Chapitre 03 : Ingénierie des systèmes embarqués à base d'UML

Analyse des exigences et spécification

1. Types d'exigences

Il existe différentes manières de regrouper les exigences (besoins) liées aux systèmes informatiques. Il y a ceux qui distinguent les exigences fonctionnelles et non fonctionnelles. Mais il y a aussi une classification plus précise proposée par le standard IEEE 830 (en 1998) qui distingue les exigences suivantes :

- **Exigences fonctionnelles :** que doit faire le logiciel ? Quelles sont les fonctions (services) du logiciel ?

- **Exigences d'interfaces externes :** comment le logiciel interagit-il avec les individus, avec l'OS, avec d'autres logiciels, avec le hardware ?

- **Exigences de performance** : vitesse d'exécution, temps de réponse, disponibilité, temps de détection d'erreur, temps de recouvrement d'erreur…

- **Exigences de qualité** : portabilité, correction, maintenabilité, sécurité...

- **Exigences imposées à l'implantation** : langage, politique de gestion de données, de gestion de ressources, l'OS, consommation d'énergie, encombrement mémoire, poids…

2. Processus d'analyse des exigences/besoins

Les activités d'analyse des exigences regroupent toutes les activités permettant d'élaborer et de maintenir un document correspondant aux exigences imposées au système. Ces activités sont souvent regroupées en quatre familles d'activités : étude de la faisabilité du système, identification et analyse des exigences, spécification des exigences et validation de ces exigences.

2.1. Etude de la faisabilité L'objectif de cette phase est de répondre aux questions suivantes:

- Est-ce que le système va contribuer au développement de l'organisation ? Apporte-t-il un plus ? En quoi va-t-il aider à résoudre des problèmes s'il y en a ?

- Est-ce que le système peut être réalisé avec la technologie actuelle, dans les délais requis et avec les coûts indiqués ?

- Est-ce que le système peut être intégré aux autres systèmes déjà existants ?

2.2. Identification des exigences et leur analyse

L'identification des exigences est liée à chaque domaine de systèmes (banques, automobile, commande d'installation industrielle, BD, multimédia...).

Les questions à se poser pour déterminer ce que doit faire le système sont donc étroitement liées à la nature du système. Il faut être capable à ce niveau de répertorier les exigences, de les classer par catégorie, de leur donner des priorités, de déterminer les liens entre les exigences, les conflits entre exigences.

Le travail est itératif et peut se poursuivre en parallèle avec le début des autres étapes du cycle de développement. Beaucoup de praticiens dans le GL conseillent l'identification des scénarios de fonctionnement du système (ce sont les use cases d'UML) pour mieux voir les interactions du système avec son environnement et les interactions entre les sous-systèmes. Certains scénarios (cas courants d'utilisation) sont souvent faciles à déterminer, d'autres au contraire (cas rares) ne peuvent être identifiés que très tard dans le cycle de vie ou carrément une fois que le système réalisé se trouve en situation Analyse des exigences et spécification

2.3. Spécification des exigences

Une fois les exigences identifiées et classées, il faut utiliser des techniques formelles ou non pour spécifier (décrire) ces exigences en vue d'une utilisation dans les phases suivante du cycle de développement. Comme nous le verrons par la suite, la tendance est de plus en plus vers l'utilisation de techniques formelles (ou au moins semi-formelles) pour spécifier les exigences.

2.4. Validation des exigences

Valider les exigences avec le client (l'utilisateur). Il est toujours conseillé (et obligatoire dans certains cas) de valider les exigences avant de passer aux phases suivantes. Si les erreurs et mauvaises interprétations sont découvertes tard dans le cycle de développement, elles coûtent cher (voire très cher) pour être corrigées.

Pendant l'étape de validation, plusieurs contrôles doivent être effectués :

- **Contrôle de validité** : est-ce les fonctions prévues sont prévues pour être utilisées par les bons acteurs, aux bons endroits, dans les bonnes conditions ?

- **Contrôle de cohérence** : est-ce que certaines exigences ne sont pas en conflit ou contradictoires avec d'autres ?

- **Contrôle de complétude** : est-ce tout ce dont a besoin a été dit pour concevoir et réaliser le système ?

- **Contrôle de réalisme** : est-ce les objectifs fixés peuvent être atteints avec les technologies ciblées ?

- **Vérifiabilité :** pour éviter les malentendus avec le client, contrôler si tout a été écrit de manière à pouvoir vérifier assertions (affirmations, dires, discours...) des uns et des autres. Cette vérifiabilité est facilitée par l'utilisation de méthodes de spécification formelles.

2.5. Classification des exigences

La première phase cruciale que les développeurs de logiciel doivent aborder avec toute l'attention qui s'impose est celle de l'analyse des exigences. C'est à ce niveau que l'on doit savoir pourquoi on veut développer le logiciel et ce qu'il va faire. La difficulté principale (qui conditionne d'ailleurs tout le reste des activités du logiciel) est que le client (utilisateur) ne sait pas toujours exactement ce qu'il veut.

L'utilisateur du logiciel a parfois des idées vagues voire incohérentes de ce qu'il veut et c'est au fur et à mesure que le logiciel est réalisé que l'utilisateur recentre et affine ses exigences. Il s'agit là d'un constat que les ingénieurs du logiciel ne doivent pas oublier.

Même si l'on ne sait toujours pas dire avec précision tout ce que l'on veut sur son futur logiciel, une catégorisation des exigences a été proposée pour faciliter l'identification des exigences :

- **exigences utilisateur** : indiquent ce que le logiciel doit faire et sous quelles conditions de fonctionnement ;

- **exigences système** : indiquent des éléments sur le fonctionnement du logiciel (ergonomie des interfaces, aspects graphiques...) ;

- **exigences logiciel** : apportent des détails sur des contraintes de conception et implantation.

Ces différents besoins sont décrits dans des documents qui sont lus et pris en considération par différents intervenants : responsable clientèle, ingénieurs, architectes de systèmes, développeurs de logiciel...

Tous ces intervenants apportent une attention particulière seulement aux parties de ces documents qui les concernent. Un autre critère très répandu est celui qui consiste à séparer les exigences en deux :

- Exigences fonctionnelles : elles indiquent ce que le système doit faire et comment il doit réagir aux différentes situations (événements) qui peuvent se présenter. Elles indiquent aussi les liens entre les entrées et sorties du système.

- **Exigences non fonctionnelles** : elles indiquent des contraintes sur la manière dont le service du système est rendu.

Ces besoins incluent notamment : les contraintes de temps, l'espace mémoire, le débit, la fiabilité, l'ergonomie, l'utilisation de standards et de règles de développement, de sécurité, de lisibilité, de maintenabilité, de coûts, d'éthique...

Certains parlent de besoins techniques au lieu de non-fonctionnels. Même si on tente de faire cette classification, il est difficile dans la pratique de séparer les exigences de manière claire. Par exemple, on peut considérer la Sécurité comme un besoin fonctionnel ou non-fonctionnel selon l'importance de cet aspect pour l'utilisateur et selon le niveau de détail sur l'élément considéré. On peut dire qu'un logiciel est globalement sûr ou non (il s'agit d'une qualité du logiciel, donc d'un critère non fonctionnel). Mais pour réaliser la sécurité, on a besoin de fonctions d'identification, authentification, chiffrement (il s'agit bien d'aspects fonctionnels).

Il faut noter que lorsqu'on parle de non-fonctionnel, on s'intéresse à des aspects (des propriétés) globaux d'un système et non à des caractéristiques d'éléments individuels du système. En général, lorsqu'une contrainte non fonctionnelle n'est pas satisfaite, le service rendu par le système n'est pas utile, non utilisable, non crédible...

Par exemple, si un système ne répond pas aux contraintes de fiabilité dans le secteur de l'avionique, il ne peut pas être certifié (et on imagine mal un produit non certifié pour commander un avion). Un autre exemple, dans le domaine du temps réel, un système qui fournit des résultats hors délais n'est pas utile (voire dangereux) pour commander des procédés industriels.

En général, les contraintes non fonctionnelles quantitatives sont difficiles à déterminer car il faut préciser des valeurs (un temps de réponse de 5 ms, une fiabilité de 98%, par exemple) à partir de données sûres. Les contraintes non-focntionnelles qualitatives sont parfois plus aisées à annoncer (par exemple, un système rapide, un débit moyen, une interface conviviale, un produit facile à utiliser...).

Pour gérer des contraintes qualitatives, il faut parfois les traduire en contraintes quantitatives (par exemple, un débit élevé peut se traduire par un débit compris entre 1 et 10 Mb/s)

3. Spécification des exigences

Comme dans d'autres domaines de l'ingénierie (l'automobile, le bâtiment, la chimie...), tout produit logiciel doit être spécifié le plus clairement possible.

Attention, en informatique le terme 'spécification' peut être utilisé à différents niveaux du développement du logiciel : spécification des exigences, spécification d'implantation, spécification technique d'un module, spécification d'une propriété, spécification de tests, spécification de procédures de tests, spécification de données, spécification de traitements...

Bref, il s'agit d'une activité dont le but est de préciser (clairement) un aspect. Dans ce paragraphe, nous nous intéressons à la spécification des besoins (ou exigences).

Il est souhaitable qu'une spécification soit claire, non ambiguë et compréhensible, complète, cohérente (sans contradictions). Les descriptions en langue naturelle manquent souvent de précision. C'est la raison pour laquelle on préfère utiliser (même si cela est difficile à faire) des méthodes formelles.

Deux critères orthogonaux sont souvent utilisés pour classer les techniques de spécification :

- **Le caractère formel (utilisation de formalisme)** : on distingue des spécifications informelles (en langue naturelle), semi formelles (souvent graphiques, dont la sémantique est plus ou moins précise) et formelles (quand la syntaxe et la sémantiques sont définies formellement par des outils mathématiques).

- **Le caractère opérationnel ou déclaratif :** les spécifications opérationnelles décrivent le comportement désiré; par opposition, les spécifications déclaratives décrivent seulement les propriétés désirées.

Selon le point de vue du spécifieur et/ou de la méthode utilisée, la spécification peut porter sur un deux ou tous les aspects suivants :

- les fonctions (la spécification du Quoi ?)

- les données d'informations et d'état (la spécification de Qui ?) qui circulent entre les fonctions.

- le comportement ou aspect dynamique (la spécification du Quand ?).

3.1. Spécifications en langue naturelle

Elles sont très souples, conviennent pour tous les aspects et pour tout le monde, sont très facilement communicables à des non spécialistes. Malheureusement, elles manquent de structuration, de précision et sont difficiles à analyser. Des efforts peuvent être faits pour les structurer (spécifications standardisées) : chapitres, sections, items, justifications,

commentaires, règles, etc. Il faut noter que souvent cette forme de spécification est le point de départ pour le cycle de développement du logiciel.

3.2. Spécifications dans des langages spécialisés

Des langages semi formels spécialisés pour spécifier des systèmes ont été proposés. Ils comportent des sections et champs prédéfinis, ce qui force à une certaine structuration. Certains utilisent aussi des langages de haut niveau comme des 'pseudo codes' pour décrire les fonctionnalités attendues. Certains domaines d'activité ont leurs propres langages de spécification, définis et acceptés par les équipes de développement du logiciel dans ces domaines. Par exemple, dans le domaine de l'avionique, PDL (Program Design Language) est largement utilisé par Airbus notamment.

3.3. Spécification avec des diagrammes de flots de données

Il s'agit d'une technique semi-formelle et opérationnelle. Les DFD (Design Flow Datagrams) décrivent des collections de données manipulées par des fonctions. Les données peuvent être persistantes (dans des stockages) ou circulantes (flots de données). Souvent, des conventions graphiques sont utilisées pour représenter :

- les fonctions (ou processus) du système ;

- les flots de données (le plus souvent la direction d'un flux de données est matérialisée par une flèche) ;

- les entités externes qui interagissent avec le système mais qui ne sont pas spécifiées dans leur détail ;

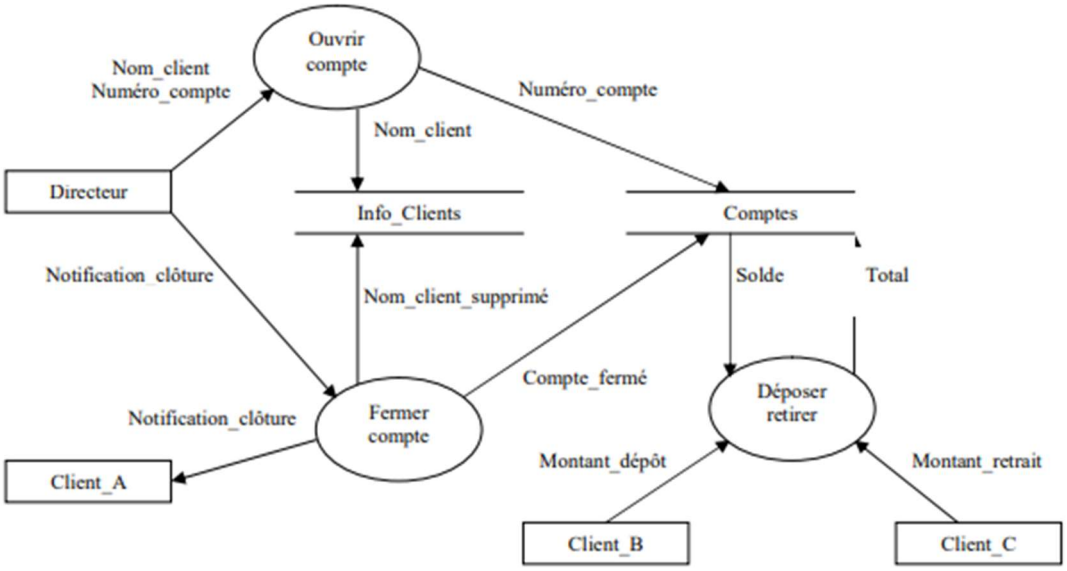
- les points de stockage d'information.

Les DFD peuvent apparaître à différents niveaux d'abstraction du système (niveau le plus élevé, niveau des interfaces..., niveau d'implantation réelle), selon les besoins.

La figure suivante montre un DFD qui illustre partiellement le fonctionnement d'une banque. Il y a quatre entités externes (Directeur, Client_A, Client_B et Client_C), trois processus (Ouvrir_Compte, Fermer_Compte, Deposer_Retirer), deux stockages (Info_Clients et Comptes) et 8 flux (Nom_client, Numéro_compte, Notification_clôture, Montant_Retrait, Montant_Dépôt,

Solde, Total, Nom_client_supprimé, Compte_fermé).

Les DFD sont simples et faciles à comprendre par des non informaticiens qui participent dans des projets de logiciel. Les DFD ont été intégrées à de nombreuses méthodes de conception comme SA-RT, DEMARCO largement utilisés. Les DFD sont généralement insuffisants à eux seuls et doivent être complétés par d'autres moyens de spécification.



exemple de DFD

4. Notion de modèle de système

La notion de modèle est utilisée, à différents stades du développement du logiciel, pour avoir une vue (faire une abstraction) d'un système afin d'en étudier :

- l'aspect externe du système et son interaction avec son environnement

- le comportement du système (aspect dynamique)

- la structure (statique) du système, notamment ses composants et ses données notamment.

On peut avoir des modèles de données, de traitements, d'architectures, de composants et leur composition, des événements... La modélisation consiste à se focaliser sur certains aspects (détails) et ignorer les autres afin de se faire une vue du système.

Les modèles de comportement sont souvent spécifiés à l'aide de diagrammes de flux (DFD) ou les machines à états. Les modèles de données sont souvent spécifiés à l'aide de techniques entités-associations, parfois avec les types abstraits.

5. Conception

Pendant la phase de conception, on propose une solution au problème posé et spécifié lors de la phase précédente (celle d'analyse des exigences). Ici on répond à la question « Comment faire ? ».

Par solution, on entend : une architecture du système (architecture logicielle et architecture physique) et une description détaillée des modules, des interfaces utilisateurs, des données...

La conception donne lieu à un dossier avec souvent une partie destinée au client (présentation de la solution) et une partie pour les réalisateurs (conception technique). La frontière entre ce qui est destiné aux développeurs (programmeurs) et au client n'est pas facile à identifier car elle dépend du type de client considéré.

La phase de conception repose essentiellement sur le choix d'une architecture et des modules qui la composent. Ce choix peut être guidé ou non par des exigences issues de la phase d'analyse des exigences.

Par exemple, le client peut exiger un système décrit en UML et fonctionnant sous Linux et la communication devra faire appel à CORBA supporté par un réseau Internet.

5.1. Notion de module et décomposition de système

Un module est un composant d'une application, contenant des définitions de données et/ou de types de données et/ou de fonctions et constituant un tout cohérent. On peut définir un module comme un fournisseur de ressources ou de services pour d'autres modules. Un module interagit avec les autres modules ; les relations entre modules doivent être identifiées et spécifiées avec précision. Les approches et méthodes de conception insistent beaucoup sur la notion de module. Cependant, elles ne disent rien sur la quantité, la qualité ou le type d'informations nécessaires pour définir un module. Le choix des modules, donc de l'architecture du système, est souvent subjectif ; il dépend des habitudes et de l'expérience du concepteur.

Si on propose un système nouveau à plusieurs concepteurs qui vont travailler de manière séparée, il fort probable d'avoir autant de décompositions modulaires qu'il y a de concepteurs.

Il y a principalement deux approches pour identifier les modules :

- **Approche fonctionnelle** (ou orientée dataflow) : un module est un sous-système du système global.

Chaque sous-système réalise une fonction bien identifiée dans le système. On peut par exemple identifier dans un système industriel des fonctions de gestion de stock, de supervision, de contrôle qualité, de commande des appareils de production...Toutes ces fonctions constituent des soussystèmes industriels. SADT (Structured Analysis Design Technique) et SASD (Structured Analysis, Structured Design) sont deux des méthodes fonctionnelles les plus utilisées.

- **Approche orientée objet** : les modules principaux correspondent aux objets concrets ou abstraits du domaine de l'application ('exemple, un robot, un capteur, une caméra...). Les objets regroupent données et traitements. Ce sont des entités autonomes qui collaborent pour réaliser le système global. Des relations (spécialisation, héritage, généralisation...) sont définies entre les classes d'objets permettant plus de souplesse dans la réutilisation des objets. UML est l'une des techniques fondées sur la notion d'objet qui tend à se généraliser de plus en plus dans le cycle de conception de logiciel.

Que l'approche soit fonctionnelle ou orientée objet, on cherche à diviser le système en entités (modules) plus faciles à comprendre et réaliser séparément.

Des approches plus récentes et plus complexes permettent de mieux maîtriser le cycle de développement du logiciel : approches orientée Patterns et approches orientées Aspects. Un design pattern est une solution de conception commune à des problèmes récurrents dans un contexte donné. Un aspect de conception peut être la synchronisation, la communication, la gestion des exceptions, le partage de ressources... Les méthodes de design pattern et orientées aspects constituent des méthodes au-dessus du concept d'objet pour identifier les

bons schémas et squelettes de logiciel afin de faciliter la réutilisation des composants logiciels et d'exploiter le plus possible le savoir-faire faire acquis grâce à des développements antérieurs.

Un des aspects importants des modules est celui des traitements qu'ils effectuent. Ces traitements sont basés sur des algorithmes (basés sur la théorie des graphes, l'analyse numérique, la théorie des probabilités, les heuristiques...). Le concepteur des modules doit avoir une bonne connaissance dans le domaine des algorithmes pour proposer des modules efficaces. La décomposition en objets ou modules ne permet pas par elle-même de choisir les bons algorithmes.

5.2. Types de conception

5.2.1. Conception d'architecture et ADL

Ce type de conception focalise sur la structuration du système en sous-systèmes. Chaque sous-système doit avoir un rôle bien identifié. Les relations de coopération (échanges de données et de signaux) entre les sous-systèmes sont clairement définies. Une attention particulière doit être portée aux sous-systèmes et données critiques et/ou qui jouent un rôle important dans le contrôle du fonctionnement du système. Différentes solutions peuvent être envisagées selon les situations : solution centralisée, solution répartie, mono-site, solution réseau, solution avec mémoire partagée, solution avec base de données, solution client-serveur, solution peer-to-peer, solution dirigée par les événements, solution dirigée par le temps... La notion de composant est souvent utilisée dans les approches de conception d'architecture.

Des langages dits, xADL (Architecture Description Languages), sont en cours de définition et d'expérimentation. On peut citer : AADL, ACME, Aesop, C2, MetaH, ModeChart, RAPIDE, SADL, Unicon, Wright.

La communauté ADL considère qu'une architecture (logicielle ou matérielle) est un ensemble de composants interconnectés les uns aux autres pour remplir une certaine fonction globale.

Généralement, trois types de concepts sont utilisés dans une architecture :

- Les objets (ou composants) qui définissent des rôles ou des fonctions à réaliser ;
- Les interfaces qui définissent les caractéristiques (visibilité, paramétrage…) des modules communicants.
- Les connexions représentent les interfaces sous la forme de graphe.

Pour être qualifié de ADL, un langage devrait répondre au moins aux exigences suivantes :

- Être adapté à la description de la communication entre composants d'une architecture ;
- Supporter la notion de tâches, de création de tâches et de raffinement de tâche ;
- Fournir une base pour faciliter le passage à l'implantation, voire permettre la dérivation de l'implantation à partir de l'architecture ;
- Offrir des capacités permettant de prendre en compte les styles les plus communément utilisés pour décrire des architectures ;
- Permettre une représentation hiérarchique (abstraction) et faciliter le prototypage.

Les ADLs ont en commun les aspects suivants :

- Utilisation de langages avec une sémantique formellement définie ;
- Utilisation de représentations graphique et textuelle équivalentes ;
- Capacités de modélisation de systèmes distribués ;
- Capacités de raisonnement par abstraction.

Les ADLs diffèrent selon les points suivants :

- Manière de prendre en compte les aspects temps réel (échéances, priorités des tâches…) ;
- Possibilité de prendre en compte et combiner plusieurs styles de description ;
- Possibilités (outils) d'analyse des architectures que l'on vient de décrire ;
- Gestion des instances d'architectures (réutilisation dans différents projets…).

5.2.2. Conception orientée objet

Elle met la notion d'objet au centre de la conception et consiste identifier des objets et classes d'objets, de les hiérarchiser, spécifier les interfaces des objets, spécifier les relations entre objets (spécialisation, généralisation, héritage, concurrence...).

5.2.3. Conception de systèmes critiques

On parle en général de systèmes critiques, lorsque l'on traite des systèmes informatiques en interaction directe avec leur environnement. On dit aussi systèmes temps réel et embarqués (par exemple, systèmes d'aide à la navigation, de pilotage d'aéronef, de commande de procédé chimique, de surveillance de malades...). Les approches de conception de ces systèmes focalisent sur les aspects liés à la dependability qui regroupe la disponibilité1 (availability), la fiabilité (reliability2), la sûreté (safety3) et la sécurité(security4).

Elles mettent l'accent sur les stimuli (données en provenance des capteurs) qui arrivent au système et sur les réponses générées par le système (commandes envoyées à l'environnement). Des contraintes de temps réel sont souvent prises en compte par de tels systèmes afin de tenir compte de la dynamique de l'environnement du système.

6. Validation

D'après la terminologie de l'IEEE (norme 729), la faute est à l'origine de l'erreur qui se manifeste par des anomalies dans le logiciel qui peuvent causer des pannes :

- Faute
- erreur
- anomalie
- panne.

La validation consiste à éviter ou à retrouver et éliminer les fautes avant qu'elles ne conduisent à des pannes.

Validation est le nom donné aux activités de contrôle et analyse pour s'assurer que le logiciel et conforme à sa spécification. La validation doit se faire à toutes les étapes (analyse des exigences, conception, implantation).

La rigueur avec laquelle le processus de validation est effectué est directement liée aux spécificités de l'environnement dans lequel le logiciel sera utilisé (centrale nucléaire, système de téléconférence, éditeur de textes....). Dans certains cas, le client peut accepter un produit

développé à moindre coût et le corriger (faire le déboguage) au fur et à mesure de son utilisation, dans d'autres, le client accepte de payer le prix fort et ne veut pas entendre parler de défaillances (il veut un produit 'Zéro défaut'). Par conséquent, les procédures de Validation et leurs coûts varient considérablement d'un domaine d'application à un autre.

Selon ce que l'on veut valider et le degré de validation, on peut faire appel à différentes approches : vérification, test et simulation.

Il est clair que ces techniques ne sont pas exclusives. En effet, on peut pour une même phase, appliquer deux ou trois techniques soit pour réduire le coût de la validation en validant certains aspects par la preuve et d'autres par les tests ou la simulation, soit parce que le système à valider est trop critique pour se contenter d'un seul type de validation, soit parce que les aspects à valider ne peuvent être tous valider par une seule technique.

Validation et Vérification sont souvent confondues et pourtant elles sont différentes :

- Validation : on veut répondre à la question « développons-nous le bon produit ? »

- Vérification : on veut répondre à la question « développons-nous correctement le produit ? »

La vérification concerne la preuve de propriétés alors que la validation est un processus plus général qui va jusqu'à la satisfaction du client vis-à-vis du produit.

6.1. Vérification informelle : Inspection de fautes dans les programmes

La liste des questions suivantes permet d'aider à éviter (ou retrouver de manière manuelle) des fautes dans les programmes. La liste des questions que l'on se pose pour inspecter le code et déterminer les fautes est évidemment beaucoup plus longue et dépend de chaque contexte, langage et application.

Fautes liées aux données

- Est-ce que toutes les variables sont initialisées avant l'utilisation de leurs valeurs ?

- Est-ce que toutes les constantes ont été nommées ?

- Est-ce que les indices des tableaux sont corrects ?

- Est-ce les chaînes de caractères sont correctement dimensionnées ?

- Y a-t-il des situations suspectes de débordement ('overflow') ?

Fautes de contrôle

- Est-ce la condition de chaque test est correcte ?

- Est-ce que l'on est sûr que chaque boucle se termine ?

- Est-ce que les instructions composées sont correctement construites ?

- Est-ce que chaque Case contient un break quand cela est nécessaire ?

Fautes liées aux entrées/sorties

- Est-ce que toutes les variables d'entrée sont valides ?

- Est-ce que toutes les variables de sortie ont une valeur valide avant d'être envoyées vers l'interface externe ?

- Est-ce des entrées particulières peuvent conduire à des anomalies ? Lesquelles ?

Fautes liées aux interfaces :

- Est-ce que tous les appels de fonctions/méthodes ont des paramètres ?

- Est-ce les paramètres formels coïncidents avec les paramètres d'appel ?

- Est-ce que les paramètres sont dans le bon ordre ?

- Si certains composants accèdent une mémoire partagée, est-ce que ces composants ont tous le même modèle de la structure partagée ?

Fautes liées à la gestion de la mémoire

- Si une structure avec des liens a été modifiée, est-ce que tous les liens ont été correctement mis à jour ?

- Est-ce que l'espace mémoire alloué dynamiquement, a été correctement alloué ?

- Est-ce que l'espace non utilisé est libéré correctement ?

Fautes liées aux exceptions

- Est-ce les situations d'exceptions ont été identifiées et leurs traitement clairement définis ?

6.2. Simulation

Une des techniques les plus utilisées, en particulier dans le monde industriel, pour valider un système est la simulation. Elle consiste à définir un modèle pour le système ou pour son environnement et à exécuter les programmes correspondants sur machine et analyser ensuite les résultats. Il existe de nombreux outils de simulation. Ils se distinguent par leurs capacités à couvrir de larges spectres de fonctionnements du système simulé, leurs temps d'exécution, leurs interfaces... leurs coûts.

Le mot "Simulation" recouvre des usages assez différents. De façon générale, il y a simulation chaque fois qu'un modèle (informatique, pour nous) reproduit la configuration du système à étudier et surtout son évolution dynamique. Le simulateur est ainsi une "maquette" logicielle, complétée par un modèle de l'environnement, que l'on fait évoluer pour y mesurer les grandeurs critiques. Simuler, c'est expérimenter sur un modèle.

De nombreux systèmes admettent une description selon des variables évoluant de manière discontinue. On parle alors de systèmes discrets. La simulation opère sur un modèle "comportemental", c'est à dire une description du système. C'est le type de situation qui est la plus utilisée en informatique et qui est appelée simulation par événements discrets.

Le champ d'applications de la Simulation est évidemment très large (toutes les formes d'applications et de programmes informatiques peuvent être simulées). En particulier, la simulation est utilisée avec succès à la validation de bon fonctionnement de mécanismes "complexes" (typiquement, les protocoles, et plus généralement toutes organisations dont la complexité engendre une combinatoire délicate à maîtriser).

Un autre usage du mot, ce qu'on appelle la simulation d'environnement, qui recouvre les applications du genre "Simulateur de vol", et systèmes d'entraînement analogues. La mise au point de systèmes informatiques temps réel fait appel à cette technique. Il s'agit là de reproduire des conditions rares et délicates, et de procéder au réglage des paramètres de contrôle du logiciel impliqué (régulation de surcharges, pannes d'organes, etc).

La simulation d'un système nécessite de décrire les variables caractéristiques de ce phénomène, les états du système et les événements qui font évoluer l'état du système. On appelle système discret un système dans lequel la description d'état ne comporte que des variables discrètes (i.e. à variation non continue). Les instants d'évolution de l'état surviennent donc de façon "discrète" (c'est à dire qu'il n'y en a qu'un nombre fini dans un intervalle de temps fini). On les appelle des événements (on parle aussi d'événements discrets).

La description d'états dépend du problème posé, c'est-à-dire du système à étudier mais aussi des aspects qu'on en privilégie.

La simulation par événements va consister, une fois le modèle décrit, à le faire évoluer, en mettant à jour la description d'état à chaque événement. Cela signifie que le Simulateur va sauter d'un événement au suivant.

L'exécution d'un événement est instantanée (dans le temps du modèle). Chaque événement engendre d'autres événements. Puisque "il ne se passe rien" entre les événements, c'est-à-dire puisque les variables significatives n'évoluent qu'à ces instants, il n'est pas utile de considérer les périodes inter-événements!

Lorsque le simulateur passe d'un événement au suivant, la date saute de la valeur courante à celle du nouvel événement.

La simulation est souvent partielle, elle ne constitue pas une preuve que le produit est correct, mais elle permet seulement d'affirmer que si les cas simulés (et seulement ces cas) vont se présenter dans la réalité, alors le système fonctionnera correctement.

En fin, il faut noter que la simulation n'est pas seulement un outil de résolution numérique, mais aussi une discipline de modélisation : aide à la description et à la compréhension des mécanismes étudiés.