UPMC Paris Universitas – Master Informatique – STL

Cours Composant
3. Conception par Contrat

UPMC Paris Universitas

4 février 2013

Plan du cours

1. Conception par Contrat
   - Métaphore du service
   - Triplets de Hoare
   - Des spécifications aux contrats
2. Implantation des contrats
3. Vérification

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.
Elle est originellement basée sur les types de données algébriques (ADT) et la métaphore conceptuelle des contrats industriels.

Référence

- **Livre** : Bertrand Meyer, *Conception et programmation objet*. 2005, Eyrolles
- **Site** : `http://www.eiffel.com`

Métaphore du service

Service

La notion de **service** est omniprésente dans l'industrie en général, et l'informatique en particulier. Cette notion est au centre de la conception par contrat.
Un **service** est proposé par un **fournisseur** à son (ou ses) **client**(s)

Le **contrat** de service indique :

- les conditions que doit remplir un client pour obtenir ce sevice
  $\Rightarrow$ ce sont les **prérequis** ou **préconditions** du service, imposées au client
- les conditions offertes par le fournisseur au client
  $\Rightarrow$ ce sont les **garanties** ou **postconditions**, imposées au fournisseur à condition que le client respecte les prérequis.

Exemple de contrat

Service : « voyage en train de Paris à Lyon sur le TGV XXX »

Exemple de contrat

Service : « voyage en train de Paris à Lyon sur le TGV XXX »

Prérequis :

- « acheter son billet »
- « composter son billet »
- « monter dans le train avant l'heure du départ »
- « s'asseoir à la bonne place »

Exemple de contrat

Service : « voyage en train de Paris à Lyon sur le TGV XXX »

Prérequis :

- « acheter son billet »
- « composter son billet »
- « monter dans le train avant l'heure du départ »
- « s'asseoir à la bonne place »

Garanties :

- « départ à l'heure indiquée »
- « voyage sûr et agréable »
- « arrivée à l'heure prévue »

Exemple de contrat

Service : « voyage en train de Paris à Lyon sur le TGV XXX »

Prérequis :

- « acheter son billet »
- « composter son billet »
- « monter dans le train avant l'heure du départ »
- « s'asseoir à la bonne place »

Garanties :

- « départ à l'heure indiquée »
- « voyage sûr et agréable »
- « arrivée à l'heure prévue »

+ gestion du **risque**

Validation du contrat

Un contrat peut-être :

honoré par le fournisseur
$\Rightarrow$ le client peut exploiter les garanties du fournisseur (ex. le client est bien arrivé à destination, à l'heure).

rompu par le client car il n'assure pas les prérequis
$\Rightarrow$ le fournisseur ne fournit pas les garanties (ex. le client rate le train, ou ne paye pas son billet)

rompu par le fournisseur car il n'offre pas les garanties
$\Rightarrow$ le client peut demander des compensations (ex. le train arrive en retard)

Contrats logiciels

Dans le logiciel :

Service	classe (ou interface)
Fournisseur	concepteur/implémenteur de la classe
Client	utilisateur de la classe (ex. autre classe, main, etc.)
Contrats	contrainte sur l'utilisation de la classe et ses méthodes

- **propriétés observables**
- **invariants** de classe
- pour chaque méthode
  - **préconditions** d'invocation
  - **postconditions** après exécution

Triplets de Hoare

La base théorique des contrats logiciels est la **logique de Hoare** (cf. cours 6 et 7).

Triplet de Hoare

$\{ P \}\ C\ \{ Q \}$

précondition P formule de logique

code C fragment de code source

postcondition Q formule de logique

Interprétation

Si P est vraie (précondition vérifiée) et si l'on exécute C alors Q doit-être vraie (postcondition vérifiée).

En conception par contrat, le code C est le corps d'une méthode

Méthodologie (I)

Pour concevoir un contrat de classe, il faut :

- déclarer les propriétés observables
- définir l'invariant de classe
- pour chaque méthode, décrire ses préconditions et postconditions

Comment ?

- approche « intuitive »
- approche « encadrée » : à partir des spécifications

Des spécifications aux contrats

Démarche

Analyse	$\rightarrow$	Conception
Spécification	$\rightarrow$	Contrat

Service : *nom du service spécifié*
Types : *dépendances des types élémentaires*
Require : *dépendances de services*
Observators :
fonctions d'observation de l'état
...
Constructors :
fonctions de construction
...
Operators :
fonctions de modification
...
Observations :
Axiomes d'observation
...

Passage des spécifications aux contrats

- **Service** $\rightarrow$ classe ou interface
- **Observators** $\rightarrow$ propriétés observables
- **Operators** $\rightarrow$ méthodes et préconditions
- **Observations** $\rightarrow$ invariant et postconditions

Exemple : interrupteur

Service : Switch
Types : boolean
Observators :
    isOn : [Switch] $\rightarrow$ boolean
    isOff : [Switch] $\rightarrow$ boolean
    isWorking : [Switch] $\rightarrow$ boolean
Constructor :
    init : $\rightarrow$ [Switch]
Operators :
    switch : [Switch] $\rightarrow$ [Switch]
        pre switch(S) **require** isWorking(S)
Observations :
[invariants]
    isOff(S) = $\neg$isOn(S)
[init]
    isOn(init()) = false
[switch]
    isOn(switch(S)) = $\neg$isOn(S)

Remarque : sous-spécification (observations de isWorking)

Exemple : propriétés observables

Service : Switch
Types : boolean
Observators :
- isOn : [Switch] → boolean
- isOff : [Switch] → boolean
- isWorking : [Switch] → boolean

Constructor :
- init : → [Switch]

Operators :
- switch : [Switch] → [Switch]
  pre switch(S) **require** isWorking(S)

Observations :
[invariants]
- isOff(S) = ¬isOn(S)

[init]
- isOn(init()) = false

[switch]
- isOn(switch(S)) = ¬isOn(S)

Propriétés observables : isOn, isOff, isWorking

Exemple : invariants

Service : Switch
Types : boolean
Observators :
isOn : [Switch] $\rightarrow$ boolean
isOff : [Switch] $\rightarrow$ boolean
isWorking : [Switch] $\rightarrow$ boolean
Constructor :
init : $\rightarrow$ [Switch]
Operators :
switch : [Switch] $\rightarrow$ [Switch]
pre switch(S) **require** isWorking(S)
Observations :
[invariants]
$\boxed{\textbf{isOff(S)} = \neg\textbf{isOn(S)}}$
[init]
isOn(init()) = false
[switch]
isOn(switch(S)) = $\neg$isOn(S)

Invariants : $\forall S$, isOff(S) = $\neg$isOn(S)

Exemple : préconditions

Service : Switch
Types : boolean
Observators :
isOn : [Switch] $\rightarrow$ boolean
isOff : [Switch] $\rightarrow$ boolean
isWorking : [Switch] $\rightarrow$ boolean
Constructor :
init : $\rightarrow$ [Switch]
Operators :
switch : [Switch] $\rightarrow$ [Switch]
pre switch(S) require isWorking(S)
Observations :
[invariants]
isOff(S) = $\neg$isOn(S)
[init]
isOn(init()) = false
[switch]
isOn(switch(S)) = $\neg$isOn(S)

Contrat de la méthode switch :

Précondition : isWorking(S)

Exemple : postconditions

Service : Switch
Types : boolean
Observators :
isOn : [Switch] $\rightarrow$ boolean
isOff : [Switch] $\rightarrow$ boolean
isWorking : [Switch] $\rightarrow$ boolean
Constructor :
init : $\rightarrow$ [Switch]
Operators :
switch : [Switch] $\rightarrow$ [Switch]
pre switch(S) **require** isWorking(S)
Observations :
[invariants]
isOff(S) = $\neg$isOn(S)
[init]
isOn(init()) = false
[switch]
isOn(switch(S)) = $\neg$isOn(S)

Contrat de la méthode switch :

Précondition : isWorking(S)

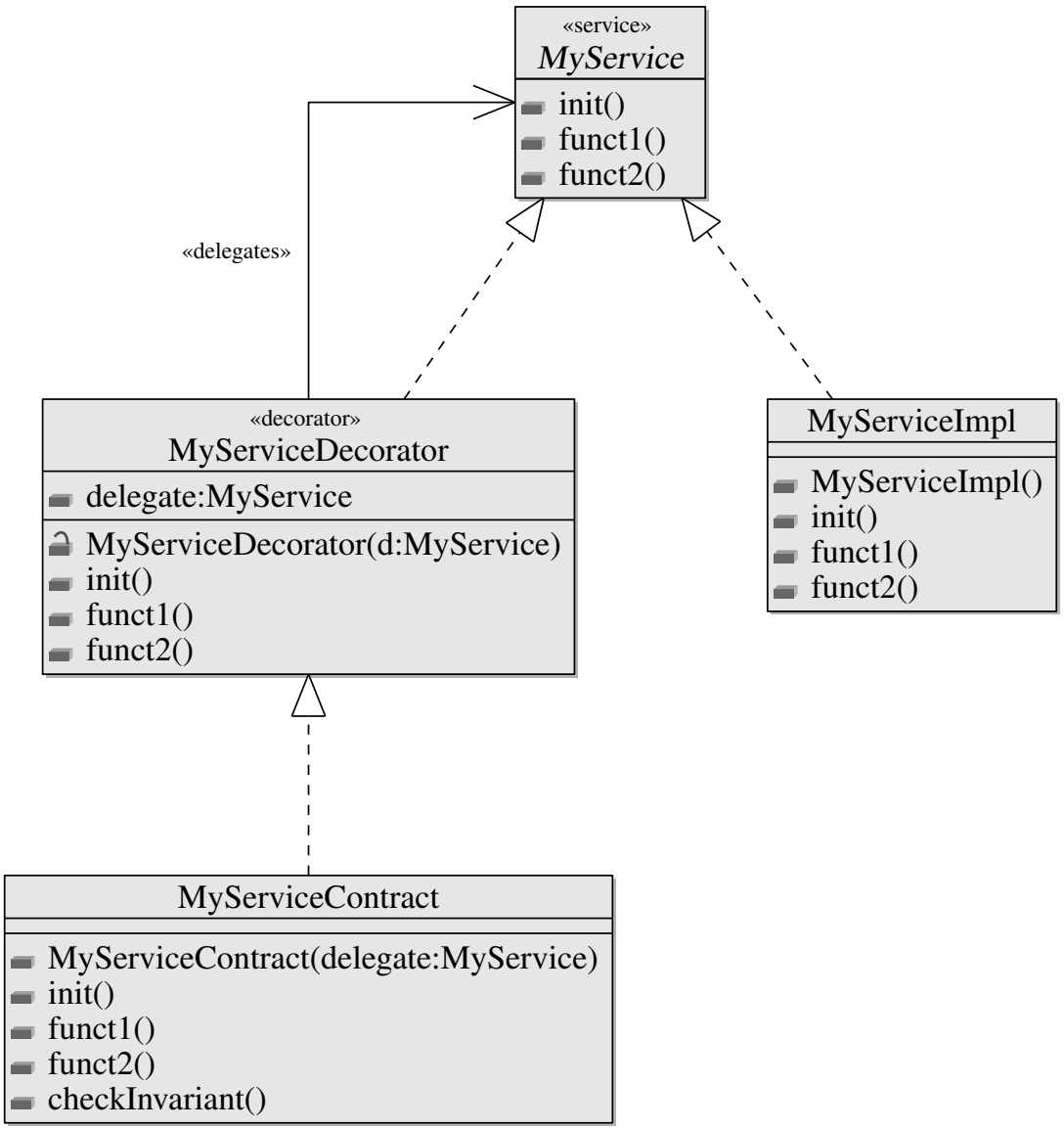
Postcondition : isOn(switch(S)) = $\neg$isOn(S)

Méthodologie (II)

Du contrat à l'implémentation

1. Programmation par contrat « classique »
   $\Rightarrow$ classe contractualisée
2. Contrats de composants
   - Interface de service
   - Classe(s) d'implémentation

Pattern décorateur



Besoins

- Séparation (interface de) service et implémentation(s)
- Séparation code d'implémentation, code de vérification
- Activation/Désactivation par instance de la vérification

$\Rightarrow$ **design pattern décorateur**

Interface de service

```
public interface Switch {
    /* observators */
    public boolean isOn();
    public boolean isOff();
    public boolean isWorking();

    /* invariants */
    // inv : isOn()==not isOff();

    /* Constructors */
    // post : isOn()==false
    public void init();

    /* Operators */
    // switch :
    // pre : isWorking()
    // post : isOn() == not old.isOn()
    public void switch();
}
```

Implémenation(s)

```
public class SwitchImpl implements Switch {
        private boolean state;
        public SwitchImpl() { init(); }
        public void init() { state=false; }
        public boolean isOn() { return state;}
        public boolean isOff() { return !isOn(); }
        public boolean isWorking() { return true; }
        public void switch() { state = !state; }
}
```

Remarque : une implémentation plus « réaliste » considérerait la partie matérielle de l'interrupteur, les possibilités de panne, etc.

Méthodologie (III)

Du contrat implémenté à la vérification

1. Vérification statique $\Rightarrow$ outils
2. Vérification dynamique
   - Décorateur de service
   - Implantation du contrat

Décorateur de service

```
public abstract class SwitchDecorator implements Switch {
    private Switch delegate;
    protected SwitchDecorator(Switch delegate) { this.delegate = delegate }
    public boolean isOn() { return delegate.isOn(); }
    public boolean isOff() { return delegate.isOff(); }
    public boolean isWorking() { return delegate.isWorking(); }
    public void init() { delegate.init(); }
    public void switch() { delegate.switch(); }
}
```

Implémentation du contrat

```
public class SwitchContract extends SwitchDecorator {
    public SwitchContract(Switch delegate) { super(delegate); }
    protected void checkInvariant() {
        // inv : isOff() == not isOn()
        if( !(isOff()==not isOn()))
            throw new Error("Wrong invariant");
    }
    public void init() {
        super.init();
        // invariant
        checkInvariant();
        // Postcondition
        if( !(isOn())) throw new Error("Precondition : isOn() must be true");
    }
    etc.
```

Contrat de méthode

```
public void switch() {
        // Précondition
        if( !(isWorking())) throw new Error("Precondition : isWorking() must be true");

        // Pre-invariant
        checkInvariant();

        // Captures
        boolean old_isOn = isOn();

        // Traitement
        super.switch();

        // Post-invariant
        checkInvariant();

        // Postcondition
        if( !(isOn()== not old_isOn)) throw new Error("Postcondition blabla...");
    }
}
```

Instanciation

```
Switch s1 = new SwitchImpl();
// sans vérifications
s1.switch(); // ⇒ isOn()

Switch s2 = new SwitchContract(s1);
// avec vérifications
s2.switch(); // ⇒ isOff()
```

Remarque : en TME utilisation de l'outil TamagoCC pour la génération automatique du code de vérification (selon un pattern similaire).

Méthodologie (IV)

Des contrats aux tests

- Tests unitaires, pour chaque méthode contractualisée
  - validation/invalidation des préconditions
  - vérification de l'invariant avant et après le test
  - vérification des postconditions internes (observations internes) après le test
- Tests d'intégration
  - vérification des postconditions correspondant aux observations externes (exemple : observation sur les cuves après remplissage du tuyau)

Avantage : fournit un cadre systématique pour les tests basés sur les spécifications.

Attention : il s'agit d'un cadre minimal, il faut compléter avec une approche de test « traditionnelle » basée sur l'implémentation.

Fin