Principes de bonne conception

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

GRAPS: 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)

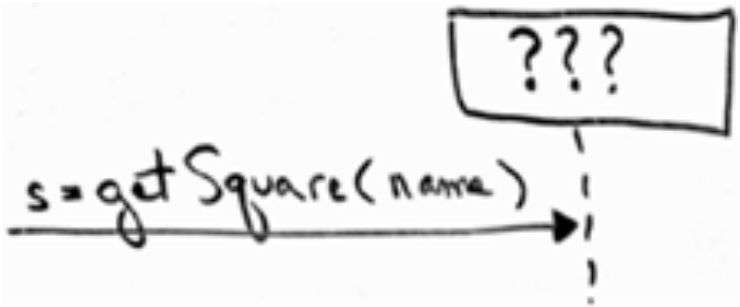
1. Spécialiste de l'information

- 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 bibliothèque, qui a la responsabilité de connaître les livres ? Qui a la responsabilité de connaître le nombre d'exemplaires disponibles d'un livre ?

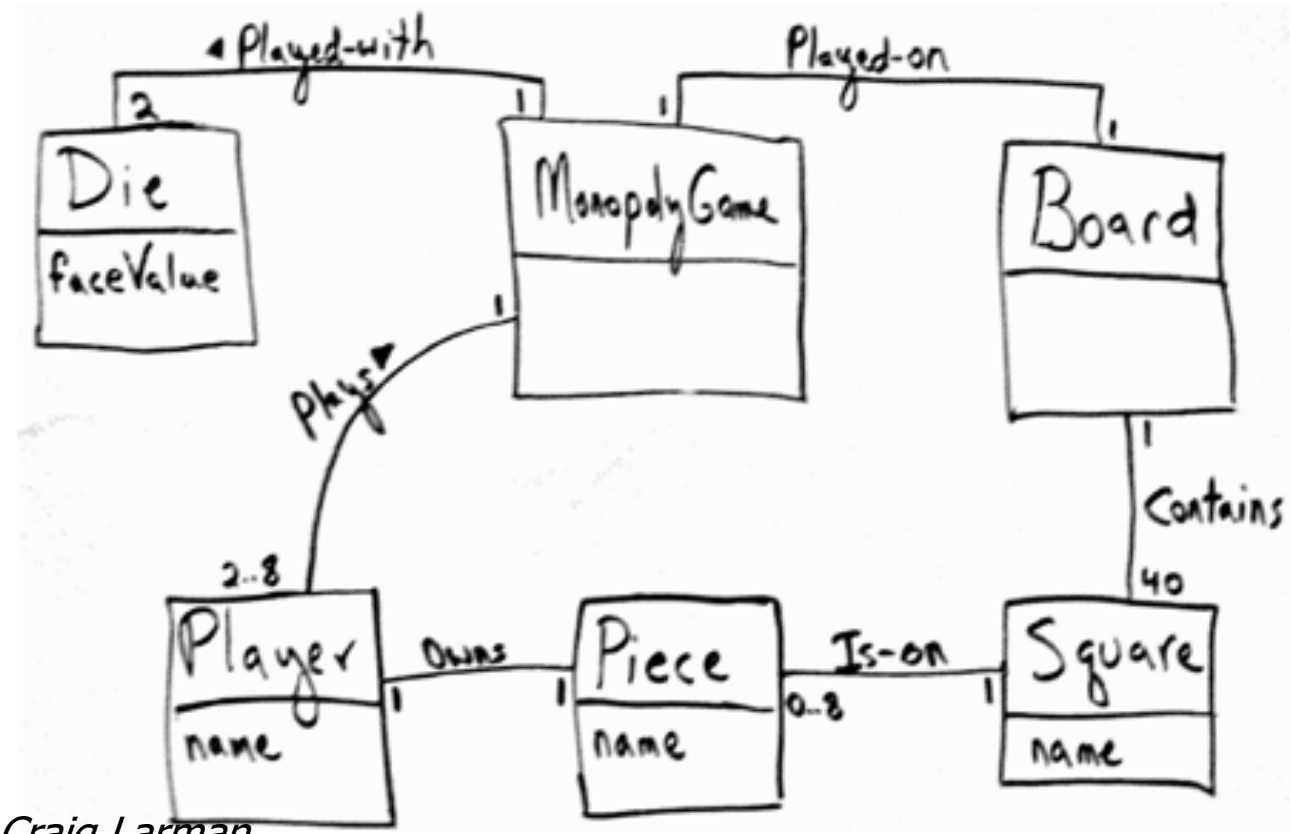
2. Créateur

- 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

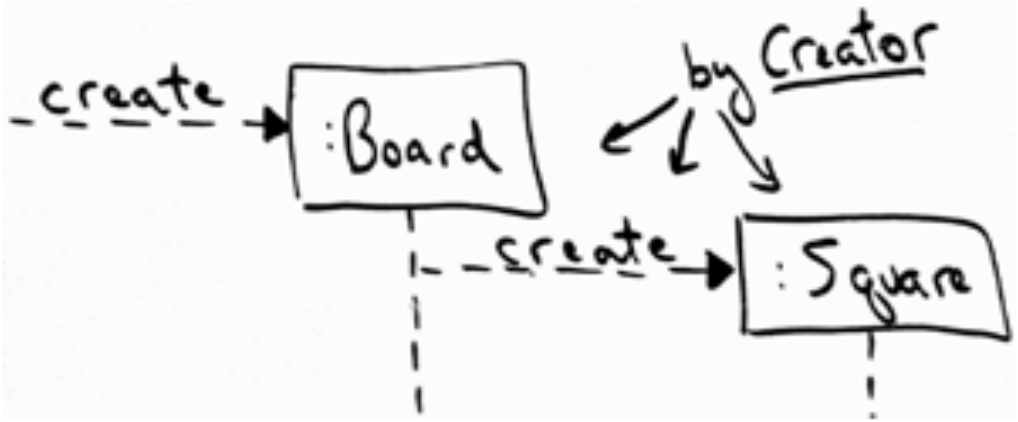
2. Créateur

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



2. Créateur

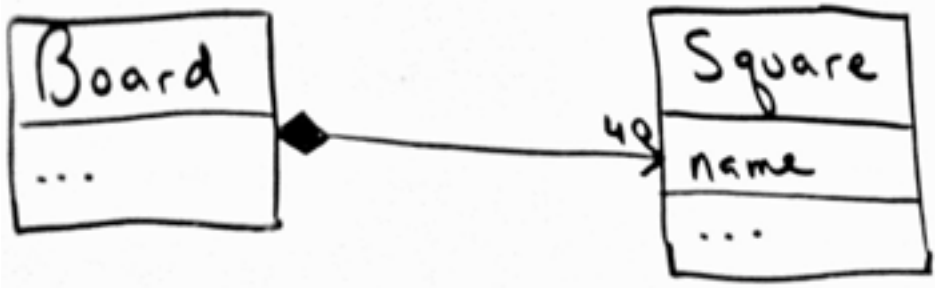
- On peut s'appuyer sur le diagramme de séquences…



Applying UML and Patterns – Craig Larman

2. Créateur

- Et l'association est donc une composition (les cases disparaissent avec le plateau) :



Applying UML and Patterns – Craig Larman

- Dans la bibliothèque, qui a la responsabilité de créer les livres ?

3. Faible couplage

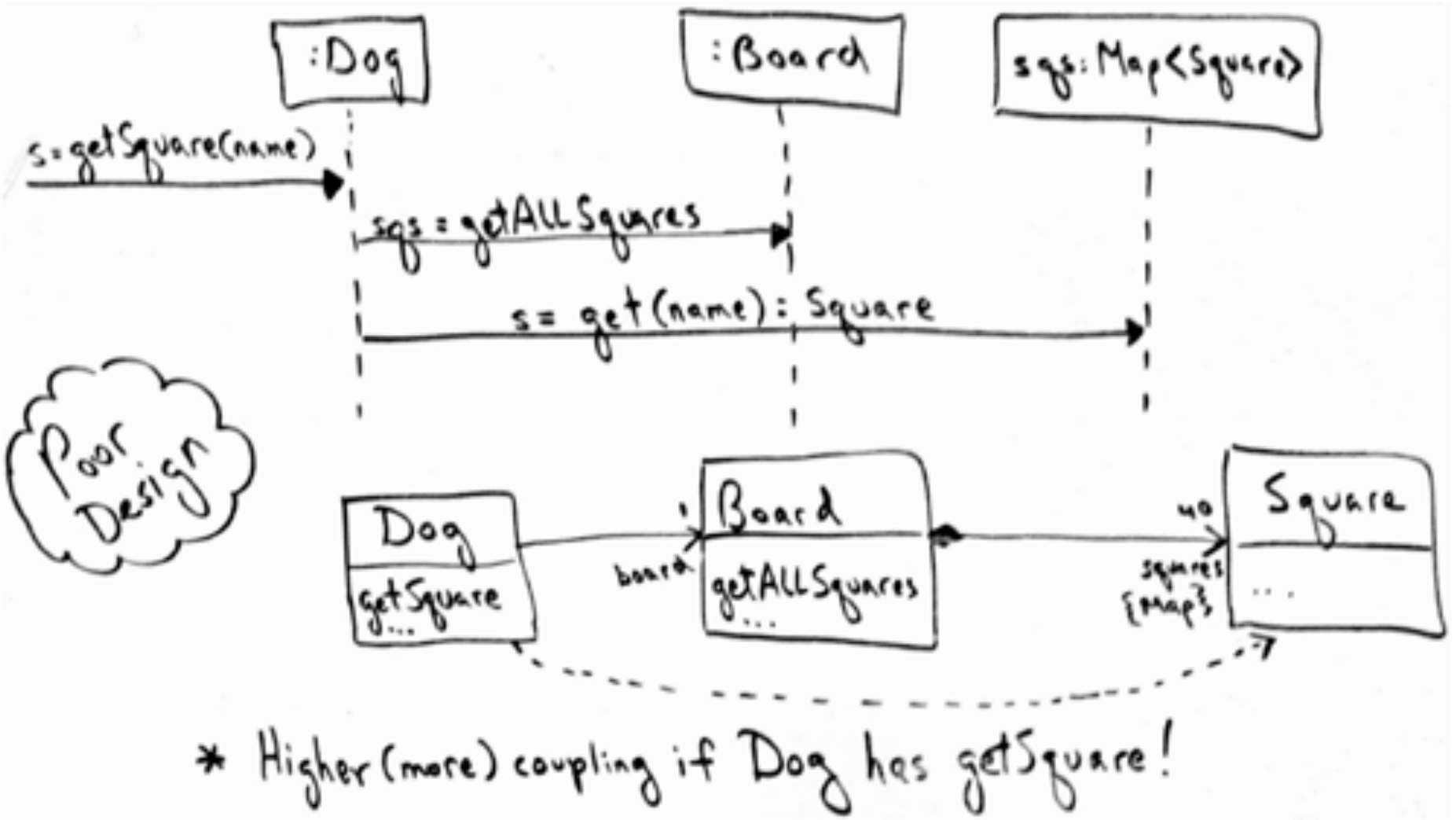
- Problème
  - Comment minimiser les dépendances, réduire l'impact des changements, et augmenter la réutilisation ?

- 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



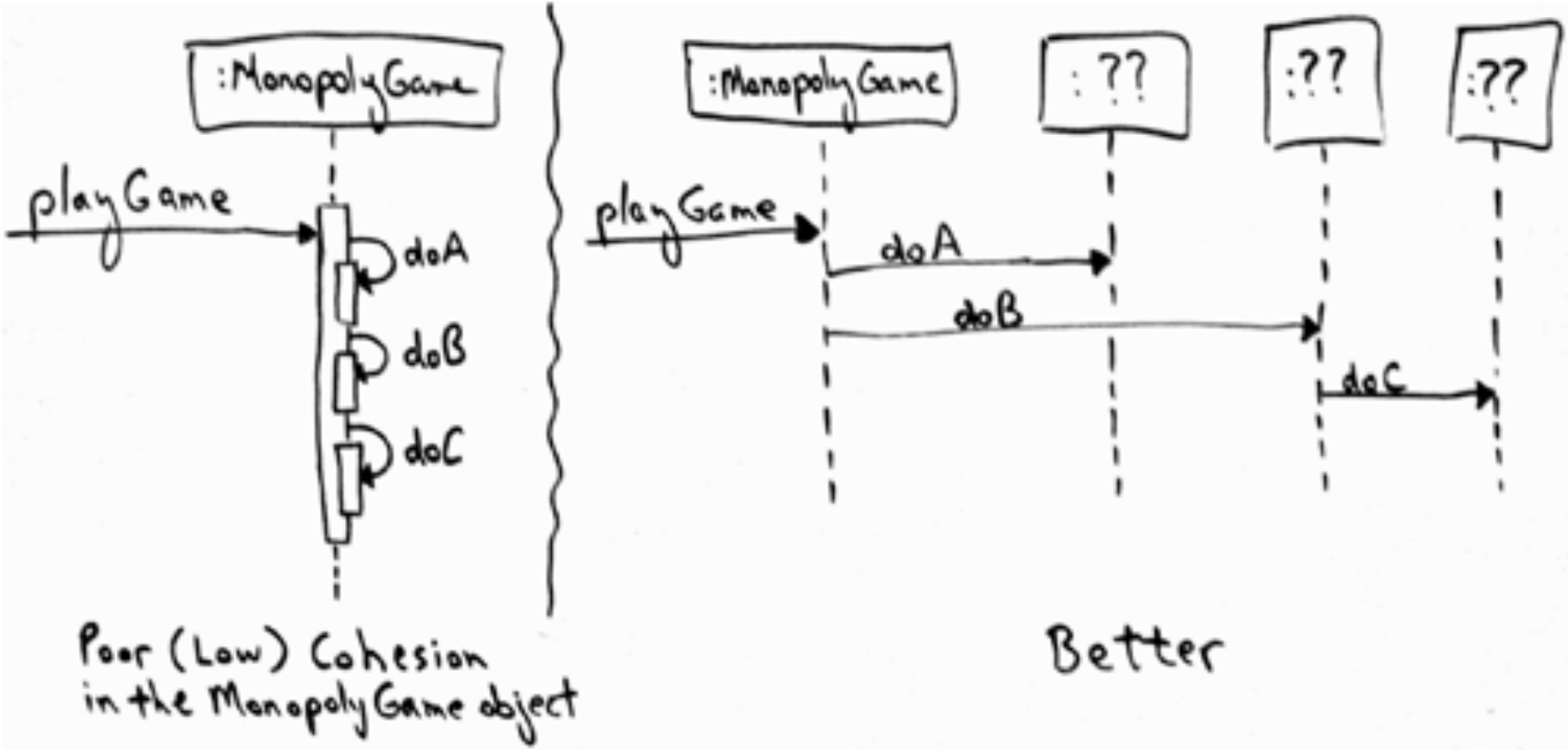
3. Faible couplage

- 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)

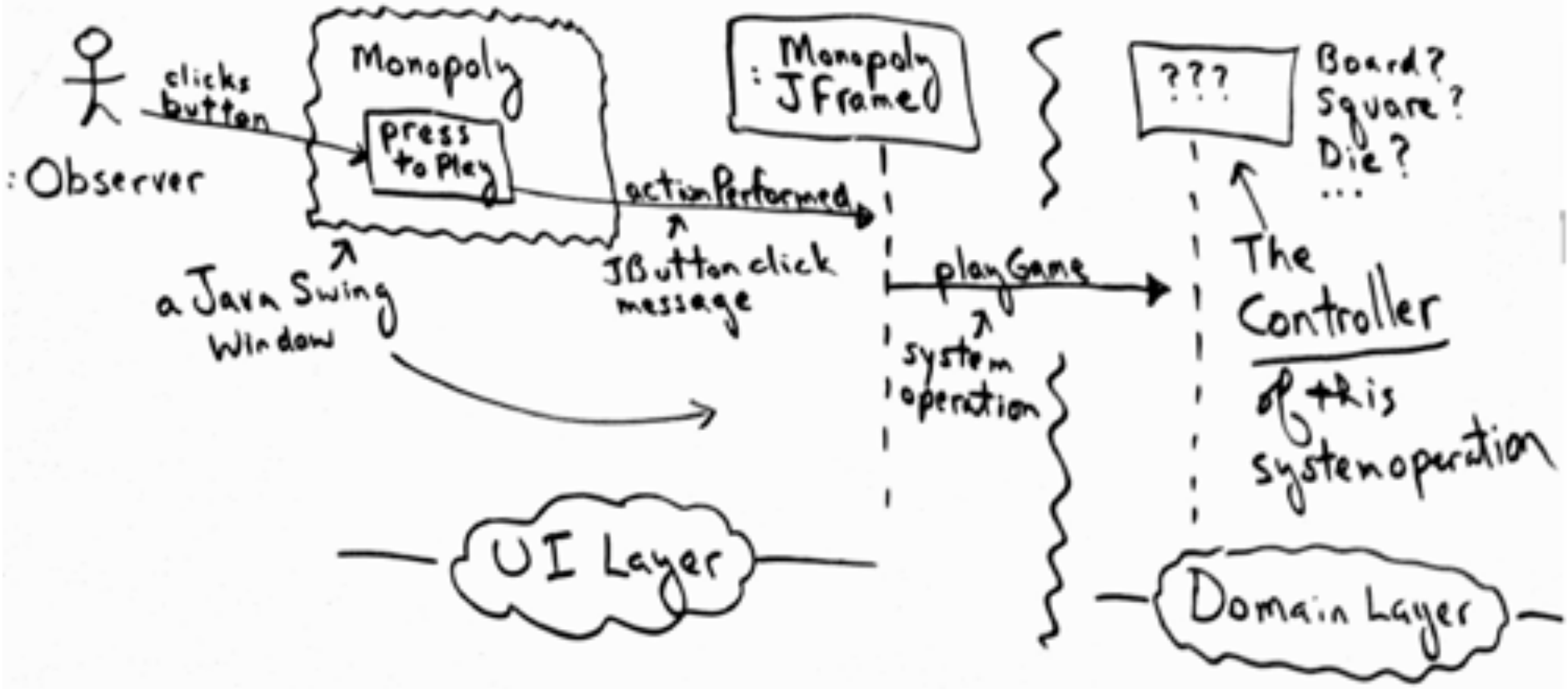
- Une classe de forte cohésion a un petit nombre de méthodes, avec des fonctionnalités hautement liées entre elles, et ne fait pas trop de travail
  - Pouvez-vous décrire la classe en une seule phrase ?

5. Contrôleur

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

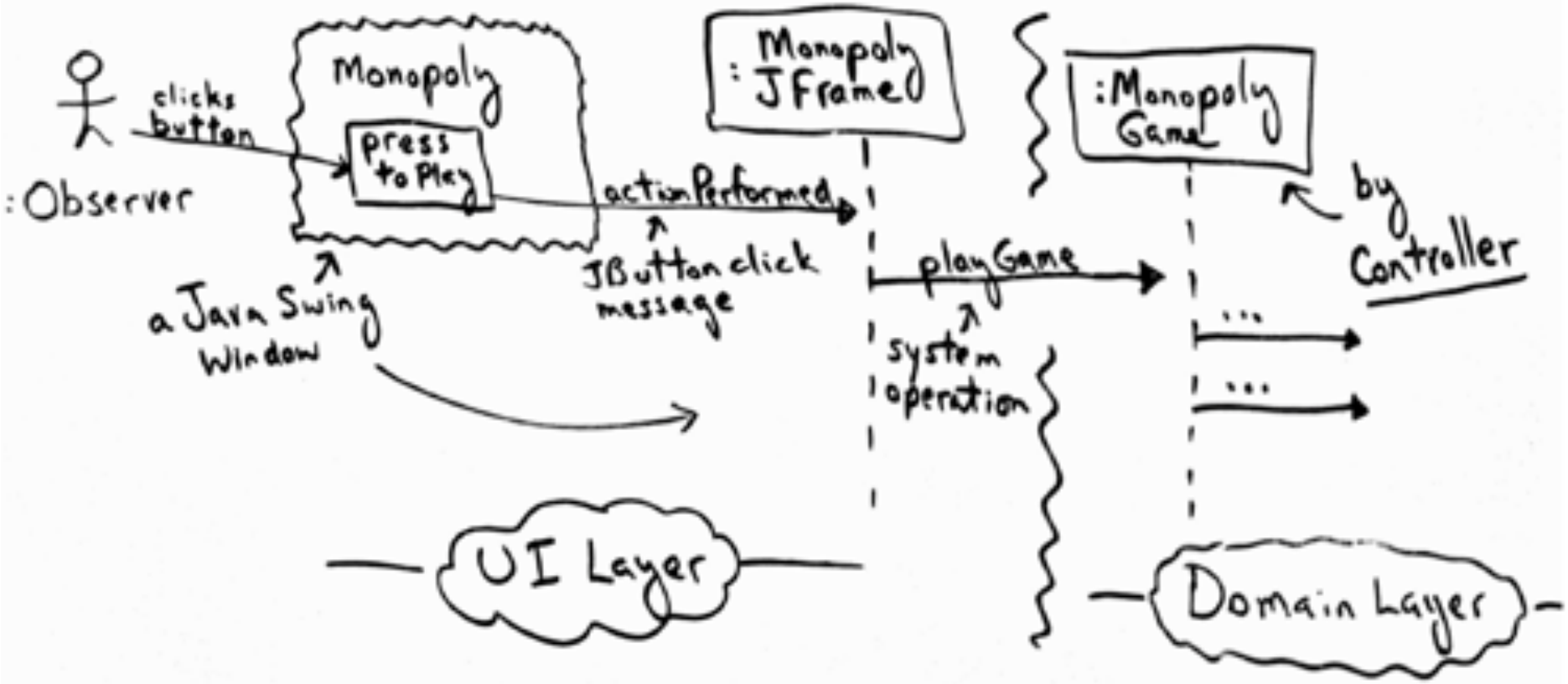
- Solution
  - Choisir (ou inventer) un objet qui endosse explicitement le rôle
    - Représente le système global ou un sous-système majeur
    - Représente un scénario d'un case d'utilisation

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**

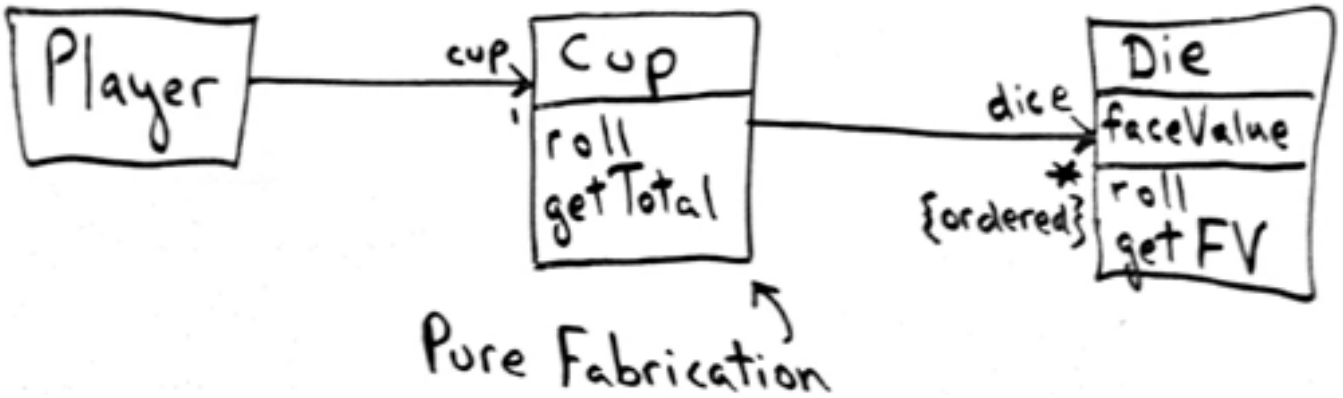
6. Polymorphisme

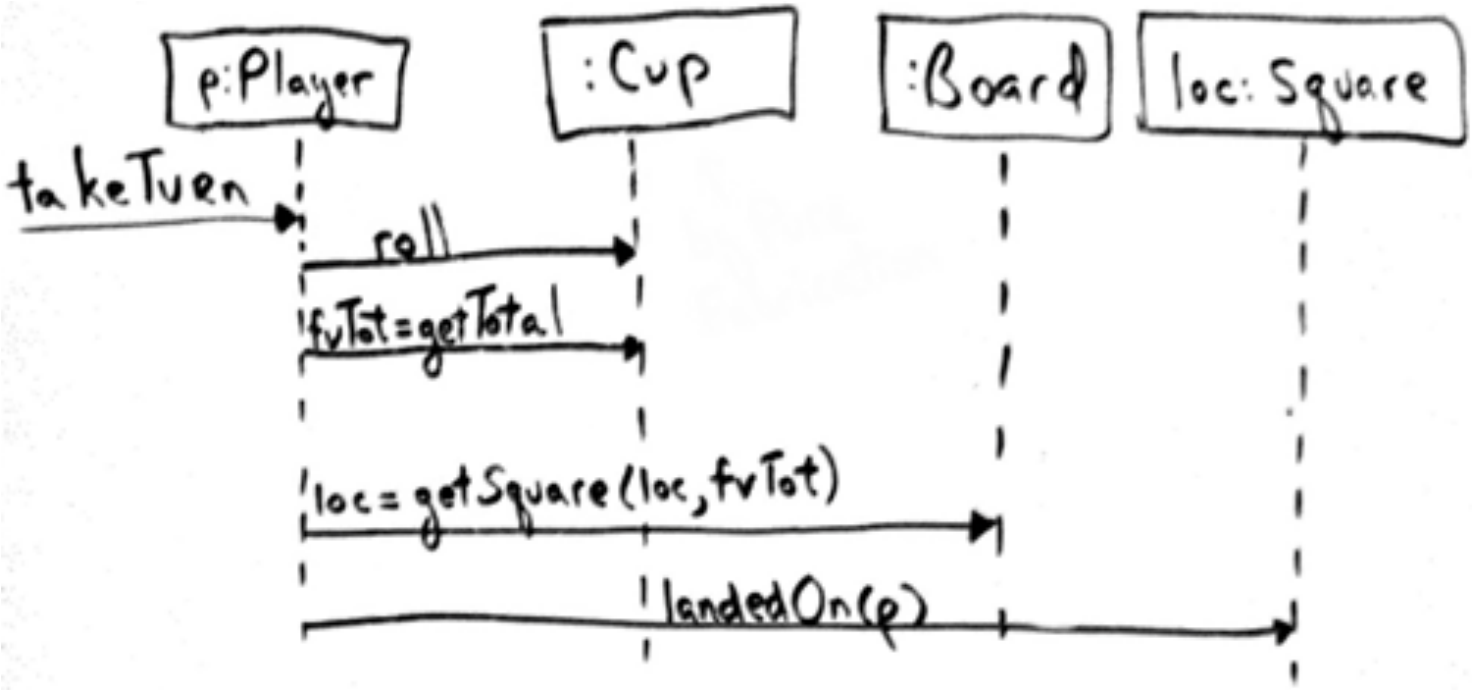
- Les cases du Monopoly ?
  - En fonction de la case sur laquelle le joueur atterit, le comportement est différent…
- Les adhérents de la bibliothèque ?
- Des exemples dans le DiceForge de l'an dernier ?

7. Pure invention

- Problème :
  - Que faire quand les concepts du monde réels (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





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/

Open/closed principle (OCP)

- Les entités logicielles doivent être ouvertes à l'extension
  - Le code est extensible
  - Par héritage
  - Par composition
- Mais fermées aux modifications
  - Le code a été écrit et testé, on n'y touche pas sauf pour corriger un bug ou effectué un refactoring avec de bonnes raisons

Open/closed principle (OCP)

- Comment ajouter la gestion des cartes de crédit ?

```
void checkOut(Receipt receipt) {
  Money total = Money.zero;
  for (item : items) {
    total += item.getPrice();
    receipt.addItem(item);
  }
  Payment p = acceptCash(total);
  receipt.addPayment(p);
}
```

Bien / Pas bien ?

```
Payment p;
if (credit)
  p = acceptCredit(total);
else
  p = acceptCash(total);
receipt.addPayment(p);
```


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

Open/closed principle (OCP)

- Solution

```
public interface PaymentMethod {void acceptPayment(Money total);

void checkOut(Receipt receipt, PaymentMethod pm) {
  Money total = Money.zero;
  for (item : items) {
    total += item.getPrice();
    receipt.addItem(item);
  }
  Payment p = pm.acceptPayment(total);
  receipt.addPayment(p);
}
```

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

Liskov substitution principle (LSP)

- Les **instances d'une classe** doivent être **remplaçables** par des instances de leurs **sous-classes** sans altérer le programme

Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T

Liskov substitution principle (LSP)

- Le contrat défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classes dérivées
- L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être substituée à la classe qu'il utilise
- *Polymorphime : la classe dérivée a un comportement différent (spécialisée)* ***mais conforme***

Liskov substitution principle (LSP)

```
public class Rectangle {
  private double height;
  private double width;

  public double area();

  public void setHeight(double height);
  public void setWidth(double width);
}
```

Dérivons une classe Square de Rectangle…

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

Liskov substitution principle (LSP)

```
public class Square extends Rectangle {
  public void setHeight(double height) {
    super.setHeight(height);
    super.setWidth(height);
  }

  public void setWidth(double width) {
    setHeight(width);
  }
}
```

Un problème ?

Quel est le contrat des setters de Rectangle ?

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

Liskov substitution principle (LSP)

```
private static Rectangle getNewRectangle(){
     // it can be an object returned by some factory ...
      if ….
            return new Square();
 }

 public static void main (String args[]){
    Rectangle r = LspTest.getNewRectangle();
    r.setWidth(5);
    r.setHeight(10);
   // user knows that r it's a rectangle. It assumes that he's
able to set the width and height as for the base class
    System.out.println(r.getArea());
```

Affichage ? 100

Intention ? 50

Interface segregation principle (ISP)

- Les clients n'ont pas à être forcés à dépendre sur des interfaces qu'ils n'utilisent pas

- Il faut privilégier plusieurs interfaces, petites, cohésives, spécifiques à des besoins clients

Interface segregation principle (ISP)

Une interface pour les messages affichés par un automate de retrait bancaire :

- Que faire si on veut en plus un message spécifique qui dit qu'il y a un coût supplémentaire en cas de retrait ?
- Ajouter une méthode dans l'interface Messenger ?

```
public interface Messenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
  tellCardWasSiezed();
  askForAccount();
  tellNotEnoughMoneyInAccount();
  tellAmountDeposited();
  tellBalance();
}
```

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

Interface segregation principle (ISP)

Découpage en plusieurs interfaces dédiées

```
public interface LoginMessenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
}

public interface WithdrawalMessenger {
  tellNotEnoughMoneyInAccount();
  askForFeeConfirmation();
}

publc class EnglishMessenger implements LoginMessenger, WithdrawalMessenger {
  ...
}
```


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

Dependency inversion principle (DIP)

- Réduire les dépendances sur les classes concrètes
  - *Program to interface, not implementation*

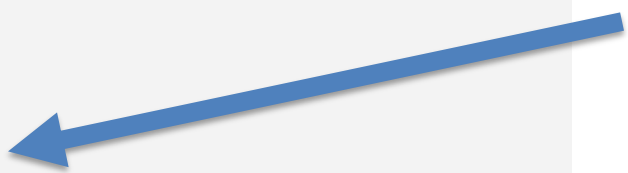
Dependency inversion principle (DIP)

```
public interface Reader { char getchar(); }
public interface Writer { void putchar(char c)}

class CharCopier {

  void copy(Reader reader, Writer writer) {
    int c;
    while ((c = reader.getchar()) != EOF) {
      writer.putchar();
    }
  }
}

public Keyboard implements Reader {...}
public Printer implements Writer {…}
```



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

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/