

Cours Composant

3. Conception par Contrat

©Frédéric Peschanski

Sorbonnes Universités – UPMC – LIP6

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

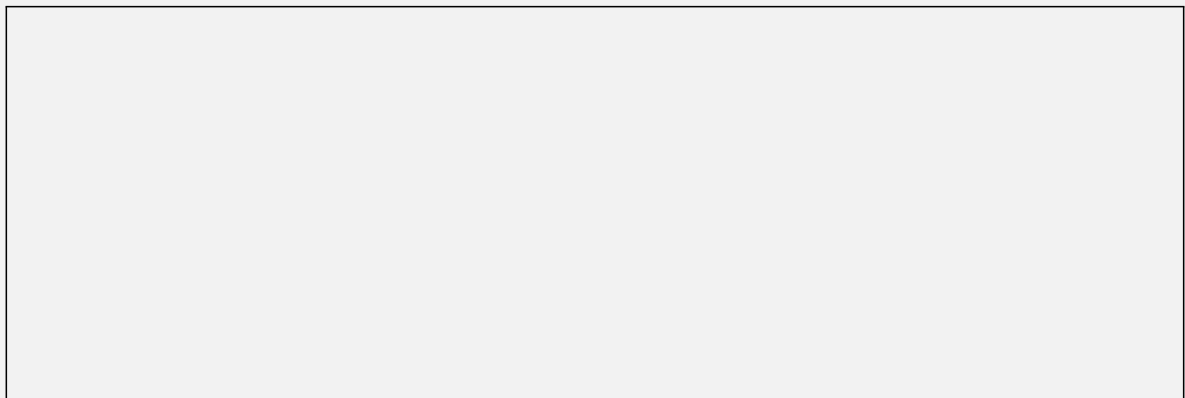
- ▶ **Livre** : Bertrand Meyer, *Conception et programmation objet*. 2005, Eyrolles
- ▶ **Site** : <http://www.eiffel.com>

Métaphore du service

Service

Un **service** est proposé par un **fournisseur** à son (ou ses) **client(s)**

Le **contrat de service** indique :



Exemple de contrat

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

Prérequis :

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

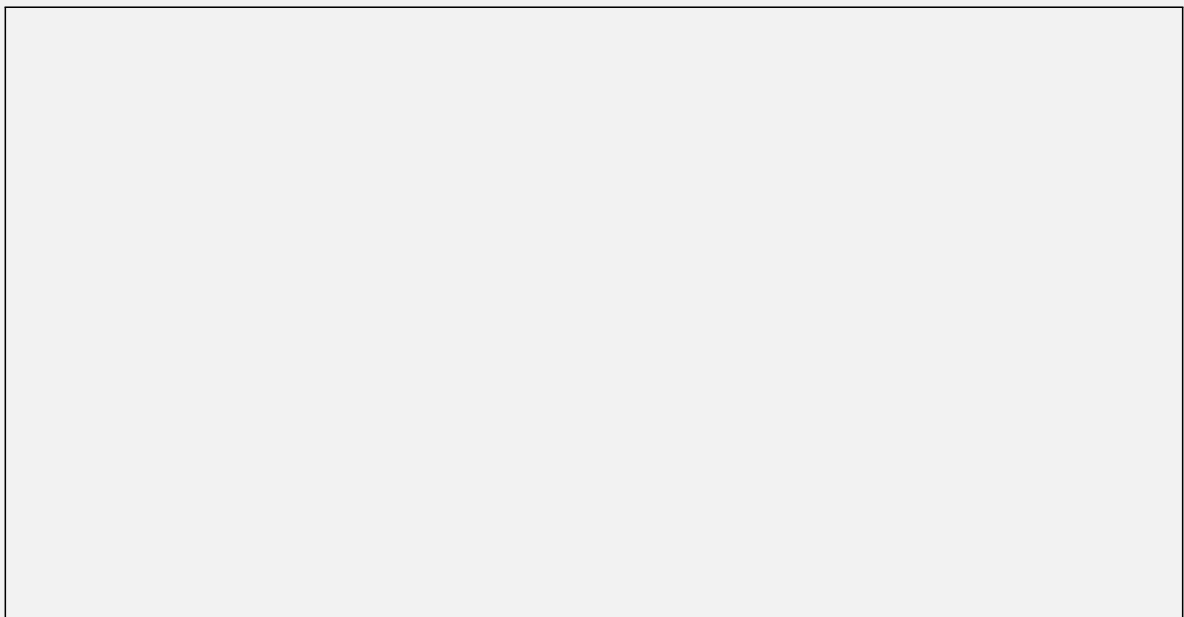
Garanties :

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

+ gestion du **risque**

Validation du contrat

Un contrat peut-être :



Contrats logiciels

Service Interface + annotations

Fournisseur Classe(s) d'implémentation de l'interface

Client code dépendant directement ou indirectement de l'interface

(ex. variable, invocation de méthode, etc.)

Contrats contrainte sur l'utilisation de l'interface et de ses méthodes

- ▶ **propriétés observables** \approx **accesseurs**
- ▶ **invariants** de type (interface/classes d'implémentation)
- ▶ pour chaque méthode
 - ▶ **préconditions** d'invocation
 - ▶ **postconditions** après exécution

Rupture Notion de "blâme"

- ▶ par le client \Rightarrow exception (au mieux) ou comportement indéfini (au pire)
- ▶ par le fournisseur \Rightarrow **BUG** !

Remarque : 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 de développement

Trois étapes fondamentales



Intérêts de l'approche :

- ▶ traçabilité de la spécification
- ▶ augmentation de la qualité du logiciel
- ▶ faible dépendance vis-à-vis de l'implémentation
- ▶ approche complémentaire des méthodes agiles et classiques.

Inconvénients :

- ▶ investissement plus important (mais rentable !)
- ▶ non-régression des contrats (donc de la spec.)

Des spécifications aux contrats

Démarche

Analyse → Conception
Spécification → Contrat

Service : *nom du service spécifié*

Observers :

fonctions d'observation de l'état

...

Constructors :

fonctions de construction

...

Operators :

fonctions de modification

...

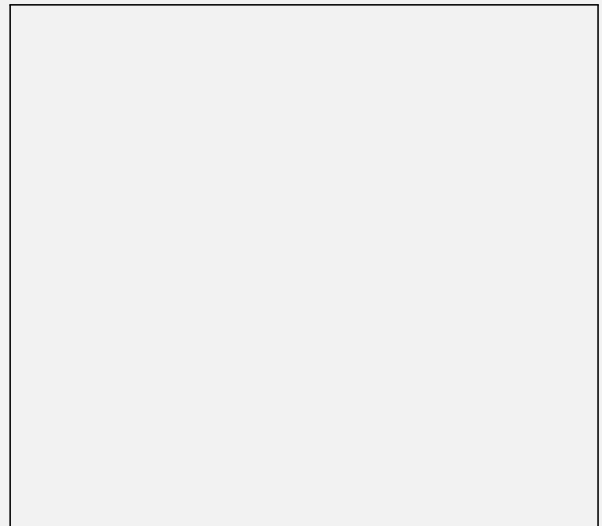
Observations :

invariants de minimisation

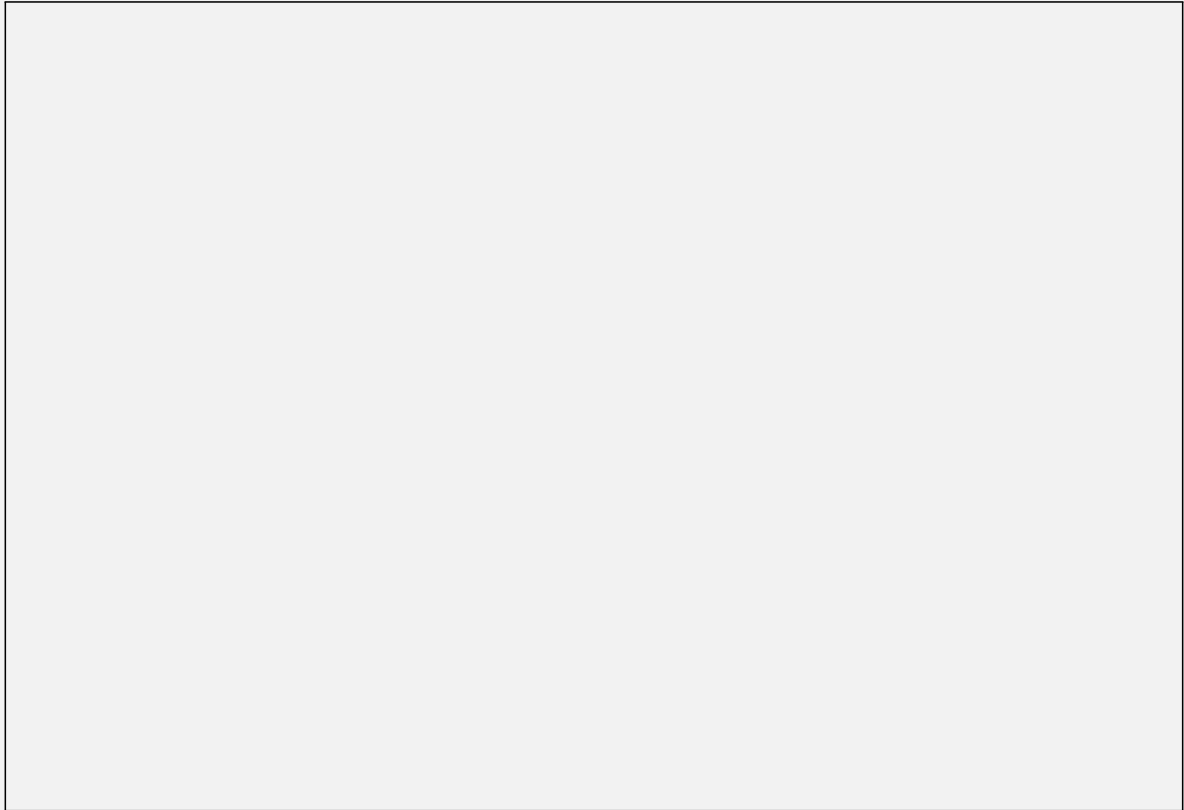
invariants de service

autres observations (postconditions)

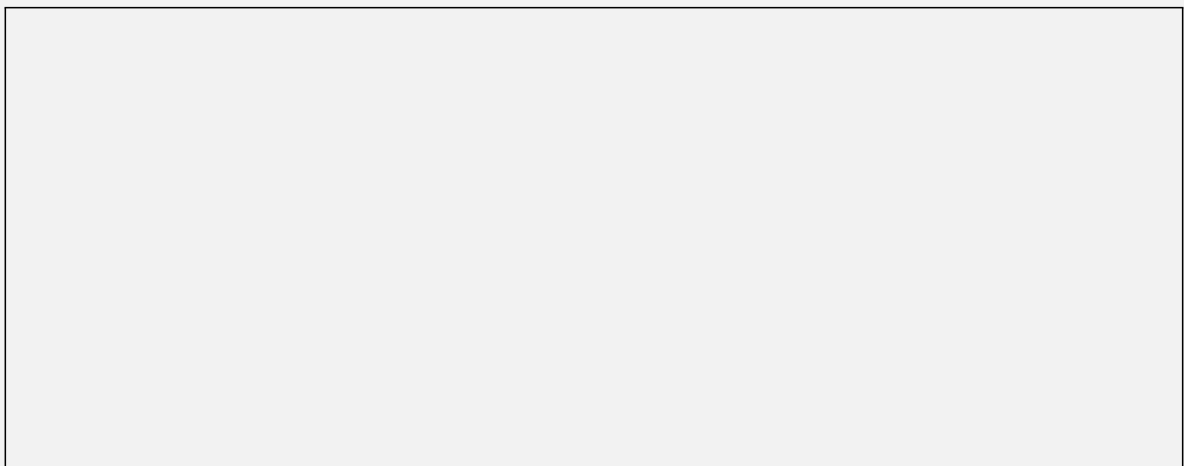
...



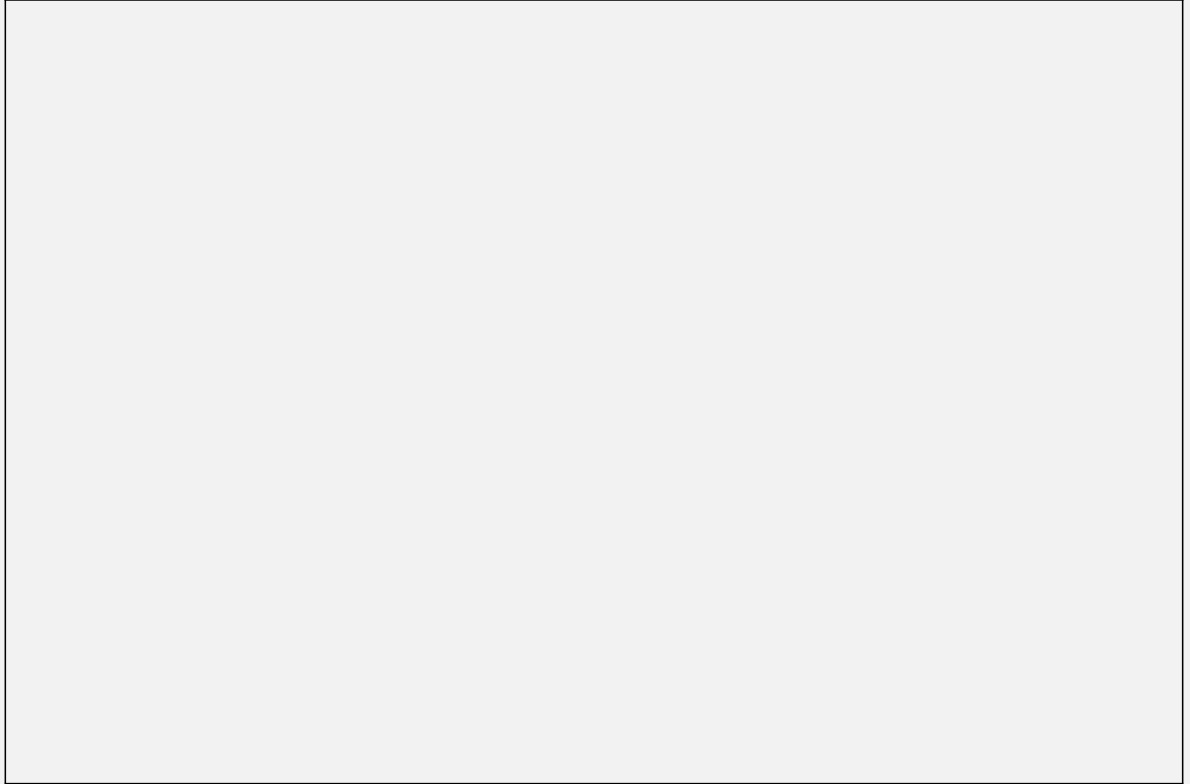
Exemple : interrupteur



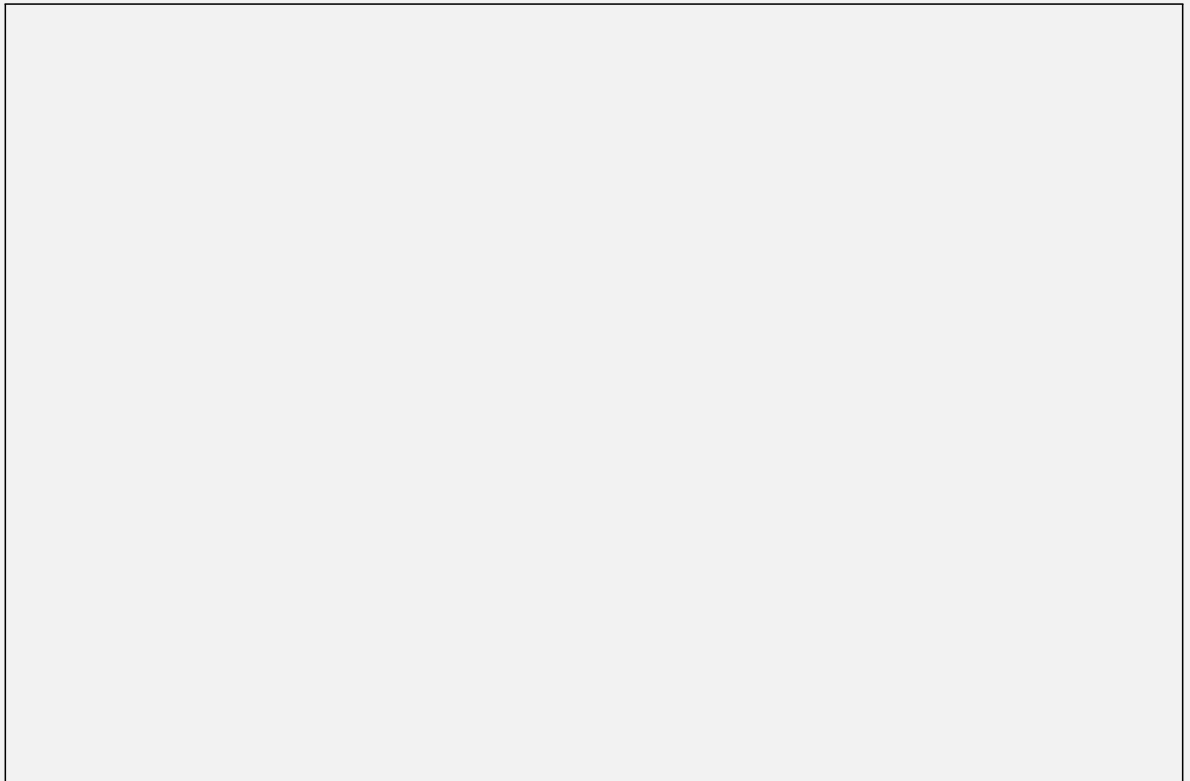
Exemple : propriétés observables



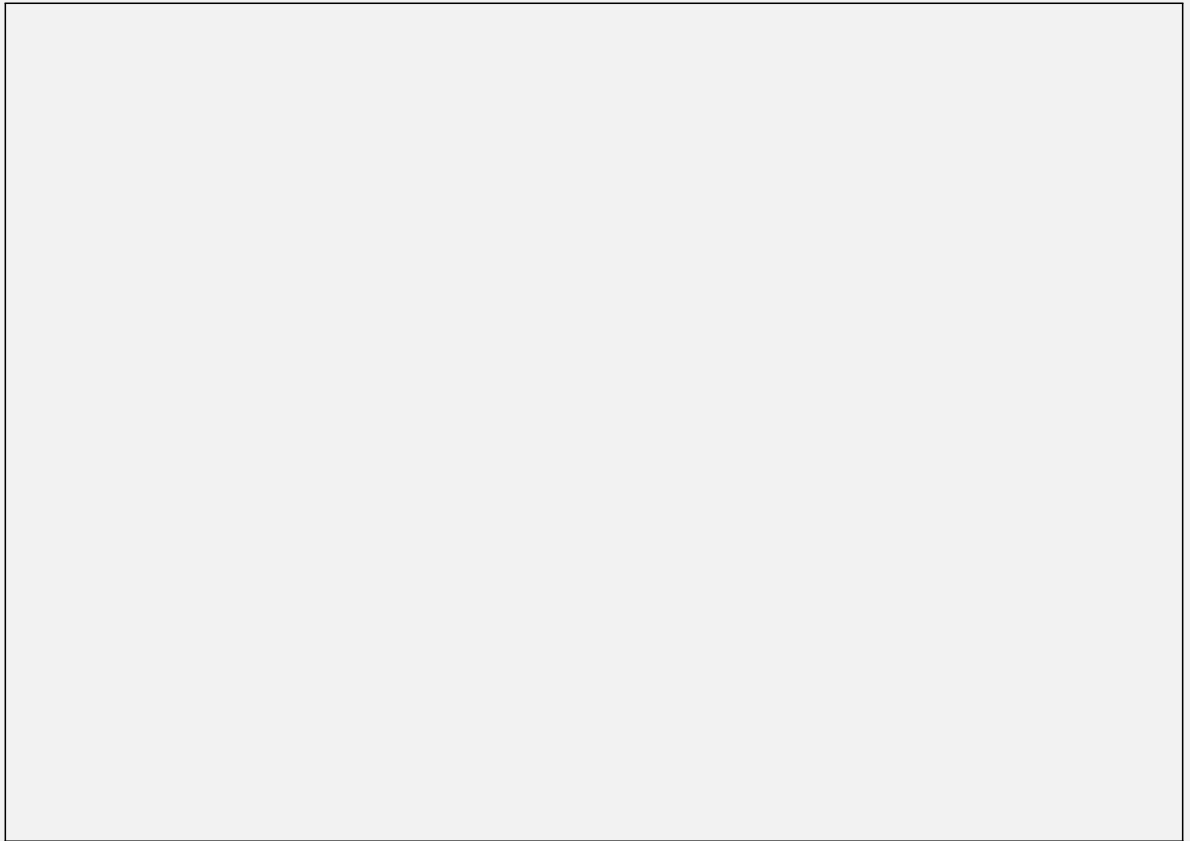
Exemple : invariants I – minimisation



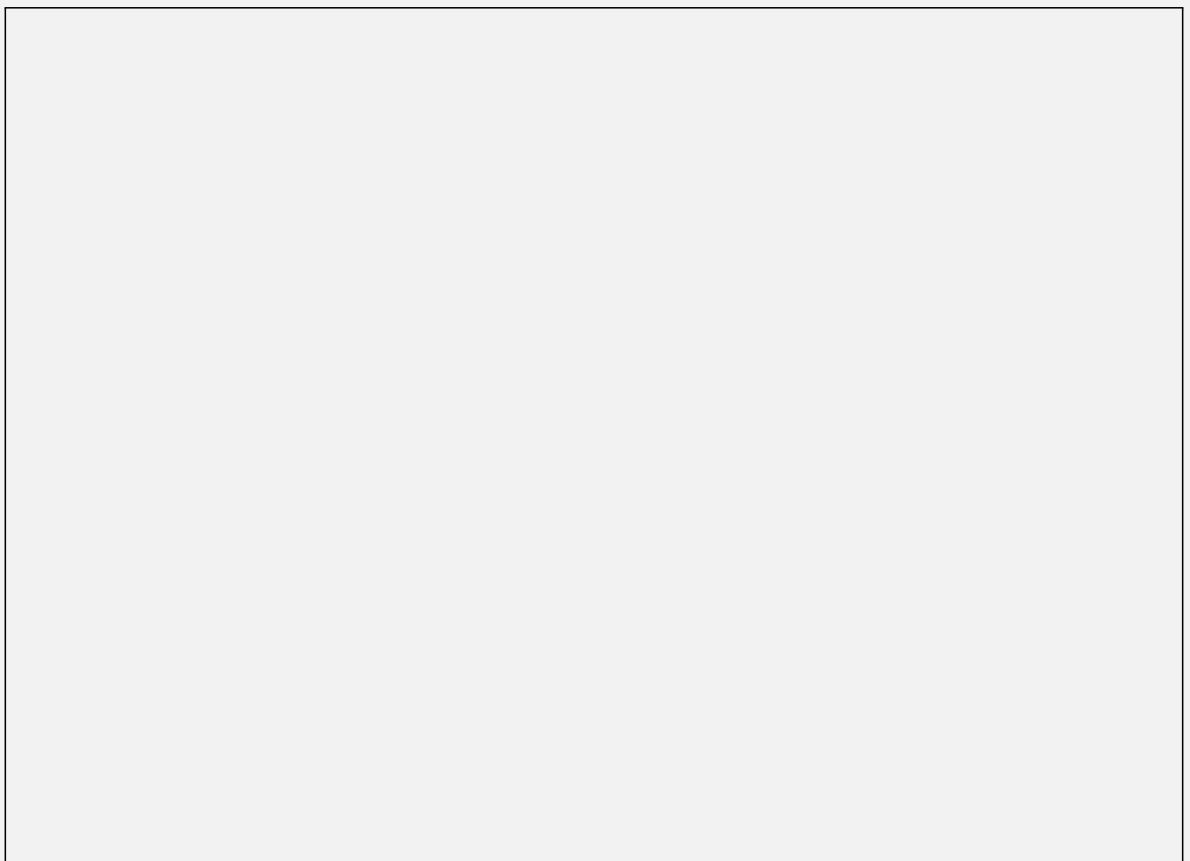
Exemple : invariants II



Exemple : préconditions

A large, empty rectangular box with a thin black border, intended for writing preconditions. It occupies the majority of the page below the title.

Exemple : postconditions

A large, empty rectangular box with a thin black border, intended for writing postconditions. It occupies the majority of the page below the title.

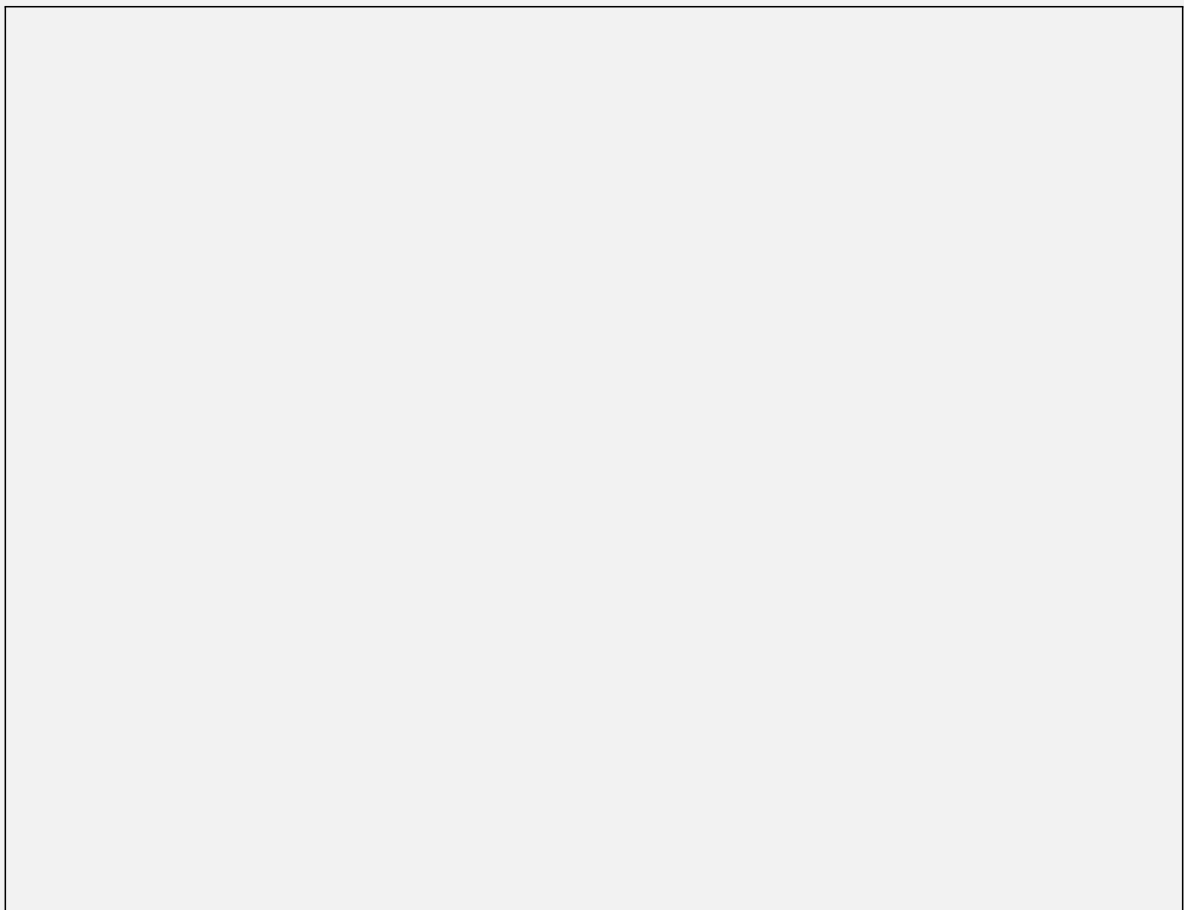
Implémentation des contrats = tests embarqués

Besoins

- ▶ Séparation (interface de) service et implémentation(s)
- ▶ Séparation code d'implémentation vs. code de vérification des contrats
- ▶ Activation/Désactivation par instance de la vérification (bonus)

⇒ design pattern **décorateur**

Implémentation des contrats = décorateur



Interface de service

```
public interface Switch {  
    /* observers */  
    public boolean isOn();  
    public boolean isOff();  
    public int getCount();  
    public boolean isWorking();  
  
    /* invariants */  
    // inv : isOff() == !isOn()  
    // inv : getCount() >= 0  
  
    /* Constructors */  
    // post : isOn() == false  
    // post : getCount() == 0  
    public void init();  
  
    /* Operators */  
    // press :  
    // pre : isWorking()  
    // post : isOn() == !isOn()@pre  
    // post : getCount() == getCount()@pre + 1  
    public void press();  
}
```

Annotations Traduction du contrat en expressions booléennes “quasi-Java”

Syntaxe : **<expr>@pre**
expression de postcondition évaluée au moment de la précondition

Implémentation du code métier

```
public class SwitchImpl implements Switch {  
    private boolean state;  
    private int count;  
    public SwitchImpl() { init(); }  
    public void init() { state=false; count = 0; }  
    public boolean isOn() { return state; }  
    public boolean isOff() { return !isOn(); }  
    public int getCount() { return count; }  
    public boolean isWorking() { return true; /* ahem ... */ }  
    public void press() { state = !state; count += 1; }  
}
```

Remarque :

- en CPS on ne s'intéresse qu'accessoirement aux implémentations (il faudra tout de même en faire une... Sinon c'est pas drôle ...).

Vérification des contrats

Du contrat implémenté à la vérification

1. Vérification statique \Rightarrow outils
2. Vérification dynamique \Rightarrow tests embarqués
 - ▶ 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 int getCount() { return delegate.getCount(); }  
    public boolean isWorking() { return delegate.isWorking(); }  
    public void init() { delegate.init(); }  
    public void press() { delegate.press(); }  
}
```

Tests embarqués : invariants

```
public class SwitchContract extends SwitchDecorator {
    public SwitchContract(Switch delegate) {
        super(delegate);
    }

    public void checkInvariant() {
        // inv : isOff() == !isOn()
        if( !(isOff() == !isOn()))
            throw new InvariantError("isOff() == !isOn()");
        // inv : getCount() >= 0
        if( !(getCount() >= 0))
            throw new InvariantError("getCount() >= 0");
    }

    // ... etc.
```

Test embarqués : contrat d'initialisation

```
public class SwitchContract extends SwitchDecorator {

    // ... etc.

    public void init() {
        super.init();
        // invariant (après initialisation)
        checkInvariant();
        // Postconditions
        // post : isOn() == false
        if( !(isOn() == false))
            throw new PostConditionError("isOn() == false");
        // post : getCount() == 0
        if( !(getCount() == 0))
            throw new PostConditionError("getCount() == 0");
    }

    // ... etc.
```

Tests embarqués : Contrat de méthode

```
public void press() {  
    // Précondition  
    if( !(isWorking())) throw new PreconditionError("...blabla...");  
  
    // Pre-invariant  
    checkInvariant();  
  
    // Captures  
    boolean isOn_at_pre = isOn();  
    int getCount_at_pre = getCount();  
  
    // Traitement  
    super.press();  
  
    // Post-invariant  
    checkInvariant();  
  
    // Postcondition  
    // post : isOn() == !isOn()@pre  
    if( !(isOn() == !isOn_at_pre)) throw new PostConditionError("...blabla...");  
    // post : getCount() == getCount()@pre + 1  
    if( !(getCount() == getCount_at_pre + 1)) throw new PostCondition ...  
}  
}
```

Instanciation

```
Switch s1 = new SwitchImpl();  
// sans vérifications  
s1.press(); // ⇒ isOn()  
  
Switch s2 = new SwitchContract(s1);  
// avec vérifications  
s2.press(); // ⇒ isOff()
```

Etape suivante : tests basés sur les modèles

Des contrats aux tests (cf. cours 5)

- ▶ 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

Fin