INFOF-204 : Analyses et méthodes informatiques

Robin P.

année académique 2015 - 206

Table des matières

1 Software Engineering

Comme toute branche, la création de programmes informatiques est soumise à des méthodes de réalisation. Étant donné que le client réclamant le programme et les développeurs ne sont pas les mêmes personnes, les choses évidentes pour l'un ne le sont pas pour l'autre. Dès lors, il a fallu trouver un moyen efficace de faire communiquer client et équipe de développement de manière efficace. En effet, les deux plus grandes causes d'échec des programmes sont la mauvaise compréhension des besoins du client et la mauvaise communication au sein de l'équipe de développement.

De plus, à l'instar des problèmes classiques de conception (ingénierie civile), l'ingénierie informatique [1] est soumise à des règles qui lui sont propres. Par exemple, un produit *fini* (sous-entendu délivré) n'est pas un produit définitif. La maintenance et le suivi du produit font bien souvent partie inhérente du contrat.

1.1 Waterfall

La méthode dite *Waterfall* est une méthode datant des années 1970 conçue fondamentalement par un principe de phases :

La récolte des besoins est la première étape où il y a rencontre avec le client afin de comprendre précisément ce qu'il demande (explicitement et surtout **implicitement**).

L'analyse est la phase où les analystes récupèrent les résultats des entrevues avec le client et établissent précisément les besoins.

L'architecture consiste à penser la structure du produit : concevoir les différents modules qui vont être d'application, les *packages* et *libraries* qui devront être utilisés.

L'implémentation est la phase où les développeurs codent les modules qui leur sont « imposés » par l'architecture et les lient avec les libs.

Les tests servent à s'assurer que le produit est délivrable (non buggé et fait ce qu'il faut).

La maintenance ne se fait qu'après avoir livré le produit, elle permet au client d'avoir un produit à jour par rapport à ses besoins et fonctionnel.

La caractéristique majoritaire de cette méthode est que les différentes phases sont réalisées par des personnes ou équipes différentes qui doivent communiquer les unes avec les autres à l'aide de documents. Il y a donc risque de mal interpréter lors du passage d'une équipe à l'autre, voire de mal exprimer la documentation pour l'équipe suivante. Cette méthode est cependant pratique pour pouvoir avoir un document de documentation très complet et pouvoir dès lors faire du travail d'analyse sur le projet même sans devoir se plonger dans le code de l'application (à condition de maintenir ce document à jour pour ne pas avoir une documentation *outdated*). Un gros désavantage du Waterfall est qu'il est totalement inadapté à la réalisation d'un projet où le besoin du client évolue pendant le développement car les besoins sont récoltés au tout début et le produit n'est livré qu'une fois toutes les étapes (sauf la maintenance) réalisées. Il peut y avoir plusieurs années entre la récolte et la fin du code. Si les besoins ont changé, le client risque de recevoir un produit ne correspondant plus à ses attentes.

1.2 Procédé itératif

Pour se débarrasser autant que possible des problèmes de communication et des risques de non-conformité aux besoins du client venant du Waterfall, d'autres méthodes ont été trouvées. Par exemple, les procédés itératifs ont pour principe de récolter une première fois les besoins du client pour avoir une *idée* du produit fini mais de faire des *itérations de développement* (durant entre 2 et 4 semaines en général) où les équipes de développement et d'analyse travaillent ensemble pour pouvoir livrer au client le produit au fur et à mesure afin qu'il puisse participer à la redirection du projet s'il prend une mauvaise tournure (non conforme aux attentes). Les chances de réussite du projet sont augmentées par l'implication du client dans l'organisation.

2 L'orienté objet

Plus qu'une manière de développer, la *programmation* orientée objet est une manière de concevoir les projets. En effet, il est possible de concevoir une application en la pensant objet dès le tout début. **Attention**, cela ne veut pas dire que les objets et classes définis lors de la conception seront ceux et celles qui se retrouveront dans le code de l'implémentation.

1. Software engineering.

Le principe général et fondamental est d'avoir des objets « indépendants » les uns des autres pouvant à la fois manipuler de l'information et la traiter au sein d'une boite noire, une interface.

2.1 Les ADT

Les Abstract Data Types ont pour principe de séparer l'implémentation interne de la spécification externe. Le but est d'avoir une interface prédéfinie et statique qui permet de lister toutes les fonctionnalités que peut proposer un *objet* sans préciser comment il les réalise. Lors du développement, si le code est conçu proprement, des courts modules sont conçus et interagissent les uns avec les autres. Un code est propre s'il est *découplé*, à savoir si les objets sont le moins dépendants les uns des autres. En effet, les objets **doivent** interagir mais **ne doivent pas** connaitre la représentation interne les uns des autres. De plus, le fait de concevoir un code découplé et dont l'interface est séparée de la définition permet d'avoir un code beaucoup plus lisible et beaucoup plus maintenable. De fait, si un changement de représentation interne est à réaliser dans le code, il suffit de changer la méthode associée au message. Si ce principe de découplage n'a pas été appliqué, il faudrait aller modifier **toutes** les zones du code qui font appel à cette partie de la structure. Les désavantages étant qu'il est très facile d'oublier un endroit et donc d'avoir un code non fonctionnel avec une erreur dans un code déjà testé et *a priori* fonctionnel et que faire plusieurs fois la même modification à plusieurs endroits du code est contre-productif.

Afin de réduire encore plus les modifications quand il doit y en avoir, il est intéressant d'utiliser l'interface de l'ADT au sein même de l'ADT : que les méthodes de traitement n'accèdent aux attributs, dans la mesure du possible, qu'à travers les méthodes prévues à cet effet. De cette manière, le code ne sera pas répété et dès lors les modifications d'adaptation et de maintenance seront simplifiées et accélérées. Il faut garder en tête que de multiplier le nombre d'appels de méthode peut avoir pour effet de ralentir le programme final par rapport à un code optimisé au maximum mais dans la **grande majorité des cas**, on peut se permettre de perdre un peu en temps d'exécution pour gagner beaucoup en lisibilité et en temps de maintenance.

2.2 Les objets

Un objet est *précisément* défini par ses trois caractéristiques : son **identité**, son **état** et son **comportement**. L'identité d'un objet représente son unicité : il est possible d'avoir deux objets identiques, ayant le même état mais d'identité différente, étant deux entités différentes. L'état d'un objet permet de le différencier des autres objets de même type et de particulariser son comportement (cela est *globalement* représenté par la valeur de ses attributs). Enfin, son comportement est sa manière de réagir aux messages qui lui seront envoyés par les autres objets.

La notion de message n'est pas à confondre avec la notion de méthode, bien qu'elles soient fort reliées. Un message est une transmission particulière d'un objet à un autre. Il représente la demande d'exécution d'un traitement particulier et est représenté par « ce qui est demandé ». La méthode quant à elle est une réalisation d'un traitement demandé par un message. La méthode est représentée par « comment l'objet réalise ce qui lui est demandé ».

Lors de la compilation (ou de l'exécution), il faut que le compilateur (ou l'environnement d'exécution) détermine quel code exécuter (quelle méthode appeler) lors d'une transmission d'un message. Le *Method Lookup* est le procédé permettant de déterminer quelle méthode appeler (sera expliqué après l'héritage et le polymorphisme).

2.3 L'héritage

L'héritage est un principe permettant de réutiliser une classe afin de la spécialiser, de la peaufiner. Lors d'un héritage, on dit que la classe D hérite (ou dérive) de la classe B lorsque la classe D contient tous les attributs de B (plus éventuellement les siens) et sait traiter au moins tous les messages que D sait recevoir. Si la classe D est définie uniquement par ce que sait faire B, l'héritage ne sert à rien, les deux classes sont équivalentes.

Une classe fille (dérivée) a pour intérêt de pouvoir redéfinir le comportement de réaction aux messages qu'elle reçoit. Dès lors, si D dérive de B, une classe quelconque, alors en redéfinissant le comportement de D par rapport à B, on permet à D de recevoir les mêmes messages que B mais en les traitant différemment.

De plus, un langage manipulant l'orienté objet et l'héritage gère également des pointeurs ou des références. Ces types particuliers sont une manière de nommer (référencer un objet sans le manipuler directement). Étant donné qu'un *bon héritage* de D dérivant B veut dire que D *est une sorte de* B, il est possible d'assigner un objet de type D à la référence si elle est sur type B. En effet, le principe

de la référence signifie qu'à travers ce nom, on veut manipuler un B. Comme D *est un* B (en réalité, un objet de type D est **au moins** de type B), l'assignation est cohérente : en manipulant la référence, on manipulera un B.

2.3.1 Les classes abstraites

Soit une classe B n'ayant aucun attribut et définissant uniquement les messages qu'un objet l'instanciant peut recevoir, on dit que B est *abstraite*. Dès lors, B ne peut être instanciée, il ne peut exister d'objet de type B. Le principe d'une telle classe est de servir d'interface : si A est une classe abstraite et que les classes B, C et D dérivent de A, alors on sait que ces trois classes spécialisées ont un lien entre elles qui est qu'elle doivent toutes pouvoir réagir aux mêmes messages, ceux imposés par A. Lorsqu'une classe abstraite est implémentée (et donc devenue instanciable), on parle de *concrétisation*. Il faut cependant que **toutes** les méthodes abstraites de la classe soit redéfinies afin de concrétiser la classe. Si ce n'est pas le cas, la nouvelle classe est également abstraite et devra être redéfinie pour pouvoir être instanciée. L'intérêt de tout cela est dans le polymorphisme.

2.3.2 Le polymorphisme

Le principe de base du polymorphisme est que plusieurs objets de classes différentes peuvent réagir (chacun à leur manière) au même message. Dès lors, on peut pousser le découplage jusqu'au fait où l'envoyeur du message ne sait même pas précisément à qui il envoie le message, tout ce qu'il sait est que le récepteur du message sait le traiter (comme il le veut). Le lien entre polymorphisme et héritage est le suivant.

En considérant la hiérarchie suivante : soient A, B, C, D et E cinq classes telles que B, C et D héritent de A et E hérite de C. Supposons comme vu plus haut que A définit un certain comportement de réaction à un message donné et que B, C, D et E redéfinissent chacun un nouveau comportement en rapport avec leur cohérence interne. Si une méthode manipule une référence sur un objet de type A, on peut lui fournir soit un objet de type A, soit de type B, C, D ou encore E. Or chacune de ces classes a redéfini le comportement de réponse audit message. Dès lors, selon l'objet passé à travers la référence, le comportement sera différent ; le même message aura été envoyé, mais le comportement aura été différent, ce qui est bien la définition du polymorphisme.

2.3.3 Method Lookup, this & super

Le principe du *method lookup* peut donc être expliqué maintenant. En effet, il n'est pas pertinent d'uniquement regarder le type de récepteur du message et d'appeler la méthode ayant le même nom que le message au sein de cette classe car il se peut que la méthode n'existe pas au sein de cette classe et donc qu'il faille aller voir plus haut (sous-entendant dans les classes parentes). De plus, avec le polymorphisme, il faut bien faire la différence entre le type de l'objet et le type de la référence sur l'objet car le type de la référence donne en réalité une *borne inférieure* de la hiérarchie de classes.

De par ce polymorphisme, il faut que les objets gardent un moyen de se retrouver. C'est le principe de **this** qui est soit un pointeur, soit une référence (dans les langages plus récents) vers l'objet lui-même. Ceci permet à un objet de s'appeler lui-même pour un sous-traitement d'un message mais cela permet également de savoir de quel type est l'objet. Par exemple, soient B et D deux classes telles que D dérive de B. Si B définit deux méthodes m1 et m2. Supposons que m1 est redéfini par D. Dès lors, le code suivant affichera « `B::m1, D::2` » et pas « `B::m1, D::m1` ».

```
class B {
    void m1() {
        print("B::m1, ");
        this.m2();
    }
    void m2() {print("B::m2");}
}

class D inherits B {
    void m2() {print("D::m2");}
}

new D().m1();
```

Il faut savoir que le mot-clef `this` est donc dynamique : ce qu'il représente n'est pas fixé lorsque le code est écrit. En effet, dans le code ci-dessus, `this` peut signifier plusieurs choses différentes selon qui reçoit le message « m1 ». Il existe un autre mot-clef très important dans la notion d'orienté objet, c'est le mot-clef `super`. Contrairement à `this` qui représente l'objet traitant le message, `super` représente la classe parente. Dès lors, super est statique car la classe parente peut être déterminée en lisant le code source. En effet si la classe D dérive de B, `D.super` (ou `super(D)` en fonction du langage et des choix de syntaxe) représentera **toujours** la classe B [2].

3 Unified Modeling Language

UML est un standard de modélisation de projet. Il permet de structurer et d'exprimer l'analyse d'un projet en utilisant la notion d'objet. Le concept fondamental d'UML est la *vue*. Il existe 5 vues différentes qui se partagent 9 types de diagrammes.

3.1 Les vues

1. Les **use case views** représentent le fonctionnement de l'application depuis l'extérieur : comment les utilisateurs (acteurs) interagissent avec le système. Les diagrammes utilisés sont **use case** et **activity diagrams**.
2. Les **logical views** représentent une vision haut niveau de l'application : elles doivent décrire son fonctionnement. Les logical views représentent l'aspect statique de l'application. Les diagrammes utilisés sont **class (+ object)**, **state**, **sequence**, **collaboration** et **activity diagrams**.
3. Les **component views** servent majoritairement aux développeurs : elles représentent les blocs composant l'application, les dépendances entre les modules, etc. Les diagrammes utilisés sont les **component diagrams**.
4. Les **deployment views** représentent l'architecture de déploiement de l'application : quel module est lié à quelle machine. Les diagrammes utilisés sont les **deployment diagrams**.
5. Les **concurrency views** représentent les interactions, communications et nécessités de synchronisation entre différents modules. Les diagrammes utilisés sont les **sequence diagrams**.

C'est la *use case view* qui va être faite en première afin de diriger le développement et la création des autres vues. Cependant, la vue la plus utilisée et la plus complète est la *loical view*.

3.2 Les diagrammes

Comme mentionné plus haut, les diagrammes sont au nombre de 9 et sont utilisés dans une ou plusieurs vue(s).

1. Le **use case diagram** représente les fonctionnalités offertes par l'application. Il est composé d'un rectangle contenant les activités et d'acteurs (représentés par des bonshommes) qui sont reliés aux fonctionnalités exprimant ainsi *qui peut faire quoi*.
2. Le **class diagram** représente la structure interne de l'application, sa hiérarchie, les relations entre les différentes classes (au sens analytique). Les conventions sont de représenter les classes en trois rectangles superposés : le premier contient le nom de la classe (*CamelCase*), le second contient les attributs nommés ainsi qu'un indice de visibilité (+, -, # pour *public*, *private*, *protected*) et le dernier contient la signature des méthodes.
3. L'**object diagram** est assez proche du *class diagram*. Il est plus précisément un exemple de class diagram (et n'a donc pas de sens sans ce dernier). Les conventions sont de représenter un class diagram en donnant une valeur aux attributs et en soulignant le nom des objets.
4. Le **state diagram** représente l'évolution des objets du point de vue de leur état. Il est composé de rectangles contenant l'état et de flèches identifiées par une action représentant le passage d'un état à un autre.
5. Le **sequance diagram** représente une vue temporelle de l'évolution des interactions entre les composants (les objets).
6. Le **collaboration diagram** est très proche du *sequence diagram*, représente les mêmes choses mais d'un point de vue spatial.
7. L'**activity diagram** est assez proche du *state diagram* si ce n'est que les rectangles représentent des actions et non des états et que les flèches reliant les rectangles représentent les éléments déclencheurs des actions [3].
8. Le **componenet diagram** représente les entités physiques finales nécessaires et leurs dépendances.
9. Le **deployment diagram** représente les différentes unités finales ainsi que leurs interactions.

2. S'il est question d'héritage multiple, le mot clef super est remplacé (par exemple en C++, le nom de la classe parente suivi de « : : » doit précéder le message).
3. Une flèche étiquetée par un message entre crochets désigne une condition.

3.3 Les relations de classes

Deux classes peuvent être en relation pour plusieurs raisons. En effet, il est possible qu'elles soient **associées** ce qui est le cas quand l'une utilise l'autre (exemple : la classe `TextParser` utilise la classe `FileManager` pour traiter le message `read`), il est également possible qu'une soit **généralisation** de l'autre ce qui est le cas lorsqu'il y a héritage (exemple : la classe `Stack` hérite de la classe `Deque`), il est aussi possible qu'une soit **raffinement** de l'autre ce qui est le cas lorsqu'une classe D hérite d'une classe abstraite A sans totalement la concrétiser (en réalité, la classe raffinée décrit la même chose que la classe initiale mais à un niveau d'abstraction différent), ou encore, il peut y avoir **concrétisation** de l'une par l'autre ce qui le cas lorsqu'une classe D hérite d'une classe abstraite A en implémentant tout le nécessaire afin de ne pas être abstrait à son tour.

3.3.1 L'association

Deux classes associées sont deux classes se connaissant parce que l'une utilise l'autre. Ce peut être le cas par exemple car l'une est attribut de l'autre, parce que l'une est passée en paramètre dans une méthode de l'autre, parce que l'une est déclarée dans l'autre, etc. Une association normale entre deux classes se fait par une ligne allant d'une classe à l'autre. *A priori*, cette relation est bidirectionnelle. Si elle ne l'est pas, ce doit être précisé explicitement par un triangle montrant l'orientation accompagné d'un message signifiant le type d'association (ows, uses, refers, etc.) De plus, la multiplicité d'une relation est 1/1 par défaut (ou juste 1 si la relation est monodirectionnelle) mais peut être explicitée si ce n'est pas le cas.

Il existe également des types particuliers d'associations : les agrégations et les compositions. Une relation d'agrégation est marquée par le fait qu'une des deux classes est partie intégrante de l'autre (exemple : un livre peut avoir une couverture, donc il y a une relation d'agrégation entre le livre et la couverture). Une composition est une agrégation encore plus forte dans le sens où la classe conteneur n'a de raison d'exister que si la classe contenue existe (exemple : un livre a des pages, et ces pages sont **physiquement** directement dans le livre, donc il y a une relation de composition entre le livre et les pages).

Concernant les notations, une association simple se note par une simple ligne, une agrégation se note par une ligne avec un diamant vide du côté du conteneur et une composition se note par une ligne avec un diamant plein du côté du conteneur. De plus, une relation de généralisation se note par une ligne avec un triangle creux dirigé vers la classe générale et un raffinement se note par une ligne en pointillés (ou tirets) avec un triangle creux dirigé vers la classe générale.

4 Ingénierie des besoins

Il existe plusieurs types de besoins : une classification s'opère en fonction de qui sera amené à lire ces spécifications et donc de leur objectif. Les besoins lus par le client ou ceux lus par les architectes et les développeurs ne sont pas les mêmes : ils font preuve d'un niveau d'abstraction différent.

- **Besoins de l'utilisateur** D'un niveau d'abstraction plus haut, les besoins de l'utilisateur sont *a priori* majoritairement destinés au client ainsi qu'aux architectes. Ils reprennent les fonctionnalités que doit pouvoir offrir l'application et dans quelles conditions celles-ci doivent se dérouler.
- **Besoins du système** Les besoins du système sont plus bas niveau et sont là pour reprendre les détails du fonctionnement du système et sont *a priori* destinés à être lu par les développeurs, architectes et ingénieurs.

Ces deux *classes* de besoins peuvent encore se subdiviser (toutes deux) en 3 sous-classes :

- besoins fonctionnels ;
- besoins non-fonctionnels ;
- besoins de domaine.

Les besoins fonctionnels sont les nécessités du point de vue des fonctionnalités directes de l'application ou du système. *Que doit-on pouvoir faire avec ?* **Attention** : le but est d'exprimer **ce que** doit faire l'application et non **comment** l'application doit le faire. Les besoins fonctionnels de l'utilisateur sont très abstraits, représentent des actions générales. Les besoins fonctionnels du système doivent être beaucoup plus détaillés : décrire les entrées/sorties, les cas exceptionnels, etc. Ces besoins-là (tant du côté utilisateur que système) doivent être précis et complets en sachant qu'il faut les comprendre directement du client et extrapoler.

Les besoins non-fonctionnels quant à eux sont les besoins ne représentant pas une fonctionnalité directe mais plutôt des contraintes relatives au projet (exemple : nombre de connexions entrantes limitées) ou des *propriétés* du projet (exemple : fiabilité pour un

système en temps réel). Afin de limiter l'interprétation de ces besoins, il est commun d'essayer de quantifier dans la mesure du possible afin de rendre les nécessités vérifiables.

Les besoins de domaine sont les besoins n'étant pas directement requis par le projet mais étant *imposés* par le domaine d'application (règles de développement dans le domaine médical ou dans le domaine de la sécurité informatique, etc.)

Les besoins étant récupérés directement chez le client (la plupart du temps à l'oral), il faut tenter de les faire expliciter dans la mesure du possible. Ensuite, lors de la rédaction du rapport des besoins, il faut être le moins jargonnant et le plus simple possible dans l'expression pour faciliter la communication et limiter les mauvaises interprétations. De plus, la transmission des besoins du système par langage écrit (ou oral) est à proscrire dans la mesure du possible : il faut préférer les schémas (UML) et éventuellement le formalisme mathématique pour les projets le permettant.

4.1 Software Requirement Document

Le SRD est le document relatif au projet dans lequel les détails sont expliqués. Il en existe plusieurs standards selon les cas de projets. Le but du SRD est à la fois d'avoir une documentation complète du projet et à la fois d'avoir un support d'échange entre les différentes équipes de travail. Si le projet est de relativement petite amplitude ou est réalisé seul(e), un SRD n'est pas nécessaire et un simple document de type *compte-rendu* fait l'affaire.

Un SRD est amené à changer, même après la fin de la première analyse. Lorsque le projet se modifie ou que l'analyse, l'architecture est modifiée, il faut mettre à jour le SRD. Il faut donc en garder un historique en affichant les changements majeurs d'une version à l'autre.

5 Modélisation dynamique

L'UML offre plus que simplement une représentation *statique* de l'application : il y a moyen de modéliser son aspect dynamique (son comportement, son évolution, etc.) En effet, en UML, il y a quatre diagrammes destinés à la représentation dynamique. Deux concernant le dynamisme inter-objets et deux concernant le dynamisme intra-objet :

- Le **dynamisme inter-objets** :
  - le diagramme de séquence (*time-driven*) ;
  - le diagramme de collaboration (*space-driven*).
- Le **dynamisme intra-objets** :
  - le diagramme d'état (*state/event-driven*) ;
  - le diagramme d'activité (*work/services-driven*).

5.1 Diagrammes d'interactions

Les diagrammes de séquence et de collaboration sont strictement sémantiquement équivalents : ils expriment la même chose mais selon un point de vue différent. Ils peuvent donc tous deux être convertis l'un en l'autre sans perte d'information.

Sequence Diagram Les diagrammes de séquence sont orientés écoulement du temps. Ils sont représentés verticalement avec un flux d'exécution de haut en bas. Le nom des classes (ou objets si instanciation) étant en haut et leur durée de vie étant une ligne verticale. On représente les messages envoyés de l'un à l'autre par des flèches horizontales (ou obliques pour montrer une latence explicitement) qui peuvent être accompagnées de contraintes ou de structures algorithmiques (à éviter : plutôt dans les diagrammes d'activité).

Collaboration Diagram Les diagrammes de collaboration sont utiles lorsqu'il y a trop d'objets pour avoir un diagramme de séquence clair. Dans le cas d'un diagramme de collaboration, il n'y a plus de dimension de temps à proprement parler : l'ordre des opérations est représenté par une numérotation (à plusieurs niveaux hiérarchiques). Le diagramme se concentre sur les canaux de communication et de collaboration entre les objets. Ce type de diagramme est **légèrement** plus bas niveau car des pseudo-instructions peuvent y être exprimées. De plus, ce type de diagramme permet de représenter la *visibilité* d'un objet par rapport à un autre : est-il connu car il est un attribut, un paramètre, une variable locale, etc. ?

5.2 Diagrammes de structure interne et de synchronisation

Les diagrammes d'état et d'activité servent à montrer le fonctionnement interne et la structure des objets.

Statechart Diagram Un diagramme d'état est un ensemble des rectangles contenant les états dans lesquels peut se trouver un objet. Ces rectangles sont reliés par des flèches étiquetées par un couple action déclencheur/action déclenchée. Un tel diagramme commence au cercle noir plein et finit au cercle noir plein entouré. Si plusieurs flèches avec le même élément déclencheur sortent d'un état, c'est que l'objet passe dans une combinaison de ces deux états-là. Un état est une définition récursive : si plusieurs sous-états font partie du même état génral (exemple : dans un jeu, un joueur peut être en partie et pendant son tour, en partie et pendant le tour de l'adversaire qui sont tous deux des sous-états de l'état en partie qui s'oppose à l'état en organisation du profil.)

Activity Diagram Un diagramme d'activité est une forme particularisée de diagramme d'état mais où les rectangles ne représentent pas les états mais bien les actions en cours réalisées par l'objet. Il y a une notation pour les branchements conditionnels (diamant avec une branche entrante et deux sortantes ou deux branches entrantes et une sortante), une notation pour les *forks* (une barre épaisse horizontale avec une flèche entrante et deux flèches sortantes ou deux flèches entrantes et une flèche sortante). Il y a également une notation pour les traitements en parallèle au sein d'une action (comme les états récursifs pour un statechart diagram), à savoir : une action contient elle-même plusieurs points d'entrée (cercles noirs pleins) et plusieurs points d'arrivée (cercles noirs pleins entourés) tels que la sortie de l'état ne peut se faire qu'une fois tous les points d'arrivée atteints.

Remarque : le branchement conditionnel (*branch/merge*) représente un choix de branchement : l'un **ou** l'autre ; la (dé)synchronisation (*fork/join*) représente une séparation **obligatoire** des traitements telle que l'ordre d'exécution n'est pas important.

5.3 Diagrammes restants

Les component diagrams représentent les inter-dépendances entre les composants (modules, objets, systèmes, etc.) alors que les deployment diagrams représentent les différents nœuds d'exécution (serveur, database, clients, etc.)

6 Les tests

Afin de *s'assurer* (ou du moins afin de se convaincre) que le code produit est fonctionnel, il faut le tester. On peut soit tester l'application finie (la lancer puis tenter d'utiliser toutes les fonctionnalités afin de trouver les bugs) ou on peut également faire des **tests unitaires** dont le but est de tester indépendamment un module en le soumettant à différents cas d'utilisation.

6.1 Invariant, pré/post-conditions

Pour définir ce que peut et doit faire un module (bout de code), il est commun de l'exprimer de manière formelle. On parle donc en terme d'invariant pour décrire une réalité qui ne change pas tout au long du déroulement du traitement (par exemple, pour une pile size ≥ 0), de pré-condition pour décrire ce qui est supposé comme vrai pour supposer le traitement (par exemple : pointeur $\neq$ NULL), et de post-condition pour décrire ce qui est assurément vrai à la fin du traitement (par exemple : size > 0 voire size - initialSize $== 1$ pour une méthode d'ajout dans une liste, file, pile, etc.)

Les conditions comme cela peuvent être exprimées à l'aide d'assertions. Les assertions sont fournies dans la plupart des langages modernes et donnent un moyen de s'assurer qu'une condition est vraie, si elle ne l'est pas, soit le programme s'arrête soit une exception est lancée (langages récents). De plus, les assertions peuvent être supprimées pour les versions *release* car *a priori*, si le code a bien été testé, les erreurs de pré/post-conditions ne devraient pas être là. Les assertions aident également à documenter les conditions d'exécution car elles sont exprimées assez formellement et placées directement dans le code. Pour l'argument de clarté, une méthode `invariant` de signature `bool invariant(void)` peut être utilisée pour exprimer l'invariant d'une classe.

6.2 L'écriture de tests

Les tests doivent être écrits en même temps que le code pour s'assurer tout au long du développement qu'un module sur lequel reposeraient d'autres modules ne soit pas bancal. De plus, une fois les tests écrits, il est important de lancer **l'entièreté** des tests régulièrement afin de voir si le nouveau code est correct **et** que le code produit précédemment fonctionne toujours. En effet, les tests permettent également de vérifier la retrocompatibilité du code. Du code produit un an auparavant peut avoir été modifié ou ses conditions d'utilisation peuvent avoir changé. Dès lors, il faut être sûr qu'il est toujours fonctionnel et cohérent avec l'architecture actuelle.

La structure générale d'un test se fait en trois points :

1. la création de contexte ;
2. l'envoi d'un stimulus à l'objet testé ;
3. la vérification des données résultant du traitement.

De plus, il apparait qu'un bon test doit se concentrer sur **un et un seul** traitement afin de ne pas cacher un bug par un traitement successif ou même afin de bien localiser l'erreur si erreur il y a. De plus, un test doit être déterministe au totalement automatique.

Remarque : la différence entre **échec** et **erreur** est qu'un échec est un test (ou une assertion) qui ne passe pas et une erreur est une condition qui n'a pas été prévue.

Aujourd'hui, il existe plusieurs *frameworks* de tests. Le principe est de fournir une interface simple (à coup de classes) permettant d'implémenter des méthodes internes qui feront les tests et des méthodes **spéciales** (souvent appelées `setup` et `teardown` ou équivalent) destinées à établir et détruire le contexte. La terminologie parle de testcase pour désigner **un test unitaire** (une classe simple avec une méthode de test (et éventuellement des sous-méthodes de traitement)), de testfixture pour désigner un test plus complet (avec création et destruction de contexte). Les tests sont basés sur le principe d'assertion afin que le framework puisse voir si la condition est vérifiée ou non. Il y a également des procédés de vérifications d'exceptions lancées.

7 Implémentation et théorie du design

Cette dernière section traite des règles d'implémentation des problèmes rencontrés lors de l'implémentation des projets.

7.1 Règles de codages et bonnes pratiques

Il existe beaucoup de *règles* de bonnes pratiques qui sont à considérer comme propositions afin d'avoir un code bien pensé et propre. Certaines concernent le **code** en lui-même, les fichiers contenant les caractères : comment nommer ses variables, ses méthodes, comment structurer le code affiché, etc. D'autres concernent la structure derrière le code et les règles d'organisation et d'analyse.

Par exemple, la **Loi de Pelrine** dit qu'un objet ne doit prendre comme responsabilités que ce qu'il ne peut déléguer et qu'il ne doit laisser en *public* dans son interface que le strict nécessaire à l'extérieur pour lui envoyer les messages relatifs aux responsabilités qu'il a prises. En effet, cette loi permet d'avoir des objets beaucoup plus stables : une nouvelle fonctionnalité impliquerait un nouvel objet avec de nouveaux services à rendre et **un** nouvel appel dans la méthode de traitement à modifier. Dès lors, les modifications sont amoindries. Cependant, prise au pied de la lettre, cette règle risque de diviser un projet en un nombre incalculable de toutes petites classes. Elle a plus de sens lorsqu'elle est couplée avec la **loi de Demeter** qui dit qu'un objet ne doit pouvoir envoyer un message qu'à un argument qui lui est passé (dans une méthode) ; un objet **qu'il** a créé ; lui-même ou son père. Dès lors, on a des petites classes ayant chacune une série de traitements précis à accomplir et ne s'éparpillant pas, n'envoyant des messages qu'aux objets qui leur sont très proches.

La loi de Parnas (principe d'information cachée) fait le lien entre les règles de code écrit et de structure de code : elle dit qu'un développeur ne doit donner aux utilisateurs de son module que le nécessaire pour **utiliser** son module et pas plus. Autrement dit, un module ne doit donner comme seule information le nécessaire à son utilisation. Cela se rapproche assez fort du principe d'encapsulation disant qu'il faut séparer la déclaration d'une classe (son interface) *a priori* disponible à tous les objets l'utilisant de sa définition (son implémentation) qui ne doit être disponible pour personne d'autre que l'objet lui-même (et éventuellement les classes en descendant dans la hiérarchie d'héritage).

Concernant le code écrit, il faut savoir que ce code est relu beaucoup de fois, soit par le développeur du module soit par d'autres développeurs (projets open-source, équipe de maintenance, large équipe de développement, etc.) Dès lors, il faut écrire *proprement*

afin que ceux qui sont amenés à lire le code ne perdent pas de temps en déchiffrage. Il existe pour cela des *conventions de nommage*. Certaines sont générales, d'autres sont propres aux langages. Par exemple, peu importe le langage, une variable doit porter pour nom un nom commun, une fonction (méthode) doit porter pour nom un verbe ou une phrase verbale, si la fonction renvoie un booléen, il est commun de la faire débuter par *is*, etc. Concernant les règles propres aux langages, certains langages (Java) recommandent le camelCase, d'autres (Python) recommandent la séparation par underscore, etc.

En plus des *grands standards* qui sont au niveau du langage, lors d'un travail en équipe, il est également important d'établir des règles afin que tout le monde produise un code semblable sur la forme. Dès lors, il est facile pour un développeur de se plonger dans le code d'un autre. En plus d'augmenter la productivité directe, cela augmente la facilité de maintenance, de débuggage et la l'introduction de nouveaux développeurs au projet.

Toujours dans l'idée d'un code lisible, il faut donner un nom correspondant bien à ce que fait la variable/méthode/classe/autre et surtout un nom explicite afin que le nom soit suffisant pour comprendre un morceau de code tiers sans devoir plonger dedans pour le comprendre. De plus, la subdivision en plein de sous-méthodes permet, lors d'un héritage, à la classe fille d'avoir un grand nombre de méthodes à redéfinir, ce qui augmente la granularité de la redéfinition. De plus, une méthode ne demande pas, elle ordonne. Si une méthode contient des conditions sur le type et la nature de ses variables pour adapter son comportement, c'est qu'il y a une mauvaise utilisation du polymorphisme. Plus précisément, pour obtenir une bonne découpe des méthodes et fonctions, il faut qu'une fonction fasse **une et une seule** action. Dès lors, lorsqu'une fonction contient deux (ou plus) parties distinctes (par exemple séparées par un double retour à la ligne ou par un commentaire décrivant le sous-bloc), c'est qu'il y a découpe à faire.

7.2 Couplage et cohésion

Il est souvent dit qu'un bon code (orienté objet) doit faire preuve de **couplage faible** et de **cohésion forte**. Le couplage entre deux objets est l'interdépendance entre ces deux objets et surtout la connaissance que l'un a de l'autre. Par exemple, soient trois classes A, B et C et trois objets a, b et c des mêmes types. Si la classe A contient un objet de type B et que la classe B contient un objet de type C, alors il faut que la classe A ne connaisse pas la classe C, qu'elle ne sache pas qu'il y a un objet de type C dans la classe B. Chaque classe doit connaitre le strict minimum sur les autres.

La cohésion par contre est une sorte de couplage interne : s'il y a une forte cohésion au sein d'une classe, c'est que tout ce qui la compose a une certaine cohérence. Il faut que les différents services qu'une classe peut fournir soient cohérents entre eux. Si ce n'est pas le cas, c'est fort probablement qu'il faut faire une classe intermédiaire (séparée, nested, descendante, etc.) Il faut donc faire attention car en tentant un couplage faible, on risque de finir avec des classes fourre-tout qui sont de cohésion faible également. Ceci est à proscrire. De plus, en poussant la cohésion à être la plus forte possible, on risque d'avoir de tous petits objets faisant très peu et devant constamment faire appel à tous les autres objets ce qui donne donc un couplage fort et une cohésion forte. Il faut trouver le juste milieu.