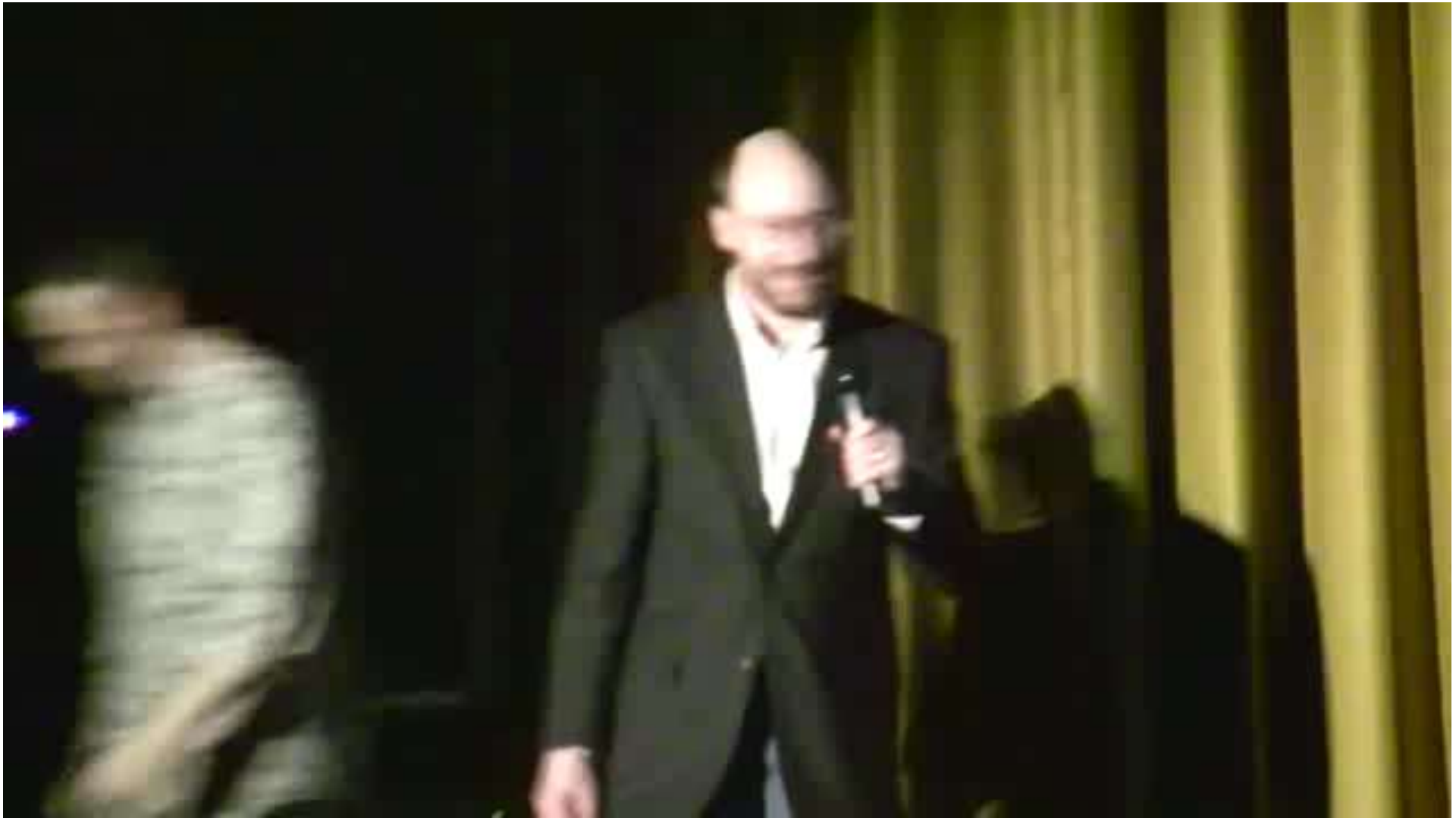
Patterns

Partie 1 : Introduction

Bienvenue en 3ème

- La dernière ligne droite avant de devenir des informaticiens compétents.
- Compétent ? Qu’est-ce que cela peut bien vouloir dire ?

The Psychology of Incompetence



Dunning–Kruger

- People tend to hold overly favorable views of their abilities in many social and intellectual domains. The authors suggest that this overestimation occurs, in part, because people who are unskilled in these domains suffer a dual burden: Not only do these people reach erroneous conclusions and make unfortunate choices, but their incompetence robs them of the metacognitive ability to realize it. Across 4 studies, the authors found that participants scoring in the bottom quartile on tests of humor, grammar, and logic grossly overestimated their test performance and ability. Although their test scores put them in the 12th percentile, they estimated themselves to be in the 62nd. Several analyses linked this miscalibration to deficits in metacognitive skill, or the capacity to distinguish accuracy from error. Paradoxically, improving the skills of participants, and thus increasing their metacognitive competence, helped them recognize the limitations of their abilities.

Que faire pour être sûr de sa propre compétence ?

- Ne pas considérer sa compétence comme acquise par défaut.
- Toujours chercher à la perfectionner.
  - Se tenir au courant des évolutions du métier.
  - Toujours apprendre.

Ce que cela veut dire en informatique

- Ce qui dirige la compétence : la meilleure pratique (en Anglais: best practice)
  - Ensemble de règles, techniques, manières de faire, solutions.
  - Correspond à la meilleure manière actuellement connue d'adresser un problème particulier.
  - Evolue au cours du temps avec les évolutions technologiques, l'expérience, la compréhension des problèmes, etc…

Les designs patterns dans tout cela

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
- Identifient des solutions récurrentes et bonnes à des problèmes récurrents en informatique.
  - Catégorisation (créationelle, structurelle, comportementale)
  - Livre : ***Design Patterns: Elements of Reusable Object-Oriented Software*** (ISBN 0-201-63361-2)

Patterns & compétence

- Les patterns donnent des solutions éprouvées pour des problèmes récurrents.
  - Un développeur compétent se doit d'être capable d'identifier ces problèmes.
  - Un développeur compétent se doit d'être capable d'implémenter la solution correspondante.
  - Un développeur compétent se doit d'être capable de communiquer aves ses pairs en utilisant le vocabulaire des patterns.
- C'est l'équivalent des gammes en solfège !

Patterns et incompétence

- Tout comme il existe des solutions récurrentes bonnes, il existe des solutions récurrentes mauvaises.
  - Anti-patterns.
- Tout développeur qui se veut compétent doit veiller activement à ne pas tomber dans les pièges des anti-patterns.
  - A utiliser comme une check-list de choses à éviter : http://en.wikipedia.org/wiki/Anti-pattern

Exemples d’anti-patterns

- **Paralysie par l’analyse : dévouer un temps disproportionné à l’analyse.**
- **Abstraction inverse : l’abstraction fournie est trop complexe, alors que l’utilisateur n’a qu’un besoin simple.**
- **Programmation spaghetti : absence de structure.**
- **Usine à gaz : structure complexe résultant en de sérieux problèmes de performance.**
- **Erreur de copier/coller : duplication du code et de ses erreurs.**
- **Réinventer la roue (carrée) : réinventer une solution (avec le risque de se tromper) plutôt que d’utiliser la solution standard déjà existante.**
- **L'objet divin : composant assurant trop de fonctions essentielles.**

http://en.wikipedia.org/wiki/Anti-pattern

Contexte des patterns : L'informatique de gestion

- Logiciels de grande envergure
  - Milliers/millions de ligne de code
  - Dizaines/centaines de développeurs
- Logiciels de longue durée de vie
  - Evolutivité
  - Maintenance
- Qualité de service
  - Correction
  - Performance

Principes généraux d'ingénierie logicielle

- Millions de ligne de code et centaines de développeurs
  - Besoin d'architecture, d'organisation
  - Diviser pour régner = principe de **Separation of concerns**
    - Le programme est divisé en sous-parties indépendantes les unes des autres.
    - Couplage lâche (loose coupling) : la réunion de ces parties est facilement modifiable car il y a peu d'interdépendances directes.
    - Haute cohésion (high cohesion) : chacune de ces parties s'occupe d'une et une seule problématique particulière, et la résoud complètement.

Principes généraux d'ingénierie logicielle

- Evolutivité et maintenance
  - Principe d'open/closed : les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension mais fermées à la modification.
    - La partie fermée à la modification est le cœur du module. Le code y change peu, est éprouvé par l'expérience, a une grande maturité.
    - La partie ouverte est facilement configurable et se greffe facilement au cœur.

Principes généraux d'ingénierie logicielle

- Correction
  - Tests (unitaire, intégration, fonctionnel, régression).
  - Complexité intrinsèque $\leftrightarrow$ accidentelle
    - Intrinsèque : dûe au problème même à résoudre.
    - Accidentelle : celle que l'on rajoute en plus.
- But à atteindre
  - Contrôler la complexité globale pour garder la complexité accidentelle aussi basse que possible.
  - Avoir du code aussi testable que possible.

Patterns

Partie 2 : Les patterns à proprement parler

Les patterns créationnelles

- But : instancier des objets !
  - En prenant en compte le separation of concern, l'open/closed, la testabilité et la complexité.

Builder

+ Separation of Concern
Open/Closed
+ Testabilité
Gestion complexité

- Construire un objet complexe à partir d'objets plus simples.
- Vous l'avez déjà rencontré : Dependency Injection et Assembly (ex. : Spring)
  - Permet de facilement remplacer un objet par un mock en vue de unit tester

Abstract Factory

Separation of Concern
Open/Closed
+ Testabilité
+ Gestion complexité

- Vous l'avez déjà rencontré.
- Permet de résoudre un problème de type
  - Méthode devant instancier un sous-type d'un type particulier, le sous-type étant un paramètre de l'appel de la méthode.

Factory method

Separation of Concern
+ Open/Closed
Testabilité
Gestion complexité

- Abstract class qui appelle une méthode abstraite quand elle doit instancier un objet
  - Les sous-classes spécifient l'objet à instancier en implémentant la méthode.

Prototype

Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré en Javascript
- La création d’objets est basé sur le clonage d’un objet passé en paramètre.

Singleton

+ Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré.
- Une seule instance pour une classe donnée.
- Attention la meilleure pratique pour le singleton : dependency injection et non pas static object !

Patterns Structurelles

- Concerne la composition des classes et objets.
  - Permet d'implémenter le separation of concern en offrant une solution pour combiner les différentes parties.
  - Permet d'implémenter le principe d'open/closed en offrant une solution pour permettre de fixer une partie du code tout en permettant son extension.

Adapter et Facade

+ Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- But : changer la signature (les méthodes publiques).
  - L'adapter change la signature d'un objet en l'encapsulant.
  - La facade présente une signature simplifié sur un ensemble complexe d'objets.

Composite

Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré : arbres binaires, listes chaînées.
- Combine plusieurs objets similaires pour se comporter comme un objet unique.

Decorator

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré : JScrollbar.
- Changement dynamique de comportement
  - Principe open/closed

Flyweight

Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Réduit le coût de création/manipulation de nombreux objets similaires.
  - A la place d'instancier plusieurs objets identiques, on n'en instancie qu'un que l'on réutilise

Patterns comportementales

- Gestion de la communication entre les objets.

Proxy

+ Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Vous l'avez déjà rencontré : RMI
- Représentant d'un autre objet pour :
  - Contrôle d'accès
  - Réduction de coût
  - Réduction de la complexité

Strategy, Command & Visitor

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Vous avez déjà rencontré le visitor en structures de données.
- Externalise une partie du comportement d'un algorithme.
  - Encapsule ce comportement dans un objet qui est appelé quand nécessaire, avec les paramètres nécessaires.
- Visitor ≈ Strategy sur un composite, avec autant de méthodes que de type de nœuds différents.
- Command ≈ Strategy qui ne reçoit pas de paramètre à l'invocation mais plutôt encapsule une commande déterminée à l'avancée mais qui devra n'être effectuée que plus tard.

Observer

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré en IHM
- L’objet observé est paramétré par des stratégies qu’il invoquera lorsqu’un événement particulier intervient.

Template Method

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Externalise une partie du comportement d'un algorithme.
  - Dans une méthode abstraite.
- Les réalisations concrètes devront fournir cette partie de l'implémentation.

Chain of responsabil

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Chaînage d'un ensemble d'objets de traitement.
  - On soumet un objet de commande à la chaîne.
  - Chaque objet traitement soit traite la commande, soit passe la commande à l'objet traitement suivant.

Interpreter

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Contexte : composition d'opérations simples (p.ex. un script d'un jeu vidéo)
  - On utilise un composite d'objets spécifiant le script d'opérations.
  - L'interpreter lit et exécute le composite.

Iterator

+ Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Vous l’avez déjà rencontré.
- Itère sur une liste.
- En Java 5, on bénéficie en plus du for each.

Mediator

+ Separation of Concern
+ Open/Closed
Testabilité
+ Gestion complexité

- Objet servant uniquement à coupler d'autres objets.
- A la place d'avoir l'objet A qui appelle les méthodes de l'objet B (et réciproquement)
  - A appelle des méthodes du médiateur M qui s'occupera d'appeler B (et réciproquement).
  - Diminue le couplage entre A et B.
  - Un mock du médiateur permet de casser complètement l'inter-dépendance entre A et B.

State

+ Separation of Concern
Open/Closed
Testabilité
+ Gestion complexité

- Soit un objet.
  - Il a un état principal et un état secondaire.
  - Les méthodes qui manipulent l'état secondaire ont un comportement différent dépendant de l'état principal.
  - On encapsule dans des classes différentes les différents états principaux possibles.
    - Ces classes fournissent le comportement unique de manipulation de l'état secondaire en fonction d'un état principal spécifique.

Suite du cours…

- Comme vous le constatez, vous connaissez déjà beaucoup de ces patterns.
- Le but de ce cours est :
  1. Maîtriser tous les patterns.
  2. Dans du code déjà existant, identifier les patterns présents.
  3. Dans du code déjà existant, identifier les patterns qui amélioreraient ce code, et les implémenter (refactoring).
  4. Pour une problématique donnée, avant l'écriture du code, identifier les patterns adéquats et les implémenter.

Méthodologie du cours

- A chaque séance sera proposé un (ou plusieurs) exercice(s) d'un des types suivant :
  - Identification d'un pattern déjà implémenté dans du code.
  - Identification d'un pattern pour la résolution d'un problème donné + son implémentation.
  - Identification d'un pattern pour l'amélioration d'un code déjà existant + son adaptation (refactoring).
- L'évaluation finale consistera en des exercices similaires.

Prérequis

- En plus d'être en 3ème bac à l'IPL, ces exercices requièrent une connaissance minimale des patterns.
- Il vous est donc demandé pour la semaine 3 de vous être familiarisé avec tous les patterns de cette présentation.
  - Vous résumerez/synthétiserez/schématiserez toutes les patterns sur une face A4.
  - Vous remettrez cette feuille au début du cours de la semaine 3 (pas remis = pas accepté au cours), avec votre nom dessus.
  - Cette feuille vous sera rendue au début de l'examen et **vous pourrez l'utiliser pendant ce dernier**.

Références utiles

- **Principes généraux**
  - http://fr.wikipedia.org/wiki/Principe_ouvert/ferm%C3%A9
  - http://fr.wikipedia.org/wiki/SOLID_(informatique)
  - http://fr.wikipedia.org/wiki/Ne_vous_r%C3%A9p%C3%A9tez_pas
  - http://fr.wikipedia.org/wiki/Couplage_(informatique)
  - http://fr.wikipedia.org/wiki/Principe_KISS#En_informatique
  - http://fr.wikipedia.org/wiki/Antipattern
  - http://fr.wikipedia.org/wiki/YAGNI

- **Patrons**
  - http://eliteserveur.eu/ebooks/Design%20patterns%20-%20T%EAte%20la%20premi%E8re.pdf
  - http://fr.wikipedia.org/wiki/Patron_de_conception
  - L'ancien syllabus est aussi disponible sur l'ecampus