Génie logiciel

Architecture

Louis-Edouard LAFONTANT

geodes

Université de Montréal

Architecture

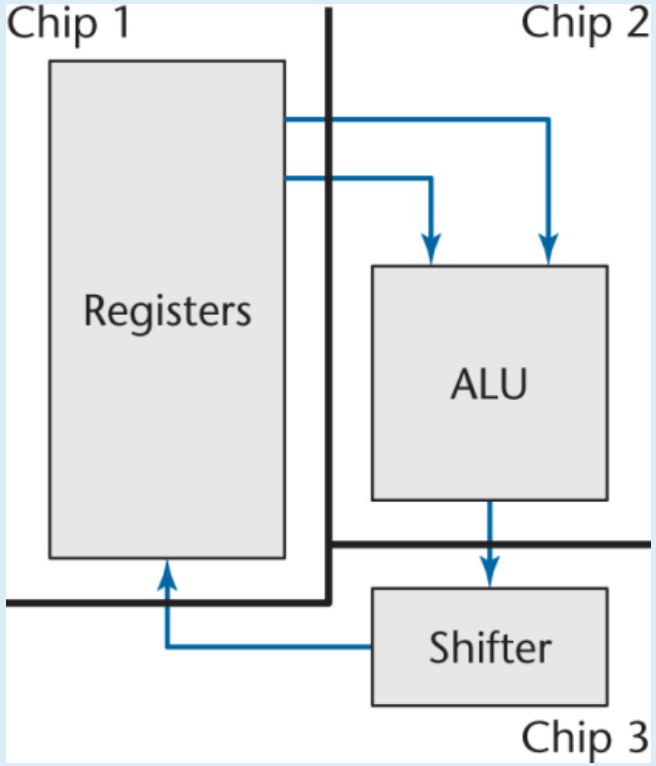
Quelle stratégie employée pour atteindre nos objectifs?

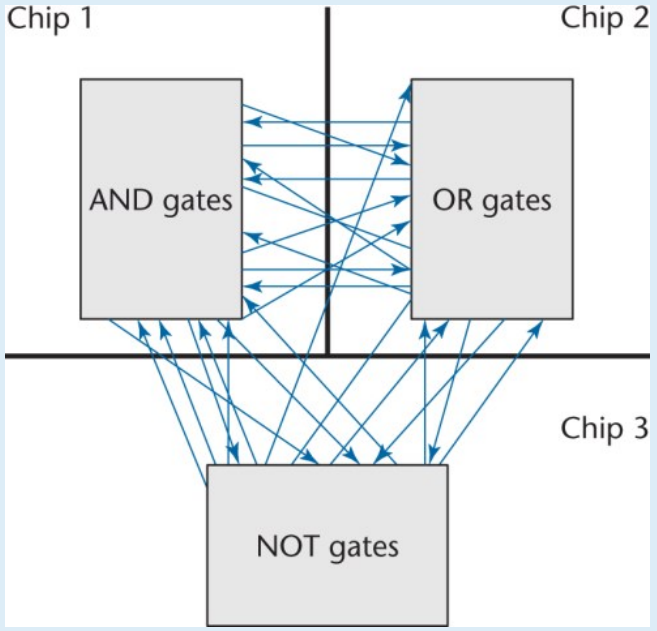
Guider par les besoins non-fonctionnels.

Conception architecturale

- Aperçu haut niveau du système
  - **Composants** principaux, leur propriétés et leurs collaborations
  - Exigences **non-fonctionnelles** dirigent l'architecture
    - Portabilité, fiabilité, robustesse, sécurité, etc.
- Conception des **interfaces utilisateurs** du logiciel
- Conception des **bases de données**
- Conception des **points de contrôle** du logiciel
  - Correction, sécurité, tolérance de fautes, protection des données
- Conception du **réseau**: communication entre processus distribués
- Allocation de chaque composant aux **ressources matérielles**

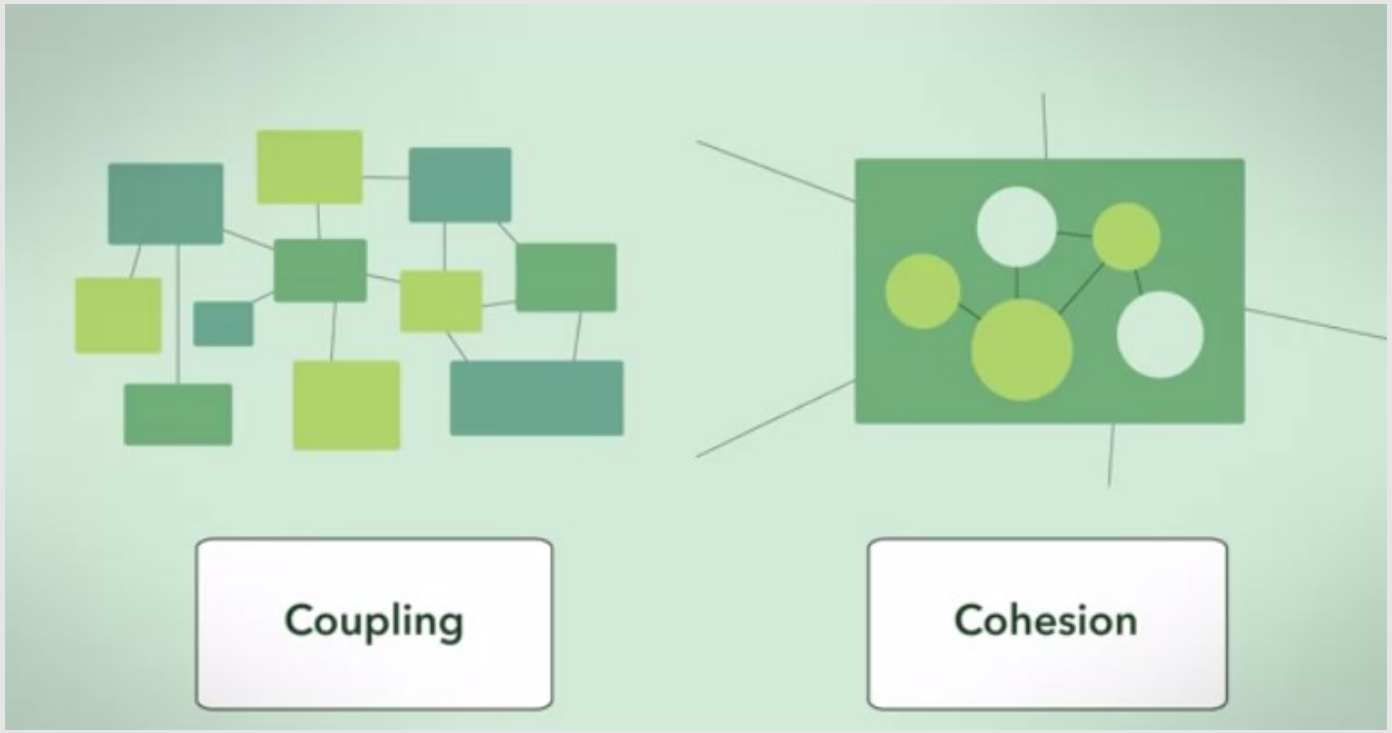
Conception modulaire vs. monolithique





- Les deux designs sont équivalent d'un p.d.v. fonctionnel
- Le 2e est difficile à
  - Comprendre
  - Localiser les fautes
  - Étendre ou améliorer
  - Réutiliser dans un autre produit
- Les modules doivent être comme le premier design
  - **Maximiser** les relations **au sein** du module
  - **Minimiser** les relations **entre** les modules

Couplage et cohésion



Décomposer le design en modules de sorte à **maximiser les interactions au sein du module** et **minimiser les interactions entre les modules**

- **Cohésion** d'un module: Degré d'interaction au sein du module
- **Couplage** d'un module: Degré d'interaction entre les modules

Couplage et cohésion dans le code

```
// Deux choses se passent, pas évident
// Couplage fort
int x = lireVecteur(v);


// Compréhensible mais pas réutilisable
int x = lireVecteurEtCalculerSomme(v);


// Deux actions séparées et avec des
// bons noms
// Forte cohésion
var V = lireVecteur(v);
int x = calculerSomme(V);
```

Couplage et cohésion en orienté-objet

- `Car` permet d'accéder à son attribut `speed`: couplage fort
- Comment résoudre ce problème ?
  - Ajouter une méthode `faster(:int)` dans `Car`
  - Changer speed via une interface publique

```
Driver john = new Driver();
Car ford = new Car("Ford", "red");

public class Driver {
  Car myCar;
  public void goFaster(int speed) {
    myCar.speed += speed;
  }
}
```

Bon couplage

- Ne pas baser la dépendance sur des structures complexes mais sur des paramètres homogènes
  - Simples paramètres ou structures de données dans lesquelles tous les éléments sont utilisé par le module invocateur
- Exemples
  - `afficherHeureArrivée(noVol);`
  - `Multiplication(no1, no2);`
  - `chercherTâchePrioritaire(fileTâches);`
- Conséquence d'un couplage fort entre modules M et N
  - Changer M nécessite de changer N
  - Si le changement dans N n'est pas fait, il y aura des fautes dans M

Bonne cohésion

- Module effectue des **actions**,
  - chacune ayant son **propre point d'entrée**,
  - avec du **code indépendant** pour chaque action,
  - toutes effectuées sur la **même structure de donnée**
- Type de donnée abstrait
  - Inné au bon usage du paradigme orienté-objet

```
class EmployeeData {
  String name;
  int eId;
  String position;
  double salary;
}
```

```
class EmployeeManager {
  HashMap<int,EmployeeData> data;
  void add(EmployeeData ed) {
    data.put(ed.eid,ed);
  }
  void remove(EmployeeData ed) {
    data.remove(ed.eid);
  }
  void update(EmployeeData ed) {
    if (data.get(ed.eid) != null)
      data.replace(ed.eid, ed);
    else throw Exception();
  }
}
```

Bonne conception

- **Faible couplage** et **forte cohésion**
- **Réutilisation**
  - Réutilisation d'un composant d'un produit pour faciliter le développement d'un autre produit ayant des fonctionnalités différentes
- **Conception modulaire**
  - Décomposition en modules indépendants et interchangeables qui contiennent tout ce qui leur est nécessaire pour exécuter un aspect d'une fonctionnalité
    - Modules cohésifs faiblement couplés
    - Modules réutilisables
- **Conception évolutive**
  - Minimiser l'effort d'apporter des modifications au logiciel après le développement initial

Mauvaise conception

- **Rigidité**
  - Logiciel difficile à modifier car chaque changement impacte beaucoup trop de parties du système
    - ➢ Diminuer le couplage
- **Fragilité**
  - Quand on effectue une modification, des parties imprévues du système ne fonctionnent plus
    - ➢ Construire des modules indépendants
- **Immobilité**
  - Difficile de réutiliser un composant dans une autre application car on ne peut la démêler de l'application courante
    - ➢ Conception et programmation modulaire

Réutilisation logiciel

Réutilisation

Réutilisation **opportuniste** (accidentelle)

- Construction du système
- Durant le développement, des parties du systèmes sont identifiées comme réutilisables et sont utilisées dans le produit

Réutilisation **systématique** (délibérée)

- Construction de composants réutilisables
- Système est construit en assemblant ces composantes
  - **Programmation orientée composants** *(Component-based design)*

Quand réutiliser?

- A-t-on des composants **développés à l'interne** disponibles pour implémenter cette exigence ?
- Y a-t-il des composants ou librairies **en vente libre** disponibles pour implémenter cette exigence ?
- Les **interfaces** pour les composants disponibles sont-elles **compatibles** avec l'architecture du système ?

Pourquoi réutiliser?

- Mettre les produits le plus rapidement sur le marché
  - Pas besoin de concevoir, implémenter, tester et documenter une composant réutilisé
- But de la réutilisation: **faciliter la création de nouveau logiciel.**
  - Extensions
  - Modifications
  - Corrections

Obstacles à la réutilisation

Syndrome du « **pas inventé ici** »

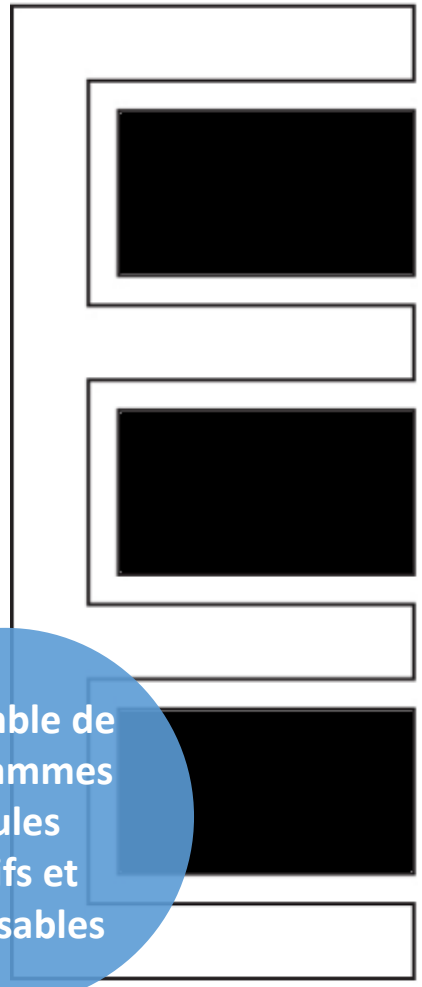
Crainte de **défauts** dans des routines potentiellement réutilisables

Questions relatives à la **gestion d'une librairie** de composants réutilisables

Coût de la réutilisation

Problèmes **légaux**

Manque **d'ouverture du code source** des composants commerciales disponibles



Ensemble de programmes /modules cohésifs et réutilisables

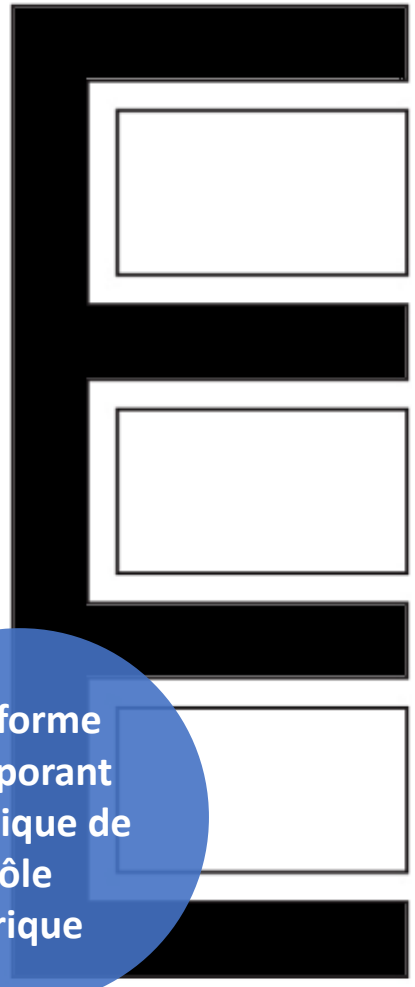
Librairie

jQuery write less, do more.

log4net

Leaflet

NumPy



Plateforme incorporant la logique de contrôle générique

Framework

django

Rocket

.NET Core

spring

Composants encapsulés avec interface concise et rigoureuse

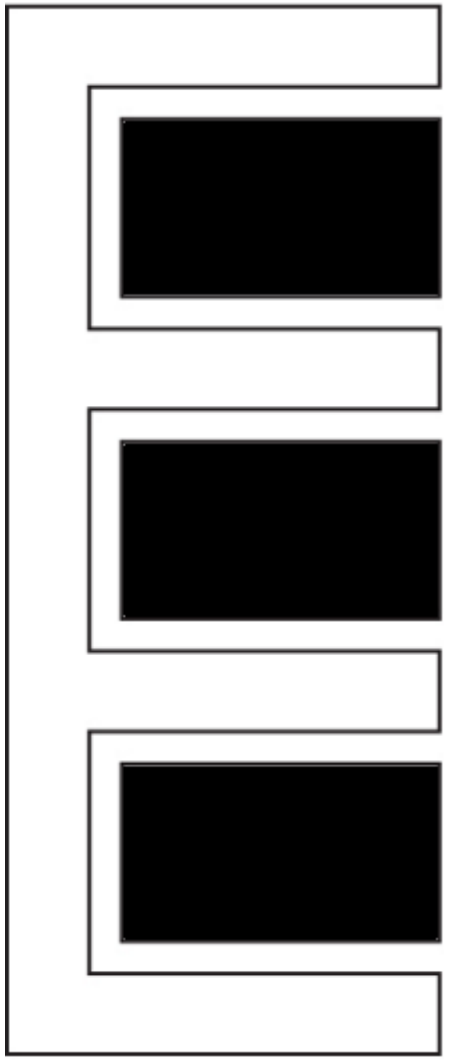
Programmation orientée composant

Symfony

aurelia

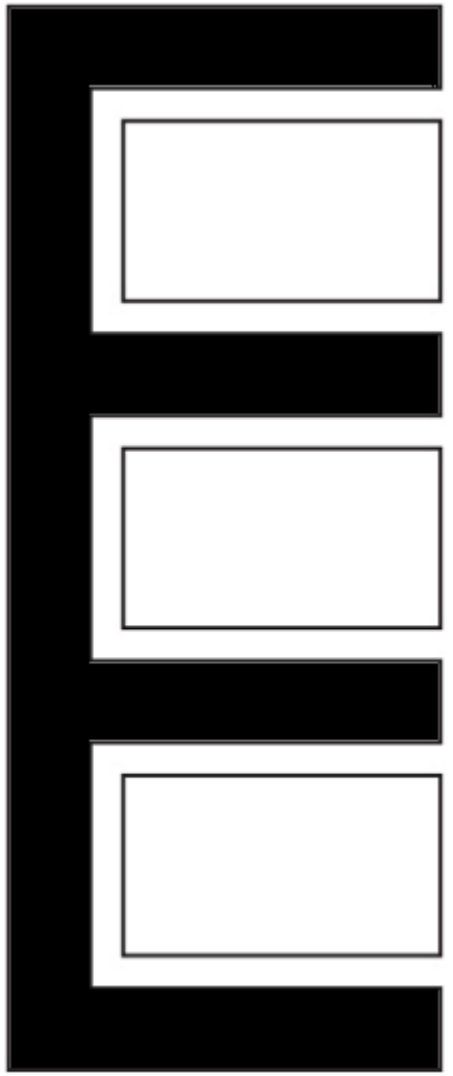
Librairie

- Ensemble de programmes/modules réutilisables
  - Ex: logiciel scientifique, librairie de classes pour le GUI
- Utilisateur de la librairie est responsable de la logique de contrôle (partie blanche)
- Une librairie fournit une API pour l'utiliser
  - Votre code fait des appels au code de la librairie



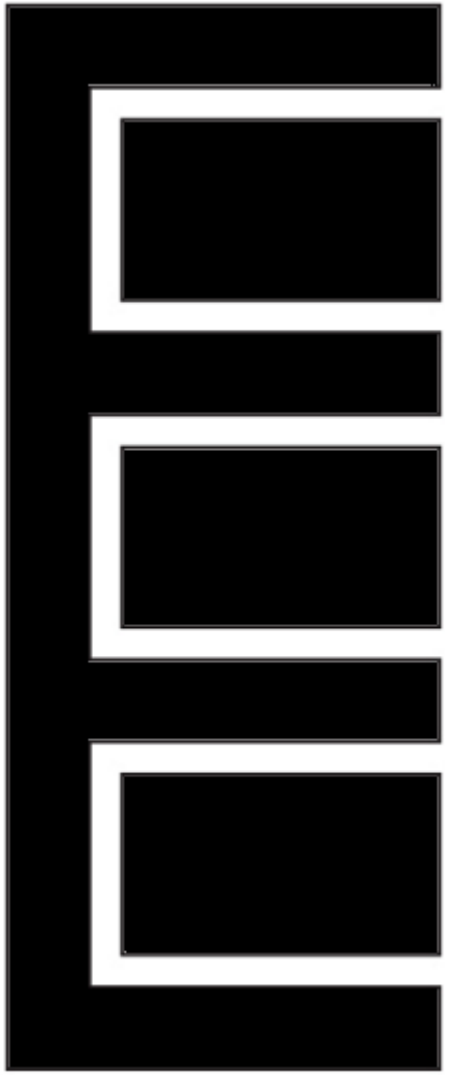
Framework

- La plateforme incorpore la **logique de contrôle générique**
- Utilisateur personnalise les fonctionnalités génériques pour les besoins de l'application
  - **Configuration**
- Utilisateur insert des programmes spécifiques à l'application (blanc dans la figure)
  - **Plug-in**
- Une plateforme fournit une API à laquelle votre application doit se conformer
  - La plateforme **fait appel** à votre code



Programmation orientée composants

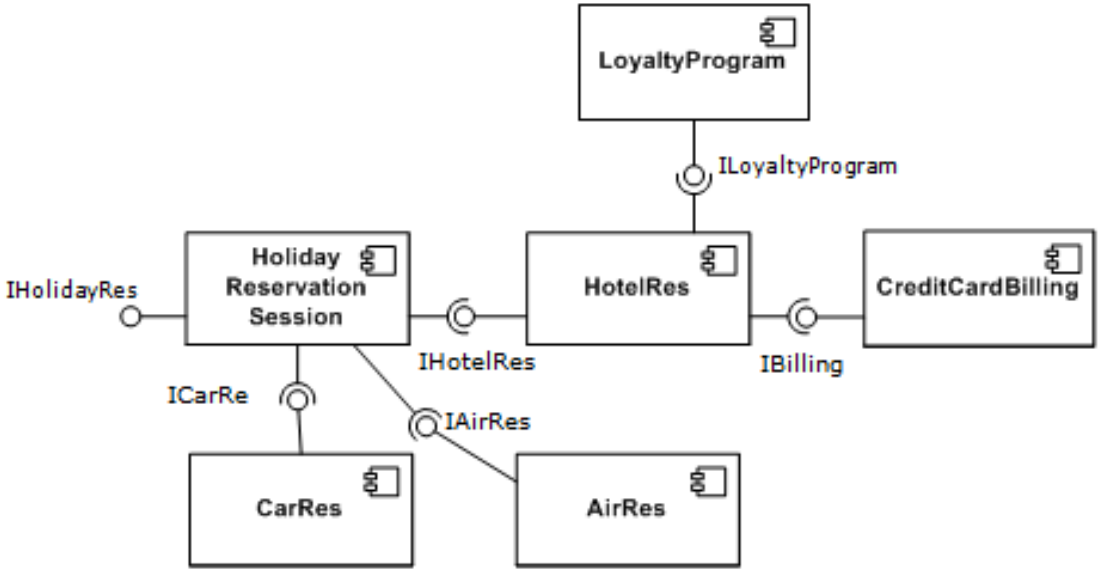
- Combine les deux approches
- Mise en place d’une plateforme applicative
- Identification des librairies réutilisables
- Tâche est d’emballer les librairies dans du code que la plateforme peut invoquer



Programmation orientée composants

- Concevoir en assemblant des composants fortement encapsulés avec une interface concise et rigoureuse

- But: maximiser la **réutilisation**



Intégration de composants

- Quelle est la difficulté d'**intégrer** ce composant dans le système actuel
  - **Adaptation** entre système et nouveau composant
- **Interface** avec l'architecture et l'environnement externe
- **Échange des données** compatible entre les composants
- Déterminer les **activités communes** à plusieurs composants
  - Manipulation et gestion des données
- Méthode de gestion et accès aux **ressources matérielles** cohérente pour tous les composants de la librairie

Prévoir de nouvelles intégrations

- Introduction d'un composant qui encapsule l'implémentation d'une **fonctionnalité** ou d'un **service**
- Système doit offrir une série de **points d'ancrage** qui permettent d'adapter le système et le faire évoluer
- **Minimiser** les modifications au système