La communication comme négociation du sens : le cas du développement informatique

Ophir Paz, Kim Savaroche

LA COMMUNICATION COMME NEGOCIATION DU SENS : LE CAS DU DEVELOPPEMENT INFORMATIQUE.

Ophir Paz & Kim Savaroche

Ophir Paz est doctorant en sciences cognitives. Ingénieur logiciel expérimenté et enseignant en intelligence artificielle, il travaille sur les méthodologies de développement informatique à la lumière des recherches sur la cognition.

Kim Savaroche est doctorante en sciences cognitives. Ingénieure logiciel diplômée, elle se spécialise dans la vision par ordinateur en rapprochant les techniques de traitement d'images et la recherche fondamentale.

Résumé

À travers l'exemple du développement informatique, nous nous proposons de montrer que c'est dans la communication – entendue comme l'échange et la négociation – plus que dans l'imposition de convention ou l'abondance de documentation, qu'émerge une cristallisation du sens du code. Elle rend possible un entendement autour d'une inscription toujours en évolution, et permet ainsi de limiter les défaillances de la machine causées par des erreurs d'interprétations humaines.

Mots clés

Ingénierie logicielle, *refactoring*, interprétation, négociation, technologie cognitive.

Le développement informatique n'est pas un objet d'étude courant en communication. Il se définit comme le processus ayant pour but la production de logiciels par la conception, l'écriture et la maintenance du *code*. Ce terme désigne une inscription numérique, structurée selon une syntaxe rigoureuse, constituant une liste d'instructions à exécuter par l'ordinateur.

Le plus souvent, la description du programme attendu est consignée dans un document — les *spécifications* — qui sont transmis à l'équipe qui se charge du développement. Le logiciel produit est ensuite testé pour vérifier qu'il est bien conforme à

ces attentes. Dans l'acception traditionnelle, un bon programme informatique est donc celui qui induit un comportement de la machine correspondant parfaitement aux spécifications.

Il est déjà manifeste que ce cette procédure soulève des questions de communication classiques autour de la compréhension de ces spécifications par les différentes parties. Ce sujet est notamment adressé aujourd'hui par les méthodes *agiles,* qui préconisent par exemple la répétition des cycles de production raccourcis plutôt que de longues phases de spécification, de programmation puis de test. Nous nous intéressons ici à une autre problématique : le rôle central du code dans la communication entre développeurs.

Si le code a pour objet la production d'un logiciel conforme aux spécifications, l'ingénierie s'attache à des indicateurs de qualité différents : sa lisibilité et sa maintenabilité. Ces critères subjectifs sous-entendent une activité humaine centrale pour le processus de développement. L'objectif est ici de mettre en lumière que les échanges entre membres de l'équipe sont un enjeu clé de l'ingénierie logicielle.

Code informatique et interprétation

A priori, le programmeur donne des instructions à l'ordinateur sans ambiguïté sous forme de code. La machine exécute toujours ces instructions de la même façon. Comme lors de l'appui sur un interrupteur pour allumer une ampoule, ce processus ne laisse *a priori* de place ni pour l'interprétation ni pour la négociation.

Pourtant, dans cet exemple, l'interprétation est déjà là dans la conviction que l'interrupteur est actionnable d'une pression du doigt ou dans la présomption qu'il aura un effet sur l'ampoule grâce à un câblage dissimulé dans le mur. Et cette simple supposition peut déjà être erronée : il est par exemple courant d'appuyer sur un interrupteur entre deux pièces et de voir la mauvaise lampe s'allumer. Il est toutefois probable que le concepteur du système électrique ait sciemment positionné l'interrupteur en supposant que les utilisateurs anticiperont son effet sans se tromper. Malgré le caractère systématique du comportement d'une réalisation technique, la perception de son fonctionnement et de ses effets reste soumise à interprétation.

Le code informatique se matérialise par une suite d'instructions inscrites sous forme de texte sur un support numérique. Les mots-clés du langage vont agir comme autant d'interrupteurs qui vont déclencher une action du système. Ces actions vont se concrétiser par des flux électriques modifiant les états des différents composants de l'ordinateur, souvent trop complexes

pour être appréhendés par un développeur. Ce dernier associe à chacune de ces instructions basiques une interprétation correspondant à une action élémentaire. Par exemple une instruction *add* est interprétée comme une addition même si sa mise en œuvre passe par des opérations binaires le plus souvent ignorées.

Comme dans un texte classique, les mots utilisés, mais aussi leur agencement ou les symboles de ponctuations impactent l'interprétation du code. C'est ainsi le nom de l'instruction ainsi que le contexte dans lequel elle apparaît qui définissent la compréhension de son usage — comme une étiquette collée sur un interrupteur — indépendamment de ce qu'elle induit effectivement au niveau électronique.

Le rôle du développeur consiste donc à combiner un ensemble d'instructions données pour composer de nouvelles instructions (des *fonctions*) qui induisent des comportements plus complexes. À ce titre, celles-ci doivent être nommées et vont être à leur tour saisies par ce libellé par un autre développeur. Cet enjeu est majeur et constitue une des problématiques les plus courantes en ingénierie logicielle. Pour reprendre le cas des interrupteurs, l'étiquetage de celui qui "éteint toutes les lumières, baisse les rideaux et ferme la porte“ peut prendre plusieurs formes. Cela pourrait être "Fermer et éteindre" ou encore "Tout fermer", dans tous les cas il est difficile d'en déduire tous ses effets avant de l'avoir actionné. Il en est de même pour les fonctions dans le code informatique : comme seule leur mise en œuvre révèle leur résultat réel, ils doivent au mieux suggérer leur usage. L'échange et la négociation sont alors primordiaux pour trouver un nom et un agencement qui fasse sens pour tous les utilisateurs.

Ces fonctions servent elles-mêmes de briques pour en générer d'autres, ce qui aboutit à une superposition de couches dont les éléments s'éloignent de l'objet physique pour devenir des constructions dont le sens sera interprété en contexte. Si ces constructions permettent la réalisation de programmes informatiques complexes par composition, elles sont fréquemment à l'origine de dysfonctionnements causés par une mauvaise perception de leurs actions effectives.

Le disfonctionnement qui a causé l'explosion d'Ariane 5 en 1996 en est un exemple typique. Les nombres sont matérialisés par une valeur binaire (un ensemble de *bits*, chacun valant 0 ou 1) dans la mémoire de la machine. Chaque nombre se voit ainsi allouer un *espace mémoire* pour stocker sa valeur binaire, dont la taille va déterminer la limite de ce nombre. Un nombre positif encodé sur 8 bits est compris entre 0 et 255, sur 16 bits il peut

valoir entre 0 et 65536. Lors du décollage de la fusée, un nombre nécessitant 16 bits a été stocké dans un espace mémoire de 8 bits, il a donc été tronqué, faussant un ensemble de calculs et déclenchant une cascade d'erreurs qui a abouti à la destruction du système.

Une enquête a montré que le module fautif avait été validé sur Ariane 4 et réutilisé directement dans Ariane 5 – dont certaines mesures pouvaient être plus élevées. Cet exemple montre comment une simple affectation de valeur peut avoir des conséquences totalement inattendues.
Ce cas d'école a permis d'améliorer la fiabilité des fusées suivantes avec une méthode surprenante appelée *software redundancy* (redondance logicielle) : il s'agit de confier à des équipes de développeurs distincts des mêmes spécifications, ce qui permet d'avoir deux programmes induisant le même comportement mais avec des implémentations différentes. En cas de défaillance du premier système, le second prend le relais avec un jeu d'instructions différents, potentiellement plus robuste dans un contexte inattendu. Le fait que les mêmes spécifications et la même batterie de test peuvent correspondre à deux implémentations différentes est ici exploité pour la sécurité du système.

Fort heureusement, la plupart des disfonctionnements informatiques ne causent pas des explosions. Ils impliquent tout de même d'effectuer des opérations de maintenances, nécessitant de comprendre où ont eu lieu les erreurs d'interprétation.

Maintenance et négociation

Si le comportement d'un objet technique est souvent spécifié en amont, il doit régulièrement être modifié pour être réadapté, affiné, pour mieux correspondre à ses différents contextes de mise en œuvre.

Dans le cas d'un logiciel, prévenir les erreurs fonctionnelles est théoriquement possible en le testant de façon drastique, mais il n'y a pas de critères universels pour garantir la qualité du code qui a permis de le produire. C'est pourquoi la préoccupation principale de l'ingénierie logicielle n'est pas d'obtenir un comportement parfait — souvent temporaire — mais plutôt d'assurer la capacité du système à être maintenu et à évoluer.

Une solution souvent préconisée consiste à utiliser des conventions qui émergent dans les communautés de développeur. Celles-ci fournissent une nomenclature de base

commune et quelques critères de qualité quantifiables. Il existe ainsi des conventions typographiques (position des majuscules dans les noms), d'agencement (indentation) ou même parfois de complexité (longueur d'une fonction) qu'il convient d'adopter pour correspondre aux habitudes d'une équipe. Cependant, comme dans la langue, le respect de ces structures d'attente (Baker et al., 1998) permet l'échange mais ne garantit pas une compréhension mutuelle. Le code d'une fonction peut respecter toutes les conventions mais ne pas faire sens lors de la sa lecture par un autre développeur.

Dans le cas d'Ariane, la limitation de la valeur d'une mesure à 255 faisait sens dans la version 4, dont le développeur n'avait pas anticipé l'évolution dans l'itération suivante. D'un autre côté elle a probablement dérouté un programmeur de la version 5 qui n'avait pas anticipé cette limite lors de la réutilisation du module.

À travers ces opérations de maintenance se joue donc une forme d'entendement entre celui qui a écrit et celui qui vient relire et modifier. Lors de la conception s'opère une forme de *mise* (Guignard, 2012) sur le sens que va donner un futur programmeur au code. Inversement, la ressaisie implique une mise sur les intentions lors de la conception. Comme dans une conversation où les parties reformulent leurs propos jusqu'à s'accorder, le code est adapté pour mieux suggérer son fonctionnement dans son contexte d'utilisation.

Ce processus appelé *refactoring* consiste à modifier le code sans changer le comportement de la machine. Il peut s'agir par exemple de renommer un élément ou d'extraire un ensemble d'instructions pour en créer une nouvelle fonction. Comme la factorisation ou le développement en mathématiques, l'objectif est d'en ajuster la perception humaine sans avoir d'incidence sur le résultat. Par ces manipulations de réagencement et de renommage, le sens du code est modifié pour mettre en avant l'erreur ou l'évolution à apporter. La liste d'instructions est ensuite modifiée pour que le programme puisse satisfaire le comportement attendu.

Alors que dans certains domaines de l'ingénierie les opérations de maintenance sont compliquées à mettre en œuvre, la spécifié du développement logiciel réside dans la rapidité de réalisation des modifications. Le cycle d'évolution du produit est alors considérablement raccourci. Là où la modification de la position d'un interrupteur sur un mur nécessite au mieux plusieurs heures de travail, une transformation complexe du code est exécutée en quelques minutes grâce aux outils de développement modernes. Cela rend possible la révision d'une

même portion du code à plusieurs reprises par différents développeurs dans un période restreinte.

Néanmoins certaines évolutions qui paraissent simples au premier abord peuvent remettre en cause le paradigme de fonctionnement d'un programme en profondeur. Prenons une application de messagerie dans laquelle on souhaiterait rajouter la capacité d'échanger du texte écrit en gras. C'est une option courante qui est intuitivement relativement simple à mettre en place. Pourtant elle pose aux développeurs des problématiques de modification du protocole de communication des messages, de gestion des cas où un utilisateur dispose de cette fonctionnalité et l'autre pas, ou nécessite de vérifier que l'interface ait la capacité d'afficher du texte mise en forme. Le programme a-t-il été conçu pour permettre la transmission de contenu complexe ou pour l'échange de messages minimalistes en consommant un minimum de données ?

En fonction de l'organisation du code, cette évolution a pu être anticipée et être triviale à mettre en place, ou aller à l'encontre du schéma de fonctionnement du programme et nécessiter un travail conséquent. Plus qu'une mise sur l'interprétation du code par un autre développeur, c'est ici un pari sur le cours de l'évolution du logiciel qui est en jeu. Dans le cas d'un changement en profondeur, il est nécessaire de s'aligner sur une nouvelle interprétation qui advient au terme d'une négociation au sein de l'ensemble de l'équipe, supportée par l'inscription numérique.

Code et *Coding*

Interprété, modifié, le code est à la fois le moyen d'altérer mais aussi de faire sens du comportement du programme. Il est le lieu de la négociation car il constitue l'environnement dans lequel se cristallisent les termes qui permettent de penser le fonctionnement du logiciel et d'échanger sur ses évolutions.

Par exemple, la réalisation d'une évolution comme le rajout d'un bouton dépend de la facilité à modifier la génération de l'interface. S'il existe déjà une fonction permettant de créer un bouton, le développeur a déjà un outil à sa disposition dont il peut imaginer la saisie. Si au contraire une telle fonction n'a pas été mise en place au préalable, il devra baser sa réflexion sur une combinaison d'instructions données plus complexe pour parvenir à ses fins (dessiner un rectangle, afficher du texte, gérer une interaction avec la souris, etc.).

De même lorsqu'un développeur doit intégrer un calcul dans un logiciel, il en fait sens à partir des instructions qui sont accessibles dans le code et non des symboles mathématiques. Il existe par exemple des contraintes techniques limitant la valeur d'un nombre (illustré par l'échec d'Ariane 5) qui restreignent les opérations arithmétiques sur un ordinateur. Celles-ci sont ignorées en mathématique alors qu'elles sont perçues par un programmeur dès la lecture de la formule.

Le code n'est donc plus simplement le lieu d'inscription du résultat des négociations — un simple moyen de communication — mais fait partie intégrante de la cognition des développeurs. Il agit comme un prisme qui va structurer la perception des spécifications, du produit final, mais aussi des autres représentations gravitant autour du projet (Steiner, 2010).

Toutes les activités de négociation du sens présentées n'ont donc pas seulement pour intérêt de se mettre d'accord sur ce qui est inscrit et sur le sens à lui donner, mais aussi de cristalliser une perception plus globale de l'environnement de travail.

C'est donc sur la qualité cette activité — le *coding* — plus que sur celle du code lui-même que reposent les critères chers à l'ingénierie logicielle.

Le respect des critères déterminés par les conventions reste important pour constituer une base d'entendement, mais c'est toutefois cette capacité à soutenir la cognition qui garantit la pérennité du code. En cela la discussion, la négociation et le refactoring sont décisifs pour accompagner la complexification d'un programme.

Méthodologies

Il existe des méthodologies pour favoriser le *coding*. Elles sont parfois considérées comme contreproductives en entreprise car elle ne favorise pas la production quantitativement mesurable de nouveau code et lance régulièrement des débats animés au sein des équipes. Nous avons démontré précédemment à quel point ces échanges sont, de fait, fondamentaux.

Ces méthodologies sont apparues de façon empirique au sein de petites équipes ; elles se répandent et évoluent pour s'adapter à de nouvelles structures. Le caractère spontané de leur apparition atteste néanmoins de l'importance implicite du *coding* dans le développement.

La *code review* décrit la présentation par un développeur d'une partie de code qu'il a écrite ou modifiée à l'ensemble de son équipe. Ces échanges a posteriori permettent de s'accorder sur une interprétation et s'accompagnent souvent de refactoring pour refléter le résultat des négociations entre l'auteur et ses partenaires.

Le *pair programming* est une méthode où deux développeurs travaillent conjointement sur la même tâche. Un développeur écrit du code pendant que l'autre observe et fait des commentaires. En amont d'une *code review*, une négociation en temps réel est engagée qui amorce la cristallisation du sens par un partage déjà effectif entre deux membres de l'équipe. Favorisant l'échange, c'est également une pratique efficace pour l'intégration de nouveaux développeurs.

Enfin, le *mob programming* est une version étendue du *pair programming* : au lieu de deux développeurs, c'est l'ensemble de l'équipe qui traite une tâche en même temps. Si elle peut paraître déroutante, elle mobilise de concert les différentes interprétations des membres de l'équipe et permet une propagation accélérée de la connaissance pour résoudre des problèmes complexes. Il est possible d'y intégrer des membres non-techniques afin de confronter leur interprétation d'une problématique avec celle inscrite dans le code.

Conclusion

Au-delà de la création d'un logiciel répondant à des spécifications, le développement informatique est un processus centré sur l'activité humaine.

La perception du code implique toujours une interprétation de son fonctionnement, le plus souvent sans corrélation avec son action effective sur la machine qui l'exécute. Plus le système est complexe, plus il est le résultat d'une accumulation de ces interprétations.

Le travail en équipe nécessite un entendement sur ces différentes interprétations. C'est l'activité de négociation du sens qui permet aux différentes parties de s'accorder sur la signification des instructions.

Enfin en tant qu'outil technique, il est constitutif de la cognition des développeurs. Il devient un prisme sur la perception et sur les moyens d'actions dans son environnement de mise en œuvre. Il est donc à la fois le support de la négociation du sens et le lieu de sa cristallisation — qui assure une compréhension entre le

développeur auteur d'une partie du code et celui amené à la maintenir.

C'est cette forme de communication — le *coding* — plus que des critères de qualité objectivés, qui est la clé pour garantir la lisibilité et préserver la maintenabilité lors du processus de développement.

Bibliographie

Baker, C.F., Fillmore, C.J., Lowe, J.B., (1998). The Berkeley FrameNet project. COLING-ACL '98: Proceedings of the Conference, Montreal, Canada.

Clark, A. (2003). Natural-Born Cyborgs. Minds, Technologies and the Future of Human. Oxford/New York: Oxford UP.

J. J. Gibson, (1979). The Ecological Approach to Visual Perception. Houghton Mifflin.

Goody, J. (1979). La raison graphique. La domestication de la pensée sauvage. Les Editions de Minuit.

Guignard, J.-B. (2008). Le corps et l'idée ou les simulacres d'incarnation : le cas de la linguistique cognitive.

Guignard, J.-B. (2012). Les grammaires cognitives. Toulouse: Presses Universitaires du Mirail.

Rastier, F. (1987). Sémantique interprétative. Paris: Presses universitaires de France.

Simondon, G. (1958). Du mode d'existence des objets techniques. Paris: Aubier.

Simondon, G. (1989). L'individuation psychique et collective. Paris: Aubier.

Steiner, P. (2010). Philosophie, technologie et cognition. Intellectica (p.7-40).