UNIVERSITÉ LIBRE DE BRUXELLES

2018-2019

Analyse et Méthodes

Notes du cours INFO-F-204

Mark Diamantino Caribé
UNIVERSITÉ LIBRE DE BRUXELLES

Un client a un problème. Nous devons relier la demande du client à une solution.

Prendre l'ordinateur et coder : mauvaise solution.

Moyennement, 30% de projets informatiques réussissent.

Pourquoi ?

1) Analyse de la problématique du client ;
2) Travail en équipe (car le client veut le logiciel rapidement) → communication.

Solution : Software engineering.

C'est une discipline différente d'autre types d'ingénieries car il n'y a pas de contraintes physiques. Le cerveau humain est la seule contrainte.

Histoire : Après 20 ans de logiciel (années 70) on s'est rendu compte qu'il fallait suivre une méthodologie. Inspiration → Ingénierie civile. Différences avec la construction d'un pont.

Pont	Logiciel
On le voit	On ne le voit pas
Ne bouge plus	Évolue continuellement
L'environnement ne change pas	Les utilisateurs/exigences peuvent changer

Évolution du logiciel :

- Est dégradante (Dégradation inévitable, il faut la ralentir)
- L'ossature est rarement mise à niveau (Il ne faut pas avoir peur de changer du code)
- Chaque modification est de plus en plus compliquée (Il faut adapter le code aux possibles modifications)

Méthode Waterfall (développement séquentiel)

C'est la première méthode. On procède par phases :

1) Requirements Collection → On demande au client ce dont il a besoin. Il faut comprendre le client et son métier. Rédaction de la documentation.
2) Analysis → Modélise et précise les besoins (« WHAT »). Analystes définissent les besoins, la forme de l'application. Le client peut ne pas comprendre la documentation, donc il faut lui montrer des prototypes d'écrans. L'analyste peut utiliser l'orienté objet afin de faire un lien entre la conception et l'implémentation.
3) Design → Modélise et précise une solution (« HOW »). Les architectes conçoivent l'architecture de l'application, font des diagrammes et disent quelles librairies utiliser.
4) Implémentation → Création de la solution adéquate. Les codeurs reçoivent la description des architectes et développent sans poser de questions. Chacun travail en parallèle sur un module.
5) Testing → Procédure de vérification de la solution, le but est celui de trouver le nombre maximum de comportements problématiques ou erreurs.
6) Maintenance → Corrective, de compatibilité (l'environnement du logiciel change), perfective (client veut ajouter des fonctionnalités)

Problèmes :

- Divisée en couches (communicant par des documents et chaque phase doit être terminée avant d'entamer la suivante)
- Communication
- Un codeur ne connaît pas la finalité, donc ne peut pas détecter un problème potentiel.
- Le coût de correction d'une erreur est exponentiel car il faut remonter la cascade.
- La documentation n'est pas vraiment explicite pour le client.
- Approche très lente. En plusieurs années les besoins du client peuvent changer.
- La documentation ne reste jamais à jour → code devient illisible.

Avantages :

- Projet bien documenté (au début)
- Projet bien planifié. Utile pour les gros projets.

Méthode itérative (ou développement agile)

Il s'agit d'un enchainement de petits cycles Waterfall.

On livre des morceaux de code chez le client. En fonction de son feedback (car on va bien évitement pas tout cerner) on modifie à nouveau le code et on le livre.

Ne se base pas sur une analyse importante car c'est quand on a les mains dans le code qu'on se pose les questions.

Dans la pratique → Beaucoup de cycles itératifs de quelque semaines (normalement deux) afin d'enrichir de plus en plus l'application.

Avantages :

- Plus de contact avec le client
- Un rythme
- Moins d'échec
- Moins de temps
- Plus de motivation

Orienté Objet

Type de données abstraits

La définition (ou spécification) est séparée de l'implémentation.

Pourquoi ?

- L'implémentation peut changer. On peut la modifier sans modifier toute l'application → meilleure maintenance.
- Facilite la réutilisation et l'assemblage d'éléments.
- Permet d'écrire des petits modules. Le code bien découplé est synonyme d'un bon architecte.

Conseils :

- Essayer d'être le plus abstrait possible.
- Réutiliser les méthodes déjà codées (voir « Héritage »).
- Avoir des bouts de code qui ne connaissent pas beaucoup des autres.

Les Objets

Avec les ADT, il y avait séparation entre le traitement et les données. Les objets réunissent les deux. L'union entre traitement et données est plus proche de la pensée humaine.

Qu'est-ce qu'un objet ?

Conceptuellement : Un objet a une réalité unique.

Exemple réel, une craie possède :

- Un état → longueur, poids, couleur ;
- Un comportement → sa taille diminue quand on l'utilise ;
- Une identité → Ce qui permet de la distinguer d'un autre objet avec le même état et comportement.

On traduit le concept de craie (la réalité du client) en un programme qui gère les mêmes objets.

En résumé : Objet = état + comportement + identité.

Dans la pratique :

Un objet est une structure de données (cachées par principe d'encapsulation) qui répond à un ensemble de message.

- L'état → La structure, les attributs ;
- Le comportement → L'ensemble de messages qui décrivent son comportement. Les méthodes mais attention (voir « Interaction entre les objets ») ;
    - Décrivent le comportement à l'envoie et la réception d'un message par un autre objet ;
    - Peuvent accéder et manipuler la structure interne de l'objet.
- L'identité → Un identifiant ou la place de l'objet en mémoire.

Interaction entre les objets

On passe des objets en paramètre à un message. Exemple : lion.manger(what).

Différence entre messages et méthodes :

Les méthodes décrivent le comportement à la réception d'un message par un autre objet : ce dernier voit s'il accepte le message et comment il va l'exécuter. Le « Method Lookup » est le mécanisme qui permet de choisir la méthode qui va être appelée en réponse à un message. Dans l'exemple

précèdent, « what » reçoit le message et ses méthodes décident son comportement, la méthode « manger » représente le message.

Polymorphisme

Message ≠ Appel de fonction. Quand on envoie un message on dit quel objet doit le recevoir. Une fonction peut porter sur n'importe quoi. L'interprétation du message dépend de l'objet. Exemple : lancer une craie ≠ lancer une éponge. Cependant, les deux objets sont lançables.

Donc, on envoie le même message à deux objets différents qui vont répondre diversement.

En se basant sur ce concept, le polymorphisme permet une programmation beaucoup plus générique car, le message peut être envoyé sans savoir quel est le type de l'objet sur lequel la méthode va s'appliquer.

Classe

Décrit la structure d'un objet, c'est-à-dire l'ensemble des entités qui composent l'objet.

L'objet connaît une classe car il est une instanciation de celle-ci.

Quelle est l'utilité d'une classe ? → Créer des objets.

L'héritage

L'héritage permet de spécialiser un objet.

L'idée est de décrire un objet abstrait, puis des objets qui en héritent et récupèrent tout.

Principe

Les sous-classes :

- Héritent tous les attributs et les méthodes de leur classes parentes ;
- Peuvent surcharger les méthodes de leur classes parentes pour s'adapter à d'autres besoins (overriding[1])

Le but de l'héritage n'est pas d'éviter le copier/coller. C'est de permettre :

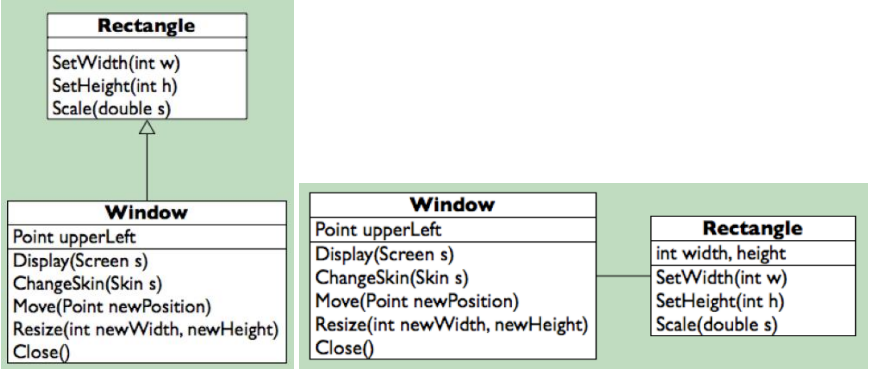
- Le polymorphisme ;
- Un code facilement maintenable ;
- Spécialiser l'objet.

Règle fondamentale → Pour bénéficier de l'héritage, il faut pouvoir se dire :

1) « B est une sorte de A » ;
2) « Tout ce qu'on sait faire avec A, doit pouvoir se faire avec B » (Si par exemple, B hérite de A et que des choses n'ont pas de sens pour B, l'héritage n'est pas correct).

[1] Overriding ≠ Overloading (Permet d'avoir plusieurs fonctions de même nom mais signature différente).

3) « Je dois hériter ou utiliser ? » Attention ! L'héritage ≠ usage. Exemple : Une classe « fenêtre » ne devrait pas hériter d'une classe « rectangle » car une fenêtre n'est pas une sorte de rectangle. Nous voulons plutôt faire usage d'un rectangle. L'objet fenêtre va déléguer[2] une partie de son travail à quelqu'un d'autre (partage de la responsabilité).



Types d'héritage

- Héritage simple → Une classe hérite d'une et une seule classe au niveau directement ascendant. Structure à arbre où chaque classe a un parent et des enfants ;
- Héritage multiple → Une classe hérite de plusieurs classes. Les langages modernes (C++ exclu) ne permettent plus ce type d'héritage car pose plus de problèmes que de solutions.

Self/this et super

Self/this → Est une référence vers l'objet courant. Attention, il s'agit d'un lien dynamique car l'objet courant pourrait être une classe fille par exemple. Donc, au moment de la compilation nous ne connaissons pas quel objet est « this ». Idée fausse → Self référence toujours la classe la plus basse dans la hiérarchie.

Super → Est le parent de l'objet courant. Attention, il s'agit d'un lien statique car il suffit de regarder la classe dont on hérite et on a trouvé. C'est connu à la compilation. C'est le mécanisme pour se référer aux méthodes « overridden ». Il faut l'utiliser si nous voulons réécrire ou enrichir la méthode du père (donc dans une méthode qu'on surcharge). Si la méthode est différente, il s'agit d'un « bad super send ».

Références

Un bout de code manipule un objet par des références. Elles permettent de nommer un objet.

Quand nous avons une référence sur un objet de type A, on peut pointer vers un objet B (ou C), si B hérite de A. Signification → On peut manipuler un objet de type B (ou C) comme un A.

Le C++

Le C++ laisse énormément de liberté.

[2] Délégation → Concept qui permet à un objet de reporter une tâche à un autre objet.

Attention ! Pas de polymorphisme sans « virtual ». Par défaut les méthodes ne sont pas virtuelles, donc il faut ajouter « virtual » à la méthode parente qui doit être redéfinie. One peut utiliser « override » dans les classes dérivées car signale si nous avons redéfini une méthode qui n'existe pas dans les classes parentes.

Attention aux modificateurs de visibilité. Il faut toujours hériter de manière publique. Dans le cas contraire il y a le risque de réduction de visibilité et donc la violation du principe de polymorphisme.

Accès résultant		**Dérivée : mode d'héritage**		
		public:	**protected:**	**private:**
Base : accès	**private:**	private	private	private
	protected:	protected	protected	private
	public:	public	protected	private

.

Framework

À la suite de l'évolution informatique on ne développe plus soit même mais on utilise des API, c'est des Framework (comme un plugin). Il s'agit d'un typique exemple d'application de l'orienté objet.

Les Framework externes sont très utilisés.

Principe typique de Framework qu'on utilise → Changer le comportement d'une méthode sans devoir la redéfinir.

Classe Abstraite

Il s'agit d'une classe :

- Que nous ne pouvons pas instancier directement ;
- Dont l'implémentation n'est pas complète (doit contenir au moins une méthode abstraite[3]).

Quelle est l'utilité ?

→ Créer une classe définissant un ensemble d'éléments en commun, afin que d'autres puissent en héritent ;

→ La classe n'instancie pas des objets car il s'agit d'une abstraction de quelque chose que va être spécialisée par les classes qui en héritent.

Quand j'hérite d'une classe abstraite, il faut implémenter toutes les méthodes abstraites.

[3] Une méthode avec une signature mais pas l'implémentation. Elle ne peut qu'exister que dans une classe abstraite.

Placement des méthodes

Il est conseillé de placer une nouvelle méthode le plus haut dans la hiérarchie des classes. L'implémentation d'une nouvelle méthode devrait se reposer sur des méthodes déjà définies.

En cas contraire, l'héritage et le polymorphisme sera empêché, car il faudra réimplémenter la nouvelle méthode.

UML – Unified Modeling Language

Dans les années 90 il y avait une cinquantaine de systèmes de modélisation différents. Il y avait besoin d'un langage unifié, ce qui a été publié en 97.

Conceptuellement → L'UML est une boite à outils. On ne nous dit pas comment (ou dans quel ordre) les utiliser. Le processus est propre à celui qui l'utilise.

En pratique → Il s'agit d'un langage semi-formel[4] de modélisation orienté objet qui est basé sur un méta-modèle. Un méta-modèle est un modèle qui définit un modèle. Son objectif est de passer à un niveau d'abstraction supérieure, car on modélise le modèle qui permet de construire.

Composition

Deux types d'éléments en UML :

- 5 Vues → Représentations complètes du système. Mettent en évidence certaines parties du système. Chaque vue peut être représentée par un ou plusieurs modèles.
- 9 Diagrammes → Décrivent les fonctionnalités du système du point de vue de l'utilisateur de ce système.

Objectifs/Avantages :

- Facilite la communication
- Facilite l'organisation du travail
- Uniformise tous les langages de modélisation
- Est indépendant des langages de programmation utilisés

Les vues :

- Vue des cas d'utilisation (ou « Use case view ») → C'est la description des fonctionnalités du système vue par des acteurs [5]externes. Elle correspond aux besoins attendus par chaque acteur (QUOI et QUI). On y trouve les diagrammes « use case » et « activity ». Cette vue est utilisée par les clients, designers, développeurs et testeurs.

- Vue logique (ou « Logical vue ») → C'est une vue de haut niveau (conceptuelle) C'est la définition du système vu de l'intérieur. Explique COMMENT les fonctionnalités sont

[4] Car formel dans les notations mais pas dans la sémantique.
[5] Entité avec laquelle le système intéragit.

conçues ou fournies. On y trouve les diagrammes d'objets et de classes. Cette vue est utilisée par les designers et les développeurs.

- Vue d'implémentation (ou « Component view ») → Définit les dépendances entre les modules et leur organisation. C'est très proche de l'implantation car il y a des classes, bibliothèques, fichiers de configurations et bases de données (c'est-à-dire les composantes de la solution finale). Cette vue est utilisée par les développeurs.

- Vue des processus (ou « Concurrency view ») → C'est la vue temporelle et technique qui permet de décrire les notions de tâches concurrentes et de synchronisation. Décrit comment les composants interagissent entre eux. Cette vue est utilisée par les intégrateurs de systèmes et les développeurs.

- Vue de déploiement (ou « Deployment view ») → Indique les composants qui vont s'exécuter. Par exemple : le client, le serveur, la carte à pouce. Il s'agit d'une description de l'architecture physique de chaque élément du système.

Les diagrammes :

- Diagramme de cas d'utilisation → Décrit les fonctionnalités du système du point de vue de l'utilisateur et son utilisation typique. Une boite contient ce qu'on fait. Les acteurs sont représentés par des personnes. Ne doit pas montrer une des fonctionnalités du système mais plutôt expliquer un service.

- Diagramme de classe → Représente les classes à haut niveau. Ici on ne peut qu'utiliser le nom des classes. On peut voir les liens entre les classes.

- Diagramme d'objet → Permet de représenter les instances des objets des classes et les relations entre ces objets. C'est donc une représentation statique car ne permet pas de représenter les messages envoyés entre les objets.

- Diagramme d'état → Représente les différents états d'un programme, objet ou sous-système.

- Diagramme de séquence → Représente les séquences de communication entre les objets. On modélise les interactions, ce sont comme des messages envoyés.

- Diagramme de collaboration → Comme le digramme de séquence mais on met en évidence la structure, pas le temps qui passe.

- Diagramme de component → Identifie les composants et les relient.

- Diagramme d'usage où chaque cube représente une unité de traitement ou de stockage, serveur ou base de données. On y met les canaux de communication.

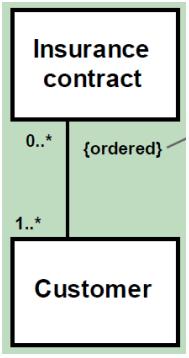
- Diagramme d'activité → Représente le côté comportemental de l'application. On y montre les étapes comme on montrerait l'algorithme d'une méthode. On peut y mettre des choix, des actions. Il s'agit d'un diagramme de haut niveau.

Concepts qu'on retrouve dans tous les diagrammes

Stéréotype → Est une syntaxe qui permet d'ajouter de la sémantique à la modélisation des classes. Concrètement, un stéréotype permet au concepteur d'étendre le vocabulaire de l'ULM afin de créer des éléments qui ont des propriétés spécifiques. Exemple : le stéréotype « acteur », on peut avoir une icône avec ceci.

Tagged values → Sont une association entre un nom et une valeur. Par exemple on peut associer une clé et une valeur à un élément.

Notes → Rajoute des informations de façon informelle.



Contraintes → Symbole « {…} ». Limitent toujours l'usage d'un élément. Il y en a de plusieurs types.

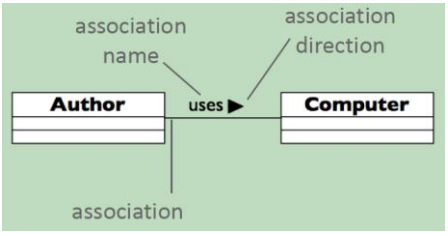
Les plus utilisés sont :

- {ordered}, {xor}
- {implicit} → La relation est conceptuelle.
- {changeable} →Des liens entre des objets pourront être enlevés, ajoutés ect…
- {addonly} → D'autres liens pourraient être ajoutés à l'opposé de l'association
- {frozen} → Un lien ne peut pas être modifié une fois ajouté.

Multiplicité → Permet de représenter, qu'un objet utilise plusieurs autres objets ou qu'un objet est composé par plusieurs autres objets. Exemples :

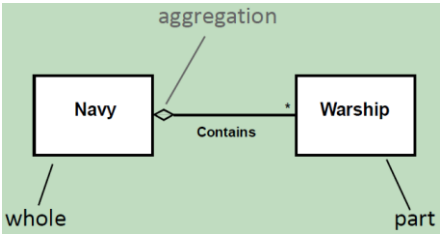
- « * » signifie de 0 à l'infini
- « 0..2 » signifie de 0 à 2
- « 2 » signifie 2
- « 1,3,5,7 » signifie 1 OU 3 OU 5 OU 7
- « 1, 3..5, 7..* » signifie (1) OU (un nombre entre 3 e 5 compris) OU (un nombre entre 7 et l'infini)

Relation entre les classes



Association → Une association indique que les objets instanciés des deux classes sont en relation, il se connaissent et peuvent s'envoyer des messages.
Symbole : ——————

Attention ! Il s'agit d'une simple ligne, ce n'est pas une flèche, la direction est donnée par un triangle en-dessus de la ligne.

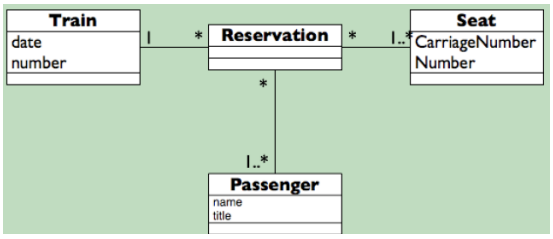


Agrégation → Conceptuellement, indique que quelque chose fait partie d'une autre. En pratique, indique qu'un objet peut être composé d'un autre objet. Exemple : La flotte est constituée de plusieurs bateaux. L'un a besoin de l'autre car la flotte sans bateaux n'est plus une flotte.
Symbole : ◇——

Composition → C'est comme l'agrégation mais la relation est plus forte. Conceptuellement, si on détruit le « père » alors il faut tout détruire. Dans le cas d'une flotte, il ne s'agit pas d'une composition car si on détruit la flotte, les bateaux peuvent vivre en dehors. Attention ! Il n'y a pas de vérité générale pour le choix.
Symbole : ◆——

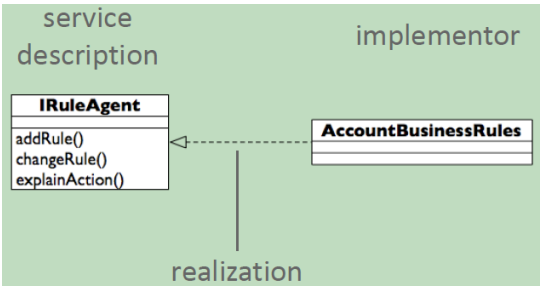
Généralisation → Il s'agit de l'héritage. Symbole : ——▶

Autres relations



Relation ternaire → C'est une relation entre trois classes. Normalement est représenté par un losange mais parfois peut se transformer en classe, comme dans la figure.

Raffinement → Est une relation d'une classe vers elle-même. Ceci permet de raffiner la classe (on la redéfini pour l'améliorer mais elle a le même rôle que la version précédente). Utilité principale historique.



Réalisation → Est une relation entre 2 classes définissent la même chose et dont l'une n'hérite pas de l'autre obligatoirement, les deux sont compatibles. Possède les avantages de l'héritage multiple mais sans les inconvénients, c'est-à-dire le multi-polymorphisme sans avoir de multi-héritage.
Symbole : - - - - - - - - - - - ▷

L'ingénierie des besoins

La phase de capture des besoins est : la plus formelle, la plus importante, la plus subjective, celle avec le moins d'outils. Généralement la principale cause d'échec d'un projet est la compréhension des besoins du client.

Les besoins

Tout d'abord on se concentre sur ce que doit faire le système, pas comment il va le faire. C'est parce que les besoins décrivent le système. Généralement le client ne décrit pas la problématique en détail, donc c'est à nous de lui faire expliquer tout.

User et System Requirements

Les « User Requirements » c'est ce que l'utilisateur veut de manière brute et abstraite (le jargon du client). Les « System Requirements » sont la même chose mais de façon plus formelle (doivent rester compréhensibles). Dans les deux cas, on reste toujours que dans le « QUOI ? ».

Besoins fonctionnels ou non

Les besoins fonctionnels sont les services (classiques) que l'application doit rendre.

Les besoins non-fonctionnels sont des caractéristiques du système qui ne rajoutent pas des fonctionnalités. Ces derniers besoins sont importants même s'ils n'ajoutent pas des services car, peuvent avoir un impact non-négligeable. Ils peuvent être : robustesse, sécurité, etc. C'est aussi la façon dont on va construire le logiciel. Mais attention ! Tout requête du client doit être mesurable. Par exemple, le client peut nous demander un logiciel facile à utiliser, mais concrètement comment mesurer la simplicité ? En heures de formation ?

Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurence
Robustness	Time to restart after failure Percantage of events causing failure Probability of data corruption on failure
Portability	Percentage of target-dependent statements Number of target systems

Les « Domain Requirements » sont les besoins du domaine (généralement évidents pour le client). Des choses peuvent être considérés évidentes dans un domaine, par exemple la décélération d'un train. Le langage naturel est la principale cause d'ambiguïté, donc il faut l'éviter.
Rappel : le but est d'enlever les ambiguïtés. Donc, les informations doivent être exprimées de manière directe, précise et simple.

Comment écrire les besoins ?

Concrètement il s'agit de simple documentation. Il faut utiliser :

- Un langage simple et naturel (pas de synonymes, pas de longues phrases, pas de jargon logiciel ou notations formelles) car le but est d'éviter les ambiguïtés ;
- Des tables ;
- Des listes ;
- Des schémas ;
- Des diagrammes intuitifs ;

- Mettre en évidence des parties du texte (exemple : souligner, italic, bold…) ;
- Traçabilité → Numéroter les besoins car : Permet d'éviter le mélange des besoins (une phrase peut en exprimer plusieurs donc c'est pratique d'utiliser des outils) ; Permet de vérifier que les tests couvrent tous les besoins du client.
- Généralement un Template. Les entreprises de logiciels ont un Template qui représente un format standard permettant de toujours savoir quoi écrire. De plus les documents se ressemblent tous et sont donc plus faciles à lire.

Attention ! C'est notre responsabilité de voir et chercher les bons interlocuteurs afin de faire ce qui est vraiment utile pour le client. Exemple : Vois les gens qui utiliseront concrètement le logiciel dans le quotidien.

SRD – Software Requirements Document

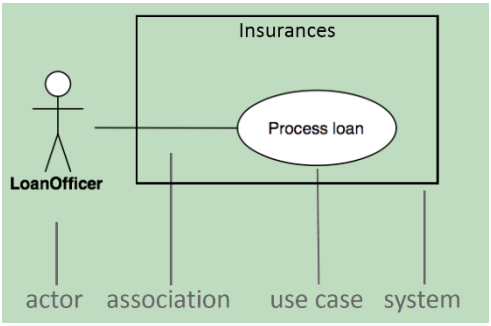
Sont les « User Requirements » mais plus détaillés. Comprennent de la documentation (parfois en langage non-naturel) mais aussi du UML.
Il faut être plus formels → on utilise une structure logique, des définitions, une table des matières etc. Astuce : on définit un dictionnaire des terminologies

Il existe des standard ISO pour ces documents (IEEE/ANSI 830-1998), c'est intéressant pour s'assure qu'on n'a rien oublié. Dans tous les cas, le niveau de détail doit être adapté au projet.
Attention ! Le SRD n'est pas figé, doit être maintenu. Il faut donc avoir une traçabilité des différentes versions afin d'expliquer les changements (système de gestion des versions).

Use Case Diagram

Elle décrit la vue des utilisateurs et des besoins. Il s'agit d'une vue centrale en UML car les besoins définissent le programme. Il y a une prévalence de besoins fonctionnels car les non-fonctionnels sont souvent décrits dans des documents en dehors.

La création du Use Case est itérative car on ne pourra jamais identifier tout dès le début.



Les acteurs → Quelque chose ou quelqu'un (pas spécialement une personne). On les représente par un bonhomme, dont le nom est le rôle. Quelquefois il semble qu'on n'a pas besoin d'un acteur mais en vérité quand quelque chose se passe, c'est pour servir quelqu'un.

Use cases → Description de l'ensemble d'actions que fait le système. Chaque use case est un exemple d'utilisation car on comprend souvent l'exemple que l'abstraction (c'est plus parlant). Ces exemples sont initiés par un acteur. En pratique on peut utiliser une description textuelle structurée ou de l'UML.
Attention ! On ne décrit que les informations qui entrent et sortent. Ce qui se passe derrière n'est pas important ici.

Description de l'use case → Généralement textuelle et s'adressant à des lecteurs non-techniques. On commence par les acteurs, puis les préconditions (ce qui doit être vrai avant que l'use case se

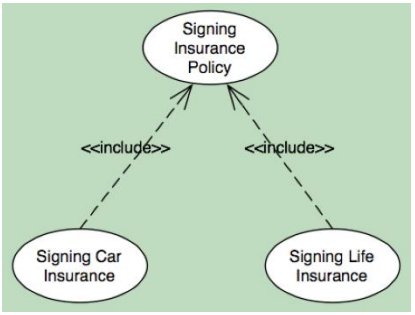
déclenche), puis les postconditions (propriété toujours vraie après l'appel de la méthode). L'étape fondamentale est l'interaction entre le système et l'utilisateur (on explique ce que l'acteur voit)

Relations

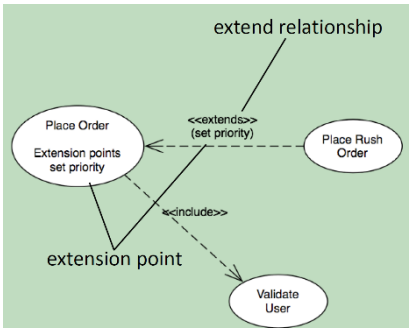
Généralisation entre les acteurs → Des acteurs héritent d'autres. Par exemple un client normal peut avoir moins de droits par rapport à un client vendeur.

Généralisation d'use-case → Permet de spécialiser un use-case.

« Include » → Permet l'inclusion du contenu d'un use case dans un autre. L'objectif est de définir le comportement quelque-par et l'inclure ailleurs. Sert donc à factoriser les use case, les réutiliser. Cette fonctionnalité ressemble à l'include de C++.



La relation d'extensibilité revient à décrire un use case en le laissant vide, car plus tard on pourra l'étendre à travers des « points d'encrage » (extension point). L'objectif est de s'en servir pour ajouter des options ou configurations.



Modélisation dynamique

Sert à la représentation de l'aspect dynamique du logiciel.
Des exemples de « dynamique » : L'envoie de messages entre objets, leur vie, etc.
Concrètement, on décrit le nécessaire, c'est-à-dire comment des objets coopèrent, communiquent, changent d'état.

L'UML possède 4 diagrammes :

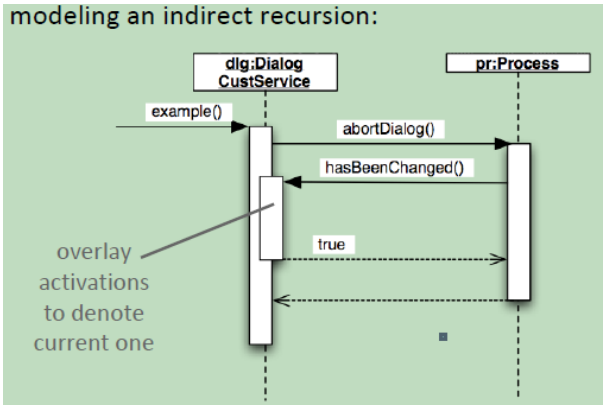
1) Sequence diagram → Dynamisme entre les objets
2) Collaboration diagram → Dynamisme entre les objets
3) State diagram → Dynamisme dans les objets
4) Activity diagram → Dynamisme dans les objets

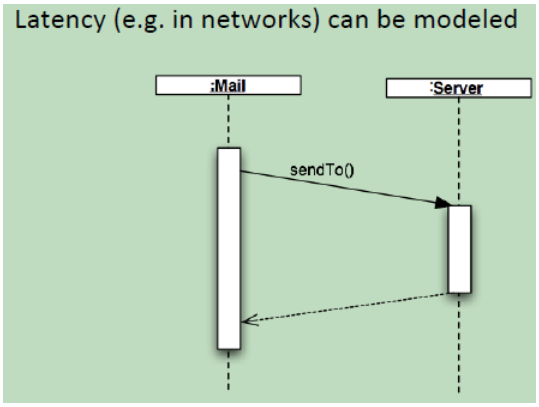
Les diagrammes d'interaction :

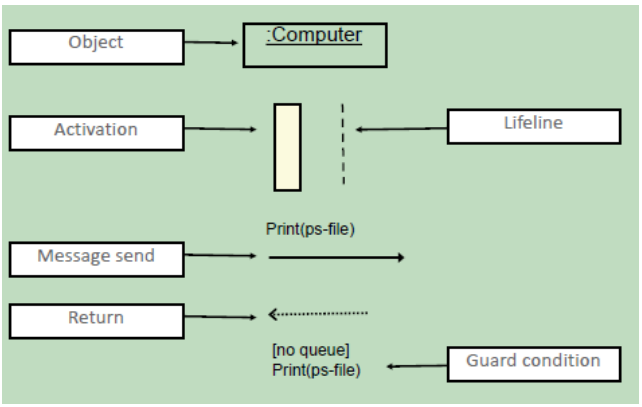
- Sequence Diagram → Se focalise sur le temps (l'aspect temporelle de l'interaction et de la communication). L'interaction entre les objets est exprimée par l'envoie et la réception des messages en fonction du temps.
  Deux formes :
  Forme générique (:Computer) → Permet de documenter plusieurs histoires (les alternatives possibles. Boucles, conditions et branchements sont possibles.
  Forme d'instance (C1:Computer)→ Une seule histoire. Pas de boucles, conditions ou branchements.
  Types de messages par rapport au temps :
  Synchrones → J'attends la réponse à mon message.
  Asynchrones → Je n'attends pas et j'exécute autre chose.

Réponses ---------->

Activation → Rectangle. Montre que l'objet est actif.
Multiplicité → On peut envoyer un message plusieurs fois.
Overlaid activations → Activations spéciales qui s'empilent. Un objet appelle l'autre, l'autre demande plus d'information au partent, le parent répond.
Guard → Condition sur l'envoi de messages.
Latence → Flèche oblique.





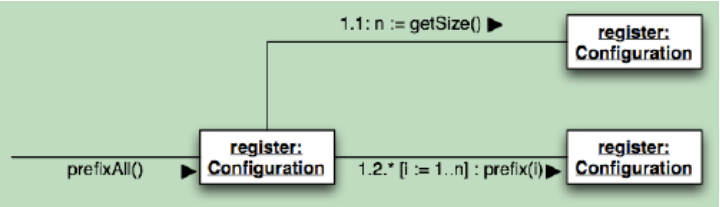


Collaboration Diagram → Se focalise sur la structure (l'aspect structurel de l'interaction et la communication). Montre les liaisons statiques et dynamiques des objets.
Ce diagramme vient en aide du Sequence diagram, qui risque être trop complexe parfois.
Nous pouvons numéroter les messages, poser des conditions et des descriptions.
Concrètement on y retrouvera :

1) Descriptions de messages.
   return-value := MessageName (ArgumentList)
   predecessor = sequence-number ',' … '/' → Utile pour exprimer la synchronisation (exemple : le message peut être envoyé si 1,2.3 ont terminé)

   sequence-expression = [intigers | name] [recurrence] ':'
   recurrence = '*' '[' iteration-clause ']' → Indique qu'un message est par d'une action effectuée en réponse à une autre action. La clause d'itération est par

exemple : [i :=1..n] ; la valeur de retour peut être explicitée.

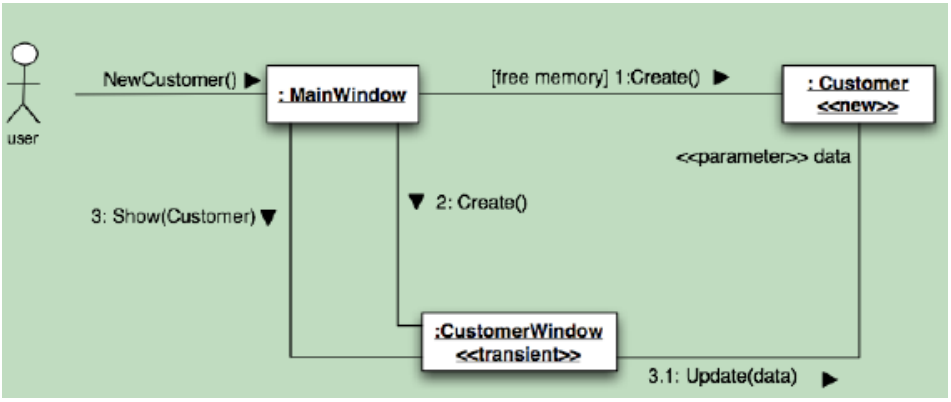


2) Stéréotypes de visibilité → Permet de savoir comment un objet sait qu'un autre existe, d'où il vient.

3) Stéréotypes d'existence → Permet de connaître le type d'existence d'un objet pendant son interaction.

<<association>>	receiver is known as an attribute(default)
<<global>>	receiver is global
<<local>>	receiver is local in the sending operation
<<parameter>>	receiver is parameter in the sending operation
<<self>>	receiver is the sending object
<<destroyed>>	receiver is destroyed during interaction
<<transient>>	receiver is created and destroyed during interaction

Exemple résumant points 2 et 3 :



UML Meta Modèle

Un métamodèle est le modèle d'un modèle. Dans notre cas on représente ce qu'on peut faire en UML (sa sémantique), à travers l'UML. Un exemple analogue est le compilateur C qui est écrit en C.

Chaque diagramme en UML est une instance du métamodèle.

Élément → Tout ce qu'on manipule en UML (classes, objets, associations, conditions…).
Deux typologies :

- Élément modèle → Représente l'abstraction du système ;
- Élément visuel → Représentation graphique de l'élément modèle.

Package → Est un conteneur d'éléments ou d'autres packages. Sert à les grouper en un ensemble cohérent. L'intérêt est de permettre à l'utilisateur de l'UML de mieux structurer les diagrammes. Il existe comme pour les classes, une notion de visibilité. Les éléments sont par défaut privés. Des éléments peuvent être partagés entre les paquets, en créant des relations de dépendance.

Mécanismes

Les mécanismes sont des briques de bases, utilisés dans chaque diagramme.

Des exemples de mécanismes sont : les notes, les stéréotypes, tes « tagged values », les relations de dépendances.

Stéréotype → Est un mécanisme qui permet d'étendre la sémantique de l'UML. On s'en sert pour classer des éléments. Un exemple de stéréotype est <<acteur>>.
On peut aussi créer des stéréotypes. Par exemple, dans le cas d'une application embarquée de trains, les stéréotypes nous permettraient de caractériser les classes qui s'occupent de la gestion de freins.

Tagged values → Association entre nom et valeur.

Notes → Des commentaires

Le test

Introduction
Nous utiliserons principalement les tests unitaires.
Test unitaire → On ne teste pas l'application mais ses composantes de manière indépendante des autres.

En pratique :

1- Isoler la méthode ou la classe
2- Écrire un code supplémentaire simple et linéaire afin de « simuler » le monde avec lequel les composantes isolées interagissent.
   Attention ! Il faut écrire un test en fonction de comment le code devrait fonctionner, PAS juste en regardant le code.

D'autres approches :

- Test exploratoire → On joue avec l'application. Ce n'est pas organisé. Ce n'est pas sûr car on essaie du bon fonctionnement du code.
- Des scripts qui injectent des données à l'application.

Attention ! Un test est un moyen de montrer qu'il y a une erreur, PAS de montrer qu'il n'y en a aucune.

Conditions

Invariante :

- Expression booléenne toujours vraie ;
- Permet d'écrire des preuves réelles avant (mieux que les tests) afin de tester le code avant de l'implémenter.
  Exemple : « La taille du stack est toujours >= 0 ». Si un jour cela devient faux, on a commis une erreur.

Préconditions et Postconditions sont des expressions assurées être vrais avants ou après quelque-chose.

Précondition :

- Doit être vérifiée pour que le traitement d'un message soit correct ;
- Ceux qui envoient le message doivent s'assurer que la précondition sera vraie ;
- Contrainte pour le monde extérieur mais soulagement pour l'objet ;
- On peut essayer d'avoir le moins de préconditions afin d'imposer moins de contrainte pour le monde extérieur ;
- On peut avoir un comportement de « best effort » si la précondition rate.

Postcondition :

- Contrainte pour l'objet, mais un bénéfice pour l'appelant.

Assertion :

- Permet d'exprimer une expression booléenne qui doit être vraie à un endroit donné.
- Conceptuellement → Permet :
  - d'expliquer ce qu'on s'attend à un certain endroit ;
  - de formaliser des préconditions, invariants et postconditions.
- Concrètement → Il s'agit d'une macro (« assert() » en C++) qui ne fait pas véritablement partie du code car n'en change pas le comportement.
  Avec une assertion, dès que l'invariant est violé, on sait exactement où l'erreur se situe.

Pendant la compilation → On peut choisir si activer les assertions ou pas.
Recommandation : Très utile avec la précondition car on détecte les problèmes plus tôt.

Test unitaire
Les test unitaires sont généralement écrit sous forme d'objets. Ces objets effectuent un « test suite ».
Un Framework peut se baser sur les assertions : Si aucune assertion n'est lancée, le test passe.
Intéressant :

- On peut également tester le comportement d'une erreur ;
- On peut inverser le retour du test (rater et non réussir) ;
- On peut asserter qu'une assertion soit produite.

Rédaction des tests

Normalement les développeurs écrivent du code et des tests au même temps (parfois d'abord les tests, puis le code).
Parfois on écrit d'abord les tests, puis le code.
Extreme programming → Technique de programmation fortement itérative qui se base là-dessus.
Elle est efficace car :

- Permet de clarifier les idées d'implémentation.
- Permet de tester d'abord la spécification

Le degré du test dépend de l'entreprise. Plus il est élevé, plus le niveau de confiance en son codé est élevé. Il s'agit d'une démarche de sécurité en plus qui permet de changer le code plus facilement.

Intéressant : Avant, on écrivait beaucoup de documentation, mais c'était difficile à maintenir. Il faut investir du temps dans le code, son auto-documentation et les tests. Les tests expliquent l'application.

Méthodologie de rédaction de tests → Si on a un bug, il faut écrire un test afin de le reproduire dans l'environnement de développement.

Mise en place des tests unitaires

Le test lui-même est découpé en parties. Les Framework des tests unitaires existent dans presque tous les langages.

CppUnit (en C++) → Un testcase hérite de « CppUnit ::TestCase. » La classe du test doit avoir le nom du test. Sa méthode « runTest() » utilise l'assertion pour tout vérifier, elle prend en paramètre une condition qui doit être vraie.

Définitions importantes :

- « Échec » assertion qui rate mais qu'on avait prévu rater.
- « Erreur » bug non détecté par une assertion.

Qualité des tests unitaires

Un test de qualité doit :

- Avoir un comportement déterministe et répétable. Ceci implique qu'on doit avoir le même résultat à chaque fois, sinon les résultats ne voudront riens dire.
  Conseil → Faire attention aux donnés/opérations aléatoires.
- Le lancement doit être très automatisé ;
- Tourner pendant longtemps (avant, pendant, après modifications) ;
- Être une source de documentation ;
- S'adapter facilement aux changements du code (être moins sensible aux changements) ;
- Traiter les choses difficiles en premier, pas des choses logiques ;

Framework de test

Les Framework de tests unitaires existent depuis 20 ans, originaires de la communauté SmallTalk.

Les éléments clés :

- « testcase » → Histoire qui englobe plusieurs méthodes de test ;
- « fixtures » → Un contexte. Un TestFixture permet de fixer un environnement logiciel pour exécuter des tests logiciels. Il s'agit d'un environnement constant à chaque exécution. Les fixtures sont mises en place dans les méthodes d'initialisation « setUp » (appelées avant chaque test) et de désactivation « tearDown » (appelées après les tests).
- « test suites » → Ensemble de testcases ;
- « test runner » → Ce qui fait déclencher les tests (On y enregistre les test qui seront ensuite exécutés) ;
- « outputter » → Enregistre le flux de sortie, (contient les résultats des tests).

Intéressant : Bien faire un logiciel est un défi. Les « Design patterns » sont des petites recettes qui nous aident à mieux apprendre des compositions typiques dont on s'en sert souvent. Un exemple de « Design pattern » est le « Listener » qui écoute et veut être prévenu quand un autre objet change.

À la fin, on obtient le nombre de tests passés. Attention ! Même si un jour on obtient un score de 100%, le programme pourrait ne pas être toujours bon. Il ne faut donc pas s'arrêter parce qu'on a « assez » testé.