QUELQUES BONNES PRATIQUES DE DÉVELOPPEMENT LOGICIEL À L'USAGE DU STATISTICIEN SELFEUR

(OU « SAVOIR COMPTER, SAVOIR CODER »)

Emmanuel L'Hour, Ronan Le Saout** et Benoît Rouppert****

En plus de compétences en méthodologie statistique et d'une bonne connaissance des sources de données disponibles, le métier de statisticien nécessite une bonne maîtrise des outils informatiques. Les programmes informatiques écrits permettent non seulement de produire des résultats, mais ils peuvent aussi devenir des livrables du travail réalisé, que ce soit en tant qu'élément de preuve ou en tant qu'outils réutilisables pour d'autres travaux. Dans cette optique, le statisticien se doit d'acquérir les bonnes pratiques de développement logiciel qui lui permettront de garantir une appropriation facile de ses programmes par d'autres utilisateurs, ou une ré-appropriation par lui-même, au-delà de la période pendant laquelle le développement a eu lieu.

Ces bonnes pratiques couvrent tous les aspects du cycle de développement logiciel : la définition des exigences, l'architecture du programme, les styles de programmation, les choix techniques, les outils de développement et les tests. Elles permettront de garantir une bonne réponse au besoin des utilisateurs, après une étape de questionnement de ces besoins, et une fiabilité des résultats obtenus tout en maîtrisant le coût de programmation. Et plus important encore : en rendant les programmes facilement lisibles, elles aideront le producteur de statistiques publiques à communiquer sur ses choix méthodologiques et sur la façon d'utiliser les données, renforçant ainsi la confiance que lui accordent ses utilisateurs.

In addition to skills in statistical methodology and a good knowledge of available data sources, the profession of statistician requires to be comfortable with IT tools. Programs not only produce the statistical results, they can also become deliverables, either as evidence or as reusable tools for future work. Having this in mind, the statistician must acquire software development best practices that will allow him to guarantee an easy understanding of his programs by other users, or re-appropriation by himself in future developments.

These best practices cover all aspects of the software development cycle: definition of requirements, program architecture, programming styles, technical choices, development tools and testing. They will make it possible to guarantee an appropriate answer to users' needs, after a step of questioning these needs, and quality of the results obtained while controlling the development costs. And more important, by making the programs easily readable, they will help the producer of official statistics to communicate on his methodological choices and how to use the data, and thus strengthen the confidence of his users.

* Chef du Service statistique, direction interrégionale Insee La Réunion-Mayotte, *emmanuel.lhour@insee.fr*

** Expert en méthodologie statistique, Commissariat général au développement durable, *ronan.lesaout@developpement-durable.gouv.fr*

*** Ancien chef du département des Production et infrastructure informatiques, Insee

DE LA PROGRAMMATION À TOUS LES ÉTAGES

Parmi les statisticiens apparaissent de plus en plus des profils qui combinent des compétences en méthodologie statistique, des connaissances sur le contenu métier des sources de données, et une maîtrise des outils informatiques. Ce troisième élément prend une importance croissante, notamment par la capacité à « savoir coder »[1], c'est-à-dire savoir écrire un programme informatique. Mais derrière cette formule facile à retenir, se cache une diversité de pratiques de programmation, telles que :

- celle du responsable des retraitements post-collecte d'une enquête ;
- celle du chargé d'études s'appuyant sur des méthodes statistiques plus ou moins sophistiquées pour établir un résultat ;
- celle du chercheur désirant rendre ses travaux reproductibles ;
- ou celle du « diffuseur » qui veut mettre à la disposition d'un large public une application de visualisation des données.

Le point commun entre ces exemples est qu'ils mobilisent des compétences souvent associées à l'informatique et parfois ignorées du statisticien. Elles relèvent du dialogue avec l'utilisateur, de la conceptualisation de son processus de travail, des méthodes de travail en équipe et de la connaissance des méthodes et des outils du jour. Cet article vise donc à présenter ici quelques outils conceptuels à l'usage du statisticien pour se construire une aisance rédactionnelle avec ses programmes.

LE SELFEUR, UN «DÉVELOPPEUR» EXPERT DE SON DOMAINE MÉTIER

La profession de statisticien public n'est pas homogène. Elle recouvre en fait une diversité de métiers.

> *« Le terme de « selfeur » est un néologisme qui renvoie à la partie de l'activité du statisticien qui mobilise de la programmation informatique à travers une expertise « métier ». »*

L'Insee en identifie très précisément 38, regroupés en 6 familles professionnelles (production, études, action régionale, informatique, fonctions support, stratégie et pilotage)[2]. Les termes « développement » et « programmation » (au sens informatique) n'y apparaissent que pour l'analyste-programmeur. Ceci ne signifie évidemment pas qu'il y a absence de programmation informatique dans les autres activités de la statistique publique. En particulier, les descriptifs des métiers de production, d'études ou de l'action régionale[3] font mention de l'utilisation de logiciels statistiques. Le terme de « selfeur » est un néologisme qui renvoie à la partie de l'activité du statisticien qui mobilise de la programmation informatique à travers une expertise « métier »[4].

1. L'article s'appuie sur un document de travail des mêmes auteurs (L'Hour, Le Saout et Rouppert, 2016), dont le titre est librement inspiré par la série d'articles du Courrier des statistiques autour des techniques rédactionnelles, « Savoir compter, savoir conter » (Cotis *et alii*, 2009).
2. [NDLR] La grille des métiers de l'Insee à laquelle les auteurs se réfèrent est une note interne du département RH de l'Insee, en date du 16 janvier 2017. Les pages du site internet qui traitent des *métiers et des formations* reprennent cette typologie (hors stratégie et pilotage).
3. Dans l'organisation de l'Insee, l'action régionale recouvre une palette de travaux à destination des décideurs régionaux ou locaux (études, diffusion de données). C'est une des particularités de l'institut statistique français.
4. Le terme fait notamment référence au travail de programmation en « libre-service », en « self ».

Mais au-delà de l'expertise métier, la comparaison avec les activités de développement informatique dans un cadre « classique » laisse entrevoir une autre notion : celle de la responsabilité vis-à-vis du code. Lorsque la direction du Système d'Information (DSI) est mise à contribution dans un projet d'investissement, il est d'usage de distinguer deux natures de responsabilités : d'une part la responsabilité technique de l'équipe informatique, et d'autre part la responsabilité fonctionnelle de l'équipe « métier » (assurée par le chef de projet statistique ou l'administrateur d'application dans le cadre d'une maintenance). Le développeur informatique est alors essentiellement garant de la bonne exécution du code, tandis que la validité statistique du résultat est plutôt assumée par un responsable d'application.

« Le selfeur porte en effet la responsabilité de la bonne exécution du programme, mais aussi de la valeur statistique de ce que le programme produit. »

Dans le cas d'un développement en autonomie (ou self, ou libre-service), ces responsabilités sont souvent cumulées, même si la responsabilité technique ne fait généralement l'objet que de peu de contrôles et que, de ce fait, on a tendance à l'oublier un peu. Le selfeur porte en effet la responsabilité de la bonne exécution du programme, mais aussi de la valeur statistique de ce que le programme produit (indicateur, base de données retraitée, étude économique, outil de visualisation des résultats, etc.). Il se distingue de l'informaticien par la mobilisation de compétences spécifiques au métier statistique. Le chargé d'enquête, le chargé de l'exploitation de fichiers administratifs ou le chargé d'études intègre ainsi au travail de programmation, des savoirs spécifiques sur les concepts à mesurer : il est capable de mobiliser des références bibliographiques sur la thématique, de prêter attention à des différences de champ dans les sources mobilisées, ou de repérer les difficultés de mesure, etc .

Le selfeur programme au premier abord essentiellement pour lui-même, pendant que le développeur informaticien travaille sur des applications ou des logiciels de large utilisation. Les contraintes ne sont donc *a priori* pas identiques.

L'expérience montre qu'en réalité, informaticien et selfeur portent un même type de responsabilité. Pourquoi donc porter attention à la lisibilité des programmes de retraitements si c'est pour une enquête unique ? Pour la simple raison qu'il est fort probable que ces programmes n'auront pas un usage unique et qu'il est difficile de prévoir les usages futurs : élément de preuve associé à la reproductibilité[5] des études économiques, transmission à des collègues d'éléments méthodologiques, ré-utilisation dans un autre cadre ou pour corriger une erreur, etc. Ce faisant, un selfeur est un développeur comme les autres.

Or, si les techniques rédactionnelles font partie de la formation de base d'un chargé d'études, les « techniques rédactionnelles du code » en sont absentes. Pourtant, les compétences de génie logiciel gagneraient à se propager hors des sphères des projets informatiques. Le statisticien selfeur pourrait s'inspirer des bonnes pratiques bien connues des développeurs informaticiens, pour sécuriser son code et faciliter sa réutilisation ou la reproductibilité de son étude. Ces pratiques s'appuient sur un cadre conceptuel simple et qui a fait la preuve de son efficacité dans les développements « classiques » : un cycle de développement et des étapes-clefs à ne pas négliger.

5. Pour aller plus loin sur la conception orientée reproductibilité (*reproducibility by design*), voir par exemple (Langlais et Eprist, 2020). Sur la reproductibilité de traitements sur des données confidentielles, voir par exemple (Gadouche, 2019).

UN CYCLE DE DÉVELOPPEMENT À TROIS TEMPS

Le cycle de développement d'un logiciel met en jeu trois[6] activités principales ***(figure 1)*** :

- la première a pour objectif de déterminer ce à quoi doit répondre le programme ; c'est ce que l'on appelle les **exigences** ; elle comporte des phases d'élucidation, de spécification puis de validation de la spécification[7] ;
- la deuxième consiste à **concevoir et écrire le programme** ;
- et la troisième à **tester le programme**, c'est-à-dire vérifier qu'il fait bien ce qui est attendu de lui.

Ce processus n'est jamais linéaire. Chacune de ces activités peut faire l'objet d'itérations, à trois niveaux :

- au sein d'une même activité : on peut avoir besoin de réécrire les exigences pour être plus clair, ou corriger un *bug* dans le code (niveau 1) ;
- des allers-retours entre les activités sont également à prévoir : en codant, des exigences peuvent apparaître ambiguës, les tests peuvent identifier des *bugs* et conduire à revoir le code, voire les spécifications des exigences (niveau 2) ;
- enfin, on peut augmenter progressivement le nombre ou le périmètre des livrables du programme, par itérations de cycles (niveau 3). Par exemple, on va d'abord éditer un tableau de données sur une année, puis ajouter des années, puis faire des graphiques, puis les rendre interactifs, etc.

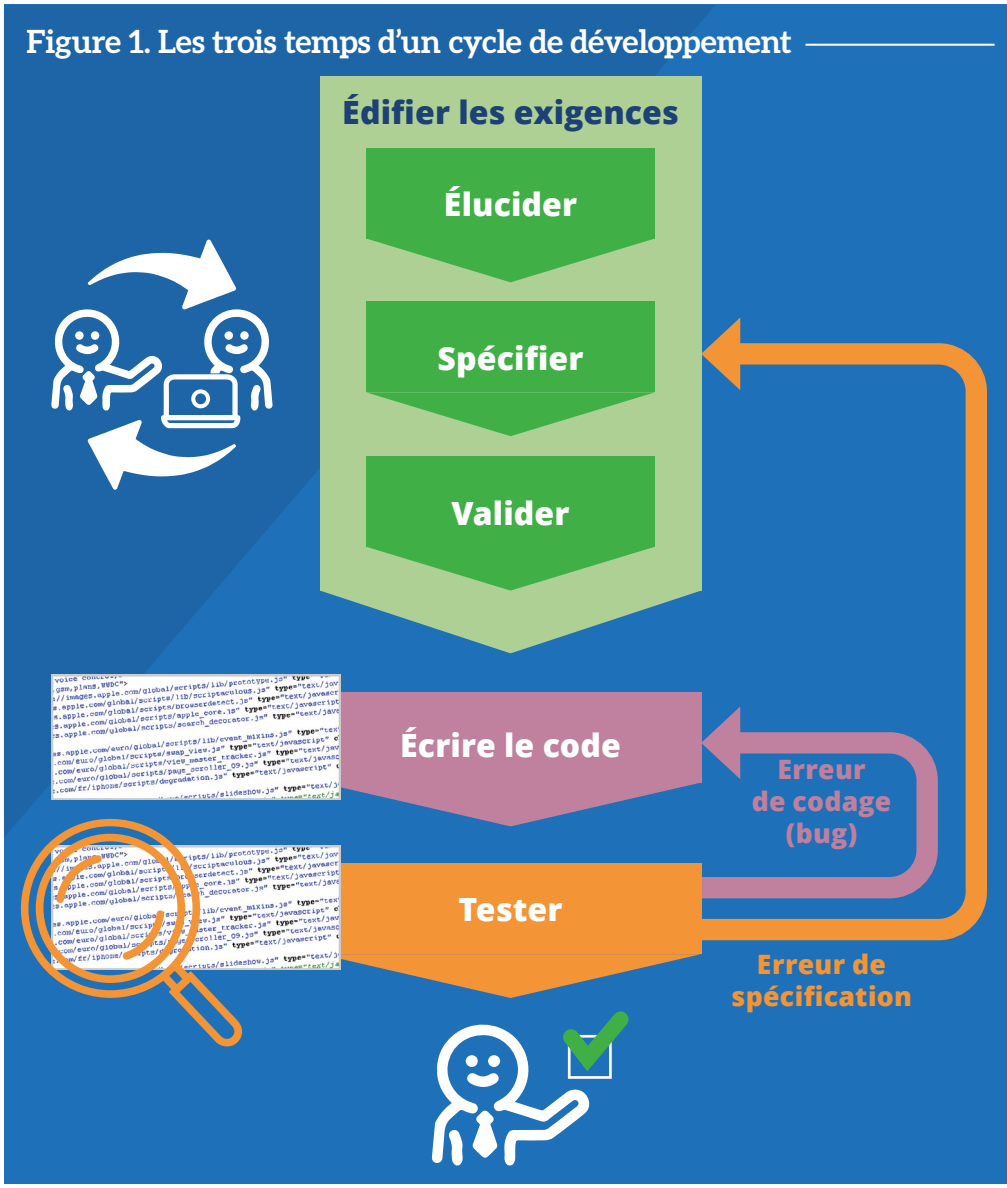
Néanmoins, le plus efficace (McConnell, 2005, chapitre 3) consiste à prévenir les itérations de deuxième niveau, signes de spécifications mal définies ou d'erreurs de codages ***(figure 1)***, qui se révèlent très coûteuses. On pourra ainsi privilégier les itérations au sein de chaque activité (premier niveau), et les itérations de cycles (troisième niveau), où l'on produit toujours quelque chose de fonctionnel en augmentant progressivement la qualité, c'est-à-dire l'adaptation du résultat à la demande.

> *« Le plus efficace consiste à prévenir les itérations de deuxième niveau, signes de spécifications mal définies ou d'erreurs de codages. »*

On conçoit que le statisticien qui écrit un code à usage unique n'éprouvera peut-être pas spontanément le besoin de conceptualiser autant le cadre de l'écriture de son programme. Mais il est probable que la méthode et les outils se révéleront tout particulièrement utiles pour le selfeur isolé, en particulier pour l'aider à gérer ses « conflits » de responsabilité. Dans les projets informatiques, l'organisation du travail collectif porte en soi la méthode, notamment les espaces de concertation en amont et en aval du développement. Si le développement en self s'insère dans un ensemble plus vaste, par exemple entre un processus de collecte doté d'applicatifs spécifiques et un processus de diffusion des données doté de ses contraintes propres, il devient également très intéressant de se pencher sur ce que les informaticiens de métier ont noté comme pièges à éviter et pratiques à préconiser. En tout premier lieu, il convient d'analyser le besoin qui justifie l'écriture d'un programme, ce qu'on nomme les exigences.

6. La littérature autour du développement piloté par les tests *(tests driven development)* utilise une typologie identique : design – code – test.
7. La littérature sur l'ingénierie des exigences *(requirement management)* est plutôt anglophone. Voir par exemple (Robertson et Robertson, 2013) ou (Cockburn, 2000). En français, on peut lire aussi (Constantinidis, 2018).

Figure 1. Les trois temps d'un cycle de développement



ÉDIFIER LES EXIGENCES…

En génie logiciel, une exigence est « *l'expression d'une condition ou d'une fonctionnalité à laquelle doit répondre un système ou un logiciel* »[8].

Établir les exigences, c'est donc identifier les besoins[9], mais également les contraintes, puis décrire ce qu'on est censé faire. Bien poser le problème, savoir quoi exiger, est crucial pour ne pas rater son objectif et éviter d'avoir à tout recommencer (voire abandonner). Un problème mal spécifié expose en général à des difficultés bien plus grandes qu'une mauvaise ligne de code (McConnell, 2005, chapitre 3).

Ne pas se précipiter vers le code et prendre le temps d'établir des exigences a de multiples vertus. Pour un informaticien, cela garantit la satisfaction de l'utilisateur et du donneur d'ordre et réduit aussi les risques d'échec du projet et la durée d'écriture du programme.

L'IREB[10] retient une typologie en trois activités majeures pour établir des exigences : les élucider, les spécifier et les valider. Pour les mettre en œuvre, les outils sont essentiellement sémantiques (distinguer le besoin du superflu ou de la contrainte, se mettre d'accord sur les termes métier utilisés), linguistiques (formuler des exigences), et visuels (faire des schémas). Le souci de sobriété doit prévaloir, car la complexité est source de fragilité (Volle, 2001a ; McConnell, 2005, chapitre 27).

… CE QUI SUPPOSE QUE L'ON CONNAISSE LES UTILISATEURS

Dans un projet informatique classique, l'utilisateur et le donneur d'ordre (appelé aussi client) sont généralement distincts. Le client donne les moyens pour que le projet se fasse (la hiérarchie qui donne l'aval du projet, le partenaire qui finance, etc.). L'utilisateur manipulera le produit du projet. C'est le rôle du développeur de satisfaire la demande du client, mais en s'assurant qu'il fournit un produit qui convient à l'utilisateur[11].

> *« C'est le rôle du développeur de satisfaire la demande du client, mais en s'assurant qu'il fournit un produit qui convient à l'utilisateur. »*

Qui sont ces utilisateurs ? Si aucun utilisateur n'est identifié *a priori*, on peut s'interroger sur l'opportunité du projet. La question n'est pas simple : elle amène une confusion possible entre les utilisateurs des données produites à l'aide du programme développé et les utilisateurs directs du programme.

Dans le premier cas, on parlera d'utilisateur final, par essence assez éloigné de l'informaticien et parfois même du selfeur. La statistique publique dispose généralement d'une représentation des utilisateurs finaux de ses productions, *via* le Cnis[12], un Cries[13], un comité d'utilisateurs d'une enquête, le comité de pilotage d'une étude en partenariat dans une région, etc. Mais ils se situent par construction très en amont du développement.

8. Voir également le site de la Specief (Société pour la promotion et la certification de l'ingénierie des exigences en langue française) *http://specief.org/index.php/ingenierie-des-exigences/*.
9. *« La définition des besoins […] est complexe et délicate, en raison du nombre et de la diversité des parties prenantes, des demandes souvent divergentes, des contraintes variées et, […] du facteur humain »* (Constantinidis, 2018).
10. L'IREB (*International Requirements Engineering Board*), organisation à but non lucratif, est le fournisseur du schéma de certification en ingénierie des exigences (CPRE pour *Certified Professional for Requirements Engineering*).
11. Sur la différence entre la satisfaction de la demande et celle des besoins, voir par exemple (Volle, 2001b).
12. Conseil national de l'information statistique, voir (Anxionnaz et Maurel, 2021).
13. Comité régional pour l'information économique et sociale.

L'utilisateur qui intéresse le développeur, c'est celui à qui il va livrer son programme, ou le résultat de son programme. Il faut donc qu'il soit en capacité d'accéder directement à cet utilisateur. Certes, à défaut, on pourra à certains moments demander à un collègue de jouer le rôle de cobaye. Mais l'idéal reste néanmoins de pouvoir observer l'utilisateur face au résultat produit : le graphique ou le tableau est-il facilement intelligible ? Quels passages de l'étude a-t-il du mal à comprendre ? Le modèle est-il clair et convaincant ? Cet outil interactif est-il aisé à manipuler ?

DANS LE CAS D'UN DÉVELOPPEMENT EN SELF, QUI SONT CES UTILISATEURS?

À première vue, on aura tendance à répondre : le selfeur lui-même. En effet, on associe spontanément au terme de « self » l'image d'un programme développé pour l'usage d'une seule personne, voire un développement à usage unique, par exemple en statistique exploratoire.

Or en pratique, la notion de self recouvre une grande diversité de situations. Rien qu'à l'Insee, sur le périmètre de la production statistique, on a récemment recensé 360 opérations statistiques outillées par des développements applicatifs en self : il s'agit d'une pratique généralisée dans toutes les activités de l'institut, qu'on retrouve dans toutes les phases des processus, de la conception à la diffusion. Le statisticien selfeur qui a écrit le code de ce type de traitements, sous-processus d'un processus plus large, est donc dans une situation assez voisine de l'informaticien dans un projet « classique ». Cependant, il ne pourra pas toujours identifier avec précision l'utilisateur de son programme et sera relativement éloigné des utilisateurs finaux du processus englobant.

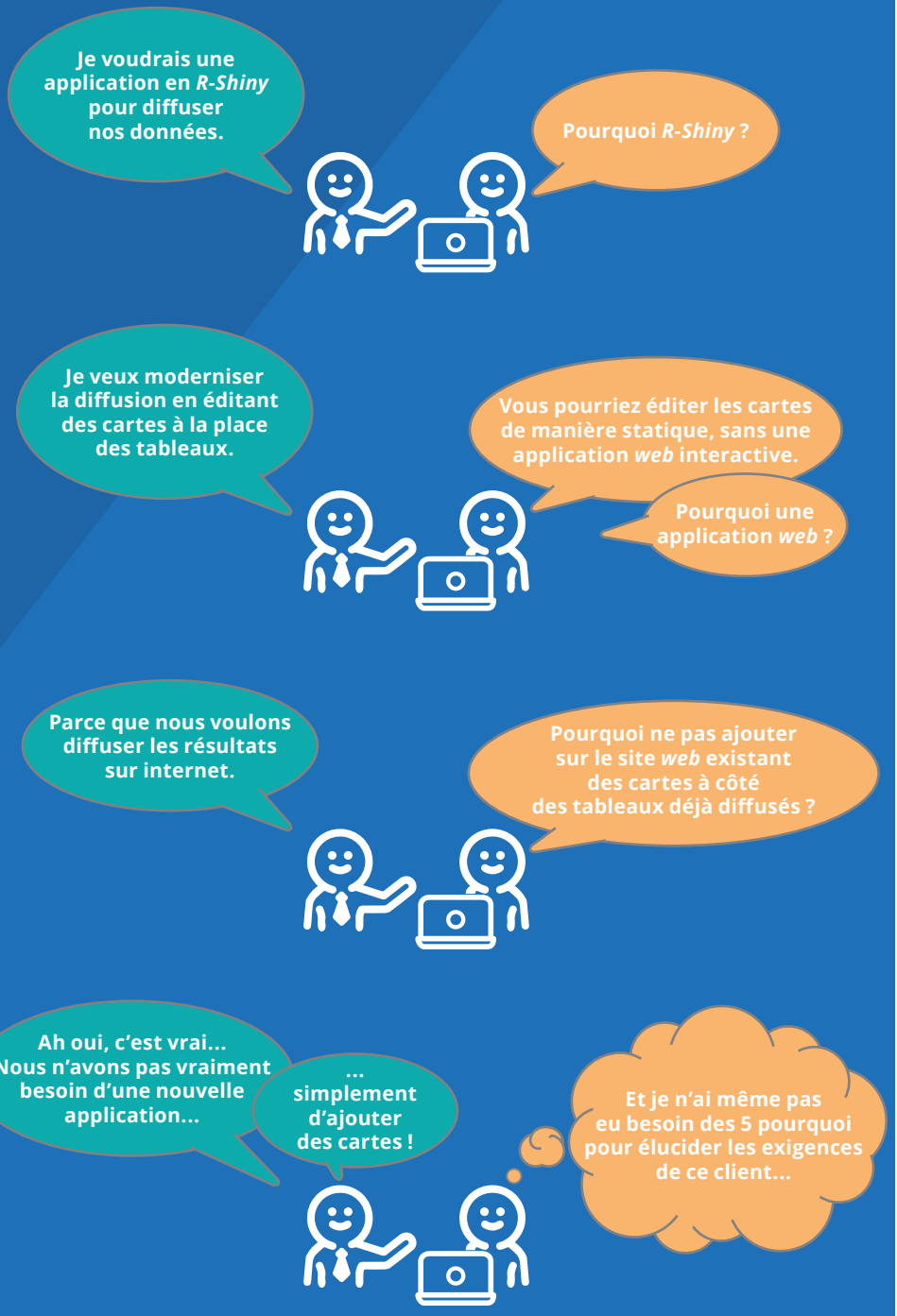
S'il est expert « métier », le statisticien selfeur se sentira particulièrement performant pour comprendre le besoin de l'utilisateur final. Pour autant, il lui faudra éviter de tomber dans le double piège qui consiste à croire, d'une part que l'utilisateur de son programme est le même que l'utilisateur final ; et d'autre part qu'il saurait mieux que son utilisateur ce dont ce dernier a besoin. Par exemple, pour développer un outil de visualisation des adresses à enquêter sur une carte, des échanges avec les enquêteurs aboutiront au choix de cartes interactives numériques plutôt que de cartes imprimées, facilitant ainsi l'usage du GPS sur le terrain.

FAIRE LA CHASSE À L'IMPLICITE, AU NON-DIT

En définitive, élucider les exigences, c'est aller au bout de la démarche de compréhension des besoins de l'utilisateur pour bien distinguer le besoin (ce qui est utile) et la contrainte (légale, matérielle, technique…) du superflu. Sans cela, des exigences que l'utilisateur ne formule pas spontanément, basées sur des besoins implicites ou des contraintes inamovibles seront découvertes tardivement, au risque de devoir abandonner ou tout refaire. Une méthode simple permet d'ailleurs de rendre explicite ce que son interlocuteur tait, souvent parce qu'il le considère comme évident : la série des « cinq pourquoi ». Celle-ci consiste très simplement à demander cinq fois à la suite à son interlocuteur pourquoi il cherche à réaliser ce qu'il demande. À l'usage, les cinq « pourquoi ? » ne seront pas systématiquement tous nécessaires pour aller au fond des choses ***(figure 2)***.

Figure 2. Établir les exigences de l'utilisateur, avant de se lancer dans la programmation

Une illustration (fictive) de la méthode dite des «5 pourquoi»



Dans un autre ordre d'idée, mais pour autant tout aussi primordial, le développeur, comme le selfeur, devront lister les besoins purement informatiques, qu'on appelle « non fonctionnels », pour s'interroger sur la faisabilité du projet avant de se lancer dans la programmation : les exigences de disponibilité ou de sécurité, ont-elles été correctement instruites ? Quel sera le nombre d'utilisateurs en simultané ? A-t-on seulement besoin d'un accès en lecture des fichiers produits, ou aussi en modification ? À quelle fréquence doit-on mettre à jour les données ? Par exemple, pour le suivi de la conjoncture économique, il est nécessaire de pouvoir actualiser les prévisions très fréquemment pendant une période précise du trimestre. Il sera donc très probablement nécessaire d'automatiser totalement l'outil développé pour atteindre l'objectif de réactivité.

MODÉLISER À L'AIDE D'OUTILS SÉMANTIQUES ET VISUELS

Le développeur qui a analysé les besoins fonctionnels auxquels son code doit répondre, et apprécié les contraintes techniques qui s'imposent à lui, doit s'assurer que sa vision est partagée avec celle de son client. Un premier pas dans cette voie consiste à formaliser **le cadre conceptuel** de son travail. Car l'usage de concepts partagés évite les incompréhensions (Evans, 2003).

> *« L'usage de concepts partagés évite les incompréhensions. »*

Pour le statisticien selfeur, une manière d'appliquer cette bonne pratique est de se poser la question des concepts qu'il manipule à travers son code. S'il travaille par exemple sur des données d'entreprises, il devra prendre en compte des concepts statistiques, d'identification ou d'analyse (Siren, Siret, entreprise profilée, taille d'entreprise…) et des concepts comptables (chiffres d'affaires, valeur ajoutée). La liste des concepts servira de référentiel pour nommer les entités du code et des données (tables, variables et leurs modalités, fonctions). À l'Insee, le référentiel des métadonnées statistiques (RMéS) centralise la définition des concepts et des sources de données de la statistique publique[14]. Il constitue donc un outil commode pour le selfeur, comme pour l'utilisateur.

Les schémas sont en outre de précieux outils de communication et de conception. Représenter graphiquement les exigences permet de renouveler la discussion avec l'utilisateur. De surcroît, si les schémas sont confus, traduisant des dépendances multiples et erratiques, cela reflète bien qu'on n'est pas prêt à se mettre à coder. C'est aussi un bon support de documentation.

La modélisation du *Generic Statistical Business Process Model* (GSBPM) répertorie et catégorise les phases d'un processus de production statistique[15]. Cette schématisation d'un processus est générique : elle s'applique quel que soit le processus et permet ainsi de ne pas oublier d'étapes importantes.

14. Pour plus d'information sur le référentiel RMéS, voir (Bonnans, 2019).
15. On trouvera plus d'information sur le GSBPM et sur un modèle qui s'en inspire dans (Erikson, 2020).

Les exemples sont indispensables pour parer au risque d'une modélisation trop abstraite. En particulier, on n'omettra pas la description des exigences en matière de validation de données ou de traitements. La validation d'un traitement peut consister à vérifier les données, individuelles ou agrégées, en cherchant à repérer des valeurs atypiques. Par exemple, un responsable d'enquête auprès d'entreprises identifiera des réponses dont le niveau ou dont l'évolution paraît suspecte, en définissant des seuils, des intervalles, etc. Un chargé d'études, avant de se lancer dans les calculs, construira un ensemble de données de référence à partir d'une bibliographie thématique, ensemble auquel il confrontera ensuite les résultats de ses calculs.

VALIDER LES EXIGENCES AVEC TOUTES LES PARTIES PRENANTES

Dans un projet informatique classique, outre l'utilisateur et le développeur, il convient que le donneur d'ordre valide également les exigences, à l'aide des **spécifications**.

Les spécifications ont ainsi un double usage : elles préparent la phase de codage, mais elles servent aussi à solliciter les utilisateurs. Elles explicitent en quoi le processus proposé répond au besoin de l'utilisateur. Cela peut passer par la description de tests fonctionnels (cf. *infra*) ou la réalisation de prototype. L'écriture des spécifications peut ainsi réduire la frontière entre les cycles de développement, avec une anticipation sur l'écriture du programme et des tests. Si les exigences ne sont pas validées, il faut alors déterminer en quoi le besoin a mal été compris et ce qui doit être revu dans les spécifications.

Notre statisticien selfeur aura ici encore peut-être du mal à identifier son donneur d'ordre, mais le terme de parties prenantes aura très certainement une vraie signification. Ira-t-il jusqu'à faire écrire puis valider des spécifications à celui qu'il aura identifié comme son donneur d'ordre ? Il faut l'essayer pour s'en convaincre : même s'il est son propre donneur d'ordre, son propre utilisateur, le selfeur a tout intérêt à décrire ce qu'il anticipe être le résultat de son programme.

ENCORE UN PEU DE PATIENCE : AVANT DE CODER, DÉFINIR L'ARCHITECTURE

Pour implémenter les exigences, il sera nécessaire de réaliser une succession de traitements (automatisés ou non). Ces traitements sont autant de processus (informatiques) : ils exigent des données en entrée et donnent en sortie leurs résultats.

En pratique, décomposer son processus de travail en enchaînements de modules « entrée-traitement-sortie » se traduit par exemple par quelques pratiques saines :

- isoler les paramètres immuables des traitements dans un fichier spécifique (variables d'environnement) ;
- éviter de mettre les statistiques calculées dans le code. On préférera éditer un fichier dédié ;
- identifier les tables en entrée, ce qui permettra de revenir facilement aux données brutes, par exemple pour modifier les imputations ;
- identifier les tables intermédiaires, ce qui facilitera les analyses explicatives (approfondir *a posteriori* un chiffre calculé, creuser des évolutions étonnantes relevées par les utilisateurs) et la recherche de *bugs*.

L'identification des différents traitements utilisés permettra de définir la **structure du programme**. Un traitement pourra lui-même être décomposé en sous-processus.

Pour définir une architecture qui favorise la lecture, l'évolution future et l'exécution du programme, il est important de découper le traitement global en un ensemble de sous-processus, qui communiquent les uns avec les autres, mais qui restent indépendants pour leurs fonctionnements internes : « *L'objectif principal de l'architecture est de soutenir le cycle de vie du système. Une bonne architecture rend le système facile à comprendre, facile à développer, facile à maintenir et facile à déployer. Le but ultime est de minimiser le coût du système pendant sa durée de vie et de maximiser la productivité des programmeurs* » (Martin, 2017).

On retrouve dans cette organisation la notion de « **barrière d'abstraction** » (Abelson, Sussman et Sussman, 1996) : pour utiliser une sous-partie du programme, il est uniquement nécessaire de savoir quelles données sont utilisées en entrée et quelles données seront produites en sortie, par contre il n'est pas nécessaire de savoir comment le traitement a été implémenté. Cette barrière permet qu'une partie d'un processus ne soit pas affectée par des modifications sans lien avec les traitements qu'il implémente. Les différentes parties peuvent alors être construites, remplacées et corrigées séparément. Une partie de programme construite suivant ce principe est appelé un **module**, et un processus construit ainsi est dit modulaire.

Une approche simple pour évaluer la modularité d'un traitement est de vérifier si le traitement est bien circonscrit (arrive-t-on à le nommer simplement ?) et si les données en entrée et en sortie sont bien identifiées. On dispose ainsi d'une « façade », un patron de conception, qui fait abstraction de la manière dont le traitement est réalisé : manuellement ou automatiquement ? Avec quel langage de programmation ?

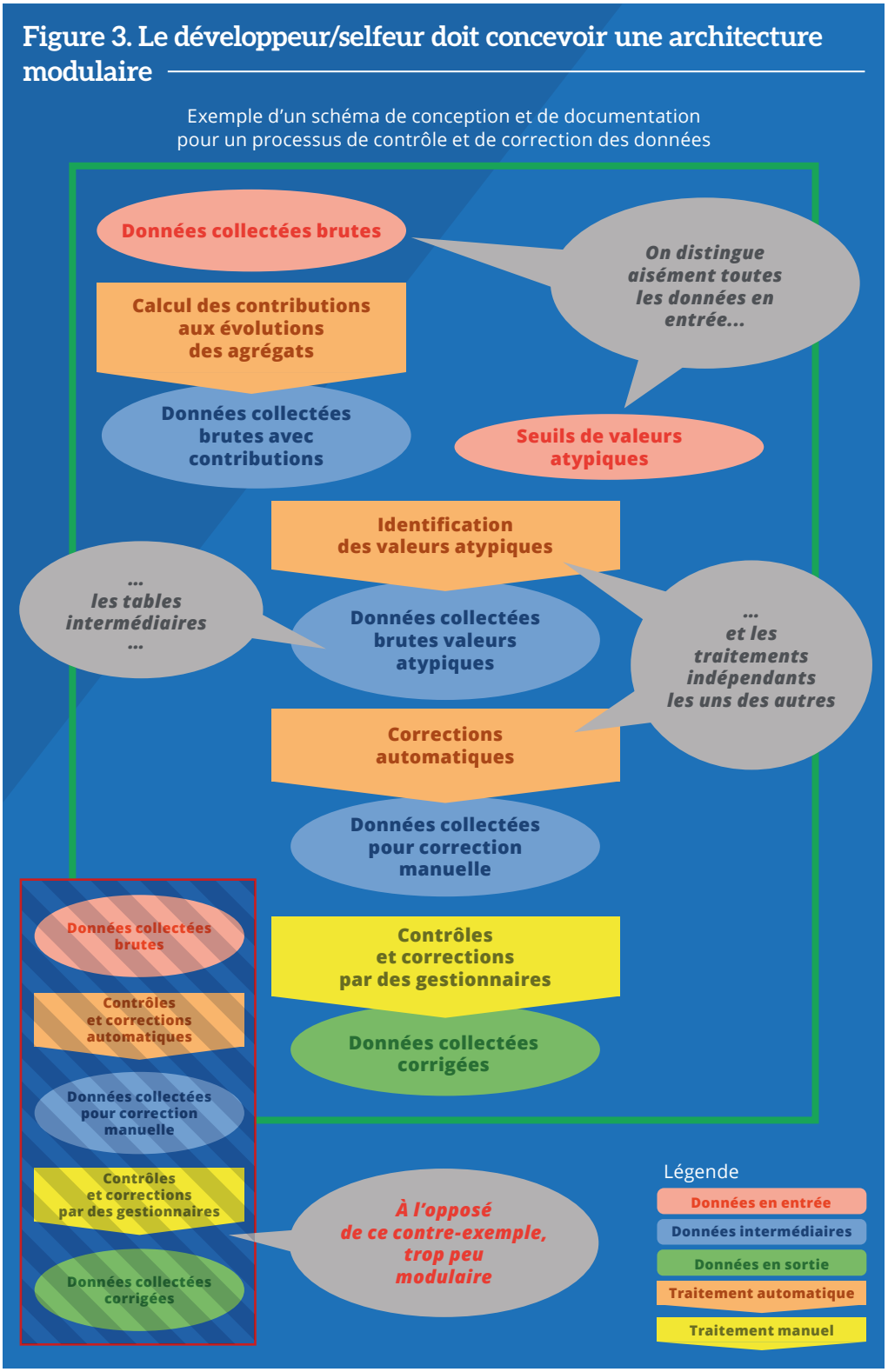
Grâce à la modularité, on se laisse la liberté de répondre à ce type de questions plus tard, et de pouvoir modifier ces réponses au cours du temps. Par exemple, pour passer des données individuelles aux données finales, une étape d'agrégation puis une étape de désaisonnalisation sont souvent nécessaires. Les identifier comme deux parties distinctes est important car cela permettra de faire évoluer séparément aussi bien les méthodologies que les outils informatiques utilisés[16].

C'est lors de l'exécution d'un programme qu'apparaît de la façon la plus flagrante l'intérêt d'un découpage en modules. Dans des projets informatiques, il est fréquent qu'un traitement complet nécessite plusieurs heures ou jours pour être réalisé de bout en bout. Si le traitement est implémenté sous la forme d'un unique programme monolithique, toute erreur en cours de route nécessitera de repartir à zéro. La seule façon efficace de corriger ce défaut est de découper le traitement en modules dont les durées d'exécution sont brèves. Si le découpage suit la logique décrite ci-dessus, alors il sera facile de corriger les *bugs* dans le code et de reprendre l'exécution des programmes à un point intermédiaire. La ***figure 3*** illustre ce que peut être la schématisation d'un processus. Ici, la modularité permet d'éviter certains écueils : tel qu'il est structuré, le programme ouvre la possibilité de connaître voire de modifier les paramètres des contrôles et des corrections automatiques, et d'exécuter séparément les calculs de contributions, les contrôles et les corrections automatiques.

16. Par exemple, l'agrégation pourra se faire avec le langage SQL alors que la désaisonnalisation utilisera un outil spécialisé comme JDemetra+.

Figure 3. Le développeur/selfeur doit concevoir une architecture modulaire

Exemple d'un schéma de conception et de documentation pour un processus de contrôle et de correction des données



ADOPTER UN STYLE D'ÉCRITURE

Écrire un programme, c'est automatiser des tâches qui devraient sinon être réalisées manuellement. Si un programme permet d'éviter les erreurs qui apparaissent régulièrement ou aléatoirement dans les opérations manuelles, il rend systématiques les erreurs qui résultent des défauts d'écriture.

Pour réduire le risque de *bug*, il faut s'assurer de la lisibilité du programme tout au long de l'écriture de celui-ci. Cela passe par une attention particulière au nommage et au style, ainsi que par la revue et le partage du code.

Quand on cherche à automatiser des traitements, les concepts métiers doivent être « traduits » en objets informatiques. Cette action a un impact direct sur la qualité d'un programme, en particulier sur sa lisibilité. Nommer correctement les données et variables permet de partager le programme avec d'autres ou de le comprendre lors d'une relecture future[17].

Le rédacteur du code, gagne, comme le rédacteur d'une étude, à travailler la simplicité de son style, à écrire pour le plus grand nombre. Il existe des styles de programmation qui se distinguent assez simplement par la manière de :

- nommer les variables (par exemple `camelCase`, majuscule pour la première lettre des mots, sauf pour le premier, et `snake_case`, mots en minuscules séparés par des tirets bas, etc.) ;
- indenter les lignes de code (généralement par deux, trois ou quatre espaces) ;
- ou d'utiliser les commentaires.

> *« Un style efficace est surtout constant, homogène d'un programme à un autre. »*

Un style efficace est surtout constant, homogène d'un programme à un autre[18]. Un langage informatique respecte un ensemble de règles formelles strictes qui réduisent fortement la liberté d'écriture du développeur. Pour autant, le développeur peut organiser le programme suffisamment à sa guise pour développer un véritable style. Et ce style comprendra toujours une dimension visuelle, contrairement à la plupart des styles littéraires. En effet l'utilisation des passages à la ligne et des espaces en début de ligne, les indentations, jouent un rôle primordial dans la mise en valeur des structures logiques.

Quand on écrit un code, il faut se relire et se faire relire. La revue de programme (*code review*) est une pratique courante de gestion de la qualité du code, en complément des tests. Tant la forme que le fond du code y sont mis à l'épreuve. S'il ne dispose pas de collègues coopératifs, le selfeur gagnera simplement à prendre de la distance avec son code, en le laissant de côté quelques jours, quelques heures, quelques instants, et à le relire ensuite avec un regard neuf. S'il peine à rentrer dedans, c'est sans doute qu'il doit rendre son code plus accessible.

17. Pour de bonnes pratiques de nommage voir par exemple (McConnell, 2005, chapitre 11 ; Martin, 2009, chapitre 2).
18. La plupart des outils de développement proposent des fonctions de mise en forme automatique. Ces fonctions permettent de respecter un style défini *a priori*, soit par la communauté informatique dans son ensemble, soit par une équipe projet. Cependant les éditeurs de logiciels statistiques sont actuellement plus pauvres de ce point de vue (SAS® et R notamment) et contraignent à suivre manuellement les règles de style choisies.

Par ailleurs, on peut coder à plusieurs. Cela peut paraître évident mais c'est assez peu pratiqué. Deux selfeurs en binôme (pratique désignée par le terme de *pair programming*) feront très souvent beaucoup plus ensemble que s'ils étaient restés chacun de leur côté (Hannay *et alii*, 2009 ; Kessler et Williams, 2002). De surcroît, cela revient à une revue de code en continu. C'est un formidable outil de formation et de tutorat pour apprendre à supprimer le code inutile et à éviter les répétitions.

Au-delà de l'équipe, il est possible de s'inscrire dans une communauté plus grande, *via* le partage de programmes en *open source*. Dans ce cadre, on développe *de facto* pour les autres, la lisibilité du code est donc primordiale. Il est indispensable de respecter les normes retenues par la communauté du logiciel auquel on contribue, ces normes étant dans la plupart des cas des standards. S'inscrire dans une démarche *open source* est un moyen très puissant de progresser dans l'appropriation des bonnes pratiques. La communauté fait bénéficier de retours d'expérience plus nombreux et plus variés, elle devient un moyen de se former par tutorats réciproques. Depuis la mise à disposition du *modèle Inès* sur son site internet, l'Insee diffuse de plus en plus de programmes en *open source* (notamment les modèles, comme *Avionic*, *Destinie*, *Mésange*, *Mélèze* et *Omphale*[19]). En cela, l'institut s'inscrit pleinement dans une pratique devenue courante parmi les *data scientists*.

Dans un contexte qui lui est donc favorable, le statisticien public selfeur qui a su définir la finalité de son développement, qui sait structurer convenablement son programme et qui saura vérifier avec l'aide d'un tiers la lisibilité de son code, a encore un dernier choix à faire.

CÔTÉ TECHNIQUE…

Le terme de « technique » est source d'un débat toujours renouvelé : doit-on maîtriser la technique pour savoir définir une cible et construire le chemin qui y mène ?

Le statisticien selfeur doit-il se transformer en informaticien pour écrire un programme dans les règles de l'art ? Si l'on y regarde de plus près, toutes les activités étudiées jusqu'à présent relèvent d'une technicité particulière, que ce soit définir comment mesurer un concept, estimer un modèle, concevoir un enchaînement de traitements, écrire un programme, le tester, etc. Ce qui se cache derrière cette dernière question c'est la réalité de l'importance des outils utilisés pour réaliser les objectifs fixés.

> *« Pas de « techniques miracles » pour diminuer la quantité de bugs dans les programmes. »*

Dans les années quatre-vingt, l'ingénieur logiciel américain Frederick P. Brooks (Brooks, 1986) a utilisé une expression restée fameuse : « *No Silver Bullet »,* autrement dit il n'y a « pas de baguette magique », ou pas de « techniques miracles » pour augmenter la productivité des programmeurs et diminuer la quantité de *bugs* dans les programmes. Brooks estime que les difficultés de réalisation des logiciels se divisent en difficultés accidentelles (langages de programmation et systèmes laborieux et malaisés à utiliser) et en difficultés essentielles (inhérentes à la production de logiciels). Or, selon lui, les difficultés accidentelles ont déjà été en grande partie éliminées, par exemple par l'adoption de langages de haut niveau ; il n'y aura donc pas dans l'avenir de nouveaux progrès techniques permettant des gains importants de productivité. Les gains (en délai de réalisation, en qualité) doivent donc d'abord être cherchés dans le travail de conception, et ensuite seulement, dans le choix des outils.

19. Les codes-sources de ces modèles sont disponibles à l'adresse suivante : *https://github.com/InseeFr*.

Tout en gardant à l'esprit leurs limites, parmi les nombreux choix techniques à faire (*web* ou pas, format de la base de données, algorithmie, méthodes *big data* ou classiques, etc.), certains se révèlent plus importants que d'autres.

LE CHOIX LE PLUS STRUCTURANT EST CELUI DU LANGAGE…

Il n'est pour autant pas nécessaire de chercher la perfection : le langage à la syntaxe la plus subtile, permettant les syntaxes les plus courtes pour implémenter les traitements les plus complexes, ou autorisant de surprenantes opérations est rarement celui qu'il faut retenir. Les langages à haut niveau d'abstraction sont nombreux, et souvent suffisamment puissants pour la nature des travaux à réaliser. Le choix du langage se fera plutôt en prenant en compte son écosystème : évolue-t-il régulièrement ? Sa documentation est-elle abondante et facilement disponible ? Est-il porté par une communauté d'utilisateurs dynamique ? Existe-t-il des librairies tierces permettant de ne pas avoir à tout redévelopper ? Peut-il facilement s'interfacer avec d'autres langages ? Existe-t-il une offre suffisante d'outils de développement et de tests pour ce langage ?

Aujourd'hui, pour un statisticien développant lui-même ses programmes, le choix se portera en premier lieu sur les langages *R* et *Python*. Ils bénéficient tous deux d'un écosystème de qualité, et répondront à la plus grande partie des besoins en statistique[20].

… DES BIBLIOTHÈQUES DE PROGRAMMES SANS BOGUES…

On s'appuie généralement sur des programmes écrits par d'autres et mis à disposition sous forme de bibliothèques (*libraries* en anglais) réutilisables. En premier lieu, il faut s'assurer qu'elles ne présentent pas de *bugs*[21]. Ce point est particulièrement sensible dès que l'on touche au domaine de la méthodologie. Les traitements à implémenter peuvent être complexes et les erreurs générées par une mauvaise implémentation peuvent avoir un impact considérable sur le résultat, tout en étant difficiles à détecter. D'autres critères sont (sans chercher à être exhaustif) la performance, la pérennité, la disponibilité d'une documentation, la facilité d'utilisation, éventuellement le coût de la licence d'utilisation. Afin de guider le selfeur vers les librairies donnant les meilleures garanties, des initiatives de certification ont vu le jour. L'Insee a créé début 2019 un Comité de certification des packages R afin d'accompagner l'utilisation de plus en plus étendue de ce langage. Toujours au sein de la communauté R, on peut citer l'initiative rOpenSci[22] qui s'inscrit dans la logique de la science reproductible (voir *infra*).

… ET DES ÉDITEURS STANDARDS (ÉVITER LES PROPRIÉTAIRES)

Les interfaces entre les modules sont un endroit stratégique où il faut faire les bons choix. Les modules d'un même traitement doivent rester indépendants d'un point de vue technologique, sinon leurs cycles de vie ne seront pas indépendants. Pour cela il faut que les technologies choisies (type de fichiers de données utilisés par exemple) introduisent

20. Une liste d'outils utilisés dans la statistique publique est disponible à l'adresse suivante : *https://github.com/SNStatComp/awesome-official-statistics-software*.
21. Une telle vérification ne va pas de soi. Le mieux est de pouvoir s'appuyer sur un tiers qui a la compétence pour le faire. À défaut il faut disposer de jeux de tests permettant de valider la librairie.
22. Voir le site *https://ropensci.org/*.

peu de contraintes. Le choix devra donc se porter sur des standards faciles à produire et à manipuler, par les machines mais si possible aussi par les humains. Concrètement on s'efforcera d'échanger les informations sous forme de fichier textes dans des formats standards reconnus et non propriétaires (par exemple des fichiers CSV, JSON ou XML). L'usage de fichiers Excel millésimés (97, 2003, etc.) est source d'erreurs en cas de changements de version. Le format des fichiers de données SAS® l'est également avec d'autres logiciels statistiques.

L'utilisation des tableurs pour faire de la statistique est un sujet qui ne fait pas consensus. Leurs défenseurs mettent en avant leur facilité d'utilisation, aussi bien pour manipuler la donnée que pour produire des graphiques. Leurs détracteurs, dont font partie les auteurs de cet article, conseillent de ne pas utiliser ces outils pour des ensembles de données qu'il n'est pas possible d'afficher en entier sur un écran. En effet, un tableur rend impossible d'appliquer le conseil qui veut qu'un programme soit fait pour être lu avant d'être exécuté. Dans un tableur tout est mélangé : les données en entrée, les règles de calcul, l'architecture générale du traitement et le résultat. Utiliser des tableurs expose à des risques importants (Powell, Baker et Lawson, 2009) : difficulté à comprendre les traitements implémentés pour celui qui ne les a pas écrits (ou qui les a oubliés), quasi-impossibilité de construire des traitements modulaires et à mettre en œuvre des tests, corruption des traitements lorsque l'on remplace par erreur la formule contenue dans une cellule par une valeur.

> *« Un tableur rend impossible d'appliquer le conseil qui veut qu'un programme soit fait pour être lu avant d'être exécuté. »*

Pour répondre aux exigences de performance, il sera parfois nécessaire de choisir des outils spécifiques. La portée de ce choix devra être restreinte au maximum et il ne devra être fait qu'après la conception du traitement, une erreur classique étant de construire le programme de façon à mettre en valeur toute la puissance de l'outil retenu. Les choix d'outils pour des raisons de performance sont les plus à même d'être caduques au fur et à mesure que les performances globales de l'informatique évoluent. Par exemple SAS® et R ont souvent été opposés sur la question de la capacité à traiter des fichiers de grandes tailles, R nécessitant de les charger en mémoire vive ou de mobiliser un serveur de base de données, SAS® étant capable de les traiter séquentiellement à partir du disque dur[23]. Mais avec l'évolution des capacités matérielles et des possibilités logicielles le débat a perdu de sa pertinence. L'opposition se focalise maintenant plus sur la capacité offerte par les deux logiciels de pouvoir partager ses programmes avec d'autres. À nouveau, un traitement modulaire facilitera le travail : des modules bien définis permettront de ne pas faire porter le choix de la technologie au-delà du périmètre pour lequel cela se justifie, et si un jour le problème de la performance ne justifie plus une technologie spécifique, le retour vers des solutions standards sera facilité.

23. Ainsi, dans un article publié par SAS®, il apparaît que le chargement des données en mémoire vive pratiqué par *R* ne lui permet pas de manipuler des jeux de données aussi volumineux que SAS® (Ames, Abbey et Thompson, 2013).

GÉRER (AUTOMATIQUEMENT) SES VERSIONS, VIRTUALISER SON ENVIRONNEMENT

Pour suivre les modifications apportées à un programme, on peut ajouter à son nom une date ou un numéro de version. Manuellement, cela devient très vite fastidieux et source d'erreur. Il existe des outils pour enregistrer les versions d'un programme, accéder à l'historique, examiner les différences entre plusieurs versions, développer plusieurs versions en parallèles, etc. L'outil de référence en la matière est *Git*. Il nécessite un endroit où déposer et partager l'ensemble des versions du programme (tels que github.com et gitlab.com) mais facilite d'autant le travail en équipe.

L'utilisation fréquente de librairies et logiciels tierces et l'évolution rapide des langages informatiques nécessitent de référencer également les versions des outils utilisés et non pas seulement les versions du programme. Un programme pourra ne pas fonctionner avec une version plus ancienne ou plus récente que celle utilisée lors de son développement. C'est ce qu'on appelle **la gestion des dépendances**[24].

Afin de faciliter l'utilisation d'un ensemble d'outils de développement informatique, des environnements complets prêts à l'emploi existent[25]. Les environnements de développement « statistique » se sont perfectionnés et diversifiés (Besse, Guillouet et Laurent, 2018). Ils améliorent l'ergonomie de développement, la reproductibilité des résultats et aussi l'apprentissage des langages de programmation. En particulier, les carnets de code (*notebooks*[26]) permettent d'intégrer le code exécutable (et modifiable) au sein d'un rapport ou d'une présentation.

Enfin, la virtualisation consiste à créer une représentation virtuelle, basée logicielle, d'un objet ou d'une ressource telle qu'un système d'exploitation, un serveur, un système de stockage ou un réseau. Ces ressources simulées ou émulées sont en tous points identiques à leur version physique. La virtualisation permet même d'isoler un environnement d'exécution (*containers*) pour un projet[27]. Le statisticien public selfeur dispose maintenant d'un tel environnement avec le SSPCloud[28].

TESTER TOUT AU LONG DU DÉVELOPPEMENT, ET MÊME APRÈS

La démarche exposée précédemment (exigences, architecture, développement) permet de maîtriser la qualité du traitement *a priori*, lors de la conception. Les tests permettent de la maîtriser *a posteriori*, en fonctionnement réel. Il en existe de plusieurs sortes ayant des objectifs divers.

Une première catégorie, les **tests fonctionnels**, a pour but de s'assurer que les traitements donnent bien les résultats attendus du point de vue du métier. Au niveau le plus fin, il s'agit de tester que chaque fonction du programme est bien implémentée, ce sont les tests unitaires[29]. On testera par exemple la création, à partir de valeurs prédéfinies, d'un relevé de prix dans la base de données. Au niveau du programme dans son ensemble, il peut

24. Pour le langage R, *Renv* fait référence actuellement.
25. En anglais, on parle d'IDE pour *Integrated Development Environment*.
26. Les *notebooks Jupyter* ont inspiré au-delà de Python, en particulier Rmarkdown pour R.
27. Par exemple avec d*ocker* : *https://www.docker.com/*.
28. Voir à ce sujet l'article de Frédéric Comte, Arnaud Degorre et Romain Lesur sur le SSPCloud, dans ce même numéro.
29. Sur les tests unitaires, se reporter par exemple à (Martin, 2009).

s'agir de scénario reproduisant les comportements types d'un utilisateur, ou bien de calculs utilisant des jeux de données de test de grande taille et présentant des cas de figure variés. On testera par exemple le calcul d'un indice de prix à partir de données détaillées pour lesquelles le résultat est connu. Lorsque l'exécution de tests de cette première catégorie est automatisée, cela permet de s'assurer qu'il n'y a pas de régression fonctionnelle lors de l'ajout de nouvelles fonctions. Par exemple pour les retraitements d'une enquête répétée dans le temps dont les méthodes d'imputation seraient modifiées.

> *« Lorsque l'exécution de tests de cette première catégorie est automatisée, cela permet de s'assurer qu'il n'y a pas de régression fonctionnelle lors de l'ajout de nouvelles fonctions. »*

Une deuxième catégorie de tests permet de s'assurer de la qualité des aspects non fonctionnels, principalement la **performance** et la **sécurité**. Pour les programmes qui connaîtront un grand nombre d'évolutions au cours de leur vie, il est préférable d'automatiser ce type de tests.

A priori, les tests fonctionnels intéresseront plus particulièrement le statisticien, les aspects non fonctionnels étant vus comme l'affaire des informaticiens. Cependant il existe des cas où la maîtrise des exigences non fonctionnelles est un enjeu du métier statistique, que ce soit pour des raisons immuables, comme le respect du secret statistique, ou pour des raisons conjoncturelles, comme le besoin actuel de pouvoir traiter en des temps acceptables des données massives. Cette notion de performance a différentes dimensions. Il peut s'agir du temps de traitement, du volume maximal de données pouvant être pris en charge par le programme, du nombre d'utilisateurs simultanés (par exemple pour un outil *web* interactif). Aucun de ces objectifs de performance n'est un absolu en soi. C'est lors de la définition des besoins que l'on doit préciser lesquels doivent être atteints.

La plupart des tests peuvent, et doivent, être menés tout au long de la réalisation du programme. De ce point de vue, les tests de performance présentent une particularité. Les modifications apportées à un programme afin d'en améliorer les performances ont parfois pour effet d'en dégrader la structure interne et la lisibilité. De telles modifications ne doivent donc être apportées que si les performances du programme d'origine ne permettent pas de réaliser le traitement dans des conditions acceptables. Ainsi, les tests de performance doivent intervenir vers la fin du processus d'écriture, lorsque sa conception aura été menée jusqu'au bout. Les méthodes pour améliorer la performance sont diverses et leur impact sur la qualité du programme varie grandement (McConnell, 2005, chapitre 26). Il faut éviter de dégrader la structure et la lisibilité du programme pour atteindre des objectifs de performance qui ne sont pas nécessaires aux utilisateurs.

LE STATISTICIEN SELFEUR: UN DÉVELOPPEUR « AGILE », AU SERVICE DE LA QUALITÉ STATISTIQUE

Les grands principes énoncés jusque-là doivent bien évidemment s'adapter à la complexité du développement en self, et surtout à l'enjeu pour l'utilisateur du code produit. Ainsi, parmi les grands courants qui structurent aujourd'hui les méthodes de développement, celui de l'agilité est probablement le plus intéressant pour le selfeur : l'agilité a pour objectif d'orienter les efforts vers ce qui a le plus de valeur pour l'utilisateur, en s'adaptant aux changements à moindre coût.

Qu'il soit son propre utilisateur, ou qu'il insère son code dans un processus statistique complexe, le statisticien selfeur gagnera à s'inspirer des outils de pilotage d'un projet agile, si ce n'est à la lettre, du moins dans l'esprit.

Il gagnera aussi à s'inspirer des enjeux de la reproductibilité des études, ou des exploitations statistiques : pour cela, il s'assurera de livrer le résultat dans un environnement technique qui permet à tout un chacun de le reproduire à l'identique et de façon automatisée.

In fine, ce qui importe, c'est que le programme développé serve à produire une statistique publique de qualité : répondant à un besoin, dans les règles de l'art (statistique et informatique), documentée, reproductible, etc. Quel que soit le critère que l'on s'efforcera de respecter, « savoir coder » contribue à démontrer qu'on « sait compter », et ne peut que renforcer la confiance dans la donnée produite.

BIBLIOGRAPHIE

ABELSON, Harold, SUSSMAN, Gerald Jay, et SUSSMAN, Julie, 1996. *Structure and Interpretation of Computer Programs.* Juillet 1996. The MIT Press. Deuxième édition. ISBN 978-0262011532.

AMES, Allison J., ABBEY, Ralph et THOMPSON, Wayne, 2013. *Big Data Analytics. Benchmarking SAS®, R, and Mahout.* [en ligne]. Actualisé le 6 mai 2013. SAS Institute Inc., Cary, NC. Technical Paper. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://support.sas.com/resources/papers/Benchmark_R_Mahout_SAS.pdf*.

ANXIONNAZ, Isabelle et MAUREL, Françoise, 2021. Le Conseil national de l'information statistique – La qualité des statistiques passe aussi par la concertation. In : *Courrier des statistiques.* [en ligne]. 8 juillet 2021. Insee. N° N6, pp. 123-142. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.insee.fr/fr/statistiques/fichier/5398693/courstat-6-art-7.pdf*.

BESSE, Philippe, GUILLOUET, Brendan et LAURENT, Béatrice, 2018. Wikistat 2.0 : Ressources pédagogiques pour l'Intelligence Artificielle. In : *Statistique et Enseignement.* [en ligne]. 6 novembre 2018. Société française de statistique (SFdS). Volume 9, pp. 43-61. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *http://statistique-et-enseignement.fr/article/view/694*.

BONNANS, Dominique, 2019. RMéS, le référentiel de métadonnées statistiques de l'Insee. In : *Courrier des statistiques*. [en ligne]. 27 juin 2019. Insee. N° N2, pp. 46-57. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.insee.fr/fr/statistiques/fichier/4168396/courstat-2-6.pdf*.

BROOKS, Frederick P., 1986. No Silver Bullet – Essence and Accident in Software Engineering. In : *KUGLER, H.-J., 1986. Proceedings of the IFIP Tenth World Computing Conference.* [en ligne]. Elsevier Science BV, Amsterdam, pp. 1069-1076. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *http://worrydream.com/refs/Brooks-NoSilverBullet.pdf*.

COCKBURN, Alistair, 2000. *Writing Effective Use Cases*. Octobre 2000. Addison-Wesley. ISBN 978-0201702255.

CONSTANTINIDIS, Yves, 2018. *Expression des besoins pour le SI. Guide d'élaboration du cahier des charges*. 11 janvier 2018. Eyrolles. 4e édition. ISBN 978-2212675771.

COTIS, Jean-Philippe, TEMAM, Daniel, BENVENISTE, Corinne, ANGEL, Jean-William, DARRINÉ, Serge, ROUMIGUIÈRES, Eve et GÉLY, Alain, 2009. Savoir compter, savoir conter. In : *Courrier des statistiques*. [en ligne]. Décembre 2009. Insee. Hors Série. [Consulté le 13 décembre 2021]. *https://www.bnsp.insee.fr/ark:/12148/bc6p06xt18x*.

ERIKSON, Johan, 2020. Le modèle de processus statistique en Suède – Mise en œuvre, expériences et enseignements. In : *Courrier des statistiques.* [en ligne]. 29 juin 2020. Insee. N° N4, pp. 122-141. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.insee.fr/fr/statistiques/fichier/4497085/courstat-4-8.pdf*.

EVANS, Eric, 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* 20 août 2003. Éditions Addison-Wesley. ISBN 978-0321125217.

GADOUCHE, Kamel, 2019. Le Centre d'accès sécurisé aux données (CASD), un service pour la data science et la recherche scientifique. In : *Courrier des statistiques.* [en ligne]. 19 décembre 2019. Insee. N° N3, pp. 76-92. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.insee.fr/fr/statistiques/fichier/4254227/courstat-3-7.pdf*.

HANNAY, Jo E., DYBA, Tore, ARISHOLM, Erik et SJOBERG, Dag I. K., 2009. The Effectiveness of Pair Programming: A Meta-Analysis. In : *Information and Software Technology.* Juillet 2009. Elsevier. Volume 51, n° 7, pp. 1110-1122.

KESSLER, Robert et WILLIAMS, Laurie, 2002. *Pair programming illuminated.* 19 juillet 2002. Éditions Addison-Wesley. ISBN 978-0201745764.

L'HOUR, Emmanuel, LE SAOUT, Ronan et ROUPPERT, Benoît, 2016. *Savoir compter, savoir coder.* [en ligne]. Juin 2016. Insee. Document de travail, Méthodologie statistique, n° M2016/04. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.bnsp.insee.fr/ark:/12148/bc6p06zrjbz*.

LANGLAIS, Pierre Carl et EPRIST, 2020. *La recherche en crise de reproductibilité ?* [en ligne]. Avril 2020. EPRIST Analyse I/IST n°30. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *https://www.eprist.fr/wp-content/uploads/2020/04/EPRIST_I-IST_Recherche-en-crise-de-reproductibilite_Avril2020.pdf*.

MARTIN, Robert C., 2009. *Coder proprement.* Février 2009. Pearson France, collection Campuspress. ISBN 978-2744023279.

MARTIN, Robert C., 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Addison-Wesley. ISBN 978-0132911221.

MCCONNELL, Steve, 2005. *Tout sur le code – Pour concevoir du logiciel de qualité, dans tous les langages*. 14 février 2005. Microsoft Press. 2e édition. ISBN 978-2100487530.

POWELL, Stephen G., BAKER, Kenneth R. et LAWSON, Barry, 2009. Errors in Operational spreadsheets. In : *Journal of Organizational and End User Computing*. [en ligne]. Juillet-septembre 2009. pp. 24-36. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *http://faculty.tuck.dartmouth.edu/images/uploads/faculty/serp/Errors.pdf*.

ROBERTSON, Suzanne et ROBERTSON, James, 2013. *Mastering the Requirements Process: Getting Requirements Right*. Éditions Addison-Wesley Professional. Troisième édition. ISBN 978-0321815743.

VOLLE, Michel, 2001a. Pour une esthétique de la sobriété. In : *site de Michel Volle*. [en ligne]. 24 mars 2001. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *http://www.volle.com/opinion/sobriete.htm*.

VOLLE, Michel, 2001b. L'expression des besoins et le système d'information. In : *site de Michel Volle*. [en ligne]. 31 décembre 2001. [Consulté le 13 décembre 2021]. Disponible à l'adresse : *http://www.volle.com/travaux/besoins.htm*.