Programmation Orientée Objet

1 – Modélisation Orienté Objet

P. Berthomé

INSA Centre Val de Loire
Département STI

25 octobre 2017

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Années 90 : Prémisses de Java

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Années 90 : Prémisses de Java

- Initié par SUN

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Années 90 : Prémisses de Java

- Initié par SUN
- 1996, première version

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Années 90 : Prémisses de Java

- Initié par SUN
- 1996, première version
- Évolutions : v1.8 en 2014, v1.9 prévue 2017 (peut-être !)

Rapide historique

Langages de programmation issus de C

Années 70 : définition de C

- Avec UNIX
- Distribué dans le monde universitaire

Années 80 : Première mouture du C++ (B. Stroustrup)

- Extension à la notion de classe, d'héritage
- Standardisation ISO en 1998

Années 90 : Prémisses de Java

- Initié par SUN
- 1996, première version
- Évolutions : v1.8 en 2014, v1.9 prévue 2017 (peut-être !)
- Notion de machine virtuelle

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

C++ et Objective C : Extensions de C, années 80

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

C++ et Objective C : Extensions de C, années 80

Eiffel : tout est objet (années 80, B. Meyer)

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

C++ et Objective C : Extensions de C, années 80

Eiffel : tout est objet (années 80, B. Meyer)

Java : Un des plus populaires actuellement, années 90

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

C++ et Objective C : Extensions de C, années 80

Eiffel : tout est objet (années 80, B. Meyer)

Java : Un des plus populaires actuellement, années 90

C# : Une version MicroSoft avec toutes les dernières évolutions

Quelques langages Orienté Objet

Un petit échantillon

SIMULA : l'un des tout premiers dans les années 60

C++ et Objective C : Extensions de C, années 80

Eiffel : tout est objet (années 80, B. Meyer)

Java : Un des plus populaires actuellement, années 90

C# : Une version MicroSoft avec toutes les dernières évolutions

PHP5, Perl : langages de script

Plan du cours

Organisation

1. Introduction et Modélisation objet

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation
5. Aspects avancés :

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation
5. Aspects avancés :
   - Généricité

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation
5. Aspects avancés :
   - Généricité
   - Les exceptions

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation
5. Aspects avancés :
   - Généricité
   - Les exceptions
6. Un exemple *à la main*

Plan du cours

Organisation

1. Introduction et Modélisation objet
2. Programmation objet en Java
3. Héritage : aspects algorithmiques
4. Héritage : aspects programmation
5. Aspects avancés :
   - Généricité
   - Les exceptions
6. Un exemple *à la main*
7. Un peu de programmation graphique

Programmation logique

Principe

- Les éléments manipulés sont des formules logiques et des entités d'un domaine

Programmation logique

Principe

- Les éléments manipulés sont des formules logiques et des entités d'un domaine
- On cherche à trouver un ensemble qui vérifie l'ensemble des prédicats

Programmation logique

Principe

- Les éléments manipulés sont des formules logiques et des entités d'un domaine
- On cherche à trouver un ensemble qui vérifie l'ensemble des prédicats
- Exemple type : PROLOG

Programmation logique

Principe

- Les éléments manipulés sont des formules logiques et des entités d'un domaine
- On cherche à trouver un ensemble qui vérifie l'ensemble des prédicats
- Exemple type : PROLOG

Programmation logique

Principe

- Les éléments manipulés sont des formules logiques et des entités d'un domaine
- On cherche à trouver un ensemble qui vérifie l'ensemble des prédicats
- Exemple type : PROLOG

```
homme(roger). homme(gerard). femme(gisele).
pere(gerard, gisele).
pere(gerard, roger).
enfant(X, Y) :- pere(Y, X).
fils(X, Y) :- enfant(X, Y), homme(X).
fille(X, Y) :- enfant(X, Y), femme(X).
```

Programmation fonctionnelle

Principe

- L'élément de base est une fonction

Programmation fonctionnelle

Principe

- L'élément de base est une fonction
- Pour faire une exécution, on applique la fonction à d'autres objets

Programmation fonctionnelle

Principe

- L'élément de base est une fonction
- Pour faire une exécution, on applique la fonction à d'autres objets
- exemple : OCaml

Programmation fonctionnelle

Principe

- L'élément de base est une fonction
- Pour faire une exécution, on applique la fonction à d'autres objets
- exemple : OCaml

Programmation fonctionnelle

Principe

- L'élément de base est une fonction
- Pour faire une exécution, on applique la fonction à d'autres objets
- exemple : OCaml

```
let rec reverse l =
   match l with
        [] -> []
 | e::l' -> reverse l' @ [e]
 ;;
(* val reverse : 'a list -> 'a list = <fun> *)
reverse [5; 4; 3; 2; 1];;
```

Programmation objet

Principe

- Les entités manipulées sont des objets sur lesquels ont applique des méthodes

Programmation objet

Principe

- Les entités manipulées sont des objets sur lesquels ont applique des méthodes
- L'encapsulation permet de dissocier la mise en œuvre de de la spécification

Programmation objet

Principe

- Les entités manipulées sont des objets sur lesquels ont applique des méthodes
- L'encapsulation permet de dissocier la mise en œuvre de de la spécification
- Les différents objets communiquent par des *messages*

Programmation objet

Principe

- Les entités manipulées sont des objets sur lesquels ont applique des méthodes
- L'encapsulation permet de dissocier la mise en œuvre de de la spécification
- Les différents objets communiquent par des *messages*
- Notion d'héritage

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)
  - l'horloge

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)
  - l'horloge
  - le programme lui-même

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)
  - l'horloge
  - le programme lui-même
  - ...

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)
  - l'horloge
  - le programme lui-même
  - ...
- Il réagit (réflexe) à ceux qu'il a reconnus et exécute alors le code spécifique

Programmation événementielle

Principe

- Le programme est en attente continuelle d'événements générés par
  - les périphériques (souris, clavier)
  - l'horloge
  - le programme lui-même
  - ...
- Il réagit (réflexe) à ceux qu'il a reconnus et exécute alors le code spécifique
- Surcouches de langages existants : Java, VB, C#, ...

Programmation parallèle

Principe

- Utiliser au mieux la puissance de calcul distribuée sur

Programmation parallèle

Principe

- Utiliser au mieux la puissance de calcul distribuée sur
  - un réseau

Programmation parallèle

Principe

- Utiliser au mieux la puissance de calcul distribuée sur
  - un réseau
  - processeurs multicœurs

Programmation parallèle

Principe

- Utiliser au mieux la puissance de calcul distribuée sur
  - un réseau
  - processeurs multicœurs
- Nécessité de synchroniser et envoyer des messages entre les différents processeurs (ou processus)

Programmation parallèle

Principe

- Utiliser au mieux la puissance de calcul distribuée sur
  - un réseau
  - processeurs multicœurs
- Nécessité de synchroniser et envoyer des messages entre les différents processeurs (ou processus)
- Mise en œuvre par des bibliothèques spécifiques au dessus de langages existants : MPI

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste

La programmation structurée

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue

La programmation structurée

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue
- Réutilisabilité : briques de logiciels

La programmation structurée

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue
- Réutilisabilité : briques de logiciels
- Compatibilité : combiner des logiciels

La programmation structurée

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue
- Réutilisabilité : briques de logiciels
- Compatibilité : combiner des logiciels

La programmation structurée

- La programmation structurée ne permet pas de gérer l'évolutivité

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue
- Réutilisabilité : briques de logiciels
- Compatibilité : combiner des logiciels

La programmation structurée

- La programmation structurée ne permet pas de gérer l'évolutivité
- Les structures de données sont difficiles à utiliser à travers les fonctions

Motivation de la POO

Qualité logicielle : souci de l'ingénieur

- Fiabilité : comportement valide et robuste
- Extensibilité : le programme évolue
- Réutilisabilité : briques de logiciels
- Compatibilité : combiner des logiciels

La programmation structurée

- La programmation structurée ne permet pas de gérer l'évolutivité
- Les structures de données sont difficiles à utiliser à travers les fonctions
- La réutilisabilité n'est pas aisée

Cas d'école

Exercice 1 [Une file particulière : file avec abandon] On souhaite modéliser le comportement réel d'une file d'attente (par exemple au RU) où certaines personnes sortent de la file d'attente sans avoir reçu le service pour lequel ils attendaient (et donc partent chercher un repas ailleurs). Pour cela, on adapte la structure de données File en rajoutant une procédure nouvelle Abandon qui :

- prend en paramètre la File et l'élément à éliminer ;
- élimine toutes les occurrences de l'élément dans la File.

Question : écrire la fonction `Abandon`

Structure de données

Structure de données File :

type Queue : **struct** { array values[CONST_N] : element_type,
variable beg : **int**,
variable end : **int**,
constant lenght <– CONST_N : **int**}

fonctions :

- enqueue(Queue IN/OUT q, el IN element_type)
- dequeue(Queue IN/OUT q) : element_type
- isEmpty(Queue IN q) : boolean

Est-ce un bon algorithme ?

Procédure Abandon(q, el)

Parameters: *q (IN-OUT) : Queue ; el : Element_type OUT*
Variables: *i : int*

```
1 for i from q.beg to q.end do
2     if q.values[i] = el then
3         dequeue(q);
4         q.beg = q.beg +1 ;
```

Modèle objet

Notion d'objet

- Entité du monde "réel"

Langage orienté objet

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini

Langage orienté objet

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit

Langage orienté objet

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

- L'abstraction : représenter un concept réel

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

- L'abstraction : représenter un concept réel
- L'encapsulation : occulter l'implémentation

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

- L'abstraction : représenter un concept réel
- L'encapsulation : occulter l'implémentation
- La modularité : faible couplage des modules

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

- L'abstraction : représenter un concept réel
- L'encapsulation : occulter l'implémentation
- La modularité : faible couplage des modules
- La hiérarchie : classement des abstractions

Modèle objet

Notion d'objet

- Entité du monde "réel"
- Comportement défini
- Réagit au changement, interagit
- Forme de donnée stable

Langage orienté objet

- L'abstraction : représenter un concept réel
- L'encapsulation : occulter l'implémentation
- La modularité : faible couplage des modules
- La hiérarchie : classement des abstractions
- Le polymorphisme : prendre plusieurs formes

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel

Un objet

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique

Un objet

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique
- Comportant :

Un objet

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique
- Comportant :
  - Des attributs : la *composition* des éléments

Un objet

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique
- Comportant :
  - Des attributs : la *composition* des éléments
  - Des méthodes : comportement vis à vis de sollicitations extérieures

Un objet

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique
- Comportant :
  - Des attributs : la *composition* des éléments
  - Des méthodes : comportement vis à vis de sollicitations extérieures

Un objet

- Instanciation de la classe

Concept de classe et d'objet

Notion de classe

- Modélisaton générale d'entités du monde réel
- Définition d'un type abstrait spécifique
- Comportant :
  - Des attributs : la *composition* des éléments
  - Des méthodes : comportement vis à vis de sollicitations extérieures

Un objet

- Instanciation de la classe
- Communication entre les objets par messages

Exemple : voiture

Caractéristiques diverses

- Couleur, type, marque
- En marche, arrêt, vitesse

Exemple : voiture

Caractéristiques diverses

- Couleur, type, marque
- En marche, arrêt, vitesse

Comportement peut modifier l'état

- démarrer : état de marche
- changer de vitesse : vitesse
- Aller à une certaine vitesse

Diagramme de classe UML

Diagramme de classe UML

- **Propriétés** : ensemble des caractéristiques

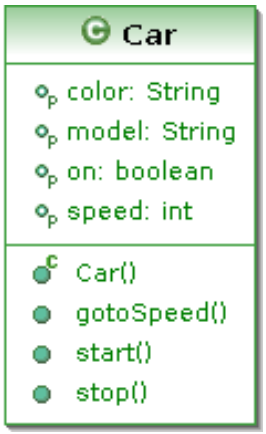


Diagramme de classe UML

Diagramme de classe UML

- **Propriétés** : ensemble des caractéristiques
- **Méthodes** : comportements

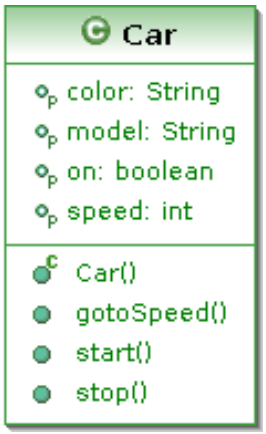
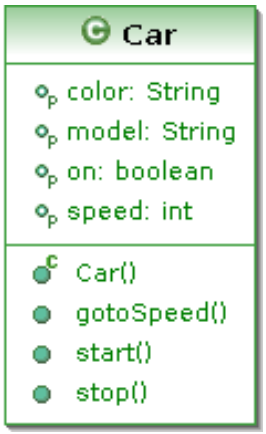


Diagramme de classe UML

Diagramme de classe UML

- **Propriétés** : ensemble des caractéristiques
- **Méthodes** : comportements
- **Constructeur** : Comportement particulier



Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère

Attribut

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé

Attribut

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de

Attribut

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de
  - accesseur : `getPropriété`

Attribut

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de
  - accesseur : `getPropriété`
  - modifieur : `setPropriété`

Attribut

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de
  - accesseur : `getPropriété`
  - modifieur : `setPropriété`

Attribut

- Un constituant de l'objet

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de
  - accesseur : `getPropriété`
  - modifieur : `setPropriété`

Attribut

- Un constituant de l'objet
- Information non disponible à l'utilisateur

Propriété vs. Attribut

Propriété

- Une information sur l'objet que l'on considère
- Peut éventuellement être calculé
- Accessible éventuellement au travers de
  - accesseur : `getPropriété`
  - modifieur : `setPropriété`

Attribut

- Un constituant de l'objet
- Information non disponible à l'utilisateur
- Créer des méthodes spécifiques

Utilisation

Petit programme pour petite voiture

1. Création d'une voiture :

```
Car myCar
myCar =   new Car("4L", "Blanc")
```

Utilisation

Petit programme pour petite voiture

1. Création d'une voiture :

```
Car myCar
myCar =   new Car("4L", "Blanc")
```

2. Vérification de l'état de la voiture

```
Print(myCar.isOn())
Print(myCar.getSpeed())
```

Utilisation

Petit programme pour petite voiture

1. Création d'une voiture :

```
Car myCar
myCar =   new Car("4L", "Blanc")
```

2. Vérification de l'état de la voiture

```
Print(myCar.isOn())
Print(myCar.getSpeed())
```

3. La démarrer et la faire partir

```
myCar.start()
myCar.gotoSpeed(90,30)
```

Ajout d'une nouvelle classe

Un feu tricolore

- Propriété :
  - couleur

Ajout d'une nouvelle classe

Un feu tricolore

- Propriété :
  - couleur
- Méthodes
  - change
  - clignote

Ajout d'une nouvelle classe

Un feu tricolore

- Propriété :
  - couleur
- Méthodes
  - change
  - clignote

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures

Nouveaux éléments

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête

Nouveaux éléments

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête
- Si le feu est orange, la voiture . . . ralentit

Nouveaux éléments

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête
- Si le feu est orange, la voiture . . . ralentit
- Si le feu est vert, la voiture passe

Nouveaux éléments

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête
- Si le feu est orange, la voiture . . . ralentit
- Si le feu est vert, la voiture passe
- Quand le feu passe au vert, les voitures arrêtées passent

Nouveaux éléments

Interaction entre classes

Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête
- Si le feu est orange, la voiture . . . ralentit
- Si le feu est vert, la voiture passe
- Quand le feu passe au vert, les voitures arrêtées passent

Nouveaux éléments

- Une liste de voitures

Interaction entre classes

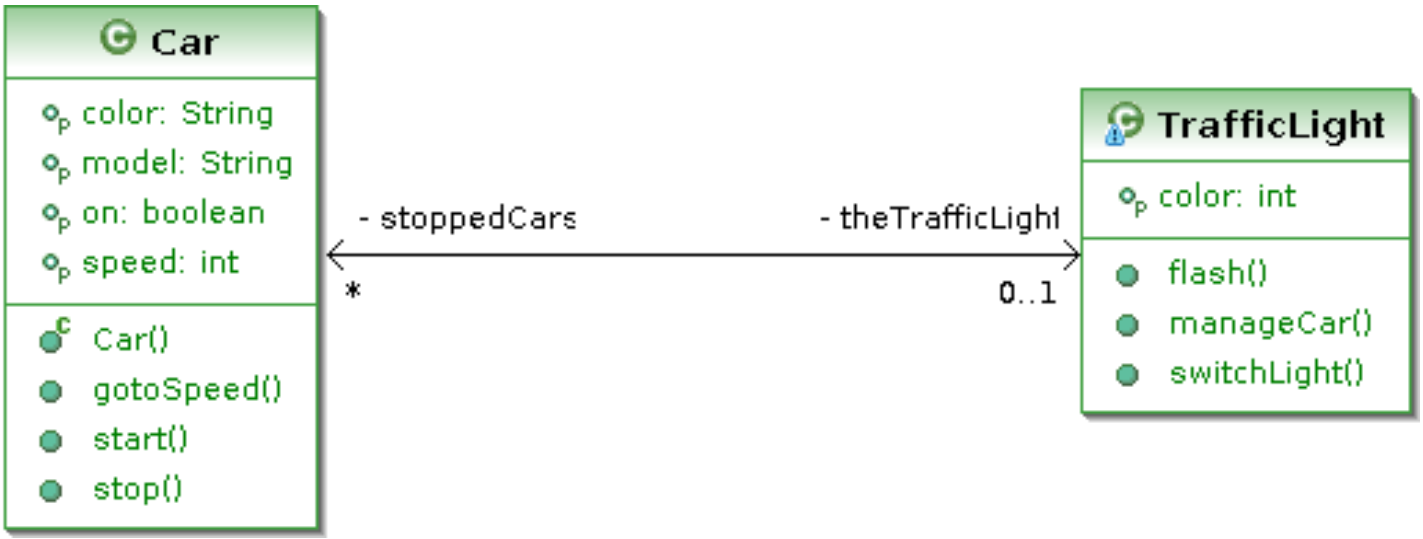
Nouvelle Fonctionalité

- Le feu tricolore voit passer des voitures
- Si le feu est rouge, la voiture s'arrête
- Si le feu est orange, la voiture . . . ralentit
- Si le feu est vert, la voiture passe
- Quand le feu passe au vert, les voitures arrêtées passent

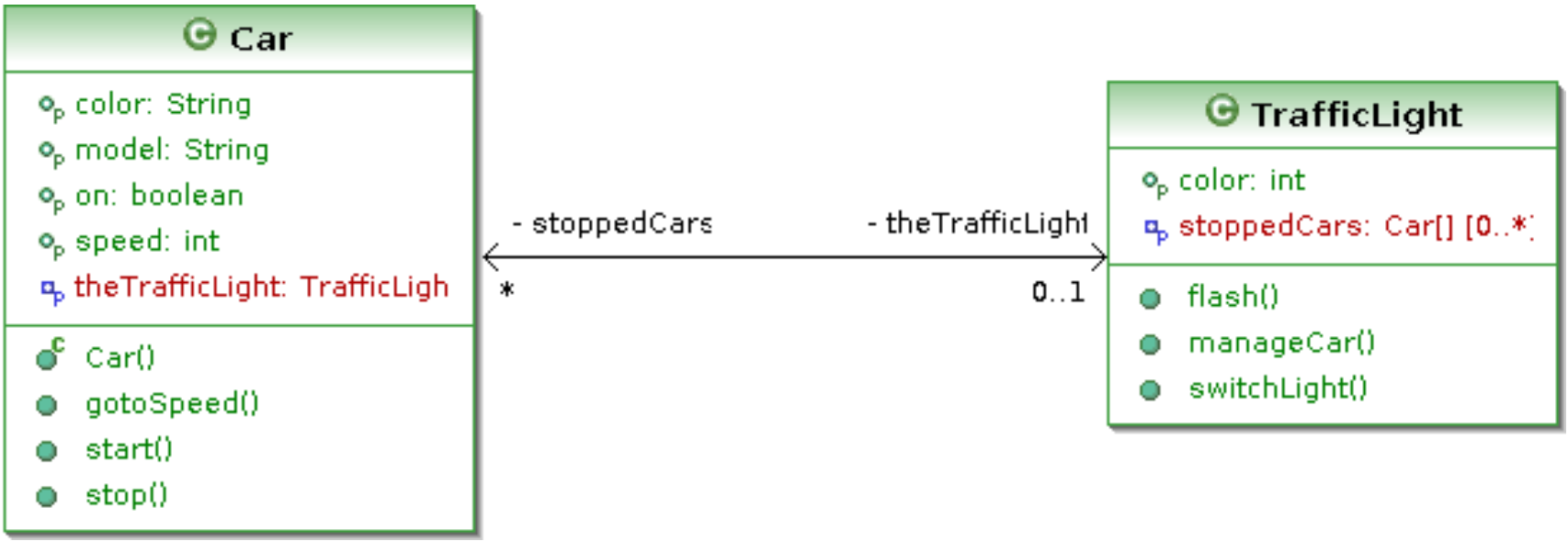
Nouveaux éléments

- Une liste de voitures
- Une méthode pour faire arriver les voitures

Nouveau Schéma



Nouveau Schéma



Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Dépendance

- Lien faible entre les classes

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Dépendance

- Lien faible entre les classes
- qui disparaît après l'appel de la méthode

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Dépendance

- Lien faible entre les classes
- qui disparaît après l'appel de la méthode
  - cas de paramètres

Différents types de liens inter-classes

Soient deux classes O1 et O2.

Association

- O1 possède un attribut de type O2
- Les messages sont envoyés de la classe O1 à la classe O2
- "je demande à la classe O2 de faire une action"
- lien fort entre les deux classes

Dépendance

- Lien faible entre les classes
- qui disparaît après l'appel de la méthode
  - cas de paramètres
  - cas de variables locales

Application au feu tricolore

Association

- Il existe une liste de voiture

Application au feu tricolore

Association

- Il existe une liste de voiture
- la méthode `switchLight()` quand elle positionne à vert le feu fait partir les voitures arrêtées

Application au feu tricolore

Association

- Il existe une liste de voiture
- la méthode `switchLight()` quand elle positionne à vert le feu fait partir les voitures arrêtées
- elle envoie un message à la classe `Car`

Application au feu tricolore

Association

- Il existe une liste de voiture
- la méthode `switchLight()` quand elle positionne à vert le feu fait partir les voitures arrêtées
- elle envoie un message à la classe `Car`
- Association de `TrafficLight` vers `Car`

Application au feu tricolore

Association

- Il existe une liste de voiture
- la méthode `switchLight()` quand elle positionne à vert le feu fait partir les voitures arrêtées
- elle envoie un message à la classe `Car`
- Association de `TrafficLight` vers `Car`

Application au feu tricolore

Association

- Il existe une liste de voiture
- la méthode `switchLight()` quand elle positionne à vert le feu fait partir les voitures arrêtées
- elle envoie un message à la classe `Car`
- Association de `TrafficLight` vers `Car`

```
TrafficLight::switchLight(){
color = (color + 1) mod 3
if color = 2 // Vert
        // On vide la liste des voitures
        for i = 0 to NbCars
                stoppedCars[i].gotoSpeed(50, 10)
                nbCars = 0}
```

Application au feu tricolore

Dépendance

- La méthode `manageCar` utilise en paramètre une voiture

Application au feu tricolore

Dépendance

- La méthode `manageCar` utilise en paramètre une voiture

Application au feu tricolore

Dépendance

- La méthode `manageCar` utilise en paramètre une voiture

```
TafficLight::manageCar(Car oneCar){
if color = 0 // Orange
        oneCar.gotoSpeed(oneCar.getSpeed()/2, 3)
if color = 1 // Rouge
        // On ajoute la voiture a la liste de Voiture a la suite des
        // autres
      stoppedCars[nbCars +1] = nbCars
      nbCars = nbCars + 1
}
```

Autres liens possibles

Auto-Association

- Quand une méthode appelle une autre de la même classe

Autres liens possibles

Auto-Association

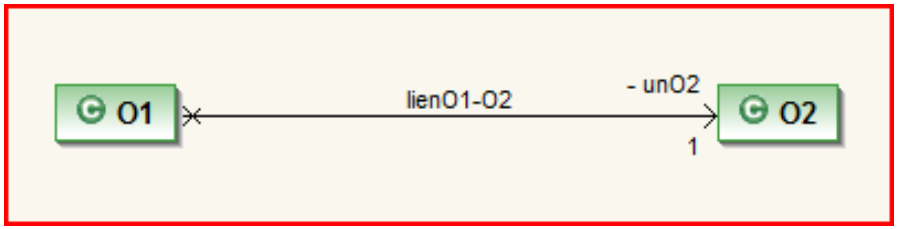
- Quand une méthode appelle une autre de la même classe
  - sur le même objet :
    La méthode `gotoSpeed()` fera sans doute appel à la méthode `changeSpeed()`

Autres liens possibles

Auto-Association

- Quand une méthode appelle une autre de la même classe
  - sur le même objet :
    La méthode `gotoSpeed()` fera sans doute appel à la méthode `changeSpeed()`
  - sur un objet de même type :
    Si on s'intéresse à la voiture placée devant

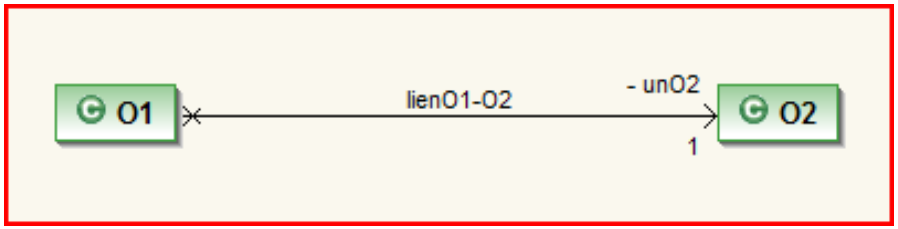
Cardinalité d'une relation



Composition

- La classe `O1` possède un attribut `unO2` de type `O2`

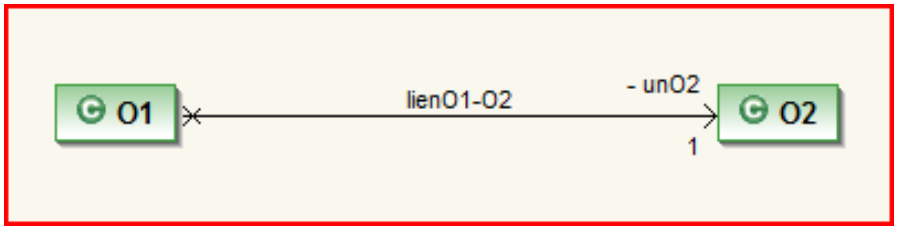
Cardinalité d'une relation



Composition

- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

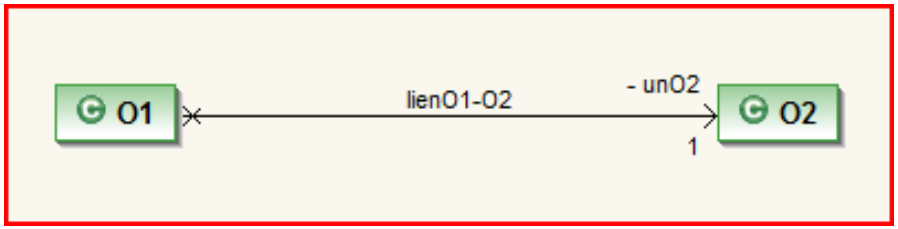
Cardinalité d'une relation



Composition

- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

Cardinalité d'une relation



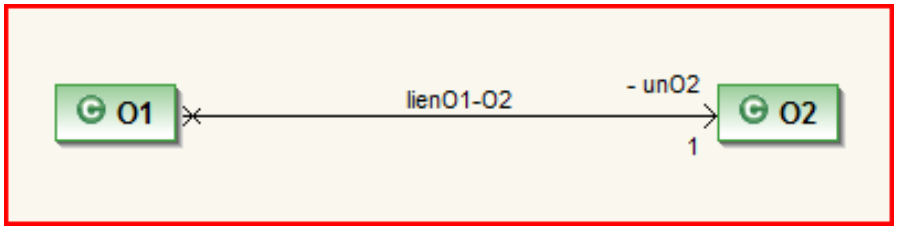
Composition

- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

Cardinalité

1 : l'attribut doit être créé

Cardinalité d'une relation



Composition

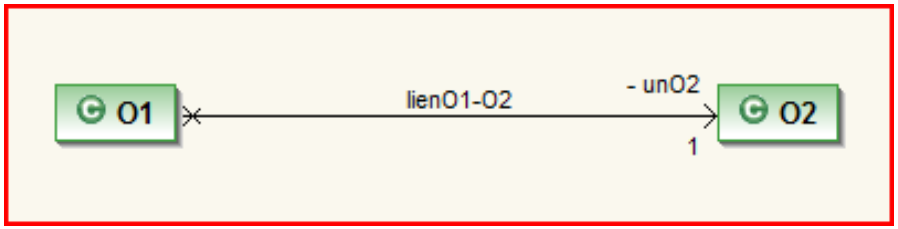
- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

Cardinalité

1 : l'attribut doit être créé

0..1 : il existe au plus un élément de ce type

Cardinalité d'une relation



Composition

- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

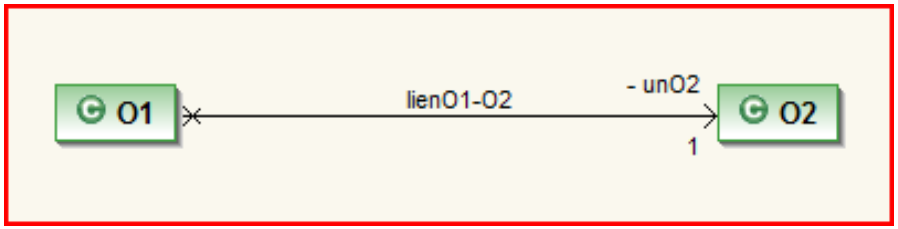
Cardinalité

1 : l'attribut doit être créé

0..1 : il existe au plus un élément de ce type

* : il existe *des* éléments de ce type

Cardinalité d'une relation



Composition

- La classe `O1` possède un attribut `unO2` de type `O2`
- Les deux éléments sont liés

Cardinalité

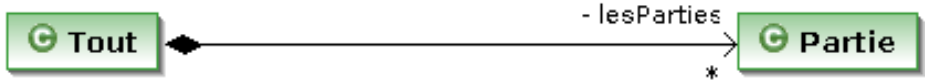
1 : l'attribut doit être créé

0..1 : il existe au plus un élément de ce type

* : il existe *des* éléments de ce type

1..* : il existe au moins un élément de ce type

Différents types d'associations



Composition

- Chaque **Tout** peut être considéré comme un regroupement de **Parties**
- Chaque **Partie** ne peut appartenir qu'à un seul **Tout**
- Les **Parties** doivent être détruites quand le **Tout** l'est (sauf si on les transfère à un autre **Tout**)

Exemple

Une voiture contient un moteur ; les deux seront détruits si on détruit la voiture

Différents types d'associations (bis)

Agrégation

- Chaque **Groupe** peut être considéré comme un regroupement de **Partie**s
- Chaque **Partie** peut appartenir à plusieurs **Groupe**s
- Les **Partie**s ne doivent pas être détruites quand le **Groupe** l'est

Exemple

Des musiciens peuvent appartenir à plusieurs groupes de musique et ne sont généralement pas détruits à la dissolution du groupe.