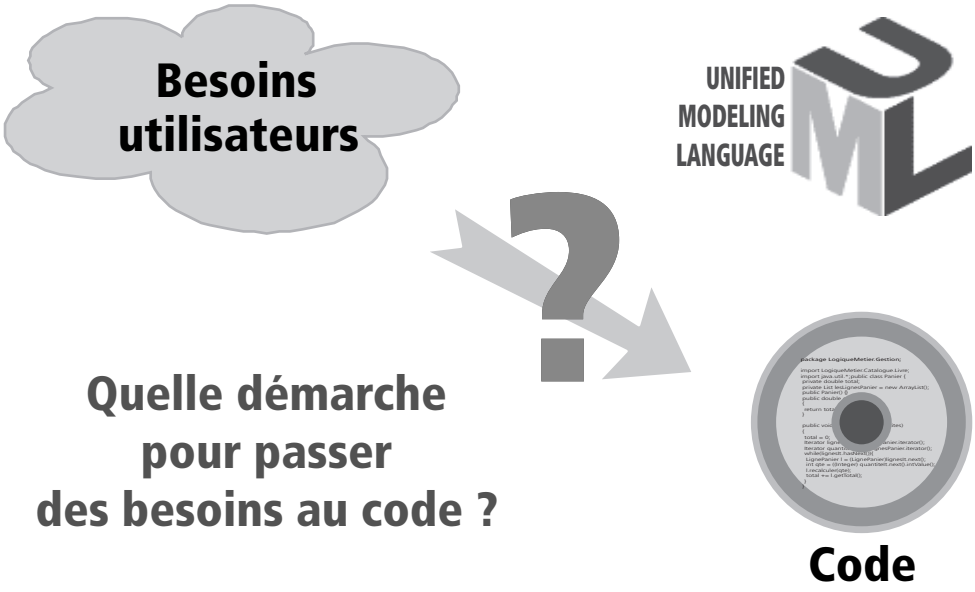
chapitre 1


Besoins utilisateurs
UNIFIED MODELING LANGUAGE
Quelle démarche pour passer des besoins au code ?
Code

Quelle démarche pour passer des besoins utilisateur au code de l'application ?

Dans ce chapitre introductif, nous dévoilons le processus simplifié que nous préconisons pour la modélisation des applications web. Après un premier tour rapide des différents types de diagrammes proposés par le langage de modélisation UML, nous introduirons ceux qui nous seront utiles. Nous présenterons également les principes fondamentaux du Processus Unifié (UP), du développement agile (avec eXtreme Programming et Scrum) et d'Agile Modeling (AM), afin d'éclairer les idées fortes auxquelles se rattache la démarche pratique adoptée dans la suite du livre.

SOMMAIRE

MOTS-CLÉS

- Modélisation
- UML
- Diagrammes
- Processus
- UP
- XP
- Scrum
- Agilité
- Web

Pourquoi modéliser ?

Le recours à la modélisation est depuis longtemps une pratique indispensable au développement logiciel, car un modèle est prévu pour arriver à anticiper les résultats du codage. Un modèle est en effet une représentation abstraite d'un système destiné à en faciliter l'étude et à le documenter. C'est un outil majeur de communication entre les différents intervenants au sein d'un projet. Chaque membre de l'équipe, depuis l'utilisateur jusqu'au développeur, utilise et enrichit le modèle différemment. En outre, les systèmes devenant de plus en plus complexes, leur compréhension et leur maîtrise globale dépassent les capacités d'un seul individu. La construction d'un modèle abstrait aide à y remédier. Le modèle présente notamment l'atout de faciliter la traçabilité du système, à savoir la possibilité de partir d'un de ses éléments et de suivre ses interactions et liens avec d'autres parties du modèle.

Associé au processus de développement, un modèle représente l'ensemble des vues sur une expression de besoins ou sur une solution technique. Pris à un niveau de détail pertinent, il décrit ou conçoit la cible de l'étape en cours. Le modèle sert donc des objectifs différents suivant l'activité de développement et sera construit avec des points de vue de plus en plus détaillés :

- Dans les activités de **spécification des exigences**, il convient premièrement de considérer le système comme une boîte noire à part entière afin d'étudier sa place dans le système métier plus global qu'est l'entreprise. On développe pour cela un modèle de niveau contexte, afin de tracer précisément les frontières fonctionnelles du système.

À RETENIR **Analogie**

Pour illustrer au mieux ce qu'est un modèle, Grady Booch a établi un parallèle entre le développement logiciel et la construction BTP. Cette analogie est judicieuse, car les plans tracés pour construire un immeuble reflètent parfaitement bien l'idée d'anticipation, de conception et de documentation du modèle. Chaque plan développe par ailleurs un point de vue différent suivant les corps de métier. Par exemple, le plan des circuits d'eau et le plan des passages électriques concernent le même immeuble mais sont nécessairement séparés. Enfin, chaque plan se situe à un niveau d'abstraction et de détail distinct suivant l'usage que l'on désire en faire. Ainsi, le plan de masse aide à anticiper les conséquences de l'implantation de l'immeuble sur son environnement, exactement comme le modèle de contexte. Viennent ensuite des plans de construction d'un étage, analogues aux modèles de conception.
Notons cependant que l'anticipation ne permet pas de prendre en compte les besoins changeants des utilisateurs, l'hypothèse de départ étant justement que ces besoins sont définis une bonne fois pour toutes. Or, dans bien des cas, ces besoins évoluent au fil du projet ; c'est pourquoi il est important de gérer le changement et d'admettre la nécessité de continuer à faire vivre nos modèles. Le processus de modélisation du logiciel doit être adaptatif et non pas prédictif, contrairement à ce qui se fait dans le BTP !

- Dans les **activités d'analyse**, le modèle commence à représenter le système vu de l'intérieur. Il se compose d'objets représentant une abstraction des concepts manipulés par les utilisateurs. Le modèle comprend par ailleurs deux points de vue, la structure statique et le comportement dynamique. Il s'agit de deux perspectives différentes qui aident à compléter la compréhension du système à développer.
- Dans les **activités de conception**, le modèle correspond aux concepts informatiques qui sont utilisés par les outils, les langages ou les plates-formes de développement. Le modèle sert ici à étudier, documenter, communiquer et anticiper une solution. Il est en effet toujours plus rentable de découvrir une erreur de conception sur un modèle, que de la découvrir au bout de milliers de lignes codées sans méthode. Pour la conception du déploiement enfin, le modèle représente également les matériels et les logiciels à interconnecter.

Le modèle en tant qu'abstraction d'un système s'accorde parfaitement bien avec les concepts orientés objet. Un objet peut en effet représenter l'abstraction d'une entité métier utilisée en analyse, puis d'un composant de solution logicielle en conception. La correspondance est encore plus flagrante lorsque les langages de développement sont eux-mêmes orientés objet. Cela explique le succès de la modélisation objet ces dernières années pour les projets de plus en plus nombreux utilisant C++, Java ou C#.

À RETENIR Qu'est-ce qu'un « bon » modèle ?

A est un bon modèle de B si A permet de répondre de façon satisfaisante à des questions prédéfinies sur B (d'après D.T. Ross).
Un bon modèle doit donc être construit :

- au bon niveau de détail,
- selon le bon point de vue.

Pensez à l'analogie de la carte routière. Pour circuler dans Toulouse, la carte de France serait de peu d'utilité. En revanche, pour aller de Toulouse à Paris, la carte de la Haute-Garonne ne suffit pas… À chaque voyage correspond la « bonne » carte !

Aujourd'hui, le standard industriel de modélisation objet est UML. Il est sous l'entière responsabilité de l'OMG.

B.A.-BA OMG

L'OMG (*Object Management Group*) est un groupement d'industriels dont l'objectif est de standardiser autour des technologies objet, afin de garantir l'interopérabilité des développements. L'OMG comprend actuellement plus de 800 membres, dont les principaux acteurs de l'industrie informatique (Sun, IBM, etc.), mais aussi les plus grandes entreprises utilisatrices dans tous les secteurs d'activité.

▸ www.omg.org

B.A.-BA Unified Modeling Language

Tous les documents sur UML élaborés dans le cadre de l'OMG sont publics et disponibles sur le site :

▸ www.uml.org.

Les bases d'UML

UML se définit comme un langage de modélisation graphique et textuel destiné à comprendre et décrire des besoins, spécifier et documenter des systèmes, esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vue.

UML unifie à la fois les notations et les concepts orientés objet (voir l'historique d'UML sur la figure 1-1). Il ne s'agit pas d'une simple notation graphique, car les concepts transmis par un diagramme ont une sémantique précise et sont porteurs de sens au même titre que les mots d'un langage.

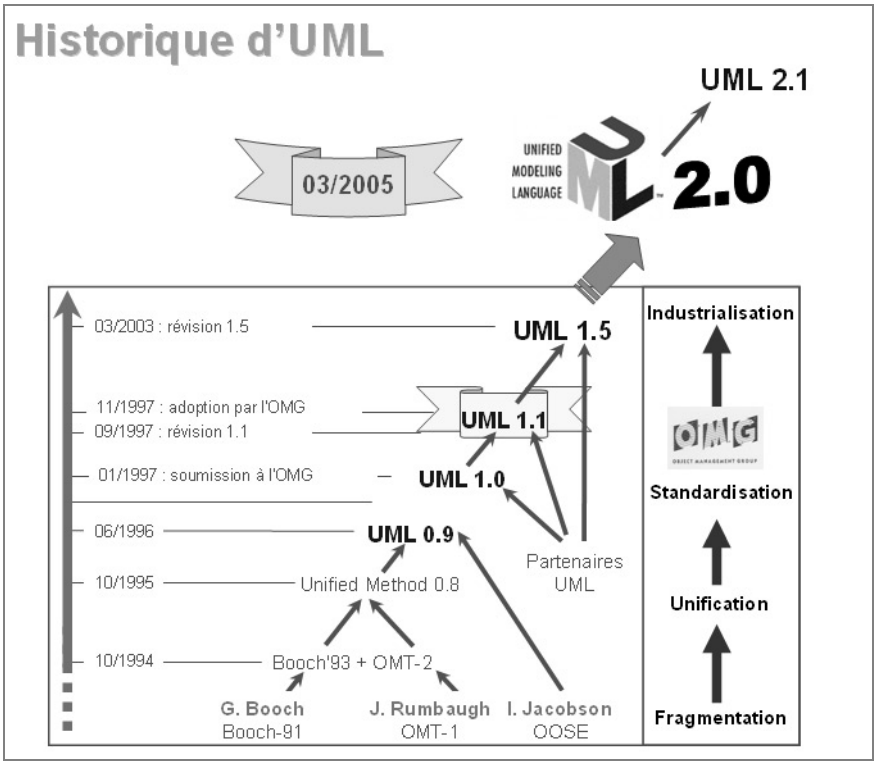


Figure 1–1
Historique d'UML

UML unifie également les notations nécessaires aux différentes activités d'un processus de développement et offre, par ce biais, le moyen d'établir le suivi des décisions prises, depuis l'expression de besoin jusqu'au codage. Dans ce cadre, un concept appartenant aux exigences des utilisateurs projette sa réalité dans le modèle de conception et dans le codage. Le fil tendu entre les différentes étapes de construction permet alors de remonter du code aux besoins et d'en comprendre les tenants et les aboutissants. En d'autres termes, on peut retrouver la nécessité d'un bloc de code en se référant à son origine dans le modèle des besoins.

UML 2 s'articule autour de treize types de diagrammes, chacun d'eux étant dédié à la représentation des concepts particuliers d'un système logiciel. Ces types de diagrammes sont répartis en deux grands groupes :

- **Six diagrammes structurels** :
  - Diagramme de classes – Il montre les briques de base statiques : classes, associations, interfaces, attributs, opérations, généralisations, etc.
  - Diagramme d'objets - Il montre les instances des éléments structurels et leurs liens à l'exécution.
  - Diagramme de packages - Il montre l'organisation logique du modèle et les relations entre packages.
  - Diagramme de structure composite – Il montre l'organisation interne d'un élément statique complexe.
  - Diagramme de composants – Il montre des structures complexes, avec leurs interfaces fournies et requises.
  - Diagramme de déploiement – Il montre le déploiement physique des « artefacts » sur les ressources matérielles.
- **Sept diagrammes comportementaux** :
  - Diagramme de cas d'utilisation - Il montre les interactions fonctionnelles entre les acteurs et le système à l'étude.
  - Diagramme de vue d'ensemble des interactions - Il fusionne les diagrammes d'activité et de séquence pour combiner des fragments d'interaction avec des décisions et des flots.
  - Diagramme de séquence - Il montre la séquence verticale des messages passés entre objets au sein d'une interaction.
  - Diagramme de communication - Il montre la communication entre objets dans le plan au sein d'une interaction.
  - Diagramme de temps – Il fusionne les diagrammes d'états et de séquence pour montrer l'évolution de l'état d'un objet au cours du temps.
  - Diagramme d'activité - Il montre l'enchaînement des actions et décisions au sein d'une activité.
  - Diagramme d'états – Il montre les différents états et transitions possibles des objets d'une classe.

Le diagramme de cas d'utilisation (figure 1-2) est utilisé dans l'activité de spécification des besoins. Il montre les interactions fonctionnelles entre les acteurs et le système à l'étude. Vous trouverez une description détaillée de son usage au chapitre 3 de cet ouvrage.

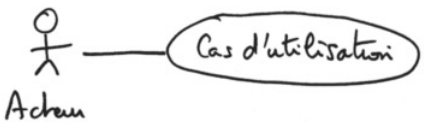
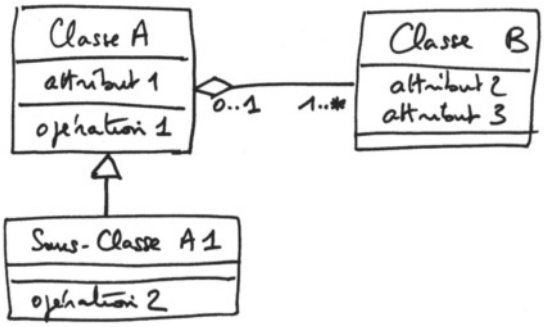


Figure 1–2
Diagramme de cas d'utilisation

Le diagramme de classes (figure 1-3) est le point central dans un développement orienté objet. En analyse, il a pour objet de décrire la struc-

ture des entités manipulées par les utilisateurs. Vous trouverez les explications relatives à cette utilisation au chapitre 5. En conception, le diagramme de classes représente la structure d'un code orienté objet. Vous retrouverez l'utilisation du diagramme de classes en conception aux chapitres 7 et 8.

Figure 1–3
Diagramme de classes



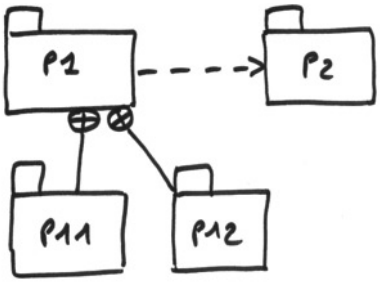


Figure 1–4
Diagramme de packages

Le diagramme de packages (figure 1-4) montre l'organisation logique du modèle et les relations entre packages. Il permet de structurer les classes d'analyse et de conception, mais aussi les cas d'utilisation. Vous verrez ces deux utilisations du diagramme de packages aux chapitres 3 et 8.

Les diagrammes de séquence (figure 1-5) et les diagrammes de communication (figure 1-6) sont tous deux des diagrammes d'interactions UML. Ils représentent des échanges de messages entre éléments, dans le cadre d'un fonctionnement particulier du système. Les diagrammes de séquence servent d'abord à développer en analyse les scénarios d'utilisation du système. Vous en trouverez des exemples au chapitre 4. Plus tard, les diagrammes de séquence et de communication permettent de concevoir les méthodes des classes comme indiqué aux chapitres 7 et 8. Nous privilégierons cependant nettement les diagrammes de séquence pour restreindre le nombre de diagrammes utilisés.

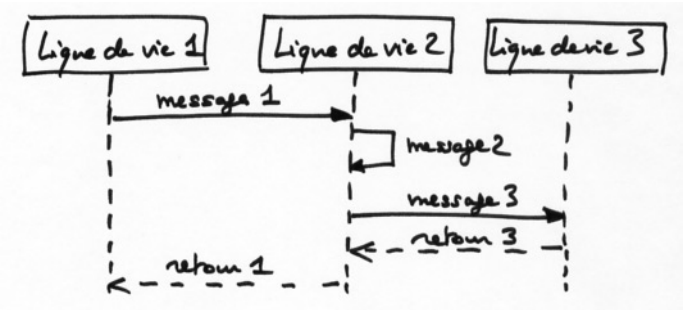


Figure 1–5
Diagramme de séquence

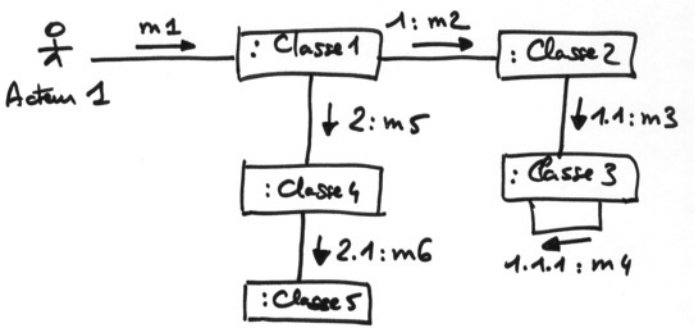


Figure 1–6
Diagramme de communication

Le diagramme d'états (figure 1-7) représente le cycle de vie commun aux objets d'une même classe. Ce diagramme complète la connaissance des classes en analyse et en conception en montrant les différents états et transitions possibles des objets d'une classe à l'exécution. Le chapitre 5 vous indiquera comment utiliser ce diagramme à des fins d'analyse.

Vous en verrez une utilisation particulière au chapitre 6 pour modéliser la navigation dans le site web.

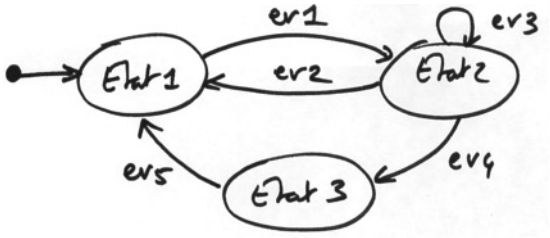


Figure 1–7
Diagramme d'états

Le diagramme d'activité (figure 1-8) représente les règles d'enchaînement des actions et décisions au sein d'une activité. Il peut également être utilisé comme alternative au diagramme d'états pour décrire la navigation dans un site web, comme illustré au chapitre 6.

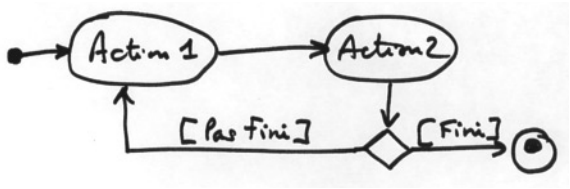


Figure 1–8
Diagramme d'activité

Le diagramme d'objets (figure 1-9) est un instantané, une photo d'un sous-ensemble des objets d'un système à un certain moment du temps. C'est probablement le diagramme le moins utilisé d'UML et nous n'en verrons pas d'illustration.

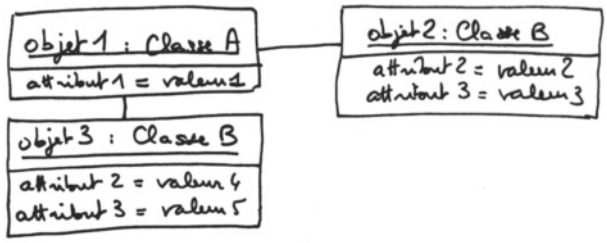


Figure 1–9
Diagramme d'objets

Le diagramme de composants (figure 1-10) montre les unités logicielles à partir desquelles on a construit le système informatique, ainsi que leurs dépendances.

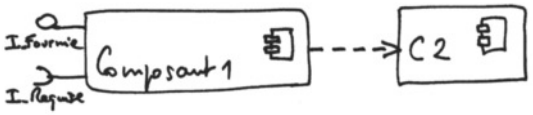


Figure 1–10
Diagramme de composants

Le diagramme de déploiement (figure 1-11) montre le déploiement physique des artefacts (éléments concrets tels que fichiers, exécutables, etc.) sur les ressources matérielles.

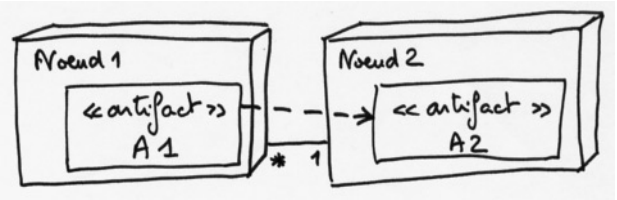


Figure 1–11
Diagramme de déploiement

Le diagramme de vue d'ensemble des interactions fusionne les diagrammes d'activité et de séquence pour combiner des fragments d'interaction avec des décisions et des flots.

Le diagramme de temps fusionne les diagrammes d'états et de séquence pour montrer l'évolution de l'état d'un objet au cours du temps et les messages qui modifient cet état.

Le diagramme de structure composite montre l'organisation interne d'un élément statique complexe sous forme d'un assemblage de parties, de connecteurs et de ports.

Dans un souci de simplicité, nous n'utiliserons pas ces trois nouveaux types de diagrammes proposés par UML 2.

L'ensemble des treize types de diagrammes UML peut ainsi être résumé sur la figure 1-12, en mettant en évidence les cinq diagrammes que nous utiliserons prioritairement.

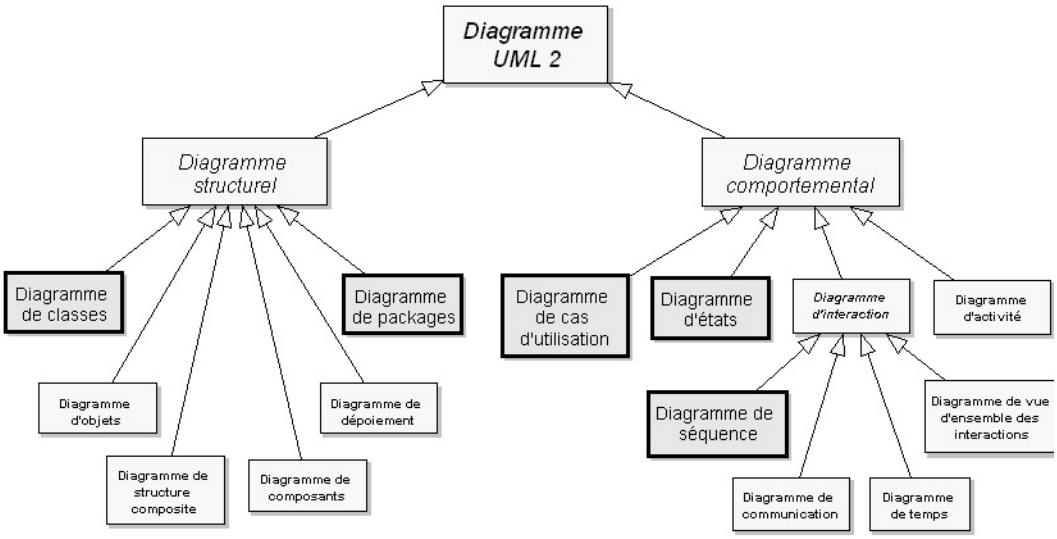


Figure 1–12 Les diagrammes UML utilisés dans notre démarche agile

Un processus simplifié pour les applications web

Le processus que nous vous proposons de suivre pour le développement d'applications web se situe à mi-chemin entre UP (Unified Process), un cadre général très complet de processus de développement, et les méthodes agiles en vogue actuellement, telles que XP (eXtreme Programming), et Scrum. Il s'inspire également des bonnes pratiques prônées par les tenants de la modélisation agile (Agile Modeling).

Les principes fondamentaux du Processus Unifié (UP)

Le Processus Unifié (UP, pour Unified Process) est un processus de développement logiciel « itératif et incrémental, centré sur l'architecture, conduit par les cas d'utilisation et piloté par les risques » :

- **Itératif et incrémental** : le projet est découpé en itérations de courte durée (environ 1 mois) qui aident à mieux suivre l'avancement global. À la fin de chaque itération, une partie exécutable du système final est produite, de façon incrémentale.

B.A.-BA **Processus de développement**

Un processus définit une séquence d'étapes, partiellement ordonnées, qui concourent à l'obtention d'un système logiciel ou à l'évolution d'un système existant. L'objet d'un processus de développement est de produire des logiciels de qualité qui répondent aux besoins de leurs utilisateurs dans des temps et des coûts prévisibles.
Plus simplement, un processus doit permettre de répondre à la question fondamentale : « Qui fait quoi et quand ? ».

- **Centré sur l'architecture** : tout système complexe doit être décomposé en parties modulaires afin de garantir une maintenance et une évolution facilitées. Cette architecture (fonctionnelle, logique, matérielle, etc.) doit être modélisée en UML et pas seulement documentée en texte.
- **Piloté par les risques** : les risques majeurs du projet doivent être identifiés au plus tôt, mais surtout levés le plus rapidement possible. Les mesures à prendre dans ce cadre déterminent l'ordre des itérations.
- **Conduit par les cas d'utilisation** : le projet est mené en tenant compte des besoins et des exigences des utilisateurs. Les cas d'utilisation du futur système sont identifiés, décrits avec précision et priorisés.

Les phases et les disciplines de UP

La gestion d'un tel processus est organisée suivant les quatre phases suivantes : initialisation, élaboration, construction et transition.

La phase d'initialisation conduit à définir la « vision » du projet, sa portée, sa faisabilité, son business case, afin de pouvoir décider au mieux de sa poursuite ou de son arrêt.

La phase d'élaboration poursuit trois objectifs principaux en parallèle :

- identifier et décrire la majeure partie des besoins des utilisateurs,
- construire (et pas seulement décrire dans un document !) l'architecture de base du système,
- lever les risques majeurs du projet.

La phase de construction consiste surtout à concevoir et implémenter l'ensemble des éléments opérationnels (autres que ceux de l'architecture de base). C'est la phase la plus consommatrice en ressources et en effort.

Enfin, la phase de transition permet de faire passer le système informatique des mains des développeurs à celles des utilisateurs finaux. Les mots-clés sont : conversion des données, formation des utilisateurs, déploiement, béta-tests.

Chaque phase est elle-même décomposée séquentiellement en itérations limitées dans le temps (entre 2 et 4 semaines). Le résultat de chacune d'elles est un système testé, intégré et exécutable. L'approche itérative est fondée sur la croissance et l'affinement successifs d'un système par le biais d'itérations multiples, feedback et adaptation cycliques étant les moteurs principaux permettant de converger vers un système satisfaisant. Le système croît avec le temps de façon incrémentale, itération par itération, et c'est pourquoi cette méthode porte également le nom de développement itératif et incrémental. Il s'agit là du principe le plus important du Processus Unifié.

Les activités de développement sont définies par cinq disciplines fondamentales qui décrivent la capture des exigences, l'analyse et la conception, l'implémentation, le test et le déploiement. La modélisation métier est une discipline amont optionnelle et transverse aux projets. Enfin, trois disciplines appelées de support complètent le tableau : gestion de projet, gestion du changement et de la configuration, ainsi que la mise à disposition d'un environnement complet de développement incluant aussi bien des outils informatiques que des documents et des guides méthodologiques.

UP doit donc être compris comme une trame commune des meilleures pratiques de développement, et non comme l'ultime tentative d'élaborer un processus universel.

Le schéma synthétique du RUP™ (Rational Unified Process)

Contrairement au processus en cascade (souvent appelé cycle en V, en France), le Processus Unifié ne considère pas que les disciplines sont purement séquentielles. En fait, une itération comporte une certaine quantité de travail dans la plupart des disciplines. Cependant, la répartition de l'effort relatif entre celles-ci change avec le temps. Les premières itérations ont tendance à mettre plus l'accent sur les exigences et la conception, les autres moins, à mesure que les besoins et l'architecture se stabilisent grâce au processus de feedback et d'adaptation.

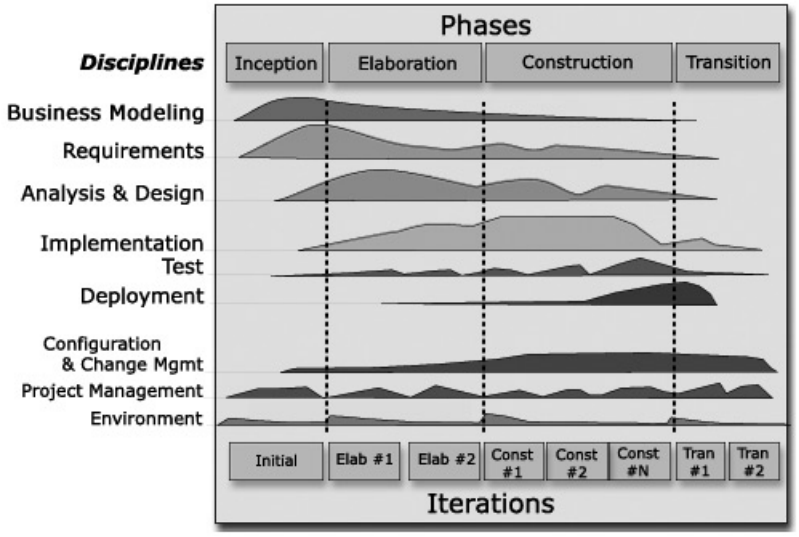


Figure 1–13
Les deux dimensions du Processus Unifié d'après RUP™

B.A.-BA OpenUP

OpenUP est une initiative intéressante pour simplifier le RUP et en proposer une version libre en tant que partie du framework EPF (*Eclipse Process Framework*) :

- http://epf.eclipse.org/wikis/openup/

Les principes du Manifeste Agile

▸ www.agilemanifesto.org

La notion de méthode agile est née à travers un manifeste signé en 2001 par 17 personnalités du développement logiciel (parmi lesquelles Ward Cunningham, Alistair Cockburn, Kent Beck, Martin Fowler, Ron Jeffries, Steve Mellor, Robert C. Martin, Ken Schwaber, Jeff Sutherland, etc.).

Ce manifeste prône quatre valeurs fondamentales :

- « Personnes et interactions plutôt que processus et outils » : dans l'optique agile, l'équipe est bien plus importante que les moyens matériels ou les procédures. Il est préférable d'avoir une équipe soudée et qui communique, composée de développeurs moyens, plutôt qu'une équipe composée d'individualistes, même brillants. La communication est une notion fondamentale.
- « Logiciel fonctionnel plutôt que documentation complète » : il est vital que l'application fonctionne. Le reste, et notamment la documentation technique, est secondaire, même si une documentation succincte et précise est utile comme moyen de communication. La documentation représente une charge de travail importante et peut être néfaste si elle n'est pas à jour. Il est préférable de commenter abondamment le code lui-même, et surtout de transférer les compétences au sein de l'équipe (on en revient à l'importance de la communication).
- « Collaboration avec le client plutôt que négociation de contrat » : le client doit être impliqué dans le développement. On ne peut se contenter de négocier un contrat au début du projet, puis de négliger les demandes du client. Le client doit collaborer avec l'équipe et fournir un feedback continu sur l'adaptation du logiciel à ses attentes.
- « Réagir au changement plutôt que suivre un plan » : la planification initiale et la structure du logiciel doivent être flexibles afin de permettre l'évolution de la demande du client tout au long du projet. Les premières *releases* du logiciel vont souvent provoquer des demandes d'évolution.

Les pratiques d'eXtreme Programming (XP)

L'eXtreme Programming (XP) est un ensemble de pratiques qui couvre une grande partie des activités de la réalisation d'un logiciel, de la programmation proprement dite à la planification du projet, en passant par l'organisation de l'équipe de développement et les échanges avec le client. Ces pratiques ne sont pas révolutionnaires : il s'agit simplement de pratiques de bon sens mises en œuvre par des développeurs ou des chefs de projet expérimentés, telles que :

- Un utilisateur à plein-temps dans la salle projet. Ceci permet une communication intensive et permanente entre les clients et les développeurs, aussi bien pour l'expression des besoins que pour la validation des livraisons.

- Écrire le test unitaire avant le code qu'il doit tester, afin d'être certain que le test sera systématiquement écrit et non pas négligé.
- Programmer en binôme, afin d'homogénéiser la connaissance du système au sein des développeurs, et de permettre aux débutants d'apprendre auprès des experts. Le code devient ainsi une propriété collective et non individuelle, que tous les développeurs ont le droit de modifier.
- Intégrer de façon continue, pour ne pas repousser à la fin du projet le risque majeur de l'intégration des modules logiciels écrits par des équipes ou des personnes différentes. Etc.

Pour résumer, on peut dire que XP est une méthodologie légère qui met l'accent sur l'activité de programmation et qui s'appuie sur les valeurs suivantes : communication, simplicité et feedback. Elle est bien adaptée pour des projets de taille moyenne où le contexte (besoins des utilisateurs, technologies informatiques) évolue en permanence.

- 🕮 *Gestion de projet Extreme Programming*, J.L. Bénard *et al.*, Eyrolles, 2002.
- 🕮 *Maîtriser les projets avec l'Extreme Programming – Pilotage par les tests-client*, T. Cros, Cépaduès, 2004
- ▸ http://fr.wikipedia.org/wiki/Extreme_programming
- ▸ http://etreagile.thierrycros.net/

Les bases de Scrum

Scrum est issu des travaux de deux des signataires du Manifeste Agile, Ken Schwaber et Jeff Sutherland, au début des années 1990. Le terme *Scrum* est emprunté au rugby et signifie mêlée. Ce processus agile s'articule en effet autour d'une équipe soudée, qui cherche à atteindre un but, comme c'est le cas en rugby pour avancer avec le ballon pendant une mêlée.

Le principe de base de Scrum est de focaliser l'équipe de façon itérative sur un ensemble de fonctionnalités à réaliser, dans des itérations de 30 jours, appelées *Sprints*. Chaque Sprint possède un but à atteindre, défini par le directeur de produit (*Product owner*), à partir duquel sont choisies les fonctionnalités à implémenter dans ce Sprint. Un Sprint aboutit toujours sur la livraison d'un produit partiel fonctionnel. Pendant ce temps, le *scrummaster* a la charge de réduire au maximum les perturbations extérieures et de résoudre les problèmes non techniques de l'équipe.

Un principe fort en Scrum est la participation active du client pour définir les priorités dans les fonctionnalités du logiciel, et choisir lesquelles seront réalisées dans chaque Sprint. Il peut à tout moment ajouter ou modifier la liste des fonctionnalités à réaliser, mais jamais ce qui est en cours de réalisation pendant un Sprint.

- ▸ http://fr.wikipedia.org/wiki/Scrum
- ▸ http://scrum.aubryconseil.com/

La modélisation agile (AM)

La « modélisation agile » prônée par Scott Ambler s'appuie sur des principes simples et de bon sens, parmi lesquels :

- Vous devriez avoir une grande palette de techniques à votre disposition et connaître les forces et les faiblesses de chacune de manière à pouvoir appliquer la meilleure au problème courant.

- N'hésitez pas à changer de diagramme quand vous sentez que vous n'avancez plus avec le modèle en cours. Le changement de perspective va vous permettre de voir le problème sous un autre angle et de mieux comprendre ce qui bloquait précédemment.
- Vous trouverez souvent que vous êtes plus productif si vous créez plusieurs modèles simultanément plutôt qu'en vous focalisant sur un seul type de diagramme.

▸ http://www.agile-modeling.com/

Le processus proposé dans cet ouvrage

Le processus que nous allons présenter et appliquer tout au long de ce livre est :

- conduit par les cas d'utilisation, comme UP, mais beaucoup plus simple ;
- relativement léger et restreint, comme les méthodes agiles, mais sans négliger les activités de modélisation en analyse et conception ;
- fondé sur l'utilisation d'un sous-ensemble nécessaire et suffisant du langage UML, conformément à AM.

Nous allons donc essayer de trouver le meilleur rapport « qualité/prix » possible afin de ne pas surcharger le lecteur de concepts et d'activités de modélisation qui ne sont pas indispensables au développement d'applications web efficaces. En revanche, nous nous efforcerons bien de montrer qu'il est important et utile de modéliser précisément certains aspects critiques de son système.

Le problème fondamental auquel ce livre va s'efforcer de répondre est finalement assez simple : comment passer des besoins des utilisateurs au code de l'application ? Autrement dit : « J'ai une bonne idée de ce que mon application doit faire, des fonctionnalités attendues par les futurs utilisateurs. Comment obtenir le plus efficacement possible un code informatique opérationnel, complet, testé, et qui réponde parfaitement au besoin ? ».

À RETENIR **Règle des 80/20**

La célèbre règle des 80/20 peut aussi s'appliquer dans notre cas : vous pouvez modéliser 80 % de vos problèmes en utilisant 20 % d'UML ! Encore faut-il savoir quels sont ces 20 % indispensables… Nous espérons que vous aurez une réponse claire et précise à cette question cruciale à l'issue de la lecture de cet ouvrage.

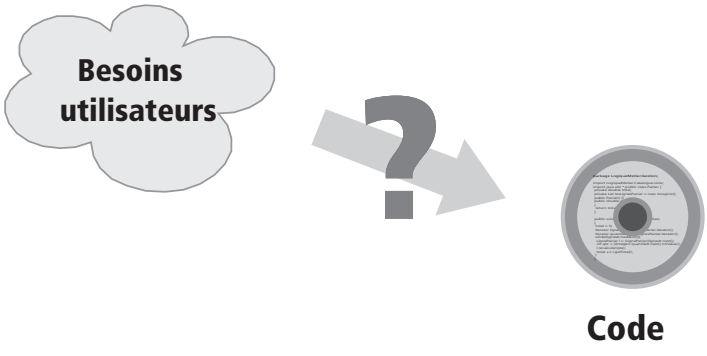


Figure 1–14
Comment passer des besoins au code ?

Il ne s'agit pas de se jeter sur l'écriture de code en omettant de formaliser les besoins des utilisateurs et d'élaborer une architecture robuste et évolutive. D'un autre côté, le but n'est pas de faire de la modélisation pour le plaisir, mais bien de produire le plus rapidement possible une application qui satisfasse au mieux ses utilisateurs !

Nous allons donc vous proposer une démarche de modélisation nécessaire et suffisante afin de construire efficacement une application web. Pour cela, nous utiliserons un sous-ensemble du langage de modélisation UML qui sera également nécessaire et suffisant pour la plupart des projets de même nature. Cette approche est le résultat de plusieurs années d'expérience dans des domaines variés. Elle a donc montré son efficacité dans la pratique.

Dans un premier temps, les besoins vont être modélisés au moyen des cas d'utilisation UML. Ils seront représentés de façon plus concrète par une maquette d'IHM (Interface Homme-Machine) destinée à faire réagir les futurs utilisateurs. La figure 1-15 montre bien de quoi nous partons et là où nous voulons arriver.

▸ La technique des cas d'utilisation sera expliquée au chapitre 3.

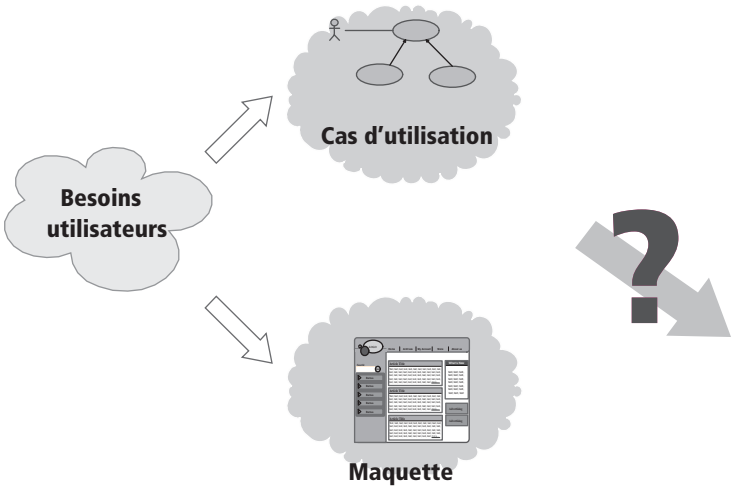


Figure 1–15
Les besoins donnent lieu à des cas d'utilisation et à une maquette

Repartons maintenant du but, c'est-à-dire du code que nous voulons obtenir, et remontons en arrière, pour mieux expliquer le chemin minimal qui va nous permettre de joindre les deux « bouts ». Dans le cadre de systèmes orientés objet, la structure du code est définie par les classes logicielles et leurs regroupements en ensembles appelés *packages* (paquetages en français). Nous avons donc besoin de diagrammes représentant les classes logicielles et montrant les données qu'elles contiennent (appelées attributs), les services qu'elles rendent (appelés opérations) ainsi que leurs relations. UML propose les diagrammes de classes pour véhiculer toutes ces informations.

Nous appellerons ces diagrammes « diagrammes de classes de conception » pour indiquer qu'ils sont à un niveau de détail suffisant pour en dériver automatiquement ou manuellement le code de l'application. Ils seront présentés aux chapitres 7 et 8.

B.A.-BA **Maquette**

Une maquette est un produit jetable donnant aux utilisateurs une vue concrète mais non définitive de la future interface de l'application. Cela peut consister en un ensemble de dessins réalisés avec des outils spécialisés tels que Dreamweaver, Adobe Illustrator ou plus simplement avec Powerpoint ou même Word. Par la suite, la maquette intégrera des fonctionnalités de navigation pour que l'utilisateur puisse tester l'enchaînement des écrans, même si les fonctionnalités restent fictives.
La maquette est développée rapidement afin de provoquer des retours de la part des utilisateurs. Elle permet ainsi d'améliorer la relation développeur-client. La plupart du temps, la maquette est considérée comme jetable, c'est-à-dire que la technologie informatique employée pour la réaliser n'est pas forcément assez robuste et évolutive pour être intégrée telle quelle. Pensez à l'analogie de la maquette d'avion qui est très utile en soufflerie, mais qui ne peut pas voler !

La figure 1-16 montre ainsi cette étape préliminaire au codage, mais il reste encore beaucoup de chemin à parcourir…

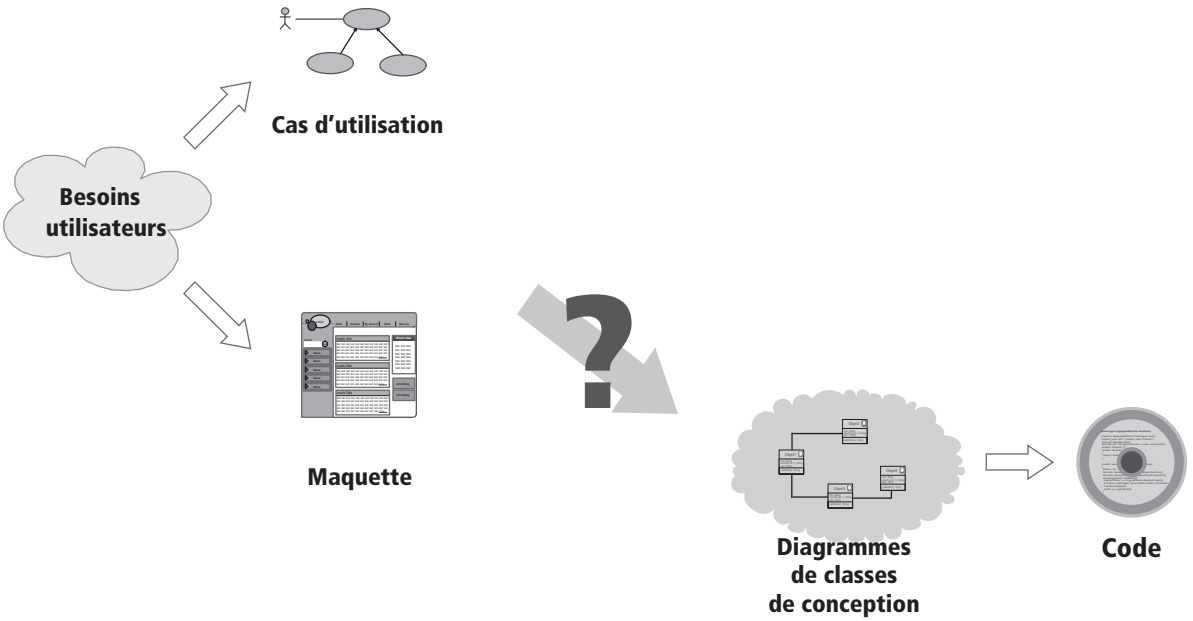


Figure 1–16 Les diagrammes de classes de conception donnent la structure du code.

Les diagrammes de classes de conception représentent bien la structure statique du code, par le biais des attributs et des relations entre classes, mais ils contiennent également les opérations (aussi appelées méthodes) qui décrivent les responsabilités dynamiques des classes logicielles. L'attribution des bonnes responsabilités aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet. Pour chaque service ou fonction, il faut décider quelle est la classe qui va le/la con-

tenir. Nous devons ainsi répartir tout le comportement du système entre les classes de conception, et décrire les interactions induites.

Les diagrammes d'interactions UML (séquence ou communication) sont particulièrement utiles au concepteur pour représenter graphiquement ses décisions d'allocation de responsabilités. Chaque diagramme d'interaction va ainsi représenter un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système. Dans ce genre de diagramme, les objets communiquent en s'envoyant des messages qui invoquent des opérations sur les objets récepteurs.

On peut donc suivre visuellement les interactions dynamiques entre objets, et les traitements réalisés par chacun. Les diagrammes d'interaction aident également à écrire le code à l'intérieur des opérations, en particulier les appels d'opérations imbriqués.

La figure 1-17 ajoute une étape du côté du code, mais ne nous dit pas encore comment relier tout cela aux cas d'utilisation.

MÉTHODE Allocation des responsabilités

Avec un outil de modélisation UML, à chaque fois que vous déclarez un message entre deux objets, vous pouvez créer effectivement une opération publique sur la classe de l'objet récepteur. Ce type d'outil permet vraiment de mettre en œuvre l'allocation des responsabilités à partir des diagrammes d'interaction.

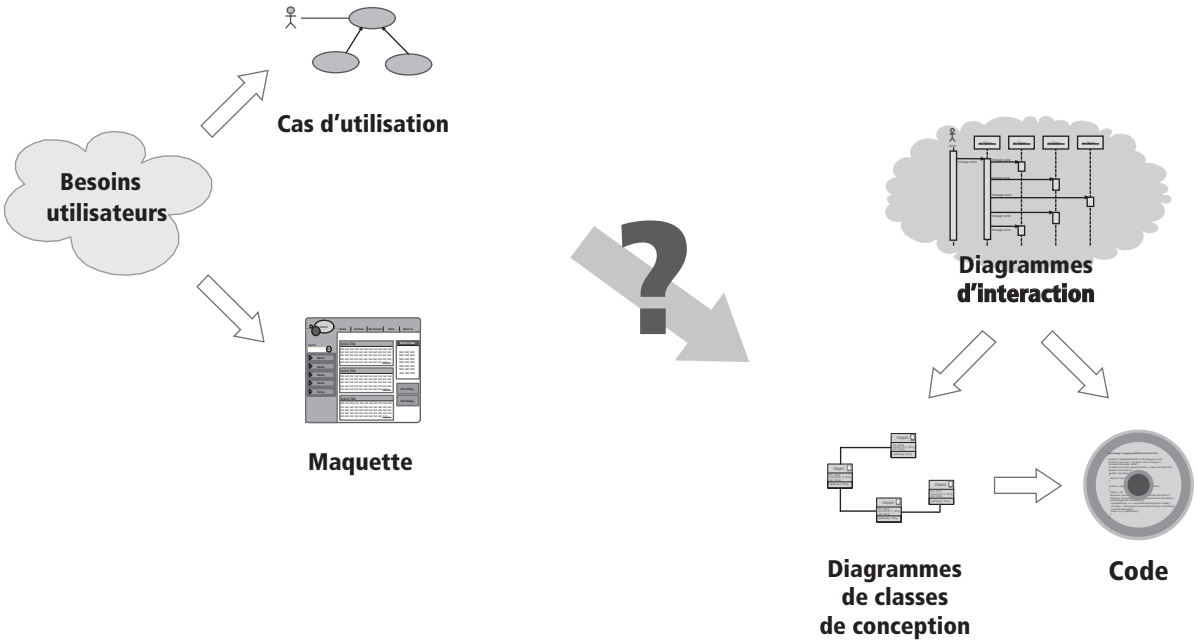


Figure 1–17 Les diagrammes d'interaction nous aident à attribuer les responsabilités aux classes.

Comment passer des cas d'utilisation aux diagrammes d'interaction ? Ce n'est ni simple ni direct, car les cas d'utilisation sont au niveau d'abstraction des besoins des utilisateurs alors que les diagrammes d'interaction se placent au niveau de la conception objet. Il faut donc au moins une étape intermédiaire.

▸ Les diagrammes d'interaction seront utilisés intensivement aux chapitres 7 et 8.

> La description textuelle détaillée des cas d'utilisation ainsi que le diagramme de séquence système seront présentés au chapitre 4.

Chaque cas d'utilisation est décrit textuellement de façon détaillée, mais donne également lieu à un diagramme de séquence simple représentant graphiquement la chronologie des interactions entre les acteurs et le système vu comme une boîte noire, dans le cadre du scénario nominal. Nous appellerons ce diagramme : « diagramme de séquence système ».

Par la suite, en remplaçant le système vu comme une boîte noire par un ensemble choisi d'objets de conception, nous décrirons l'attribution des responsabilités dynamiques, tout en conservant une traçabilité forte avec les cas d'utilisation. La figure 1-18 montre ainsi les diagrammes de séquence système en tant que liens importants entre les cas d'utilisation et les diagrammes d'interaction.

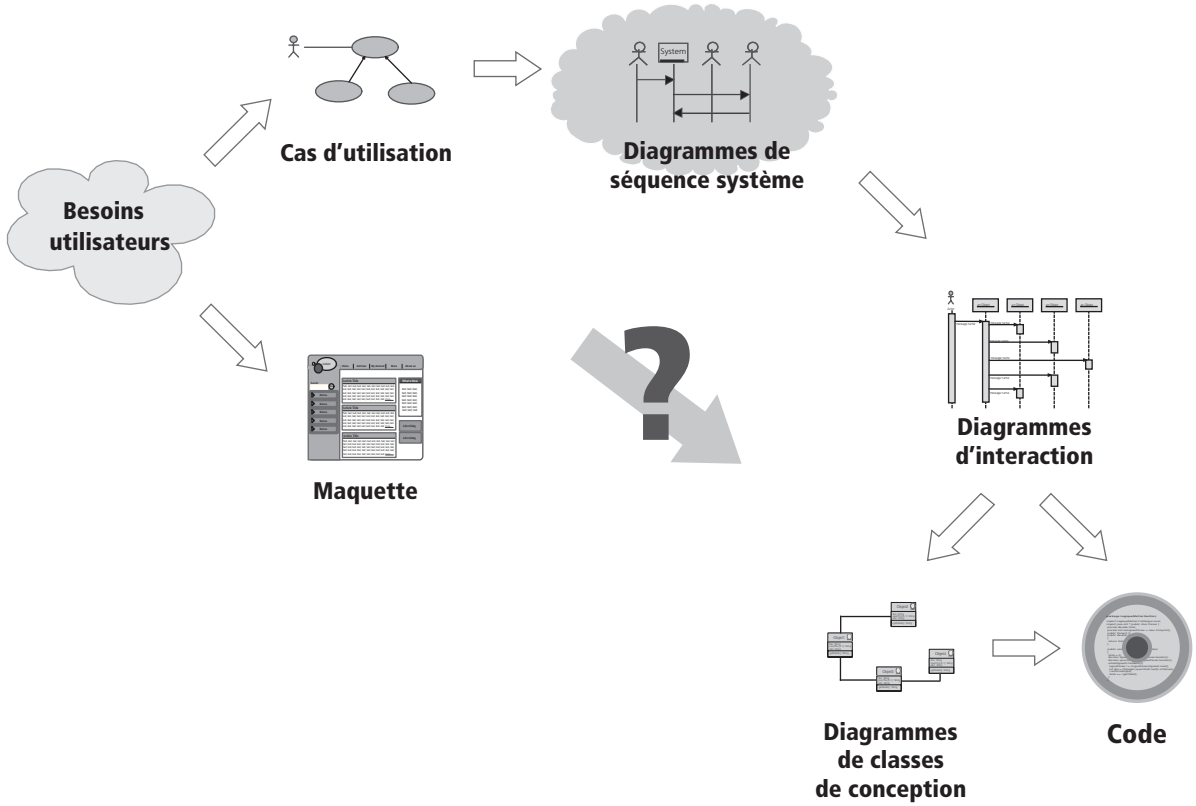


Figure 1–18 Les diagrammes de séquence système fournissent le squelette des diagrammes d'interaction.

Maintenant, comment trouver ces fameuses classes de conception qui interviennent dans les diagrammes d'interaction ?

Le chaînon manquant de notre démarche s'appelle les diagrammes de classes participantes. Il s'agit là de diagrammes de classes UML qui décrivent, cas d'utilisation par cas d'utilisation, les trois principales

classes d'analyse et leurs relations : les dialogues, les contrôles et les entités. Ces classes d'analyse, leurs attributs et leurs relations vont être décrits en UML par un diagramme de classes simplifié utilisant des conventions particulières.

Un avantage important de cette technique pour le chef de projet consiste en la possibilité de découper le travail de son équipe d'analystes suivant les différents cas d'utilisation, plutôt que de vouloir tout traiter d'un bloc. Comme l'illustre la figure 1-19, les diagrammes de classes participantes sont particulièrement importants car ils font la jonction entre les cas d'utilisation, la maquette et les diagrammes de conception logicielle (diagrammes d'interaction et diagrammes de classes).

▸ Les diagrammes de classes participantes seront détaillés au chapitre 5.

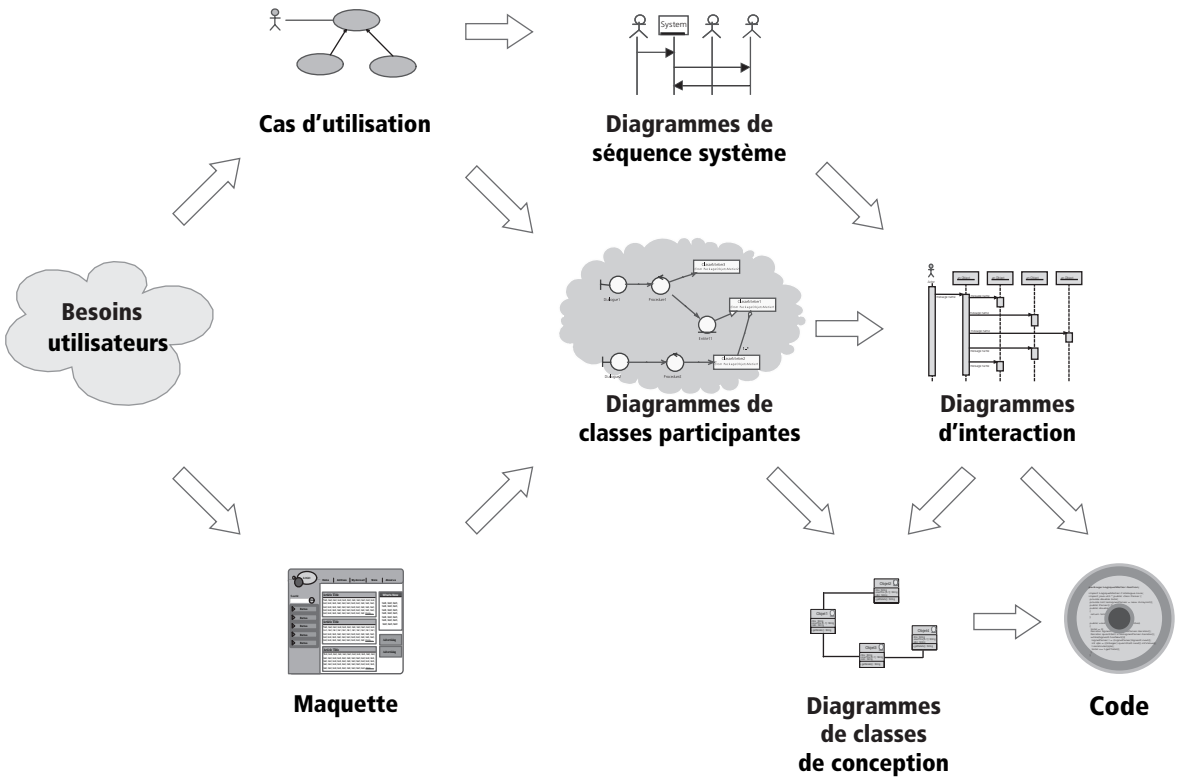


Figure 1–19 Les diagrammes de classes participantes font la jonction entre les cas d'utilisation, la maquette et les diagrammes de conception logicielle.

Pour que le tableau soit complet, il nous reste à détailler une exploitation supplémentaire de la maquette. Elle va nous permettre de réaliser des diagrammes dynamiques représentant de manière formelle l'ensemble des chemins possibles entre les principaux écrans proposés à l'utilisateur. Ces diagrammes, qui mettent en jeu les classes participantes de type « dialogues » et « contrôles », s'appellent des diagrammes de navigation.

▸ Les diagrammes de navigation seront présentés au chapitre 6.

B.A.-BA **Dialogues, contrôles, entités**

Les classes qui permettent les interactions entre le site web et ses utilisateurs sont qualifiées de « dialogues ». C'est typiquement les écrans proposés à l'utilisateur : les formulaires de saisie, les résultats de recherche, etc. Elles proviennent directement de l'analyse de la maquette.
Celles qui contiennent la cinématique de l'application seront appelées « contrôles ». Elles font la transition entre les dialogues et les classes métier, en permettant aux écrans de manipuler des informations détenues par un ou plusieurs objet(s) métier.
Celles qui représentent les règles métier sont qualifiées d'« entités ». Elles proviennent directement du modèle du domaine, mais sont confirmées et complétées cas d'utilisation par cas d'utilisation.

La trame globale de la démarche est ainsi finalisée, comme indiqué sur la figure 1-20.

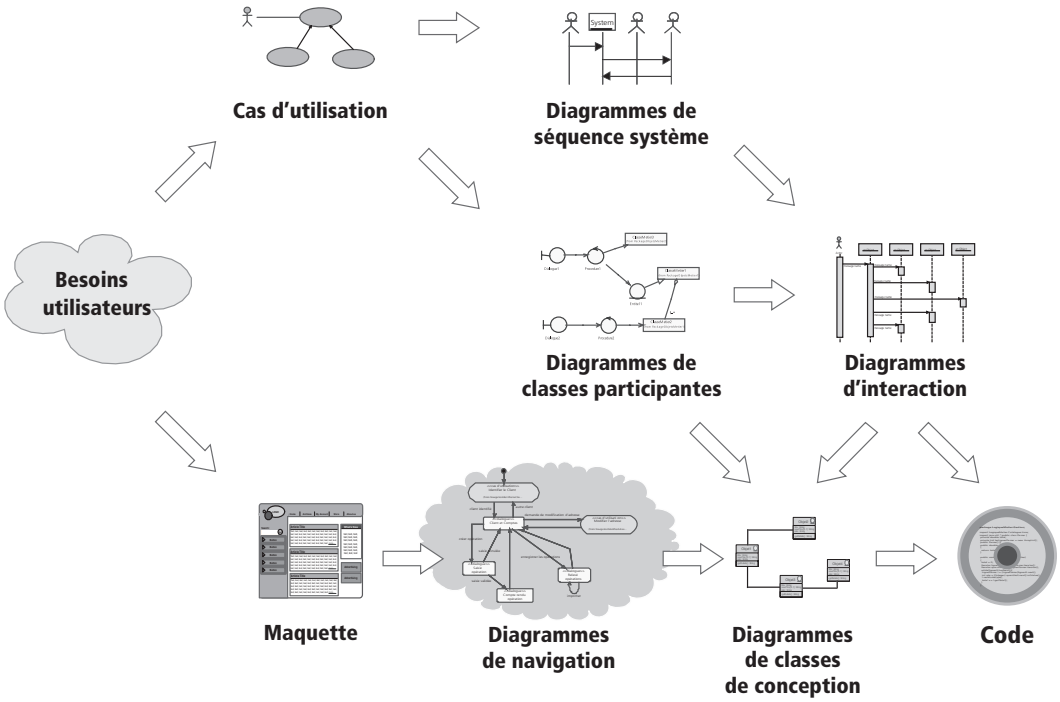


Figure 1–20 Schéma complet du processus de modélisation d'une application web

Méthode **Quelques types de diagrammes seulement**

Nous n'utilisons pas les treize types de diagrammes proposés par UML 2, mais seulement un tiers, en insistant particulièrement sur les diagrammes de classes et les diagrammes de séquence. Cette limitation volontaire permet une réduction significative du temps d'apprentissage de la modélisation avec UML, tout en restant largement suffisante pour la plupart des projets.
C'est cela aussi la « modélisation agile » !

Nous avons obtenu un schéma complet montrant comment, en partant des besoins des utilisateurs formalisés par des cas d'utilisation et une maquette, et avec l'apport des diagrammes de classes participantes, on peut aboutir à des diagrammes de conception à partir desquels on dérivera du code assez directement.

Organisation du livre

Après ce premier chapitre introductif, qui nous a permis de poser les bases de la démarche, nous allons détailler une par une, dans la suite de ce livre, les étapes permettant d'aboutir à l'ensemble des livrables de la figure précédente.

La figure 1-21 replace ainsi chacun des chapitres suivants dans le cadre de la démarche de modélisation.

Ce schéma général nous servira d'introduction pour chaque chapitre du livre, afin de bien replacer l'étape courante dans la démarche globale de modélisation d'une application web.

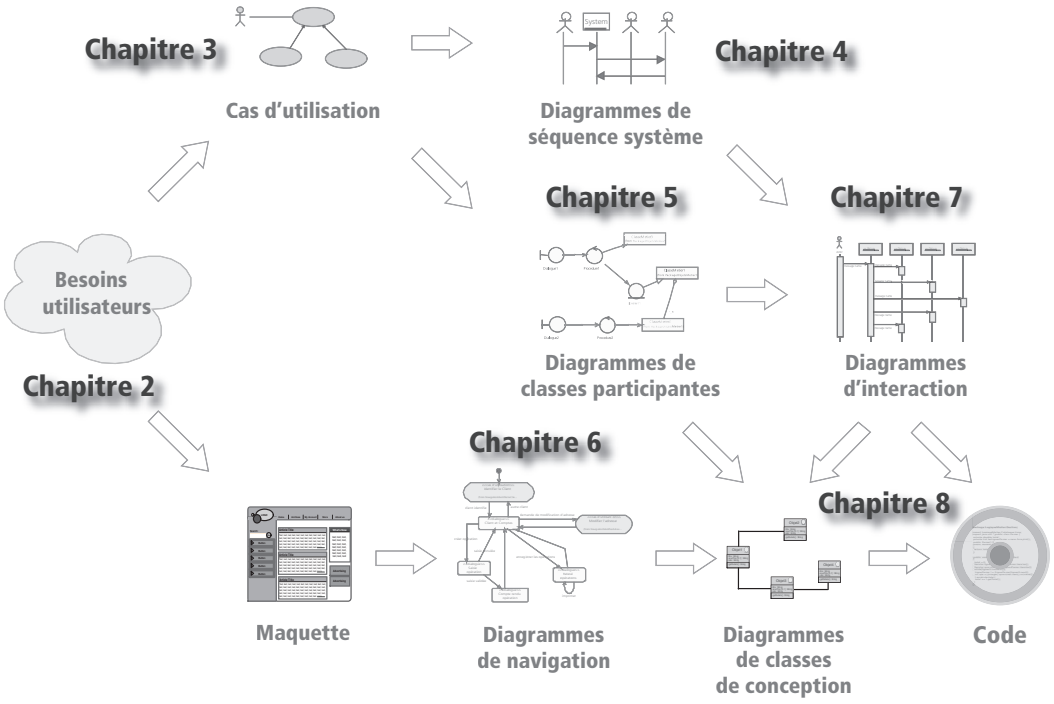


Figure 1–21 Les chapitres du livre replacés dans la démarche globale de modélisation