TD GLA 1 et 2

Invariants and Contracts over a Bank Model

Bu 21.9.20

- Trouver les invariants exprimés explicites et implicites dans ce ``cahier de charge'' (en francais).
- Formuler en MOAL les invariants qui n'apparaissent pas déjà dans le diagramme.

```
subsection‹Uniqueness constraints›
text‹⬚‹
   definition  inv_Account1 (σ) ≡  is_Unique(.number(σ))(Account(σ))

     (*  ou plus court, en applicant la syntactic convention: *)

   definition inv_Account1 ≡  is_Unique(.number)(Account)

     (* Une variante qui exprime que le ∗‹pair› de IBAN et BIC
        est unique, s'exprime comme suivant *)

   definition get_pair (a::Account) ≡ (a.number, a.bank.id)
   definition inv_Account1' ≡  is_Unique(get_pair)(Account)

     (*  Analogously for all the different cases, we construct *)

   definition  inv_Bank1 ≡  is_Unique(.title)(Bank)

   definition  inv_Bank2 ≡ is_Unique(.id)(Bank)
›
```

- Trouver les invariants exprimés explicites et implicites dans ce ``cahier de charge'' (en francais)

```
subsection‹Basic data constraints›
text‹
 eg. Le plafond des comptes épargne est positif.

  ⬚‹definition inv_Savings1(σ) ≡  ∀ a ∈ Savings(σ).  a.ceiling (σ) > 0›

 en appliquant la σ-convention:

  ⬚‹definition inv_Savings1 ≡  ∀ a ∈ Savings. a.ceiling > 0›

Analogously: Transfer.amount, max_overdraft, overdraft_interest, etc. etc.

›
```

```
subsection‹Slightly more evolved data-constraints›

text‹
▸ Un client peut être titulaire d'un ou plusieurs comptes
  *‹dans une banque› (dans le diagramme). Rappel: Ce genre
  de contraintes est exprime en MOAL par:

      ⬚‹definition inv_Account_bank ≡  ∀ a∈Account. | a.bank | = 1›

▸ Un compte est hébergé par une unique banque (dans le diagramme).

      ⬚‹definition inv_Account_bank ≡  ∀ a∈Account. | a.bank | = 1›

▸ Un compte ne peut appartenir qu'à un seul client (dans le diagramme).
  Ce contrainte, est-ce realiste, d'ailleurs ? (Non, par la loi,
  plusieurs personnes peuvent avoir controle sur un compte....)

▸ Le solde du compte doit toujours être inférieur ou égal au plafond.

      ⬚‹definition inv_Savingsceiling ≡  ∀ s∈Savings. s.balance ≤ s.ceiling›

▸ Le découvert est interdit sur les comptes épargne.

    ⬚‹definition inv_Savingsbalance ≡  ∀ s∈Savings. s.balance ≥ 0›
›
```

```
text‹
  Les titulaires de comptes chèques âgés de moins de 25 ans ont un découvert autorisé de 250 ...

   ⬚‹definition invCheckingovdft1 ≡
                ∀ c ∈ Checking.       (c.bank.global_time).before(c.owner.born.add_months(25*12))
                                  ⟶   c.overdraft_max = -250.0
    ›

  This makes global timing consistency necessary see below.


  Les titulaires de comptes chèques âgés de plus de 25 ans ont un découvert autorisé égal à 500.

    ⬚‹definition invChecking_ovdft2  ≡
                ∀ c ∈ Checking.     ¬(c.bank.global_time).before(c.owner.born.add_months(25*12))
                                  ⟶   c.overdraft_max = -500.0›

  (IMPLICIT!) consistence de temps est necessaire: staff peut seulement etre embauche apres sa
  naissance, et le global_time  doit etre apres les dates de naissance et dates d'embauche, dates
  des transfers doivent être apres la date de naissance ...

    ⬚‹definition invBornpast ≡
                ∀ b ∈ Bank.  ∀ c ∈ b.client.  (c.born) .before(b.bank.global_time)›
```

Chaque client d'une banque doit y posseder un compte:

⬚‹definition $inv_{AcountConsistency1} \equiv$
$\forall\ b \in Bank.\ \ \forall\ c \in b.client.\ \exists\ a \in Accounts.\ a.owner = c \wedge a.bank = b$›

Alternative a l'implication: filtrage des Accounts sur ceux de la banque.

Tout propriétaire d'un compte est client de la banque hébergeant ce compte.

⬚‹definition $inv_{AcountConsistency2} \equiv$
$\forall\ a \in Account.\ a.owner \in\ \ a.bank.client$›

Dans chaque banque, un client ne peut être titulaire que d'un seul compte chèque.

⬚‹definition $inv_{AcountConsistency3} \equiv$
$\forall\ b \in Bank.\ \forall\ c \in b.clients$
$|\{\ a \in Accounts\ |\ a.banque = b \wedge a.owner = c \wedge a \in Checking\}| \leq 1$›

Un diagramme de classe est ``open world'': cela veut dire mentionner les classes ⬚‹Savings› et ⬚‹Checking› exige que ce classes existent, mais n'excluent pas l'existence d'autres. Consequemment, ceci est necessaire pour capturer l'intention du cahier de charge:
``Un compte est soit un compte chèque, soit un compte épargne'':

⬚‹definition $inv_{AcountConsistency4} \equiv\ \forall\ a \in Accounts.\ a \in Savings\ \vee\ a \in Checking$ ›

ou bien alternativement:

⬚‹definition $inv_{AcountConsistency4} \equiv (\ Accounts\ =\ Savings \cup Checking\ )$ ›

. . .›

- TD2 : contracts

```
text‹⬚‹
    definition identity X ≡ X
    definition moins25ans(c) ≡ ... (* compare bank clock with client birthday. *)

    (* un peu en "high-level-notation" *)
    Checking create_account(c:: Client, b :: Bank, i :: IBAN, cc: Currency, ovi::FPN)
    pre   isUnique(identity)({i ∈ IBAN | ∃ a ∈ Account. i = a.number ∧ i.bank=b} ∪ {i}) ∧
          c ∉ {ccc∈Client | ∃a∈Checking. a.bank = b ∧ a.owner = ccc}

    post  isNew(result) ∧ result.number=i ∧ result.bank=b ∧ result.owner=c
          ∧ result.currency=cc ∧ result.balance=0
          ∧ if moins25ans(c)
            then result.max_overdraft = 250
            else result.max_overdraft = 500
          ∧ result.overdraft_interest = ovi ›
›
    or stripped off the pre-post notation and representing its semantics in bare-bone logics:
    (we assume that the generator create returns the object as result value in this version,
     this is not clearly defined in UML.)
       ⬚‹
        definition pre_create_account(c:: Client, b :: Bank, i :: IBAN, cc: Currency, ovi::FPN) ≡
              isUnique(identity)({(i,_) ∈ IBAN×BIC | ∃ a ∈ Account. i = a.number ∧ i.bank=b}
                                 ∪ {(i,}) ∧
              c ∉ {ccc∈Client | ∃a∈Checking. a.bank = b ∧ a.owner = ccc}
        definition postcreate_account(c:: Client, b :: Bank, i :: IBAN, cc: Currency,
                                      ovi::FPN,result, savings::Boolean) ≡
              isNew(result) ∧ result.number=i ∧ result.bank=b ∧ result.owner=c
              ∧ result.currency=cc ∧ result.balance=0 ∧ savings = (result ∈ Savings)
              ∧ if moins25ans(c)
                then result.max_overdraft = 250
                else result.max_overdraft = 500
              ∧ result.overdraft_interest = ovi
        ›
```

- TD2 : contracts

```
definition moins25(b::Bank,c::Client) ≡ ((c.born).add_month(25*12))
                                        .before(b.global_time)

(* version without an explicit result value for the constructor *)

create_account(b::Bank,name::String, n::IBAN,c::Currency, ceil:FPN, savings::Boolean)
pre  ∃ c ∈ Client. c.name = name
    ∧ (b.id, n) ∉ ({bic | ∃b∈Bank. b.id = bic} × {iban | ∃a∈Account. a.number=iban})
    ∧ ¬savings ⟶  {c∈Checking | c.owner.name = name} = {}
    ∧ ¬savings ⟶ |{c∈Checking | c.owner.name = name}| = 0

post if savings
     then ∃ a ∈ Savings.  isNew(a) ∧ a.currency = c ∧
                          a.balance = 0 ∧ a.number = n ∧ a.bank = b ∧
                          a.owner.name = name ∧ a.ceiling = ceil
     else ∃ a ∈ Checking. isNew(a) ∧ a.currency = c ∧
                          a.balance = 0 ∧ a.number = n ∧ a.bank = b ∧
                          if moins25(b,a.owner)
                          then a.max_overdraft = 250.00
                          else a.max_overdraft = 500.00
```

- TD2 : contracts

```
definition pre_create_account(b::Bank,name::String, n::IBAN,c::Currency, ceil:FPN, savings::Boolean)≡
          ∃ c ∈ Client. c.name = name
                  ∧ (b.id, n) ∉ ({bic | ∃b∈Bank. b.id = bic} × {iban | ∃a∈Account. a.number})
                  ∧ ¬savings ⟶  {c∈Checking | c.owner.name = name} = {}
                  ∧ ¬savings ⟶ |{c∈Checking | c.owner.name = name}| = 0

definition post_create_account(b::Bank,name::String, n::IBAN,c::Currency,
                        ceil:FPN, savings::Boolean,result) ≡
          if savings
          then ∃ a ∈ Savings.  isNew(a) ∧ a.currency = c ∧
                               a.balance = 0 ∧ a.number = n ∧ a.bank = b ∧
                               a.owner.name = name ∧ a.ceiling = ceil
          else ∃ a ∈ Checking. isNew(a) ∧ a.currency = c ∧
                               a.balance = 0 ∧ a.number = n ∧ a.bank = b ∧
                               if moins25(b,a.owner)
                               then a.max_overdraft = 250.00
                               else a.max_overdraft = 500.00
›
text‹
⬚‹
definition pre_balance (a) ≡ true
definition post_balance (a::Account,result::FPN) ≡ (result = a.balance)
›
›
```

```
text‹⬚‹
definition pre_withdraw(a::Account, m::FPN) ≡
            a.montant≥0 ∧
       (*   simpler variant excluding exceptional behaviour (no need for result):
            if a ∈ Savings
           then old(a.withdraw_authorized) ∧ old(a.balance)-m ≥ 0
           else (old(a.balance)-m ≥ 0)
                ∨ (old(a.balance) - m - old(a.overdraft_interest) ≥ max_overdraft) *)

definition post_withdraw(c::Account, m::FPN,result) ≡
            if c ∈ Savings
            then if c.withdraw_authorized ∧ old(c.balance) ≥ montant
                 then c.balance = old(c.balance) - montant
                      ∧ result=true
                 else result=false ∧ c.balance = old(c.balance)
            else if  old(c.balance) - montant < 0
                 then if old(c.balance) - montant- c.overdraft_interest  > c.max_overdraft
                      then result = true ∧
                           c.balance = old(c.balance) - montant - c.overdraft_interest
                      else result = false ∧
                           c.balance = old(c.balance)
                 else result = true ∧
                      c.balance = old(c.balance) - montant
            ∧ modifiedOnly({c})›
}
›
```

```
text‹
⬚‹
definition pre_deposit(a::Account,amount::FPN)
           ≡ old(a.amount)≥0 ∧ (a ∈ Savings ⟶ old(a.balance) + amount ≤ a.ceiling)
definition post_deposit(a::Account,amount::FPN,result::unit) ≡
           if a ∈ Savings
           then a.balance = old(a.balancae) + amount
           else if old(a.balance) + amount < 0
                then a.balance = old(a.balance) + amount - a.overdraft_interest
                else a.balance = old(a.balance) + amount
           ∧ modifiesOnly({a})
›
›
```

(* Meta-commentaire : l'annotation ::IBAN, ::BIC etc est une annotation avec un typ.
En general, l'inference de type dans un systeme comme Isabelle ou OCaml permet
d'inferer ce type automatiquement. Dans notre contexte, cette annotation peut etre
lu comme un commentaire *)

```
text‹
⬚‹
definition get_account(i::IBAN, bic::BIC) ≡
                    hd([x ∈ Account | x.number=i ∧ x.bank.id=bic])
                    (* any Account has a unique pair of IBAN and BIC,
                       so there is only one element in this list *)
definition pre_exec_transfer(t) ≡
           t.date = t.account.bank.global_time
           ∧ ¬ t.transferred
           ∧ pre_withdraw(t.account,t.amount)
           ∧ pre_deposit(get_account(t.target, t.target_bank),t.amount)

definition post_exec_transfer(i::IBAN, bic::BIC, result) ≡
           t.transferred = true
           ∧ post_withdraw(t.account,t.amount)
           ∧ post_deposit(get_account(t.target, t.target_bank),t.amount)
           ∧ modifiesOnly({t,get_account(t.target, t.target_bank), t.account})

›
›
```