SORBONNE UNIVERSITE– FACULTE DES SCIENCES- MASTER INFORMATIQUE – MU4IN502

Ingénierie du Logiciel

Fiches de cours

Yann Thierry-Mieg

2020/2021

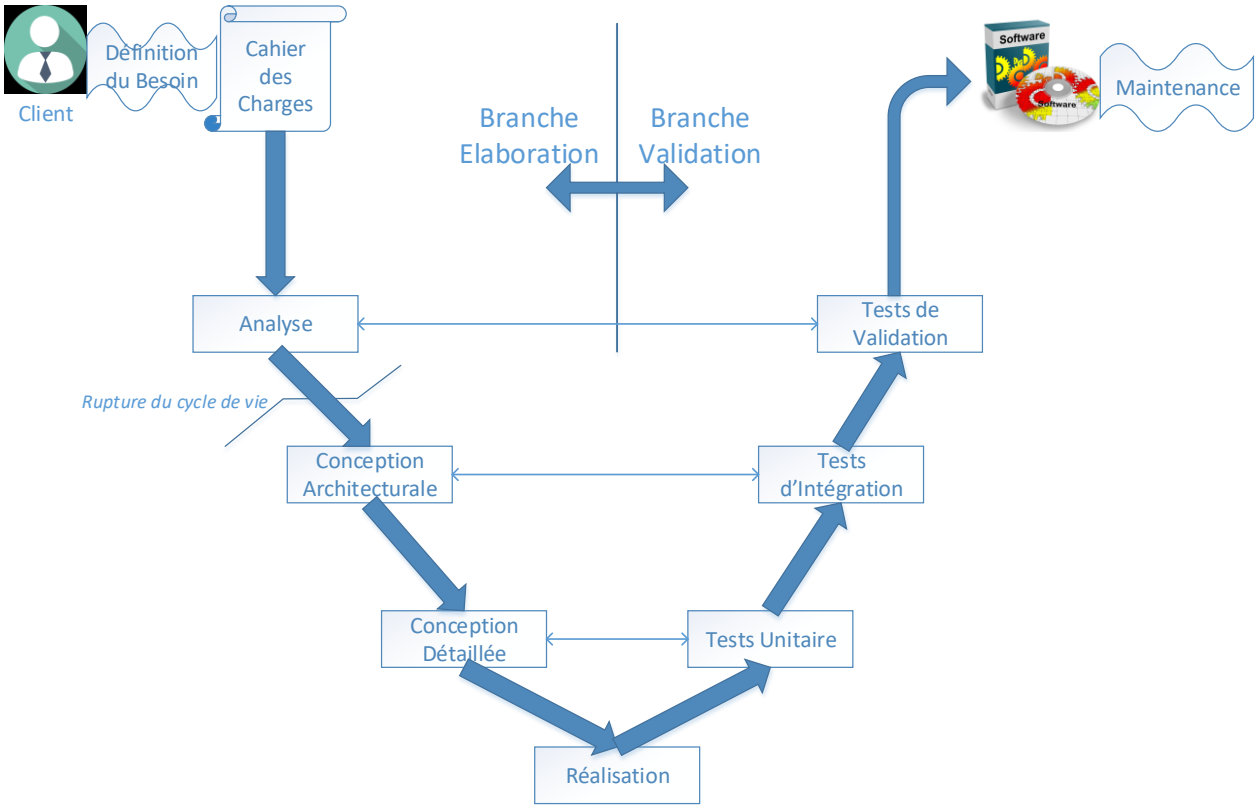
Supports en ligne : https://pages.lip6.fr/Yann.Thierry-Mieg/IL/

Fiches de Cours : Les points clé
M1 Informatique – Ingénierie du logiciel
Sorbonne Université

Le cycle en V

Le cycle en V est utilisé comme support pour discuter les différentes activités du développement dans la majeure partie de l'UE. Cette partie récapitule les principaux points à retenir sur ce cycle, sans les détailler.

Une méthode de développement découpée en étapes et en tâches bien définies. A chaque étape de construction correspond une étape de validation, permettant d'assurer que les objectifs sont atteints. Attention à l'ordre dans lequel on réalise les activités de la branche validation : on distingue l'activité de définition des tests, qui se fait en même temps que l'étape correspondante d'élaboration, de l'activité d'exécution des tests, où l'on confronte le résultat du développement aux tests. La définition des tests précède toujours la réalisation correspondante.



Analyse

Finalité : Passer de la description informelle et lacunaire exprimée dans le domaine métier du client que constitue le cahier des charges à une spécification *technique* du besoin, adaptée au travail de conception de l'équipe de développement.

Scope : Le système à développer est une boite noire, dont on spécifie seulement les entrées sorties et comportements observables. Les acteurs sont les utilisateurs finaux du système, ou les systèmes logiciels et matériels avec lesquels le système interagit.

Activités : Spécifier les utilisateurs du système et le rôle que joue le système pour eux. Spécifier les scenarios d'interaction avec le système (ou *workflow*). Spécifier les données manipulées par le système et les relations entre ces données (leur *structure*), ainsi que la façon de les mettre à jour au fil des interactions (leur *dynamique)*.

Artefacts : Définition des acteurs en les liant au domaine métier. Diagramme(s) de cas d'utilisation récapitulant les missions du système vis-à-vis de ses acteurs. Diagramme de classe métier pour fixer la structure des données. Fiches détaillées de cas d'utilisation précisant les *workflow* et la *dynamique*. Diagrammes de séquence de niveau analyse pour préciser la signature des échanges de données.

Tests de Validation

Finalité : S'assurer que l'ensemble du besoin utilisateur est correctement capturé. Mettre en place les mécanismes permettant de s'assurer que le produit final répondra à ce besoin. En phase de recette, confronter le produit final aux tests de validation.

Scope : Homogène à la phase d'analyse ; le système est une boite noire.

Activités : Elaborer le cahier de tests de validation. Le faire approuver par le client. Passer la procédure de recette.

Artefacts : Appuyé par les fiches détaillées et les diagrammes de séquence d'analyse, construction d'un jeu de tests de validation couvrant le besoin.

Conception Architecturale

Finalité : Proposer une solution satisfaisant le besoin identifié en analyse. Définir une architecture pour le système. Découpe *fonctionnelle* et *structurelle* du système en composants. Spécifier les interactions entre ces composants.

Scope : Les composants sont des boites noires, dont on spécifie principalement les entrées sorties. Pour un composant donné, les acteurs sont les autres composants. Les acteurs physiques sont assimilés à leurs interfaces homme machine et matérialisés par des composants. Le système global est constitué d'un ensemble d'instances de composants en interaction.

Activités : Découpe structurelle des données de l'application (diagramme de classe métier) pour en affecter la responsabilité aux composants. Découpe fonctionnelle des missions du système (opérations des diagrammes de séquence d'analyse) sur les composants. Raffinement des séquences d'analyse vis-à-vis de la découpe en composants. Identification des configurations et instanciations de composant pertinentes.

Artefacts : Diagrammes de composant spécifiant leurs interfaces requises et offertes (signatures). Diagrammes de structure interne spécifiant les configurations nominales pour l'instanciation. Diagrammes de séquence spécifiant les interactions entre composants (une ligne de vie par instance de composant).

Tests d'Intégration

Finalité : Permettre le développement en parallèle des composants. Spécifier et fixer la signature des opérations d'interface, ainsi que les contrats ou séquences d'invocations légitimes. Permettre de valider en isolation que le fonctionnement de chaque composant répond aux besoins.

Scope : Homogène à l'analyse architecturale, chacun des composants est une boite noire.

Activités : Définition des séquences de test pertinentes pour un composant. Définition de composants testeurs et bouchon. Définition des configurations adaptées pour exécuter les tests de chaque composant.

Artefacts : Diagrammes de composant pour les bouchons et testeurs. Diagrammes de structure interne pour les configurations de tests. Diagrammes de séquence ou scripts décrivant les tests d'intégration.

Conception Détaillée

Finalité : Proposer une ligne d'implémentation pour chacun des composants. Choisir les structures de données adaptées au besoin, découper les responsabilités de chaque composant sur des classes, organiser les classes entre elles (utilisation de *design pattern*).

Scope : On considère chaque composant séparément et on spécifie la structure des classes qui le réalisent. Le comportement externe du composant est déjà capturé par les tests d'intégration.

Activités : Définition de classes *façade* réalisant un point d'entrée pour chaque composant. Elaboration de classes de réalisation adaptées à l'implantation de la solution, définition et raffinement des signatures et interactions au sein du composant, ébauches algorithmiques des méthodes les plus complexes.

Artefacts : Diagrammes de classe pour chaque composant, typiquement annotés (pseudo-code). Au besoin, diagrammes de séquence mettant en jeu des instances des classes du composant.

Test Unitaire

Finalité : Couvrir et tester l'ensemble des méthodes de chacune des classes identifiées.

Scope : On considère chaque classe au sein du composant individuellement.

Activités : Définition des tests unitaires exécutables.

Artefacts : Tests exécutables couvrant les classes de réalisation.

Réalisation

Finalité : Implanter les classes identifiées en conception détaillée dans un langage cible.

Scope : Niveau le plus fin ; codage des classes et opérations.

Activités : Codage.

Artefacts : Code source, binaires exécutables.

La phase d'analyse

La phase d'analyse a pour objectif de répondre à la question : **QUOI ?**

La phase d'analyse sert à modéliser (spécifier) la **compréhension des besoins client**.

Cette phase sert aussi à bien définir les **contours de l'application** (scope).

On oppose en analyse **les acteurs** au **système.**

Grâce à la phase d'analyse, on sait alors ce qui doit être intégré dans la solution logicielle mais aussi ce qui ne doit pas être intégré.

Dans cette phase, on utilisera des diagrammes de cas d'utilisation (section I) pour lesquels on rédigera des fiches détaillées de cas d'utilisation (section II). On construira aussi des diagrammes de classe « métier » (section III), et des diagrammes de séquence opposant le système aux acteurs (section IV). Enfin, on écrira dans l'étape de validation les tests de validation (section V).

I. Diagramme de Use Case

Acteur :

Définition Entité **externe** au système, qui **interagit** avec lui. L'acteur joue un rôle particulier vis-à-vis du système. Une même entité physique peut être représentée par plusieurs acteurs et réciproquement.

Un acteur est symbolisé par une figurine de bonhomme (stickman), parfois placé dans un rectangle quand on souhaite appuyer l'idée que cet acteur est un intervenant logiciel ou matériel plutôt qu'un humain.

L'acteur est lié à un use case par un arc plein, sans flèches aux extrémités, qui représente une **communication** entre l'acteur et le use case. En analyse on s'intéresse particulièrement à la nature des données échangées au cours de ces interactions.

Quand l'acteur n'a pas l'initiative de l'interaction avec le système (i.e. le système utilise l'acteur), l'interaction (l'arc qui lie l'acteur au use case) entre l'acteur et le use case est qualifiée par le stéréotype <<secondary>>.

> *NB : Les acteurs doivent interagir **directement** avec le système. Par exemple, le client d'une agence de voyage en ligne est acteur (il interagit via le site web avec le système), le client d'une agence traditionnelle ne l'est pas (c'est l'agent de voyage qui interagit avec le système, pour le compte du client).*

Les acteurs peuvent être organisés en utilisant la relation d'héritage, symbolisé par une flèche à large tête triangulaire. Si un acteur A dérive de B, cela signifie que tous les use case offerts à B sont aussi offerts à A. Ceci permet des diagrammes plus compacts.

> *Attention cependant à respecter les rôles définis par le cahier des charges : même si l'acteur « directeur » peut faire toutes les actions que l'acteur « stagiaire » peut faire, il est incorrect de placer un héritage entre les deux (car on ne peut pas dire que le directeur « est un » stagiaire).*

Globalement, les acteurs représentent des rôles métier ou des systèmes logiciels existants, il est souhaitable de les nommer en utilisant les termes du client. Dans le document d'analyse, on trouvera une description métier précise des acteurs qui sont définis.

(Sous-)Système :

Définition Ce cadre rectangulaire qui entoure les cas d'utilisation permet de spécifier quelle partie du système offre le support pour réaliser les cas d'utilisation. Il matérialise aussi la frontière entre ce que le système offre et l'extérieur (qui contient les acteurs).

Cas d'utilisation :

Définition Un cas d'utilisation ou use case représente **une action de bout en bout** qui **apporte quelque chose** aux acteurs. Un cas d'utilisation se décrit comme **un ensemble d'interactions** entre un acteur et le système, déclenché par l'acteur. On utilise indifféremment dans ce support le terme anglais « use case » et « cas d'utilisation ».

Un use case a un début, déclenché (on parle de *trigger*) par l'acteur. Ensuite une interaction plus ou moins longue entre système et acteur(s) s'ensuit. A la fin d'un use case, l'acteur devrait pouvoir quitter son poste, il a accompli ce qu'il cherchait à obtenir.

Un use case regroupe plusieurs scénarios ou séquences d'interaction possibles. En particulier on distingue le scénario dit « nominal » (où tout se passe pour le mieux) des alternatives (où des choix moins usuels sont employés) et des exceptions (où la tache n'arrive pas à être menée à bien). Ces scénarios seront obligatoirement spécifiés séparément, à l'aide de fiches détaillées de cas d'utilisation, décrites plus bas.

Les use case sont matérialisés par des bulles contenant un court texte, qui doit de préférence être formulé comme une action (e.g. « retirer de l'argent » plutôt que « retrait d'argent ») énoncée du point de vue de l'acteur (e.g. pas « délivrer de l'argent »).

L'objectif des use case est de recenser les services qu'offre le système à ses acteurs. La granularité ne doit pas être trop fine : on n'est pas au niveau d'une action atomique (jouer un coup) mais au niveau d'une action complète (jouer une partie).

Dans tous les cas, il est essentiel d'avoir une **granularité homogène** dans la description.

> *NB : On trouvera dans la littérature des diagrammes plus détaillés, pour « zoomer » sur une fonctionnalité, souvent à grand renfort de <<extend>> et <<include>>. Cependant, dans l'ue, les diagrammes demandés en phase d'analyse ne doivent pas se noyer dans les détails.*

Liens entre use case

A -- <<include>> -> B : Au cours du déroulement de **A**, une étape **obligatoire** est de réaliser les interactions décrites dans le use case **B**.

L'association est matérialisée par une flèche pointillée et munie du stéréotype <<include>>, et se lit « A inclut B ».

En terme de fiche détaillée, on doit donc trouver dans la description de **A** une étape qui spécifie : « Etape 4 : réaliser le cas d'utilisation **B** ».

Attention, on spécifie ici que pour accomplir **A** à un certain moment **on doit faire B**. Il faut ici se poser la question de fusionner B dans A, et de ne conserver que le use case A. On ne matérialisera B que si c'est aussi une étape d'autres use case (i.e. B est utilisé en plusieurs endroits).

A -- <<extend>> -> B : Au cours du déroulement de **B**, **parfois (de façon optionnelle),** une étape est de réaliser les interactions décrites dans le use case **A**.

L'association est matérialisée par une flèche pointillée et munie du stéréotype <<extend>> et se lit « A étend B ».

En terme de fiche détaillée, on doit donc trouver dans la description de B une étape qui spécifie : « si l'utilisateur choisit cette option, alors réaliser le cas d'utilisation **A** ».

> *NB : Au sens UML strict, le use case B doit déclarer un « point d'extension », c'est-à-dire l'étape à laquelle on peut basculer pour aller réaliser le comportement décrit dans A. L'association <<extend>> déclare alors quel point d'extension elle étend. Dans l'ue, on n'utilisera pas ce mécanisme, redondant vis-à-vis de la spécification des fiches détaillées de use case. L'outil RSA créera des points d'extension pour vous en TME.*

Attention au sens des flèches, A vient compléter le comportement décrit dans B, il l'étend, mais B est le use case principal. Alors que pour le cas de l'<<include>>, si A inclut B, c'est A le use case principal.

De nouveau, si l'on sépare ainsi la description des deux use case, c'est soit parce que l'un d'eux est réutilisé ailleurs, soit parce que l'action est très complexe et mérite d'être détaillée pour préserver la granularité globale du diagramme.

A ---|> B : A dérive ou hérite de B, A « est un » B spécialisé. Dans certains cas, on peut vouloir exprimer un raffinement d'un comportement existant.

L'association est matérialisée par une flèche pleine avec une grosse tête triangulaire et se lit « A dérive de B », ou « A hérite de B ».

En terme de fiche détaillée, A peut redéfinir certaines étapes de l'interaction décrite dans B, voire même en ajouter. A ne doit pas remettre en cause le comportement décrit par B, mais plutôt le spécialiser pour traiter des cas particuliers. Ce mécanisme est souvent utilisé en combinaison avec des cas

d'utilisation abstraits (porteurs du stéréotype <<abstract>>), dans lesquels une ou plusieurs étapes ne sont pas pleinement spécifiées. Dans ce cas, le cas d'utilisation dérivé va définir les étapes manquantes.

Cette relation d'héritage est assez peu intuitive, et on l'évitera dans l'ue, de même que l'utilisation des cas d'utilisation abstraits.

> ***ATTENTION :*** *L'objectif des diagrammes de use case est de* ***recenser et organiser*** *les fonctionnalités qu'offre le système****. Il n'y a pas de lien entre use case qui permette d'exprimer « on fait A, puis on fait B ».*** *Les pré-conditions dans les fiches détaillées permettent cependant de spécifier qu'une action doit avoir eu lieu pour déclencher un use case. Il ne faut donc pas chercher à établir une relation d'ordre entre use case sur les diagrammes.*

Le cas particulier des traitements périodiques

Il arrive fréquemment qu'un système doive régulièrement effectuer une action de façon automatisée, comme vider des logs, faire une maintenance… Cette action est bien un objectif que doit remplir le système, donc un cas d'utilisation, mais l'on ne dispose pas d'acteur pour déclencher le trigger, et un cas d'utilisation doit toujours être lié à un acteur.

On introduit alors de façon artificielle un acteur supplémentaire, nommé « Time », « Cron », « Daily schedule », « traitements par lots », « déclenchement périodique » ou une variante sur ce thème. Cet acteur représente le fait qu'à intervalle régulier l'action de maintenance sera déclenchée. Le scénario d'interaction de la fiche détaillée démarre alors typiquement par « 1. Le 1er janvier de chaque année, l'acteur *traitements par lots* déclenche …. ».

Exemple

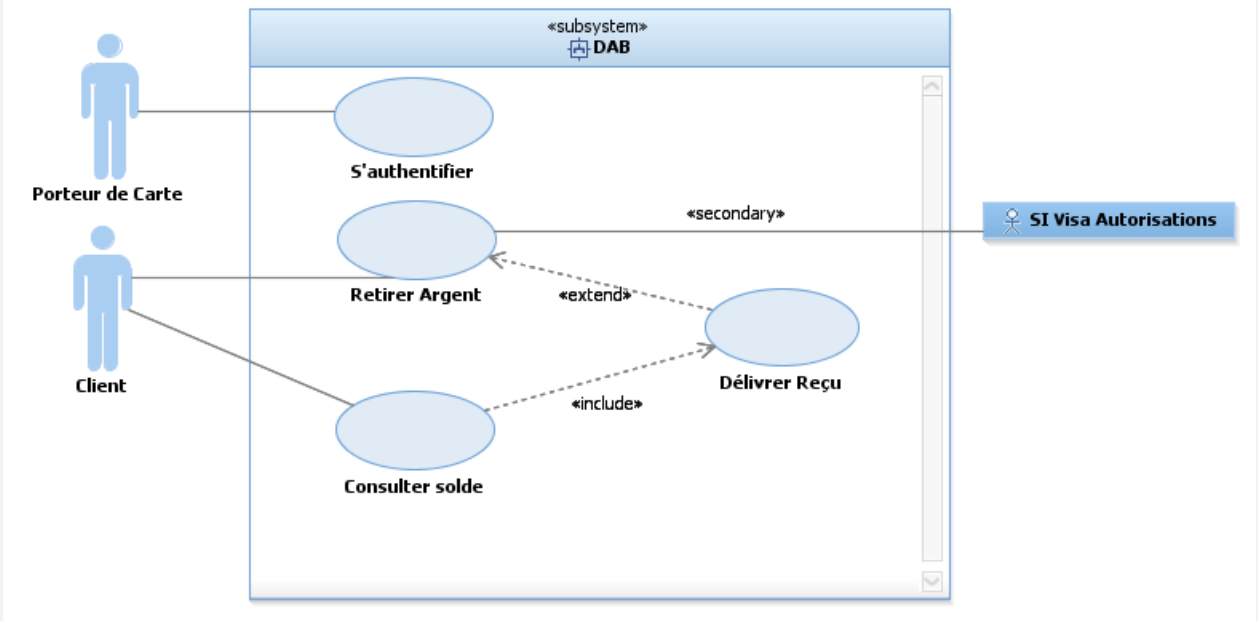


Figure 1 : Un exemple simple de diagramme de cas d'utilisation.

II. Fiches Détaillées de cas d'utilisation

On rédige pour chaque cas d'utilisation une fiche détaillée qui spécifie le déroulement des interactions. On utilise le langage naturel, mais on se donne un cadre qui aide à avoir une présentation homogène des cas d'utilisation, et à ne pas oublier des rubriques.

Rubriques d'une fiche détaillée

Selon l'entreprise, ce cadre varie dans sa présentation et les rubriques attendues. En général, on trouve au moins les rubriques suivantes :

Nom du cas d'utilisation : un titre pour ce cas d'utilisation, cohérent avec le nom sur le diagramme de cas d'utilisation. On peut également numéroter les cas d'utilisation pour faciliter les références croisées.

Date/Version/Auteurs : ces champs sont principalement utiles dans un contexte collaboratif.

Description : En une ou deux lignes, une description informelle du cas d'utilisation. On se concentre sur ce que cela apporte à l'acteur.

Acteurs : On mentionne les acteurs (principaux et secondaires) qui interviennent dans le déroulement du cas d'utilisation.

Pré-conditions : Les pré-conditions doivent être remplies pour pouvoir débuter le cas d'utilisation. Si une pré-condition est violée le cas d'utilisation ne peut pas débuter (il faut imaginer que cette fonctionnalité est grisée dans l'interface). Donc une pré-condition ne se vérifie pas au cours du scénario d'interaction, au moment du trigger (étape 1 du scénario), elle doit déjà être vérifiée.

Scénario Nominal : Le scénario nominal est la séquence d'interaction dans les cas où tout se passe « bien ». Ce scénario est découpé en étapes, que l'on numérote pour faciliter les références croisées. Chaque étape doit commencer par « Le système fait… » ou « L'acteur fait… ». Dans une étape, on ne doit pas mélanger les actions du système et de l'acteur. La première étape du scénario (**trigger)** doit être à l'initiative de l'acteur.

Post-conditions : A l'issue du scénario nominal ou d'un scénario alternatif, les post-conditions doivent être vérifiées. Les post-conditions expriment le plus souvent le fait que l'objectif du use case a été atteint. Un scénario exceptionnel peut mener à une violation de ces post-conditions.

Alternatives/Exceptions : Cette rubrique permet de spécifier les scénarios alternatifs qui peuvent se produire mais qui mènent tout de même au résultat souhaité, et les scénarios exceptionnels dans lesquels le cas d'utilisation va s'interrompre sans avoir atteint l'objectif de l'acteur.

Par convention, on numérote ces scénarios **A1, A2 …** pour les alternatives et **E1, E2…** pour les exceptions. Donnez à chaque scénario un titre qui explique sa nature. On commence chaque scénario par : « A l'étape 3 du scénario nominal, si XX se produit, alors… ». Les étapes sont ensuite numérotées, et suivent la forme des étapes du scénario nominal (système vs. acteur). Pour les alternatives, on termine souvent par « reprendre le scénario nominal à l'étape 5 ».

Si nécessaire, on pourra également fournir les rubriques suivantes :

Hypothèses : On décrit ici des hypothèses que le système n'est pas capable de contrôler, mais qui doivent être vérifiées. Typiquement on cite ici des contraintes matérielles.

Exigences non-fonctionnelles : On spécifie ici d'éventuelles contraintes non-fonctionnelles sur le système : temps de réponse, volumétrie, utilisation de polices XL, internationalisation…

Conseils pour élaborer les fiches détaillées

Pour élaborer les fiches détaillées de cas d'utilisation, il est fortement recommandé de faire des schémas représentant l'IHM envisagée. Ces maquettes d'IHM sont une base importante pour permettre de visualiser les scénarios et leurs enchaînements, ainsi que pour discuter avec le client et s'assurer que le besoin est correctement capturé. Ces maquettes indicatives (ce sont des dessins à ce stade) peuvent être insérées dans le document d'analyse à côté des cas d'utilisation correspondant. Cependant, les séquences d'interaction ne doivent pas s'appuyer excessivement sur ces IHM : elles sont plus générales, elles décrivent les échanges avec le système, sans faire trop de suppositions sur l'aspect.

Dans les étapes du scénario qui correspondent aux actions du système, il faut mettre l'accent sur ce que le système fournit, vu de l'extérieur par l'acteur. On ne cherche pas ici à spécifier comment le système réalise l'action, mais quel est l'effet perceptible pour l'utilisateur.

> *Attention à bien respecter le cadre et la forme de ces fiches détaillées. En particulier, il ne faut pas mélanger scénario nominal et alternatives. La séquence nominale doit être linéaire. Le niveau de détail recherché dans les fiches détaillées est relativement fin ; il faut lever toute ambigüité du cahier des charges.*

Exemple

UC01 : « Emprunter une vidéo ».

Date de création : 20/12/2014. Date de mise à jour : 1/1/2015.
Responsable : M. Dupont.
Version : 1.1.

Description: Un client du magasin souhaite emprunter une cassette vidéo *via* le distributeur automatique.

Acteur: Client.

Préconditions :
Le client possède une carte qu'il a achetée au magasin.
Le distributeur est alimenté en cassettes.

Séquencement nominal

1. Le client introduit sa carte.
2. Le système vérifie la validité de la carte.
3. Le système vérifie que le crédit de la carte est supérieur ou égal à 1 euro.
4. Appel du cas « UC02 : Rechercher une vidéo ».
5. Le client a choisi une vidéo.
6. Le système indique, d'après le crédit sur la carte, pendant combien de temps (tranches de 6 heures) le client peut garder la cassette.
7. Le système délivre la cassette.
8. Le client prend la cassette.
9. Le système rend la carte au client.
10. Le client prend sa carte.

Enchaînements alternatifs
A1 : Le crédit de la carte est inférieur à 1 euro.
L'enchaînement démarre après le point 3 de la séquence nominale :

3. Le système indique que le crédit de la carte ne permet pas au client d'emprunter une vidéo et l'invite à la recharger.
4. Appel du cas d'utilisation « recharger carte ».

La séquence nominale reprend au point 4.

Enchaînements d'exception
E1 : La carte introduite n'est pas valide.
L'enchaînement démarre après le point 1 de la séquence nominale :

2. Le système indique que la carte n'est pas reconnue.
3. Le distributeur éjecte la carte.

E2 : La cassette n'est pas prise par le client.
L'enchaînement démarre après le point 6 de la séquence nominale :

7. Au bout de 15 secondes le distributeur avale la cassette.
8. Le système annule la transaction (toutes les opérations mémorisées par le système sont défaites).
9. Le distributeur éjecte la carte.

E3 : La carte n'est pas reprise par le client.
L'enchaînement démarre après le point 8 de la séquence nominale :

9. Au bout de 15 secondes le distributeur avale la carte.
10. Le système consigne cette erreur (date et heure de la transaction, identifiant du client, identifiant du film).

E4 : Le client a annulé la recherche (il n'a pas choisi de vidéo).
L'enchaînement démarre au point 4 de la séquence nominale :

5. Le distributeur éjecte la carte.

Post-conditions
Le système a enregistré les informations suivantes :

- La date et l'heure de la transaction, à la minute près : les tranches de 6 heures sont calculées à la minute près.
- L'identifiant du client.
- L'identifiant du film emprunté.

Rubriques optionnelles
Contraintes non fonctionnelles
Le distributeur doit fonctionner 24 heures sur 24 et 7 jours sur 7.
La vérification de la validité de la carte doit permettre la détection des contrefaçons.
Contrainte liée à l'interface homme-machine : avant de délivrer la cassette, demander confirmation au client.

UC02 : « Emprunter une vidéo ».

Date de création : 10/5/2014.
Responsable : M. Durand
Version : 1.0

Description: Un client veut rechercher une vidéo *via* le distributeur automatique.

Acteur: Client.
Préconditions : le client est déjà connecté au système avec une carte valide

Séquencement nominal
(le choix du film se fait par genre)

1. Le client a sélectionné la recherche de vidéo, par exemple comme une étape de UC01.
2. Le système demande au client quels sont ses critères de recherche pour un film. Les choix possibles sont : par titre ou par genre de film.
3. Le client choisit une recherche par genre.
4. Le système recherche les différents genres de film présents dans le distributeur et les affiche. Les choix possibles incluent au moins action, aventure, comédie et drame.
5. Le client choisit un genre de film.
6. Le système affiche la liste de tous les films du genre choisi présents dans le distributeur.
7. Le client sélectionne un film.

Enchaînements alternatifs
A1 : Le client choisit une recherche par titre.
L'enchaînement démarre après le point 2 de la séquence nominale :
A1.1. Le client choisit une recherche par titres.
A1.2. Le système affiche la liste de tous les films classés par ordre alphabétique des titres.
La séquence nominale reprend au point 6.

Enchaînements d'exception
E1 : Le client annule la recherche.
L'enchaînement peut démarrer aux points 2, 5 et 7 de la séquence nominale :
Appel de l'exception E4 du cas « Emprunter une vidéo ».

Post-conditions
Le système a mémorisé le film choisi par le client.

Rubriques optionnelles
Contraintes liées à l'interface homme-machine
Quand une liste de films s'affiche, le client peut trier la liste par titres ou par dates de sortie en salles.
Le client peut se déplacer dans la liste et la parcourir de haut en bas et de bas en haut.
Ne pas afficher plus de 10 films à la fois dans la liste.

Figure 2 : Un exemple de fiche détaillée de cas d'utilisation, adapté de *« UML 2, Pratique de la modélisation, 3eme Ed. »*, 2010, Charroux, Osmani, Thierry-Mieg. L'appel en SN4 de UC01 se traduirait sur le diagramme par un <<include>> de UC01 sur UC02.

III. Diagramme de classe métier

Le diagramme de classe métier, ou diagramme de classe d'analyse, sert à représenter l'ensemble des informations que devra manipuler le système. Pour organiser les relations qui existent entre ces informations, on élabore un diagramme de classe simplifié, qui ne porte pas d'opérations. On peut voir ce diagramme comme le schéma d'une éventuelle base de données qui stockerait les données métier manipulées par le système. Les classes métier ne sont pas destinées à être implémentées tel quel, ce ne sont pas des diagrammes de niveau conception.

On utilisera uniquement les éléments suivants du diagramme de classe :

La classe métier

Représenté par un rectangle muni de trois compartiments :

- **Le nom** de la classe qui doit être pertinent pour le métier, donc souvent un terme tiré du cahier des charges.
- **Les attributs** listés sous la forme **nomAttribut :typeAttribut**. On se limitera à des types simples pour les attributs, et indépendants de tout langage cible : réel, entier, string, date. Pour les attributs qui sont des listes (on dit multi-valués en UML) on note **nomAttribut :typeAttribut[*]**. Par exemple : « valeurs:int[*] » pour une liste de valeurs entières.
- **Les opérations** laissées vide sur ces diagrammes métier. On peut même éviter de dessiner ce compartiment (dans l'outil également on peut *click-droit->filtrer* les compartiments à afficher).

L'énumération

Représentée par une classe portant dans le compartiment nom le stéréotype <<enum>>. Une occurrence d'énumération, comme son homonyme en java ou c++, prend une valeur parmi une liste donnée explicitement.

Le compartiment attribut porte une liste de valeurs qui sont les valeurs que peut prendre l'énumération. On ne représentera pas le compartiment opérations pour les enum.

Associations

On utilisera exclusivement des associations bi-directionnelles simples : elles sont représentées par un trait plein liant les deux classes, sans décoration particulière aux extrémités.

L'association porte à chacune de ses extrémités

- **Le nom de rôle**, c'est-à-dire le rôle que joue l'instance vis-à-vis de l'autre classe. Par exemple, sur une association entre une classe Chien et une classe Homme, le rôle de l'homme vis-à-vis du chien serait « maître ».

- **La cardinalité**, c'est-à-dire le nombre d'instances connectées. Cette cardinalité prend la forme : min,max. Les cardinalités les plus fréquemment employées sont
  - 0,1 : il peut y avoir une instance connectée (mais pas nécessairement)
  - 1 : il doit y avoir exactement une instance connectée
  - * : une liste d'instances est connectée

On fera attention sur les diagrammes de classe métier à bien renseigner toutes les cardinalités.

> *NB : On ne spécifie ni la navigabilité (tête de flèche), ni une nature pour le type de lien (par exemple agrégation ou composition avec des losanges blancs ou noirs). Ces choix techniques relèvent de la conception, mais ne sont pas pertinents pour décrire les données métier. Pensez plutôt à une implémentation sur BDD (ou ces notions de navigabilité et de contenance n'ont pas de sens) qu'à une réalisation en Java.*

Héritage

Il est représenté par une flèche à large tête triangulaire, et s'avère parfois utile pour organiser les données. Sémantiquement, si A dérive de B (noté A --|> B), A porte en plus des siens propres tous les attributs et associations de B.

On utilisera l'héritage avec modération dans ces diagrammes métier ; l'objectif n'est pas de faire de l'orienté-objet au sens de ce qui sera fait en conception (design patterns…).

Exemple

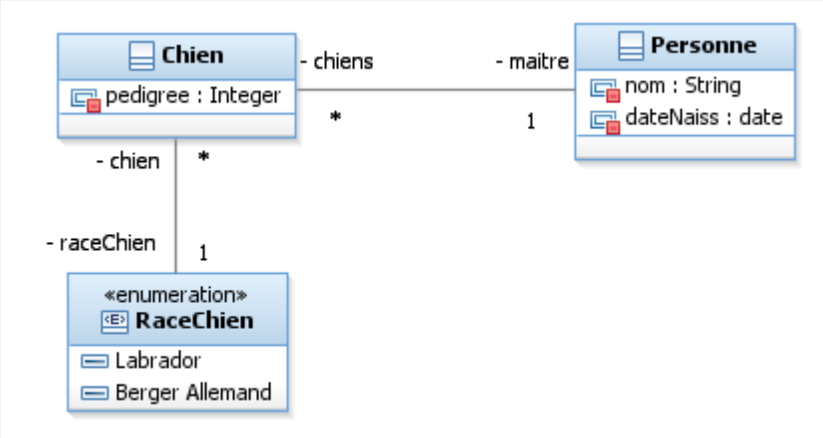


Figure 3 : Un diagramme de classe métier simple pour un chenil. Une personne peut avoir 0 ou plusieurs *chiens*. Tout chien doit avoir exactement un maître (qui est une Personne). Chaque chien a une raceChien qui est soit « Labrador », soit « Berger Allemand ». Réciproquement, à une race de chien peut être associée 0 ou plus instances de Chien. Les attributs de classe sont limités aux types de base.

IV. Diagramme de séquence niveau analyse

Le diagramme de séquence de niveau analyse permet de spécifier la nature exacte des échanges entre le système et les acteurs au cours de la réalisation des cas d'utilisation. Il oppose le système (une ligne de vie) aux acteurs (une ligne de vie par acteur qui interagit).

La classe système

Pour construire ces diagrammes, on introduit de façon artificielle une classe « système » qui porte toutes les opérations que les acteurs invoquent. Cette classe va permettre de recenser les responsabilités du système vis-à-vis des acteurs. L'essentiel dans l'élaboration de ces diagrammes est de réussir à obtenir la signature des opérations du système.

Cette classe fictive représente l'ensemble du système, et a donc connaissances de toutes les données du système. On peut imaginer que l'ensemble des éléments du diagramme de classe métier lui est accessible. Cependant, on évitera de faire figurer cette classe fictive sur le diagramme de classe métier, et de la lier par des associations aux classes métier, étant donné que sa nature est très différente d'une classe métier.

> *NB : Le système ne doit pas invoquer d'opérations sur les acteurs : on ne cherche pas à identifier les responsabilités de ces derniers. De plus, il est incorrect en UML de diriger un message vers un acteur. Les affichages du système se traduisent par des invocations du système sur lui-même (par exemple « afficherListeDisponible () »)*

On pourra distinguer parmi les opérations du système

- les responsabilités « publiques », c'est-à-dire les opérations qu'invoquent les acteurs, dont la signature doit être figée le plus vite possible. Cette signature ne sera en principe pas remise en cause dans les phases suivantes du développement. Elle doit être cohérente avec les fiches détaillées de cas d'utilisation où l'information que fournissent les acteurs a déjà été décrite. La signature doit s'astreindre à n'utiliser que des types simples (string, bool, int, float, date). Ces opérations correspondent aux étapes des scénarios de la fiche détaillée ou « L'acteur fait… ».
- Les responsabilités privées, c'est-à-dire les étapes modélisées sur le diagramme comme des invocations du système sur lui-même. Ces responsabilités qu'on peut découvrir au cours de ce travail de spécification ne sont pas le cœur de ce que l'on cherche à spécifier dans cette étape. Cependant, il peut être utile de les spécifier, car il faudra traiter ces besoins (indirects vis-à-vis des acteurs) dans la phase de conception. Leur signature peut être plus approximative, elle ne sera réellement spécifiée qu'en phase de conception. Typiquement on trouve dans ces opérations privées des contrôles « verifierDroitUtilisateur() » et des affichages « afficherListeUtilisateurAutorisés() ». Ces opérations correspondent à des étapes des scenarios de la fiche détaillée du type « Le système fait … », sur lesquelles on veut mettre l'accent (typiquement parce qu'elles paraissent complexe).

En dehors de ces invocations sur la classe « système », on pourra éventuellement représenter les créations et destructions d'instances des classes métier par le système. Cependant, on n'affectera pas de responsabilité (opérations) à ces classes métier (donc pas d'invocations d'opérations du système sur les classes métier).

Les acteurs

On considère dans ces diagrammes que la ligne de vie de l'acteur représente aussi son IHM si nécessaire. En d'autres termes, on ne cherche pas à représenter chaque clic de l'utilisateur (si c'est un humain), mais

plutôt de spécifier quelles sont les informations que fournit l'utilisateur au système dans le déroulement du scénario.

Ce niveau de description est cohérent avec l'information spécifiée sur les fiches détaillées, où l'on fait abstraction de la façon dont les informations fournies au système sont saisies par l'utilisateur.

Exemple

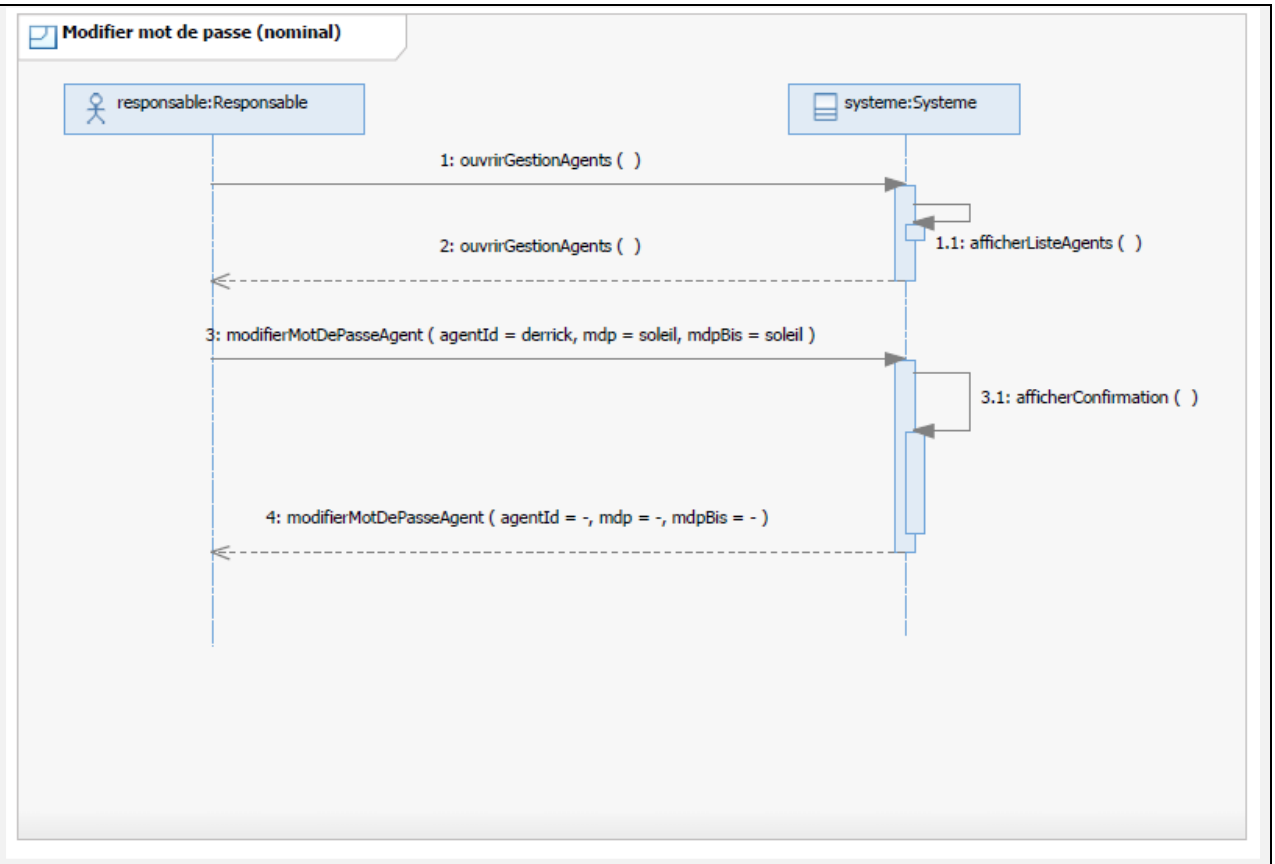


Figure 4 : Un diagramme de séquence de niveau analyse, tiré d'une annale de 2010. Comme on le voit, les affichages produits par le système sont décrits comme des appels du système sur lui-même. La ligne de vie de l'agent fournit d'un bloc les informations utiles, on ne modélise pas le processus de saisie lui-même.

Ce qui donne au niveau de la classe système les responsabilités :

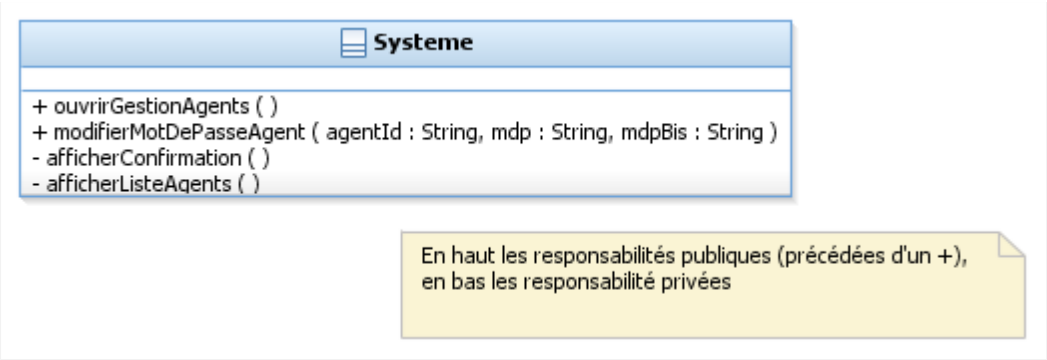


Figure 5 : Diagramme de classe présentant les responsabilités du système, induit par le diagramme de séquence de la figure 4. Cette classe « système » fictive n'est pas une classe métier mais plutôt un support pour spécifier de façon détaillée les responsabilités du système.

V. Tests de validation

Définition

Les tests de validation permettent d'offrir au client un moyen permettant de s'assurer que le modèle d'analyse couvre bien ses besoins.

Sans tests de validation, il n'y a qu'un unique sens de discours (du client vers l'équipe de réalisation, à travers le cahier des charges). Il faut absolument établir un discours inverse (de l'équipe vers le client) afin de bien s'assurer de la pertinence du modèle d'analyse.

En pratique, dans notre approche du cycle en V, ces tests seront rédigés par l'équipe de développement mais devront être validés par le client. Il est donc important de noter que les tests de validation doivent pouvoir être lu par des **utilisateurs**. Ils sont donc rédigés en langage naturel.

Les tests de validation doivent définir le comportement de l'application. Le système y est vu comme une boîte noire, on ne s'intéresse qu'aux effets perceptibles par les acteurs. On distingue deux types de comportements :

- Comportement nominal : contient une suite d'actions normalement suivies par l'utilisateur du logiciel et définit le résultat que doit pouvoir vérifier l'utilisateur.
- Comportement erreur : contient une suite d'actions que l'utilisateur ne devrait pas pouvoir effectuer et définit un résultat qui ne devrait pas pouvoir être vérifié par l'utilisateur.

Ces deux types de comportements partagent les mêmes caractéristiques :

- Ils sont définis à l'aide d'une suite d'actions que doit pouvoir effectuer (ou essayer d'effectuer) l'utilisateur.
- Ils contiennent une information qui doit pouvoir (ou non) être vérifiée par l'utilisateur.

Structure d'un test de validation

Le **titre** définit le titre du test !

L'**id** définit l'identifiant unique du test parmi tous les tests de validation de l'application. Il est utilisé pour les références croisées.

Le **contexte** définit un état dans lequel on doit se trouver afin de pouvoir exécuter le test. Ce contexte peut être obtenu à travers l'exécution réussie d'autres tests (on parle de suite de tests).

Exemples :

- Le jeu d'essai de la base de test TEST_01 (fourni avec les tests) a été chargé dans l'application
- La base de données doit contenir au moins 3 livres,
- Il faut être connecté à internet,
- L'utilisateur doit disposer d'un login et d'un mot de passe,
- L'exécution réussie du TV 03 a permis de connecter l'utilisateur

L'**entrée** définit l'ensemble des données d'entrée nécessaire pour l'exécution du test. L'entrée peut faire référence à des fichiers si besoins (les fichiers devront alors être fournis avec le test). Pour exécuter le test il est important que l'entrée soit entièrement spécifiée, c'est-à-dire que le testeur n'ait pas à inventer de données. Ceci assure que le test sera reproductible.

Exemples :

- Un login « toto » et un mot de passe « soleil »
- Un numéro de commande « C002 »
- Une date de départ « 01/01/2010 » et de retour « 28/01/2010 »

Le **scénario** définit l'ensemble des actions qui devront être réalisées par l'utilisateur. Cette séquence doit être aussi séquentielle que possible (pas de si, pas de boucle, pas de tant que, etc.) Sinon, c'est que le test contient plusieurs petits tests.

Le **résultat attendu** définit le résultat qui devra être vérifié (ou non) par l'utilisateur. Cette rubrique est essentielle.

Le **moyen de vérification** définit la façon dont doit être faite la vérification (visuelle, fichier, script, exécutif, …). Les effets attendus du test doivent pouvoir être testés par les utilisateurs, de l'extérieur du système.

Exemple

- Résultat Attendu : L'utilisateur « toto » est créé avec mot de passe « soleil ».
- Moyen de vérification : se loger à partir de l'écran d'accueil en utilisant ces identifiants (cf TV04).

Titre :	**id**
Contexte :	
Entrée :	
Scénario :	
Résultat attendu :	
Moyens de vérification :	

Il est important que ces tests soient lisibles ! Un test doit être non-ambigu dans son verdict (soit le résultat attendu est atteint, soit il ne l'est pas) et reproductible à l'identique.

Il est important que ces tests soient nombreux car chaque test ne fonctionne que sur un unique ensemble de données (entrée). Afin d'être complet, il faut alors définir beaucoup de tests. Comme on ne peut pas tout tester, l'essentiel est d'**avoir une bonne couverture des fonctionnalités**. On couvrira donc au moins chacun des scénarios nominaux et alternatifs décrits dans les cas d'utilisation.

Exemple

Titre : modification par le responsable du mot de passe d'un utilisateur	**Id TV03**
Contexte :	Le responsable est logé sur le système (cf. TV01). Il a ouvert l'interface de « gestion des utiisateurs » (cf TV02).
Entrée :	matricule : « Agent Derrick », nouveau mot de passe : « soleil »
Scénario :	1. Le responsable choisit l'agent Derrick dans l'interface de gestion des agents. 2. Il sélectionne la fonction « mettre à jour le mot de passe » 3. Il saisit le nouveau mot de passe deux fois dans une boite (affichage masqué) 4. Il confirme
Résultat attendu :	Le système affiche une confirmation, le mot de passe a été mis à jour.
Moyens de vérification :	Visuel pour la confirmation, ouvrir une session « agent derrick » avec ce nouveau mot de passe « soleil » sur le smart-phone (cf TV04) pour la deuxième partie.

La phase de Conception

La phase de conception a pour objectif de répondre à la question : **COMMENT ?**

L'objectif est de concevoir une découpe du système en **composants** faiblement couplés qui réponde aux besoins identifiés en analyse.

Cette phase sert est découpée en deux étapes :

- La conception générale ou architecturale vise à définir une découpe de l'application en composants. On définit les responsabilités de chaque composant vis-à-vis des autres à l'aide d'interfaces requises et offertes. La granularité à cette étape est au niveau composant ; on ne détaille pas encore la réalisation interne des composants.
- La conception détaillée permet de préparer la phase de réalisation, on y spécifie de façon aussi complète que possible la façon dont chaque composant sera implémenté en pratique. Cette étape fait appel à une bonne connaissance des mécanismes objet (design patterns…) et des bibliothèques ou framework offerts par la plate-forme ciblée pour l'implémentation.

Ce chapitre commence par rappeler les définitions utiles pour élaborer des diagrammes de composants (section I). Les principales instanciations envisagées des composants sont décrites à l'aide de diagrammes de structure interne (section II). On présente ensuite les règles qui guident l'élaboration des diagrammes de classe et de séquence en conception (section III). La section IV présente les étapes de validation qui correspondent à la conception, au niveau composant avec les tests d'intégration, et au niveau détaillé avec les tests unitaires.

I. Diagramme de Composant

Composant:

Définition Un composant est une entité logicielle qui **requiert et offre des interfaces** pour l'interaction avec le reste du système. Un composant est peut être **instancié** en un ou plusieurs exemplaires dans le système. Une **interface requise** (ou **utilisée <<usage>>** dans l'outil**,** symbolisée par un *socket*, **--C**) spécifie une dépendance du composant sur une réalisation de cette interface. Une **interface offerte** (ou **réalisée <<realization>>** dans l'outil, symbolisée par un *lollipop,* **--O**) désigne le fait que le composant réalise (implémente) cette interface. La signature des opérations des interfaces ne porte que des types simples (float, int, string, date…) ou des types abstraits (interfaces) et est spécifiée séparément de la définition du composant.

Le composant est une unité de grain moyen pour le développement de l'application. En focalisant sur les dépendances **fonctionnelles** (interfaces, donc purement opérations), et en interdisant les dépendances **structurelles** (liens d'association directs entre classes de composants différents) le modèle de composants favorise l'émergence d'architectures **faiblement couplées**.

Exemple

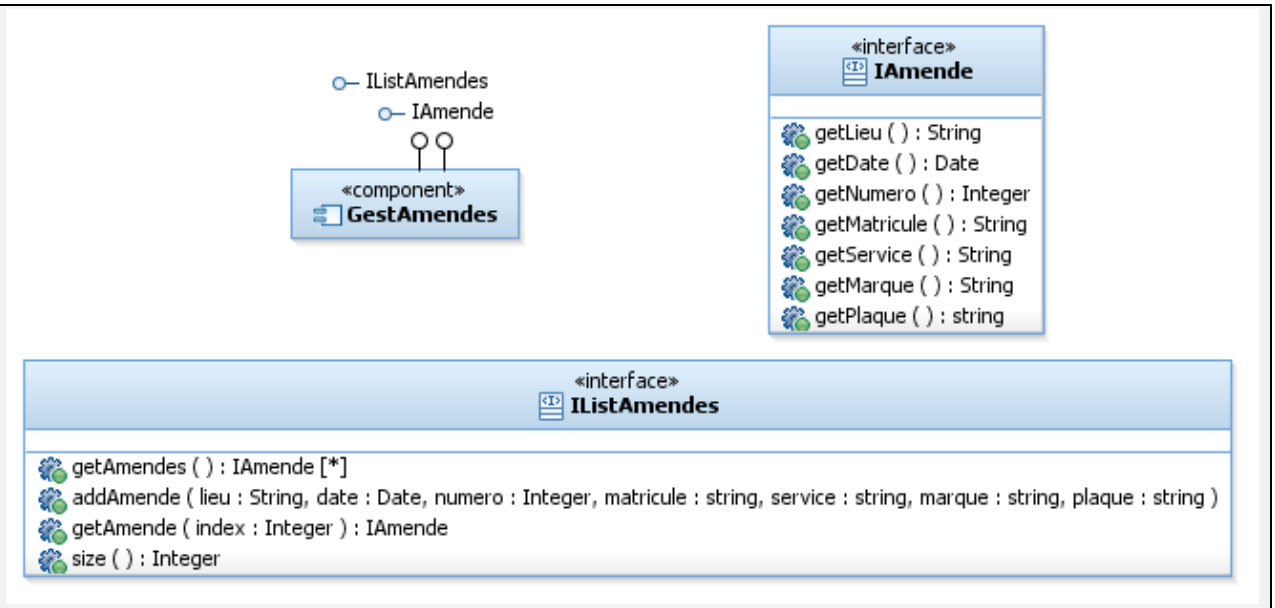


Figure 6 : Un exemple de définition d'un composant « bout de chaine » : il offre deux interfaces (liées entre elles par leurs signatures) et n'en requiert aucune. Cet exemple, tiré d'une annale de 2010, montre une façon de modéliser un composant qui joue le rôle de référentiel pour un jeu de données (ici des amendes). Ce cas est assez fréquent ; on trouvera souvent sur l'interface « IListXXX » également des opérations permettant la modification, la suppression, et des opérations de recherche diverses pour l'interrogation des données.

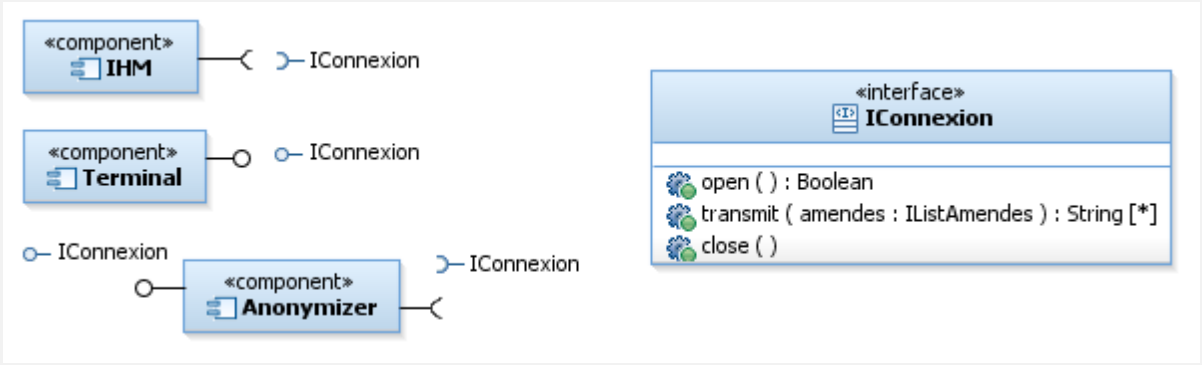


Figure 7 : Tiré du même examen, ici trois composants sont représentés :

* L'IHM (interface homme machine) qui n'offre aucune interface logicielle aux autres composants ici et requiert une IConnexion

* Le terminal, réalisation bout de chaine de l'interface IConnexion

* L'anonymizer, un composant qui filtre une partie de l'information qui le traverse. Il offre une IConnexion (son entrée) et requiert une IConnexion (sa sortie).

L'intérêt ici est de pouvoir soit connecter directement une occurrence d'IHM à une occurrence du terminal, soit d'intercaler une occurrence de l'anonymizer entre les deux. Ces deux déploiements peuvent être décrits au moyen de diagrammes de structure interne.

II. Diagramme de Structure interne

Le diagramme de structure interne permet de modéliser la façon dont un *classeur* (*classifier* en anglais, essentiellement soit une classe, soit un composant) est construit à partir d'éléments plus simples. Dans l'ue, on s'en sert pour spécifier à l'échelle du système une instanciation des composants dans une configuration particulière. On l'emploie donc en phase de conception architecturale pour compléter le diagramme de composants en fournissant une **topologie de connexion** des **instances de composants**.

Part / Instance

Définition Un **part** est une instance d'un élément du système (classe ou composant), muni d'un **nom de rôle** vis-à-vis du composant englobant et éventuellement d'une cardinalité donnant le nombre d'occurrences jouant ce rôle dans le classeur englobant (par défaut 1 seule occurrence).

Le diagramme de structure interne met en valeur l'aspect compositionnel d'une application. Dans l'ue, on s'en servira exclusivement pour décrire la façon dont un système est construit en assemblant des instances de composants. Le « composant » englobant est donc le système en entier sur ces diagrammes.

Connecteur d'assemblage

Définition Un **connecteur d'assemblage** lie deux **part** entre eux. Quand les part sont des instances de composants, le connecteur d'assemblage (*assembly connector*) modélise le lien entre l'interface requise de l'un et l'interface offerte de l'autre qui doivent être congruentes. Ce lien orienté se représente par un trait plein muni en son milieu d'une *socket* accrochée à un *lollipop* (--CO--).

> *Attention à ne pas confondre ce lien avec les définitions de composants. Le connecteur d'assemblage lie des **instances** de composants. La définition d'un composant ne spécifie pas comment il sera connecté (très délibérément !), mais simplement ses interfaces requises et offertes.*

Hiérarchie, Ports, Connecteurs de délégation

Les diagrammes de composant supportent naturellement une spécification hiérarchique : un composant peut être spécifié comme étant l'assemblage de composants plus simples. Dans une approche qui procède par raffinements successifs, cela permet de découper progressivement le système en parties plus simples, selon le principe de développement « diviser pour régner », qui consiste à séparer un problème complexe en plusieurs problèmes simples.

Définition Un **port** d'un composant représente un point d'interaction entre le composant et son environnement ; il peut porter une ou plusieurs interfaces offertes ou requises. Il est représenté par un petit carré à la frontière du composant.

Un **connecteur de délégation** lie un port du composant à un part contenu dans le composant. Il spécifie que l'interface requise (resp. offerte) du composant englobant est requise (resp. offerte) par cette instance (*part*) contenue dans le composant. Ce lien orienté se représente par un flèche du port vers le part (interface offerte) ou du part vers le port (interface requise).

La définition hiérarchique des éléments du système est un élément essentiel pour maîtriser la complexité. Cependant en termes de modélisation technique dans l'outil, l'introduction des ports (nécessaire pour utiliser les connecteurs de délégation) alourdit sensiblement la modélisation. En pratique on évitera donc dans l'outil de faire appel à ces mécanismes. On pourra cependant faire des diagrammes qui « zooment » sur un composant pour obtenir la même idée de décomposition hiérarchique.

Exemple

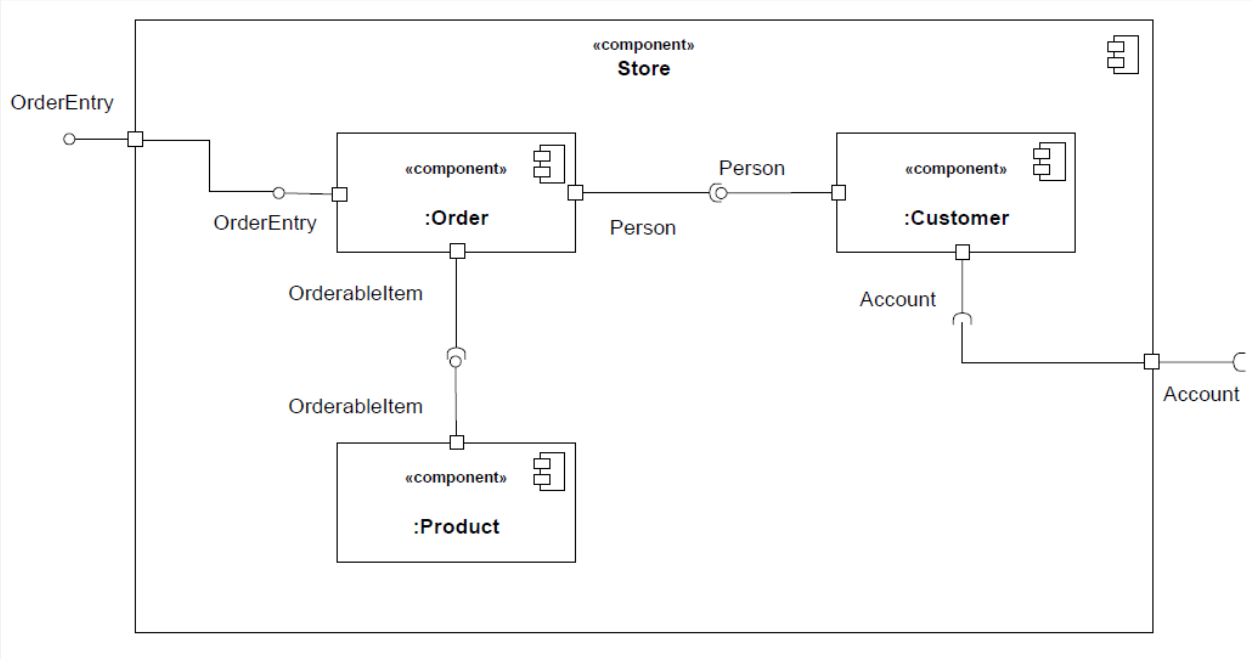


Figure 8 : Cet exemple récapitulatif (tiré de la norme) montre un diagramme de structure interne détaillant la réalisation du composant « Store ». Le « Store » contient trois part, instances des composants « Order », « Customer », et « Product ». L'interface requise du part « :Order » est satisfaite dans ce déploiement par le part « :Product » (qui offre cette interface), ce qui est modélisé par un connecteur d'assemblage.

Ce diagramme exhibe aussi l'utilisation des ports simples et des connecteurs de délégation. Le magasin (Store) offre « OrderEntry » à son environnement ; ce diagramme montrant la structure interne du composant en boîte blanche montre que cette interface est en fait offerte par le part « :Order ». Idem

pour « Account », interface requise du « Store ». On peut ainsi d'abord raisonner sur le Store vis-à-vis du reste du système, puis par raffinement regarder comment implémenter son comportement.

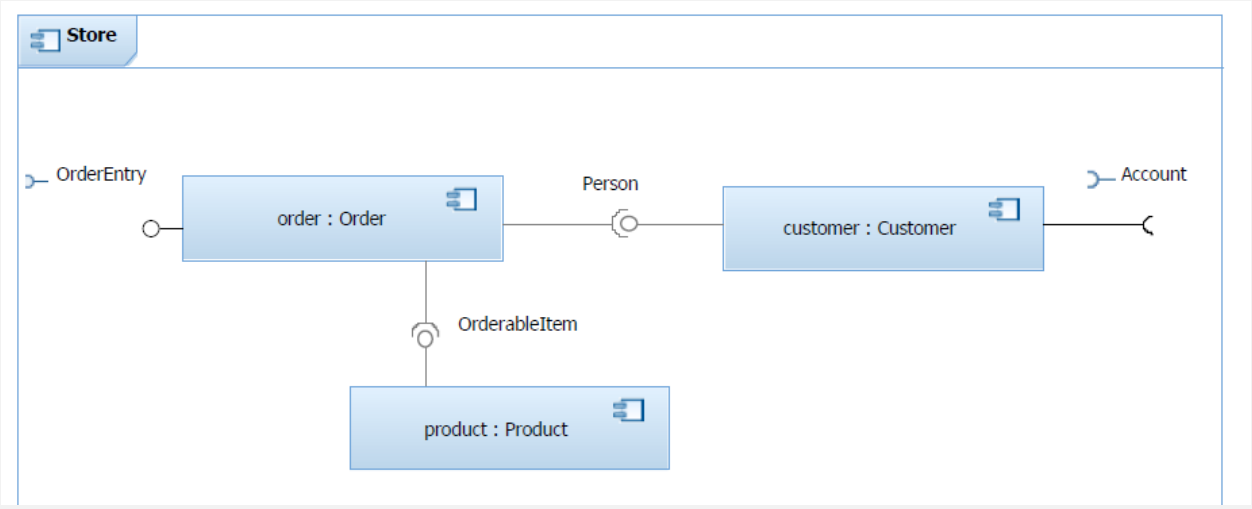


Figure 9 : Dans l'outil RSA utilisé dans l'ue, on se contentera de cette deuxième modélisation, moins lourde car ne nécessitant pas l'utilisation des ports. On y retrouve globalement la même information que sur le diagramme précédent. On considère ici (avec un léger abus de notation) que les interfaces qui ne sont pas satisfaites « débordent » du composant englobant. Quand le diagramme de structure interne représente tout le système (et pas un composant ou un sous-système), a priori toutes les interfaces requises/offertes sont satisfaites par des connecteurs d'assemblage.

III. Conception Générale et Conception Détaillée

Les diagrammes de classe de niveau conception vont détailler comment les composants sont réalisés à l'aide de classes de conception. Ces classes (contrairement aux classes métier utilisées en analyse) ont pour objectif de servir de support à l'implémentation du composant. En conséquence elles portent des opérations, et font appel aux divers mécanismes de l'objet : héritage, délégation, application de design patterns…

> *NB : La définition des éléments d'un diagramme de classe ou de séquence (présentés en amphi) ne fait pas l'objet d'une section dans ce document ; ce sujet est déjà largement couvert dans les supports de LI342 (UML pour les développeurs) et LI314 (Programmation Orientée-objet en Java), ainsi que dans divers livres et supports web décrivant UML.*

Conception Générale

La conception générale a pour objectif de découper le problème à traiter, tel qu'il est spécifié à l'issue de l'analyse, sur des composants plus simples qui traitent des sous-problèmes. On ne cherche pas à cette étape à complètement détailler la réalisation des composants, mais plutôt à raisonner sur un découpage structurel de l'application (en termes de responsabilités des composants vis-à-vis des données) et sur un découpage fonctionnel (en termes d'interfaces requises et offertes par les composants).

Découpe Structurelle

Dans un premier temps, on va s'inspirer des diagrammes élaborés en analyse, qui donnent déjà une bonne idée des données qu'il faut stocker et manipuler. L'approche proposée consiste à découper le diagramme de classe métier sur les composants, en évitant au maximum d'avoir des associations liant des classes de composants différents (dépendances structurelles). Ensuite, à l'aide d'interfaces ou d'identifiants, on élimine les dépendances structurelles restantes entre composants.

Découpe Fonctionnelle

On raffine ici la spécification des échanges entre composants de l'application. Pour cela on réalise des diagrammes de séquence où chaque ligne de vie représente un composant de l'application. Ces diagrammes ont pour objectif de fixer les signatures des opérations des interfaces offertes et requises des composants. C'est aussi l'occasion de valider la découpe structurelle, en s'assurant que le composant a bien accès aux données nécessaires pour réaliser les traitements. Si ce n'est pas le cas, il faudra introduire une dépendance du composant vers celui qui porte la responsabilité des données utiles, à travers une interface supplémentaire ou en enrichissant une interface existante.

Exemple

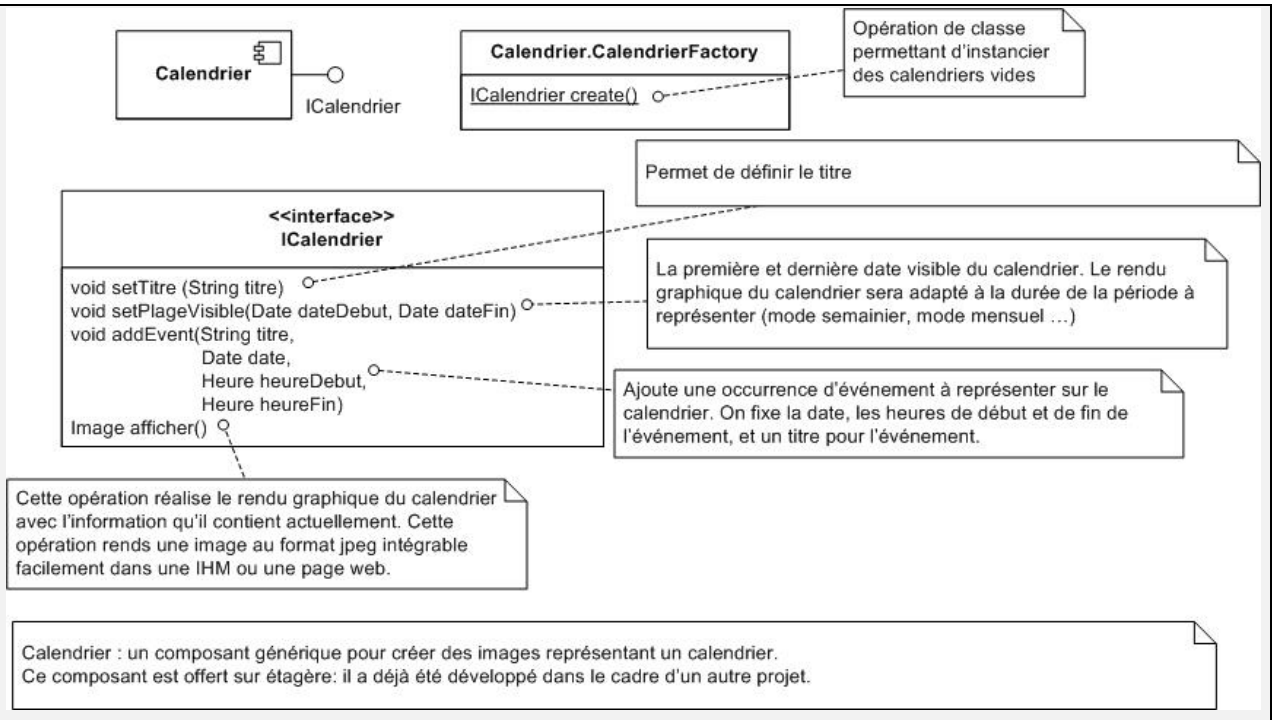


Figure 10 : La spécification du composant Calendrier, cohérent avec le diagramme de la figure 11. Dans cet examen ce diagramme était fourni. L'accent est mis sur la signature et la sémantique des opérations. On voit également la définition d'une *factory static*, cf ci-dessous la section conception détaillée.

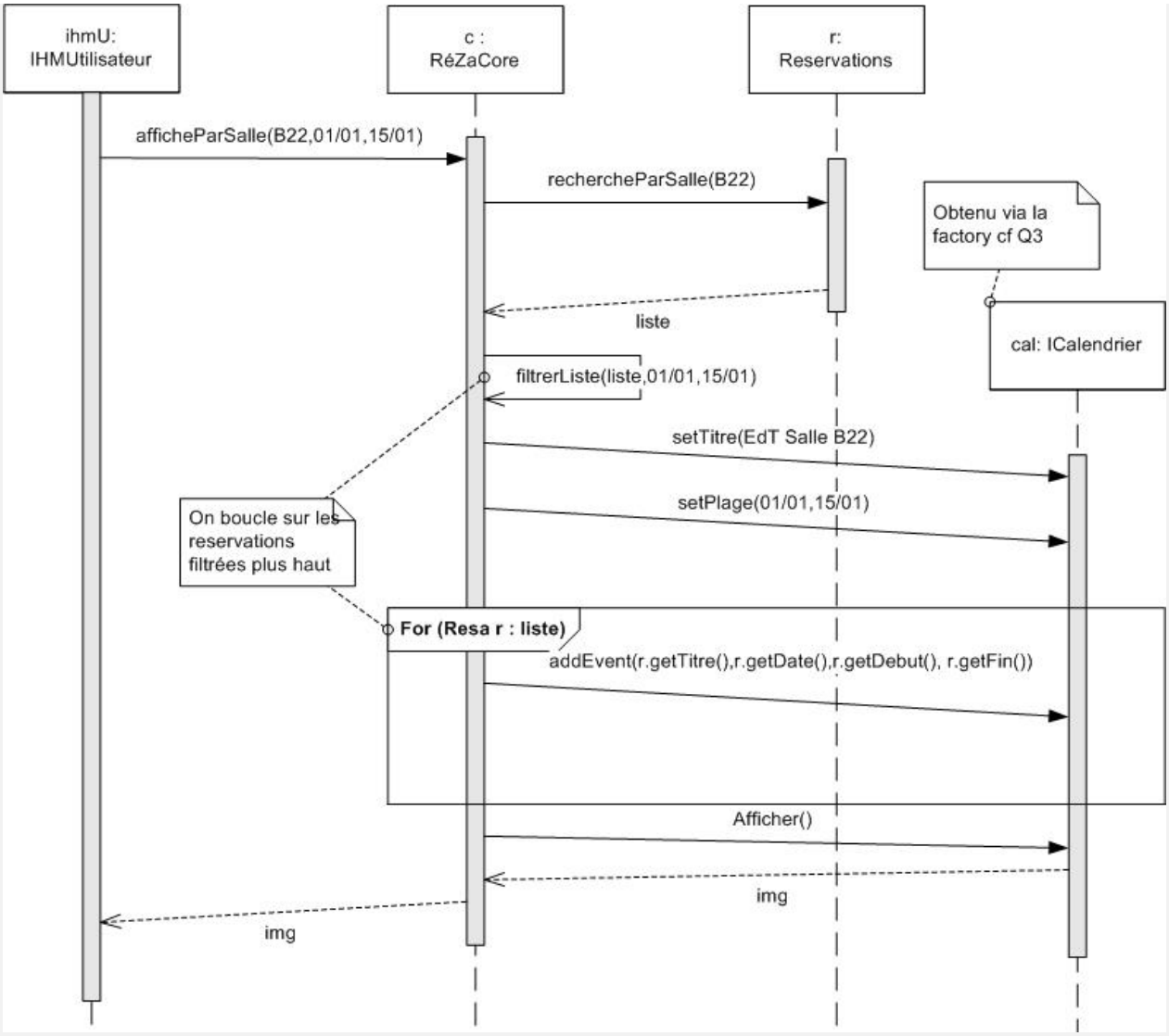


Figure 11 : Un diagramme de séquence de niveau conception générale tiré d'une annale de 2009. On y voit une ligne de vie par composant et la façon dont le besoin identifié en analyse (*afficheParSalle*) est satisfait par les composants qui interagissent. On déduit de ce type de séquences d'interaction les interfaces requises/offertes des composants (cf Figure 10).

Conception Détaillée

On construit à ce stade pour chaque composant un diagramme de classe, qui doit être cohérent avec le diagramme de composant : si le composant offre une interface « IOffert », on doit trouver une classe du composant qui implémente cette interface. Réciproquement, si le composant requiert une interface « IRequis » on doit trouver une classe du composant qui utilise cette interface, soit via un attribut/association typé « IRequis », soit via la signature de certaines opérations.

On trouvera également une classe jouant le rôle de *façade* pour le composant, c'est-à-dire un point d'entrée pour accéder aux fonctionnalités du composant. C'est cette classe qui sera instanciée quand on cherche à instancier le composant. Un mécanisme de *Factory* peut être mis en place pour cacher le type de cette classe : une classe portant une opération static qui rend une occurrence de la façade, mais typée par l'interface qu'elle réalise.

> *NB : Cette approche simple avec façade et factory « static » est utilisée dans l'ue pour lier le modèle de composants à sa réalisation en Java SE ; une implémentation plus directe dans une plateforme supportant les composants (J2EE, .Net, OSGI, …) est aussi possible.*

A l'intérieur d'un composant, on commence à orienter les associations entre classes (un document porte une liste d'exemplaires ? ou est-ce l'exemplaire qui porte une référence vers le document ?). On va aussi spécifier comment les associations sont réalisées (liste chaînée, tableau taille fixe, table associative de type HashMap…) L'instanciation de design patterns (Adapter, Composite, Strategy,…) pour organiser les traitements et les données est indiquée à ce stade.

Exemple

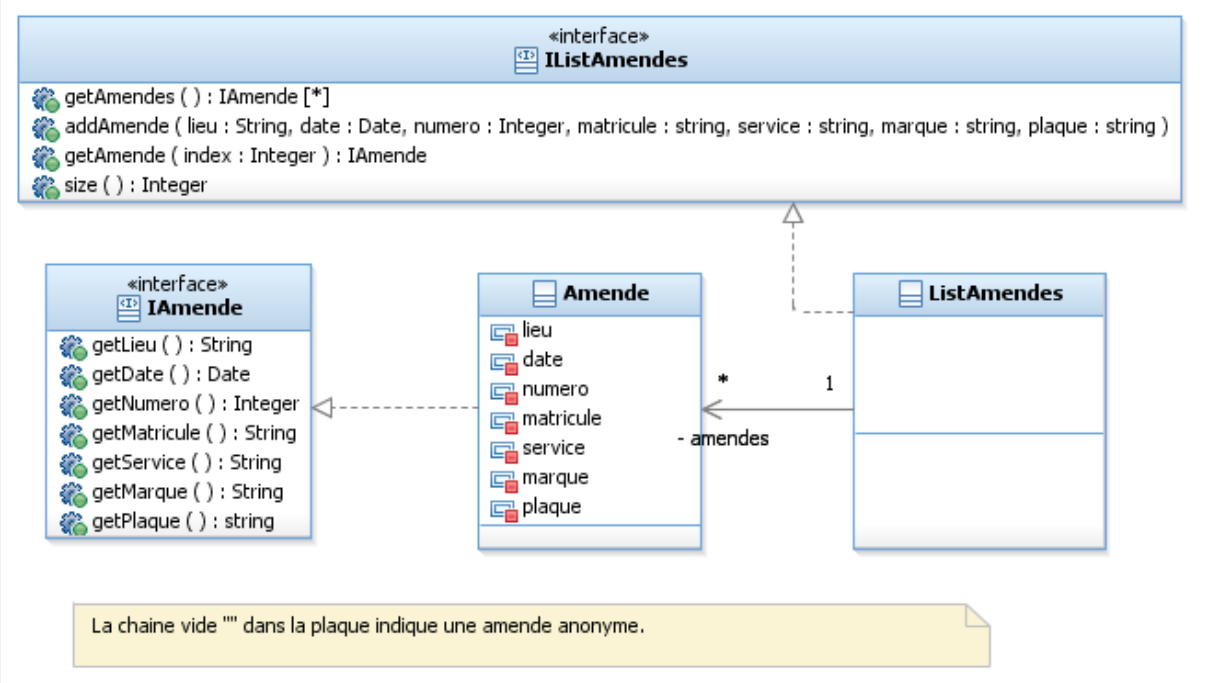


Figure 12 : Une spécification détaillée du composant **GestAmendes**, cohérente avec la spécification de ce composant (cf. Figure 6). Le lien d'association entre les amendes et les agents qui les ont déposées, identifié en analyse, est ici éliminé en passant par un identifiant : le « matricule » est un identifiant permettant d'interagir avec le composant responsable de la gestion des agents.

Plan Indicatif du contenu des séances

1. Introduction
Mise en place, organisation de l'UE : équipes ~6 étudiants, appliquer la démarche présentée à un projet sur un cahier des charges au fil des séances de TME, deux soutenances de présentation (slides) du résultat. Partiel sur l'analyse, examen sur la conception.

Motivation: complexité croissante du logiciel, défi d'adaptation aux technos, améliorer la productivité et la qualité, mieux répondre au besoin

Définitions : Génie Logiciel. Méthode de Développement. Rôles et Intervenants. Artefacts du développement (Docs, modèles, code, tests, ...)
Nécessité de la *diversité* (points de vues, rôles)
Le modèle comme un référentiel ; les diagrammes = vues sur une partie du modèle selon un point de vue, cohérence via le modèle.
UML, historique, un standard généraliste pour modéliser, survol des points de vues/types de diagrammes proposés.

2. Analyse 1 :
Introduction et positionnement du cycle en V.

Définir l'objectif du système sans fixer sa réalisation.
Du cahier des charges à la spécification technique du besoin.
Diagrammes de Use case, acteurs, mission, frontière du système.
Modélisation des données du problème avec des classes "Métier"

3. Analyse 2 :
Spécifier les comportements et les interactions.
Fiches détaillées de cas d'utilisation (spécifier !).
Maquettes d'IHM.
Diagrammes de séquence acteurs vs. système, signature des échanges et lien responsabilités (à implanter car tiré d'un besoin utilisateur) du système.

4. Tests de validation et transition vers la conception
Définitions : Notion de test (testeur, input, output, oracle), complétude...
Test de validation : en boite noire sur le système entier
Lien diagramme de séquence (entrées/sorties)
Spécification d'un test reproductible.
Matrice de traçabilité (lier les développements aux enjeux), outils de gestion et de suivi du besoin (requirements).
Table des matières indicative d'une spécification technique du besoin.

Transition analyse/Conception : rupture du cycle/point de vue sur le système.
Du "quoi ?" au "comment?".

5. Conception Architecturale
Motivation : diviser pour régner, maîtriser la complexité, limiter l'impact des modifications sur une partie du système, découpler les parties de la solution.

Description d'une architecture logicielle (ADL): connecteur, composant, topologie de connexion indépendants les uns des autres.

Modèles de composants : interfaces (dépendances fonctionnelles), composants (+connecteurs = interface offertes ou requises).
Instanciation/configuration des composants et diagramme de structure interne.
Présentation des diagrammes UML pour spécifier les composants et leurs configurations d'instanciation.

Gestion de l'hétérogénéité (langages, plateformes), des dépendances (évolutions indépendantes des versions) …
Services d'une plateforme composants (nommage réparti, sécurité, marshal/unmarshal, instanciation à chaud, persistance...)

Présentation rapide de plateformes orienté composant : corba/IDL, osgi/eclipse, COM/.Net, J2EE/RMI, WS/SOAP/WSXML/REST-json...
Ouverture vers le Service Oriented Architecture (SOA), SaaS…

Application au cycle :
Système boite noire (analyse) => ensemble de composants
Décomposition fonctionnelle : à partir des diagrammes de séquence d'analyse, obtenir des diagrammes de séquence inter-composant.
Décomposition structurelle : à partir des données du diagramme de classes métier, affecter la responsabilité des données aux composants.

6. Conception Détaillée

Retour sur les composants (exemples fréquents):
Les composants de modèle jouant le rôle de référentiel (ajouter, supprimer, naviguer, modifier des données).
Les composants représentant des vues ou IHM (pas d'interface offerte).
Les composants d'observation, de façade, d'adaptation, de décoration (cf. design pattern).

Validation des interactions inter-composant, faisabilité des composants.

Conception détaillée d'un composant en orienté objet.
Lien entre le composant et les classes le réalisant : interfaces offerte (réalisée) ou requises (utilisée).
Réalisation de composants "à la main" en J2SE.
Réalisation d'un composant de modèle sur un SGBDR.
Conception OO d'une fonctionnalité décrite par une interface + des scénarios d'invocation.
Lien vers les Design Pattern : Facade, Factory, Composite, Adapter, Decorator.

7. Tests d'Intégration et Unitaires
Validation des composants : test en isolation (composants bouchons, testeurs, configuration pour le test)
Séquences pertinentes de test, contrat d'utilisation d'un composant (protocol state machine).

Configurabilité des composants
Injection de dépendances type guice, lien vers AOP
Instanciations dans des scenarios variés, rôle des mock/bouchons.

Application au cycle :
Construction des tests pertinents à partir des séquences inter-composant, issues de l'analyse.
Tester au niveau composant : mise en place avec JUnit (pour simuler nUnit "à la main")

Survol de JUnit et du test unitaire.
Réalisation/Codage : contenu vu dans d'autres UE.

Conclusion sur le cycle en V.

8. Cycles Agiles
Introduction à la qualité logicielle
Définition d'une métrique, métriques courantes de qualité (couverture de tests, métriques de complexité...)
CMMI, ISO... rôle et conditions de délivrance
Certification, logiciel critique (ouverture sur la vérification formelle) vs. Développement pour le web.

Forces et faiblesses du cycle en V : lourd mais bien spécifié (systèmes critiques, gros projets, grosses équipes...).
Problèmes de Time-to-market, réactivité technologique, faible visibilité client, ...
Mal adapté aux petites structures ?

Cycles itératifs, amélioration continue PDCA, prototypage rapide.
Méthodes agiles, positionnement historique.
Principes agile : présentation du manifeste et de ses valeurs.

Roles : responsabilité collective, implication client, ...
Outils : intégration continue, test-driven, red/green, user stories, CRC cards...
Pratiques : open-space, oral, réunion quotidienne, time box, taches/kanban, binômes, ...

Exemple plus structuré de SCRUM : sprint, daily scrum, backlog, burndown chart, ...

Mise en action, exemples en pratique.

9. Modèles, méta-modèles, langages
(Séance d'ouverture vers le M2, vers des UE comme Ingénierie des Modèles, liens sur ILP)

Défauts d'UML : vaste, compliqué, pas de sémantique opérationnelle.
Evolution vers des petits langages exécutables/opérationnels : Domain Specific Languages.

Principal résultat du développement d'UML : technologies FOSS standardisées (OMG) pour la modélisation et les modèles. Implantations de référence du standard (EMF).

Méta-modèle, grammaire, syntaxe concrète et abstraite.
Lien sur Ontologie, diagramme de classe métier, BD O-O, séparation M/VC.

Méta-Méta modèle : MOF, et ça suffit !
Implantation concrète dans EMF (présentation du framework et des principaux outils/services).

Paradigmes model-driven : Un Modèle est un graphe, toute information est graphe, tout calcul est une transformation de modèle.
Transformations M2T/M2M/T2M.

Outils et applications : intérêt en pratique d'un outil qui prenne un langage en argument pour un développeur de langage,
Définition en pratique d'un petit DSL (exemple avec XText).

10. Conclusion
Retour global sur ce que l'on doit retenir de l'UE, bilan, synthèse.
La réalité en entreprise vs l'idéal présenté dans le cours.

Retours sur les modèles de composants, conception détaillée, exemples.
Préparation à l'examen (discussion sur des Annales), ce qu'il faut faire et ne pas faire...