CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

Cours 8

Conception et programmation par contrats

Conception par interfaces et conformité

- Par son utilisation d'interfaces offertes et requises, la programmation par composants met encore plus l'accent sur les *interfaces* dans la conception que la programmation par objets.
- Jusqu'à maintenant, nous avons vu les interfaces comme un moyen pour les composants d'*exposer* les services offerts et requis, mais dans un sens assez limité du terme.
- De fait, pour mettre en place une connexion, nous avons appliqué une notion de *conformité* entre interfaces requises et offertes qui est essentiellement celle des appels de méthodes en Java.
  - En réalité, un peu plus, car nous pouvons utiliser le connecteur pour rendre conforme deux interfaces, mais cela sera fait d'une manière purement *opérationnelle*.
- Cette notion de conformité est donc uniquement axée sur la correspondance des noms de méthodes, des types des paramètres et des types de résultats, au sens de Java.

Conformité sémantique

- Depuis longtemps, la conception logicielle s'intéresse à renforcer la notion de conformité entre services offerts et requis pour inclure ce qu'ils font ou doivent faire et ce, avec deux objectifs :
  1. Pouvoir exprimer de manière plus abstraite ce qu'on attend et ce ce que fait un service, c'est à dire sa *sémantique*.
     - L'idée-phare est de *spécifier* un service dans un langage *formel*, ayant une sémantique (mathématique) connue, forte et bien définie, permettant aussi d'échanger avec les experts métiers.
  2. Pouvoir appliquer une forme de *conformité sémantique* de fond entre interfaces requises et offertes plutôt qu'une simple *conformité lexicale des signatures*.
- Pour voir comment atteindre ces objectifs, nous allons explorer un continuum de moyens allant :
  - du plus formel avec les *types abstraits de données* et la *sémantique axiomatique*
  - jusqu'au plus pragmatique avec la *conception* et la *programmation par contrats*.

Plan

1. Types abstraits de données
2. Conception par contrats
3. Contrats sur des propriétés non-fonctionnelles
4. Contrats et héritage
5. Contrats et gestion de la concurrence

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre l'objectif de la conception logicielle de séparer la spécification de ce que fait le logiciel (le *quoi*) de la manière de la faire (le *comment*, son implantation).
   - Comprendre les type de données abstraits (TAD) et leur *spécification* comme types algébriques.
   - Comprendre la distinction entre axiomatisation des TAD et l'axiomatisation de leur implantation.
   - Comprendre les rudiments de la sémantique axiomatique et sa relation avec la spécification par types algébriques.
2. Compétences à acquérir
   - Savoir définir des types abstraits de données simples sous la forme de types algébriques.
   - Savoir dériver informellement la validité d'appels aux opérations d'un TAD et la validité de séquences d'instructions simples par l'approche axiomatique.

Notion de types abstraits de données

- Les types abstraits de données ont été proposés pour capturer l'essence des types de données par les *propriétés* de leurs opérations qu'ils proposent plutôt que par leur *implantation*.
- Le terme *abstrait* fait référence à deux principes importants :
  1. La *séparation* entre ce que font les opérations et la manière de le faire (complémentarité du *quoi* et du *comment*).
  2. L'*encapsulation* de l'implantation dans une boîte noire, sous une frontière opaque, pour *abstraire* le reste du programme des choix qui y sont faits et y assurer le maintien de la cohérence de la représentation (invariants de représentation).
- Le domaine de la conception logicielle a engendré de nombreux langages de spécification pour les types abstraits de données.
- Nous allons nous intéresser à une forme de spécification particulière (algébrique/axiomatique) pavant la voie à une *approche semi-formelle pragmatique* pour améliorer la qualité du logiciel.

Exemple de la pile à taille bornée

- Dans une approche de types algébriques, un type abstrait de données (TAD) est défini par ses opérations *typées*, classées en :
  1. observateurs : opérations en lecture seule (état courant)
  2. constructeurs : opérations de création
  3. fonctions : opérations en écriture (potentiellement)
- et des *axiomes i.e.*, des expressions logiques :
  1. invariants : « toujours » vrais
  2. préconditions : devant être vraies avant l'appel d'une opération
  3. postconditions : devant être vraies après l'appel d'une opération
- Ces éléments sont exprimés dans un langage *mathématique* le plus formel possible permettant de raisonner (prouver des propriétés) sur les TAD. Ceci implique que :
  1. Toutes les opérations (sauf les constructeurs) prennent comme premier paramètre une valeur du TAD sur laquelle elles opérent.
  2. Les fonctions qui « changent » l'état de la valeur du TAD retournent en réalité une *nouvelle* valeur du TAD en résultat.

Exemple de la pile à taille bornée

Service BoundedStack<G>
Observers
empty(BoundedStack<G> s) -> boolean
full(BoundedStack<G> s) -> boolean
size(BoundedStack<G> s) -> int
capacity(BoundedStack<G> s) -> int
Constructors
new(int capacity) -> BoundedStack<G>
Functions
push(BoundedStack<G> s, G g) -> BoundedStack<G>
pop(BoundedStack<G> s) -> BoundedStack<G>
top(BoundedStack<G> s) -> G
Axioms
For all x:G, c:int and s:BoundedStack<G>
[invariants]
size(s) >= 0 and size(s) <= capacity(s)
empty(s) = (size(s) = 0)
full(s) = (size(s) = capacity(s))
empty(new(c))
capacity(new(c)) = c
not empty(push(s,x))
not full(pop(s))
top(push(s,x)) = x
pop(push(s,x)) = s
[preconditions]
new(c) require c >= 0
push(s,x) require not full(s)
pop(s) require not empty(s)
top(s) require not empty(s)
[postconditions]
push(s,x) ensure size(s) = size(s@old) + 1
pop(s) ensure size(s) = size(s@old) - 1
top(s) ensure size(s) = size(s@old)

- Les préconditions s'appliquent sur les valeurs au moment de l'appel.
- Les postconditions s'appliquent sur les valeurs après l'exécution.
  - La notation @old réfère à la valeur avant l'exécution de l'opération.
  - La notation @result permet de désigner le résultat d'une opération autre que la valeur du TAD retournée le cas échéant.

Raisonnement à partir des axiomes

- Les axiomes permettent de raisonner sur le programme en les appliquant successivement à une séquence d'opérations.
- Ex.: Soit Σ les axiomes de BoundedStack<G>, montrez que $c > 2 \implies$ not full(push(push(new(c),x1),x2)).

soit s0 = new(c), s1 = push(s0,x1), s2 = push(s1,x2) alors
$\Sigma \vdash$ capacity(s0) = c and capacity(s1) = c and capacity(s2) = c
$\Sigma \vdash$ size(s0) = 0
$\Sigma \vdash$ size(s1) = size(s0) + 1 = 1
$\Sigma \vdash$ size(s2) = size(s1) + 1 = 2
$\Sigma \vdash$ capacity(s2) = c and c > 2 $\implies$ size(s2) < capacity(s2)
$\Sigma \vdash$ size(s2) < capacity(s2) $\implies$ not full(s2)

- Cette déduction, en réalité peu formalisée, met toutefois en évidence un autre principe des axiomes d'un TAD : la *non redondance*.
  - Éviter d'ajouter des axiomes quand ils peuvent être déduits d'autres axioms déjà présents.

Extension à la représentation interne

- Tels que nous les avons présentés, les TAD ne s'intéressent qu'aux propriétés *abstraites* ou *externes*.
- Principes qui en découlent :
  1. Toutes les observations sur les valeurs du TAD utilisées dans les axiomes sont déclarées parmi ses observateurs.
  2. Toute précondition imposée par une opération doit pouvoir être vérifiée (observée) par le client *avant* d'appeler l'opération.
  3. Toute postcondition promise par une opération doit pouvoir être vérifiée (observée) par le client *après* l'exécution de l'opération.
- Lorsqu'on passe à l'implantation d'un TAD, on choisit une *représentation interne* pour les valeurs du type de données (TD).
  - Les manipulations de cette représentation peuvent faire l'objet d'*axiomes de représentation* :
    1. invariants de représentation : devant être maintenus sur les données internes de représentation du TD.
    2. pré- et postconditions de représentation : devant être vraies sur les données internes de représentation du TD respectivement avant et après l'exécution des opérations.

Exemple de la pile avec représentation interne

Service BoundedStack<G>
Representation
topIndex : int
items : Array<G>
Observers
empty(BoundedStack<G> s) -> boolean
full(BoundedStack<G> s) -> boolean
size(BoundedStack<G> s) -> int
capacity(BoundedStack<G> s) -> int
Constructors
new(int capacity) -> BoundedStack<G>
Functions
push(BoundedStack<G> s, G g) ->
BoundedStack<G>
pop(BoundedStack<G> s) ->
BoundedStack<G>
top(BoundedStack<G> s) -> G
Axioms
For all x:G, c:int and s:BoundedStack<G>
[invariants]
size(s) >= 0 and size(s) <= capacity(s)
empty(s) = (size(s) = 0)
full(s) = (size(s) = capacity(s))
empty(new(c))
capacity(new(c)) = c
not empty(push(s,x))
not full(pop(s))
top(push(s,x)) = x
pop(push(s,x)) = s
[preconditions]
new(c) require c >= 0
push(s,x) require not full(s)
pop(s) require not empty(s)
top(s) require not empty(s)
[postconditions]
push(s,x) ensure size(s) = size(s@old) + 1
pop(s) ensure size(s) = size(s@old) - 1
top(s) ensure size(s) = size(s@old)
Representational axioms
For all x:G, c:int and s:Stack<G>
[representational invariants]
capacity(s) = size(items)
size(s) = topIndex + 1
For all i >= 0 and i <= capacity(s)
not empty(s) => ((i <= topIndex => items[i] != #undefined)
and (i > topIndex => items[i] = #undefined))
[representational preconditions]
[representational postconditions]
new(c) ensure size(items) = c
push(s,x) ensure topIndex = topIndex@old + 1
pop(s) ensure topIndex = topIndex@old -1
top(s) ensure @result = items[topIndex]

#undefined est une valeur de plein droit, distincte de toute autre valeur manipulable.

Nature des axiomes internes

- Les axiomes de représentation n'ont pas la même interprétation que les axiomes externes.
  - D'abord, ils s'adressent à l'*implanteur* du TD plutôt qu'au client.
  - Les conditions n'étant pas vérifiables par le client, le code d'implantation du TD est seul responsable d'assurer la véracité de tous ces axiomes.
    Ex.: le fait que « topIndex = topIndex@old + 1 » soit vraie après l'appel à push ne peut être ni vérifié ni assuré par le client ; c'est nécessairement le code du TD qui doit le faire.
- Il s'agit donc bien uniquement de s'assurer de l'exactitude du *comment* (l'implantation) par rapport au *quoi* (la spécification) alors que les axiomes externes s'adressent aux appels des services par le client *i.e.*, uniquement du *quoi*.
- Cela nous amène à faire le lien avec l'implantation des TAD par du code dans un langage de programmation comme Java.

Raisonnement sur l'implantation à partir des axiomes

- En Java, un type abstrait va être implanté par une classe et exécuté sur un objet, donc le code sera *impératif*.
  - L'affectation permet de changer les valeurs des variables.
  - Les fonctions du TAD seront implantées par des méthodes qui vont *modifier en place la représentation interne* plutôt que de *retourner une nouvelle valeur* du TD.
    - Ex.: pop(BoundedStack<G> s) -> BoundedStack<G>
      devient `public void pop()` sur la classe `BoundedStack<G>`.
  - Condition : pas de changement de nature (classe) de l'objet.
- Passer de la spécification mathématique à l'implantation demande d'appliquer les axiomes dans le contexte (état mémoire) d'un programme : c'est ce que fait la *sémantique axiomatique*.
- Élément de base : triplet de Hoare { P } `I` { Q }, où `I` est une instruction, interprétée sémantiquement de la manière suivante :

  Toute exécution de `I` dans un état où l'axiome P est satisfait se terminera dans un état où l'axiome Q sera satisfait.

Vision axiomatique de la sémantique des programmes

- La sémantique formelle d'un programme établit une relation entre ses entrées possibles et les sorties souhaitées.
  - En sémantique *dénotationelle*, on exprime la sémantique d'un programme par la *fonction mathématique* qu'il calcule.
- En sémantique *axiomatique*, on établit une relation entre :
  - une formule logique *supposée* vraie sur ses entrées et son état initial et
  - une formule logique *démontrée* vraie sur les résultats et l'état final.

  Traditionnellement la première est appelée *précondition* et la seconde *postcondition*.
- Définir une sémantique axiomatique pour un langage de programmation consiste à définir comment chacune des instructions du langage *transforme* une précondition *supposée* vraie en une postcondition *prouvée* vraie compte tenu de ce que chaque instruction fait.

Sémantique axiomatique d'un programme I

- L'idée de voir la sémantique des instructions comme une *transformation d'axiomes* va permettre de construire la sémantique d'un programme par transformation de pré- en postconditions instruction par instruction :
  $(\{ P \}\ \mathtt{I}_1\ \{ R \}) \wedge (\{ R' \}\ \mathtt{I}_2\ \{ Q \}) \wedge (R \Rightarrow R') \implies \{ P \}\ \mathtt{I}_1 ; \mathtt{I}_2\ \{ Q \}$
- Soit Inv, P et Q respectivement l'invariant d'un TAD et les pré- et postconditions d'une fonction *f* de ce TAD, on peut s'assurer que l'implantation de *f* par une méthode *m* de corps `B` satisfait aux axiomes du TAD en dérivant le triplet de Hoare :
  $$\{ \text{Inv} \wedge P \}\ \mathtt{B}\ \{ \text{Inv} \wedge Q \}$$
  à partir des instructions de `B`. On pourra faire de même avec l'invariant de représentation du TD.

Sémantique axiomatique d'un programme II

- Pour l'appel à une méthode, il faut lier le contexte appelant au contexte appelé.

  Contexte appelant
  $\{ P \}$ `x.m(...)` $\{ Q \}$

  Contexte appelé
  $\{ Inv \wedge P' \}$ `B` $\{ Inv \wedge Q' \}$

  avec $P \Rightarrow P'$ et $Q' \Rightarrow Q$, modulo les *changements de variables* et *substitutions de valeurs* dûs au changement de contexte.

- Ex.: appel de new.

$\{ y = 10 \}$
`s := new(y)`
$\Rightarrow$ $\{ c = 10 \wedge c \geq 0 \}$
`exec new(c)`
$\{ capacity(new(10)) = 10 \wedge empty(new(10)) \}$

$\{ y = 10 \wedge capacity(s) = y \}$ $\Leftarrow$

Plan

1 Types abstraits de données

2 Conception par contrats

3 Contrats sur des propriétés non-fonctionnelles

4 Contrats et héritage

5 Contrats et gestion de la concurrence

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les limites de la sémantique axiomatique et de types algébriques pour prouver formellement l'exactitude des logiciels.
   - Introduire la conception et la programmation par contrats comme une forme affaiblie ou semi-formelle parmettant d'augmenter pragmatiquement la qualité des logiciels.
   - Comprendre comment les concepts de l'approche par contrats s'étendent à la programmation par composants.
   - Élaborer une notion de conformité entre interfaces requises et offertes qui intègre les concepts de l'approche par contrats.
2. Compétences à acquérir
   - Savoir passer d'une spécification par types abstraits algébriques à une implantation sous la forme de classes ou de composants en utilisant l'approche contractuelle.
   - Savoir déterminer si une interface requise contractualisée est conforme à une interface offerte contractualisée.
   - Savoir interpréter les différents types de violations des contrats et agir sur le code pour corriger les erreurs qui les ont provoquées.

De la sémantique axiomatique aux contrats

- Une sémantique axiomatique est obtenue par *démonstration*.
  - Le seul fait d'arriver à dériver par preuve formelle la formule finale à partir de la formule initiale garantit l'exactitude du programme.
  - Mais c'est une tâche complexe, trop lourde pour un humain et dont l'automatisation complète pour tout programme demeure encore un objectif de recherche à ce jour, malgré les progrès constants dans les dernières décennies.
- Pragmatiquement, un moyen alternatif pour obtenir des programmes plus fiables consiste à utiliser une forme « affaiblie » de sémantique axiomatique :

  La conception et la programmation par contrats.
- Le principe général de l'approche par contrats est :
  - Si on ne peut pas *démontrer formellement* la sémantique axiomatique d'un programme, on peut au moins *vérifier à l'exécution* que tout ou partie des formules successivement attendues sont vraies.
  - *Vérifier ne donnera jamais une preuve*, mais cela augmentera au fil des exécutions la *confiance dans l'exactitude* du programme.

Contrats comme axiomatisation faible

- L'approche par contrats propose donc une forme « *affaiblie* » de sémantique axiomatique et de types de données abstraits.
- Les deux principaux affaiblissements sont :
  1. Le fait que les quantifications universelles "pour toute valeur possible" vont être interprétées comme "pour chaque valeur rencontrée dans une exécution effectivement réalisée".
  2. Au lieu de démontrer les axiomes, ils seront simplement vérifiés comme vrais *pour les données rencontrées* dans les exécutions effectivement réalisées.
- La situation devient comparable au degré de garantie obtenu par des tests par rapport à une preuve formelle, mais en vérifiant non seulement les sorties par rapport aux entrées mais aussi des états intermédiaires.
- D'ailleurs, l'approche du test logiciel systématique appelée test fondé sur les modèles (*model-based testing*) consiste à dériver systématiquement à partir des axiomes un ensemble de cas de tests à la fois couvrant et minimal.

Conception et programmation par contrats

Conception par contrats

Conception des relations clients/serveurs entre entités logicielles autour d'obligations réciproques formulées par des assertions.

Préconditions : obligations faites à l'appelant de fournir des paramètres et d'appeler le client dans un état tels que ses préconditions sont satisfaites.

Postconditions : engagements pris par l'appelé de rendre un résultat et de se retrouver dans un état tels que ses post-conditions sont satisfaites, *sous réserve que les préconditions étaient satisfaites*.

Programmation par contrats

Application des principes de la conception par contrats à des programmes grâce à des mécanismes intégrés dans les langages de programmation ou par des instructions ajoutées manuellement.

Le langage Eiffel

- Langage à objets conçu par Bertrand Meyer avec pour objectif d'intégrer la conception par contrats grâce à des mécanismes prévus dans le langage dès sa conception.
- En Eiffel, une classe définit son invariant et une méthode définit ses pré- et postconditions, sauf cas exceptionnel.
- Les travaux de Meyer font toujours autorité dans le domaine :
  - Il a étudié en profondeur tous les mécanismes de la programmation par contrats pour proposer des solutions pour leur implantation et leur intégration dans les programmes.
  - Il a également étudié en profondeur les conséquences de cette intégration à la fois sur la programmation et sur la conception logicielle.
  - Il a publié une littérature scientifique très importante sur ces sujets qui demeure une source inestimable d'avis et de conseils sur la bonne utilisation de l'approche par contrats.

Contrats et composants

- Le passage du contexte de la programmation par objets à celui de la programmation par composants exige de prendre en compte les caractéristiques spécifiques de cette dernière :
  - L'accent mis sur le développement indépendant des composants clients et fournisseurs ainsi que leur assemblage par des tiers.
  - L'introduction des interfaces requises pour rendre possible le développement indépendant des composants (clients).
- Relation client/serveur, des objets aux composants :
  - En programmation par objets, le développeur d'une classe cliente connaît ses classes fournisseures et peut donc gérer les engagements réciproques dans son code.
  - Avec les composants, le développeur d'un composant client est supposé pouvoir implanter ce dernier sans connaître à l'avance les composants fournisseurs qui vont lui être connectés.
  - Les contrats sur les interfaces requises permettent donc de rassembler et de rendre visibles en un endroit les signatures et engagements réciproques sur lesquels sont fondés les appels faits partout dans le code du client.

Contrats offerts et requis

- Rôle fondamental des interfaces requises : permettre au développeur d'un composant client d'écrire du code d'appels de services alors qu'il ne connaît ni la signature ni les contrats des services offerts.
  - Il écrit donc des appels valides selon l'interface requise *sous l'hypothèse* que le fournisseur offrira une interface conforme.
  - On externalise ainsi (boîte noire) les vérifications à faire en le faisant en deux étapes : le code du client versus son interface requise puis l'interface requise versus l'interface offerte.
- L'introduction de *contrats requis* va permettre au développeur d'écrire le code du composant client de manière à satisfaire et utiliser les engagements de ce contrat requis alors qu'il ne connaît pas encore le *contrat offert*.
- On obtient donc un composant client qui respecte les exigences du fournisseur mais à une condition fondamentale :

  Que les contrats offerts soient *conformes* aux requis !

Relations entre axiomes offerts et requis

- Comment étendre la conformité entre interfaces requises et offertes à la conformité des contrats requis et offerts ?
- Soient : P_r, Q_r les pré- et postconditions requises
  P_o, Q_o les pré- et postconditions offertes
  l'appelant doit faire en sorte que P_r soit respectée et s'appuie sur le fait que Q_r sera vraie alors que l'appelé doit faire en sorte que P_o et Q_o soient aussi respectées.
- Voici les deux conditions pour que *tout appel* respectant P_r et exigeant que Q_r le soit *reste valide* après connexion à une signature et un service qui respecte P_o, Q_o :
  1. $P_r \Rightarrow P_o$: ainsi dès que P_r est vraie, P_o le sera (*modus ponens*).
  2. $Q_o \Rightarrow Q_r$: ainsi dès que Q_o est vraie, Q_r le sera (*idem*).
- Ex.:
  $P_r =_{def} x > 10$ et $P_o =_{def} x >= 0$ alors $P_r \Rightarrow P_o$
  $Q_o =_{def} @result < 100$ et $Q_r =_{def} @result < 500$ alors $Q_o \Rightarrow Q_r$.

Conformité entre contrats offerts et requis

- Littéralement, une interface requise contractualisée est conforme à une interface offerte contractualisée si elles sont compatibles au sens de Java et si pour toutes les signatures compatibles :
  - les préconditions requises impliquent les préconditions offertes et
  - les postconditions offertes impliquent les postconditions requises.
- Plus formellement, pour $i = 1, \ldots, n$ soient :
  - une interface requise R *définissant* les signatures s_r^i auxquelles sont attachées des préconditions P_r^i et les postconditions Q_r^i et
  - une interface offerte O *incluant* des signatures s_o^i auxquelles sont attachées des préconditions P_o^i et les postconditions Q_o^i

  alors R et O sont compatibles noté $R \cong_I O$ si :
  - les signatures sont conformes au sens de Java, noté $s_r^i \cong_s s_o^i$,
  - les préconditions sont conformes, noté $P_r^i \cong_{pre} P_o^i$ si $P_r^i \Rightarrow P_o^i$ et
  - les postconditions sont conformes, noté $Q_r^i \cong_{post} Q_o^i$ si $Q_o^i \Rightarrow Q_r^i$

  modulo les changements de variables et substitutions de valeurs.

Interprétation de la violation des contrats

- Dans un contrat classique (à la Eiffel), les violations des contrats sont interprétées comme des erreurs du programme :
  - Une violation des préconditions est interprétée comme une faute de l'appelant qui n'a pas fourni des paramètres appropriés ou fait son appel dans un état correct de l'objet.
  - Une violation des postconditions est interprétée comme une faute de l'appelé qui n'a pas rendu un résultat approprié ou laissé l'objet dans un état correct.

  Comme on ne les prouve pas formellement, ces violations sont vérifiées à l'exécution et signalées par des exceptions explicites.
- Pour les contrats offerts et requis, cela reste vrai pour P_r et Q_o respectivement, mais pour P_o et Q_r c'est un peu différent :
  - Si P_o est violée alors que P_r ne l'était pas, c'est la *connexion* qui est en cause car les interfaces ne sont pas conformes.
  - Si Q_r est violée alors que Q_o ne l'était pas, c'est aussi la *connexion* qui est en cause pour la même raison.

  Il faut donc signaler des exceptions *différentes* dans ces cas, pour affiner les causes du problème.

Et les axiomes de représentation ?

- Les invariants ainsi que les axiomes de représentation sont essentiellement l'affaire du composant fournisseur.
  - Il doit en garantir le respect par son propre code.
  - Les invariants peuvent être vérifiés de l'extérieur grâce aux observateurs implantés comme méthodes/services.
  - Les axiomes de représentation ne sont pas visibles dans l'interface offerte, mais uniquement dans l'implantation du composant.
- En cas de violation des invariants ou des axiomes de représentation, des exceptions différentes doivent être signalées :
  - Pour les invariants, une exception du même genre que celles signalées par une violation de P_r ou Q_o.
  - Pour les axiomes de représentations, des exceptions internes comme par exemple erreur interne avec éventuellement une exception plus précise comme cause pour distinguer les violations d'invariants, de pré- ou de postconditions de représentation.

Typologie des exceptions de violation des contrats

On peut proposer une hiérarchie d'exceptions susceptibles d'être levées selon les cas de violation précédents (l'indentation représente l'héritage entre types d'exceptions) :

- `ContractException`
  - `PreconditionException`
  - `PostconditionException`
  - `InvariantException`
  - `ConnectionContractException`
    - `ConnectionPreconditionException`
    - `ConnectionPostconditionException`
  - `ImplementationContractException`
    - `ImplementationInvariantException`
    - `ImplementationPreconditionException`
    - `ImplementationPostconditionException`

Plan

1 Types abstraits de données

2 Conception par contrats

3 Contrats sur des propriétés non-fonctionnelles

4 Contrats et héritage

5 Contrats et gestion de la concurrence

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les limites du principe d'abstraction fonctionnelle que soutendent les approches à la TAD.
   - Introduire les notions de propriétés non-fonctionnelles et de qualités de service appliquées aux entités logicielles.
   - Comprendre les besoins engendrés par la contractualisation des propriétés non-fonctionnelles.
   - Comprendre la notion de conformité des contrats non-fonctionnels et son articulation avec la programmation par contrats.
   - Aborder la complexité de la vérification statique et dynamique des contrats non-fonctionnels.
2. Compétences à acquérir
   - Savoir exprimer des contraintes sur les propriétés non-fonctionnelles.
   - Savoir déterminer la conformité entre contrats non-fonctionnels.
   - Appréhender comment peut se faire la vérification des contrats non-fonctionnels.

Limites de l'abstraction par TAD

- La notion d'abstraction des TAD a rendu d'immenses services à l'informatique en permettant de produire des programmes plus robustes et plus faciles à maintenir.
- Cependant, l'idée selon laquelle il serait toujours possible d'ignorer complètement les choix d'implantation d'un type de données ne tient pas vraiment la route.
  - Dans plusieurs types de structures de données, la performance des opérations diffère selon les choix d'implantation ; la bonne solution dépend donc de l'utilisation prévue.
    Ex.: accès en lecture versus écriture.
- Aujourd'hui, il n'existe plus guère de systèmes informatiques qui ne soient partiellement répartis, embarqués et temps réel.
  - Les propriétés temporelles, la performance et plus généralement la consommation de ressources (énergie) deviennent des aspects cruciaux de leur bonne conception et implantation.
  - L'abstraction peut induire en erreur, ex.: tableur et fenêtres (G. Kiczales), évitement d'obstacles en robotique, ...

Propriétés non-fonctionnelles, contraintes et contrats

- Une *propriété non-fonctionnelle* désigne une qualité *observable* sur une entité logicielle portant sur le comment et non le quoi.
  - Certaines sont statiques : taille du code, nombre de modules, etc.
  - D'autres sont dynamiques : absence d'étreinte fatale, durée d'exécution, taille mémoire, énergie consommée, etc.
- À partir du moment où les propriétés sont observables et leurs valeurs comparables entre elles, il devient possible de poser des *contraintes* sur ces dernières.
  Ex.: duree < 0,01 secondes
- Un *contrat non-fonctionnel* définit un ensemble de contraintes imposées sur les valeurs de propriétés non-fonctionnelles elles-mêmes mesurées sur des opérations ou des entités logicielles.
- Une approche formelle de la spécification de ces propriétés considère des contraintes plus abstraites.
  Ex.: complexité $\in \mathcal{O}(n^2)$

Relation avec la qualité de service

- Un autre terme souvent utilisé pour désigner certaines propriétés non-fonctionnelles est la *qualité de service* (QdS).
  - Les qualités de service sont des propriétés *mesurables*.
- La QdS est un domaine qui a connu une forte effervescence au tournant des années 1990-2000.
  - L'explosion des utilisations des réseaux a poussé vers l'apparition de réseaux à garanties de service.
  - Une application phare : la vidéo à la demande sur réseau IP.
- Ce domaine a donc eu deux grandes orientations :
  - la mise en œuvre de la QdS, et donc comment prendre une contrainte de QdS et faire en sorte que le réseau/système l'assure effectivement ;
  - la spécification de la QdS et la vérification la conformité d'une implantation (déploiement) aux exigences d'une spécification.
- Les contrats non-fonctionnels participent plutôt de la seconde orientation.

Expression des contrats non-fonctionnels et de QdS

- Besoin pour les contrats de QdS : langage permettant
  1. d'exprimer les contraintes requises et offertes sous des formes nouvelles par rapport aux axiomes classiques,
  2. de vérifier la conformité d'un contrat offert avec un requis, et
  3. d'ouvrir vers la vérifications statique et dynamique des contrats.
- Un exemple : QML (*Quality of service Modeling Language*), dont les concepts fondamentaux sont :
  - ***dimension*** : propriété nommée, avec son type de valeurs et autres informations (unité de mesure, ...) ;
  - ***type de contrats*** : un ensemble de dimensions caractérisant une catégorie de QdS comme la performance, la disponibilité, la sécurité ou le temps ;
  - ***contrat*** : instantiation d'un type de contrats avec des contraintes particulières pour les dimensions de QdS qu'il présente ;
  - ***profil*** : association d'un contrat à une interface.
    Ex.: pour une même interface des profils *premium* versus *free* distincts selon la QdS offerte.

Exemple QML : serveur de taux de change

```
interface RateServiceI {              // Functional interface
  Rate latest(Currency from, Currency to);
  Forecast analysis(Currency c);
}

type Reliability = contract {
  numberOfFailures: decreasing numeric no / year; // small is beautiful
  TimeToRepair: decreasing numeric sec;
  availability: increasing numeric;               // big is beautiful
}
type Performance = contract {
  delay: decreasing numeric msec;
  throughput: increasing numeric no / sec;
}
systemReliability = Reliability contract {
  numberOfFailures < 10 no / year;
                                                  // statistical constraints
  TimeToRepair { percentile 100 < 20000; mean < 500; variance < 0.3 }
  availability > 0.8;
}
rateServerProfile for RateServiceI = profile {
  require systemReliability;                      // apply to all operations
  from latest require Performance contract {      // refinements by addition
    delay { percentile 50 < 10 msec;              // of constraints
            percentile 80 < 20 msec;
            percentile 100 < 40 msec; }
  };
  from analysis require Performance contract { delay < 4000 msec; };
}
```

Conformité à la QML I

- En QML, la conformité est une notion définie depuis les profils jusqu'aux contraintes :
  - Un profil P est conforme au profil Q si les contrats associés à P sont conformes à ceux associés à Q.
  - Un contrat X est conforme au contrat Y si toutes les contraintes définies par X sont conformes aux contraintes définies par Y.
  - Soient des contraintes c_1, c_2 sur une même dimension, c_1 est conforme à c_2 si c_1 est plus restrictive que c_2.
- Être « plus restrictive » dépend d'une autre notion importante en qualité de service : quelles valeurs sont les plus intéressantes ?
  - les plus grandes ? $\rightarrow$ les dimensions « `increasing` »
  - les plus petites ? $\rightarrow$ les dimensions « `decreasing` »

Ex.: delay $< 5 \implies$ delay < 10 (decreasing)
throughput $> 99\% \implies$ throughput $> 90\%$ (increasing).

Conformité à la QML II

- Conformité des contrats non-fonctionnels sur les interfaces :
  - interface requise $\Leftrightarrow$ contrat de QdS *requis R*
  - interface offerte $\Leftrightarrow$ contrat de QdS *offert O*
  - connexion : exige la *conformité* entre contrats de QdS

$$O \Longrightarrow R$$

Ex.: durée offerte < 5 $\Rightarrow$ durée requise < 10

Les contraintes de QdS se comportent plutôt comme des post-conditions puisqu'elles expriment des garanties offertes par les services en s'appuyant sur leur implantation.

Vérification des contrats non-fonctionnels

- Bien plus complexe que celles des contrats classiques :
  - Certaines contraintes sont vérifiables statiquement dans le meilleur des cas. Ex.: absence d'interblocage.
  - Beaucoup requièrent une vérification dynamique par essence.
- La vérification dynamique exige de *mesurer* les propriétés, ce qui interpelle des disciplines en elles-mêmes : la détection d'interblocage ou l'instrumentation et la mesure de performance.
- La mesure obtenue, il faut déterminer si cette mesure satisfait ou non la contrainte *i.e.*, la *sémantique* de la *satisfaction*.
  Ex.: Des données mesurées satisfont-elles une contrainte statistique ?
  C'est aussi une discipline en soi : celle des *tests statistiques*.
- Une violation peut signaler une exception, mais elle peut aussi avoir une interprétation économique :
  - Quelle compensation un client est-il en droit d'attendre si un service ne remplit pas son contrat de qualité de service ?

Plan

1 Types abstraits de données

2 Conception par contrats

3 Contrats sur des propriétés non-fonctionnelles

4 Contrats et héritage

5 Contrats et gestion de la concurrence

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la continuum entre systèmes de types dans les langages de programmation et axiomes.
   - Comprendre les règles de typage en cas de redéfinition dans les langages à objets : covariance et contravariance.
   - Comprendre par analogie les règles de raffinement des contrats lors d'un héritage entre interfaces.
   - Comprendre comment les règles de raffinement peuvent être assurées par une simple transformation du contrat héritant.
2. Compétences à acquérir
   - Savoir définir un contrat raffiné par rapport à un contrat hérité.
   - Savoir utiliser le patron percolateur pour définir des contrats raffinés théoriquement sûrs.
   - Savoir détecter et interpréter les violations des contrats raffinés pour distinguer les erreurs sur les conditions des erreurs de raffinement dues au patron percolateur.

Typage et axiomes

- L'approche des TAD nous invite à voir les types de données sous l'angle des opérations que l'on peut appliquer aux données et de leurs propriétés axiomatiques.
- Une vision complémentaire est la vision *ensembliste* : un type de données définit d'abord un ensemble de valeurs.
  - Ex.: le type `int` définit un ensemble d'entiers inclus dans $\mathbb{Z}$ et le type `double` définit un ensemble de réels inclus dans $\mathbb{R}$.
- Cette vision facilite l'interprétation des axiomes d'un TAD comme une *extension* du système standard de types.
  - Soit une opération `op(int i)` avec une précondition $i \geq 0$.
  - La *combinaison* du type `int` et de la précondition revient à typer `i` par l'ensemble $\{i \in \texttt{int} \mid i \geq 0\}$ *i.e.*, $\mathbb{N}$.
- Pourquoi cette distinction entre types « standards » et axiomes ?
  - Vérifier la conformité de types axiomatiques dans les programmes exigerait de faire des preuves mathématiques à la compilation.
  - Les systèmes de types standards offrent un compromis sacrifiant la précision au profit d'une vérification efficace à la compilation.

Règles de conformité des types

- Avant d'attaquer l'héritage des contrats, l'analogie avec le typage incite à regarder les règles de typage en cas de redéfinition de méthodes lors de l'héritage.
- Oublions Java pour un instant pour examiner le problème en général, peu importe le langage à objets.
- Considérons une méthode `m` définie dans une classe `A` et redéfinie dans une classe `B` héritant ($\rightarrow$) de `A` :

Classe	Type de retour	Nom	Paramètre
`A`	`T`	`m(`	`X i )`
$\uparrow$			
`B`	`U`	`m(`	`Y i )`

- Quand peut-on dire que que ces deux définitions sont conformes du point de vue des types ?

Conformité au sens de l'affectation des objets

- Supposons un programme dans lequel on déclare une variable de type `A` *i.e.*, `A var`.
- Le programmeur qui écrit un appel à la méthode `m` de la forme `V t = var.m(a)` va s'assurer (typage statique) :
  1. Que la valeur de `a` est conforme au type `X`, ce qui se traduit par le fait que cette valeur est affectable à une variable de type `X` *i.e.*, une affectation `X i = a` serait valide.
  2. Que le type `V` est conforme au type `T`, ce qui se traduit par le fait que le résultat est affectable à une variable de type `V`.
- Si la valeur affectée à `var` (typage dynamique) est une instance de `B`, c'est la redéfinition de la méthode `m` dans `B` qui est appelée.
- Pour être sûr que l'appel à la redéfinition ne provoque pas d'erreur de type, on doit satisfaire deux conditions :
  1. Une valeur de type `X` est affectable dans une variable de type `Y`.
  2. Une valeur de type `U` est affectable dans une variable de type `T`.

  Or ceci est vrai lorsque `X` hérite de `Y` et `U` hérite de `T`

Covariance et contravariance

- Si nous intégrons le résultat de cette déduction à notre schéma, on obtient :

Classe	Type de retour	Nom	Paramètre
`A`	`T`	`m(`	`X i )`
↑	↑		↓
`B`	`U`	`m(`	`Y i )`

- Ce schéma, qui *garantit statiquement* la conformité des types lors de la redéfinition, est dit :
  - *contravariant* dans les paramètres, parce que la flèche d'héritage entre `X` et `Y` va dans le *sens contraire* de celle entre `B` et `A` ;
  - *covariant* dans le résultat, parce que la flèche d'héritage entre `U` et `T` va dans le *même sens* que celle entre `B` et `A`.

Mais en pratique ?

- Malgré son intérêt théorique, ce schéma n'est pas si courant ni dans les programmes ni dans les langages :
  - Java préfère la règle d'*invariance* dans les paramètres pour interpréter comme surcharge de `m` le fait d'utiliser des types différents.
- Il est aussi considéré comme contre-intuitif dans plusieurs cas :
  - Considérons un exemple correct sur les nombres à la Java :

Classe	Type de retour	Nom	Paramètre
`A`	`Number`	`m(`	`Integer i )`
↑	↑		↓
`B`	`Double`	`m(`	`Number i )`

  - On souhaiterait redéfinir la méthode pour des types plus précis, mais comment l'interpréter ? redéfinition ? surcharge ?
  - Les langages où la sélection de la méthode se fait uniquement sur la classe de l'objet receveur de l'appel rendent la redéfinition de méthodes du genre « opérateurs » n-aires insatisfaisante (solutions potentielles : « multiple dispatch », multi-méthodes).

Conditions d'héritage des axiomes

- Le détour par les règles de typage lors de la redéfinition et la vision ensembliste permettant d'unir typage et axiomes nous donne une bonne intuition des règles d'héritage des contrats :
  - Les préconditions posées sur la méthode $\{P_{\mathtt{X}}\}$ `A::m(X)` doivent être plus restrictives que celles posées sur $\{P_{\mathtt{Y}}\}$ `B::m(Y)` pour que le type (`X`, $P_{\mathtt{X}}$) définisse un sous-ensemble du type (`Y`, $P_{\mathtt{Y}}$), donc $P_{\mathtt{X}} \Rightarrow P_{\mathtt{Y}}$.
  - Les postconditions posées sur la méthode `B::m(U)` $\{Q_{\mathtt{U}}\}$ doivent être plus restrictives que celles posées sur `A::m(T)` $\{Q_{\mathtt{T}}\}$ pour que le type (`U`, $Q_{\mathtt{U}}$) définisse un sous-ensemble du type (`T`, $Q_{\mathtt{T}}$), donc $Q_{\mathtt{U}} \Rightarrow Q_{\mathtt{T}}$.
- Le cas des invariants s'obtient par un raisonnement similaire :
  - Puisque c'est `B::m` qui va s'exécuter, c'est son code qui va devoir garantir que l'invariant définit dans `B` sera maintenu.
  - On en déduit que si $Inv_{\mathtt{B}}$ est plus restrictif que $Inv_{\mathtt{A}}$, le maintien d'$Inv_{\mathtt{B}}$ assurera *de facto* le maintien d'$Inv_{\mathtt{A}}$, donc $Inv_{\mathtt{B}} \Rightarrow Inv_{\mathtt{A}}$.

Conformité entre contrats hérité et redéfini

- Les conditions auxquelles nous sommes arrivés permettent d'assurer que l'héritage des contrats sera bien un *raffinement* du contrat de `A` par le contrat de `B` (en généralisant un peu) :

  $Inv_{\mathtt{B}} \Rightarrow Inv_{\mathtt{A}}$ $(\forall m)\ Pre_{\mathtt{A::m}} \Rightarrow Pre_{\mathtt{B::m}}$ $(\forall m)\ Post_{\mathtt{B::m}} \Rightarrow Post_{\mathtt{A::m}}$

  - Cependant, en pratique, *prouver* les implications précédentes est souvent difficile voire indécidable en général.
- Si on ne fait que *vérifier*, une simple transformation appelée patron Percolateur (en Eiffel) permet d'assurer que les implications soient toujours vraies :
  - $Inv'_{\mathtt{B}} \equiv_{def} Inv_{\mathtt{A}} \wedge Inv_{\mathtt{B}}$
  - $(\forall m)\ Pre'_{\mathtt{B::m}} \equiv_{def} Pre_{\mathtt{A::m}} \vee Pre_{\mathtt{B::m}}$
  - $(\forall m)\ Post'_{\mathtt{B::m}} \equiv_{def} Post_{\mathtt{A::m}} \wedge Post_{\mathtt{B::m}}$

  Par simple déduction logique (*modus ponens* et coupure) on voit que ces nouveaux axiomes vont toujours vérifier les implications précédentes ; encore faut-il que le code les assure.

Vérification des contrats raffinés

- La vérification des contrats raffinés suppose, comme dans le cas des contrats offerts et requis, d'être en mesure non seulement de détecter les violations des conditions mais également de détecter les faux raffinements.
- Le patron percolateur est un artifice qui permet d'assurer *logiquement* les conditions de raffinement, mais pas d'assurer la non contradiction entre les conditions (par exemple, la condition combinée $Inv_{\mathtt{A}} \wedge Inv_{\mathtt{B}}$ est-elle vraie ?).
  - Si, après l'exécution de `B::m`, $Inv_{\mathtt{B}}$ est vrai mais pas $Inv_{\mathtt{A}}$ alors la violation est interprétée comme une *erreur de raffinement*.
  - Idem si lors de l'appel $Pre_{\mathtt{A::m}}$ est vraie mais pas $Pre_{\mathtt{B::m}}$ et si au retour $Post_{\mathtt{B::m}}$ est vraie mais pas $Post_{\mathtt{A::m}}$.
- En testant systématiquement (dans le bon ordre) toutes ces conditions dans le corps de `B::m`, on pourra distinguer les violations de conditions par le code des faux raffinements et signaler une exception appropriée à chaque cas.

Plan

1 Types abstraits de données

2 Conception par contrats

3 Contrats sur des propriétés non-fonctionnelles

4 Contrats et héritage

5 Contrats et gestion de la concurrence

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la relation entre les invariants de la programmation contractuelle et la gestion de concurrence.
   - Comprendre comment les contraintes posées par les accès en parallèle aux variables et structures de données liées par des invariants doivent se traduire en politiques d'accès, elles-mêmes mises en œuvre grâce aux outils de synchronisation.
2. Compétences à acquérir
   - Savoir comment analyser les invariants d'une composant pour déterminer les besoins d'exclusion mutuelle et les sections critiques dans son code.
   - Savoir choisir de manière justifiée l'outil de synchronisation approprié pour gérer l'entrée dans les sections critiques d'un code tout en maximisant le parallélisme potentiel.

Liens entre contrats et concurrence

- Nous avons vu au cours 5 que les préconditions, déterminant si une opération peut être exécutée à un instant donné, peuvent être réinterprétées dans un contexte de parallélisme.
  - Pour une collection séquentielle, la non satisfaction d'une précondition est clairement une erreur, alors que pour une collection concurrente elle peut être réinterprétée comme une condition d'attente jusqu'à ce qu'un autre *thread* fasse une action qui permettra de la satisfaire.
    Ex.: un *thread* bloqué sur le dépilement d'un pile vide sera débloqué par un *thread* empilant une valeur sur cette pile.
- L'invariant, pour sa part, s'exprime par une ou plusieurs conditions logiques qui lient des variables et des données entre elles dans une *cohérence conjointe*.
  - Nous avons indiqué au cours 5 que la raison d'être de l'exclusion mutuelle et des sections critiques est de protéger les accès à ces ensembles de variables et données de manière à préserver leur cohérence.

Partitionnement des invariants

- Faire d'un composant un moniteur garantit le respect de ses invariants mais réduit le parallélisme et donc la performance.
  - L'analyse des invariants d'un composant peut faire apparaître des *ensembles disjoints* de variables et structures de données dont les accès peuvent être protégés indépendamment les uns des autres.
- L'idée consiste donc :
  1. À bien définir par des invariants toutes les relations de cohérence entre les variables et données du composant.
  2. À partir des invariants, partitionner les variables et données en ensembles disjoints *i.e.*, pas liés par des invariants communs.
  3. Protéger chacun de ces ensembles de manière distincte.
  4. Déterminer, dans le code, toutes les sections critiques accédant à chacun de ces ensembles pour contrôler l'entrée dans ces dernières par l'acquisition du verrou, alors relâché à la sortie.
- La programmation contractuelle permet de systématiser l'étape de développement visant à assurer la cohérence des accès tout en maximisant le parallélisme potentiel dans l'exécution du code.

Partitionnement des opérations et politiques d'accès

- La protection d'un invariant requiert l'accès exclusif lorsqu'on modifie mais également lorsqu'on lit, pour éviter de lire des données incohérentes si elles sont en cours de modifications.
  - Il faut donc exécuter en exclusion mutuelle les lectures *et* les écritures sur les données liées par un même invariant.
  - Un simple verrou `ReentrantLock` à acquérir à l'entrée de toutes les sections critiques en lecture ou en écriture permet cela.
    C'est la politique d'exclusion mutuelle totale !
- Toutefois, il n'y a pas de risque à exécuter en même temps plusieurs séquences d'instructions qui lisent des données car elles ne peuvent engendrer aucune incohérence.
  C'est la politique de lectures multiples mais d'écriture unique !
  - On peut autoriser plusieurs lectures en même temps mais les écritures ne doivent pas être retardées indéfiniment.
  - Cet idiome très courant justifie l'existence en Java du `ReentrantReadWriteLock` coordonnant un verrou en lecture et un verrou en écriture pour garantir ces différentes contraintes.

Activités à réaliser avant le prochain TME

1. Identifier les invariants dans vos composants du projet.
2. Documenter les politiques d'accès aux données impliquées dans ces invariants et les moyens de la mise en œuvre de ces politiques (verrous, structures de données concurrentes, code en exclusion mutuelle, etc.).