Quelques principes de bonne conception épisode 1

S.O.L.I.D. ?

- Single responsibility principle (SRP) : une classe n'a qu'une seule responsabilité
- Open/closed principle (OCP) : un élément logiciel (classe ou méthode) doit être ouvert à l'extension mais fermé à la modification
- Liskov substitution principle (LSP) : les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-types sans «casser» le programme
- Interface segregation principle (ISP) : il faut privilégier plusieurs interfaces spécifiques à des besoins clients
- Dependency inversion principle (DIP) : il faut dépendre des abstractions, pas des réalisations concrètes

Single responsibility principle (SRP)

- Combien de responsabilités ici ?

```
class Employee {
  public Pay calculatePay() {...}
  public void save() {...}
  public String describeEmployee() {...}
}
```

- SRP implique forte cohésion et faible couplage

https://jrebel.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/

GRASP?

General

Responsibility

Assignment

Software

Pattern

Patrons logiciels généraux d'affectation des responsabilités

Responsabilité ?

- Une **Abstraction** d'un comportement
  - Les méthodes **ont** des responsabilités
  - Une classe **n'est pas** une responsabilité
  - Un cas d'utilisation **n'est pas** une responsabilité

- Et en *Conception Orientée Objets ?*
  - Les objets collaborent par leur responsabilité à réaliser une objectif

GRASP: objectifs

- Penser systématiquement le logiciel en termes de :
  - Responsabilités
  - Par rapport à des rôles (des objets)
  - Qui collaborent

- Réduire le décalage entre représentation « humaine » du problème et représentation informatique

Responsabilité en GRASP

- Responsabilité imputée à :
  - Un objet seul
  - Un groupe d'objets qui collaborent pour s'acquitter de cette responsabilité
- GRASP aide à :
  - Décider quelle responsabilité assigner à quel objet (à quelle classe)
  - Identifier les objets et responsabilités du domaine
  - Identifier comment les objets interagissent entre eux
  - Définir une organisation entre ces objets

Types de responsabilité

- Responsabilité de **FAIRE**
  - Accomplir une action (calcul, création d'un autre objet)
  - Déclencher une action sur un autre objet (déléguer à celui qui **SAIT** faire)
  - Coordonner les actions des autres objets (déléguer à X, puis Y et en fonction du résultat, etc.)

Types de responsabilité

- Responsabilité de **SAVOIR**
  - Connaître les valeurs de ses propriétés (attributs privés)
  - Connaître les objets qui lui sont rattachés (références, associations…)
  - Connaître les éléments qu'il peut dériver (la taille d'une liste)

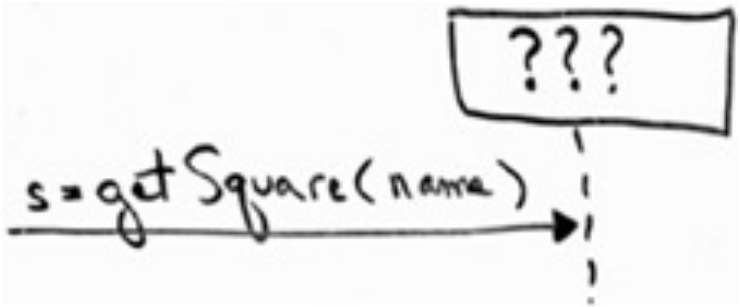
1. Spécialiste de l'information

- Problème
  - Quel est le principe général d'affectation des responsabilités des objets ?

- Solution
  - Affecter la responsabilité à la classe spécialiste de l'information, c'est-à-dire la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité

1. Spécialiste de l'information

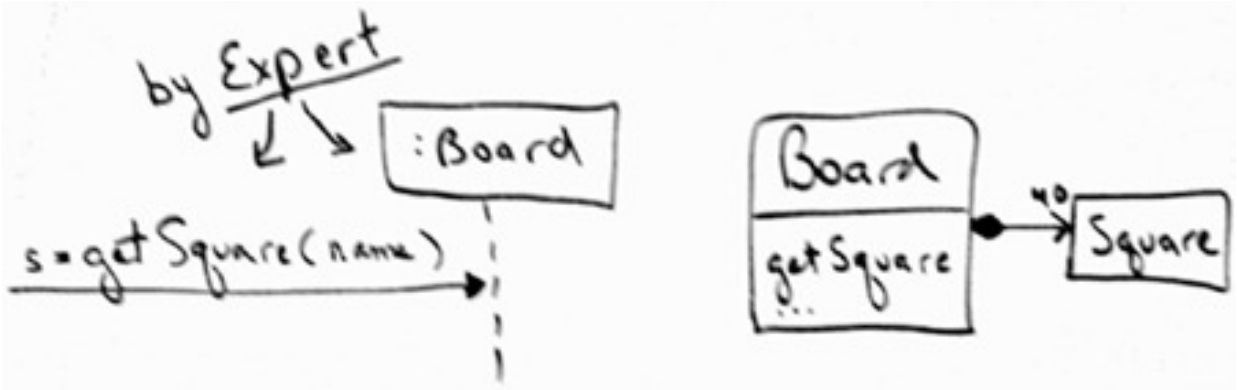
- Exemple du Monopoly
  - Qui est responsable de l'accès à une case donnée du jeu ?



Applying UML and Patterns – Craig Larman

1. Spécialiste de l'information

- Board!



- Les plus utilisé de tous les patterns GRASP
- L'accomplissement d'une responsabilité nécessite souvent que l'information nécessaire soit répartie entre différents objets

Applying UML and Patterns – Craig Larman

Dans la main de poker, qui a la responsabilité de contrôler la saisie des mains ? De les comparer ? Qui a celle de déterminer ce qu'on contient une main ?

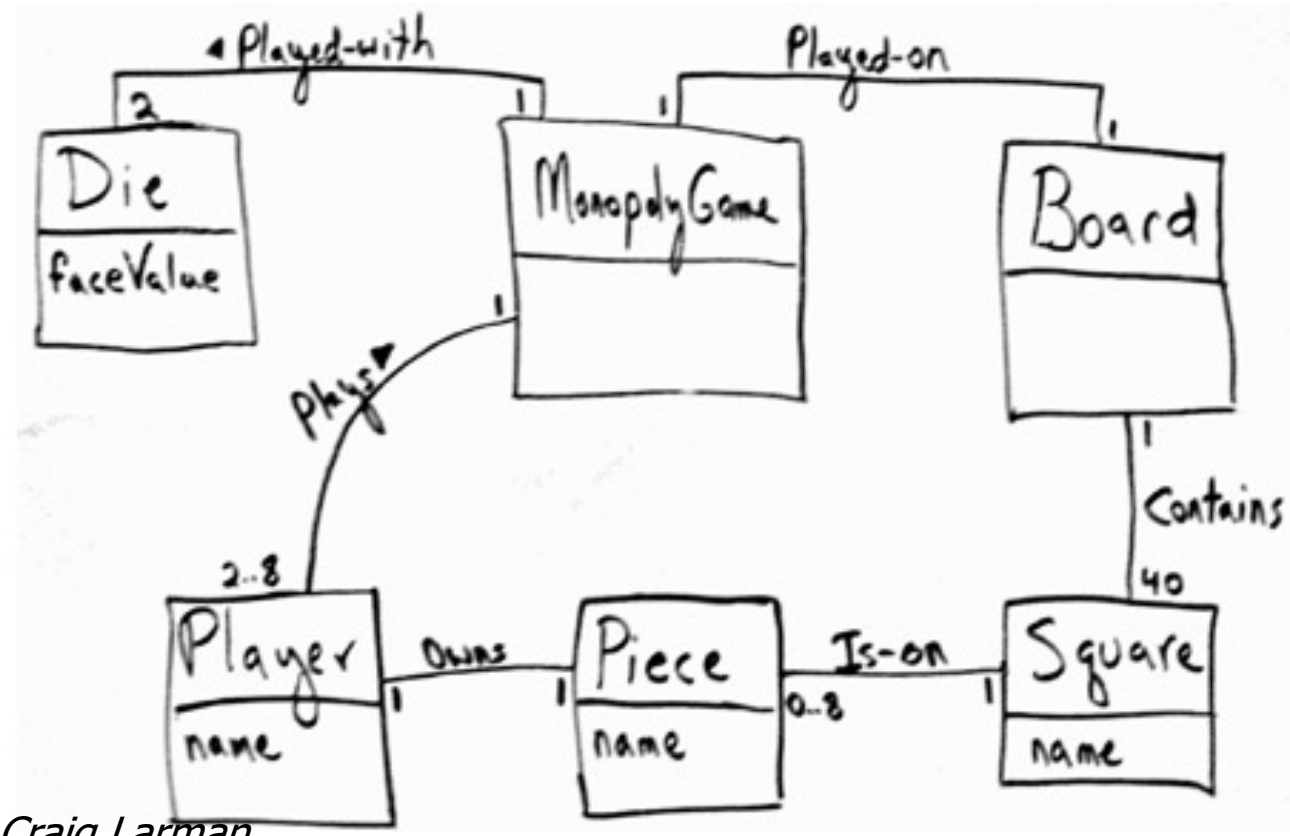
2. Créateur

- Problème
  - Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe ?

- Solution
  - Affecter à une classe B la responsabilité de créer une instance de A si :
    - B contient ou agrège des objets A, ou
    - B utilise étroitement des objets A, ou
    - B connaît les données utilisées pour initialiser les objets A

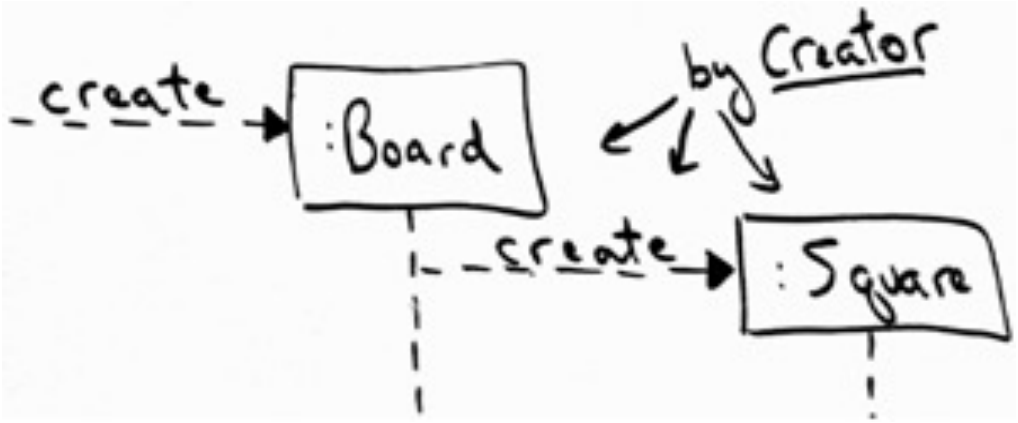
2. Créateur

- Exemple du Monopoly
  - Qui crée les cases (Square) ?



2. Créateur

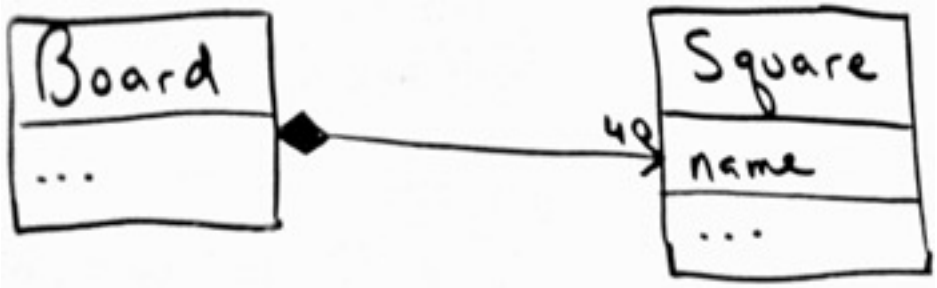
- On peut s'appuyer sur le séquencement des actions dans le code



Applying UML and Patterns – Craig Larman

2. Créateur

- Et obtient des compositions (les cases disparaissent avec le plateau) :



Applying UML and Patterns – Craig Larman

- Dans la main de poker, qui a la responsabilité de créer les cartes ? La main de chaque joueur ?

3. Faible couplage

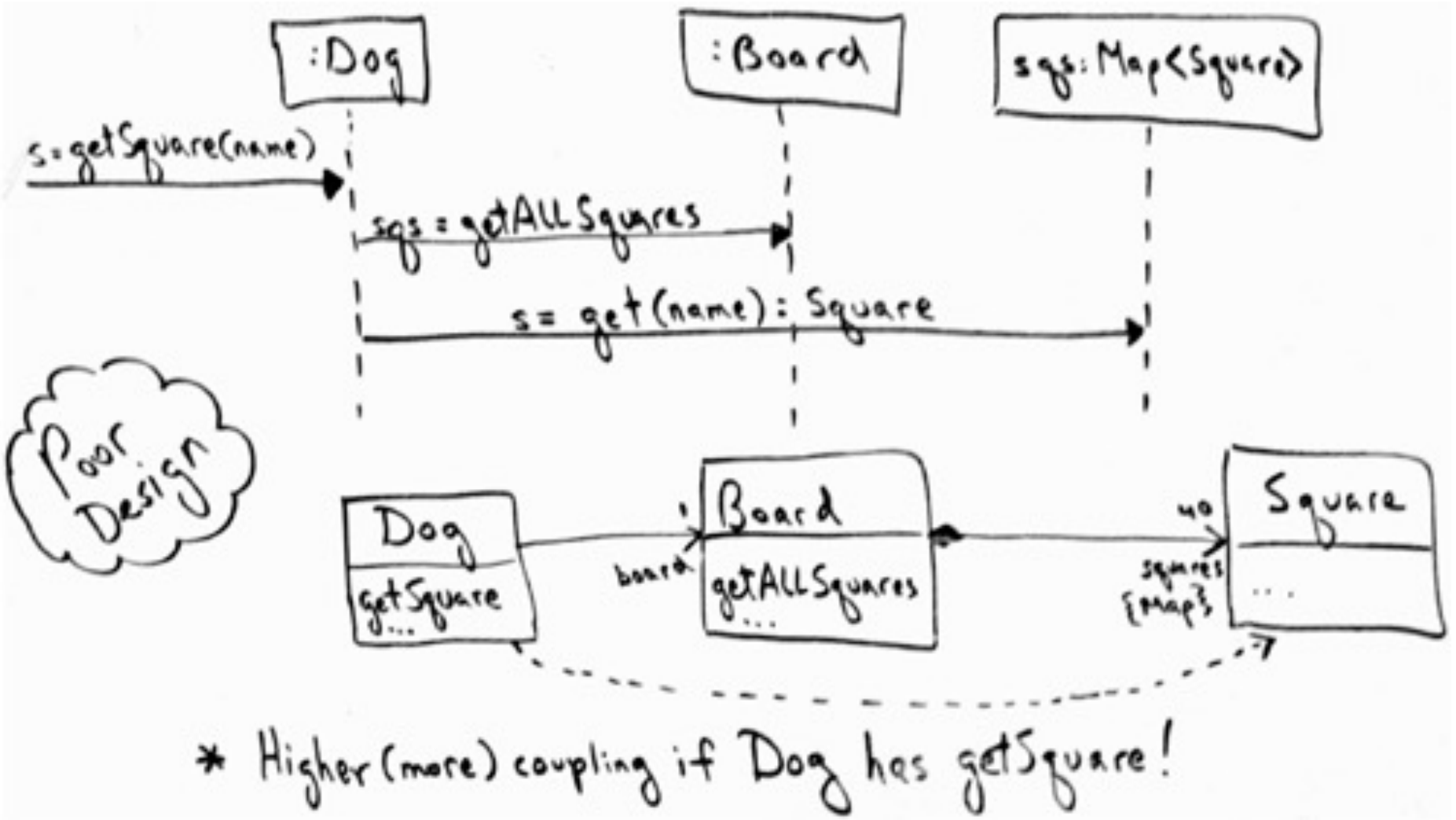
- Problème
  - Comment minimiser les dépendances, réduire l'impact des changements, et augmenter la réutilisation ?

- Solution
  - Mesurer le couplage « en continu »
  - Identifier les différentes solutions à l'affectation de responsabilité
  - Affecter une responsabilité de sorte que le couplage reste faible

3. Faible couplage

- Exemples classiques de couplage de TypeX vers TypeY en orienté objet :
  - TypeX a un attribut qui réfère à TypeY
  - TypeX a une méthode qui référence TypeY
  - TypeX est une sous-classe directe ou indirecte de TypeY
  - TypeY est une interface et TypeX l'implémente

3. Faible couplage



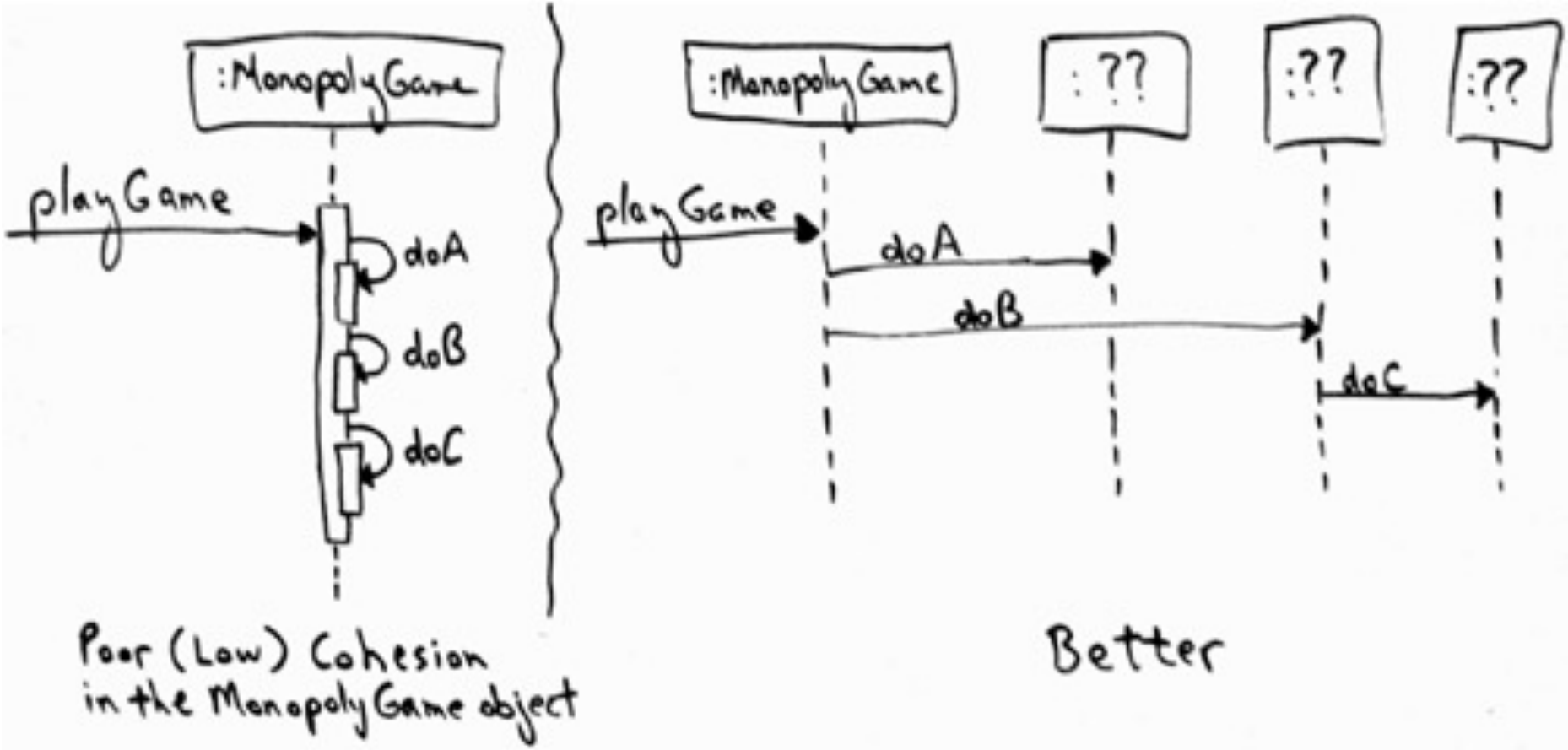
3. Faible couplage

- Couplage fort :
  - Un changement force aux changements dans les classes liées
  - Les classes prises isolément sont difficiles à comprendre
- Il n'y a pas de mesure absolue du seuil d'un couplage trop fort…
- Un fort couplage n'est pas dramatique avec des éléments très stables (java.util)

4. Forte cohésion

- Problème
  - Comment maintenir la complexité gérable ?
    - Avoir des objets compréhensibles et facile à gérer
- Solution
  - Comme pour le couplage faible : mesurer, caractériser, choisir

4. Forte cohésion



Applying UML and Patterns – Craig Larman

4. Forte cohésion

- Problème de la faible cohésion sur les classes
  - Difficiles à comprendre
  - Difficiles à réutiliser
  - Difficiles à maintenir
  - Fragiles (constamment affecter par le changement)

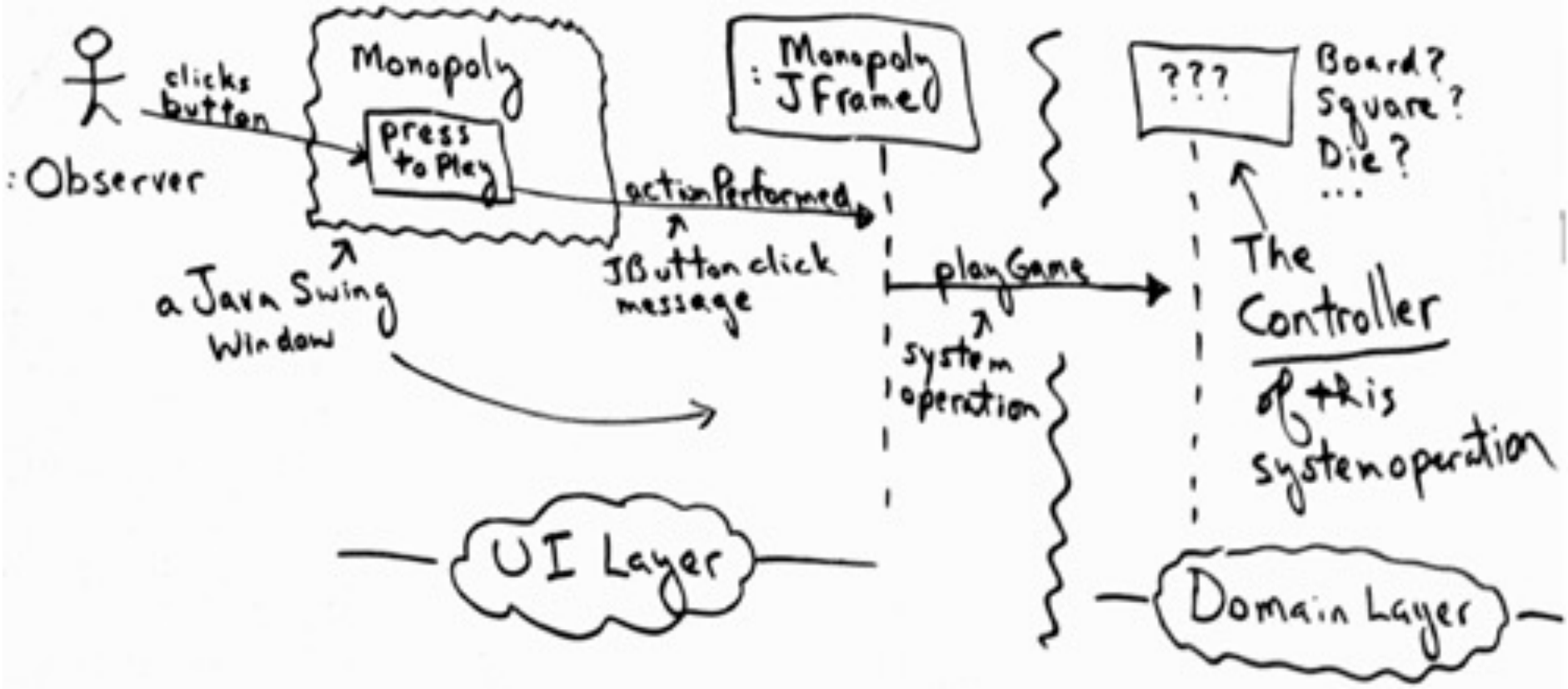
- Une classe de forte cohésion a un petit nombre de méthodes, avec des fonctionnalités hautement liées entre elles, et ne fait pas trop de travail
  - Pouvez-vous décrire la classe en une seule phrase ?

5. Contrôleur

- Problème
  - Quel est le premier objet qui coordonne le système (après l'interface homme-machine) ?

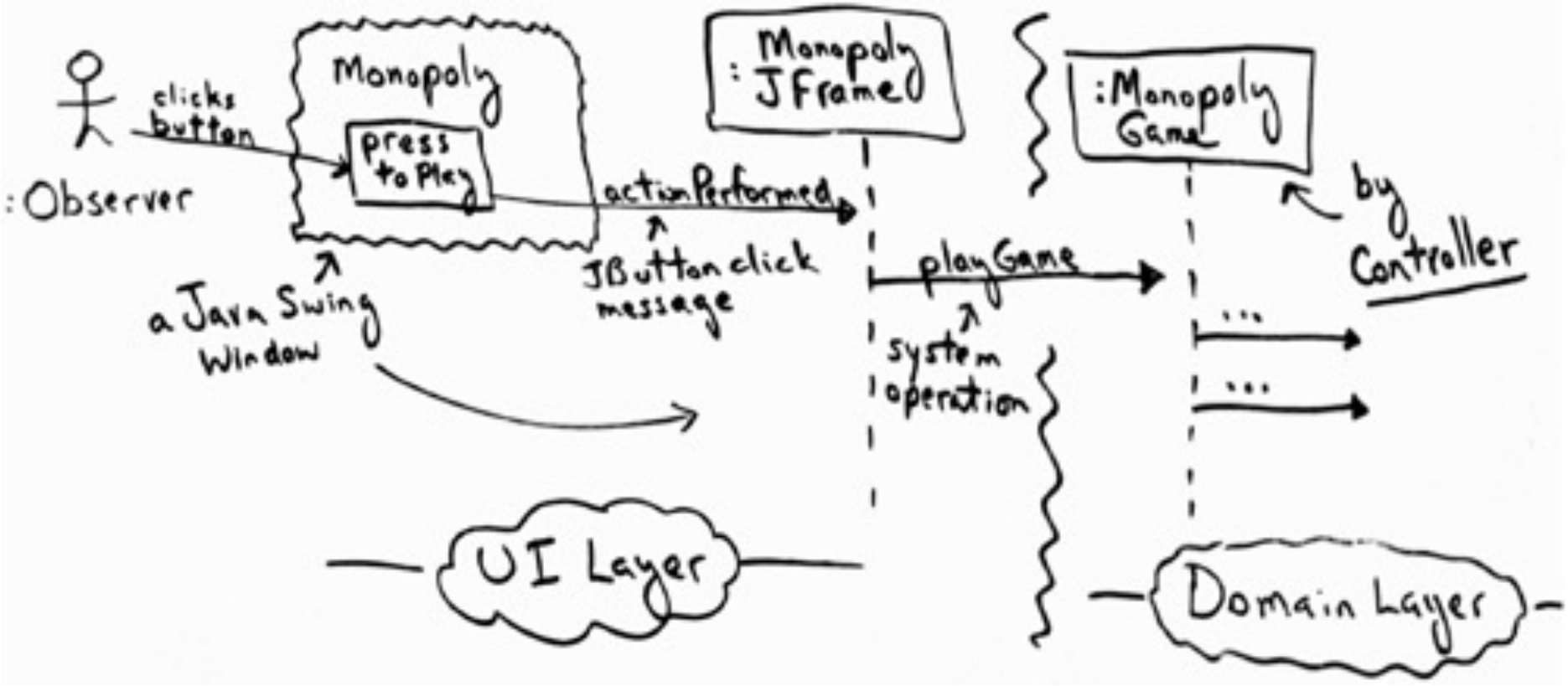
- Solution
  - Choisir (ou inventer) un objet qui endosse explicitement le rôle
    - Représente le système global ou un sous-système majeur
    - Représente un scénario d'un case d'utilisation

5. Contrôleur



Applying UML and Patterns – Craig Larman

5. Contrôleur



Applying UML and Patterns – Craig Larman

6. Polymorphisme

- Problème
  - Comment gérer des alternatives dépendantes des types ?
  - Comment créer des composants logiciels « enfichables » ?

- Solution
  - Quand les fonctions varient en fonction du type, affecter les responsabilités au point de variation
  - **Pas de if/then/else sur des instanceOf**

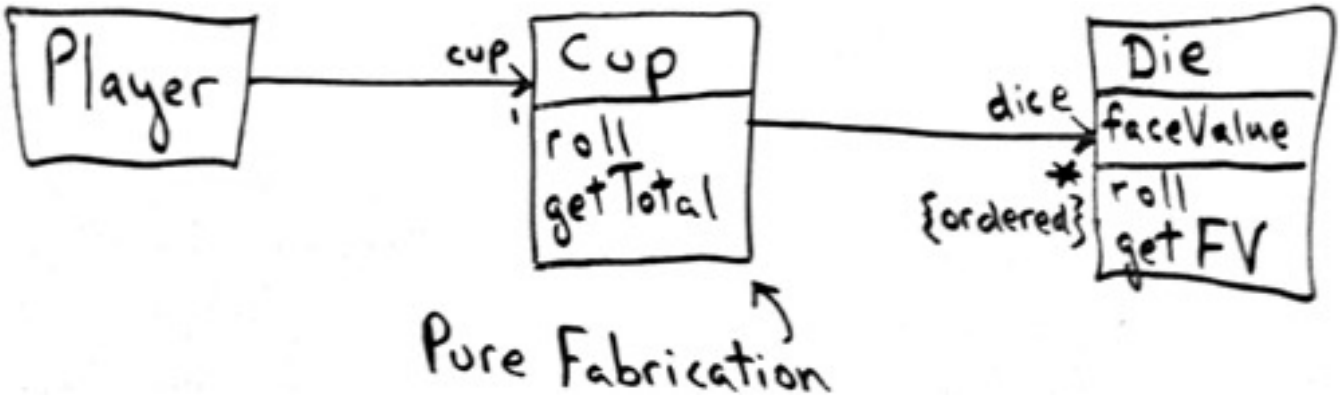
6. Polymorphisme

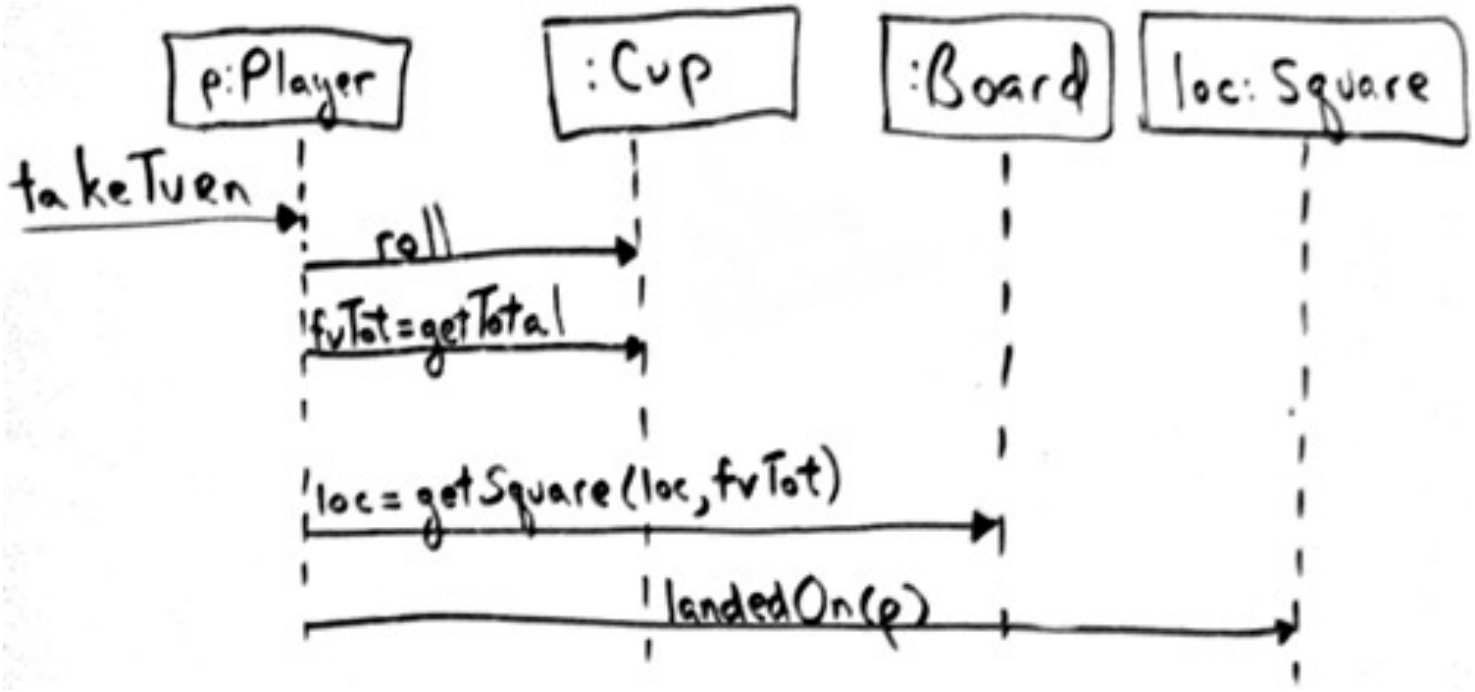
- Les cases du Monopoly ?
  - En fonction de la case sur laquelle le joueur atterrit, le comportement est différent…

7. Pure invention

- Problème :
  - Que faire quand les concepts du monde réel (les objets du domaine) ne sont pas utilisables vis-à-vis du respect d'un faible couplage et d'une forte cohésion ?
- Solution :
  - Fabriquer de toutes pièces une entité fortement cohésive et faiblement couplée

7. Pure invention





Sources

- Cours de conception orientée objets – Mireille Blay-Fornarino – IUT de Nice
- Conception GRASP et SOLID – Sébastien Mosser – UQAM Montréal
- Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition – Craig Larman
- https://jrebel.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/