

IFT 2255 – Génie logiciel

Conception

Des exigences à la conception

- Une fois que les exigences sont comprises et claires, la **transformation** de l'analyse à la conception peut commencer
- Étape difficile
 - Transforme des intangibles en intangibles
 - Exige de la créativité
 - Le processus de conception n'est pas algorithmique
 - Compromis pour atteindre les différents buts de la conception
- Le workflow de conception détermine la **structure interne** du logiciel: **comment** il va atteindre ses buts
- Bonne conception contribue à la **qualité** du logiciel
 - Fiabilité, correction et facilité d'évolution/maintenance

Types de conception

- Conception **architecturale** (haut niveau)
 - Définit la **structure et l'organisation générale** du logiciel
 - Décrit les principaux modules, relations entre eux et contraintes
- Conception **détaillée** (bas niveau)
 - Réalise chaque **cas d'utilisation**
 - Respecte le plan de la conception architecturale
 - Décrit le **fonctionnement interne** de chaque module
 - Prépare l'implémentation dans un langage de programmation

Objectifs d'une bonne conception

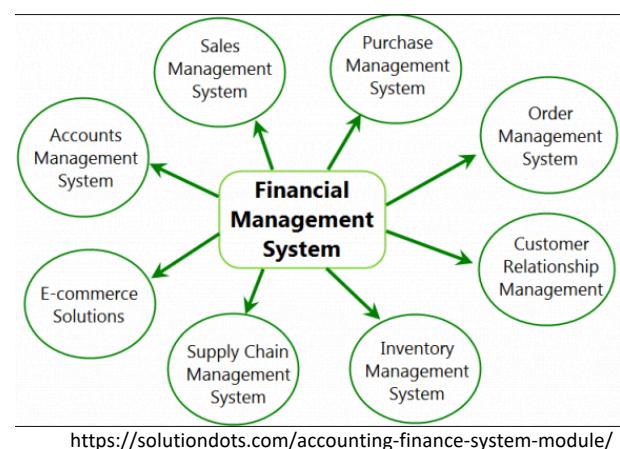
- **Forte cohésion**
 - Éléments ne sont pas réunis dans un même module par hasard, ils forment un tout pour réaliser une tâche
- **Faible couplage**
 - Modules sont relativement indépendants : dépendent le moins possible des éléments d'autres modules
- **Abstraction**
 - Décomposition intuitive exprimée en terme du problème, qui permet de se concentrer sur un module à la fois
- **Encapsulation et dissimulation d'information**
 - Détails d'implémentation propices à changer sont cachés derrière une interface stable

Conception se soucie de la maintenance

- Conception doit tenir compte
 - Des besoins **existants**
 - Des besoins **à venir**
- Obtenir une conception qui facilite l'adaptation du logiciel aux changements
 - Capacité d'**évolution**
 - **Anticipation** du changement
- Types de changements
 - Fonctionnalité, algorithme, représentation des données, périphériques, environnement, processus de développement

Décomposition du système

- Décomposer le système en **petits modules** qui sont individuellement plus facile à concevoir et implémenter
 - MSG est trop petit pour être décomposé
- Techniques de décomposition :
 - **Diviser pour régner (*divide-and-conquer*)**
 - **Affinements successifs (*stepwise refinement*)**
- Si les sous-systèmes sont indépendants, ils peuvent être implémentés par des équipes travaillant en parallèle
 - Livraison plus rapide



Décomposition fonctionnelle

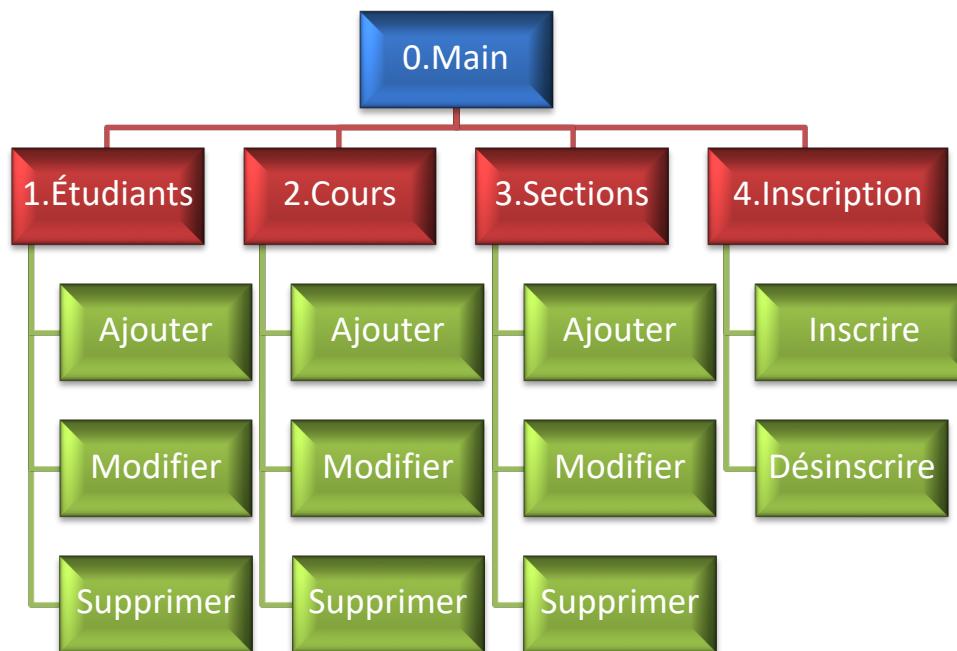
- Décomposer le système en modules par fonctionnalité

Exemple:

- Concevoir un système qui gère l'inscription à un cours
- Les exigences sont:
 - Modifier et supprimer des étudiants de la base de donnée
 - Modifier et supprimer des cours de la base de donnée
 - Ajouter, modifier et supprimer des sections d'un cours
 - Inscrire et désinscrire des étudiants d'une section
- Quels sont les modules nécessaires et comment décomposer le système ?

Décomposition fonctionnelle

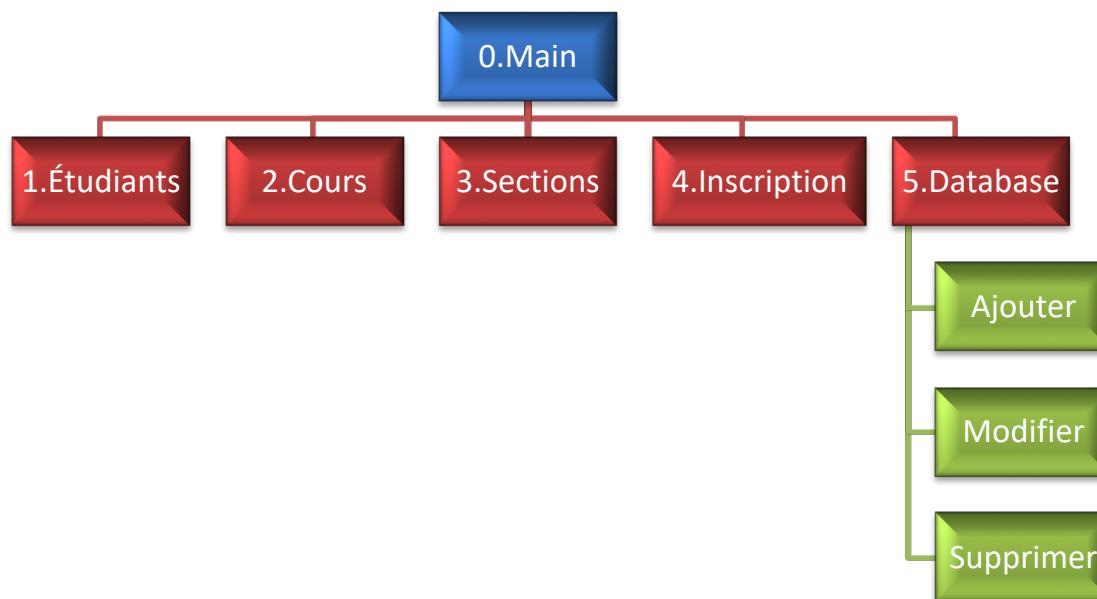
Décomposition en modules



Décomposition fonctionnelle

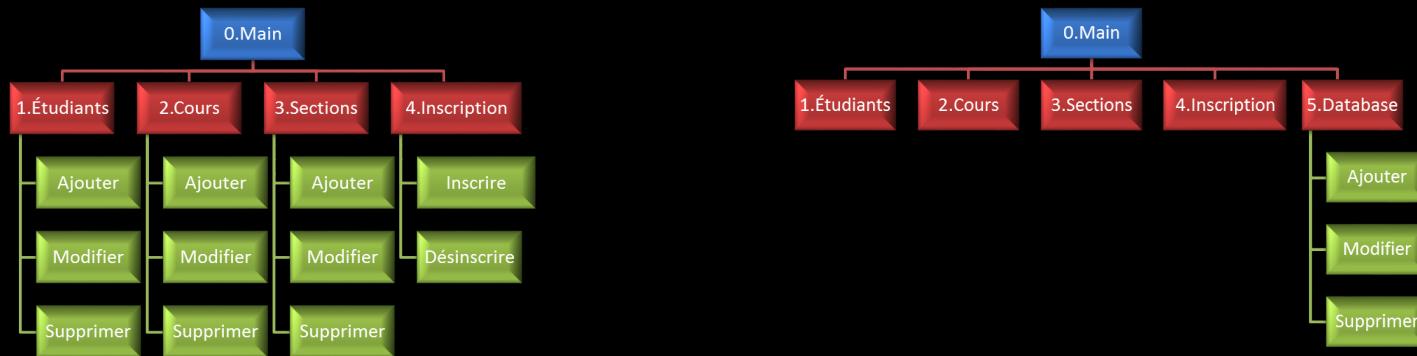
Restructurer le design

- Introduction d'un nouveau module



QUESTION

Lequel de ces deux décompositions est-il meilleur ?



- Le premier a moins de modules, mais plus de répétition
 - Opérations encapsulée dans chaque module
 - Place à l'optimisation
- Le second a plus de modules, mais encapsule toute la logique dans un seul module
 - Meilleur réutilisation et facilite la maintenance
- Une alternative est d'avoir le second, mais chaque module emballé ses opérations où toutes la logique spécifique au module est encapsulée. Les opérations de la BD deviennent des opérations CRUD génériques.

Qu'est-ce qu'un module ?

« *Une séquence lexicalement contiguë d'instructions de programme, délimité par des éléments frontières, accessible par un identifiant global.* »

[Constantine 1979]

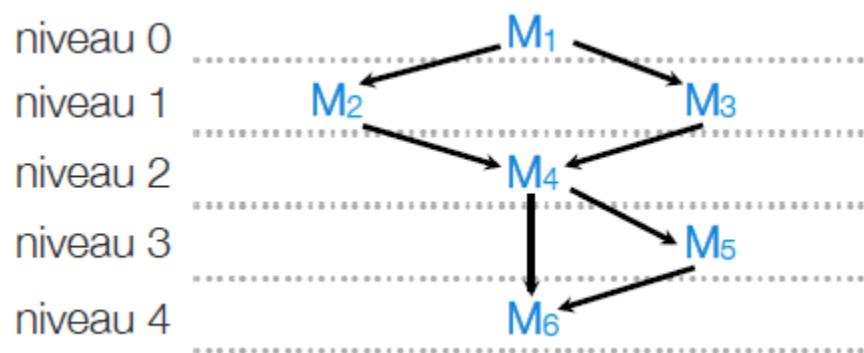
- *Lexicalement contiguë* = adjacent dans le code
- *Éléments frontières* = encapsulation
 - { ... }
 - Une méthode
 - Une classe
 - Un objet
 - Un paquet, etc.
- *Identifiant global* = nom représentatif de tout le module

Module

- Unité fournissant des ressources et/ou des services
 - Morceau de code indépendant munie d'une interface bien définie avec le reste du système
- **Système est partitionné en modules distincts**
 - $S = \{M_1, M_2, \dots, M_n\}$
- Relations entre les modules $r \subseteq S \times S$
 - **Relation « spécialise »** : raffine les détails du module plus général
 - **Relation « contient »** : les modules contenus font partie du contenant
 - **Relation « utilise »** : fait référence à ou dépend d'un autre module

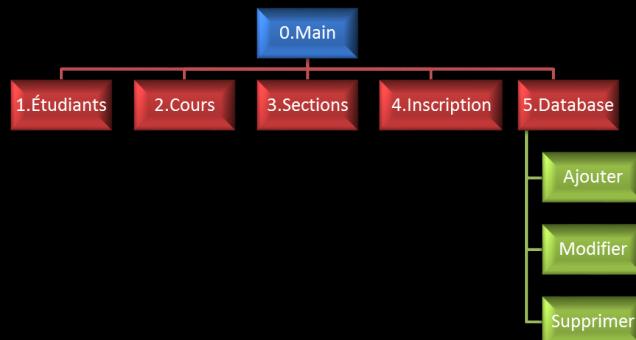
Ériger une hiérarchie

- Facilite la compréhension de la structure par niveau d'**abstraction**
- Facilite les **tests** unitaires
- Permet de bâtir un système partiel mais fonctionnel: de manière **incrémentale**



QUESTION

Identifier les modules et les relations entre eux



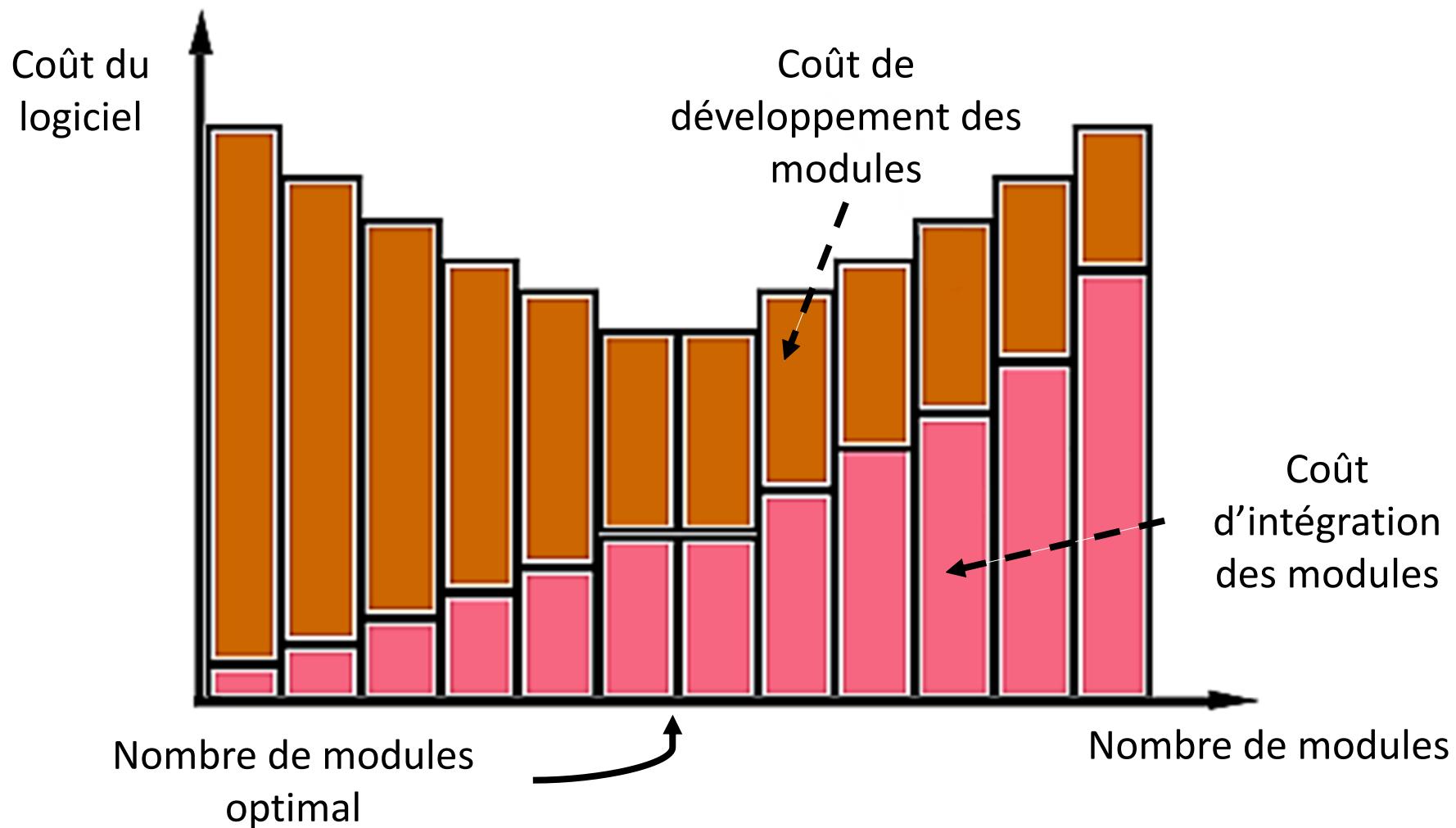
- Main contient les 5 modules
 - BD contient les 3 opérations
- Les modules 1 à 4 utilisent le module 5

Interface et implementation de modules

- Interface: **partie publique**
 - Ensemble des ressources (opérations, attributs, etc.) rendues accessibles aux modules clients
 - Conception d'un module M ne nécessite que les interfaces des autres modules que M pourra utiliser
- Implémentation : **partie privée**
 - Façon dont les ressources sont concrètement représentées et réalisées dans le module
- Séparation des préoccupations
 - Quoi offrir ? Analyse et conception
 - Comment le réaliser ? Conception et implémentation

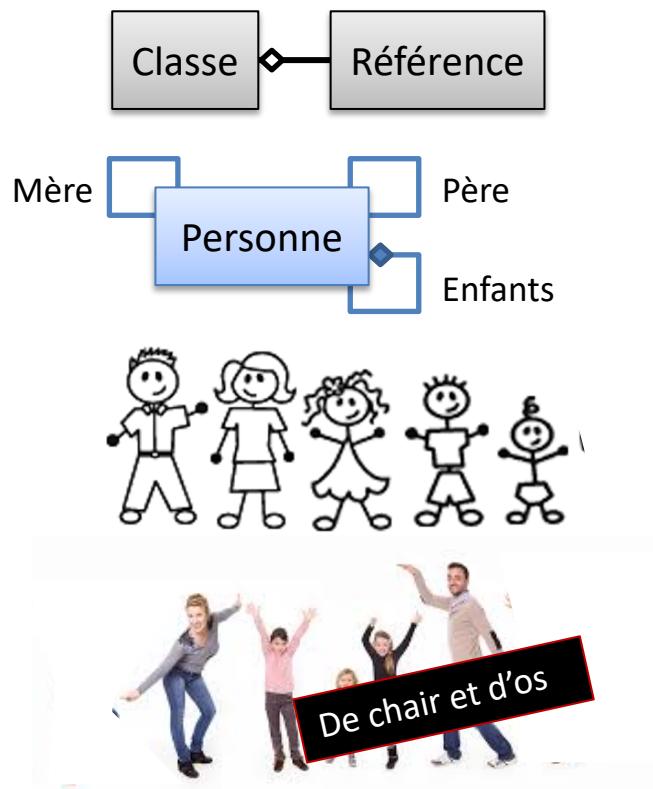
Compromis de la modularisation

Quel est le bon nombre de modules ?



Conception orientée- objet (OO)

Conception orientée-objet



Modèle linguistique

M3

MetaMetamodel
MOF

Modèle d'ingénierie

M2

Metamodel
UML

Modèle utilisateur

M1

Model
Classes & Objects

Monde réel

M0

Monde réel
Instances - Code

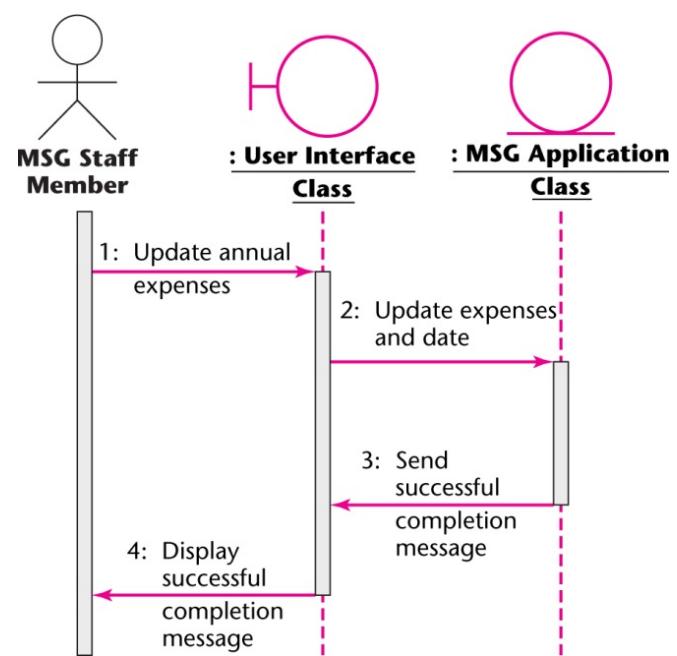
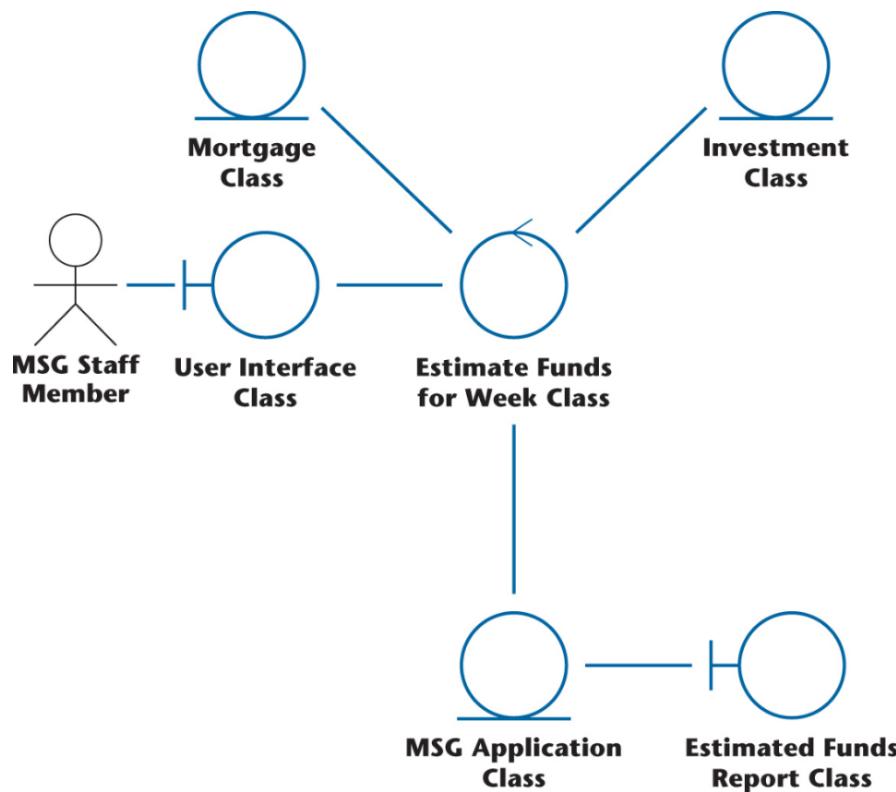
OMG's Model-Driven Architecture

Conception orientée-objet

- But: concevoir le logiciel en terme d'objets
 - Instances des classes extraites du modèle d'analyse
- Si utilise un langage qui n'est pas orienté-objet (ex: C, Ada)
 - Conception de type de donnée abstrait
- Si utilise un langage non typé (ex: FORTRAN, COBOL)
 - Conception par encapsulation des données

Communication entre les classes

- Déterminer les messages envoyés entre les objets
- Si objet A envoie message M à B, alors B doit avoir une méthode m() qui peut recevoir M



Collaborations entre classes

Construire un **diagramme de séquence UML**

1. Placer les objets interfaces et entités sur le diagramme
2. Pour chaque tâche représentée par un objet contrôle, identifier les **messages** qui doivent être échangés entre les objets pour réaliser la tâche
3. Faire correspondre la **signature d'une opération** à chaque message
4. Ajouter les flèches d'envois de messages correspondantes dans le diagramme de collaboration ou de séquence
5. Mettre à jour le diagramme de classes : ajouter les opérations identifiées

Diagramme de séquence UML

- Décrit les **interactions entre les objets**, instances du diagramme de classe
 - Montre la **trace** des messages échangés
- **Ordre** dans lequel les méthodes sont invoquées
 - Objet A invoque la méthode d'un autre objet B
 - Une méthode peut également en invoquer une autre
- **Un diagramme de séquence par CU**
 - Des fois plus si on veut séparer les scénarios alternatifs
- Doit être **cohérent avec le diagramme de classe**

Diagramme de séquence pour MSG

EstimerFondsDisponiblesPourSemaine

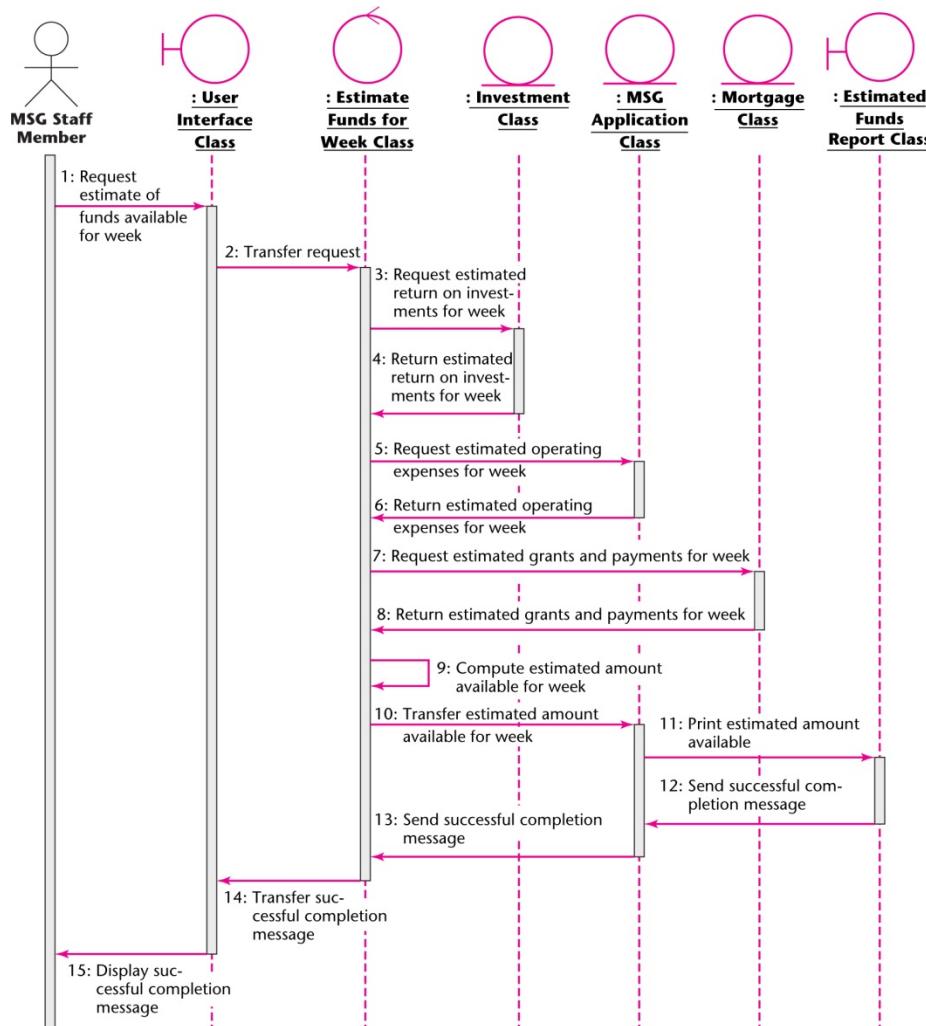


Diagramme de séquence pour MSG

GérerActif

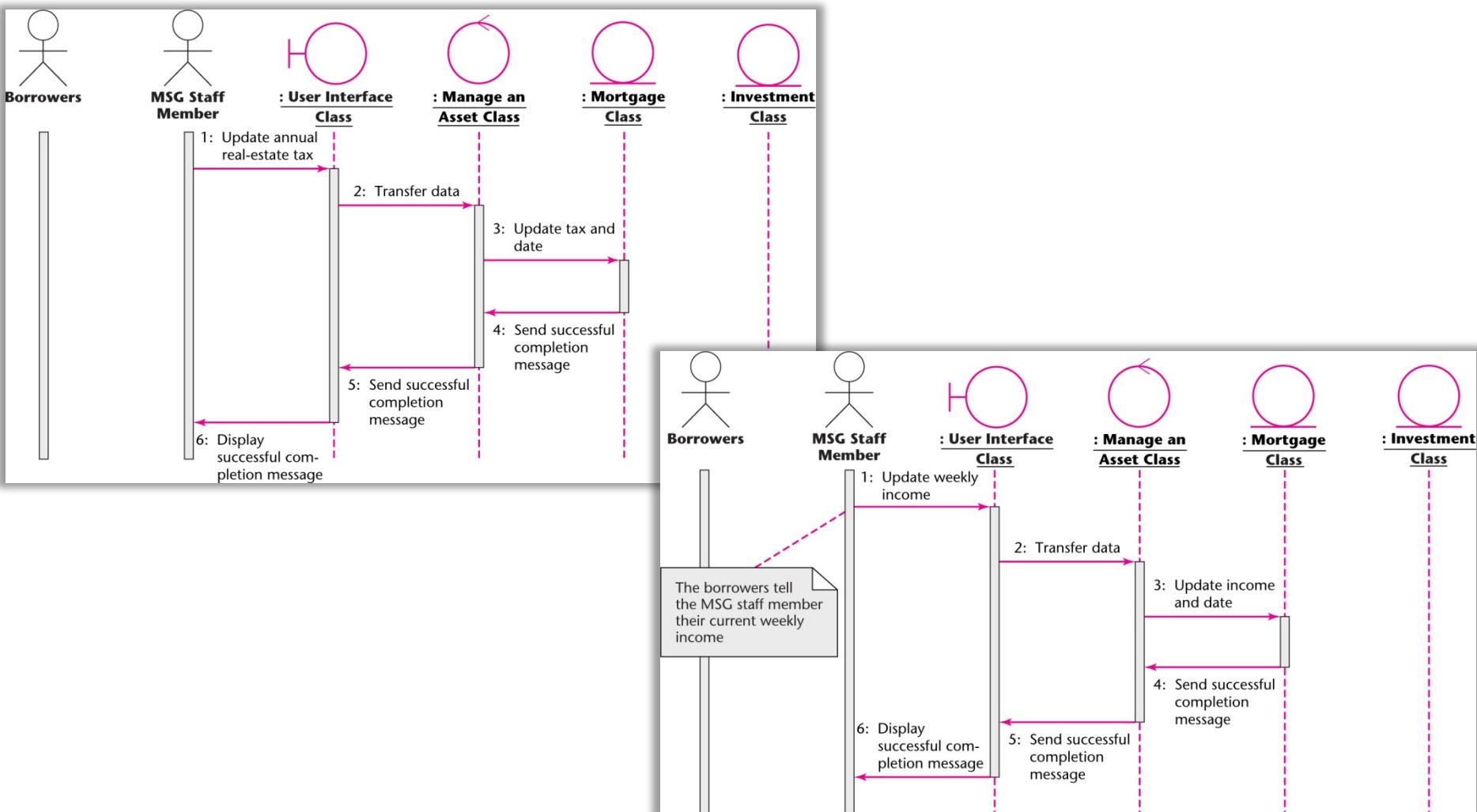


Diagramme de séquence pour MSG

MAJDépensesOpérationAnnuelles

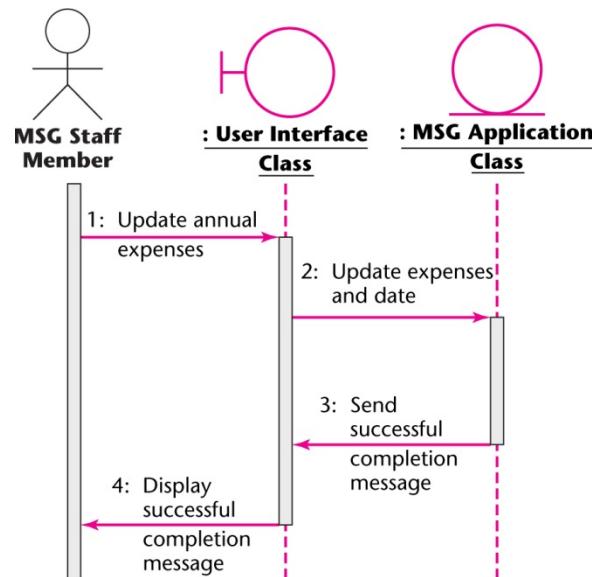
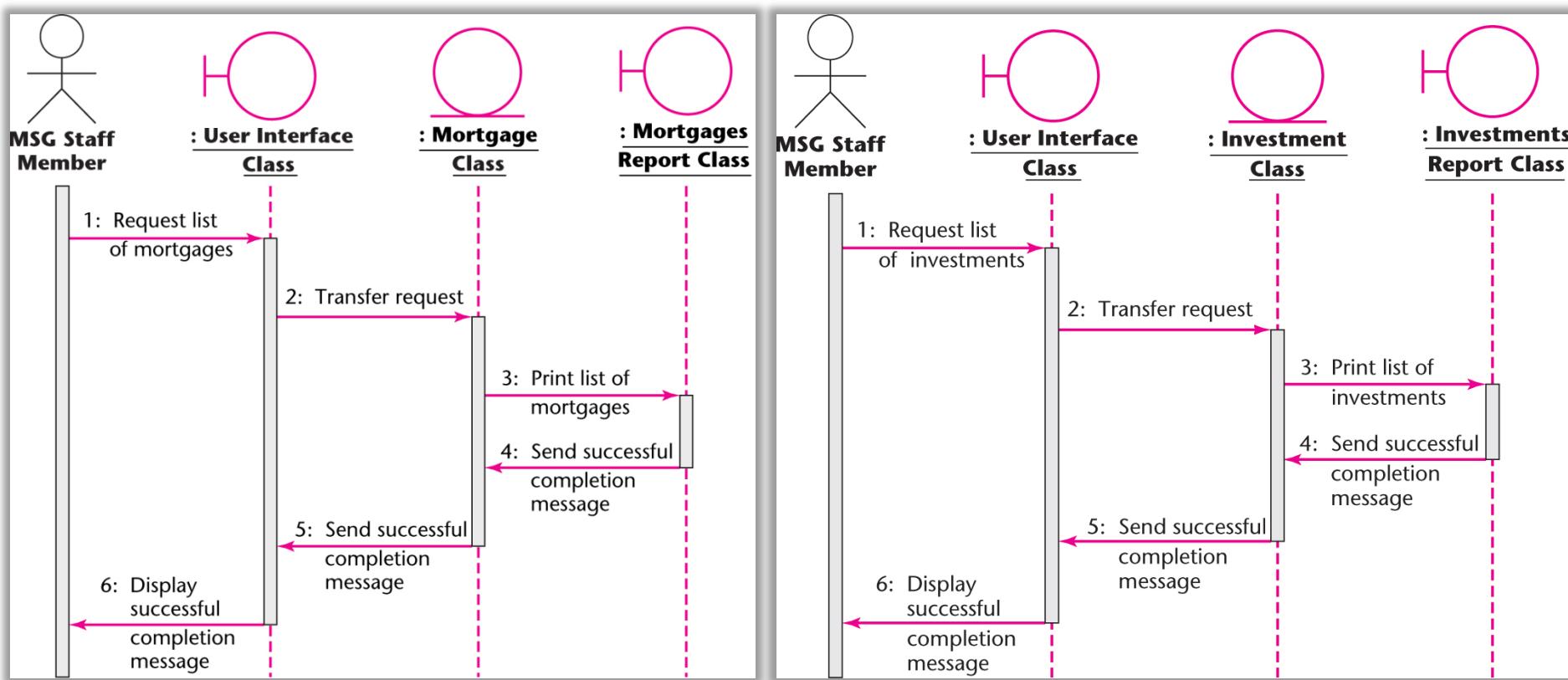


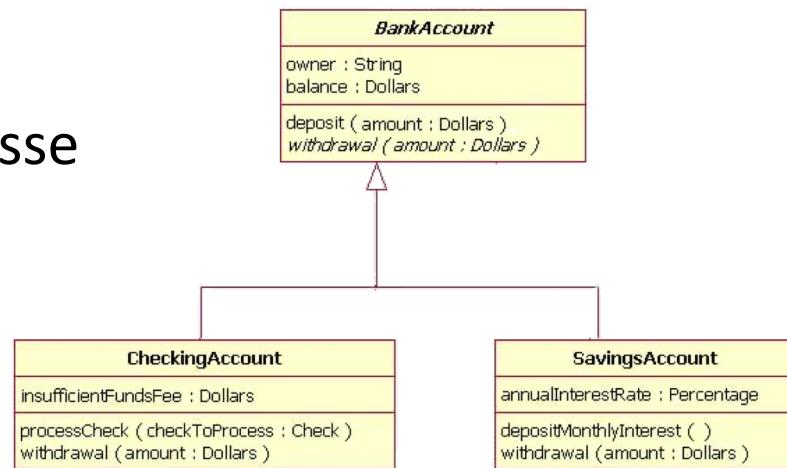
Diagramme de séquence pour MSG

ProduireRapport



Production du diagramme de classe

- Produire un **diagramme de classe** du design détaillé
- Plus de **classes** que dans le modèle d'analyse
 - N'ajouter que si nécessaire
- Déterminer les **attributs** de chaque classe
 - État des objets
 - Propriétés caractéristiques
 - Type et format des attributs
- Identifier les **méthodes** à assigner à chaque classe
 - Opérations publiques nécessaires pour réaliser les CU
 - Méthodes privées/protégées pour les aider



Production d'un diagramme de classes

Identifier les classes

- Classes du modèle d'analyse et du domaine
- Quels **objets** envoient ou reçoivent des **messages** ?
 - Invocateur et receveur
- Quelle **information** est nécessaire pour chaque opération ?
 - Paramètres, types de donnée abstraite
- Quelle information est **retournée** par chaque opération ?

Production d'un diagramme de classes

Identifier les méthodes

- Quelles **actions** le système doit-il effectuer ?
 - Au moins **une** méthode publique par CU
- Dans chaque CU, identifier le **sujet** de chaque action
 - Sujet est un **acteur**
 - **Humain**: requête envoyée au logiciel via clavier, souris, touché, ou autre appareil de saisie d'entrée
 - **Autre système**: requête envoyée en invoquant une méthode publique ou générée par un changement aux données surveillées

Production d'un diagramme de classes

Identifier les méthodes

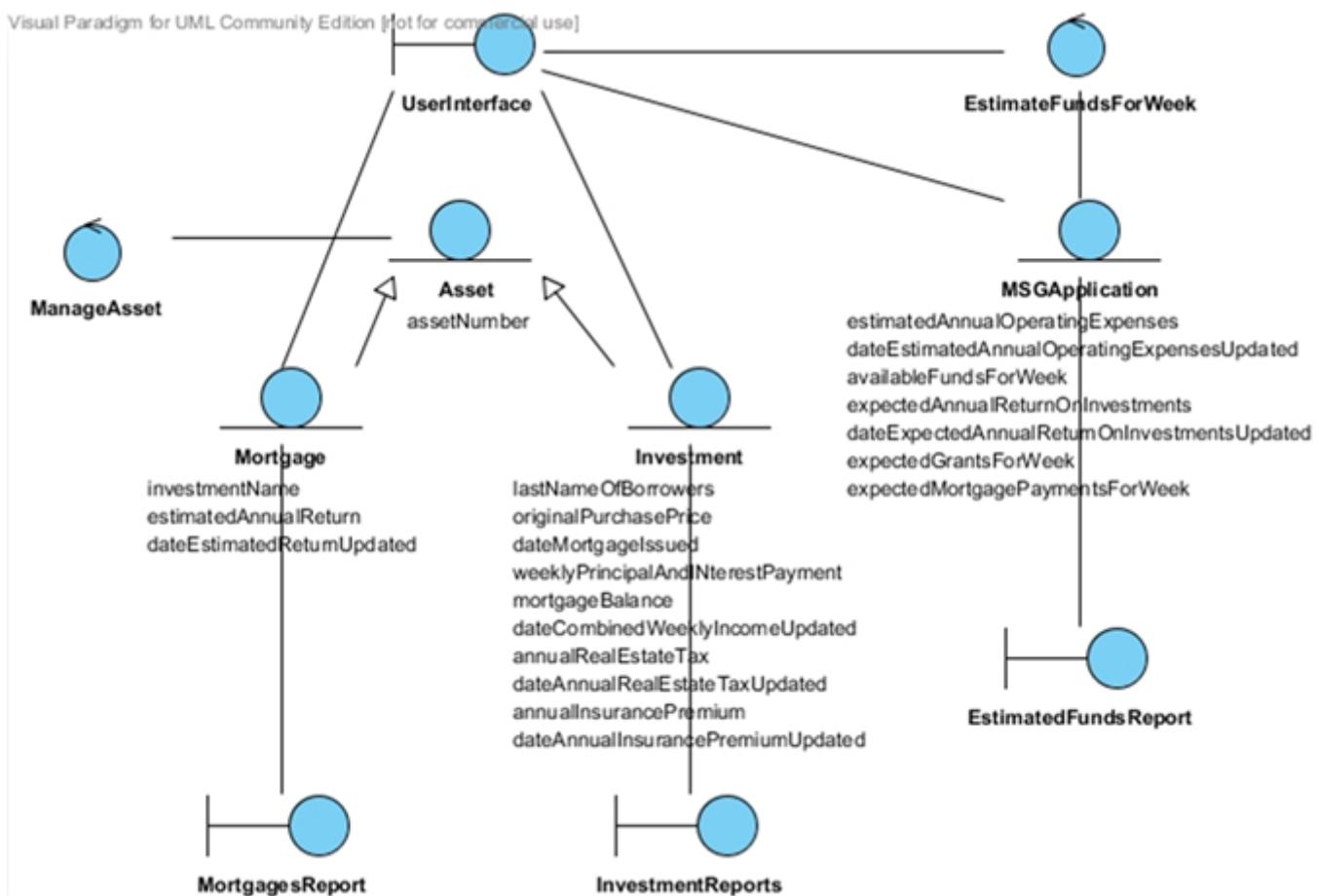
- Quelles **actions** le système doit-il effectuer ?
 - Au moins **une** méthode publique par CU
- Dans chaque CU, identifier le **sujet** de chaque action
 - Sujet est un **objet agissant sur un acteur**
 - Livre une réponse à un acteur
 - Affiche un menu, boîte de dialogue, info à l'écran ou retourne une valeur
 - Sujet est un **objet agissant sur un autre objet**
 - Appel de méthode, passage de message
 - Retour de valeur, d'objet ou transfère de contrôle

Production d'un diagramme de classes

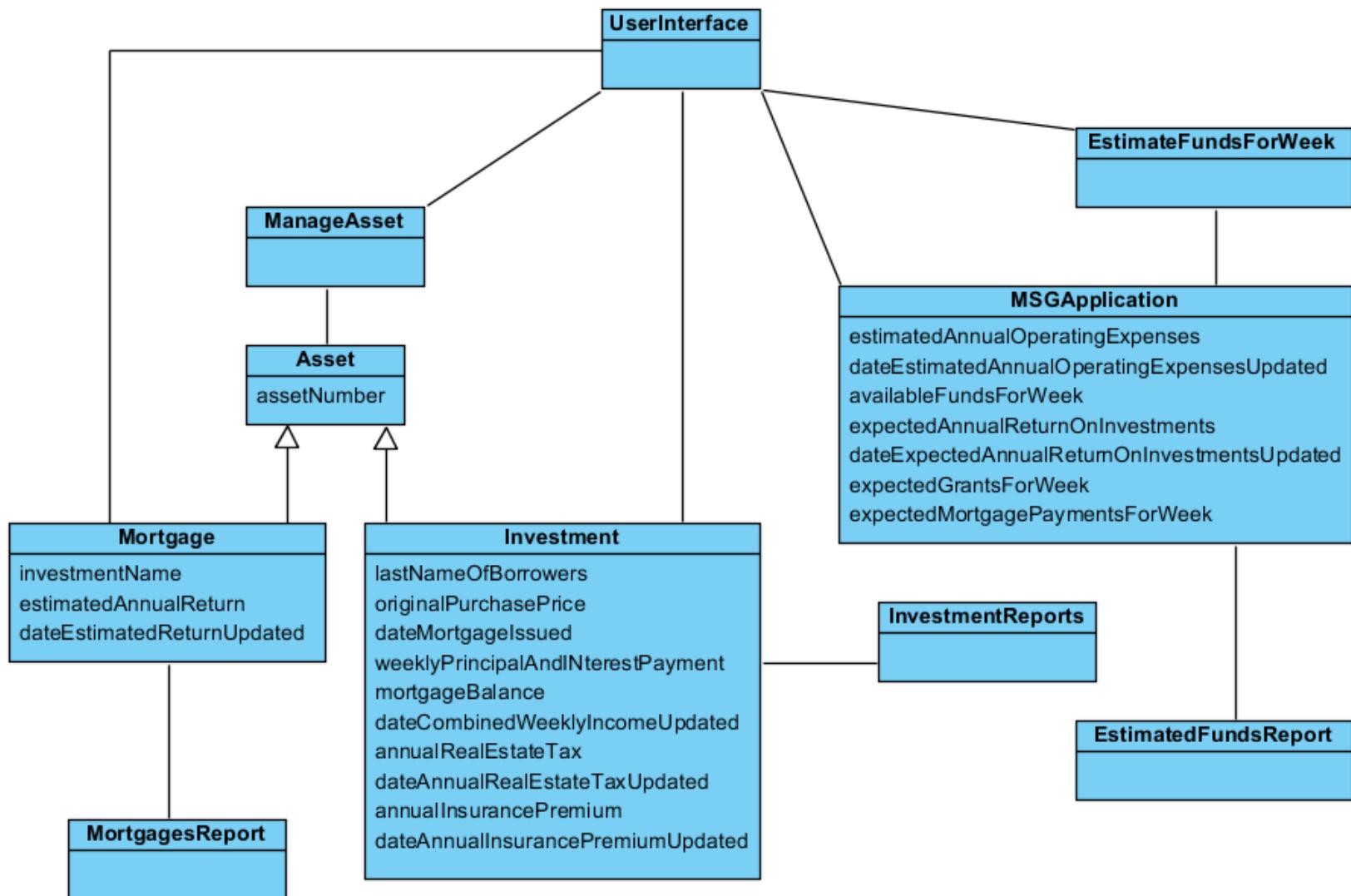
Identifier les attributs

- De quelle information un objet a-t-il besoin pour effectuer une action ?
 - Données pour créer l'objet
 - Données pour calculer ou traiter un message reçu
 - Données à passer en paramètre à une méthode d'un autre objet
 - Information caractéristique de l'état de l'objet
- Types d'attributs
 - Types **primitifs**: booléen, nombre, texte
 - Types **abstraits**: DateTime, classes du diagramme de classes (typiquement modélisé comme association)

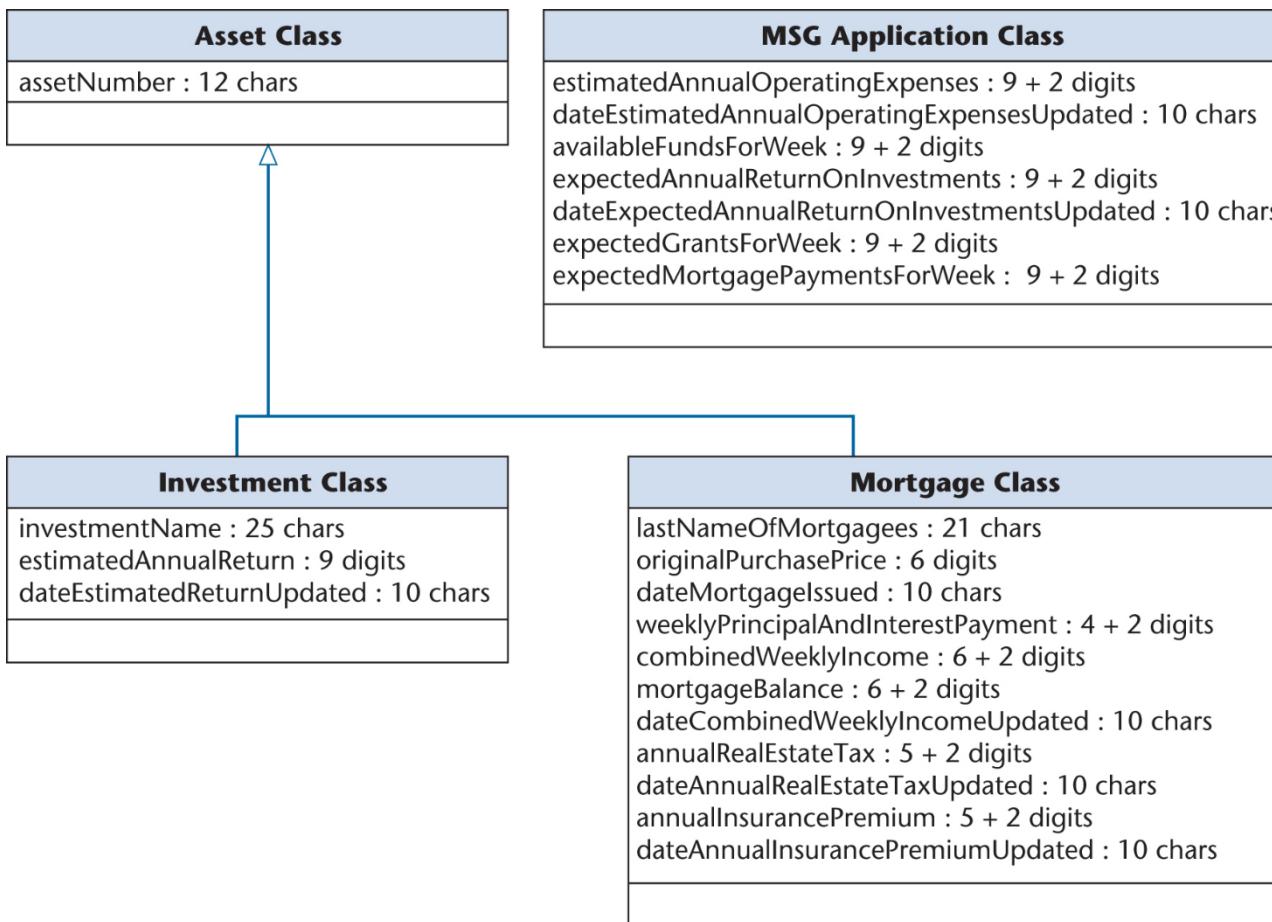
Utiliser le diagramme de classes participantes



Convertir en diagramme de classes UML



Détail des attributs



Opérations des classes pour MSG

«entity class» MSG Application Class
<ul style="list-style-type: none">- <<static>> estimatedAnnualOperatingExpenses : float- <<static>> estimatedFundsForWeek : float - <<static>> getAnnualOperatingExpenses () : float- <<static>> setAnnualOperatingExpenses (e : float) : void+ <<static>> getEstimatedFundsForWeek () : float+ <<static>> setEstimatedFundsForWeek (e : float) : void+ <<static>> initializeApplication () : void+ <<static>> updateAnnualOperatingExpenses () : void+ <<static>> main ()

- main(): point d'entrée
- Mise en place et initialisation de l'application
- Getter et setter pour les attributs privés
- Pourquoi statique ?

Opérations des classes pour MSG

«boundary class»
User Interface Class
<pre>+ <<static>> clearScreen () : void + <<static>> pressEnter () : void + <<static>> displayMainMenu () : void + <<static>> displayInvestmentMenu () : void + <<static>> displayMortgageMenu () : void + <<static>> displayReportMenu () : void + <<static>> getChar () : char + <<static>> getString () : string + <<static>> getInt () : int</pre>

- Afficher les menus et sous-menus
- Recevoir et analyser (*parse*) les entrées

Opérations des classes pour MSG

«entity class»
Asset Class
assetNumber : string
+ getAssetNumber () : string
+ setAssetNumber (a : string) : void
+ abstract read (fileName : RandomAccessFile) : void
+ abstract obtainNewData () : void
+ abstract performDeletion () : void
+ abstract write (fileName : RandomAccessFile) : void
+ abstract save () : void
+ abstract print () : void
+ abstract find (s : string) : Boolean
+ delete () : void
+ add () : void

- Classe abstraite
- Attributs et méthodes communs remontés
- Implémentation des méthodes abstraites déléguée aux sous-classes
- Actions génériques

Opérations des classes pour MSG

«entity class» Investment Class	
<ul style="list-style-type: none"> - investmentName : string - expectedAnnualReturn : float - expectedAnnualReturnUpdated : string + getInvestmentName () : string + setInvestmentName (n : string) : void + getExpectedAnnualReturn () : float + setExpectedAnnualReturn (r : float) : void + getExpectedAnnualReturnUpdated () : string + setExpectedAnnualReturnUpdated (d : string) : void + totalWeeklyReturnOnInvestment () : float + find (findInvestmentID : string) : Boolean + read (fileName : RandomAccessFile) : void + write (fileName : RandomAccessFile) : void + save () : void + print () : void + printAll () : void + obtainNewData () : void + performDeletion () : void + readInvestmentData () : void + updateInvestmentName () : void + updateExpectedReturn () : void 	

«entity class» Mortgage Class
<ul style="list-style-type: none"> - mortgageeName : string - price : float - dateMortgagelssued : string - currentWeeklyIncome : float - weeklyIncomeUpdated : string - annualPropertyTax : float - annualInsurancePremium : float - mortgageBalance : float + <<static final>> INTEREST_RATE : float + <<static final>> MAX_PER_OF_INCOME : float + <<static final>> NUMBER_OF_MORTGAGE_PAYMENTS : int + <<static final>> WEEKS_IN_YEAR : float + getMortgageeName () : string + setMortgageeName (n : string) : void + getPrice () : float + setPrice (p : float) : void + getDateMortgagelssued () : string + setDateMortgagelssued (w : string) : void + getCurrentWeeklyIncome () : float + setCurrentWeeklyIncome (i : float) : void + getWeeklyIncomeUpdated () : string + setWeeklyIncomeUpdated (w : string) : void + getAnnualPropertyTax () : float + setAnnualPropertyTax (t : float) : void + getAnnualInsurancePremium () : float + setAnnualInsurancePremium (p : float) : void + getMortgageBalance () : float + setMortgageBalance (m : float) : void + totalWeeklyNetPayments () : float + find (findMortgagelID : string) : Boolean + read (fileName : RandomAccessFile) : void + write (fileName : RandomAccessFile) : void + obtainNewData () : void + performDeletion () : void + print () : void + <<static>> printAll () : void + save () : void + readMortgageData () : void + updateBalance () : void + updateDate () : void + updateInsurancePremium () : void + updateMortgageeName () : void + updatePrice () : void + updatePropertyTax () : void + updateWeeklyIncome () : void

Opérations des classes pour MSG

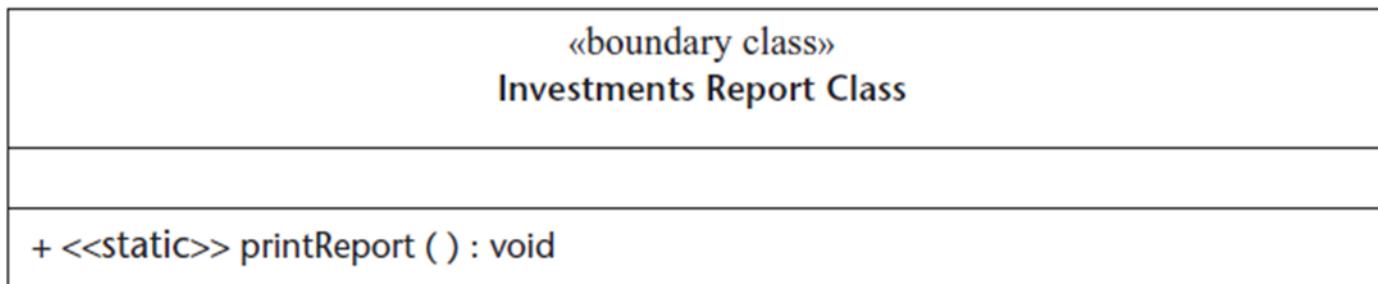
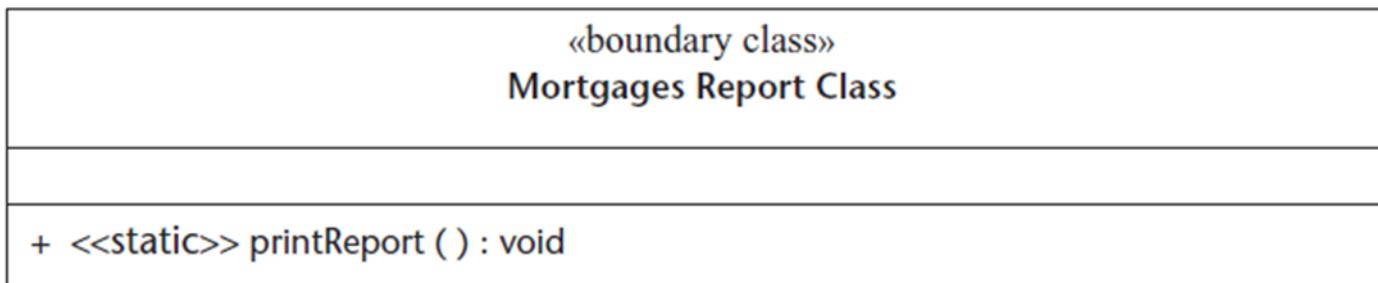
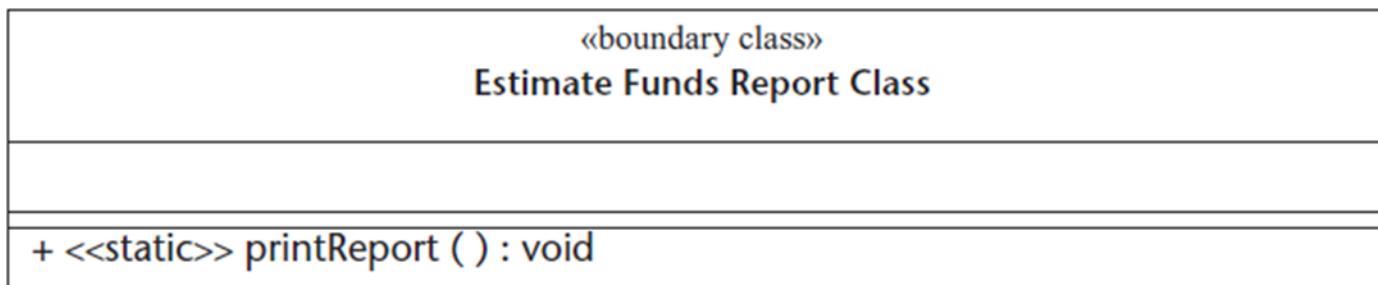
«control class»
Manage an Asset Class

+ <<static>> manageInvestment () : void
+ <<static>> manageMortgage () : void

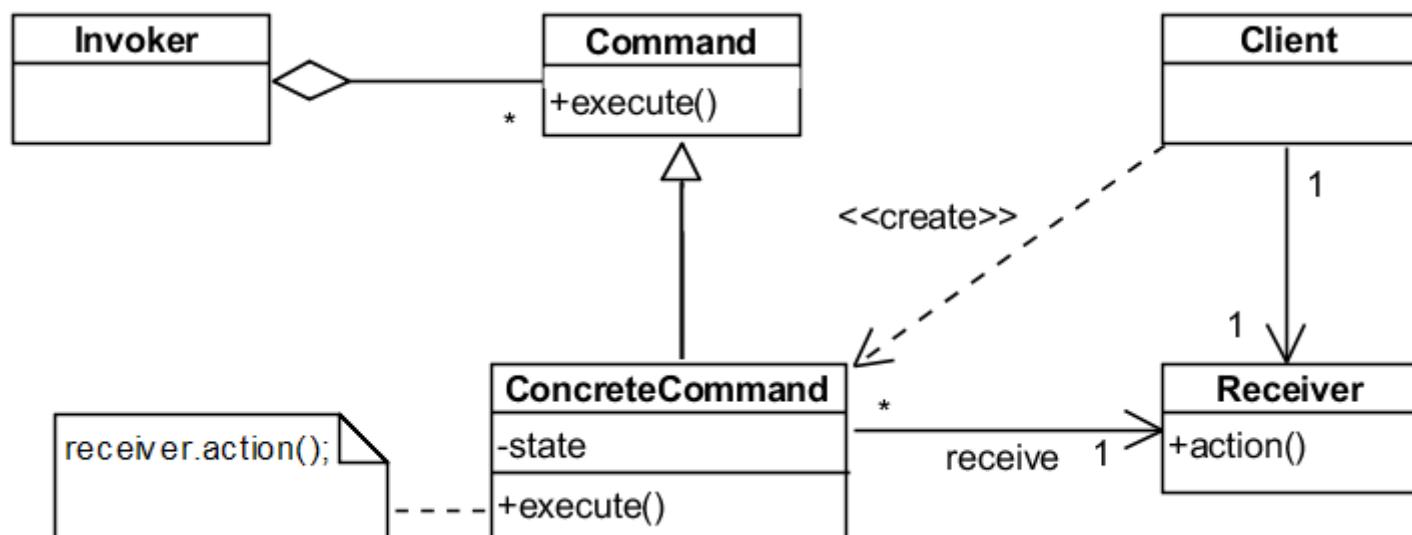
«control class»
Estimate Funds for Week Class

+ <<static>> compute () : void

Opérations des classes pour MSG



Bon exemple de diagramme de classe UML



Bon exemple de diagramme de séquence UML

