UPMC Paris Universitas – Master Informatique – STL

Cours Composant
4. Conception par Contrat II
Aspects avancés

UPMC Paris Universitas

18 février 2013

Plan du cours

Rappels : Conception par contrat

La **conception par contrat** (ou programmation par contrat) encourage les concepteurs de logiciel à spécifier, de façon **vérifiable**, les interfaces de composants logiciels.

1. Spécifications semi-formelles des services
2. Définition des contrats de service
3. Implémentation des fournisseurs
4. Code de vérification (manuel/automatique)
5. Tests

Spécifications et contrats (ex. Light)

Contrats génériques

Il est possible de spécifier des services paramétrés en fonction de types ou d'autres services.

Exemple : une pile générique

Service : Stack<T>
Types : boolean, int, List<T>
Observators :
    getTop : [Stack<T>] $\rightarrow$ T
        precondition top(S) **require** $\neg$isEmpty(S)
    getElems : [Stack<T>] $\rightarrow$ List<T>
    getSize : [Stack<T>] $\rightarrow$ int
    isEmpty : [Stack<T>] $\rightarrow$ boolean
Constructors :
    init : $\rightarrow$ [Stack<T>]
Operators :
    push : [Stack<T>] $\times$ T $\rightarrow$ [Stack<T>]
    pop : [Stack<T>] $\rightarrow$ [Stack<T>]
        precondition pop(S) **require** $\neg$isEmpty(S)

Exemple : pile générique (suite)

Observations :
[invariants]
getTop(S) = get(getElems(S),size(getElems(S)))
getSize(S) = size(getElems(S))
isEmpty(S) = (getSize(S)=0)
[init]
getElems(init(S)) = $\emptyset$
[push]
$\forall 1 \leq i \leq$size(getElems(S)), get(getElems(push(S,e)),i) = get(getElems(S),i)
getTop(push(S,e)) = e
[pop]
$\forall 1 \leq i <$ size(getElems(S)), get(getElems(pop(S),i)) = get(getElems(S,i))
size(getElems(pop(S)) = size(getElems(S)) - 1

New Observator : PoidsTotal

Contrats et héritage

Héritage

Concept fondamental de P.O.O

- Utilisé dans de nombreux contextes :
  Classification, sous-typage (héritage d'interface), spécialisation, extesion
- Polymorphisme
  - Compatibilité de type : **héritage d'interface** ⇒ « EST UN » (subsomption)
  - Compatibilité sémantique : **héritage d'implémentation** ⇒ « EST SUBSTITUABLE PAR » (substituabilité)

⇒ héritage source de nombreux problèmes architecturaux et autres bugs !
Il ne suffit pas d'hériter, voir ex. EIFFEL

Contrats et héritage

Quelles contraintes sur les invariants, préconditions et posconditions ?

Définitions

Definition (Précondition (resp. postcondition))

- Une précondition (resp. postcondition) est locale à la déclaration.
- une précondition déclarée dans le super est transmis à tous les héritiés

Definition (Précondition effective (resp. postcondition effective))

La précondition effective (resp. postcondition effective) est la condition qui sera vérifiée à l'exécution.

Definition (Patron de percolation)

Le patron de percolation indique comment générer la condition effective avec toutes les conditions locales.

Raffinement de spécification

Nous proposons de mettre en œuvre un modèle d'héritage sûr en nous basant sur la relation de raffinement entre spécifications : **refine**

Exemple : Light et ColorLight

Service : Light
Types : boolean
Observators :
    isOn : [Light] $\rightarrow$ boolean
Constructors :
    init : $\rightarrow$ [Light]
Operators :
    switchOn : [Light] $\rightarrow$ [Light]
        precondition switchOn(L) **require** $\neg$isOn(L)
    switchOff : [Light] $\rightarrow$ [Light]
        precondition switchOff(L) **require** isOn(L)
Observations :
[invariants]
[init]
    isOn(init()) = false
[switchOn]
    isOn(switchOn(L)) = true
[switchOff]
    isOn(switchOff(L)) = false

Exemple : raffinement (suite)

Service : ColorLight
Types : enum Color { RED, ORANGE, GREEN }

Refine : Light

Observators :

getColor : [ColorLight] → Color
isBlinking : [ColorLight] → boolean
precondition isBlinking(L) **require** Light.isOn(L)

Constructors :

init : → [ColorLight]

Operators :

change : [ColorLight] → [ColorLight]
precondition change(L) **require** Light.isOn(L) ∧¬ isBlinkling(L)
blinkMode : [ColorLight] → [ColorLight]
precondition blink(L) **require** Light.isOn(L) ∧ getColor(L)=ORANGE

Exemple : raffinement (suite)

Observations :

[invariants]

isBlinking(L) $\Longrightarrow$ getColor(L)=ORANGE

[init]

Light.isOn(init()) = false

getColor(init())= ORANGE

isBlinking(init()) = true

[switchOn]

getColor(switchOn(L)) = ORANGE

isBlinking(switchOn(L)) = true

[switchOff]

[change]

Light.isOn(change(L)) = Light.isOn(L)

getColor(L)=RED $\Longrightarrow$ getColor(change(L))=GREEN

getColor(L)=GREEN $\Longrightarrow$ getColor(change(L))=ORANGE

getColor(L)=ORANGE $\Longrightarrow$ getColor(change(L))=RED

[blinkMode]

Light.isOn(blinkMode(L)) = Light.isOn(L)

getColor(blinkMode(L)) = ORANGE

isBlinking(L) $\Longrightarrow$ $\neg$ isBlinking(blinkMode(L)) = false

$\neg$isBlinking(L) $\Longrightarrow$ isBlinking(blinkMode(L))

Propriétés du raffinement

S' raffine S

- tous les observateurs et opérateurs de S sont « hérités » par S'
  $\Rightarrow$ il est possible de les modifier
- les constructeurs ne sont pas hérités (mais on peut utiliser les constructeurs de S pour décrire les observations de S')
- on peut ajouter des observateurs et opérateurs dans S'
- les observations non raffinées dans S' sont implicites

Correction ?

- cas simple : intersection vide entre S et $S' \setminus S$ (extension orthogonale)
- cas complexe : extension non-orthogonale

Extensions non-orthogonales

Cas à considérer

- Ajout d'un opérateur
  - Cohérence des observations
- Raffinement d'un opérateur existant
  - Modification d'une précondition
  - Modification d'une observation (invariant ou postcondition)

Raffinement de fonctions

Soit une fonction fun : $T \rightarrow U$ (ex. : fun : $\mathbb{R} \rightarrow \mathbb{R}$)

- $dom(fun) = T$ (ex. : $\mathbb{R}$)
- $cod(fun) = U$ (ex. : $\mathbb{R}$)

Une fonction rfun : $T' \rightarrow U'$ raffine fun ssi

Raffinement de fonctions

Soit une fonction fun : $T \to U$ (ex. : fun : $\mathbb{R} \to \mathbb{R}$)

- $dom(fun) = T$ (ex. : $\mathbb{R}$)
- $cod(fun) = U$ (ex. : $\mathbb{R}$)

Une fonction rfun : $T' \to U'$ raffine fun ssi

- $dom(fun) \subseteq dom(rfun)$
- $cod(rfun) \subseteq cod(fun)$

Raffinement de fonctions

Soit une fonction fun : $T \to U$ (ex. : fun : $\mathbb{R} \to \mathbb{R}$)

- $dom(fun) = T$ (ex. : $\mathbb{R}$)
- $cod(fun) = U$ (ex. : $\mathbb{R}$)

Une fonction rfun : $T' \to U'$ raffine fun ssi

- $dom(fun) \subseteq dom(rfun)$
- $cod(rfun) \subseteq cod(fun)$

Exercices :

- rfun : $\mathbb{R} \to \mathbb{N}$?
- rfun : $\mathbb{N} \to \mathbb{R}$?
- rfun : $\mathbb{C} \to \mathbb{N}$?
- rfun : $\mathbb{N} \to \mathbb{C}$?

L'entonnoir encore ?

Raffinement de fonctions partielles

Un observateur, constructeur ou opérateur est une **fonction partielle** :

fun : $T_1 \times \ldots \times T_n \to T$

- avec la précondition $pre(t_1 : T_1, \ldots, t_n : T_n)$ on a $dom(fun) = \{(v_1, \ldots, v_n) \mid pre(v_1, \ldots, v_n)\}$
- avec l'observation $obs(t_1 : T_1, \ldots, t_n : T_n, t : T)$ on a $cod(fun) = \{v \mid \forall v_1, \ldots, v_n,\ obs(v_1, \ldots, v_n, v)\}$

Une fonction rfun raffine fun (rfun $\subseteq$ fun) ssi

- dom(fun) $\subseteq$ dom(rfun)
- cod(rfun) $\subseteq$ cod(fun)

Traduction :

- pre$(v_1, \ldots, v_n) \implies$ rpre$(v_1, \ldots, v_n)$
- robs$(v_1, \ldots, v_n, v) \implies$ obs$(v_1, \ldots, v_n, v)$

Modification d'une précondition

Service : S
Operators :
op : $[S] \rightarrow [S]$
precondition $op(s)$ **require** P

Service : S' **Refine** : S
Operators :
op : $[S] \rightarrow [S]$
precondition $op(s)$ **require** P'

Modification d'une précondition

Service : S
Operators :
op : $[S] \rightarrow [S]$
precondition $op(s)$ **require** P

Service : S' **Refine** : S
Operators :
op : $[S] \rightarrow [S]$
precondition $op(s)$ **require** P'

Contrainte $P \implies P'$

Modification d'une postcondition

Service : S
Observators :
obs : $[S] \rightarrow T$
Operators :
op : $[S] \rightarrow [S]$
Observations :
[op]
O

Service : S' **Refine** : S
Operators :
op : $[S] \rightarrow [S]$
Observations :
[op]
O'

Contrainte $O' \implies O$

Conjonctions et disjonctions

Conditions de raffinement

- $pre \implies rpre$
- $robs \implies obs$

Remarques

- Si $rpre \stackrel{def}{=} pre \vee pre'$ alors $pre \implies rpre$
- Si $robs \stackrel{def}{=} obs \wedge obs'$ alors $robs \implies obs$

Héritage dans les contrats

Passage spécifications → contrats

- Preconditions → préconditions de méthodes
- Observations (section invariants) → invariants de classe
- Observations (autres) → postconditions de méthodes

Soit un contrat de classe $C : \langle Inv_C,\ M_C \rangle$ avec contrats de méthode $m : \langle pre_m, post \rangle \in M_C$.
Soit un contrat de classe $C' : \langle Inv_{C'},\ M_{C'} \rangle$ avec contrats de méthode $m : \langle pre'_m, post'_m \rangle \in M_{C'}$ (+ extensions)

Conditions de Liskov

C' raffine C si et seulement si :

- $Inv_{C'} \implies Inv_C$
- $pre_m \implies pre'_m$
- $post'_m \implies post_m$

Héritage dans Eiffel

Problème : $P \implies Q$ n'est pas décidable dans le cas général
Soit un contrat de classe $C : \langle Inv_C,\ M_C \rangle$ avec contrats de méthode $m : \langle pre_m, post \rangle \in M_C$.
Soit un contrat de classe $C' : \langle Inv_{C'},\ M_{C'} \rangle$ avec contrats de méthode $m : \langle pre'_m, post'_m \rangle \in M_{C'}$.

Percolator pattern : plugin

On doit implanter le contrat $C'' : \langle Inv_{C''},\ M_{C''} \rangle$ avec $m : \langle pre''_m, post''_m \rangle \in M_{C''}$.

- $Inv_{C''} \stackrel{def}{=} Inv_C \wedge Inv_{C'}$
- $pre''_m \stackrel{def}{=} pre_m \vee pre'_m$
- $post''_m \stackrel{def}{=} post_m \wedge post'_m$

Exemple : piles et piles bornées

BoundedStack<T> est une pile (Stack<T>) de capacité limitée

Question : faut-il faire hériter

- Stack<T> de BoundedStack<T> ?
- BoundedStack<T> de Stack<T> ?

Héritage et composants require/provide

Question

Un composant C requiert un ensemble de services R et fournit un ensemble de services F : on note $C : \langle R, F \rangle$
Quelles sont les conditions pour qu'un composant $C' : \langle R', F' \rangle$ soit substituable à C ?

Héritage et composants require/provide

Question

Un composant C requiert un ensemble de services R et fournit un ensemble de services F : on note $C : \langle R, F \rangle$
Quelles sont les conditions pour qu'un composant $C' : \langle R', F' \rangle$ soit substituable à C ?

- Tout client de F peut utiliser C' donc F' raffine F
- Tout fournisseur de R doit être « branchable » sur C' donc R raffine R'

Fin