Le paysage du développement

La pratique du développement est bien sûr largement influencée par le contexte technique et organisationnel dans lequel elle a lieu. Le domaine qui nous intéresse ici, celui du développement d'applications qui s'exécutent sur le Web en général, et plus particulièrement de la sous-catégorie que forment les métatechnologies, reste jeune et faiblement structuré : les méthodes et techniques y changent plus rapidement que dans le champ de l'informatique traditionnelle. Malgré une certaine tendance à la consolidation et à la maturation, le Web ne mérite pas (encore) d'être qualifié comme un espace techniquement et professionnellement figé. Certains usages et applications sont de fait devenus des piliers d'angle plutôt stables depuis des années (forums, pages statiques, CMS, etc.) mais une bonne partie du Web, dont le domaine des métatechnologies, reste peuplé de techniques et d'usages encore fortement expérimentaux. L'innovation et la conception s'y font largement, pour paraphraser Sylvie Lainé-Cruzel [2001, p. 20], « en aveugle » parce que le terrain des possibilités techniques reste largement ouvert et les usages fluctuants. L'expérimentation de nombre de nouvelles voies, dont nous pouvons témoigner aujourd'hui avec le Web 2.0, nous parait plus résulter d'une sorte de consensus collectif implicite entre développeurs et usagers, d'un accord informel pour progresser dans certaines directions plutôt que d'autres, que d'un effort planifié ou réfléchi par des chercheurs attachés à l'industrie ou au domaine académique. Le chaos expérimental qui marque toute technique dans sa jeunesse constitue cependant le milieu à partir duquel des éléments plus stables commencent à se dégager, des éléments qui représenteront par la suite des points de référence et influenceront fortement l'évolution de l'ensemble du domaine. Ce phénomène de structuration de l'avenir par le présent, que Rosenberg [1994] nomme la « dépendance au chemin » (*path-dependecy*), ne concerne pas seulement la technique, mais il s'y manifeste très visiblement. Une fois qu'une décision technique commence à se diffuser, il devient chaque jour plus difficile de dévier de la direction qu'elle fixe. En effet, elle

développe une force d'attraction en influençant son contexte technique (elle provoque l'apparition d'objets techniques porteurs ou dépendants d'elle-même) et social (des habitudes et « arts de faire » se stabilisent). Le chaos expérimental se consolide en un chemin spécifique dont il est difficile de diverger, c'est-à-dire qu'au sein du très large éventail des directions possibles se développe une structuration qui rend certaines voies bien plus attractives que d'autres.[165] En ce qui concerne les métatechnologies, le contexte technique et social reste encore assez ouvert, bien que certains standards commencent à émerger. Le paysage n'est pourtant pas encore découpé en parcelles. Le Web en général s'est certainement industrialisé mais il reste encore beaucoup de place pour des acteurs indépendants et innovateurs – le grand nombre d'applications qui sont sorties de la nuée du Web 2.0 confirme cette intuition.

Notre travail s'intéresse surtout à ces zones qui ne sont pas encore conquises par les grandes entreprises avec leurs lourds appareils bureaucratiques, stratégiques et méthodologiques. Autour de l'Internet, des modes de production se sont installés qui s'inspirent plus de la métaphore du *bazar* que celle de la *cathédrale* [Raymond 1998]. On peut y découvrir certains éléments d'une manière de concevoir le développement du logiciel en rupture avec les habitudes et façons de penser établies, qui ouvrent un certain nombre de pistes pour interroger, depuis un angle nouveau, notre horizon méthodologique. Le design orienté-société que nous allons présenter par la suite, dont nous pensons qu'il devrait complémenter les deux grands courants étudiés ici dans le cadre du développement de métatechnologies, repose en partie sur une appréciation spécifique du paysage de la création informatique. Cette perspective part de l'idée que cette création peut être vue non seulement comme une *ingénierie* mais aussi comme un processus d'*écriture*.

2.3.1 L'écriture informatique

Afin de comprendre ce dont il est question dans l'acte de développer un objet informatique, il est nécessaire de rappeler qu'il s'agit, en fin de compte, de produire un texte ; un texte bien particulier certainement, mais qui constitue néanmoins le résultat d'un geste d'écriture. Cette dimension nous semble essentielle et nous allons l'introduire comme base de notre raisonnement sur un design du logiciel inspiré par les sciences humaines. Avant de pouvoir avancer sur ce chemin, nous sommes obligés de montrer en quoi un programme informatique est une écriture, et de poser les conséquences

[165] L'interface graphique représente un bon exemple de *dépendance au chemin* en informatique. Depuis le développement du principe WIMP (*Windows, Icons, Menus, Pointing Device*) par *Xerox* à la fin des années soixante-dix et sa popularisation par *Apple* avec l'introduction du *Macintosh* en 1984, les éléments de base n'ont guère changé. Les différents essais de proposer d'autres types d'interface ont largement échoué et avec l'expansion continue de WIMP, il semble que seules des améliorations cosmétiques aient une réelle chance d'être acceptées.

qui découlent de ce fait. Une première indication se trouve dans la question difficile de savoir ce qui constitue la *conception* dans la réalisation d'un logiciel.

Conception : schéma ou code ?

Dans un article fortement débattu et ainsi devenu célèbre, Jack W. Reeves [1992] s'interroge, à partir de sa propre pratique de programmeur / développeur / concepteur / designer[166], sur les différents possibilités de définition du design logiciel (*software design*). Il choisit comme point de départ de son argumentation la comparaison entre génie civil et génie logiciel. Pour le premier, il y aurait séparation évidente entre la conception (le *design*) d'un objet (p.ex. d'un pont), qui prend habituellement la forme d'un plan technique (*blueprint*), et sa construction réelle en tant qu'objet matériel ; deux gestes clairement différents ressortent de cette distinction : conceptualiser (*to design*) et construire (*to build*). Ces deux activités ne diffèrent pas seulement par leur contenu pratique, mais concernent également des professions distinctes, l'architecte pour la première et l'ingénieur en bâtiment pour la seconde, et, de plus, elles se trouvent dans un rapport d'enchaînement temporel où l'une doit être achevée pour que l'autre puisse commencer. Selon Reeves, une telle séparation ne pourrait pas être établie si facilement en ce qui concerne le génie logiciel. L'auteur met en doute l'idée courante selon laquelle on peut distinguer, de façon analogue au génie civil, une conception sous forme de schéma, plan ou diagramme, représentée si possible en un langage conceptuel[167], de la réalisation ou implémentation du programme écrit en un langage de programmation. Son argument se base sur le constat que le « vrai » logiciel n'est pas le listing en langage de programmation, mais le résultat de la compilation[168] du code source, la suite des signes binaires, le seul code qui peut être exécuté par la machine directement. En conséquence, Reeves affirme que le *design* ou la *conception* d'un logiciel (l'équivalent du plan dans notre métaphore précédente) ne se réduit pas à la série de schémas ou de modèles qui préparent le travail d'écriture du code, mais comprend surtout et essentiellement le code source même, tel qu'il est écrit par un développeur. L'activité de conceptualisation comporterait donc le double geste de « planifier » le code d'une part et de l'écrire de l'autre, ainsi que la circulation inévitable entre ces deux pôles tout au long du projet. Pour l'auteur, la stricte séparation des processus de conception et

[166] Malgré certaines tentatives de stabilisation terminologique, il n'existe pas de consensus sur la dénomination des personnes qui *font du logiciel*. Nous pensons que la multiplicité des titres et définitions est elle-même un indice de la pluralité de modes de production qui s'est installée dans le domaine de l'informatique.

[167] Par exemple UML (*Universal Modeling Language*) ou *Merise* qui proposent (entre autre) une syntaxe normalisée pour modéliser des données et des traitements.

[168] On appelle « compilation » le processus de traduction d'un programme depuis le code source écrit dans un langage lisible par l'être humain (i .e. *C* ou *Java*) vers le code machine (ou binaire) lisible et exécutable par l'ordinateur. Un programme peut être compilé par le programmeur avant d'être distribué ou seulement juste avant chaque exécution (*Just-In-Time Compilation*).

d'implémentation ou de réalisation relève d'une posture artificielle qui ne correspond guère à la pratique de développement où les deux activités s'impliquent mutuellement. Dans une telle perspective, la phase de construction se résume, pour l'être humain, à l'acte de cliquer sur le bouton « compiler » dans l'environnement de développement utilisé, ce dernier se chargeant de traduire le code source en code binaire.[169] De ce changement de perspective il découle qu'un logiciel est très cher à conceptualiser (parce que le processus prend beaucoup de temps) mais quasiment gratuit à construire (parce que la machine fait le travail automatiquement).

Nous ne sommes pas convaincus que le texte de Reeves fournisse une réponse finale sur ce sujet, les méthodes et pratiques de développement étant dispersées en un trop un large éventail pour toutes les ramener à une même perspective, mais nous allons voir plus bas que certaines évolutions de l'environnement technique et méthodologique se comprennent mieux avec une telle approche, et ainsi la soutiennent. Pour l'instant, nous souhaitons souligner que Reeves nous ouvre une porte intéressante sur le processus de développement en tant que pratique d'écriture. Or, hors de la question de savoir quels gestes constituent la conception et quels la réalisation, la création d'un logiciel ou d'un système d'information nécessite toujours l'écriture d'un programme, d'un code source qui quittera son statut de pur texte par la suite pour devenir un « texte-outil » [Souchier et al. 2003, p. 25].

Latour [1992, p. 255] paraphrase Austin quand il constate que « comment faire des choses avec des mots et puis tourner des mots en choses est maintenant évident pour tout programmeur » (*how to do things with words and then turn words into things is now clear to any programmer*). La première partie de cette citation se réfère évidemment à la théorie des « *speech acts* » (actes de langage) [cf. Austin 1962] qui fut parmi les premières à thématiser, de façon conséquente, la dimension performative du langage. A partir de la citation de Latour, nous sommes tentés d'ajouter une catégorie supplémentaire à la liste des différentes modalités illocutionnaires qui regroupe, chez Searle [1975], l'assertion (expression sur un état de choses), la direction (faire faire quelque chose à l'interlocuteur), la commission (faire une promesse), l'expression (exprimer un état psychologique à propos de quelque chose), et la déclaration (acte expressif qui change le statut de quelque chose, par exemple la déclaration d'un mariage). Ce qui distinguerait un tel *énoncé informatique* des fonctions classiques du langage serait le fait que les mots qui le composent (le code source) peuvent être traduits en choses (le code binaire), c'est-à-dire en une machine logicielle qui s'exécute sur l'ordinateur, la machine universelle. Ne devrait-on pas parler d'un nouveau type d'énoncé dont la dimension pragmatique pourrait être qualifiée de « procédurale » ? Cette proposition s'accorde parfaitement avec un passage

[169] La perspective de Reeves est soutenue par le fait que le processus de compilation pour un projet qui comprend une série de fichiers de code source ou d'autres ressources est habituellement appelé « *to build* » (construire). Le plus connu des outils est certainement *make* sous *Unix*.

d'Alan Kay qui souligne l'importante différence entre l'écriture « traditionnelle » et celle qui se fait dans le geste de programmation :

> « Dans l'écriture de textes, les outils qu'on génère sont rhétoriques ; ils démontrent et convainquent. Dans l'écriture informatique, les outils qu'on génère sont des processus ; ils simulent et décident. » [Kay 1990, p. 193]

Les textes produits par l'écriture informatique entretiennent un rapport spécifique avec leur audience, rapport qui diffère de celui de l'écriture telle que nous la pratiquons depuis des millénaires : aux dimensions de l'interprétation et de l'acquisition de connaissances s'ajoutent celles de l'usage et de la délégation. Le texte ne rejoint plus l'action (le comportement) seulement à travers le détour par l'information (la cognition) ; en tant qu'infrastructure et agent il se dote d'un potentiel d'action direct.[170]

Les conséquences – caractéristiques du logiciel

La double nature du code informatique, le fait d'être écriture et machine en même temps, n'est pas sans conséquence pour la dimension économique de la production du logiciel. Quand on analyse le processus de création de n'importe quel produit industriel, trois phases se dégagent immédiatement : la conception, la production et la distribution.

Nous avons vu avec Reeves que l'étape de production d'un logiciel peut être comprise comme le processus mécanique de traduction d'un texte / concept (le code source) en un objet technique (le code binaire). Economiquement, le coût de cette phase intermédiaire s'avère quasiment nul et à la *construction automatique* (compilation) s'ajoute le fait que l'information numérique peut être copiée sans perte de qualité et avec très peu d'investissement énergétique, ce qui réduit aussi quasiment à zéro la production « matérielle » comme source de dépenses. De ce fait, le logiciel diffère fortement des machines traditionnelles parce que pour lui, la confrontation avec la *matière* entendue au sens de la physique, dimension indispensable de toute machine, se fait seulement au moment où le logiciel est exécuté sur un ordinateur. En amont, le logiciel est *information* et fait partie des phénomènes « à (très(très)) petite énergie » [Baltz 2003]. Le coût de production (dans le sens industriel) concerne donc surtout le matériel (*hardware*). Mais même ici, la réduction des prix et le fait qu'un ordinateur peut exécuter tous les programmes écrits pour le système d'exploitation qu'il utilise rend cette partie de l'investissement économique très faible comparée à celui qui accompagne les produits industriels classiques.

[170] Cette conception ne considère pas la question du texte « littéraire » qui ouvrirait à d'autres interrogations, en particulier en rapport avec le texte informatique. Cf. Balpe, Jean-Pierre : *Contextes de l'art numérique*. Paris : Hermes Science 2000

La deuxième dimension, celle de la distribution, est également affectée par la matérialité étrange, la *légèreté*[171], du logiciel. Depuis un moment déjà, le coût des supports matériels est plutôt faible. D'abord les disquettes, puis les CD et maintenant les DVD sont peu chers à produire et leur faible encombrement les rend très mobiles. Les programmes voyagent facilement. Avec l'Internet, il devient encore plus facile de « transporter » un logiciel du producteur au consommateur. La plupart des applications commerciales se vendent aujourd'hui en boîte dans un magasin ou par service postal ainsi que (souvent moins cher) par téléchargement direct. Encore une fois, les coûts tendent vers zéro.[172]

Cela nous reconduit au constat de Reeves qui affirme que les coûts dans la chaîne économique d'un logiciel se concentrent sur la phase première, la conception, qui consiste, selon lui, non pas seulement en sa modélisation mais surtout au laborieux travail d'écriture et de validation[173] du code. Mais aussi ici, y regarder de plus près s'avère payant. Qu'est-ce qui est cher dans la conception ? Les outils nécessaires ? Non, parce qu'un ordinateur à 500 € est plus que suffisant comme poste de travail pour un designer ou un programmeur. L'espace immobilier ? Pas vraiment non plus, parce que la programmation est un « office work » (travail de bureau) [Turing 1948, p. 4] qui ne demande que des bureaux suffisamment grands pour mettre les postes de travail. Non, ce qui coûte cher dans la conception de logiciels, c'est la main d'œuvre : les personnes et leurs compétences. En ce sens, *le temps* devient en fin de compte la contrainte et donc le facteur de coût principal. Le temps de modéliser, de coder et de déboguer / valider, le temps d'acquérir les connaissances nécessaires, le temps de se ressourcer.

Encore une fois, les similarités avec le processus d'écriture sont frappantes. La production (l'impression) d'un livre n'est pas très chère comparée à celle des produits industriels de complexité comparable et pour les textes électroniques, il devient quasi-nul. La distribution de l'imprimé est certainement plus coûteuse que l'envoi de l'information numérique par des câbles, mais d'un point de

[171] Jeanneret et Souchier [1999, p. 99] font bien de nous rappeler que le caractère éphémère du texte – et on pourrait donc dire de toute information symbolique – ne devrait pas donner lieu au « phantasme d'immatérialité ». Nous préférons donc parler d'une « légèreté » du logiciel plutôt que d'une immatérialité.

[172] Bien évidemment, nous ne prenons pas en compte ici les coûts « cachés », c'est-à-dire les immenses investissements dans les infrastructures, les câbles, satellites, etc. Ces dépenses sont pourtant distribuées sur l'ensemble des internautes et contribuables et une fois mis en place, les coûts d'usage de ces infrastructures sont minimes.

[173] L'activité de correction / validation et de test, qui prend toujours plus de place dans le développement du logiciel – et notamment du *open source* / logiciel libre où un petit nombre de développeurs est entouré d'un très grand nombre de testeurs – montre que la séparation entre production et conception est artificielle ; il vaudrait mieux penser leur articulation sous forme circulaire. Dans la pratique de la programmation, des lignes de codes sont écrites, le logiciel est « produit » (compilé), son exécution observée et, selon la satisfaction que procure le résultat, le développeur revient sur ce qu'il vient d'écrire ou passe à l'étape suivante. En outre, le programme doit être validé dans son ensemble – beaucoup d'erreurs se montrent seulement lors de l'interaction entre les composants – et en ce qui concerne sa compatibilité avec les différents plateformes (matérielles et logicielles) ciblées. La boucle entre production et conception, tel qu'elle caractérise la correction / validation, porte son propre coût parce que la recherche et la correction d'erreurs peut être extrêmement longue et difficile.

vue historique, le livre est quand même un objet qui voyageait bien facilement et le succès des librairies en ligne montre qu'il reste, au 21ème siècle, un moyen de distribution pratiquement et économiquement intéressant. Le grand investissement dans l'écriture – aujourd'hui plus que jamais – vient du temps nécessaire pour produire quelque chose de valeur pour d'autres gens. Ce temps ne comprend pas seulement l'activité d'écriture proprement dite mais aussi celui d'acquérir les connaissances culturelles nécessaires, de trouver de l'inspiration, etc.

Une autre caractéristique importante du logiciel, qui montre encore une fois sa similitude avec l'écriture, tient à sa nature essentielle de *tentative*. Un texte n'est jamais fini au sens qu'il ne pourrait plus être repris afin de le continuer ou de le modifier. Certes, une machine mécanique nécessite également qu'on y revienne et on peut même imaginer qu'elle est perfectionnée avec le temps, mais dans le cas du texte, la « clôture » (*closure*) qui transforme l'objet en cours d'élaboration en objet fini [cf. Feenberg 2003] n'obéit pas à la même logique. Le code informatique peut être mis à jour, imbriqué dans d'autres logiciels et étendu avec une facilité qui n'a pas à se soucier des propriétés physiques de l'objet et du coût de sa production ; un clic suffit et la nouvelle version est prête à être déployée. La morphologie particulière de la *genèse* du logiciel [cf. Simondon 1958], due à sont statut entre les mots et les choses, montre qu'il ne fait pas partie du même phylum que la technique telle que nous la connaissons depuis la révolution industrielle.

Ces caractéristiques élémentaires de l'écriture du code informatique se mêlent aux évolutions du contexte technique et social qui forme l'environnement où la genèse des objets se réalise.

2.3.2 L'environnement de l'écriture

L'espace d'où vient le texte n'est jamais vide, mais toujours déjà peuplé. Un roman qui vient de paraître s'insère dans un contexte culturel et surtout littéraire qui joue le rôle d'un bassin de sens duquel il se nourrit et qui va permettre au lecteur de le *comprendre* en le re-situant dans son environnement.

> « Tout texte est un intertexte ; d'autres textes sont présents en lui, à des niveaux variables, sous des formes plus ou moins reconnaissables : les textes de la culture antérieure et ceux de la culture environnante ; tout texte est un tissu nouveau de citations révolues. » [Barthes 1973]

Le texte informatique ne s'inscrit pas seulement sur le niveau du sens et de l'interprétation comme le texte littéraire, mais surtout sur celui du fonctionnement et de l'exécution qui constitue sa véritable finalité. L'interrelation permanente d'un code aux autres codes, qu'on pourrait appeler « intertextualité informatique », n'est donc pas seulement *textuelle* au sens où chaque programme contient des

éléments de programmes antérieurs ; elle devient *opératoire* au moment où un programme fait appel à des fonctions qui lui sont proposées par d'autres codes, par exemple par le système d'exploitation, le langage de programmation, les bibliothèques utilisés, les *frameworks*[174], etc. L'intertextualité informatique implique donc aussi une *interopérabilité* et bien que ces deux dimensions interagissent, elles ont néanmoins des façons différentes de se lier au *déjà là*, à ce qui a été dit / écrit et à ce qui a été construit. L'une implique une référence à l'histoire du texte tandis que l'autre concerne l'histoire de la machine.

La relation textuelle

L'intertextualité informatique prend forme à travers les liens qui se forgent entre un texte / programme et ses prédécesseurs, des liens qui fondent une présence de ces derniers dans le code en question. Cette présence se réalise par les mêmes mécanismes que Gérard Genette [1982] identifie pour la littérature : citation, allusion et plagiat. Le réservoir des codes source librement disponibles est énorme ; on en trouve depuis longtemps dans les manuels, les magazines professionnels et, depuis l'arrivée de l'Internet, cette disponibilité est quasi-infinie. Le *copier-coller* est un geste fondamental dans la programmation et il est très commun de s'inspirer d'autres programmes pour trouver ses propres solutions. À la différence des domaines de l'ingénierie classique, les idées et savoirs ne se propagent pas seulement par la voie de l'inspiration et de l'apprentissage, mais aussi par ce geste d'intégration directe qui peut considérablement accélérer le processus de design. Contrairement à la culture littéraire, la réutilisation et l'intégration du code récupéré ne sont pas vues comme spécifiquement problématiques parce que l'originalité d'un logiciel se fonde bien plus sur son approche générale que sur la singularité du *listing*. Les contraintes imposées par le fait que tout langage de programmation est un langage formel qui ne permet pas de formulations ambiguës et n'accepte qu'une syntaxe parfaite, impliquent que le nombre de solutions à un problème spécifique est fini. Prenant en compte que le répertoire rhétorique des langages de programmation est très limité[175], il devient évident que la répétition des mêmes lignes de code est un phénomène ubiquitaire, inévitable et, par conséquent, accepté.

[174] Un *framework* peut être défini comme une série de bibliothèques de codes, ciblées sur un espace d'application précis, qui permettent de programmer à un niveau d'abstraction élevé parce que la « technicité » du travail est déléguée aux formes et fonctions qui sont fournies. En utilisant un *framework*, un programmeur peut donc gagner en robustesse et en vitesse, en perdant de la flexibilité.

[175] Les fonctions de base d'un langage de programmation se réduisent à une petite poignée de fonctions : la manipulation de la mémoire pour travailler avec des données sous forme de variables ; les boucles pour faire des itérations sur ces variables ; des tests pour le branchement conditionnel du déroulement de l'exécution ; la gestion de l'input et output (des entrées / sorties) du programme. C'est à partir de ces éléments de base que tout logiciel est produit.

Cependant, Genette ne mentionne pas seulement l'intertextualité comme principe de relation d'un texte aux autres ; mais en propose deux autres, la métatextualité et l'hypertextualité, qui méritent un examen plus fouillé. En ce qui concerne la *métatextualité*, c'est-à-dire le commentaire, propre surtout au geste critique, nous ne pensons pas que les langages de programmation la permettent en tant que telle. Car il s'agit de constructions linguistiques destinées à l'exécution par une machine, et non pas à l'interprétation par un être humain. Il existe pourtant une forme de *critique effective* telle qu'elle est opérée par les métatechnologies qui se penchent sur des flux d'informations et de communications en les structurant, filtrant, traduisant. Mais ici, le programme ne se réfère pas à d'autres programmes, mais à des données ; quand il est exécuté, il n'est plus texte, mais déjà machine. La relation n'est plus textuelle mais opérationnelle. En ce sens, la dimension métatextuelle se réalise hors du médium de l'écriture informatique, dans le langage courant des débats de programmeurs qui commentent et critiquent les travaux des autres.

Le rapport *hypertextuel* pourtant se traduit bien dans le monde de l'écriture informatique. Dans le contexte des élaborations théoriques de Genette, cette troisième dimension est implicite au sens où il n'y a pas une présence directe d'un texte dans un autre, mais une relation plus vague qui peut néanmoins être très forte. L'exemple canonique est celui d'*Ulysses* de James Joyce dont le principe même de son écriture était de reprendre certains points de la structure narrative de l'*Odyssée* d'Homère. Dans le cadre de notre étude des relations entre textes informatiques, on pourrait interpréter l'hypertextualité comme une relation structurale sur le plan des idées et du savoir. Chaque programme contient une stratégie, une idée de comment résoudre un problème, un récit porteur d'un savoir. Il élargit l'imagination technique en ajoutant à l'espace du possible (ce qui peut être fait) et présente en même temps des instructions sur la méthode à employer (comment le faire). Chaque programme repousse donc un peu plus loin la frontière du connu et crée ainsi la base pour les prochaines innovations. Ces stratégies et approches trouvent leurs places dans les forums, manuels et cours d'enseignement et se propagent ainsi comme les fameux « *memes* » de Richard Dawkins [1992].

Ces trois rapports d'intertextualité nous intéressent surtout parce qu'il est important pour notre argument de montrer que l'écriture informatique se fait dans un espace *plein*, peuplé d'autres textes, stratégies et idées ; « le texte est un tissu de citations, issues des mille foyers de la culture » nous dit Barthes [1984, p.67]. De façon analogue à l'espace littéraire, les domaines des connaissances techniques se structurent et stabilisent avec le temps. Pas vraiment selon les axes de l'époque ou du genre, mais selon une série de paramètres qui classent les programmes en groupes et sous-groupes, qui transforment l'espace en territoire ou, dans la terminologie de Genette, ajoutent au texte du *paratexte*

et de l'*architexte*[176]. L'ordre se fait jour dans le chaos et les connaissances et savoirs dégagent des formes stables. Un jeune développeur peut découvrir cet espace à travers les manuels de synthèse, les sites ou forums sur l'Internet et la lecture du code mis à disposition par d'autres personnes. L'informatique est toujours un domaine jeune, mais nous sommes loin des premières décennies où tout problème était inédit. Sur le pur plan de la technique, il y a déjà un vaste terrain de problèmes résolus, de stratégies prouvées et d'astuces qui continuent à fonctionner. Pour trier une série de données, il n'est pas nécessaire d'inventer : chaque étudiant en informatique apprend l'algorithme *quicksort* de C.A.R. Hoare qu'il suffit de reprendre parce qu'il est fait pour être repris. L'intertexte constitué par ces formes rhétoriques est assez bien structuré et accessible ; une boîte à outils ouverte au seul prix du temps de l'apprentissage nécessaire. Mais la stabilisation des savoirs et des connaissances ne se fait pas seulement par le biais du jeu de relations entre les textes. Elle avance aussi par un mécanisme propre à la technique : le *blackboxing* opératoire.

La relation opératoire

Les rapports intertextuels entre les programmes sont un élément important pour comprendre comment un espace vide et ouvert se peuple et se transforme en paysage ; comment les domaines se mettent en place et les courants se démarquent entre eux. Mais dans le cadre de la technique – et dans l'informatique de façon encore plus poussée – il existe un autre type de rapport entre les « individus techniques » [Simondon 1958], que nous caractérisons d'*interopérabilité*. Nous avons vu que l'intertexte joue le rôle de réservoir de fragments de code, mais aussi de stratégies, de structures, de figures rhétoriques et de savoir-faire. La technicité du texte informatique permet pourtant quelque chose de formidable : les fonctions d'un programme peuvent être utilisées par un autre programme, sans qu'il soit nécessaire de connaître le fonctionnement du premier. Cela permet de *citer* le travail de quelqu'un d'autre d'une façon *opératoire* plutôt que strictement *textuelle*. Le texte / programme cité peut rester une boîte noire dont il suffit, pour l'utiliser, de connaître les entrées et les sorties. Le fait que ce geste de *blackboxing* fasse partie de la nature du logiciel ainsi que le constat que les *modules*[177] qui en résultent sont reproductibles sans le moindre coût sont certainement l'une des raisons du succès de l'informatique.

[176] Notons que le terme « architexte » chez Genette ne porte pas le même sens que chez Jeanneret et Souchier. Chez le premier, il désigne la « pure appartenance taxinomique » [Genette 1982, p. 12], c'est-à-dire l'appartenance à un genre, tandis que les derniers l'utilisent pour désigner des dispositifs techniques de l'écriture : « le texte naît de l'architexte qui en balise l'écriture ». [Jeanneret / Souchier 1999, p. 103]

[177] Ce qui constitue un *module* peut varier selon l'environnement de développement. Nous l'entendons ici comme un ensemble délimité qui propose une série de fonctions à utiliser dans le contexte d'un autre programme.

L'exemple de base de cette logique de la boîte noire sont les langages de programmation eux-mêmes. Plus personne n'est aujourd'hui capable de programmer un logiciel moderne directement en mode binaire comme c'était la norme aux tout débuts de l'informatique. L'écriture du logiciel se fait à travers des langages formels dont la syntaxe est un mélange de notations mathématiques et de mots anglais. Un logiciel spécifique – le fameux compilateur – traduit le code source en code binaire, compris par la machine. Les premiers langages de programmation appelés « assembleur » (*assembly*)[178] ou « code machine symbolique » (*symbolic machine code*) reprenaient directement le fonctionnement logique de l'ordinateur, mais assez vite – vers le milieu des années cinquante – des langages plus abstraits comme le *Fortran* apparurent. Ces langages « cachent » des séries de commandes derrière un seul mot de leur syntaxe (une instruction) ce qui rend la programmation bien plus facile, au prix du sacrifice d'un peu de vitesse d'exécution. Des langages toujours très proches de la machine comme le *C* sont aujourd'hui complétés par des langages où de plus en plus de tâches sont « déléguées » au compilateur, par exemple la gestion de la mémoire ou l'accès aux périphériques. Dans ce cas, on peut dire que le langage contient déjà des fonctions qui ne sont pas réalisées sous forme de matériel *mais apporte des fonctionnalités qui lui sont propres*. Ces fonctionnalités sont utilisées comme des boîtes noires parce que le programmeur ne doit par connaître et, sauf rares exceptions, ne connaît pas le « texte » et les commandes qui se cachent sous la surface. Des langages de script, qui ne sont pas *compilés* mais *interprétés*[179], contiennent souvent un grand nombre de commandes qui regroupent des fonctions assez compliquées. PHP[180] par exemple propose des centaines de commandes pour faciliter des tâches complexes dans le contexte de la programmation Web. Une fonction qui nécessiterait cent cinquante lignes de code en assembleur pourrait se réaliser en trente lignes en C, en quinze en Java et en cinq en PHP. Cela fait bien sûr gagner du temps et avec les machines actuelles, la perte de performance devient souvent négligeable[181].

Ce premier niveau d'assistance fonctionnelle par boites noires est complété par l'immense nombre de bibliothèques disponibles pour tous les langages. *Java* par exemple est livré avec des milliers de fonctions qui ne font pas directement partie du langage mais peuvent être utilisées comme si elles

[178] L'assembleur est plus vieux que l'ordinateur moderne. Lady Ada Lovelace, l'assistante de Charles Babbage, l'utilisait déjà pour noter plus facilement les instructions pour la « *difference engine* » (machine à différences).

[179] Le sens du mot « interprétation » dans le contexte informatique n'est pas le même que dans la langue courante. Il implique que le programme n'est pas traduit directement et entièrement en langage machine comme dans le cas de la compilation, mais que la syntaxe de chaque ligne de code est d'abord vérifiée, puis exécutée, avant de passer à la prochaine ligne.

[180] Ce langage *open source*, écrit en *C*, a fait son apparition en 1994 et aujourd'hui, PHP (acronyme récursif : PHP : Hypertext Preprocessor) est l'un des langages les plus utilisés pour la programmation Web.

[181] On a montré que *Java* pouvait atteindre 80% des performances de *C* dans le portage d'un Jeu, *Quake 2*, et la structure de PHP, basée sur des extensions écrites en *C*, permet l'élaboration de systèmes de très grande taille et néanmoins très performants.

l'étaient ; il suffit de les charger au lancement du programme. Pour afficher une fenêtre d'interface, il suffit de quelques lignes de code, bien qu'il s'agisse d'un objet complexe qui repose dans la mémoire de la machine sous forme de milliers de zéros et de uns. Les bibliothèques proposent donc des objets standardisés dont la complexité disparaît dans la boîte noire. Le principe du *framework* pousse cette logique encore plus loin en proposant un ensemble de bibliothèques pour un contexte d'application bien précis. *Ruby on Rails*, un *framework* de programmation Web récent mais déjà très populaire, permet par exemple de développer des applications avec facilité et rapidité parce qu'il combine le langage de script *Ruby* avec de très nombreuses fonctions prédéfinies, directement orientées vers les possibilités du Web.

A la différence de ce qu'écrit Simondon [1958, p. 23f] sur la technique « classique », l'informatique ne se développe pas nécessairement de l'abstrait (l'objet composé de composants individuels) vers le concret (l'objet composé de composants spécifiquement conçus pour la machine en question et formant un tout cohérent), mais dans un mouvement d'oscillation perpétuel. Les modules sont optimisés en permanence mais l'intégration ne dépasse que rarement un certain seuil ; *le souci de réutilisation et de polyvalence importe plus que le gain de quelques petits pourcents de performance*. Un logiciel plus *concret*, qui ne fait aucun usage de bibliothèques ou de modules est certainement plus léger en consommation de mémoire et – dans les mains d'un développeur expérimenté – plus performant en vitesse d'exécution qu'un logiciel plus *abstrait*, parce qu'il ne comprend que le stricte nécessaire et ne se surcharge pas de fonctions apportées par ces boîtes noires qui restent, pour la plupart, inutilisées. En même temps, la durée de développement et donc le coût du logiciel concret seront beaucoup plus importants parce que toutes les fonctions qu'on aurait pu prendre d'une bibliothèque doivent être programmées, testées et validées. La tendance actuelle, surtout dans le contexte du Web, pointe plus vers l'élaboration et l'usages de modules complets et réutilisables que vers la réalisation de systèmes plus légers et mieux intégrés mais finalement moins flexibles et plus coûteux. Quand Simondon [ibid., p. 25] souligne que « logiquement plus simple, [l'objet technique abstrait] est techniquement plus compliqué, car il est fait du rapprochement de plusieurs systèmes complets », nous sommes tentés de suggérer que le travail de construction d'un logiciel se résume, en grande partie, à la mise en place d'une logique, c'est-à-dire du code. La facilitation sur le plan logique qui est l'effet d'une approche plus abstraite (modulaire) compense donc largement le poids supplémentaire par rapport à un développement équivalent issu d'une approche plus concrète (intégrée). La concrétisation aide à maîtriser les aspects plus proprement « techniques » dans le contexte des exemples mécaniques choisis par Simondon, comme l'échauffement des composants ou la porosité des matériaux pour le moteur de voiture, parce que chaque composant prend plusieurs fonctions à la fois (p.ex. la stabilisation physique et l'évacuation de chaleur) ce qui réduit le poids, la complexité, le coût et les possibilités d'erreur. A cause de sa *légèreté* ces problèmes ne concernent guère le logiciel et même les questions de performance d'exécution ne touchent qu'une petite partie de l'informatique, les vrais coûts venant du temps de travail des développeurs plus que du temps de

calcul. Il est généralement moins cher d'ajouter un serveur que de réécrire un logiciel pour le « concrétiser » et le rendre plus performant.

Il est intéressant de remarquer que Simondon rapproche les objets abstraits des modes de production artisanaux [ibid, p. 31] et les objets concrets de la conception et fabrication industrielle. Déplacée vers notre terrain, cette conception impliquerait que le développement du logiciel serait plus proche de l'artisanat que de l'industrie. Nous ne sommes pas seulement d'accord avec cette évaluation, nous pensons même avec Dyson [1998] que cela constitue une chance et pas nécessairement un problème. La porte d'entrée vers l'artisanat et certainement plus ouvert que celle qui va vers l'industrie et le principe de participation que nous allons présenter plus bas dépend de cette facilité d'accès.

Avec les langages de programmation, les environnements de développement, les bibliothèques et les *frameworks*, nous sommes bien évidemment en plein dans ce que Jeanneret et Souchier [1999] nomment « architexte ». Nous avons montré plus haut comment on peut utiliser ce terme qui parle de ce texte qui *«* balise l'écriture *»* pour comprendre comment les logiciels influencent les pratiques auxquelles ils donnent lieu. Nous pouvons voir maintenant que les créateurs et programmeurs se trouvent également entouré d'architextes qui leur proposent un répertoire rhétorique ou poétique dont ils peuvent faire usage ou non. Loin d'être un génie solitaire, l'auteur de l'écriture informatique est aujourd'hui entouré par le travail d'un très grand nombre d'autres personnes avec lesquelles il dialogue à chaque ligne de code qu'il écrit :

> « Les choses n'existent pas sans être pleines d'hommes, et, plus elles sont modernes et compliquées, plus les hommes y pullulent. » [Latour 1993b, p. 33]

L'interopérabilité ne se réduit pas à l'intégration de fonctions programmées par d'autres dans son propre programme, mais elle implique aussi la communication avec d'autres logiciels pendant son exécution. Une application Web par exemple pourrait communiquer avec un serveur SQL pour lire des données stockées, télécharger des informations prises sur un autre site par un flux XML, envoyer un email par le service SMTP installé sur la machine et lancer le service de sauvegarde du système. Un ordinateur ordinaire constitue aujourd'hui l'habitat de milliers de programmes dont un certain nombre tourne en permanence quand la machine est allumée. Quelques-uns ne savent que remplir leur tâche bien spécifique, mais d'autres proposent des fonctions à leurs collègues. C'est un paysage de coopération qui déplace la tâche de programmation de l'écriture de fonctions vers l'écriture d'interactions avec d'autres logiciels qui savent déjà faire ce qui est souhaité. Le développeur ressemble de plus un plus à un organisateur qui gère les flux entre un *spécialiste* logiciel et ses semblables. Lucien Sfez a bien remarqué cette nouvelle donnée :

> « Avec les nouvelles technologies, le technicien est un programmateur et se rapproche donc du concepteur classique, l'exécutant disparaît peu à peu du paysage technique. La machine se charge d'accomplir la tâche de l'exécutant, elle est programmée pour cela. La technicité des opérations techniques bascule du coté de la logistique, de la conception intellectuelle et donc d'un savoir livresque. » [Sfez 2002, p. 57]

Cette remarque nous paraît extrêmement lucide et elle indique déjà la direction de l'argument que nous sommes en train de construire par une suite de petits pas laborieux. Nous pensons que le terrain du développement informatique est en train de changer profondément et le paysage qui se dessine à l'horizon nous semble plus ouvert à l'interfaçage avec les sciences humaines que jamais.

Les nouvelles plateformes

Dans les débats concernant le Web 2.0, apparaît souvent le mot *plateforme*. Dans ce contexte précis, il caractérise le développement technique propre au Web, mais nous pensons que ce terme est bien utile à l'entendre en un triple sens, dont seul le premier transparaît dans le terme Web 2.0.

Selon certains commentateurs [p.ex. O'Reilley 2005], le Web serait devenu une *plateforme* au sens où il n'est plus nécessaire d'installer un logiciel chez le client pour proposer un service. Des moteurs de recherche, des clients *webmail*, des logiciels collaboratifs et d'autres applications Web tournent essentiellement à l'intérieur du navigateur. Cela introduit un changement important de plus dans le processus de production du logiciel dont nous avons discuté plus haut. Le problème de la distribution s'est déjà beaucoup simplifié pour le logiciel en général par rapport aux produits matériels, grâce à la possibilité d'envoyer des supports bon marchés ou de proposer un téléchargement. Le Web en tant que plateforme va encore plus loin en éliminant complètement le processus de distribution au sens classique. Toute application Web est basée sur le principe client-serveur où une partie du code seulement tourne chez le client, une tranche variable du traitement étant effectuée côté serveur. Le logiciel est « distribué » à nouveau, chaque fois qu'on fait appel à lui, car il n'a pas de présence permanente sur le client. De ce fait des cycles de mis à jour extrêmement rapides[182] deviennent possibles, ce qui transforme la logique de produit en logique de service ; un service en permanente évolution, qui peut évoluer en fonctionnalité sans aucune intervention sur le poste du client. Du point de vue du développement, ce changement permet d'agir de façon beaucoup plus flexible, et de conserver un fort contrôle sur l'évolution du logiciel parce qu'en principe, il n'en existe qu'une seule version à gérer, celle qui tourne sur le serveur.

[182] Certains services, par exemple *Flickr* (http://www.flickr.com), ont adopté une fréquence de mise à jour qui va jusqu'à des intervalles de seulement 30 minutes entre deux versions. Un moteur de recherche comme *Google* est aussi en évolution permanente parce que les index sont vérifiés et élargis en permanence. La clôture dont parle Feenberg [2003] disparaît complètement dans une telle situation et cède la place à une co-évolution entre outil et usage ou l'un stimule l'autre.

Le deuxième sens de *plateforme* se réfère à la modularité croissante des langages et des environnements de développement. Le processus de programmation Web ressemble aujourd'hui bien plus à un jeu de *Lego* qu'à l'activité mathématico-logique des débuts de la programmation. Les outils en sont arrivés à un degré d'intégration tel que la création d'un système d'information ou de communication (de véritables *média* informatiques) est devenu plutôt simple, ce qui explique en partie l'arrivée en force de non-diplômés dans les professions de l'informatique et parmi les développeurs. Lorsqu'on distingue, avec Jeanneret et Souchier [1999], deux niveaux distincts dans l'écriture informatique, le calcul formel et mathématique « en bas » d'une part et l'ensemble des signes affichés sur l'écran d'autre part, c'est-à-dire un niveau logique et un niveau sémiologique auquel le premier donne naissance, on doit aussi prendre en compte les niveaux intermédiaires qui s'installent entre les deux. Des outils de création comme *Macromedia Flash*[183] et des langages de programmation toujours plus « verbaux » et symboliques, nous orientent vers une direction ou l'écriture informatique ne ressemble guère à la notation mathématique mais bien plus au dessin d'un plan en architecture. Vue de là, l'informatique sous-jacente et constitutive du Web actuel donne lieu à une plateforme de création qui ouvre un espace large et divers de possibilités, depuis des pratiques plutôt simples et ludiques jusqu'à la création lente et laborieuse de composants hautement concrétisés. Le degré élevé d'abstraction, au sens de Simondon, ouvre la porte à la participation de nouvelles populations qui peuvent acquérir une *culture technique* [cf. Simondon 1958], par le bricolage et l'improvisation, culture qui peut évoluer à la vitesse de chacun.

Nous pouvons finalement parler d'une *plateforme sociale et communicative,* en observant que le Web en tant que structure d'information et de communication est aussi devenu un environnement d'échange qui encadre le développement selon plusieurs axes. On y trouve déjà des fragments de code, des bibliothèques, des tutoriels et d'autres contenus tout-faits, mais on y rencontre surtout une communauté ouverte et animée qui aide à résoudre quasiment tout problème technique imaginable. Il est difficile de mesurer les changements apportés par la mis en réseau des informaticiens, des *hackers* et des amateurs de la technique dans la pratique du développement. Mais surtout pour les débutants, l'intelligence collective et réactive de la communauté est un outil d'apprentissage remarquable.

Ces trois éléments sont en train de bouleverser le développement du logiciel dans le domaine du Web mais aussi dans les secteurs plus classiques. La création technologique est aujourd'hui plus accessible que jamais et la vitalité du développement Web et de l'*open source* montre que les anciennes frontières strictes entre les ingénieurs et les usagers commencent finalement à devenir

[183] Des « authoring tools » comme *Flash* ou *Director* permettent de créer des objets « interactifs » simples à travers une interface graphique, sans devoir écrire de code (ce qui reste pourtant une option).

poreuses. Depuis la révolution industrielle, le mariage entre les sciences et les techniques n'a pas cessé de se renforcer [Neirynck 2005, p. 140]. Voyons-nous aujourd'hui les premières fissures dans cette alliance ? Il est bien trop tôt pour l'affirmer, mais plusieurs exemples montrent que les modes de développement du logiciel peuvent prendre des formes inédites, innovatrices et porteuses d'inspiration.

Open source[184]

Nous avons déjà dit plus haut que la réalité du développement logiciel diverge souvent beaucoup des méthodes et processus décrits et recommandés dans les manuels informatiques. En décrivant la pratique telle qu'elle a réellement lieu, on parle d'une activité « désordonnée, ad-hoc et non-théorique » (*messy, ad hoc, atheoretical*) [Coyne 1995, p. 32], qui consiste en « bricolage, heuristique, sérendipité et improvisation » (*bricolage, heuristics, serendipity, and make-do*) [Ciborra 2004b, p. 19] et qui est le résultat d'un « anarchisme méthodologique et théorique » (*methodological and theoretical anarchism*) [Monarch et al. 1997, p. 337]. Paul Graham [2003] rapproche les créateurs de logiciel des peintres, et Freeman J. Dyson [1998] constate que malgré l'arrivée de poids lourds comme Microsoft, le *software* reste un domaine artisanal où l'important sont les compétences des individus plus que les méthodes structurées et structurantes. Il semble bien que le projet d'exorciser l'approximatif et l'improvisation dont le génie logiciel se faisait le fer de lance n'ait jamais vraiment réussi et tous les auteurs que nous venons de citer affirment que cet échec est finalement plutôt positif.

Le mouvement *open source*[185] nous paraît un excellent exemple pour montrer comment des modes de production nouveaux peuvent conduire à des logiciels de grande qualité. À un premier niveau, le terme « *open source* » se réfère à une certaine façon de rendre publics et de partager des programmes. Il implique que ces programmes ne sont pas seulement mis à disposition sous forme de code binaire, mais que leur code source, lisible pour les être humains, devient aussi accessible à tous. Pour qu'il soit qualifié d'*open source* il faut que les créateurs d'un logiciel permettent au public de le modifier et de le redistribuer. À un second niveau, le terme désigne une communauté de personnes qui s'est développée autour de cette attitude d'ouverture et de partage ; cette communauté est aujourd'hui à l'origine d'une partie considérable et croissante de la production de logiciels. Pour quasiment tout programme propriétaire il existe aujourd'hui un équivalent *open source*.

[184] Ce chapitre est en grande partie une traduction d'un chapitre de Rieder et Schäfer [2005].

[185] En France, on parle généralement du « logiciel libre » bien que les deux termes anglais, « open source » et « free software » ne désignent pas nécessairement la même chose. Le premier peut être appliqué à tout logiciel dont le code source est disponible et modifiable tandis que le deuxième se limite aux licences qui prévoient que tout produit dérivé doit être publié sous la même licence libre.

Bien que le paysage ne manque pas de diversité, il est possible d'esquisser un *idéal-type* qui aide à comprendre les grands traits de son fonctionnement. D'abord, il est certainement impossible d'imaginer le mouvement *open source* sans l'existence de l'Internet. Des plateformes comme *SourceForge*[186], complétées par des *mailing-lists* (listes de diffusion) et *newsgroups* (groupes de discussion), constituent les outils utilisés pour organiser et coordonner une force de travail volontaire et dispersée sur la planète entière. Un projet est habituellement lancé avec un embryon de programme, écrit par un individu ou un groupe, et publié sous une licence *open source* sur la toile avec l'invitation à qui le veut de participer à son développement. Lorsqu'il réussit à attirer suffisamment d'attention, un processus vivant se met en place : suivant la maxime « sortir tôt, sortir souvent » (*release early, release often*) des versions du logiciel sont publiées régulièrement et toute personne intéressée peut ajouter du code, documenter ou éliminer des *bugs*. La décision d'intégrer une nouvelle fonction ou une autre modification dans le projet est le plus souvent prise par un modérateur (un individu ou un groupe) qui se base sur un processus communautaire qui ressemble à l'évaluation par les pairs dans le monde scientifique. La structure linéaire du génie logiciel classique est ainsi transformée en une succession rapide de design, construction et évaluation où les spécifications, le design de l'interface et les tests sont concomitants et le projet soumis à des ajustements permanents. Ici, la collaboration constitue l'« outil » principal capable de faire face à la complexité. Les plateformes techniques disponibles sur le Web assurent les différentes interactions en fournissant l'infrastructure qui donne sa visibilité au projet, en assurant la communication entre les participants et la coordination de tous les aspects du développement. En ce sens, elles représentent les *média* qui rendent possible l'émergence de ce qu'on pourrait appeler une « usine virtuelle » où un public divers et dispersé concentre son *intelligence collective* [cf. Stadler / Hirsh 2002].

Le mouvement *open source* se distingue largement de l'ingénierie traditionnelle en ce qui concerne l'attitude générale ainsi que les normes sociales qui y ont cours. La rigueur mathématique est moins valorisée qu'un style de communication ouvert et soucieux des autres. Comme dans beaucoup d'autres subcultures, la démonstration de ses compétences et non pas l'affichage de ses diplômes constitue la base du capital symbolique d'un individu. L'inclusion, la discussion, la collaboration et la circulation ouverte des informations sont plus importantes que l'attribution des tâches, statuts et responsabilités.

Au niveau institutionnel, la scène *open source* est devenue un élément important dans le processus de socialisation et de formation des programmeurs. Ses communautés très vivantes facilitent la recherche d'aide et permettent d'apprendre par le soutien amical et l'imitation d'individus très compétents. La multiplicité des codes accessibles et la culture participative constituent un

[186] http://sourceforge.net

environnement d'apprentissage qui profite à tous les niveaux de connaissance. Alors que le génie logiciel est traditionnellement lié aux institutions plutôt autoritaires de l'école et de l'université, la communauté *open source* complète ces institutions en fournissant un cadre de formation par la pratique, l'imitation joyeuse et l'acquisition autodidacte.

Pour montrer que les produits fabriqués par ce mouvement représentent des éléments importants dans le domaine du logiciel, nous allons brièvement discuter de trois exemples : le système d'exploitation *Linux*, le serveur Web *Apache* et le navigateur *Firefox*.

Linux naît en 1991 quand un étudiant finlandais, Linus Torvalds, écrit un programme « noyau » (*kernel*), le cœur de tout système d'exploitation, comme un projet de loisir personnel et le publie sur Internet, invitant d'autres personnes à participer à son développement. Depuis, *Linux* est devenu un système moderne, robuste et complet – et probablement le dernier concurrent réel de *Microsoft Windows*. Il est gratuitement disponible et constamment maintenu et augmenté par des milliers de volontaires autour de la planète. La plupart des entreprises du *Fortune 500* ainsi que les administrations de Paris, Vienne et Munich utilisent *Linux*, surtout sur leurs serveurs. Une raison de ce succès est certainement le coût d'acquisition, mais d'autres facteurs interviennent, tels que la fiabilité, l'indépendance de plateforme et la possibilité de corriger des erreurs sans passer par un vendeur externe.

Le projet *Apache* est né en 1995 et depuis, ce logiciel est devenu le serveur Web dominant avec une part de marché supérieure à 69%[187]. *Open source* et gratuit il reste sous le contrôle de l'*Apache Software Foundation*, une entreprise à but non lucratif qui aide à organiser le développement, propose de l'assistance légale au projet et protège la marque. *Linux* et *Apache*, couplés à la base de données libre *MySQL* et au langage de programmation Web également *open source*, PHP, forment la plateforme la plus répandue pour le développement d'applications dynamiques sur le Web : LAMP.

Le navigateur Web *Firefox* a poussé à partir d'un code publié en 1998 par l'entreprise *Netscape*, qui se trouvait, déjà à ce moment-là, dans des difficultés financières. Après une série de produits qui n'ont guère eu de succès, la *Mozilla Foundation* publie *Firefox* en version 1.0 à la fin de 2004. Porté par la forte critique qui frappait alors *Internet Explorer* de *Microsoft*, surtout à cause des problèmes de sécurité récurrents, le navigateur *open source* à réussi à capturer une partie considérable du marché[188] pendant l'année 2005. Il fournit également un bon exemple pour voir comment la communauté *open source* rend possible la participation de non-programmeurs à ses projets : à travers *Bugzilla*, un outil

[187] Netcraft ServerWatch Octobre 2005 (http://www.serverwatch.com/stats/article.php/3554746)

[188] En Europe, Firefox est à 36% en Slovénie, 33% en Finlande, 24% en Allemagne et 17% en France. Cf. XiTi Browser Survey, Avril 2006 (http://www.xitimonitor.com/etudes/equipement14.asp)

pour documenter des *bugs*, tout le monde peut reporter des erreurs et demander de façon officielle des fonctions pour les futures versions. Les utilisateurs compétents peuvent élargir les possibilités du navigateur par des extensions, les « *plug-ins* », sans connaître le code de l'application centrale. *Firefox* n'est finalement pas seulement un logiciel mais aussi une communauté qui produit des logos, des t-shirts, des images et des fonds d'écran ainsi qu'une campagne professionnelle de marketing viral.

Il est certainement trop tôt pour porter un jugement final sur l'*open source* et l'importance véritable qui sera la sienne dans le moyen et long terme, mais il nous apprend à penser l'*interaction sociale* comme dimension supplémentaire des relations *intertextuelle* et *interopérationnelle*. L'Internet transforme le développement du logiciel au sens où il le sort de son isolement et lui fournit une plateforme dans les trois sens que nous avons mentionnés. Le mouvement *open source* est certainement une indice majeur qui nous permet de diagnostiquer que la phase de réticulation en cours est en train de changer une fois encore l'idée que nous nous faisons de la création de dispositifs informatiques. Bien que le terme « changer » puisse laisser penser que les formes plus anciennes de création seraient en train de disparaître, ce n'est pas le cas. La diversification des objets techniques et des usages conduit naturellement à une diversification sur les plans du développement et du design. Au lieu de perdre notre énergie dans la perpétuelle lutte des disciplines pour la prééminence, il serait plus intéressant de réfléchir à la question de savoir quel contexte technique, organisationnel, social et culturel est compatible avec quelles méthodes et de quelle façon ils nous est possible de réfléchir les enjeux qui se préparent par rapport aux métatechnologies à l'intérieur du champ du développement. Les problèmes auxquelles nous sommes confrontés en tant que constructeurs de logiciels ou de systèmes d'information évoluent constamment et nos réponses devraient évoluer au même rythme.

2.3.3 L'élargissement de l'espace problématique

Les perturbations et extensions de la théorie et surtout de la pratique du développement du logiciel sont inséparables de la pénétration de l'ordinateur dans les *pores* de nos sociétés. La création d'un logiciel de calcul de masse critique pour la physique, d'un système de gestion de relations client pour une entreprise ou d'une plateforme de discussion ne soulèvent pas les même problèmes et difficultés. Le travail de développement est certainement similaire dans les trois cas, au sens où les outils techniques – les langages de programmation et les méthodes de modélisation – peuvent être largement identiques. L'investissement de réflexion et les connaissances à mettre en oeuvre seront pourtant bien différents dans chaque cas. Simuler un calcul nécessite d'abord une compréhension mathématique complète du problème. Une fois que les formules sont en place, leur mise en algorithme est banale. L'informatisation des relations clients implique une très bonne connaissance des structures organisationnelles et des flux de travail de l'entreprise en question ; la difficulté technique consiste à garantir que les données et leurs relations sont représentées avec fidélité. Dans le cas d'une plateforme de discussion, il est surtout important de savoir comment la communication en ligne doit fonctionner, comment les échanges se structurent et comment on arrive à créer une interface conviviale.

Techniquement, il s'agit d'un développement Web standard, sans rien à inventer. Ces exemples rapides illustrent le fait que le développement logiciel est peut-être homogène en ce sens qu'à un moment ou un autre il faut écrire du code, programmer, concevoir une interface, mais il ne l'est pas par rapport aux questions soulevées qui concernent le domaine de l'activité envisagée et son contexte.

Routine et recherche

Les termes routine et recherche mettent en évidence une différence importante. Chaque processus de développement contient une très grande part de travail de routine pour qui, dès le départ, le chemin entre le *status quo* et l'état désiré est peu problématique parce qu'il ne touche pas de problèmes inédits [cf. Berger / Luckmann 1966]. Les problèmes sont connus et les développeurs familiers du domaine les ont déjà résolus plusieurs fois auparavant. La seule façon d'apporter un peu de créativité à ce travail qui est finalement de pure exécution consiste à optimiser le code pour lui donner plus de robustesse, de performance ou d'élégance interne. Souvent pourtant, un projet de logiciel implique une partie de recherche qui dépasse les connaissances actuelles des personnes qui participent au projet. Avant de passer au développement proprement dit il sera dans un tel cas nécessaire d'entamer une recherche dans les différentes sources d'information à disposition – livres, sites Web ou forums – afin de conduire le problème dans l'espace de la maîtrise. Il sera certainement nécessaire de faire de nombreuses expérimentations parce que les connaissances acquises ne se traduisent pas directement en savoir faire. C'est ici que se fait l'innovation.

Le rapport entre routine et recherche est relatif aux connaissances des individus et de l'équipe. Pour un jeune développeur, beaucoup de problèmes impliqueront une phase de recherche avant de devenir routine. L'arrivée des nouvelles générations est donc automatiquement une source d'innovation parce que les recherches de ceux qui ne connaissent pas encore des solutions standard vont – en partie – apporter de nouvelles réponses. Il est également possible de confronter les tâches routinières à un esprit de recherche lorsqu'on souhaite rompre volontairement avec les traditions établies à titre individuel, dans l'entreprise ou dans la profession entière.

Outre ces deux situations particulières, il existe un terrain de recherche qui ne consiste pas seulement à revisiter des endroits connus, mais qui pose des problèmes et des questions nouveaux pour la communauté toute entière. Cette région frontalière est souvent occupée par les universités et les laboratoires de recherche, mais chaque développeur est de temps en temps confronté à un problème pour lequel il n'y a pas encore une solution toute-faite. Nous pensons que ce terrain d'innovation s'est beaucoup étendu depuis la naissance de l'ordinateur et sa dispersion dans maints domaines de la vie de tous les jours ; cela implique, de notre point de vue, de nouvelles alliances entre la technique et les sciences.

Nouvelles frontières

Cette région frontalière en expansion est directement liée à l'application de l'informatique à des domaines qui n'étaient pas touchées par le numérique auparavant. Ici, il manque de l'information et de l'expérience pour organiser le problème de manière rigoureuse et les progrès se font de façon aveugle et par conséquent expérimentale. Historiquement, il n'est pas faux de dire que toute l'informatique était à l'origine un terrain « sauvage ». On ne savait pas ce qui était possible, où se trouvaient les limites et l'espace était pleinement ouvert. Avec les premières expérimentations, des connaissances et des métaphores se sont mises en place et ont donné une première orientation. La science se proposait de structurer le champ des recherches, et, au début, c'étaient bien sûr les mathématiques qui travaillaient le terrain, accompagnées des métaphores que nous avons rencontrées au début de notre travail. Progressivement, d'autres disciplines ont été interpellées par l'informatique dans son évolution expansive et le contact avec la machine universelle les a chaque fois transformées. La psychologie devient psychologie cognitive par le biais de l'intelligence artificielle, la bibliothéconomie se transforme en science de l'information à travers l'étude des systèmes d'information. Il n'y a pas de relation qui ne change pas les deux entités qu'elle rassemble.

L'ubiquité de l'informatique, dont on peut parler à juste titre dans les sociétés industrielles aujourd'hui, a pour effet que la région frontalière du savoir en formation qui sépare le savoir routinier de l'inconnu et du non-maitrisé est beaucoup plus étendue et bien plus diverse qu'elle ne la jamais été auparavant. Chaque nouvelle confrontation avec le monde naturel et surtout avec le monde culturel – et donc l'être humain – donne lieu à une nouvelle zone, un nouveau champ de bataille.

Nous avons vu plus haut qu'au moment du passage de la phase expérimentale à l'industrialisation, le génie logiciel chercha du côté de l'ingénierie pour trouver des méthodes structurantes et créatrices d'ordre. Pour faire face à la vulgarisation de l'informatique, le design orienté-usager amena la psychologie (dans un sens plus large que psychologie cognitive), mais aussi les sciences sociales, à comprendre les usagers à travers des recherches de profondeur variable. Depuis que les ordinateurs sont devenus des technologies phares de la communication, la branche « communication » des SIC se trouve elle aussi en contact permanent avec ces objets.

L'évolution des zones frontalières en est évidemment arrivée à un point où une nouvelle fois, des questions encore inhabituelles pour les développeurs entrent dans l'équation. Nous l'avons vu à propos des métatechnologies : les critères qui permettent de définir un « bon » logiciel deviennent problématiques au moment où les enjeux sociaux et culturels concernant l'information et la communication se manifestent. Selon le sociologue Niklas Luhmann [1995, p. 183], les média « contribuent à la construction de réalité de la société » (*leisten einen Beitrag zur Ralitätskonstruktion der Gesellschaft*) et quand les NTIC jouent le rôle *outils publics* et de *média de masse*, les créateurs de ces objets sont nécessairement confrontés à des problèmes qui ne s'imposaient pas à eux auparavant. Dans le contexte d'un moteur de recherche à destination globale, il n'y a pas de « bonne » réponse à la