Université de Lomé | CENTRE INFORMATIQUE ET DE CALCUL (CIC)

Conception Programmation Orienter Objet

Thème de l'exposer :

Introduction à l'OCL et les mécanismes avancés : Concurrence-Séquence-Itération-Exception

Groupe 7 : AGBOZOH-KENNETH

Plan

Partie 1 : OCL

Définition de l’OCL

Définition d’une contrainte

Cas Pratique

Typologie des contraintes OCL

Exercices

Partie 2 : Mécanismes avances

Concurrence

Définition de la concurrence

Itération

Séquence

Exceptions

Partie 1 : OCL

1-Définition de OCL

C'est avec OCL (*Object Constraint Language*) qu'UML formalise l'expression des contraintes. Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML, et en particulier au diagramme de classes.

OCL existe depuis la version 1.1 d'UML et est une contribution d'IBM. OCL fait partie intégrante de la norme UML depuis la version 1.3 d'UML. Dans le cadre d'UML 2.0, les spécifications du langage OCL figurent dans un document indépendant de la norme d'UML, décrivant en détail la syntaxe formelle et la façon d'utiliser ce langage.

OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme :

- des invariants sur des classes ;
- des préconditions et des postconditions à l'exécution d'opérations :
    - les préconditions doivent être vérifiées avant l'exécution,
    - les postconditions doivent être vérifiées après l'exécution ;
- des gardes sur des transitions de diagrammes d'états-transitions ou des messages de diagrammes d'interaction ;
- des ensembles d'objets destinataires pour un envoi de message ;
- des attributs dérivés, etc.

2-Définition Contrainte

Une contrainte constitue une condition ou une restriction exprimée sous forme d'instruction dans un langage qui peut être naturel ou formel. Une contrainte désigne une restriction qui doit être appliquée par une implémentation correcte du système.

Nous avons déjà vu comment exprimer certaines formes de contraintes avec UML :
Contraintes structurelles :

- les attributs dans les classes, les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance), la cardinalité et la navigabilité des propriétés structurelles, etc. ;

Contraintes de type :

- typage des propriétés, etc. ;

Contraintes diverses :

- les contraintes de visibilité, les méthodes et classes abstraites (contrainte *abstract*), etc.

Dans la pratique, toutes ces contraintes sont très utiles, mais se révèlent insuffisantes. Toutefois, UML permet de spécifier explicitement des contraintes particulières sur des éléments de modèle.

3-Cas Pratique

Enonce :

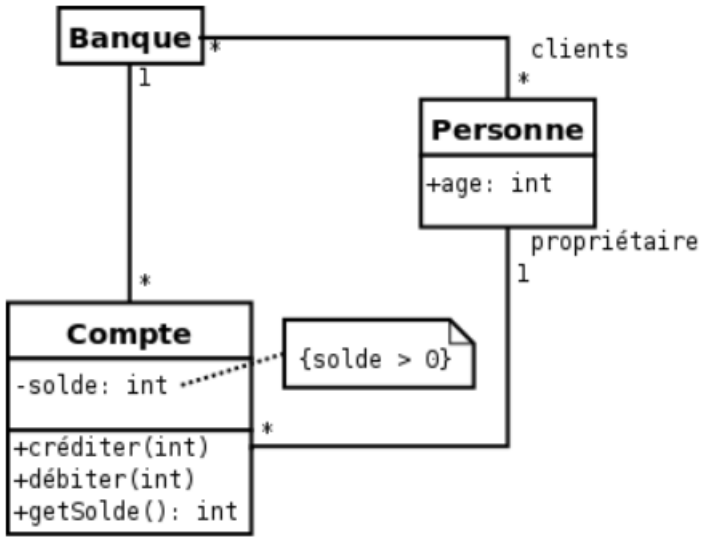
Plaçons-nous dans le contexte d'une application bancaire. Il nous faut donc gérer :

- des comptes bancaires ;
- des clients ;
- et des banques.

De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :

- un compte doit avoir un solde toujours positif ;
- un client peut posséder plusieurs comptes ;
- une personne peut être cliente de plusieurs banques ;
- un client d'une banque possède au moins un compte dans cette banque ;
- un compte appartient forcément à un client ;
- une banque gère plusieurs comptes ;
- une banque possède plusieurs clients.

Solution :



4. Typologie des contraintes OCL

4.1. Contexte (context)

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL :

- en écrivant la contrainte entre accolades ({}) dans une note. L'élément pointé par la note est

alors le contexte de la contrainte ;

- en utilisant le mot-clef context dans un document accompagnant le diagramme

a. Syntaxe

Sélectionnez
context <élément>
<élément> peut être une classe, une opération, etc. Pour faire référence à un élément op (comme une opération), il faut utiliser les :: comme séparateur.

b. Exemple
Le contexte est la classe Compte :

- **context** Compte

Le contexte est l'opération getSolde() de la classe Compte :

- **context** Compte::getSolde()

4.2. Invariants (inv)
Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.

a. Syntaxe
Sélectionnez
inv : <expression_logique>
<expression_logique> est une expression logique qui doit toujours être vraie.

b. Exemple
Le solde d'un compte doit toujours être positif.
context Compte
inv : solde > 0

4.3. Pré conditions et postconditions (pre, post)
Une pré condition (respectivement une post condition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant l'appel d'une opération.
Dans l'expression de la contrainte de la post condition, deux éléments particuliers sont utilisables :

- l'attribut result qui désigne la valeur retournée par l'opération ;
- et <nom_attribut>@pre qui désigne la valeur de l'attribut <nom_attribut> avant l'appel de l'opération.

a. Syntaxe

- Précondition :

Sélectionnez
pre : <expression_logique>

- Post condition :

Sélectionnez
post : <expression_logique>
<expression_logique> est une expression logique qui doit toujours être vraie.

b. Exemple
Concernant la méthode débiter de la classe Compte, la somme à débiter doit être positive pour que
l'appel de l'opération soit valide et, après l'exécution de l'opération, l'attribut solde doit avoir pour
valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.
context Compte::débiter(somme : Real)

 pre : somme > 0
 post : solde = solde@**pre** – somme

4.4. Résultat d'une méthode (body)
Ce type de contrainte permet de définir directement le résultat d'une opération.

a. Syntaxe
Sélectionnez
body : <requête>
<requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type du résultat de l'opération désignée par le contexte.

b. Exemple
context Compte::getSolde() : Real
body : solde

4.5. Définition d'attributs et de méthodes (def et let…in)
def est un type de contrainte qui permet de déclarer et de définir la valeur d'attributs comme la séquence let…in. def permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte.

a. Syntaxe de let…in
Sélectionnez
let <déclaration> = <requête> in <expression>
Un nouvel attribut déclaré dans <déclaration> aura la valeur retournée par l'expression <requête> dans toute l'expression <expression>.

b. Syntaxe de def
Sélectionnez
def : <déclaration> = <requête>
<déclaration> peut correspondre à la déclaration d'un attribut ou d'une méthode.
<requête> est une expression qui retourne un résultat dont le type doit être compatible avec le type de l'attribut, ou de la méthode, déclaré dans <déclaration>.
Dans le cas où il s'agit d'une méthode, <requête> peut
utiliser les paramètres spécifiés dans la déclaration de la méthode.

c. Exemple
Pour imposer qu'une personne majeure doit avoir de l'argent
context Personne
 def : argent : int = compte.solde->sum()
 context Personne
inv : age>=18 implies argent>0

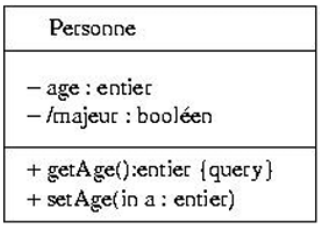
5-Exercices :

Exercice

❒ **Ajoutez un attribut mère de type Personne dans la classe Personne.**

❒ **Ecrivez une contrainte précisant**

- que la mère d'une personne ne peut être cette personne elle-même
- et que l'âge de la mère doit être supérieur à celui de la personne

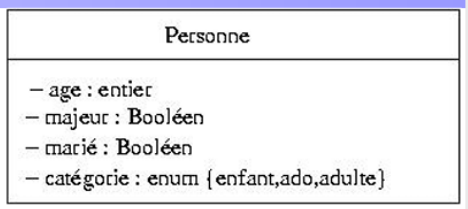



```
context Personne inv:
self.mere <> self and self.mere.age > self.age
```

Exercice

❒ **Avec la classe Personne « étendue »**

- Indiquez qu' une personne mariée est forcément majeur



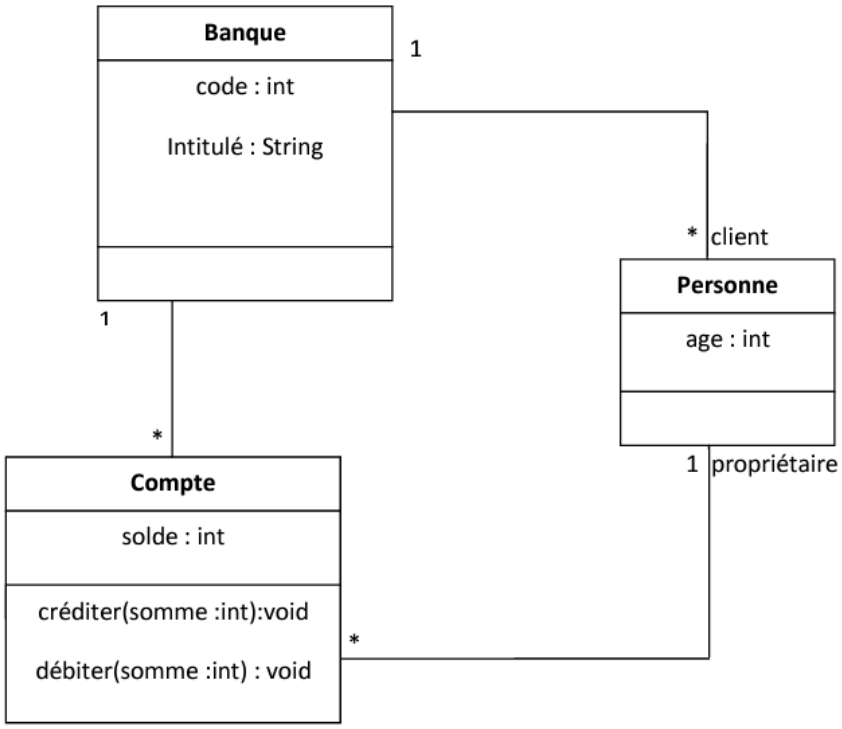

```
context Personne inv:
marié  implies majeur
```

- Trouvez une version plus compacte de l' expression suivante

```
context Personne inv majeurIf:
if age >=18 then majeur=vrai
else majeur=faux endif
```

```
context Personne inv:
majeur =  age >= 18
```

Exercice V



Donnez une contrainte OCL qui spécifie :

1. A un objet compte correspond un et un seul objet Personne.
2. La méthode débiter(somme :int), où le paramètre somme doit être positif et nouveau_solde = ancien_solde - somme.
3. Il n'existe pas de clients de la banque dont l'âge est inférieur à 18 ans
4. L'ensemble des clients de la banque associé à un compte contient le propriétaire de ce compte.

Exercice V

context Compte inv:
propriétaire -> size() = 1

context Compte::débiter(somme : Integer)
pre: somme > 0
post: solde = solde@pre – somme

context Banque **inv :**
not(clients -> exists (age < 18))

context Compte inv :
banque.clients -> includes(propriétaire)

Partie 2 : Les mécanismes avancés

1-Concurrence :

a. Définition de la concurrence

Les actions qui se déroulent en même temps sont dites concurrentes.

Le principe de concurrence évoque l'exécution simultanée de deux ou plusieurs taches.

Firure1. Sequentialflow

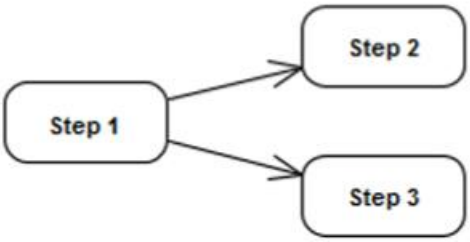
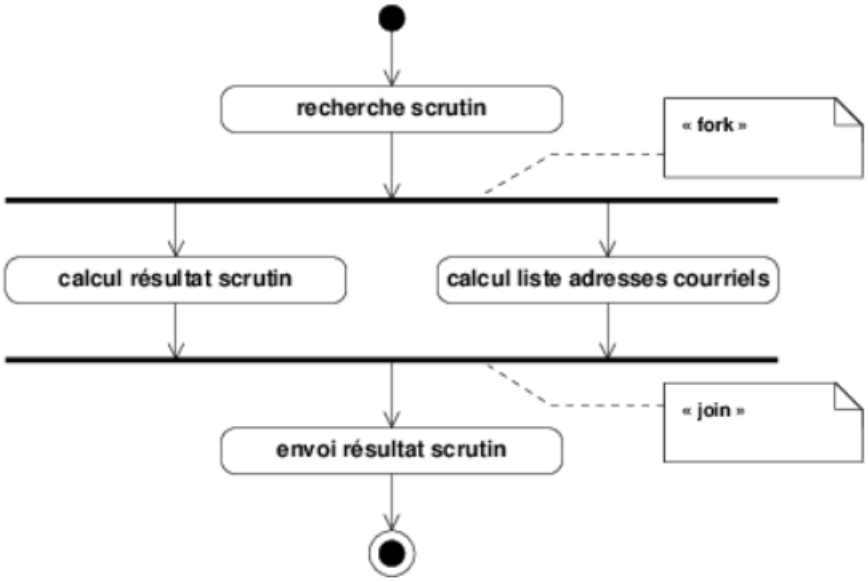


Figure 2. Concurrent flow (parallel split)

Elles sont modélisées entre deux traits épais comme des chemins parallèles, le premier de ces traits étant appelé « *fork* » et le second « *join* ». Les séquences d'actions en parallèle débutent en même temps, sont toutes exécutées, mais par définition ne finissent pas au même instant. La fin de l'action *join* intervient lorsque toutes les actions en parallèle se terminent ; *join* est donc un point de synchronisation.



Les diagrammes d’états-transitions permettent de décrire efficacement les mécanismes concurrents grâce à l’utilisation d’états orthogonaux.
Un état orthogonal est un état composite comportant plus d’une région, chaque région représentant un flot d’exécution. Graphiquement, dans un état orthogonal, les différentes régions sont séparées par un trait horizontal en pointillé allant du bord gauche au bord droit de l’état composite.
Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous-états.
Quand un état composite comporte plus d'une région, il est qualifié d'état orthogonal
Toutes les régions concurrentes d’un état composite orthogonal doivent atteindre leur état final pour que l’état composite soit considéré comme terminé.

2-ITERRATION

L'idée est simple : pour modéliser (comprendre et représenter) un système complexe, il vaut mieux s'y prendre en plusieurs fois, en affinant son analyse par étape.

Cette démarche devrait aussi s'appliquer au cycle de développement dans son ensemble, en favorisant le prototypage.

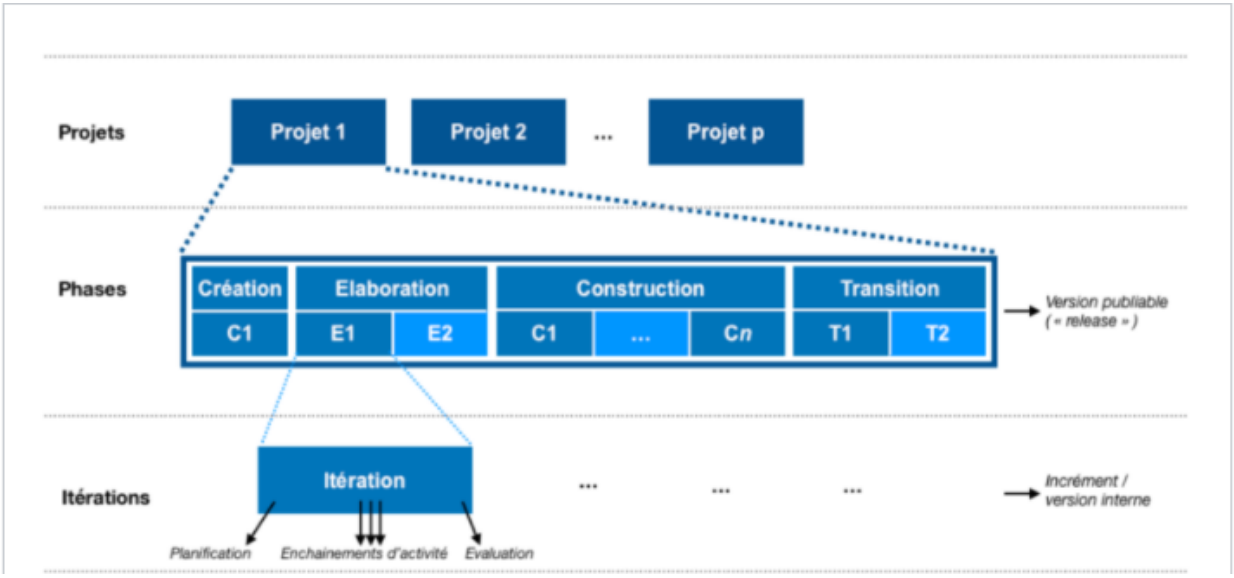
Le but est de mieux maîtriser la part d'inconnu et d'incertitude qui caractérisent les systèmes complexes.

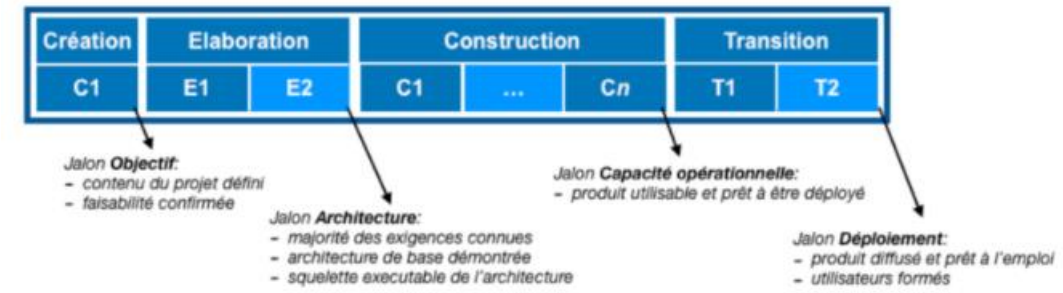
Qu’entendons-nous par itération ? C'est une séquence distincte d'activités avec un plan de base et des critères d'évaluation, qui produit une version. Le contenu d'une itération est porteur d'améliorations ou d'évolutions du système.

Que signifie un incrément ? C'est la différence (delta) entre deux versions produites à la fin de deux itérations successives.

Les itérations sont lies aux processus unifiés (UP : Unified Processus).

Un UP se caractérise par une démarche itérative et incrémentale, pilotée par les cas d'utilisation, et centrée sur l'architecture et les modèles UML. Elle définit un processus intégrant toutes les activités de conception et de réalisation au sein de cycles de développement composés d'une phase de création, d'une phase d'élaboration, d'une phase de construction et d'une phase de transition, comprenant chacune plusieurs itérations.

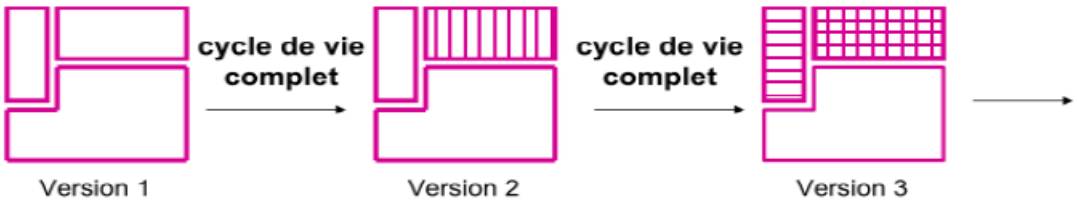




Le processus unifié préconise une planification itérative dans laquelle « le plan précède l'action ».

Le principe est qu'un plan d'ensemble détermine en fonction de la complexité du projet les itérations nécessaires pour chaque phase et positionne les phases dans le temps. Ce plan d'ensemble n'inclut pas de détail par itération. Les itérations sont planifiées de façon progressive au cours de l'avancement du projet. L'itération en cours est planifiée en détail lorsqu'elle démarre, avec un calendrier et un objectif de contenu (cas d'utilisation à traiter, changements à apporter aux livrables des itérations précédentes, composants à réaliser...). Le plan de l'itération suivante est préparé lorsque les éléments en sont connus. Il est ainsi tout d'abord estimé dans les grandes lignes, puis affiné en fonction des informations découvertes lors de l'itération en cours.

Dans le cas particulier de la phase de création, le contour du projet est en général trop incertain pour permettre une planification réaliste. Le plan de la première itération de cette phase doit donc être considéré comme une tentative de plan qui doit être ajustée si nécessaire. A l'issue de ce cette phase, la faisabilité du développement est établie et le plan d'ensemble correspondant à la vision du projet peut être produit.



3-SEQUENCE

Il s'agit d'une suite de messages entre les acteurs du système. Le diagramme de séquence représente la séquence de messages entre les objets au cours d'une interaction.

Ils définissent comment les éléments du système interagissent entre eux et avec les acteurs.

A moins que le système à modéliser soit extrêmement simple, nous ne pouvons pas modéliser la dynamique globale du système dans un seul diagramme. Nous ferons donc appel à un ensemble de diagrammes de séquences chacun correspondant à une sous fonction du système, généralement d'ailleurs pour illustrer un cas d'utilisation.

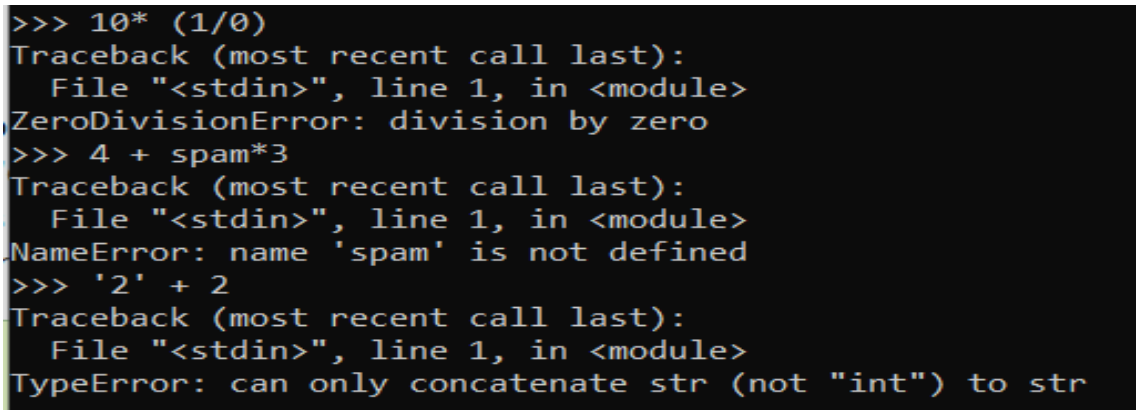
4-EXCEPTIONS

Les exceptions constituent un moyen efficace pour gérer les erreurs qui pourraient survenir dans un programme ; on peut alors tenter de traiter ces erreurs, remettre le programme dans un état normal et reprendre l'exécution du programme.

Dans cette partie, nous allons apprendre à créer des exceptions, à les traiter et à sécuriser nos programmes en les rendant plus robustes.

En programmation, quel que soit le langage utilisé, il existe plusieurs types d'erreurs pouvant survenir. Parmi les erreurs possibles, on connaît déjà les erreurs de syntaxe qui surviennent lorsque l'on fait une faute dans le code source, par exemple si l'on oublie un point-virgule à la fin d'une ligne.
Ces erreurs sont faciles à corriger car le compilateur peut les signaler.

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des ***exceptions*** et ne sont pas toujours fatales : nous apprendrons bientôt comment les traiter dans nos programmes. La plupart des exceptions toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :



La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : les types indiqués dans l'exemple sont ZeroDivisionError, NameError et TypeError. Le texte affiché comme type de l'exception est le nom de l'exception native qui a été déclenchée. Ceci est vrai pour toutes les exceptions natives mais n'est pas une obligation pour les exceptions définies par l'utilisateur (même si c'est une convention bien pratique). Les noms des exceptions standards sont des identifiants natifs (pas des mots réservés).
Le reste de la ligne fournit plus de détails en fonction du type de l'exception et de ce qui l'a causée.
La partie précédente dans le message d'erreur indique le contexte dans lequel s'est produite l'exception, sous la forme d'une trace de pile d'exécution. En général, celle-ci contient les lignes du code source ; toutefois, les lignes lues à partir de l'entrée standard ne sont pas affichées.

Vous trouvez la liste des autres exceptions natives et leur signification dans Exceptions natives.

a. Gestion des exceptions
Principe général

Le principe général des exceptions est le suivant :

- on crée des zones où l'ordinateur va *essayer* le code en sachant qu'une erreur peut survenir ;
- si une erreur survient, on la signale en *lançant* un *objet* qui contient des informations sur l'erreur ;
- à l'endroit où l'on souhaite gérer les erreurs survenues, on *attrape* l'objet et on gère l'erreur.

C'est un peu comme si vous étiez coincés sur une île déserte. Vous lanceriez à la mer une bouteille contenant avec des informations qui permettent de vous retrouver. Il n'y aurait alors plus qu'à espérer que quelqu'un attrape votre bouteille (sinon vous mourrez de faim).
C'est la même chose ici, on lance un objet en espérant qu'un autre bout de code le rattrapera, sinon le programme plantera.

La gestion des exceptions permet, si elle est réalisée correctement, de traiter les erreurs d'implémentation en les prévoyant à l'avance. Cela n'est pas toujours réalisable de manière exhaustive car il faudrait penser à toutes les erreurs susceptibles de survenir, mais on peut facilement en éviter une grande partie.
Il est possible d'écrire des programmes qui prennent en charge certaines exceptions. Regardez l'exemple suivant, qui demande une saisie à l'utilisateur jusqu'à ce qu'un entier valide ait été entré, mais permet à l'utilisateur d'interrompre le programme (en utilisant **Control-C** ou un autre raccourci que le système accepte) ; notez qu'une interruption générée par l'utilisateur est signalée en levant l'exception **KeyboardInterrupt**.

```
>>> while True:
...     try:
...             x = int(input("Veuillez saisir un nombre: "))
...             break
...     except ValueError:
...             print("Oopppssss! Ce n'est pas un nombre valide. Veuillez réessayer..")
...
```

L'instruction **try** fonctionne comme ceci :

- Premièrement, **la *clause try*** (instruction(s) placée(s) entre les mots-clés try et except) est exécutée ;
- si aucune exception n'intervient, la clause except est sautée et l'exécution de l'instruction try est terminée ;
- si une exception intervient pendant l'exécution de la clause try, le reste de cette clause est sauté. Si le type d'exception levée correspond à un nom indiqué après le mot-clé except, la clause except correspondante est exécutée, puis l'exécution continue après l'instruction try ;
- si une exception intervient et ne correspond à aucune exception mentionnée dans la clause except, elle est transmise à l'instruction try de niveau supérieur ; si aucun gestionnaire d'exception n'est trouvé, il s'agit d'une *exception non gérée* et l'exécution s'arrête avec un message comme indiqué ci-dessus.

Une instruction **try** peut comporter plusieurs clauses except pour permettre la prise en charge de différentes exceptions. Mais un seul gestionnaire, au plus, sera exécuté. Les gestionnaires ne prennent en charge que les exceptions qui interviennent dans la **clause *try*** correspondante, pas dans d'autres gestionnaires de la même instruction **try**. Mais une même clause except peut citer plusieurs exceptions sous la forme d'un tuple entre parenthèses, comme dans cet exemple :

```
...     except (RuntimeError, TypeError, NameError):
...             pass
```

Une classe dans une **clause except** est compatible avec une expression If c'est la même classe ou une classe de base de celle-ci (mais pas l'inverse – une **clause except** listant une classe dérivée n'est pas compatible avec une classe de base). Par exemple, le code suivant imprimera B, C, D dans cet ordre. n'est pas compatible avec une classe de base). Par exemple, le code suivant imprimera B, C, D dans cet ordre : Notez que si les **clauses except** étaient inversées (avec except B en premier), B, B, B aurait été imprimé - la première **clause except** correspondante est déclenchée.

```
    class B(Exception):
        pass

    class C(B):
        pass
```

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

La dernière **clause except** peut omettre le(s) nom(s) d'exception(s) et joue alors le rôle de joker. C'est toutefois à utiliser avec beaucoup de précautions car il est facile de masquer une vraie erreur de programmation par ce biais. Elle peut aussi être utilisée pour afficher un message d'erreur avant de propager l'exception (en permettant à un appelant de gérer également l'exception) :

L'instruction try … except accepte également une ***clause else*** optionnelle qui, lorsqu'elle est présente, doit se placer après toutes les **clauses except**. Elle est utile pour du code qui doit être exécuté lorsqu'aucune exception n'a été levée par **clause try**. Par exemple :

```
for arg in sys_version.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Il vaut mieux utiliser la **clause else** plutôt que d'ajouter du code à la **clause try** car cela évite de capturer accidentellement une exception qui n'a pas été levée par le code initialement protégé par l'instruction **try … except**.

Quand une exception intervient, une valeur peut lui être associée, que l'on appelle *l'argument* de l'exception. La présence de cet argument et son type dépendent du type de l'exception.

La **clause except** peut spécifier un nom de variable après le nom de l'exception. Cette variable est liée à une instance d'exception avec les arguments stockés dans **instance.args**. Pour plus de commodité, l'instance de l'exception définit la méthode **__str__()** afin que les arguments puissent être affichés directement sans avoir à référencer .args. Il est possible de construire une exception, y ajouter ses attributs, puis la lever plus tard.

```
89  try:
90      raise Exception('spam', 'eggs')
91  except Exception as inst:
92      print(type(inst))    # l'instance de l'exception
93      print(inst.args)     # arguments stockés dans .args
94      print(inst)          #__str__ permet d'imprimer directement les args,
95                           # mais peut être surchargé dans les sous-classes d'exception
96      x, y = inst.args     # ouvrir les arguments
97      print('x =', x)
98      print('y =', y)
```

```
PROBLÈMES  SORTIE  TERMINAL  CONSOLE DE DÉBOGAGE                    powershell

    for arg in sys_version.argv[1:]:
AttributeError: 'str' object has no attribute 'argv'
PS C:\Users\agboz\Documents\CIC\Semestre_1\Conception_programmation_Oriente_Objet\Python> py TD_1.py
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Si une exception a un argument, il est affiché dans la dernière partie du message des exceptions non gérées.

Les gestionnaires d'exceptions n'interceptent pas que les exceptions qui sont levées immédiatement dans leur **clause try**, mais aussi celles qui sont levées au sein de fonctions appelées (parfois indirectement) dans la **clause try**. Par exemple :

```
def fonct():
    x = 1/0

try:
    fonct()
except ZeroDivisionError as err:
    print('Gestion d une erreur d exécution:', err)
```

```
PS C:\Users\agboz\Documents\CIC\Semestre_1\Conception_programmation_Oriente_Objet\Python> py TD_1.py
Gestion d une erreur d exécution: division by zero
```

b. Déclencher les exceptions

L'instruction **raise** permet au programmeur de déclencher une exception spécifique. Par exemple :

```
>>> raise NameError('Bonjour')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Bonjour
```

L'argument de **raise** indique l'exception à lever. Il doit s'agir soit d'une instance d'exception, soit d'une classe d'exception (une classe qui dérive de Exception).

Si vous avez besoin de savoir si une exception a été levée mais que vous n'avez pas intention de la gérer, une forme plus simple de l'instruction **raise** permet de propager l'exception :

```
>>> try:
...     raise NameError('Bonjour')
... except NameError:
...     print('Une exception s est produite !')
...     raise
...
Une exception s est produite !
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Bonjour
```

c. Exceptions définies par l'utilisateur

Les programmes peuvent nommer leurs propres exceptions en créant une nouvelle classe d'exception (voir Classes pour en savoir plus sur les classes de Python). Les exceptions sont typiquement dérivées de la classe *Exception*, directement ou non.les classes d'exceptions peuvent être définies pour faire tout ce qu'une autre classe peut faire. Elles sont le plus souvent gardées assez simples, n'offrant que les attributs permettant aux gestionnaires de ces exceptions d'extraire les informations relatives à l'erreur qui s'est produite. Lorsque l'on crée un module qui peut déclencher plusieurs types d'erreurs distincts, une pratique courante est de créer une classe de base pour l'ensemble des exceptions définies dans ce module et de créer des sous-classes spécifiques d'exceptions pour les différentes conditions d'erreurs :

```
class Error(Exception):
    """Classe de base pour les exceptions dans ce module."""
    pass

class InputError(Error):
    """Exception levée pour les erreurs dans d'entrées.


        expression -- expression d'entrée dans laquelle l'erreur s'est produite
        message -- explication de l'erreur
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """ lorsqu'une opération tente une transition d'état qui n'est pas autorisée.

        previous -- état au début de la transition
        next -- tentative de nouvel état
        message -- explication de la raison pour laquelle la transition spécifique n'est pas autorisée
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

La plupart des exceptions sont définies avec des noms qui se terminent par « Error », comme les exceptions standards.

Beaucoup de modules standards définissent leurs propres exceptions pour signaler les erreurs possibles dans les fonctions qu'ils définissent. Plus d'informations sur les classes sont présentées dans le chapitre Classes.

Bibliographie

- OCL :

https://laurent-audibert.developpez.com/Cours-UML/?page=object-constraint-langage-ocl

http://deptinfo.unice.fr/twiki/pub/Linfo/CooL3/l3-specOCL-corrige.pdf

https://glpourmaster1.files.wordpress.com/2017/09/td-ocl-m1_exos_dc3a9ja_corrigc3a9s.pdf

- Exception :

https://docs.python.org/fr/3.5/tutorial/errors.html

- Concurrence :

https://123dok.net/article/concurrence-pdf-cours-informatique-mod%C3%A9lisation-uml.yr35gdpy

http://www-inf.it-sudparis.eu/COURS/CSC4002/EnLigne/Cours/CoursUML/4.12.13.html

https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-etats-transitions#L5-6

https://www.omg.org/ocup-2/documents/concurrency_in_uml_version_2.6.pdf

- Itération :

https://fr.wikipedia.org/wiki/Processus_unifi%C3%A9

- Séquence :

https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours05.html

https://www.ibm.com/docs/fr/SSCLKU_7.5.5/com.ibm.xtools.sequence.doc/topics/cseqd_v.html?view=kc

http://remy-manu.no-ip.biz/UML/Cours/coursUML5.pdf