UML :

Modèle : Un modèle est une simplification de la réalité.

UML : Langage modélisation visuelle, pour spécifier, visualiser, construire et documenter les produits d'un développement logiciel.

Liste diagrammes :

- **Statique (x6 diagrammes structurel) :**
  - Diagramme de classe (**+** (public), **–** (private), **#** (protected) et **~** (paquetage)) = desc statique des classes
  - Diagramme d'objet = instance diagramme de classe
  - Diagramme de paquetage (UML 2)
  - Diagramme de structure composite (UML 2)
  - Diagramme de composant (Vue organisationnelle)
  - Diagramme de déploiement (Vue de déploiement)
- **Dynamique (x7 diagrammes comportementaux):**
  - Diagramme de vue d'ensemble des interactions
  - Diagramme de séquence = interact entre objets dans un scénario (niveau temporel)
  - Diagramme de communication
  - Diagramme de temps
  - Diagramme de machine à états = desc du comportement d'une classe en différents états, transitions
  - Diagramme d'activité = décrire les activités d'un processus
- **Autre :**
  - Diagramme de cas d'utilisation (vue fonctionnelle, interact user/sys)

Relations :

- **Agrégation** : cas particulier de relation d'association. C'est une association déséquilibrée, où une classe joue un rôle prépondérant. Elle correspond généralement à une relation tout ou parties (composé/composant).
- **Composition** : cas particulier de la relation d'agrégation avec une sémantique plus forte : Les durées de vie du composé et de ses composants sont liées.

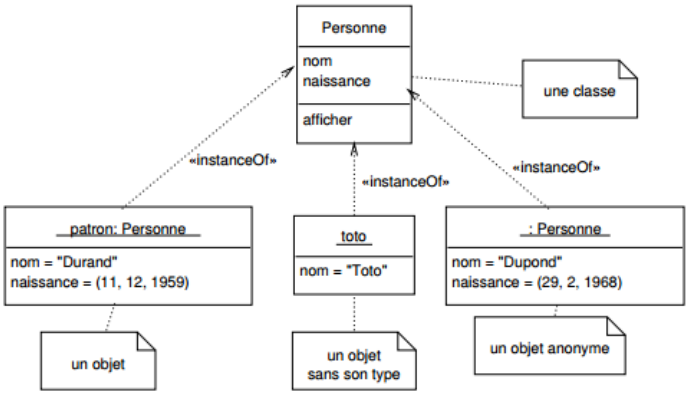


Figure 1 : Ex de diag objet

Utilité OCL :

- les invariants d'une classe ou d'un type ;
- les pré- et post-conditions d'opération ;
- les contraintes au sein d'une opération ;
- les expressions de navigation : contraintes pour représenter les chemins au sein de la structure de classes.

Diagramme de déploiement :

- la disposition physique des différents matériels (les nœuds = ressource liée à l'exécution (ordinateur) qui font partie du système
- la répartition des artefacts (=unité d'implémentation physique (fichier,…) qui « vivent » sur ces matériels.

USE CASE

Différents types d'acteurs :

- les **acteurs principaux** utilisent les fonctions principales (les clients) ;
- les **acteurs secondaires** effectuent des tâches administratives ou de maintenance (recharger la caisse) ;
- le **matériel externe** : dispositifs matériels qui font partie du domaine de l'application (l'imprimante) ;
- les **autres systèmes** : avec lesquels le système interagit (système du groupement bancaire)

Ex use case :

Titre : Retrait d'espèces

But : un client réalise un retrait d'espèce sur le compte associé à sa carte.
Acteurs : porteur carte (principal), SI GB (secondaire), imprimante (mat.)
Début : Insertion d'une carte dans un GAB en état de fonctionnement.
Enchaînements : Une fois la carte insérée, le client entre son code, puis le montant du retrait. Après identification correcte de la carte et autorisation de l'opération par le SI du groupement bancaire, le ticket et la carte sont restitués. Une fois la carte récupérée, les billets sont distribués.
Fin : La carte et l'argent ont été récupérés.
Alternatif : Le client peut demander à ne pas avoir de ticket. En cas de code erroné, le code est redemandé au client.
Exceptions : Le retrait n'est effectif que si le code est correct. Le 3ecode erroné provoque la capture de la carte par le GAB. Le client peut annuler le retrait.

Relations :

- **relation de généralisation** : un cas d'utilisation est une spécialisation d'un autre (qui peut être abstrait) ;
- **relation d'inclusion «include»** : le cas d'utilisation source comprend également le cas d'utilisation destination.
- **relation d'extension «extend»** : le cas d'extension source ajoute son comportement au cas d'utilisation destination (point d'extension). L'extension peut être soumise à condition.

Intérêt :

- capturer les besoins ;
- délimiter la frontière du système ;
- définir les relations entre le système et l'environnement ;
- permettre de dialoguer avec les clients (description textuelle)

Scénarios → Instance d'un use case

MACHINE A ÉTATS

Obj : décrire le comportement d'une classe (ou d'un sous-système), en particulier :

- les différents états possibles de ses objets,
- les transitions possibles entre ces états et les événements déclenchant
- les séquencements possibles de ses opérations.

Constituants :

- les événements (stimuli externes ou internes) ;
- les états (valeurs des objets) ;
- les changements d'états (transitions) ;
- les opérations (actions ou activités).

Actions associées à un état :

- **actions d'entrée** : exécution (instantanée) de l'action lors de l'entrée dans l'état.
- **action interne** : exécution d'une action sur événement sans changer d'état
- **actions de sortie** : exécution de l'action au moment de quitter l'état.

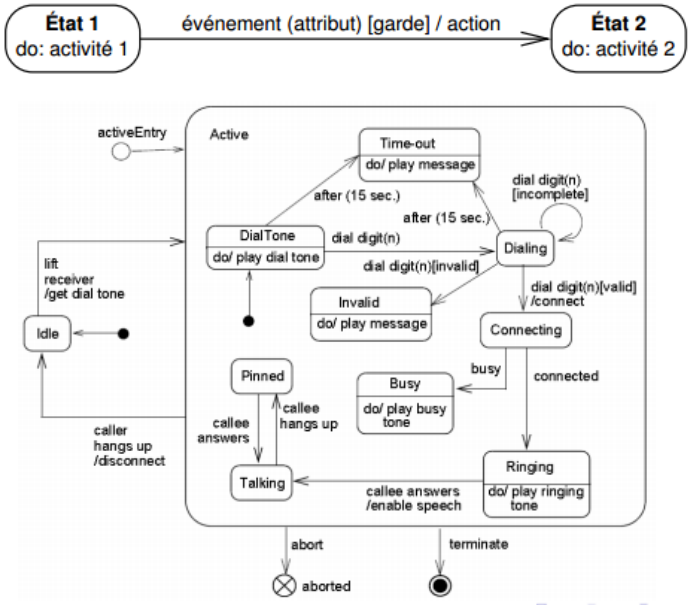


Figure 2: Ex de diagramme machine à état

INTROSPECTION

Introspection : Interroger dynamiquement les objets d'une application pour retrouver la structure de l'application :

- les classes,
- les attributs,
- les méthodes,
- les constructeurs

Intercession : Modifier l'état d'une application en s'appuyant sur les informations obtenues par introspection.

Récupérer les méthodes :

- **getMethods()** : toutes les méthodes publiques de la classe (y compris héritées)
- **getDeclaredMethods()** : toutes les méthodes déclarées dans la classe (quelque soit le droit d'accès). Donc pas les méthodes héritées.
- **getMethod(« nom », type.class, type.class)** : Récupère la méthode avec le « nom » et les types de param fournis

Utile pour :

- **évolutives dynamiquement** (prendre en compte de nouvelles classes non connues au moment de l'écriture d'une application) ;
- **adaptables dynamiquement** (modifier l'application en fonction d'information découverte à l'exécution)

PATRONS DE CONCEPTION

Pourquoi ?

- Construire des systèmes plus extensibles, plus robustes au changement
- Capitaliser l'expérience collective des informaticiens
- Réutiliser les solutions qui ont fait leur preuve Identifier les avantages/inconvénients/limites de ces solutions
- Savoir quand les appliquer

Def : Un patron de conception (design pattern) décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème de conception dans un contexte particulier.

Éléments essentiels :

- **nom** : un ou deux mots pour décrire le problème de conception considérée, ses solutions et leurs conséquences.
- **problème** : situation où le problème s'applique
- **solution** : éléments de la conception, leurs relations et collaborations. F la solution n'est pas forcément précise : idée d'architecture. F plusieurs variantes peuvent être possibles.
- **conséquences** : effets résultants et compromis induits F Les conséquences peuvent être positives ou négatives (arbitrage).

Description :

- **Nom** : nom de référence du patron I étend le vocabulaire du concepteur
- **Intention** : courte description de :
  - ce que fait le patron de conception ;
  - sa raison d'être ou son but ;
  - cas ou problème particulier de conception concerné.
- **Alias** : Autres noms connus pour le patron
- **Motivation** :
  - scénario qui illustre un cas de conception
  - montre l'architecture en classes et objets de la solution
  - aide à comprendre les descriptions plus abstraites du modèle
- **Indications d'utilisation** :
  - Quels sont les cas qui justifient l'utilisation du patron ?
  - Quelles situations de conception peuvent tirer avantage du patron ?
  - Comment reconnaître ces situations ?
- **Structure** : description de la solution sous forme de :
  - un diagramme de classe pour l'architecture ;
  - des diagrammes d'interaction pour la dynamique.
- **Constituants** : classes/objets de la solution avec leurs responsabilités
- **Collaborations** entre les constituants pour assumer leurs responsabilités
- **Conséquences** :
  - compromis induits par l'utilisation du patron

- impacts sur l'architecture de conception
- gains en terme de diminution du couplage dans la solution

- **Implantation** : Solutions types, techniques, pièges et astuces.
- **Exemples de code** : extraits de code illustrant la mise en œuvre du patron
- **Utilisations remarquables** : exemples issus de systèmes existants
- **Patrons apparentés** :
  - patrons similaires et différences essentielles
  - utilisation conjointe avec d'autres patrons

Classification des patrons :

- **fondamental** : application directe des concepts objets ;-)
- **créateur** : processus de création d'objets
- **structurel** : architecture statique du système
- **comportemental** : interactions entre objets et répartition des responsabilités

---

META MODELE

Conformité : Un modèle est **conforme** à un méta-modèle si :

- tous les éléments du modèle sont instance du méta-modèle ;
- et les contraintes exprimées sur le méta-modèle sont respectées

Propriétés transformation M2M :

- **Transformations** :
  - **endogènes** : mêmes méta-modèle source et cible,
  - **exogènes** : méta-modèles source et cible différents
- Transformations **unidirectionnelles** ou **bidirectionnelles**
- **Traçabilité** : garder un lien (une trace) entre les éléments cibles et les éléments sources correspondants.
- **Incrémentalité** : une modification du modèle source sera répercutée immédiatement sur le modèle cible.
- **Réutilisabilité** : mécanismes de structuration, capitalisation et réutilisation de transformation.

CORRECTION 2015

1.2 Réprésenter association UML en ecore ?

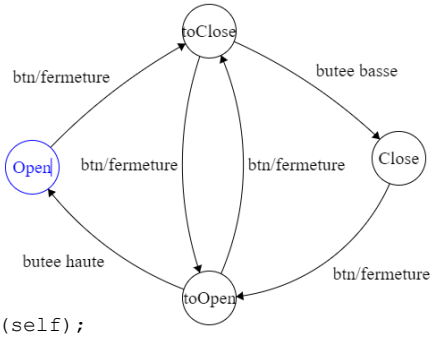
```
property processElements : ProcessElement[*|1] { ordered composes };
```

property <name> : <type> <[card|card]> { ordered [composes] }

1.3 Sirius ?

Cette technologie permet de concevoir un atelier de modélisation graphique sur-mesure en s'appuyant sur les technologies Eclipse Modeling, en particulier EMF et GMF. L'atelier de modélisation créé est composé d'un ensemble d'éditeurs Eclipse (diagrammes, tables et arbres) qui permettent aux utilisateurs de créer, éditer et visualiser des modèles EMF.

2. Machine à état



3. OCL

1 - Noms des blocs uniques :

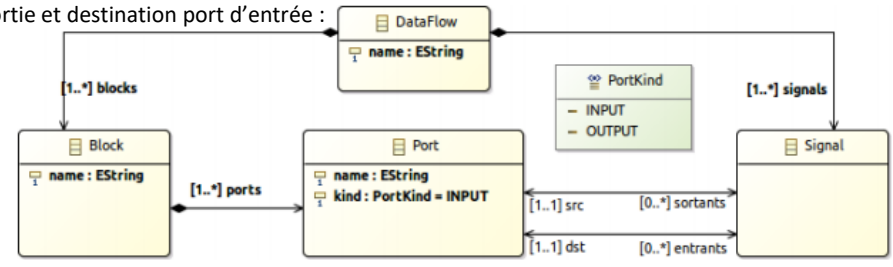
```
invariant nomUnique: DataFlow.allInstances()
          ->select(i | i.oclIsTypeOf(Block))
          ->forAll(i,j | i<>j implies i.name <> j.name);
```

2- Noms ports unique au niveau d'un modèle :

```
Inv nomPortUnique :
self.block->collectNested(port.name)->flatten()->isUnique(self);
```

3- Signal a pour source port de sortie et destination port d'entrée :

```
Context Signal
Self.src.kind = OUTPUT
Self.dst.kind = INPUT
```



4 – Au plus un signal qui connecte 2 ports donnés :

```
Context port
Port.allInstances()
     ->forAll(i,j | i<>j implies i.sortant->collect(signal.dst)->select(k|k.name = j.name)-
     >size() <=  1 and i.entrant->collect(signal.src)->select(k|k.name = j.name)->size() <=
     1 ;
```

5 – NbPorts type INPUT dans un modèle :

```
Context DataFlow :
Self.block.ports->select(i | i.kind = INPUT)->size() ;
```

```
diagram exemple {
      block b1 () returns (p0)
      block b2 () returns (p1)
      block b2 (p2,p3) returns (p4)
      from p0 to p2
      from p1 to p3
      block b3 (p5) returns ()
      from p4 to p5
}
```

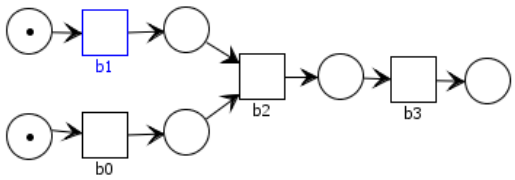
Xtext :

```
DataFlow return DataFlow :
            'diagram' name=ID '{'
               (b=Block
               |s=Signal)*
            '}' ;
Block :
       'block' name=ID '(' (p1=Port (',' p2=Port)*)* ')' returns '(' (p3=Port (',' p4=Port)*)* ')' ;

Signal :
       'from' portNameSrc=STRING 'to' portNameDst=STRING ;

Port : name=ID ;
```

Transformation PetriNet



Exemple règles ATL :

```
rule Process2PetriNet {
     from p: SimplePDL!Process
     to pn: PetriNet!PetriNet (name <- p.name)
}
rule WorkDefinition2PetriNet {
     from wd: SimplePDL!WorkDefinition
     to
     p_ready: PetriNet!Place (name <- wd.name + '_ready',
                     marking <- 1,
                     net <- wd.getProcess()),
     t_start: PetriNet!Transition (name <- wd.name + '_start', net <- wd. getProcess() ),
     a_nsed2s: PetriNet!Arc (
                      kind <- #normal,
                      weight <- 1,
                      source <- p_ready,
                      target <- t_start,
                      net <- wd.getProcess()  )
}
rule WorkSequence2PetriNet {
     from ws: SimplePDL!WorkSequence
     to a_ws: PetriNet!Arc (
          kind <- #read_arc,
          weight <- 1,
      source <- if ( (ws.linkType = #finishToStart) or (ws.linkType = #finishToFinish) )
                then thisModule.resolveTemp(ws.predecessor,'p_finished')
                else thisModule.resolveTemp(ws.predecessor,'p_started')
                endif,
     target <- if ( (ws.linkType = #finishToStart) or (ws.linkType = #startToStart) )
               then thisModule.resolveTemp(ws.successor,'t_start')
               else thisModule.resolveTemp(ws.successor,'t_finish')
               endif,
     net <- ws.getProcess() )
}
```