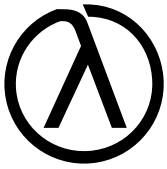

Les promesses non tenues de l’industrie du logiciel

Au moment où je commence cet article, j’ai 27 ans de carrière derrière moi dans le métier de l’ingénierie logicielle, de 1998 à 2025.

Pendant ce gros quart de siècle, j’ai vu passer pas mal de technologies qui étaient censées révolutionner la profession ou être l’avenir incontournable de tout le paysage technologique.

Cet article est un recensement de ces technologies qui n’ont pas tenu les promesses qui étaient faites par leurs promoteur·ices au moment où elles sont sorties des laboratoires des universités ou des services de recherche et développement des entreprises. Il est appelé à s’enrichir au fur et à mesure que des souvenirs me reviendront concernant de telles technologies.

1. La programmation orientée objet

Au moment où je tape ces mots, les langages orientés objet sont (encore) largement dominants dans l’industrie du logiciel. Entre 1994, année d’apparition du langage Java et aujourd’hui, en passant par 2002, année d’apparition du langage C#, l’adoption de la programmation orientée objet a indiscutablement été très large.

Mais, durant ce laps de temps, de nombreuses promesses ont été faites, qui n’ont pas été tenues.

1.1. L’éradication des bugs

La principale d’entre elles a été la suivante : **le nouveau niveau d’abstraction que permettait la programmation orientée objet, en comparaison de la programmation procédurale qu’elle était appelée à supplanter, devait permettre de créer plus simplement des logiciels plus riches et moins bugués**.

La programmation orientée objet a effectivement largement supplanté la programmation procédurale et on s’en est effectivement servi pour écrire des logiciels de plus en plus riches et de plus en plus complexes (ce que l’on doit probablement au moins autant aux gains de puissance des ordinateurs commercialisés sur la même période qu’à l’avènement de la programmation orientée objet).

Mais pour ce qui est d’avoir moins de bugs, pardon. Aujourd’hui plus que jamais, l’industrie du logiciel croule sous les bugs.

Le code que les développeur·euses écrivent est rarement juste du premier coup. Des sommes d’argent très conséquentes sont investies dans des tests et des contrôles de qualité pour détecter les bugs et les corriger avant que le logiciel atterrisse entre les mains des utilisateur·ices. Et malgré cela, il arrive encore que ces utilisateur·ices aient à subir les affres de bugs qui seront passés au travers de ces contrôles.

En fait, l'élévation du niveau d'abstraction apporté par la programmation orientée objet a peut être effectivement permis de s'attaquer à des problèmes plus complexes, avec des logiciels plus ambitieux Dans quelle mesure ? Cela peut être sujet à débat.

Mais, en revanche, il n'a quasiment rien apporté en matière de limitation des bugs.

La raison en est simple : le supplément d'abstraction apporté par la programmation orientée objet par rapport à la programmation procédurale concerne surtout l'organisation (en classes) des données traitées, mais très peu l'organisation des traitements eux-mêmes. Le contenu et le fonctionnement d'une méthode en programmation orientée objet (par exemple écrite en C++, en Java ou en C#) ressemble à s'y méprendre au contenu et au fonctionnement d'une procédure en programmation procédurale (par exemple écrite en C ou en Pascal).

En dehors de quelques notions de portées de variables et la possibilité de surcharger des méthodes (dont on peut se demander quel est en fin de compte le rapport bénéfice / risque entre la puissance d'expression qu'elle apporte et la confusion qu'elle peut induire), les traitements sont exprimés avec les mêmes structures algorithmiques en programmation orientée objet qu'ils l'étaient dans les langages procéduraux. On utilise toujours les mêmes `if`, `while`, `for`…

On a toujours des catégories de bugs qui sont des classiques depuis des décennies : déréférencements de pointeurs non valorisés (`NullPointerException` en Java), indexes hors bornes (`IndexOutOfBoundsException`)… et surtout **effets de bords mal maîtrisés**.

Nous manquons de statistiques sur le sujet, mais je ne serais pas étonné que la majorité des bugs que l'on rencontre lors du développement d'un logiciel appartiennent à cette catégorie.

Or, loin de chercher à contenir les effets de bord (comme le fait la programmation fonctionnelle pure), la programmation orientée objet repose fondamentalement sur ceux-ci. Par exemple, la notion même de mutateur d'attribut, qui est un fondement de la programmation orientée objet, n'est rien autre qu'une mise en œuvre d'effets de bord.

Si on voulait éradiquer, ou au moins contenir les bugs, alors peut être que la programmation orientée objet n'était pas le paradigme de programmation sur lequel miser.

1.2. Des apports mitigés en matière de puissance d'expression

La programmation orientée objet propose, par rapport à la programmation procédurale, des nouveautés qui sont indiscutablement des apports en matière de niveau d'abstraction et de puissance d'expression. Les **classes** d'objets, leurs **constructeurs**, leurs mécanismes d'**héritage**, la notion d'**encapsulation des données**… sont autant d'exemples de ces apports.

Mais tous ces apports théoriques rencontres parfois des limites pratiques, qui peuvent s'avérer frustrantes pour les développeur·euses (d'autant plus qu'ils auront montré de l'enthousiasme pour ce que la théorie devait leur permettre).

Un petit exemple personnel pour illustrer ce sentiment : quand cela me paraît approprié, je considère de bonne pratique de chercher à créer des classes d'objets non mutables ou avec au moins certains de leurs attributs qui soient non mutables. Il s'agit d'objets dont les valeurs des attributs sont définies une fois pour toutes à la création et ne peuvent ensuite pas être changées.

Il existe des techniques qui permettent de faire cela, au moins dans certains langages. Par exemple en Java, il suffit de déclarer `private final` tous les attributs que l'on veut rendre non mutables, de les initialiser dans un constructeur qui pendra autant d'arguments, de rendre privé le constructeur par défaut sans argument et enfin de doter chacun des attribut concerné d'un accesseur (*getter*), mais surtout pas d'un mutateur (*setter*).

Mais les **frameworks** dominants de la scène Java nécessitent que certaines des ces classes soient sérialisables. Pour cela, elles doivent implémenter l'interface `Serializable`, ce qui est anodin, mais également… être dotées d'un constructeur par défaut sans argument `public` et d'un mutateur pour chacun de leurs attributs. Du coup, à l'eau la bonne pratique.

Ah, les *frameworks*…

Notons, à titre de second exemple de limitation par rapport au pouvoir d'expression théorique que devait apporter la programmation orientée objet, que certains de ces *frameworks* nécessitent la mise en œuvre d'objets sans état (*stateless*). Pour un paradigme de programmation dont le cœur battant est la gestion de mutation d'états, ça se pose là. D'ailleurs, cette notion de *stateless* n'était qu'une façon pudique d'éviter à parler de retour à une approche procédurale, en la maquillant avec des idiomes orientés objet.

De *frameworks*, il va en être question dans la section suivante.

1.3. La valse des *frameworks*

Entre 1999 et 2014, les *frameworks* suivants, tous considérés comme majeurs dans le monde Java, se sont succédés :

- 1999 : EJB
- 2001 : EJB2
- 2002 : Spring
- 2006 : EJB3
- 2014 : Spring Boot

On voit que plusieurs de ces *frameworks* sont des itérations successives de la même approche. Ainsi à EJB (*Enterprise Java Beans*) a succédé EJB2, puis EJB3, avec à chaque fois une nouvelle « révolution » du concept. Et à chaque fois avec l'espoir de tenir les promesses techniques que l'itération précédente n'était pas parvenue à tenir, en dépit (ou plus probalement à cause) d'une complexité délirante.

Las de voir les versions successives d'EJB échouer, d'autres équipes on démarrer le projet Spring, lui aussi bien assez complexe à mettre en œuvre, jusqu'à ce que Spring

Boot vienne le simplifier substanciellement (non pas que ce soit pour autant devenu complèment trivial).

La succession de ces différents *frameworks* est la marque du fait que la programmation orientée objet n'a pas réussi à tenir ces promesses initiales. Comme elle n'y arrivait à elle seule, on l'a enrichie d'une approche *framework*, puis d'une autre quand la première a à son tour échoué, puis d'encore une autre… dans une sorte de fuite en avant incroyablement coûteuse.

Car il est à noter que chacun de ces *frameworks* était annoncé, à sa sortie, comme le *framework* ultime, sur lequel on pouvait miser de façon pérenne pour refonder l'intégralité de son système d'information. Et à chaque fois, des entreprises s'y sont laissées prendre, investissant des sommes considérables dans des développements qui allaient devenir obsolètes parfois seulement quelques années plus tard. Il est même arrivé que certains de ces développements aient été obsolètes avant d'être terminés.

J'ai pris ici l'exemple du monde Java parce que c'est celui que j'ai le mieux pu suivre, à titre professionnel, durant ces décennies. Mais je ne doute pas que les développeur·euses ayant évolué dans d'autres univers techniques à la mode orientée objet ou eu à connaître les mêmes déconvenues.

2. XML (et le web sémantique)

À la fin des années 1990 est apparu le métalangage XML. Métalangage car, plutôt qu'un langage, c'est une norme qui permet de créer des langages.

Il s'agit d'un métalangage de marquage structurel de données, inspiré du plus ancien SGML, le dépoussiérant et le modifiant de sorte qu'il soit plus aisé à *parser* par des programmes.

XML était porteur des énormes ambitions du W3C, le *World Wide Web Consortium*, organisme normalisateur du web. Il devait devenir la base des langages d'échanges de données sur tout Internet.

À commencer par les pages web. Le langage HTML, basé sur SGML, a lui même été passé à la moulinette XML pour donner le XHTML, qui était la forme de HTML préconisée dans les années 2000. Les auteur·ices de pages web étaient même alors incité·es à faire figurer sur leurs sites des logos de certification assurant que le code de leurs pages étaient bien conforme à la norme XHTML.

La norme XML s'accompagne de plusieurs spécifications associées :

XML Schema

Un langage basé sur XML pour décrire et normaliser des langages basés sur XML.

XML Schema était le remplaçant désigné du format DTD, qui remplissait la même fonction pour tous les formats dérivés du SGML.

XSL

Une norme composée de deux langages basés sur XML :

XSLT

Un langage de « feuilles de styles » permettant de définir des règles de transformation pour passer des documents d'un format XML vers un autre format XML ou éventuellement vers un format texte.

Ces transformations requièrent la mise en œuvre d'un programme processeur XSLT.

Ces feuiles de styles XSLT devaient notamment permettre la conversion de documents XML en pages XHTML pour des publications sur Internet.

XSL:FO

Un langage permettant de décrire des documents paginés et mis en forme en vue de conversions dans des formats appropriés pour des impressions, comme le PDF ou le Postscript.

Là aussi, cette conversion nécessite la mise en œuvre d'un programme processeur XSL:FO.

On imaginait ainsi que des documents XML pourraient être convertis, via des transformations XSLT, en document XSL:FO qui seraient ensuite convertis en documents PDF.

Tout ceci était mis en place pour préparer un avenir radieux dans lequel toutes les données électroniques seraient stockées dans des formats XML, selon une normalisation de format elle-même définie en XML (Schema) et qui pourraient être transformés, en fonctions des besoins, en d'autres formats XML plus adaptés, selons des règles de transformation elles-mêmes définie en XML (via la norme XSL). Bref, il y aurait du XML partout et à tous les niveaux.

À terme, on nous promettait même le **web semantique**. Au lieu de rédiger directement des pages (X)HTML, les créateurs de sites web devraient conserver leurs données dans des fichiers XML constitués de balises les plus adaptées possibles à la sémantique originelle de ces données. Idéalement, ces formats XML sémantiques devaient évidemment être normalisés en XML Schema.

Ces documents XML sémantiques devaient ensuite être associés à des feuilles des styles XSLT contenant des règles de transformation permettant de les transformer en pages XHTML. Ces transformations devaient être mise en œuvre par des processeurs XSLT embarqués dans les navigateurs web (au même titre que les interpréteurs Javascript ou CSS qu'ils embarquent déjà).

Comme chacun·e peut le constater aujourd'hui, ce web sémantique n'est pas advenu.

Le format XML implique l'utilisation de balises ouvrantes et fermantes qui peuvent être jugées comme relativement encombrantes. Ceci a poussé certain·es acteur·ices du secteur à concevoir des alternatives plutôt que de se ralier à la grande bannière unificatrice du XML.

Par exemple, la norme XML Schema était notoirement complexe et fastidieuse à mettre en œuvre, au point de se faire parfois qualifier « d'usine à gaz ». Aussi est rapidement apparue, dès 2003, une norme concurrente appelée **Relax NG**.

Celle-ci n'était pas elle même basée sur XML et permettait de définir des formats XML beaucoup simplement et concisement que le format XML Schema. Dans les grandes lignes, Relax NG permettait de faire 80% de ce que permettait XML Schema (ce qui était généralement bien suffisant) pour seulement 20% de sa complexité.

Les équipes derrière certains formats XML de renom de l'époque, comme Docbook, OpenDocument ou encore Epub, ont décidé d'opter pour Relax NG plutôt que pour XML Schema. Cela ouvrait une première brèche dans l'avenir tout en XML qu'on nous promettait.

Puis le clou fût enfoncé par le remplacement progressif de XML par JSON comme format d'échange utilisé par les applications web. Ces échanges reposent sur la technologie que l'on a appelée AJAX, pour *Asynchronous Javascript And Xml*. Le XML y était donc bien orpginellement présent.

Mais il a été progressivement remplacé par JSON, à partir de 2001, un format qui présentait l'avantage d'être beaucoup plus facilement manipulable en Javascript que le XML. On continue, par habitude, de parler d'AJAX plutôt que d'AJAJ (pour *Asynchronous Javascript And Json*), mais c'est bien JSON qui domine aujourd'hui largement les formats d'échanges entre applications et services web.

Non content de bouter XML hors des échanges entre services web, JSON lui a également taillé des croupières dans d'autres domaines dans lesquels XML s'était installé, comme celui des formats de fichiers de configuration.

Et aujourd'hui, c'est le format YAML qui tend à tailler des croupières à JSON dans ces mêmes domaines. Mais JSON demeure en revanche le maître des échanges entre services web.

L'ultime affront envers XML a sans doute eu lieu en 2009, quand le W3C lui-même s'est résolu, dans la guerre de succession à HTML4, à cesser de pousser la norme XHTML 2.0 face à la nouvelle norme HTML5, toujours basée sur SGML. Et c'est bien la norme HTML5 qui est aujourd'hui au centre des applications web modernes, reléguant la norme XHTML au rang de péripétie de l'Histoire de l'informatique.

Sur le plan des formats de documents eux-mêmes, la complexité de XML a fini par lasser. Dans le domaine de la rédaction de documentations techniques (dans le domaine de l'informatique), des formats précédemment basé sur XML (comme Docbook) tendent désormais à être remplacés par le format **Markdown**, adopté pour le seul fait qu'il est radicalement plus simple.

XML n'a totalement disparu, loin de là. On le retrouve dans de nombreux formats qui ont été mis au point entre la fin des années 1990 et le courant des années 2000, voire 2010 et qui ont perduré jusqu'à aujourd'hui, comme le format OpenDocument utilisé par la suite LibreOffice.

Mais il est beaucoup plus rarement adopté sur des projets initiés plus récemment. Au lieu d'atteindre la position ultra dominante à laquelle il était promis, il fait aujourd'hui figure de technologie un peu ringuarde.