Caractérisation du temps de travail et de la durée nécessaires pour élaborer un logiciel de saisie et de gestion de données dans un laboratoire de recherche

Eric Quinton

Caractérisation du temps de travail et de la durée nécessaires pour élaborer un logiciel de saisie et de gestion de données dans un laboratoire de recherche

Eric Quinton [1]

IRSTEA - Unité de recherche Écosystèmes aquatiques et changements globaux
50, avenue de Verdun
33612 CESTAS, France
eric.quinton@irstea.fr

RÉSUMÉ. *Pour faciliter la saisie et la gestion des données de recherche, le laboratoire Écosystèmes Aquatiques et Changements Globaux d'Irstea a développé plusieurs logiciels ces cinq dernières années. Dix d'entre eux ont été étudiés pour répondre à deux questions : est-il possible de déterminer le temps de développement nécessaire à partir d'un indicateur simple, et quel délai faut-il prévoir avant que le logiciel puisse être considéré comme achevé ?*

Le temps de développement peut être corrélé au nombre de tables de la base de données relationnelle sous-jacente. Quant au délai nécessaire pour achever un logiciel, il s'établit aux alentours de 22 mois à partir de la mise en production de la première version, pour tenir compte des évolutions liées à la prise en main de celui-ci. Cela implique de maintenir des ressources disponibles pendant toute cette période pour assurer la réussite du projet.

ABSTRACT. *To facilitate the input and management of research data, the Irstea Aquatic Ecosystems and Global Changes laboratory has developed several software packages over the last five years. Ten of them were studied to answer two questions: is it possible to determine the development time required from a simple indicator, and how long should it take before the software can be considered as complete?*

The development time can be correlated with the number of tables in the underlying relational database. As for the time required to complete a software, it is established around 22 months from the start of production of the first version. This time is necessary to take into account changes induced by the handling of it. It implies to maintain available resources throughout this period to ensure a successful project.

MOTS-CLÉS : *logiciel, développement, maturité, coût*

KEYWORDS: *software, conception, maturity, cost*

1. Introduction

Pour mener leurs travaux, les laboratoires de recherche peuvent organiser des campagnes de collecte de données, que celles-ci soient physiques (récolte d'échantillons) ou immatérielles (données issues de capteurs, observations, interviews, etc.). Le stockage de celles-ci dans les environnements numériques s'appuie de plus en plus sur les systèmes de gestion de bases de données.

Les saisies manuelles sont souvent fastidieuses et sujettes à de nombreuses erreurs. Dans les feuilles de calcul, on estime que 50 % d'entre elles contiennent des erreurs, et que le nombre de cellules erronées est de l'ordre de 1 à 2 % (Panko, 2008). De plus, l'absence de contrôle des types de données amène souvent à mélanger des données numériques avec du texte dans la même colonne, soit involontairement (mauvaise codification d'une date), soit volontairement pour exprimer une préoccupation ou remplacer une valeur discrète par un intervalle. L'exploitation de ces informations avec des outils de traitement automatique n'en est que plus difficile.

Le recours à des logiciels de saisie dédiés, qui intègrent des contrôles de cohérence et garantissent le typage des données, est souvent une solution judicieuse, dès lors que la collecte est répétitive (campagnes de récolte étalées sur plusieurs mois ou années, par des opérateurs internes ou externes, etc.) et justifie le temps passé à leur conception. Ces logiciels sont le plus souvent développés avec des interfaces web, mais peuvent également être écrits pour fonctionner en mode autonome, dans des zones où le réseau Internet ne peut être garanti.

L'unité de recherche Écosystèmes Aquatiques et Changements Globaux d'Irstea a mis en place depuis plusieurs années une dizaine de logiciels dédiés à l'acquisition des données collectées ou à leur gestion. Le code de certains de ces logiciels a été ouvert et mis à la disposition de la communauté scientifique. Ils ont été réalisés par une seule personne (administrateur de bases de données, développeur d'applications, ancien administrateur de systèmes d'informations), qui s'est appuyée sur des méthodes de développement basées sur les principes du manifeste Agile (Beck *et al.*, 2001).

1.1. Durée de mise au point du logiciel

Dans le développement traditionnel ou cycle en V (Forsberg, Mooz, 1998), le cahier des charges est rédigé et transmis à l'équipe de développement. La vérification de la conformité du logiciel s'effectue en comparant les fonctionnalités produites avec celles demandées.

Dans le développement Agile, les spécifications du logiciel sont élaborées au fur et à mesure que les premiers modules sont livrés. Entre la livraison de la première version opérationnelle et le moment où le logiciel peut être considéré comme achevé, il peut s'écouler plusieurs mois. Ce délai est nécessaire pour que le « client » puisse le découvrir, le faire évoluer pour qu'il réponde parfaitement à ses besoins, et enfin se l'approprier pour pouvoir l'utiliser. Ce processus d'appropriation a été modélisé en

trois phases : la découverte de la technologie, l'exploration, l'évaluation et l'adaptation et enfin l'utilisation proprement dite (figure 1) (Mendoza *et al.*, 2010).

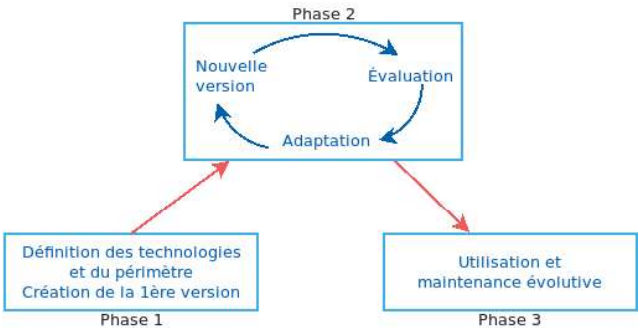

Figure 1. Les phases d'appropriation d'un logiciel (modifié d'après Mendoza et al.*). La première phase correspond à l'élaboration de la première version du logiciel, la seconde à son adaptation pour la rendre totalement utilisable et corriger les bogues de programmation. La dernière phase, dite de maintenance évolutive, est celle de son utilisation en routine.*

La phase de découverte permet de définir les contours techniques (application web ou client lourd embarqué, par exemple), mais également le périmètre de ce qui va être informatisé. Elle s'achève par la livraison de la première version utilisable.

La phase d'exploration est celle de l'adaptation. Elle comprend plusieurs étapes, comme le signalement des erreurs de programmation, la demande d'ajout de nouvelles fonctionnalités au fur et à mesure de la prise en main ou des évolutions ergonomiques. Compte-tenu du processus itératif de conception, elle peut s'étaler sur plusieurs mois, même si chaque version peut ne nécessiter que quelques jours de travail : les utilisateurs, en raison de leurs multiples activités et du délai nécessaire au processus cognitif d'appropriation, ont besoin de temps pour réaliser l'évaluation et exprimer leurs besoins complémentaires. C'est d'autant plus vrai dans les petites sociétés informatiques (Sánchez-Gordón, O'Connor, 2016), où répondre au besoin du client est vital, et où l'absence quasi-totale de cahier des charges impose une adaptation permanente. Le travail de développement dans un laboratoire de recherche peut s'apparenter à celui d'une petite société : il est rare de passer du temps à formaliser des processus s'il n'est pas nécessaire de le faire.

Enfin, la phase d'utilisation commence quand le logiciel est considéré comme mature. Il est alors utilisé en routine, et les évolutions, plus rares, sont dépendantes soit de la découverte tardive de bogues, soit de modifications dans les processus de gestion des données, soit de l'évolution des technologies utilisées (montées de version des *frameworks*, correctifs de sécurité, etc.).

La durée de la seconde phase est primordiale à connaître : même si elle nécessite peu de travaux, elle durera probablement plusieurs mois et est nécessaire pour mettre

au point le logiciel. Si les ressources ne sont pas disponibles pendant cette phase, le risque d'abandon du projet est grand faute de pouvoir l'adapter à la demande réelle.

1.2. Charge de travail

Un autre critère important avant de démarrer le développement d'un logiciel est celui de l'estimation de la charge de travail qui sera nécessaire.

Elle devrait être réalisée au plus tôt après le démarrage du projet, mais nécessite toutefois un certain recul. Pour la déterminer, deux approches sont en général utilisées : soit une estimation « à dire d'expert », soit l'utilisation d'une méthode de calcul formalisée. Jørgensen *et al.* (2009) ont estimé qu'il était probablement plus approprié de mixer les deux approches (même si l'une ou l'autre, dans certains cas, pouvait être plus adaptée), en utilisant une méthode de calcul et en la pondérant avec le regard d'un expert.

L'estimation du temps passé sur un projet de développement a largement été étudié. Papatheocharous *et al.* (2017) ont travaillé à la caractérisation des différentes phases (planification, développement, test, etc.). Al-Sabbagh et Gren (2018) se sont intéressés, quant à eux, à la relation entre la maturité de l'équipe et sa vitesse de développement. Si, depuis Boehm *et al.* (1995) et la création de la méthode COCOMO, le nombre de lignes produites reste une référence pour inférer le temps de réalisation, la manière de prédire ce nombre de lignes s'oriente de plus en plus vers l'attribution de points selon la complexité des tâches à réaliser, que ce soit à partir des cas d'utilisation produits en UML (Unified Model Language : http://www.uml.org/) (Clemmons, 2006 ; Usman *et al.*, 2014), ou des *story points* (Coelho, Basu, 2012), largement utilisés lors des développements réalisés selon l'Extreme Programming (https://www.agilealliance.org/glossary/xp/). Sampaio *et al.* (2010) se sont intéressés, quant à eux, aux facteurs de productivité et aux stratégies à mettre en œuvre pour les maximiser et limiter les facteurs défavorables. En s'appuyant sur la théorie analytique de l'investissement dans le projet (Chen, 2006), Liu *et al.* (2015) ont démontré la relation dynamique entre la durée du projet, son niveau d'incertitude, son coût initial et sa productivité.

Tous les indicateurs proposés sont basés sur la quantification formelle des besoins : cas d'utilisations, *story points*, calcul du nombre de lignes, etc., et le recours à des métriques adaptées est un préalable obligatoire. Elles nécessitent une analyse complète des besoins selon certains formalismes (cas d'utilisation, par exemple). Cette étape est rarement réalisée pour des logiciels de petite taille.

Lors de la création d'une application de gestion de données, la conceptualisation de la base de données en est une des premières étapes. Le postulat est de considérer que les traitements prévus induisent la structure de la base de données sous-jacente, et que la structure de celle-ci est (ou sera) le reflet des processus informatisés. Nous avons choisi d'étudier s'il existait une relation entre la structure de la base de données et le temps passé.

Les bases de données relationnelles sont composées principalement de tables, qui sont reliées par des contraintes d'intégrité et sur lesquelles sont définis des index. Des méthodes de calcul de leur complexité ont été proposées (Calero *et al.*, 2001 ; Pavlic *et al.*, 2008). Elles s'appuient sur divers indicateurs comme le nombre d'attributs, le nombre de clés étrangères, le nombre d'index, etc. Toutefois, il n'est pas sûr que ces indicateurs soient pertinents pour définir le temps nécessaire à la réalisation d'un logiciel, ils renseignent surtout sur la complexité intrinsèque de celles-ci. Par contre, le nombre de tables, qui sont des regroupements logiques d'informations, pourrait permettre d'appréhender la complexité du logiciel. Celui-ci propose en général divers interfaces qui permettent de consulter ou de manipuler les informations qu'elles contiennent, et celles-ci sont en général conçues pour « coller » à la structure de la base de données sous-jacente (à une table correspond fréquemment une interface de saisie dédiée).

Lors des développements réalisés selon des méthodes agiles, l'ensemble des fonctionnalités qui seront fournies ne sont en général pas connues initialement. C'est au cours des différentes itérations qu'elles seront exprimées. Toutefois, au moment du lancement du projet, le commanditaire a déjà une idée assez précise de ce qu'il attend. Si des cas d'utilisation peuvent alors être décrits, ils reflètent en général assez mal la complexité sous-jacente. La nature des données manipulées, tant en entrée qu'en sortie, va influer fortement sur le codage nécessaire.

Notre postulat est de considérer que le nombre de tables nécessaires pour représenter et traiter l'information peut être un indicateur pertinent d'estimation du temps de réalisation d'un logiciel. En effet, entre une structure de données en deux dimensions (un tableau) et des structures plus complexes nécessitant de recourir à plusieurs tables pour les représenter, la charge de travail ne sera pas identique : des écrans de saisie supplémentaires seront nécessaires, des traitements de contrôle ajoutés, la documentation technique sera plus volumineuse, etc. Lors des études préalables, les données disponibles et celles qui sont attendues sont répertoriées, et c'est en fonction de celles-ci que la structure de la base de données va être construite, avant le codage proprement dit. Cette structure peut être issue d'une modélisation en classes (on ne retiendra alors que les classes persistantes qui pourront être couplées avec les tables de la base de données), ou directement depuis un modèle logique, qu'il soit complet (avec tous les attributs) ou non. Estimer la volumétrie du développement à partir du nombre de tables serait, dans ce contexte, assez simple à mettre en œuvre, et n'a guère fait l'objet de travaux à ce jour.

Nous avons cherché à répondre à deux questions :

– est-il possible d'estimer *a priori* le temps de travail qui sera nécessaire pour réaliser un logiciel en se basant sur le nombre de tables présentes dans la base de données relationnelle sous-jacente ?

– compte-tenu du processus de développement basé sur les principes du manifeste *Agile*, quelle durée faut-il prévoir pour qu'un logiciel soit considéré comme stable et ne fasse plus l'objet que d'opérations de maintenance techniques ou fonctionnelles ?

2. Méthodologie

2.1. Logiciels étudiés

En cinq ans, le laboratoire EABX a créé plusieurs logiciels permettant la saisie ou la gestion des données acquises sur le terrain. Nous en avons retenu dix suffisamment aboutis pour cette étude. La plupart ont été conçus pour fonctionner avec une interface Web et ont été écrits en PHP/HTML. Trois d'entre eux ont été écrits en Java avec une interface native écrite avec le composant *Swing* de Java (https://fr.wikipedia.org/wiki/Swing_(Java)), deux parce qu'ils devaient fonctionner en mode autonome (pas de connexion Web possible), et le dernier parce qu'il était destiné à des opérateurs et devait être facile à installer dans un poste de travail. La liste des logiciels retenus dans cette étude est décrite dans le tableau 1.

Tableau 1. Liste des logiciels créés

Nom	Description	Langage utilisé
Sturwild	Enregistrement des déclarations de captures accidentelles d'esturgeon d'Europe	PHP/HTML
Sturio	Gestion de la station d'élevage des esturgeons d'Europe	PHP/HTML
Sturatj	Enregistrement des captures scientifiques d'esturgeon d'Europe dans l'estuaire de la Gironde	Java
Pometweb	Enregistrement des relevés de pêche effectués dans le cadre du calcul de l'indicateur DCE pour les estuaires	PHP/HTML
Transect-php	Suivi des pêches scientifiques réalisées au droit de la centrale nucléaire du Blayais	PHP/HTML
TransectJ	Logiciel complémentaire du précédent, permettant de saisir les paramètres physico-chimiques et techniques de la pêche directement sur le bateau	Java
Otolithe	Logiciel de lecture des pièces calcifiées de poissons	PHP/HTML
Alisma	Saisie des relevés floristiques effectués dans le cadre de l'indicateur DCE pour les cours d'eau	Java
Collec-science	Enregistrement et gestion des échantillons scientifiques	PHP/HTML
Usact	Données d'enquêtes sociologiques concernant les conflits d'usage	PHP/HTML

Trois logiciels ont eu leur code source publié : Alisma (https://github.com/Irstea/alisma), Collec-Science (https://github.com/Irstea/collec) et Otolithe (https://github

.com/Irstea/otolithe). Les codes des autres programmes sont disponibles dans une forge interne à Irstea.

Les logiciels Web permettent d'alimenter une base de données relationnelle PostgreSQL (https://www.postgresql.org/), et ceux écrits en Java fonctionnent avec une base de données HSQLDB (http://hsqldb.org/), dont le moteur a la particularité d'être également écrit en Java et ne nécessite pas d'installer des outils complémentaires dans le poste de travail.

2.2. Données quantifiées

Le développeur a enregistré son temps de travail sur les différents projets en recourant au logiciel *Hamster Time Tracker* (https://github.com/projecthamster/). Il a également utilisé GIT (https://git-scm.com/) pour gérer son code, et a identifié les versions avec une numérotation à trois chiffres (Preston-Werner, 2013), qui permet d'identifier les versions majeures (numérotées sur un ou deux chiffres) des versions correctives (numérotées sur trois chiffres).

Le délai nécessaire à la mise au point du logiciel a été estimé en calculant le nombre de jours calendaires entre la première version mise en production et la version considérée *à dire d'expert* comme stable.

L'ensemble de ces données est récapitulé dans le tableau 2.

Tableau 2. Détails des logiciels créés

Nom	Nbre de lignes de code	Nbre de tables	Nbre de versions majeures	Nbre de versions totales	Nbre d'heures passées	Durée calendaire (en jours) de mise au point de la version stable
Sturwild	6811	21	2	7	139	680
Sturio	28587	103	8	31	491	685
Sturatj	17161	30	3	10	311	645
Pometweb	8401	22	6	9	171	679
Transect-php	6942	27	6	20	193	516
TransectJ	5623	9	3	7	102	454
Otolithe	7573	15	6	12	239	729
Alisma	16786	22	3	7	326	395
Collec-science	16898	31	6	23	596	701
Usact	10737	73	2	6	114	275

Le nombre de lignes de code ne prend en compte que le code spécifique à l'application, c'est à dire sans le code du *framework*. Il s'agit d'un nombre estimatif, qui peut différer selon les règles de formatage du texte implémentées dans les ou-

tils de conception (retour ou non à la ligne lors de la description de tableaux, règles de mise en forme des commentaires, etc.). Le *framework* PHP (https://github.com/equinton/prototypephp) est identique pour toutes les applications. Les logiciels Java ont été construits avec la même architecture technique sous-jacente. Le nombre de tables des bases de données qui sont alimentées par les logiciels a été comptabilisé, sans discriminer les tables de référence [1] des autres. Le nombre des versions a été calculé en recherchant les étiquettes (tag) positionnées dans le dépôt de code. Le nombre d'heures a été extrait des heures enregistrées dans *Hamster Time Tracker*. Ce temps inclut le développement, les réunions internes et la reprise éventuelle des données pour alimenter la base de données. Concernant le logiciel Collec-Science, celui-ci a fait l'apport de développements réalisés par des tiers pour intégrer de nouvelles fonctionnalités (gestion des méta-données, mécanisme de traduction pour supporter plusieurs langues notamment) ou améliorer certains aspects du logiciel. Le temps qu'ils y ont consacré n'a pas été intégré dans les calculs. Toutefois, le code réalisé est relativement minime en taille par rapport à l'ensemble de l'application, même s'il a pu être décisif dans la conception globale du produit.

Les logiciels ont été élaborés sur un laps de temps de 5 ans (figure 2). Le logiciel Alisma se distingue par une période d'écriture de la première version stable exceptionnellement longue comparée aux autres applications, en raison d'une indisponibilité des scientifiques chargés de réaliser les tests initiaux.

Enfin, les logiciels Otolithe, Alisma et Collec-Science sont diffusés en dehors du laboratoire et sont disponibles depuis Internet. Ils ont fait l'objet de travaux complémentaires, comme le dépôt à l'Agence de Protection des Programmes ou la rédaction de documentations techniques complémentaires, voire la création d'un site Web pour Collec-Science.

3. Résultats

3.1. Calcul de la charge de travail

Parmi les dix logiciels analysés, trois ont été développés en Java et sept en PHP/Html. Trois d'entre eux ont fait l'objet de travaux complémentaires pour être diffusés : Alisma (Java), Collec-Science et Otolithe (PHP).

Il est probable qu'un logiciel plus complexe nécessite plus de temps à être réaliser. Nous disposons de plusieurs métriques : le temps passé quantifié en heures, le nombre de lignes de code, le nombre de tables de la base de données sous-jacente et le nombre de versions produites.

1. Tables dont les valeurs évoluent peu et qui permettent d'alimenter les listes de sélection : taxons, sexe des animaux, etc.

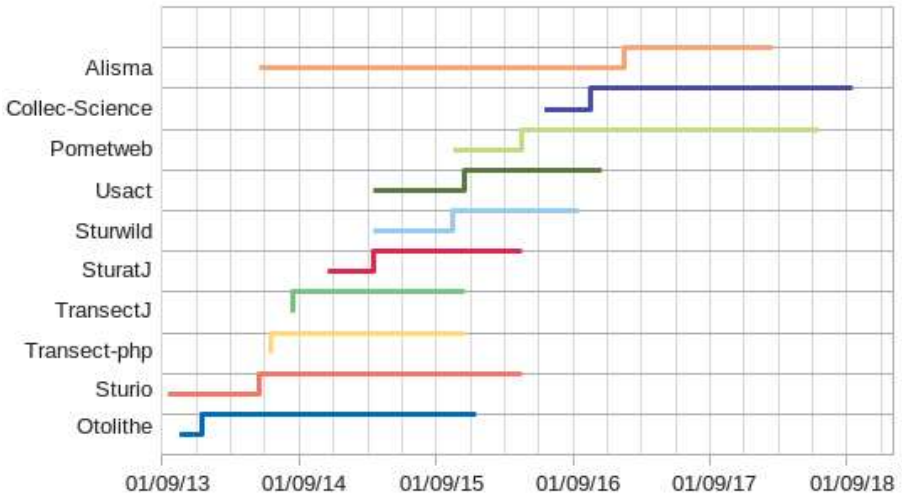

Figure 2. Calendrier de création des logiciels.
La partie basse de chaque ligne correspond à la période d'élaboration de la première version opérationnelle, la partie haute à la période de mise au point de la version stabilisée (cf. § 3.2). Le calendrier a été élaboré à partir du logiciel de gestion de code, et n'intègre pas les travaux préparatoires ou de création de la base de données précédant le début du codage.

3.1.1. Nombre d'heures rapportées au nombre de lignes de code

Nous avons d'abord analysé la vitesse de création du code, en calculant le nombre de lignes générées par heure (figure 3).

Pour la plupart des logiciels, la valeur est relativement proche (médiane de 51,5 lignes par heure pour les six logiciels aux centre du graphique).

Quatre logiciels ont des valeurs différentes. La productivité pour Usact est nettement plus importante (94 lignes par heure). Il s'agit d'un logiciel composé d'un nombre important de tables (73), mais qui sont toutes organisées sur le même modèle (règles de nommage et structuration identique des tables) : des *copier-coller* du code ou des opérations de factorisation ont permis d'accélérer largement le codage. Les travaux concernant Transect-php intègrent deux modules complexes : l'intégration des données issues de TransectJ, et la mise au point d'une interface de saisie adaptée à l'enregistrement des mesures réalisées en laboratoire. En ce qui concerne Otolithe, ce logiciel intègre un module écrit en Javascript qui permet de positionner des points sur une photo. La mise au point de ce module a été relativement longue, et justifie un temps passé plus important. Enfin, le logiciel Collec-Science intègre de nombreuses fonctionnalités avancées (lecture de codes-barres, gestion de méta-données, etc.) qui ont également été complexes à mettre au point. Ce logiciel a en outre été diffusé large-

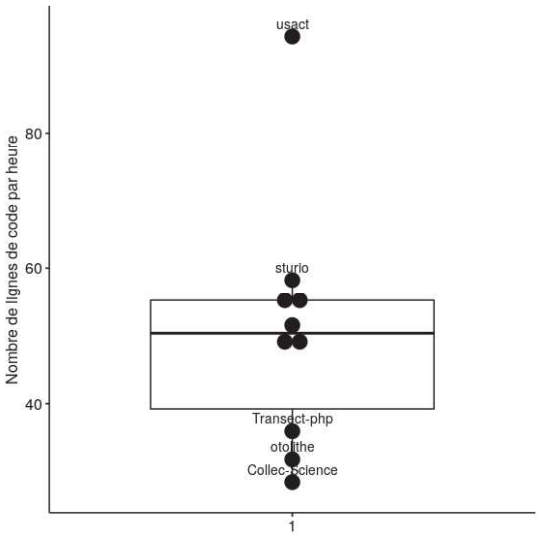

Figure 3. Nombre de lignes de code produites en une heure. La médiane vaut 51,5.

ment, et de nombreuses tâches annexes se sont rajoutées au temps de développement (rédaction de documentation technique, création d'un site Web, etc.).

Le nombre de lignes par heure peut être utilisé pour déterminer *a posteriori* si un logiciel a été plus ou moins complexe à produire, étant entendu que cette complexité peut être liée soit à un nombre de tâches annexes plus important, soit à des processus plus difficiles à coder.

3.1.2. Estimation du temps nécessaire à partir du nombre de tables

Le nombre de lignes de code rapporté au temps passé permet d'identifier les logiciels « classiques » de ceux qui sont plus consommateurs de temps. Mais l'estimation du nombre de lignes est complexe à mener, notamment dans les conceptions de type Agile : le dossier d'analyse n'est souvent que partiel et ne permet pas de disposer des métriques nécessaires pour ce calcul.

L'indicateur qui semble le plus facilement mobilisable pour estimer le temps nécessaire à la création d'un logiciel est le nombre de tables présentes dans la base de données. Nous avons vu précédemment que le temps nécessaire pour produire des lignes de code est assez constant d'un logiciel à l'autre.

La figure 4 permet de comparer le nombre de lignes produites par table et le nombre d'heures passées par table.

Elle permet de mettre en évidence plusieurs points.

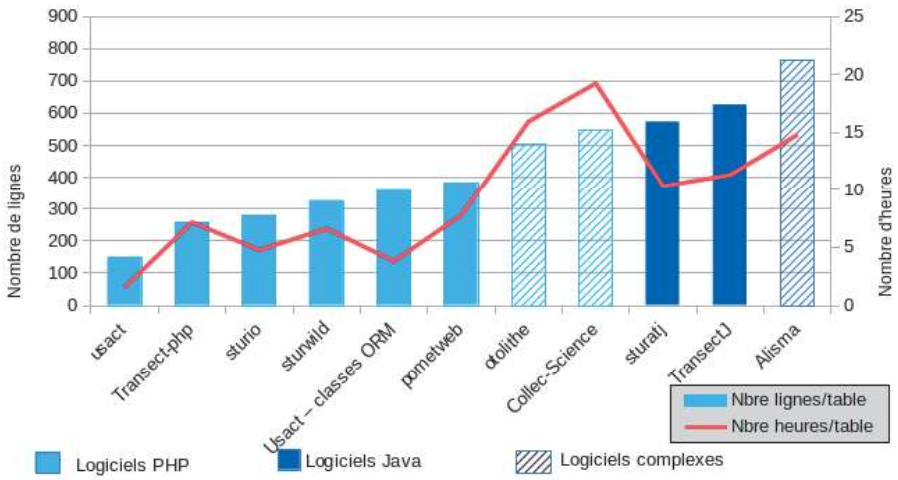

Figure 4. Nombre de lignes de code rapportées au nombre de tables.

D'une part, le nombre de lignes de code produit par table est plus important en Java qu'en Php (plus de 70 % de lignes en plus). Si trois logiciels se distinguent avec un nombre de lignes de code plus important (Collec-Science, Otolithe et Alisma), ils ont fait l'objet de fonctionnalités complémentaires, comme l'impression d'étiquettes et la lecture de codes-barres, la gestion de méta-données pour le premier, la gestion du positionnement de points sur des photos pour le second, et le calcul d'un indicateur par interrogation d'un service Web et l'ajout de fonctions d'exportation évoluées pour le troisième.

D'autre part, le temps nécessaire pour produire le logiciel rapporté au nombre de tables est assez constant pour un même langage de programmation. Les trois valeurs maximales (Otolithe, Collec-Science et Alisma) représentent l'effort nécessaire pour mettre au point les logiciels complexes, qui imposaient la mise place de solutions innovantes.

Le cas du logiciel Usact est particulier. S'il présente un nombre de lignes par table anormalement faible, c'est lié à la structure même de la base de données. Celle-ci est composée de 73 tables, mais la plupart d'entre elles ont des structures similaires. Dans les logiciels modernes, l'accès aux données stockées dans la base de données s'effectue par l'intermédiaire de classes dédiées qui encapsulent les différents accès (lecture, écriture, etc.). Elles sont souvent décrites sous le terme d'ORM : *Object Relational mapping*. Dans le cas du logiciel Usact, en raison de la conception générique du code produit pour gérer les tables de paramètres, le nombre de classes relevant du *mapping* de tables est de 30, soit une valeur significativement inférieure au nombre total de tables. En utilisant ce chiffre, le ratio du nombre de lignes par table se retrouve dans la moyenne des autres applications. Ainsi, si le nombre de tables est un indicateur facilement mobilisable, il ne semble être qu'une approximation du nombre de classes

ORM utilisées dans l'application. Ces deux chiffres sont très proches dans la plupart des cas, mais si les caractéristiques de la base de données et les méthodes de programmation l'imposent, il pourrait être utilement remplacé par le nombre de classes *ORM* générées.

Un biais dans ces estimations pourrait être lié à la capitalisation de l'expérience par le développeur : certains mécanismes mis au point pour un logiciel pourraient être utilisés dans d'autres, ce qui entraînerait un gain de temps lors de l'écriture des logiciels les plus récents. Après analyse, seul le logiciel Collec-Science a bénéficié de la mise au point de nouvelles fonctionnalités issues d'autres logiciels. On ne peut pas considérer que ce biais soit pertinent dans ce contexte.

À partir des éléments recueillis, il est possible d'envisager un indicateur simple de calcul du temps nécessaire, basé sur le nombre de tables (ou de classes *ORM* implémentées dans l'application) :

$$t = N \times ELC$$

ou *N* est le nombre de tables ou de classes *ORM*, et *ELC* un indicateur de complexité, composé de (E) : la vitesse intrinsèque de développement de l'équipe ou du développeur (expérience, gestion interne, organisation, etc.), (L) : du langage cible, et (C) : de la complexité intrinsèque liée du logiciel.

Les valeurs de E et de C sont difficiles à estimer. Il existe des indicateurs qui permettent de définir la complexité intrinsèque du code, comme l'indicateur « Cognitive complexity » (CC) (Campbell, 2018), qui compte notamment le nombre d'imbrications dans le code, et qui est calculé automatiquement notamment par SonarQube (https://sonarcloud.io). Nous avons cherché à savoir si la complexité ressentie, c'est à dire liée à la mise au point de nouveaux processus de traitement des informations, pouvait être corrélée à CC. Pour cela, les valeurs de E, L et C ont été empiriquement estimées, puis la valeur de CC a été calculée à partir de SonarQube. Comme il s'agit d'une valeur absolue, elle a été ramenée au nombre de tables de la base de données, pour permettre les comparaisons. Les résultats sont présentés dans le tableau 3.

La relation entre le facteur de complexité et le *cognitive complexity* divisé par le nombre de tables a été reportée dans la figure 5. Hormis pour Collec-Science, les valeurs sont très proches.

On peut ainsi considérer que l'estimation de E pour une équipe de développement pourrait être réalisée en se basant sur l'indicateur *Cognitive complexity* calculé sur ses productions précédentes, dès lors que l'on connaît le temps passé sur chaque projet et le langage utilisé.

L'approche présentée ici permet de définir, *a posteriori*, la vitesse intrinsèque de développement d'une équipe, résultat à la fois de son organisation, de ses compétences propres, de son implication, etc. Elle ne règle pas la difficulté d'estimation de la complexité du logiciel à écrire, qui est réalisée souvent de manière empirique à

Tableau 3. Simulation d'utilisation de l'indicateur sur le jeu de données
E : vitesse intrinsèque de développement, L : coefficient lié au langage, C : facteur de complexité, t : temps calculé,
CCt : Cognitive complexity divisé par le nombre de tables

Logiciel	Nb tables	Nb heures	E	L	C	t (en heures)	écart d'estimation du temps	CCt
Usact	73	114	7	1	0,2	102	-10 %	17
Transect-php	27	193	7	1	1	189	-2 %	90
Sturio	103	491	7	1	0,7	504	3 %	33
Sturwild	21	139	7	1	1	105	6 %	66
Usact-ORM	30	114	7	1	0,5	105	-8 %	42
Pometweb	22	171	7	1	1	154	-10 %	75
Otolithe	15	239	7	1	2	210	-12 %	145
Collec-Science	31	596	7	1	3	651	9 %	105
SturatJ	30	311	7	1,5	1	315	1 %	43
TransectJ	9	102	7	1,5	1	94	-7 %	44
Alisma	22	326	7	1,5	1,5	346	6 %	81

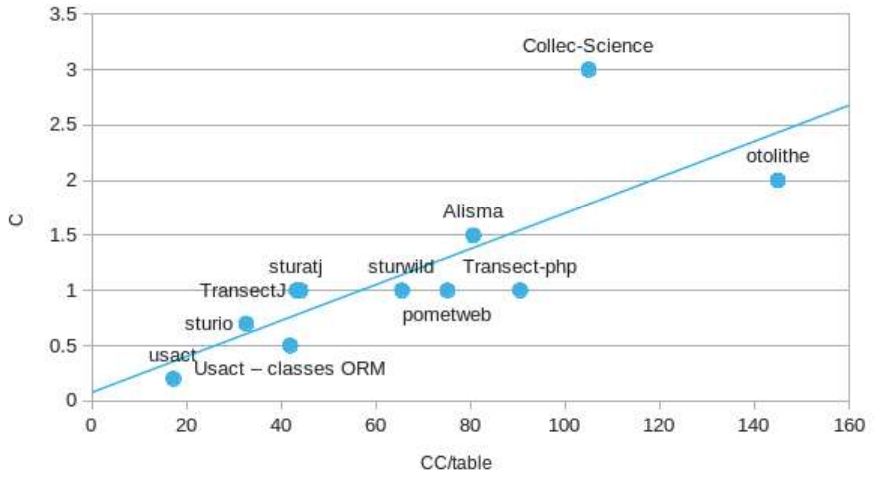

Figure 5. Relation entre le facteur de complexité (C) et le cognitive complexity divisé par le nombre de tables (CC/table)

« dire d'expert », ou en mobilisant des indicateurs basés sur des analyses plus fines, comme les *Story points* (Coelho, Basu, 2012), par exemple, s'ils sont disponibles.

L'autre inconnue, lors du démarrage d'un projet, tient à la difficulté à connaître la structure totale de la base de données, et notamment dans les développements agiles : sa structure est en général construite au fur et à mesure des *sprints* (en méthode SCRUM) (Morien, 2005). Dans la pratique, même si le détail de la base de données n'est pas encore connu dès le démarrage, la mise en place du projet implique d'en avoir une vision assez globale. Si le modèle détaillé, avec toutes les colonnes nécessaires, n'est pas encore élaboré, les principales entités (tables ou objets persistants, selon les méthodes d'analyse utilisées) sont définies lors des premiers travaux, et devraient être suffisantes pour quantifier le nombre de tables, au moins pour les besoins exprimés initialement.

Enfin, l'indicateur présenté ici ne peut fonctionner que pour les logiciels permettant de manipuler les informations stockées dans des bases relationnelles. Toutefois, compte-tenu du faible nombre de données, sa pertinence nécessiterait d'être testée dans d'autres contextes.

3.2. Calcul de la durée nécessaire pour qu'un logiciel arrive à maturité

Pour estimer la durée de mise au point d'un logiciel, la durée calendaire entre chaque version majeure a été étudiée (figure 6).

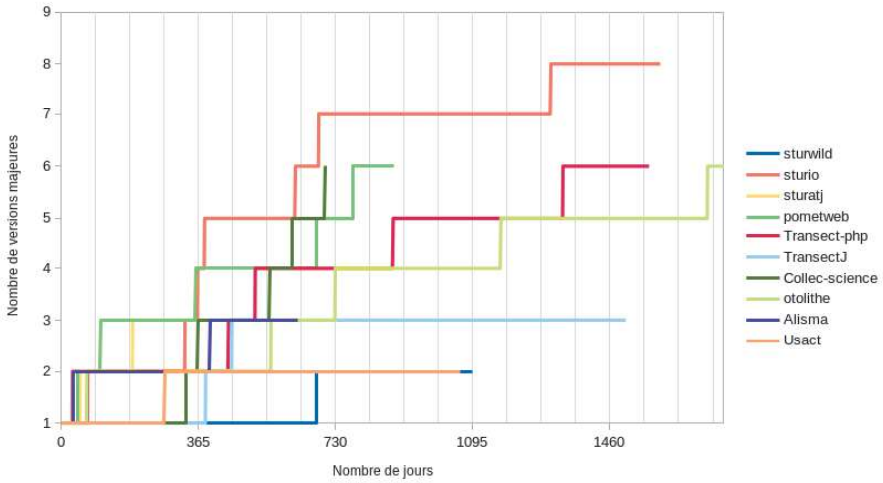

Figure 6. Durée entre chaque version majeure
Les courbes les plus longues correspondent aux logiciels les plus anciens. Dès lors qu'aucune nouvelle fonctionnalité n'est rajoutée (pas de nouvelle version majeure) pendant un laps de temps suffisamment important – environ 9 mois ici, le logiciel est considéré comme stabilisé.

Dès lors que la durée entre deux versions s'espace ou qu'une version majeure a une durée de vie approximativement supérieure ou égale à neuf mois, le logiciel est considéré comme stabilisé. Ce laps de temps a été déterminé *à dire d'expert*, en analysant chaque logiciel et l'ensemble des versions produites. Tous les logiciels ayant fait l'objet de modifications après la mise en production de la première version, aucun logiciel n'a été considéré comme stable avant la sortie de la seconde.

Cette durée s'établit entre 275 et 729 jours, pour une moyenne de 609 jours (20 mois) et une médiane de 662 jours (22 mois) (figure 7).

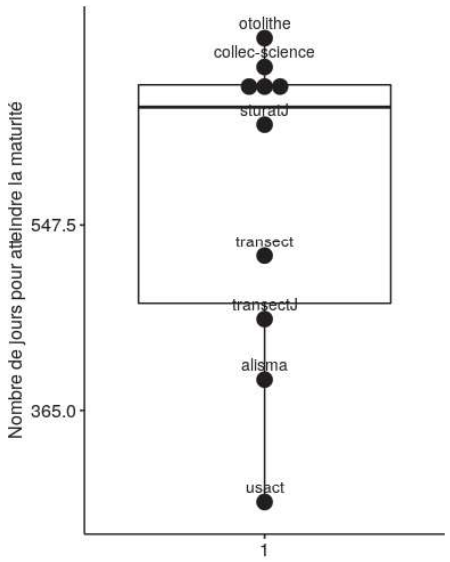

Figure 7. Nombre de jours nécessaires pour qu'un logiciel puisse être considéré comme mature.

Les logiciels Usact et Alisma se distinguent par des durées sensiblement plus faibles par rapport aux autres. Cela peut s'expliquer par le fait qu'ils ont été conçus à partir de logiciels déjà existants qui nécessitaient une réécriture (code non fonctionnel ou écrit avec des technologies obsolètes principalement) : les besoins étaient déjà connus et codés dans les outils précédents, et les utilisateurs n'ont eu besoin que d'évolutions limitées après la mise en production de la première version. Le logiciel TransectJ se caractérise par sa simplicité : il ne sert qu'à l'enregistrement de paramètres physico-chimiques lors d'opérations de pêche (un seul masque de saisie).

Six logiciels ont des valeurs très proches. En ne tenant pas compte des trois plus faibles valeurs qui s'expliquent assez facilement, l'écart n'est que de 213 jours (7 mois) entre Transect-PHP et Otolithe. Il semble raisonnable de penser que la durée nécessaire pour qu'un logiciel arrive à maturité s'établisse dans une fourchette entre dix-huit mois et deux ans, hors logiciels très simples ou inscrits dans un processus de recodage. Cette durée ne semble pas dépendante de de la prise en main du logiciel.

Les différentes applications créées sont utilisées, pour la plupart, par des personnes différentes, il n'y a pas d'effet lié à l'apprentissage de l'ergonomie. Il s'agit bien du délai nécessaire à la phase d'appropriation, qui est indépendante également de l'équipe de développement.

3.3. Discussion quant à la qualité des données

Le travail mené est empirique. Les temps de développement sont basés sur un enregistrement « au fil de l'eau » des tâches réalisées. Il s'apparente largement aux relevés réalisés par Todorovsky (1997 ; 2014), qui a calculé la répartition de ses tâches universitaires tout au long de sa carrière, entre enseignement, travaux scientifiques, tâches administratives, etc.

Le nombre de logiciels étudiés est faible, et ne permet pas des vérifications statistiques sérieuses. Toutefois, les données ont une certaine cohérence. Les logiciels ont été réalisés sur un laps de temps relativement court (5 ans), sans évolution ou rupture technologique majeure. Ils ont été écrits par un seul développeur, ce qui limite les effets liés aux compétences disparates. Le temps de travail a été enregistré de la même manière pour l'ensemble des logiciels, ce qui permet une comparaison longitudinale. La plupart ont été conçus sur une période courte : les premières versions opérationnelles de tous les logiciels ont été produites en trois ans. Toutefois, il reste global par logiciel, les différentes tâches (analyse, codage, voire reprise des données, mise en production, formation, etc.) n'ayant pas été comptabilisées individuellement : on en est réduit à considérer que chaque tâche est proportionnellement identique quel que soit le logiciel, ce qui est probablement inexact dans certains cas.

Enfin, les variations dans les résultats obtenus peuvent être expliqués et semblent bien refléter la perception de la complexité de chaque logiciel.

4. Conclusions et perspectives

L'analyse du temps passé pour créer un logiciel s'est déroulée sur plusieurs axes. Le premier consistait à vérifier s'il existait une corrélation entre le temps de développement total et le nombre de lignes de code produites. Les valeurs de la plupart des logiciels sont très proches et indépendantes du langage utilisé. Si un logiciel a été codé plus rapidement (45 % plus vite), deux l'ont été plus lentement (écart de 75 %). Ces écarts s'expliquent par la complexité intrinsèque du logiciel : dans le premier cas, le code a largement été recopié d'un module à l'autre, la structure sous-jacente étant très proche. Dans le second cas, des travaux complémentaires ont été nécessaires pour mettre au point le code (positionnement de points sur une photo, gestion de codes 2D et de méta-données notamment). Ainsi, le calcul du nombre d'heures par ligne de code semble être un indicateur pertinent de la complexité d'un logiciel.

Les méthodes permettant d'estimer le temps global nécessaire pour coder une application basées sur le nombre de lignes de code ne sont pas remises en question par ces chiffres. Toutefois, leur réelle difficulté de mise en œuvre est liée à l'estimation *a*

priori du nombre de lignes nécessaires. Toutes nécessitent une phase d'analyse poussée, avec description soit des écrans à produire, soit des cas d'utilisation ou des *User Stories*. En l'absence de ces informations, elles sont difficilement utilisables.

La productivité, lors de la conception de logiciels, est liée à de nombreux facteurs. Plusieurs classifications ont été proposées (Liu *et al.*, 2015). L'une se base sur trois composantes principales : le logiciel lui-même à produire (réutilisabilité exigée, taille, complexité), l'équipe de développement (expérience, motivation, management, etc.) et le projet lui-même (langage de programmation, implication du client, etc.) (Sampaio *et al.*, 2010).

La difficulté, pour estimer ce temps dans les petites structures, tient au fait qu'il n'existe pas ou peu d'indicateurs mobilisables facilement pour réaliser le calcul. En nous basant sur le nombre de tables présentes dans la base de données, nous avons montré qu'une certaine relation existait entre celles-ci et le temps global de développement. Toutefois, le nombre de tables ne permet pas de répondre à tous les cas de figure. Il est possible que cette valeur soit une approximation du nombre de classes *ORM* utilisées dans l'application, et que ce soit ce nombre qui soit le plus pertinent. Un seul logiciel présentait un écart important entre ces deux chiffres : il serait intéressant de vérifier si cette assertion se vérifie avec d'autres exemples.

Nous avons également cherché à savoir quelle durée était nécessaire pour mettre au point un logiciel, c'est à dire à le rendre suffisamment adapté aux utilisateurs en terme de fonctionnalités et d'ergonomie pour qu'ils puissent l'utiliser. Pour des logiciels conçus *ex nihilo*, il faut compter environ entre dix-huit mois et deux ans après la mise en production de la première version pour que les besoins d'évolutions s'espacent et qu'ils puissent être considérés comme stabilisés. Les deux cas où cette valeur était sensiblement plus faible correspondaient à des logiciels qui ont été refaits, les versions initiales ayant servi de modèle (*i. e.* de cahier des charges).

Ce délai s'explique par la nécessité pour les utilisateurs de se projeter dans l'outil créé, et c'est en manipulant les premières versions qu'ils peuvent définir leurs besoins réels. Il est indépendant de l'apprentissage de l'ergonomie (les logiciels ciblaient des utilisateurs différents), mais correspond probablement à un processus cognitif d'appropriation des usages et d'adaptation des méthodes de travail à l'informatisation des tâches. Une fois que les tâches évoluent en raison de l'informatisation, le logiciel est amené à évoluer également pour répondre aux nouveaux processus. Cette approche est pertinente pour les développements de type Agile : en accompagnant les utilisateurs et en définissant avec eux leurs besoins au fur et à mesure de l'avancement du projet, ceux-ci sont capables d'exprimer ce qu'ils en attendent.

Un des corollaires est qu'un projet nécessite un délai d'environ deux années pour aboutir une fois la première version fournie : même si le temps de développement n'est pas aussi important (il peut y avoir des périodes « creuses » avant que les utilisateurs fassent remonter leurs demandes d'évolution), des ressources doivent rester disponibles pendant toute ce laps de temps pour répondre à la demande. Que le projet soit développé en interne ou sous-traité, il est important d'intégrer ce délai dans la charge

de travail des développeurs ou dans le contrat de sous-traitance. Ainsi, le recrutement ponctuel d'un développeur sur une période courte (trois à six mois par exemple), une pratique assez répandue dans les laboratoires de recherche, ne serait pas suffisant pour aboutir à un résultat satisfaisant. S'appuyer sur des équipes pérennes semble indispensable, qu'elles soient internes à l'organisme, mutualisées, ou définies par contrat. Dans ce dernier cas, des mécanismes d'appel d'offres pourraient s'appuyer sur la fourniture d'unités de main d'œuvre, à utiliser durant une période de deux années par exemple.

Les résultats obtenus en analysant dix logiciels de gestion de données produits sur une période de cinq années nécessiteraient d'être confrontés à d'autres pratiques, et l'échantillon agrandi à d'autres laboratoires. La tentative d'estimation de la charge de travail à partir du nombre de tables de la base de données semble prometteuse notamment par sa simplicité de mise en œuvre. Elle nécessiterait toutefois d'être corroborée par d'autres études.

5. Remerciements

L'auteur tient à remercier Patrick Lambert pour ses conseils précieux tant au niveau de l'analyse des données que de la mise en forme de cet article.

Bibliographie

Al-Sabbagh K. W., Gren L. (2018, janvier). The connections between group maturity, software development velocity, and planning effectiveness. *Journal of Software: Evolution and Process*, vol. 30, n° 1, p. e1896. Consulté sur https://doi.org/10.1002/smr.1896

Beck K., Beedle M., Bennekum A. van, Cockburn A., Cunningham W., Fowler M. *et al.* (2001). *Manifesto for agile software development.* Consulté sur http://www.agilemanifesto.org/

Boehm B., Clark B., Horowitz E., Westland C., Madachy R., Selby R. (1995, décembre). Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, vol. 1, n° 1, p. 57–94. Consulté sur https://doi.org/10.1007/BF02249046

Calero C., Piattini M., Genero M. (2001). Database Complexity Metrics. *4th International Conference on the Quality of Information and Communications Technology*, p. 7. Consulté sur http://ceur-ws.org/Vol-1284/paper9.pdf

Campbell G. A. (2018). Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, p. 57–58. New York, NY, USA, ACM. Consulté sur https://doi.org/10.1145/3194164.3194186

Chen J. (2006, octobre). An analytical theory of project investment: a comparison with real option theory. *International Journal of Managerial Finance*, vol. 2, n° 4, p. 354–363. Consulté sur https://doi.org/10.1108/17439130610705535

Clemmons R. (2006, 02). Project estimation with use case points. *CrossTalk, The Journal of Defense Software Engineering*, vol. 19.

Coelho E., Basu A. (2012, août). Effort Estimation in Agile Software Development using Story Points. *International Journal of Applied Information Systems*, vol. 3, n° 7, p. 7–10. Consulté sur http://research.ijais.org/volume3/number7/ijais12-450574.pdf

Forsberg K., Mooz H. (1998, juillet). 7.17. System Engineering for Faster, Cheaper, Better. *INCOSE International Symposium*, vol. 8, n° 1, p. 917–927. Consulté sur https://doi.org/10.1002/j.2334-5837.1998.tb00130.x

Jørgensen M., Boehm B., Rifkin S. (2009, mars). Software Development Effort Estimation: Formal Models or Expert Judgment? *IEEE Software*, vol. 26, n° 2, p. 14–19. Consulté sur https://doi.org/10.1109/MS.2009.47

Liu L., Kong X., Chen J. (2015). How project duration, upfront costs and uncertainty interact and impact on software development productivity? A simulation approach. *International Journal of Agile Systems and Management*, vol. 8, n° 1, p. 39. Consulté sur https://doi.org/10.1504/IJASM.2015.068605

Mendoza A., Carroll J., Stern L. (2010). Software appropriation over time: from adoption to stabilization and beyond. *Australasian Journal of Information Systems*, vol. 16, n° 2. Consulté sur https//doi.org/10.3127/ajis.v16i2.507

Morien R. (2005). Agile development of the database: a focal entity prototyping approach. In *Agile development conference (adc'05)*, p. 103-110. Consulté sur https://doi.org/10.1109/ADC.2005.7

Panko R. R. (2008, février). Spreadsheet Errors: What We Know. What We Think We Can Do. *arXiv:0802.3457 [cs]*. Consulté sur http://arxiv.org/abs/0802.3457 (arXiv: 0802.3457)

Papatheocharous E., Bibi S., Stamelos I., Andreou A. S. (2017, octobre). An investigation of effort distribution among development phases: A four-stage progressive software cost estimation model. *Journal of Software: Evolution and Process*, vol. 29, n° 10, p. e1881. Consulté sur https://doi.org/10.1002/smr.1881

Pavlic M., Kaluza M., Vrcek N. (2008). *DATABASE COMPLEXITY MEASURING METHOD*. Consulté sur https://search.proquest.com/openview/72e3990c621d16ac14cde30b24bc5a0a/1

Preston-Werner T. (2013). *Semantic Versioning 2.0.0*. Consulté sur https://semver.org/

Sampaio S. C. d. B., Barros E. A., Aquino Junior G. S. d., Silva M. J. C. e., Meira S. R. d. L. (2010, août). A Review of Productivity Factors and Strategies on Software Development. In *2010 Fifth International Conference on Software Engineering Advances*, p. 196–204. Nice, France, IEEE. Consulté sur https://doi.org/10.1109/ICSEA.2010.37

Sánchez-Gordón M.-L., O'Connor R. V. (2016, septembre). Understanding the gap between software process practices and actual practice in very small companies. *Software Quality Journal*, vol. 24, n° 3, p. 549–570. Consulté sur https://doi.org/10.1007/s11219-015-9282-6

Todorovsky D. (1997, septembre). On the working time budget of the university teacher. *Scientometrics*, vol. 40, n° 1, p. 13–21. Consulté sur https://doi.org/10.1007/BF02459259

Todorovsky D. (2014, dec). Follow-up study: on the working time budget of a university teacher. 45 years self-observation. *Scientometrics*, vol. 101, n° 3, p. 2063–2070. Consulté sur https://doi.org/10.1007/s11192-014-1284-9

Usman M., Mendes E., Weidt F., Britto R. (2014). Effort estimation in agile software development: a systematic literature review. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering - PROMISE '14*, p. 82–91. Turin, Italy, ACM Press. Consulté sur https://doi.org/10.1145/2639490.2639503