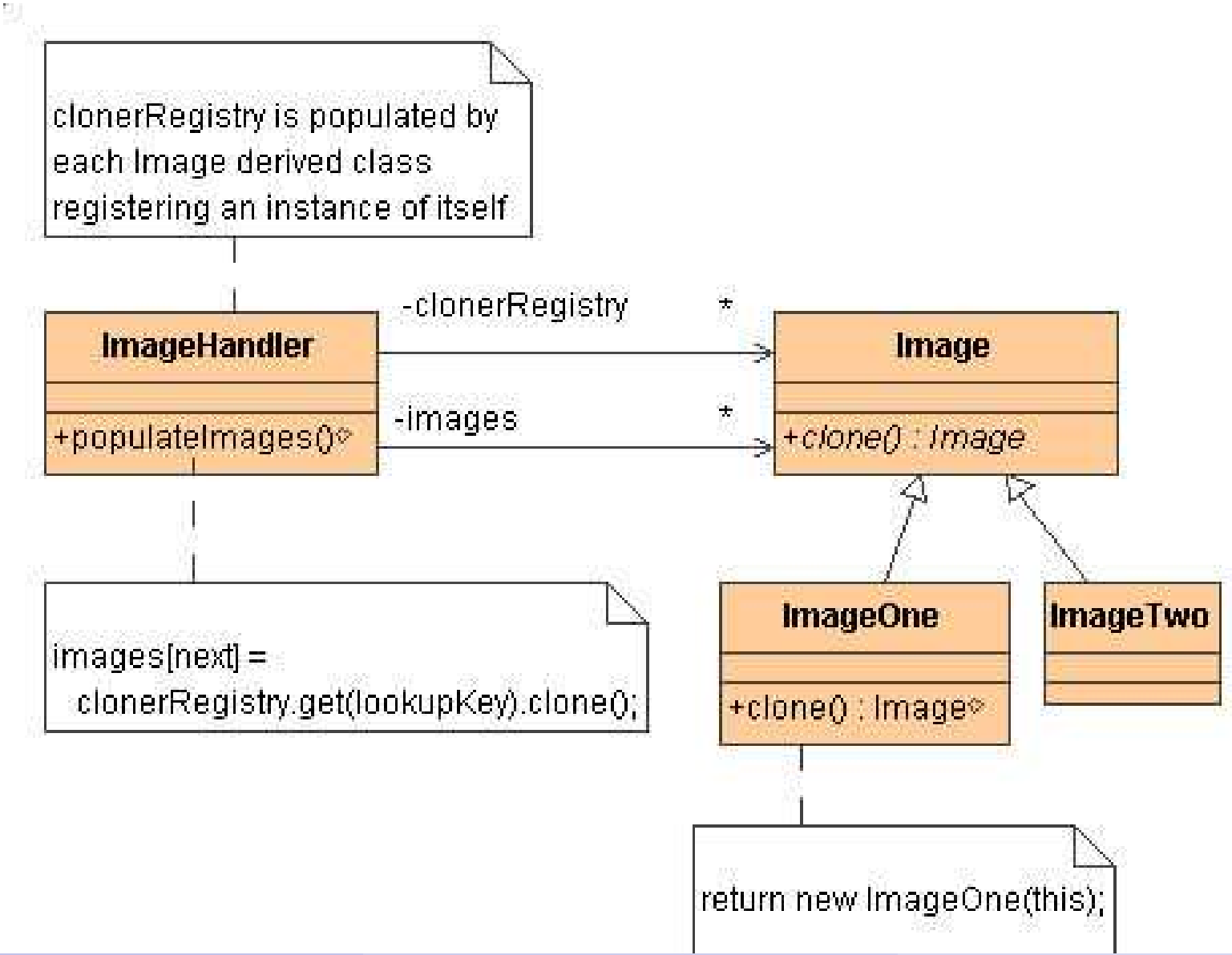
Prototype

Utilisation

- on utilise le **Prototype** lorsque :
  - l'on spécifie les objets de l'application par prototypage ;
  - et que l'on crée les nouveaux objets par copie des prototypes ;
  - pour éviter une hiérarchie de Factory parallèle à une hiérarchie de composants (comme dans le pattern Factory Method) ;
  - l'operateur `new` est considéré comme inutilisable.
- appelé aussi Exemplaire (voir Advanced C++ Programming Styles and Idioms, James O. Coplien).

Prototype

Diagramme UML



Prototype

```
class Shape
{
public:
  virtual Shape* clone() const = 0;
};

class Line:public Shape
{
public:
  virtual Line* clone() const
  {
    return new Line(*this);
  }
};
```

- la fonction abstraite clone a été surchargé d'une façon spéciale grâce à la notion de "covariant return type" ;
- on doit retourner un pointeur sur un type dérivé du type initial ;
- problème :
  - s'il existe plusieurs niveaux d'héritage, il ne faut pas oublier de surcharger la fonction clone dans les sous-sous-classes.

Prototype

Contrôle de l'implémentation

Prototype

```
class Shape
{
public:
  Shape* clone() const
  {
    Shape* pClone = doClone();
    assert(typeid(*pClone) == typeid(*this));
    return pClone;
  }
protected:
  virtual Shape* doClone() = 0;
};
```

typeid

La fonction typeid permet d'accèder à l'identifiant du type réel de l'objet.

Prototype

Commentaires

- comment reconstruire un objet envoyé dans un flux (dans un fichier, par exemple) en ne sachant seulement qu'il appartient à une hiérachie de classes ?

Flux

```
class Drawing
{
public:
  void save(std::ofstream& out);
  void load(std::ifstream& in);
};

void Drawing::save(std::ofstream& out)
{
  // écrire une entête
  for (chaque élément du dessin)
  {
    (élément courant)->save(out);
  }
}
```

Prototype

Commentaires

- la classe Drawing possède une liste hétérogène d'objets de type Shape ;
- l'enregistrement n'est pas un problème ;
- il faut s'assurer que chaque objet (comme Drawing) possède un identifiant et qu'il enregistre afin de retrouver le type de l'objet enregistré ;
- problème : comment automatiser la lecture et la reconstruction des objets ?

TD

La suite en TD !

Plan

Introduction

- proxy : utiliser un composant intermédiaire pour accèder un autre composant.
- flyweight : externalisation de l'état et utilisation de représentants ;
- adapter : obtenir un objet qui permet d'en utiliser un autre en conformité avec une certaine interface ;
- bridge : découplage de l'abstraction de l'implémentation ;
- composite : objets composés d'objets homogènes ;
- decorator : ajout dynamique de comportements ou d'états à un composant ;
- facade : interface simple pour un système complexe ;

Plan

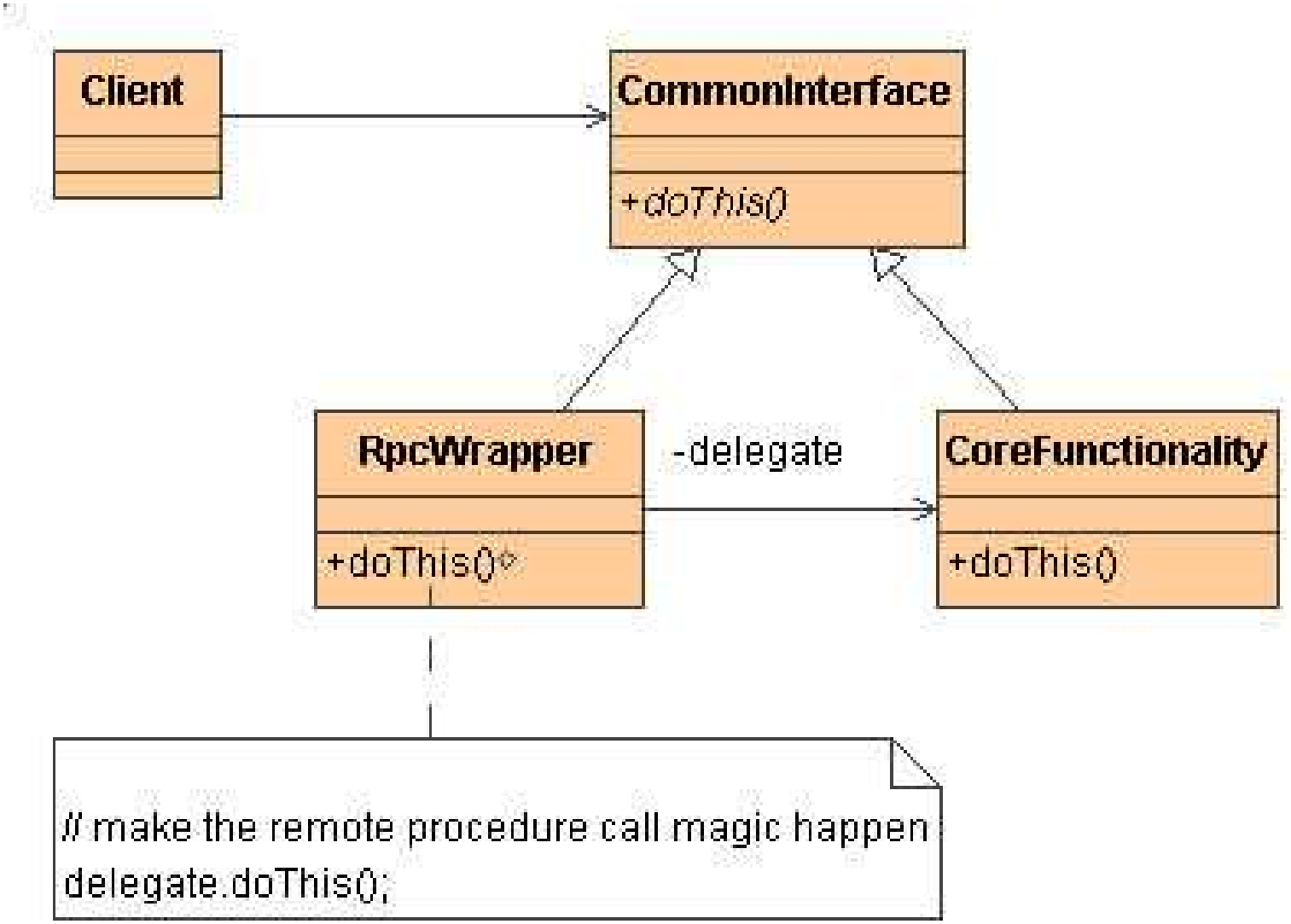
Proxy

Utilisation

- on utilise le **Proxy** lorsque l'on veut référencer un objet par un moyen plus complexe qu'un "simple" pointeur :
  - remote proxy : ambassadeur
  - protection proxy : contrôle d'accès
  - référence intelligente : persistence, comptage de références, ...
- un exemple : un objet qui reporte les opérations coûteuses au moment où on utilise réellement les résultats de ces opérations (chargement d'une image à la fin d'un document, ...).

Proxy

Diagramme UML



- les “smart pointers” sont des objets C++ qui simulent les pointeurs classiques en implémentant les opérateurs -> et * ;
- les compteurs de références font parti de cette catégorie ;
- disponibles dans la glibmm : classe RefPtr.

Smart pointers

```
template <class T> class SmartPtr
{
public:
  explicit SmartPtr(T* pointee) : pointee_(pointee);
  SmartPtr& operator=(const SmartPtr& other);
  ~SmartPtr();
  T& operator*() const
  {
    ...
    return *pointee_;
  }
  T* operator->() const
  {
    ...
    return pointee_;
  }
private:
  T* pointee_;
  ...
};
```

Proxy

Commentaires

- objectif principal : optimiser la présentation des objets en mémoire ;
- un seul représentant pour plusieurs références **tant que** le représentant n'est pas modifié !
- plusieurs stratégies :
  - une copie à chaque invocation du constructeur par recopie (`deep copy`) ;
  - une copie seulement en cas de modification (`copy on write`) - les fonctions non constantes ;

- trois stratégies, le smart pointer référence :
  - un compteur et l'objet réel ;
  - une structure composée d'un compteur et d'un pointeur sur l'objet réel ;
  - l'objet réel et l'objet réel embarque le compteur ;
- les deux premiers ajoutent une indirection coûteuse ;
- dans ce dernier cas, il faut modifier la classe d'appartenance de l'objet.

Plan

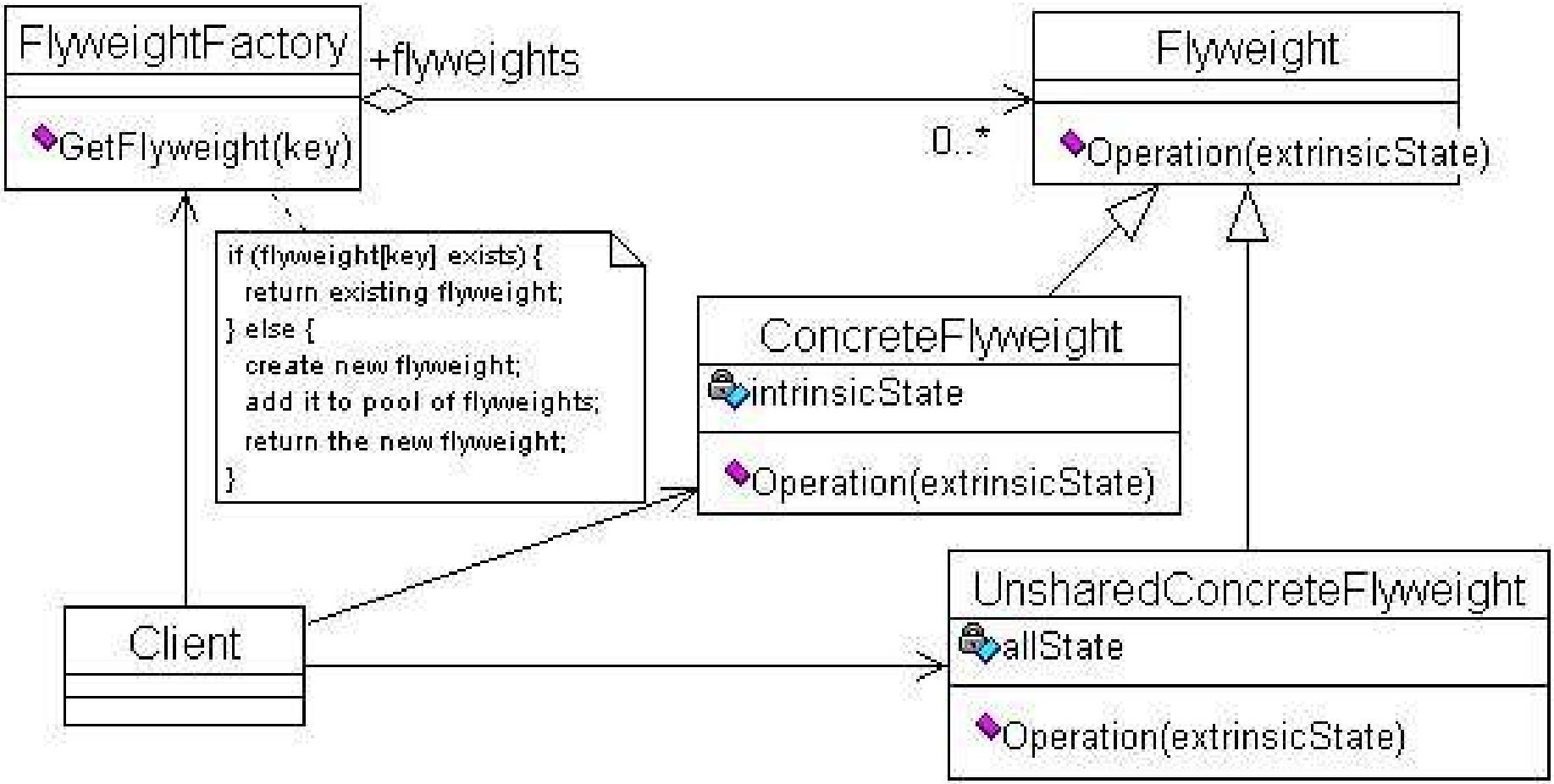
Flyweight

Utilisation

- on utilise le **Flyweight** lorsque :
  - on utilise beaucoup d'objets ;
  - les coûts de sauvegarde sont élevés ;
  - l'état des objets peut être externalisé (extrinsic) ;
  - de nombreux groupes d'objets peuvent être remplacés par quelques objets partagés une fois que les états sont externalisés ;
  - l'application ne dépend pas de l'identité des objets ;
- un exemple : les caractères manipulés dans un traitement de texte ;
  - chaque caractère correspond à un objet ayant une police de caractères, une taille de caractères, et d'autres données de formatage ;
  - un long document contenant beaucoup de caractères ainsi implémentés ...

Flyweight

Diagramme UML



Plan

Adapter

Utilisation

- on utilise l'**Adapter** lorsque on veut utiliser :
  - une classe existante dont l'interface ne convient pas ;
  - plusieurs sous-classes mais il est coûteux de redéfinir l'interface de chaque sous-classe en les sous-classant ;
  - un "adapter" peut adapter l'interface au niveau du parent ;
- appelé aussi "wrapper" ;
- exemple : mise en "conformité" de composants d'origines diverses.

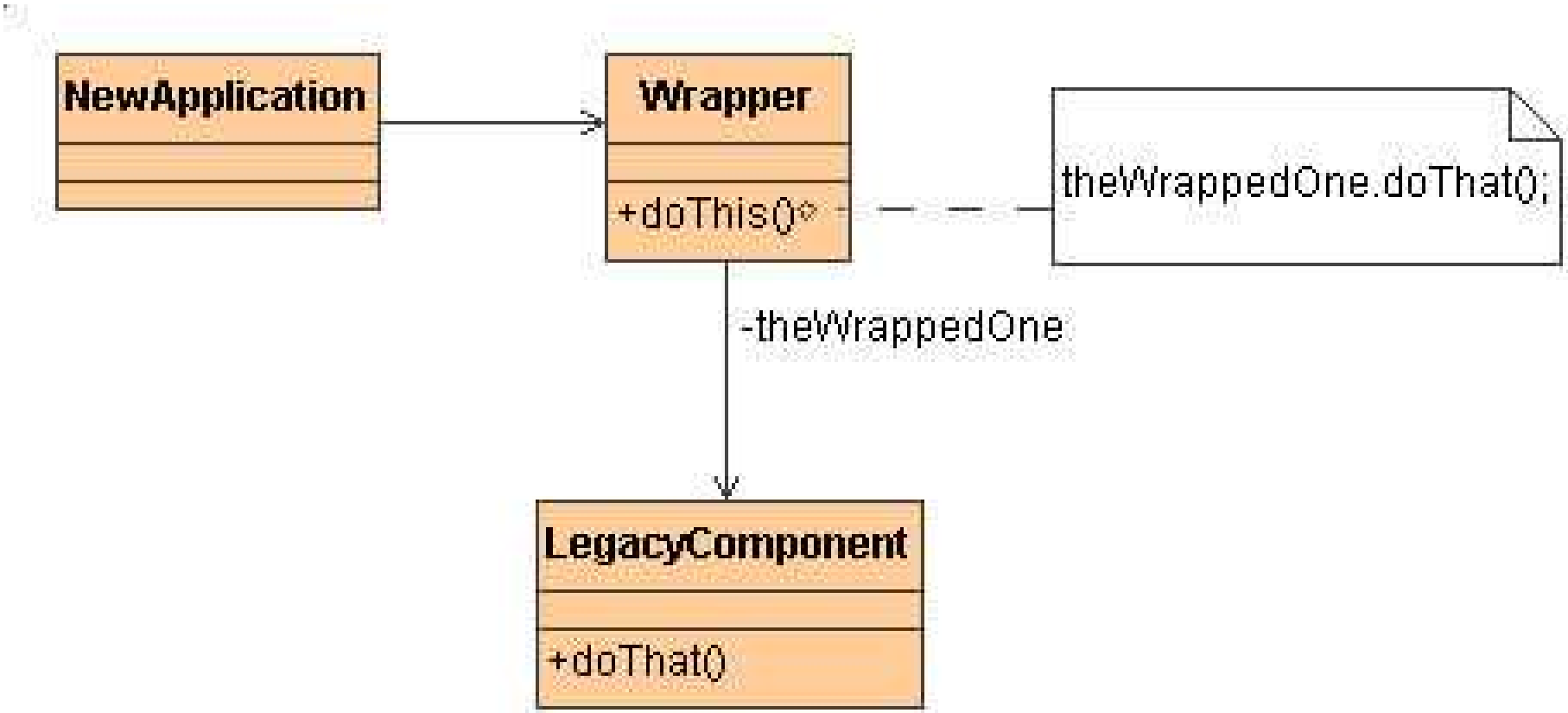
Adapter

Utilisation

- identifier les composants impliqués : les composants qui sont clients et les composants doivent être adapté ;
- identifier les intefaces que les clients désirent ;
- concevoir une classe “wrapper” qui réalise la correspondance entre la nouvelle interface et celle des classes à adapter ;

Adapter

Diagramme UML



Plan

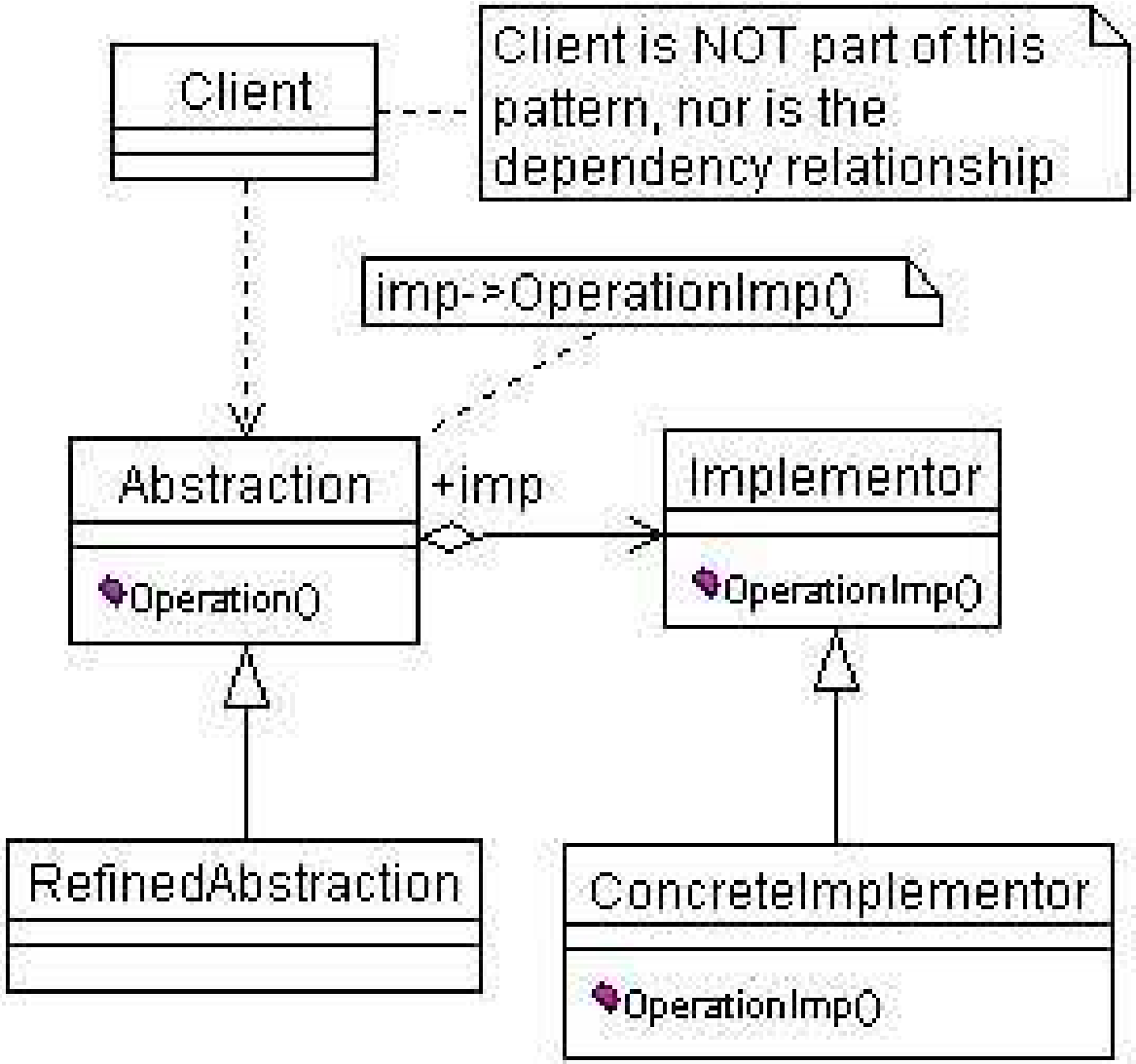
Bridge

Utilisation

- On utilise **Bridge** lorsque :
  - on veut éviter un lien permanent entre l'abstraction et l'implantation (ex: l'implantation est choisie à l'exécution) ;
  - l'abstraction et l'implantation sont toutes les deux susceptibles d'être raffinées ;
  - les modifications subies par l'implantation ou l'abstraction ne doivent pas avoir d'impacts sur le client (pas de recompilation) ;
- concept "Enveloppe/Lettre" de Coplien.

Bridge

Diagramme UML



Bridge

```
class Time {
public:
  Time() { }
  Time( int hr, int min ) { imp_ = new TimeImp( hr, min ); }
  virtual void tell() { imp_->tell(); }
protected:
  TimeImp*  imp_;
};

class TimeImp {
public:
  TimeImp( int hr, int min ) { hr_ = hr;  min_ = min; }
  virtual void tell() { cout << "time␣is␣" << setw(2)
                             << setfill(48) << hr_
                             << min_ << endl; }
protected:
  int hr_, min_;
};
```

Plan

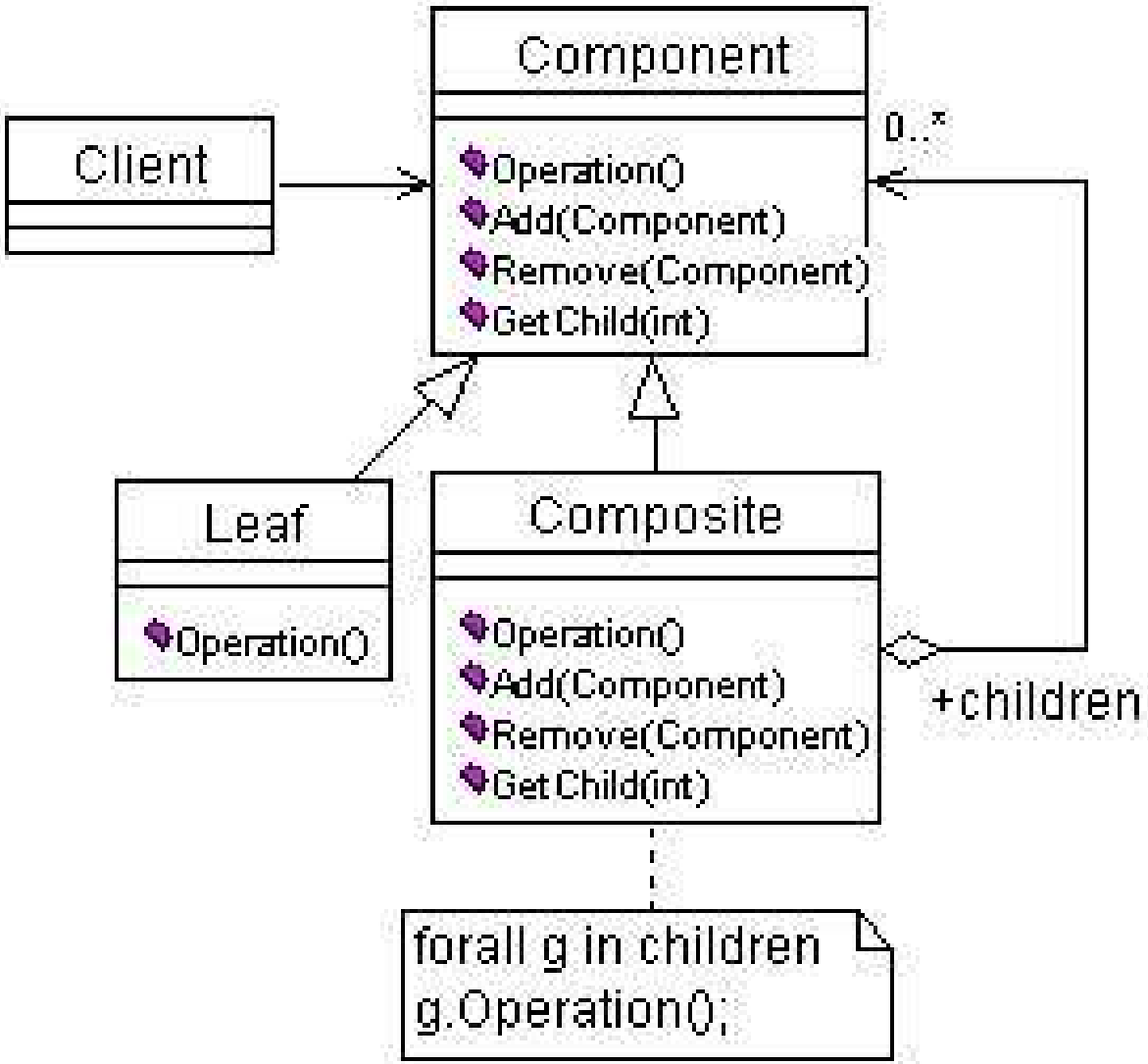
Composite

Utilisation

- On utilise **Composite** lorsque l'on veut :
  - représenter une hiérarchie d'objets ;
  - représenter des compositions récursives ;
- exemple : un système de fichiers, un répertoire est composé de fichiers et de répertoire.

Composite

Diagramme UML



Plan

Introduction

- observer : relation entre des vues et un composant observé ;
- MVC (Model-View-Controler) : idem à observer avec la notion de contrôleur ;
- visitor :

Plan

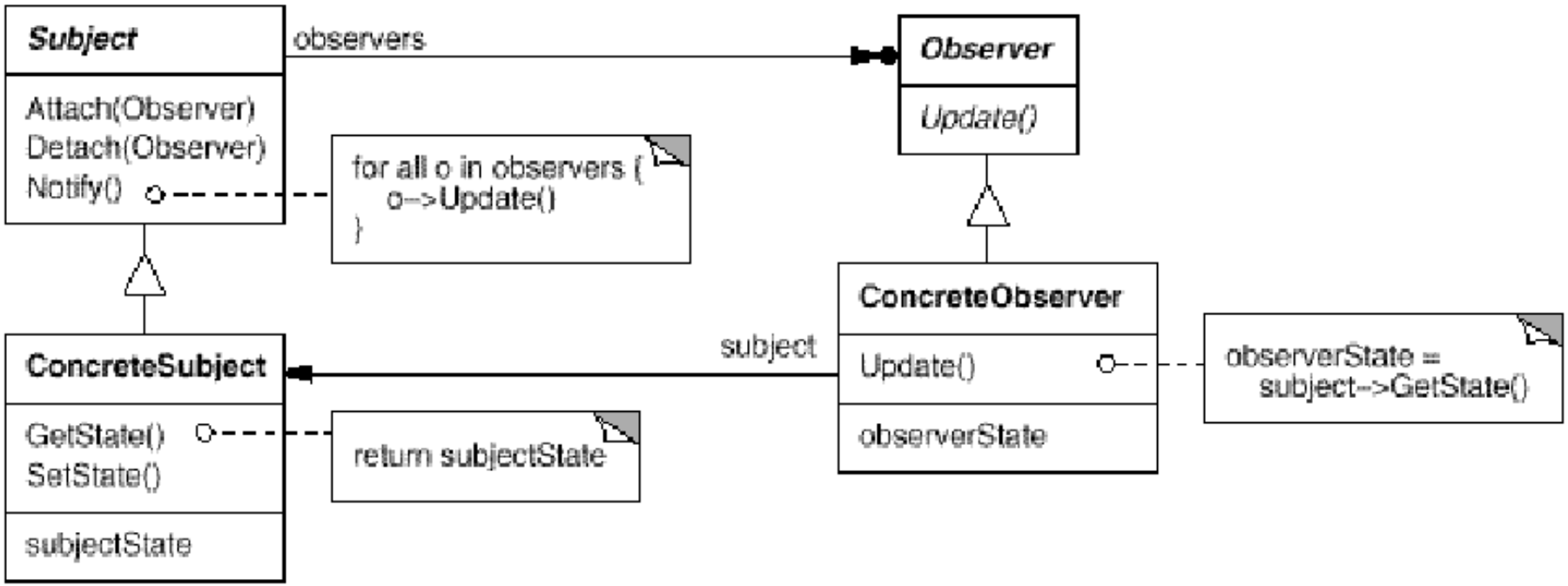
Observer

Utilisation

- On utilise **Observer** lorsque :
  - l'on veut définir des dépendances de type un à plusieurs entre plusieurs composants ;
  - un objet change d'état, tous les composants en dépendant sont avertis et sont mis à jour automatiquement ;
- c'est la partie "Vue" du pattern "Model-View-Controller" ;
- exemple : un document et ses vues.

Observer

Diagramme UML



Observer

Commentaires

- le “sujet” est associé seulement à la classe de base Obverser ;
- le “client” configure le nombre et le type d’observateurs ;
- les observateurs s’enregistrent eux-mêmes auprès du “sujet” ;
- le “sujet” envoie les événements à tous les observateurs enregistrés.

Observer

Traduction C++

Observer

```
class Observer
{
public:
  virtual void update() = 0;
};
class Subject
{
private:
  vector < Observer* > observers;
public:
  void attach(Observer* observer) {
    vector < Observer* >::iterator i;
    bool contained = false;
    for (i = observers.begin(); i != observers.end(); ++i)
      if (*i == observer) contained = true;
      if (!contained) observers.push_back(observer);
  };
...
```

Observer

```
...
  void detach(Observer* observer) {
    vector < Observer* >::iterator i;
    for (i = observers.begin(); i != observers.end(); ++i)
      if (*i == observer) {
        observers.erase(i);
        return;
      }
  };

  void notify() {
    vector< Observer* >::iterator i;
    for(i = observers.begin(); i != observers.end(); ++i)
      (*i)->update();
  };
};
```

Plan

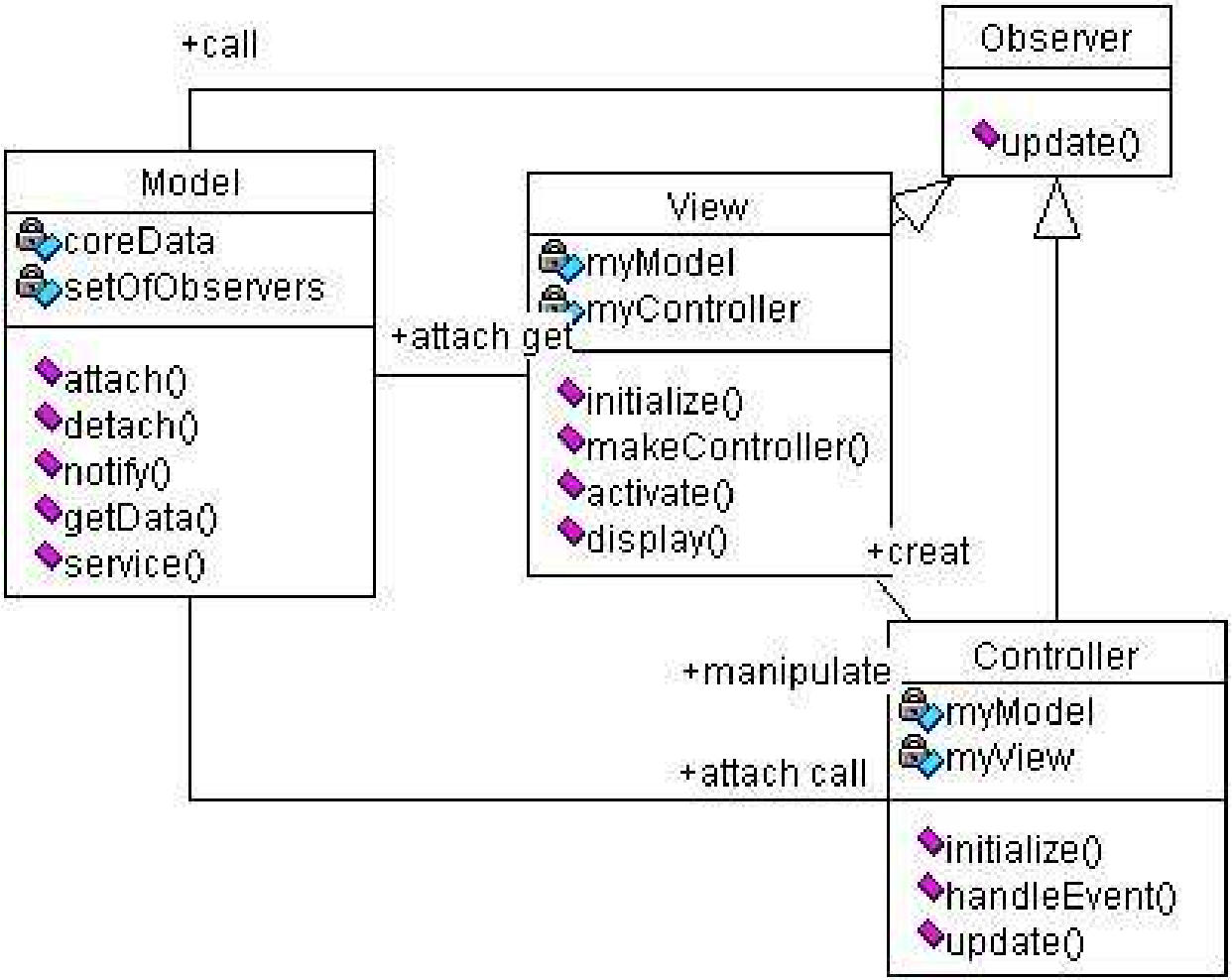
Model-View-Controler

Introduction

- le modèle gère les données ;
- la vue affiche les données ;
- le contrôleur gère la communication et les mises à jour ;
- origine : Smalltalk (Xerox Park, Palo Alto - Milieu des années 70)

Model-View-Controler

Diagramme UML



Model-View-Controler

Le modèle

- le modèle contient la donnée ;
- la mise à jour par update() ;
- le renseignement par getValue().

Model-View-Controler

Traduction C++

Le modèle

```
class Model
{
private:
  View* view;
  Controller* controller;
public:
  Model():view(0),controller(0) {}
  View* getView() { return view; }
  void setView(View* view) { this->view = view; }
  Controller* getController() { return controller; }
  void setController(Controller* controller)
    { this->controller = controller; }
  void update(...) { ... }
  ... getValue() { return ... }
};
```

- la vue affiche les composants et les données ;
- la mise à jour du texte de l'étiquette est faite par update() ;
- pour la notification de modifications, c'est le contrôleur qui écoute !
- la vue se débrouille pour les obtenir.

La vue

```
class View
{
private:
  Controller* controller;
  Model* model;
public:
  View():controller(0),model(0) {}
  Controller* getController() { return controller; }
  void setController(Controller* controller)
    { this->controller = controller; }
  Model* getModel() { return model; }
  void setModel(Model* model) { this->model = model; }
  void update(...) { ... }
  ... getValue() { return ... }
};
```

Model-View-Controler

Le contrôleur

- réveillé par les actions produites dans la vue ;
- récupère des information dans la vue ;
- met à jour le modèle ;
- récupère la nouvelle valeur dans le modèle et la transmet pour affichage à la vue.

Model-View-Controler

Traduction C++

Le contrôleur

```
class Controller
{
private:
  View* view;
  Model* model;
public:
  Controller():view(0),model(0) {}
  View* getView() { return view; }
  void setView(View* view) { this->view = view; }
  Model* getModel() { return model; }
  void setModel(Model* model) { this->model = model; }
  void action()
  {
    model->update(view.getValue())
    view->update(model.getValue())
  }
};
```

Plan

Visitor

Introduction

- représenter une opération an operation to be performed on the elements of an object structure ;
- “visitor” vous permet de définir une nouvelle opération ou une nouvelle classe sans changer les composants où ce nouvel élément doit être utilisé donc sans compilation ;
- do the right thing based on the type of two objects.
- double dispatch

Plan