

Cours CPS

1. Introduction

Frédéric Peschanski
LIP6 - SysComp - APR

UPMC Sorbonne universités

Point de vue du cours

Constat

- ▶ Vous avez des compétences en **programmation** (notamment en Java et/ou en OCaml)
- ▶ Vous avez de notions de **conception logicielle** (notamment en modélisation objet)

Mais...

- ▶ Un ingénieur est (souvent) confronté au problème de la **spécification fonctionnelle** d'un logiciel.
 - ▶ dans des domaines variées : avionique, banque, manufacture, etc.
 - ▶ avec des interlocuteurs qui ne sont *pas* informaticiens : ingénieurs « métier », décideurs, etc.

Motivation

⇒ à travers la notion de **composant**, le cours CPS s'intéresse aux relations entre ces trois activités.

Enjeux de la spécification

Objectif : se placer dans la peau d'un ingénieur « métier »

- ▶ Comment un étudiant en informatique peut-il se faire passer pour un « non-informaticien » ?
- ▶ Quel(s) langage(s)/formalisme(s) adopter pour la spécification ? la conception ? la programmation ?

Ingrédients principaux de CPS

- ▶ un langage de **types algébriques**, assez neutre mais *semi-formel* pour la partie spécification
- ▶ des **contrats logiciels** (en complément de techniques « classiques » notamment UML) pour la partie conception
- ▶ des tests embarqués et du **test basé sur les modèles** (MBT) (en complément de la programmation orientée objet en Java) pour la partie implémentation

Disclaimer

(exonération des responsabilités ?)

- ▶ CPS n'est pas un cours « techno »
(vous avez vos soirées et week-ends pour ça)
- ▶ On aborde un monde obscur – et pourtant essentiel : la spécification logicielle
- ▶ *par déformation professionnelle* on vous fait adopter le point de vue du **logiciel sûr** et des méthodes formelles dites « légères » (au moins on sait de quoi on parle ...)

Mais :

- ▶ Domaine métier sympa : modélisation de jeux vidéos (plutôt arcade)
- ▶ Équipe enseignante « du tonnerre »
- ▶ Il y a quand même pas mal de programmation et vous en retirerez finalement de ce côté là quelques bonnes pratiques.

Prérequis

Compétences techniques

- ▶ Maîtrise des **concepts objets**
(encapsulation, relations, héritage, designs patterns, etc.)
- ▶ Bonne pratique de **Java**
(Language de support du cours)
- ▶ Familiarité avec l'environnement de développement Eclipse
JDT + ant (import/export de projets, etc.)

Mais aussi (et là, *ahem* ...)

- ▶ **Maths discrètes** (ensembles, relations, fonctions, etc.)
- ▶ **Logique** (propositionnelle et premier ordre)

Organisation du cours

Organisation

- ▶ Cours : concepts fondamentaux et méthodologie
- ▶ Travaux dirigés : illustration des concepts
- ▶ Travaux sur machine encadrés : projets de programmation

Déroulement des TME

- ▶ **Préparation** : en général on part d'un projet existant
(format jar normalisé + script ant)
- ▶ **première soumission du TME en l'état**
(à la fin des deux heures)
- ▶ Possibilité d'une **seconde soumission du TME complété au plus tard** la veille de la prochaine séance (soumission après minuit = **pas** de soumission)

Evaluation

Contrôle continu (40%~50%)

- ▶ **Devoir sur table** : exercice de spécification « pure »
(première étape du projet)
- ▶ **Projet** : jeu vidéo dont le moteur est spécifié
(semi-)formellement
- ▶ **Rendus de TME**

Examen final (50%~60%)

- ▶ exercices « classiques » de type TD

Important : informations non-contractuelles

Composant logiciel

⇒ notion difficilement réductible à une définition simple

(exercice : définition d'« objet » ?)

Exemples de définition :

- ▶ **C. Szyperski**

- ▶ **B. Meyer**

Trustable components (B. Meyer)

Cours CPS : vers le composant logiciel « de confiance »



réutilisabilité Critère de qualité. Propriété d'un élément logiciel à être employés dans plusieurs contextes différents, potentiellement pas des clients variés. Important : ce critère ne se décrète pas, on le constate à posteriori

spécification logicielle Document plus ou moins formel qui décrit précisément ce que l'on attend du logiciel, tant en termes de conception que d'implémentation

garantie de qualité cf. cours 2

Composants CPS

Check-list Composant logiciel en CPS

- ▶ Les composants sont clairement spécifiés
 - ▶ Notion de **service** : **interface** (signatures) + **contraintes** sur le comportement
 - ▶ **Interface interne** : fonctionnalités du composant
 - ▶ **Interface externe** : spécification du contexte d'utilisation (dépendances explicites) : le composant a vocation à être **composé** avec d'autres composants
- ▶ **Garanties** sur les implémentations
 - ▶ Relie la spécification et ses possibles implémentations
 - ▶ Approche « légère » : conception par contrat
 - ▶ Approche « lourde » : logique de Hoare

Remarque : objets vs. composants

Les objets sont des composants « minimaux » :

- ▶ Encapsulation, cycle de vie minimal (création, utilisation, destruction)
- ▶ Interface interne minimale : classe et/ou interface

Mais il manque (entre autre) :

- ▶ des spécifications plus « sémantiques »
- ▶ l'explicitation du contexte d'utilisation (interface externe)
- ▶ les garanties sur la qualité (exception : conception par contrat)
- ▶ la problématique du déploiement

Plan des cours

1. Designs patterns pour les composants
2. Langage de spécification
3. Conception par contrat I
4. Conception par contrat II
5. Test basé sur les modèles
6. Logiciel sûr : Logique de Hoare I
7. Logiciel sûr : Logique de Hoare II
8. Modélisation de la concurrence I
9. Modélisation de la concurrence II
10. Ouverture : spécification et validation des programmes

Plan du cours 1

Designs patterns pour les composants

- ▶ Patterns des Javabeans
 - ▶ Pattern : « Observable properties »
 - ▶ Pattern : « Event-based listeners »
- ▶ Pattern essentiel des composants
 - ▶ « Require/Provide connector »

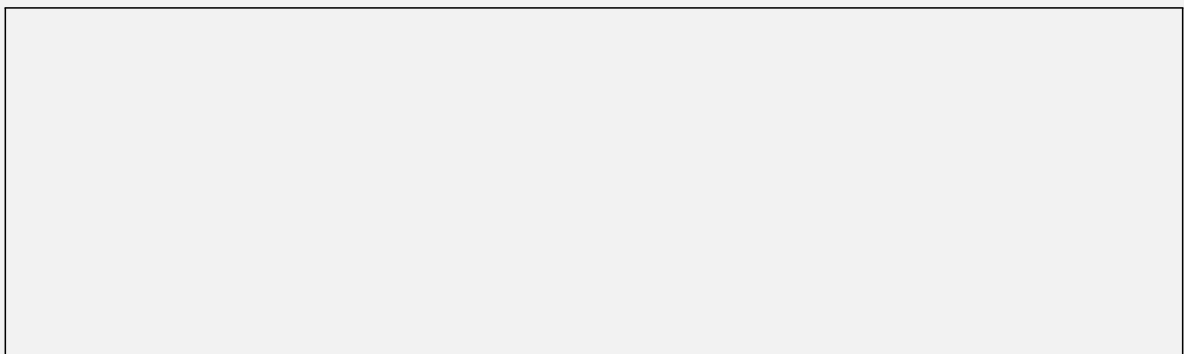
Propriétés observables

Motivation : Observabilité

Un composant doit être capable d'expliciter ce que l'on peut observer sur son état interne. Un composant doit être également configurable.

Sinon on ne peut rien spécifier à l'avance sur l'état, et rien n'est vérifiable ensuite.

Pattern : « Observable properties »








Structure du pattern

Propriété

Propriété de nom `prop` de type `Type` en lecture/écriture

```
// Access: read
public Type getProp();
// Access: write
public void setProp(Type value);
```

«component» Component
 hidden attributes
 getProp1() : Type1  getProp2() : Type2  setProp2(p2:Type2) : void  setProp3(p3:Type3) : void

Exemple : propriétés Javabeans

Propriété : `state` en *lecture seule*

```
public class Interrupteur {
    public enum { ON, OFF } StateType;
    private boolean internalState;
    public Interrupteur() { internalState = false; }

    /* Propriétés */
    public StateType getState() {
        if(internalState==true)
            return ON;
        else return OFF;
    }

    /* Fonctionnalités */
    public boolean isOn() { return state==true; }

    public void switch() { state = !state; }
}
```


Bilan pattern *Observable properties*

Intérêts

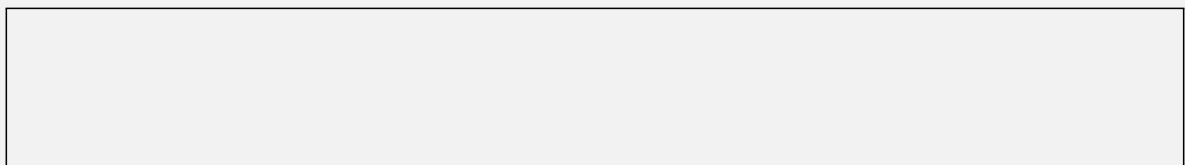
- ▶ Observabilité : concept fondamental (plus qu'il n'y paraît)
- ▶ Mise en œuvre simple

Limites

- ▶ En lui-même le pattern ne fait pas grand chose, il faut des outils pour l'exploiter
- ▶ Difficultés « cachées » : types observables

Pattern : « Event-based listeners »

Motivation



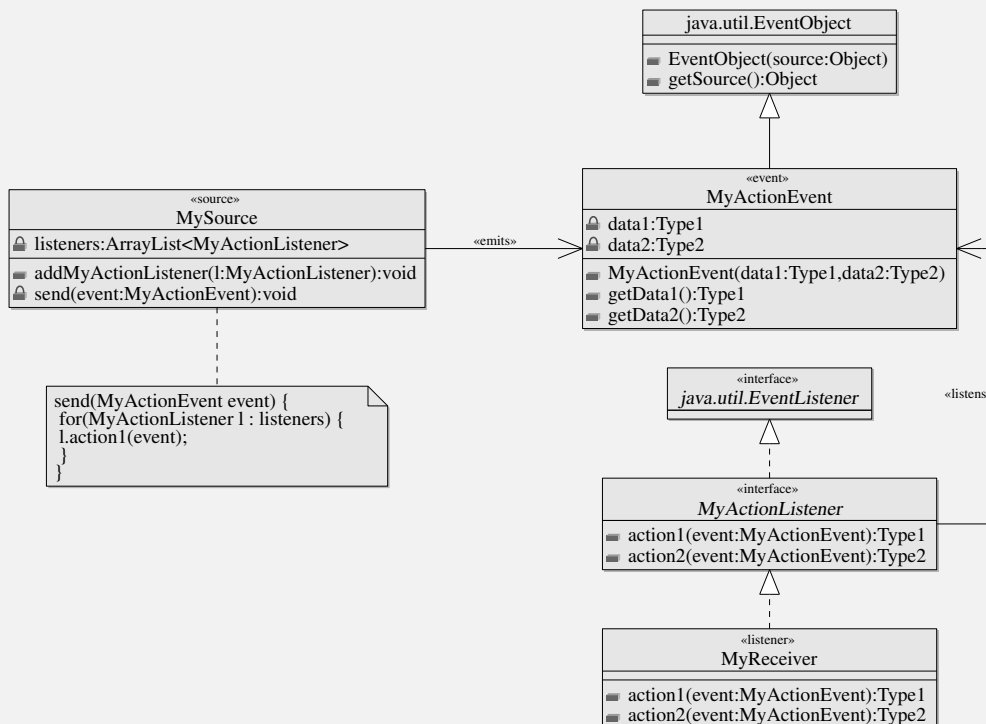
Javabeans : mode de communication événementiel

- ▶ Composant : **source** et/ou **écouteur** d'événements
- ▶ **Événement** : objet passif et immuable

Constituants

- ▶ Classes (types) d'événements : **MyEvent**
- ▶ Interfaces d'écoute (*listener* : **MyEventListener**)
- ▶ méthodes de réaction : dépend de l'événement

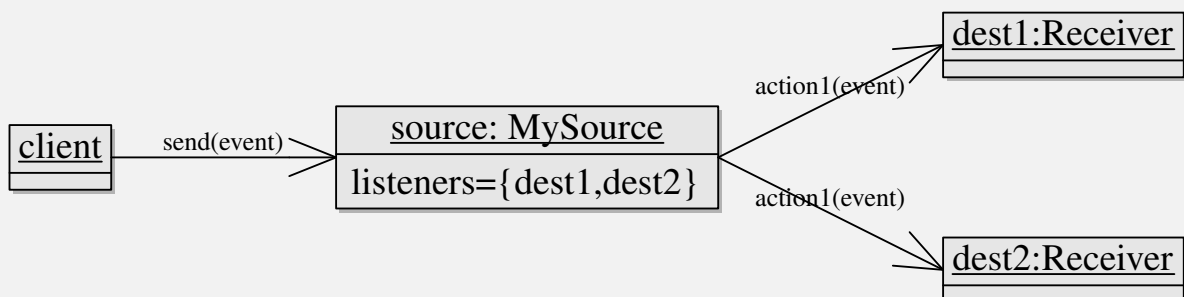
Structure du pattern



Événement : exemple d'occurrence

Préalable : les deux écouteurs sont enregistrés auprès de la source

```
source.addMyActionListener(dest1);
source.addMyActionListener(dest2);
```



Bilan pattern *Event-based communication*

Intérêts

- ▶ Découple structurellement les émetteurs et récepteurs d'événements
- ▶ Permet la communication « 1 (émetteur) vers N (récepteurs) »
- ▶ Explicite la dépendance du récepteur (interface d'écoute)
- ▶ Événements multiples et sélection par typage
- ▶ Branchements dynamiques (si nécessaires)

Limites

- ▶ Pas de découplage du contrôle : la source doit contrôler la communication
- ▶ Dépendances explicitées seulement dans un sens : le récepteur est identifié comme tel, mais pas la source
- ▶ Pattern un peu « lourd »

Pattern : « Require/Provide connector »

Motivation

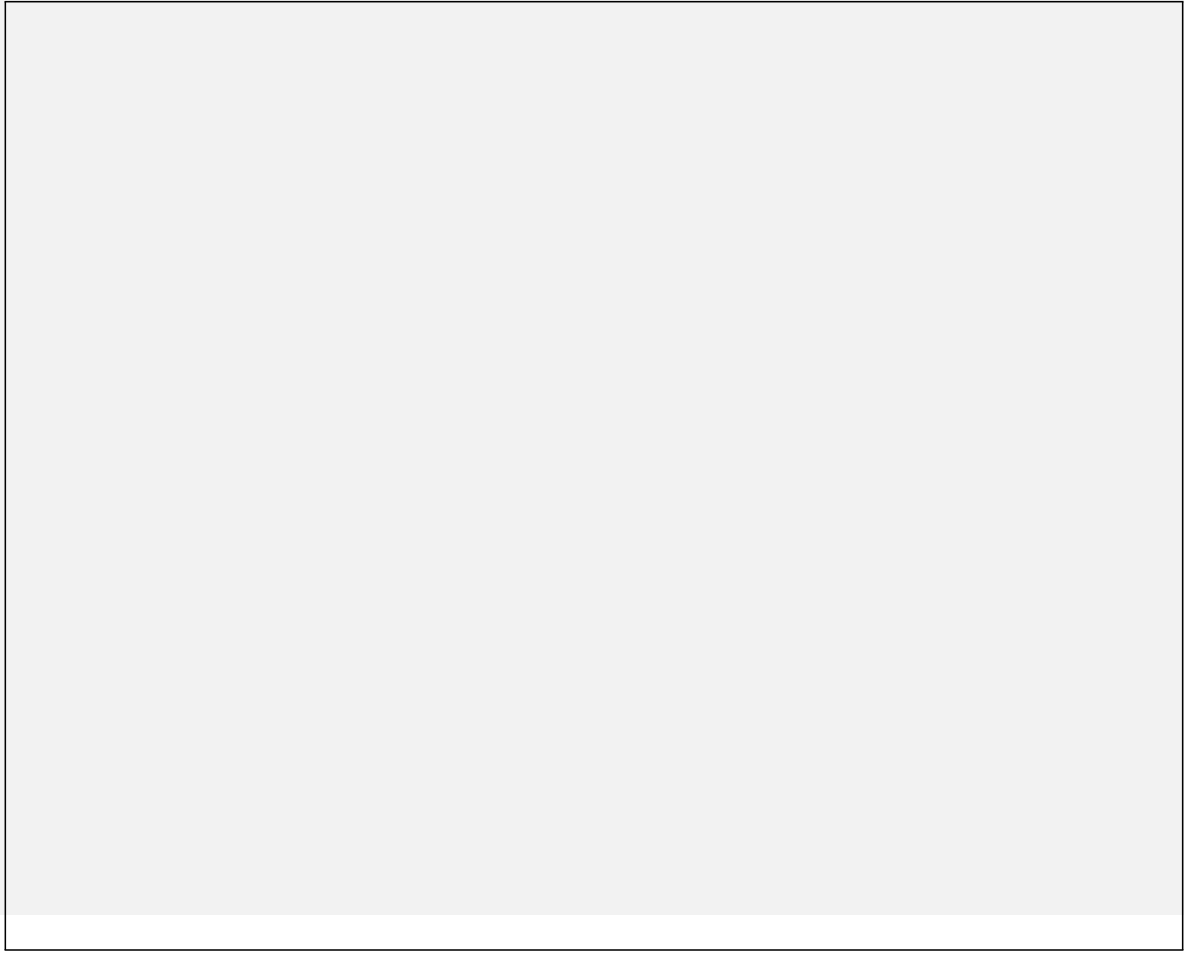
Un composant doit expliciter ses dépendances externes (bis) + Description séparées des fonctionnalités implémentées par les composants ⇒ notion d'**interface** ou de **service**¹

Principes

- ▶ Un **service** regroupe des fonctionnalités
- ▶ Client du service : interface de liaison « **require** »
- ▶ Fournisseur de service : implémente le service

1. Ne pas confondre « interface de composant » et « interface java ».

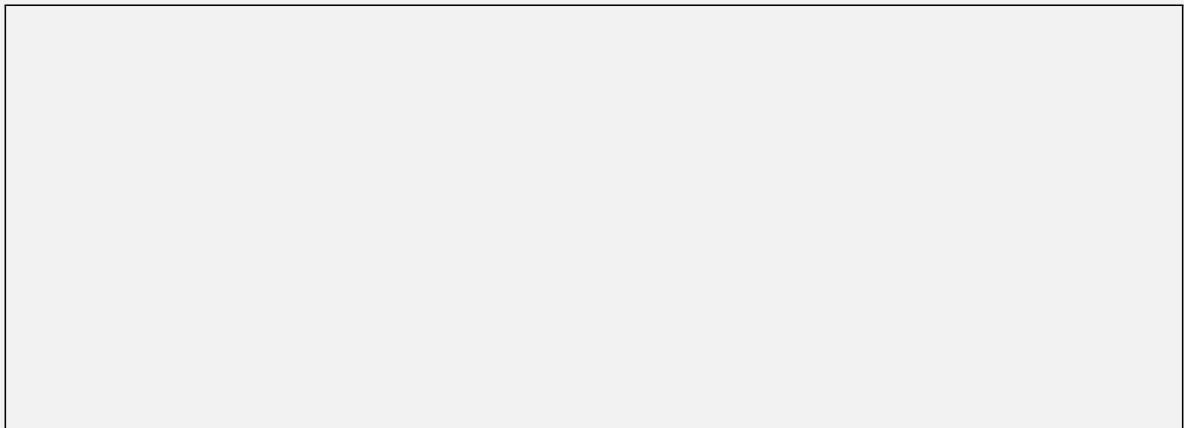
Structure du pattern



Modèle de composition

Composition d'une architecture :

```
Client client = new Client();  
Provider provider = new Provider();  
client.bind(provider);  
client.use();
```



Bilan pattern *Require/Provide*

Intérêts

- ▶ Découple structurellement les fournisseurs et clients de service
- ▶ Dépendances explicitées dans les deux sens (client et fournisseur)
- ▶ Métaphore du service et client/fournisseur (cf. conception par contrat)

Limites

- ▶ Pas de découplage du contrôle : le client gère la liaison
- ▶ Pattern un peu « lourd » (beaucoup d'interfaces en Java)
(autre approche : code java simple mais XML (ou JSon) pour décrire les liaisons. cf. plugins eclipse, GUI android, Spring, etc.)

Questions ?