Effet des tests automatisés sur l'acquisition et le transfert des compétences en programmation

Anonymisé

Résumé. Les tests de code automatisés, centraux dans les pratiques industrielles, sont exploités de façon très variable comme leviers pédagogiques dans l'enseignement de l'informatique. Cette étude interroge leur rôle comme artefacts d'apprentissage, en prenant la récursivité comme cas d'étude. À travers une expérimentation auprès de 270 étudiants, nous comparons un groupe utilisant des tests automatisés fournis par l'enseignant à un groupe fonctionnant sans cette modalité. Les résultats montrent une amélioration significative des performances pour les étudiants en difficulté sur l'exercice initial, mais une difficulté de transfert vers des notions ultérieures (ordre supérieur), où le groupe contrôle progresse mieux. L'analyse suggère que l'absence d'élaboration de cas de tests par les étudiants limite leur capacité à construire des schémas abstraits.

Mots-clés : Apprentissage de la programmation, Test automatisé, Transfert de compétence.

1 Introduction

Les tests automatisés occupent aujourd'hui une place essentielle dans les pratiques professionnelles du développement logiciel. Leur usage, structuré notamment par le développement piloté par les tests (*test-driven development* ou TDD), permet d'améliorer la qualité, la robustesse et la maintenabilité du code. En introduisant les tests en amont de la conception, le TDD influe sur la manière dont les développeurs envisagent leur activité : le test est non seulement un instrument de vérification, mais aussi un guide dans la construction des solutions.

Dans l'enseignement de l'informatique, cette pratique ne fait généralement l'objet d'un enseignement systématique que dans les filières professionnelles. Dans l'enseignement général, la pratique du test est considérée comme une « bonne pratique » sans être forcément institutionnalisée. De nombreux environnements informatiques d'enseignement proposent des tests automatisés, mais ceux-ci sont souvent utilisés comme outils d'évaluation sans être un objet d'étude en eux-mêmes.

Dans cette étude, nous proposons d'interroger le statut des tests automatisés comme artefacts d'apprentissage dans le contexte de travaux pratiques de programmation. Nous cherchons à étudier comment leur usage influe sur la manière d'apprendre à programmer, et quels sont leurs effets sur l'acquisition et le transfert des compétences en algorithmique et programmation. Nous formulons l'hypothèse que les tests automatisés,

s'ils sont intégrés dans une perspective didactique explicite, peuvent contribuer à la construction conceptuelle des étudiants.

La récursivité, souvent identifiée comme un point de rupture dans l'apprentissage de la programmation, sert ici de cas d'étude pour examiner ce que l'usage des tests révèle et transforme dans les pratiques d'apprentissage. Nous cherchons à déterminer non seulement si les tests favorisent la compréhension de la récursivité, mais également dans quelle mesure ils facilitent — ou entravent — le transfert vers une notion ultérieure, en l'occurrence les fonctions d'ordre supérieur.

2 Contexte théorique

2.1 Le développement axé sur les tests dans l'industrie

Dans le domaine professionnel, le développement axé sur les tests (*test-driven development* ou TDD) repose sur un cycle itératif (Martin, 2007) dans lequel le développement d'un programme ou d'un composant logiciel commence par l'écriture d'un test qui doit initialement échouer, se poursuit par la production d'un code minimal nécessaire pour que ce test soit validé, et se conclut par une phase de réorganisation du code (*refactoring*). Le fait de devoir écrire un test avant d'implémenter la fonction qu'il évalue est une caractéristique fondamentale car cette règle oblige à formaliser la spécification du problème considéré sous forme opérationnelle. Les tests forment alors une spécification exécutable qui documente le code et garantit sa cohérence.

Cette manière de travailler influe profondément sur l'activité de programmation, car elle structure la conception, force l'anticipation, et contraint la formalisation des comportements attendus. D'une part, elle permet la détection précoce des erreurs, d'autre part elle permet de se prémunir contre d'éventuelles régressions lors de la maintenance d'une base de code sur un temps long. Par ailleurs, elle induit une distinction nette entre deux tâches complémentaires que sont l'écriture des tests et l'implémentation des fonctionnalités, au point qu'il est possible (du moins théoriquement) de confier ces tâches à des personnes différentes.

Le TDD s'inscrit dans un ensemble plus large de pratiques, plus ou moins codifiées, consistant à intégrer l'activité de test dans les méthodes de développement. S'il met spécialement en avant le test *unitaire*, qui évalue des aspects précis d'assez bas niveau, d'autres pratiques incluent des tests portant sur l'intégration des composants entre eux ou sur les comportements attendus du point de vue des besoins de l'utilisateur (Denoo et al., 2019). Ces autres formes de test s'éloignent cependant des aspects qui nous intéressent pour la présente étude.

2.2 Tests automatisés dans l'enseignement de l'informatique

Dans l'enseignement, le traitement automatisé du code est employé sous diverses formes et avec une variété d'objectifs différents (Combéfis, 2022) : validation syntaxique, validation fonctionnelle, évaluation de qualité selon divers indicateurs, dans

des buts d'évaluation (formative ou sommative), de détection du plagiat, ou pour la production de rétroactions plus ou moins élaborées et spécialisés à la tâche considérée.

Les tests sont souvent conçus comme une forme d'une aide à la correction ou comme un moyen d'obtenir rapidement un retour sur un programme, à l'intention de l'étudiant (rétroaction pendant l'activité) ou de l'enseignant (outil diagnostique). L'emploi de tests systématiques dans le but de réaliser des évaluations fiables nécessite néanmoins une analyse a priori suffisamment fouillée des tâches de programmation envisagées pour obtenir des résultats satisfaisants en comparaison d'une évaluation classique reposant sur l'expertise de l'enseignant (Bouhineau & Puitg, 2011).

Pourtant, leur potentiel didactique dépasse largement cette fonction de validation. En adoptant l'approche instrumentale de Rabardel (1995), les tests peuvent être envisagés comme des artefacts dont les usages se construisent progressivement. En effet, dans les situations instrumentées, ils sont présents avec l'intention de l'enseignant, mais l'apprenant peut en faire des outils d'analyse, d'exploration ou de modélisation, ou encore un moyen d'entretenir la motivation par la recherche d'une récompense.

2.3 Savoirs et compétences en programmation

Pour analyser l'effet des tests automatisés sur les apprentissages en programmation, il est utile de se référer aux compétences de la pensée informatique (Declercq, 2021; Selby & Woollard, 2013). Dans cette approche, on distingue cinq domaines de compétence : anticiper (ce qui est du ressort de la planification, dont l'élaboration d'algorithmes), évaluer (ce qui consiste à considérer un programme comme objet et étudier ses propriétés, entre autres sa validité par rapport à une spécification), décomposer (un problème complexe en un agencement de problèmes plus simples), généraliser (reconnaître en une situation particulière une instance d'un problème plus général) et abstraire (identifier les informations pertinentes pour une situation donnée et ignorer les autres, définir des interfaces dans une approche modulaire, etc).

La complémentarité entre *anticiper* et *évaluer* est au cœur de l'activité de programmation dans sa nature itérative, puisque l'élaboration d'un programme alterne toujours entre écriture et validation. Le test, qu'il soit manuel ou automatisé, se situe dans la catégorie *évaluer*, il semble donc naturel de questionner l'effet de son automatisation sur le développement de cette catégorie de compétences, et par conséquent sur l'apprentissage de la programmation dans un sens plus large.

D'autre part, la dualité entre *anticiper* et *évaluer* est à rapprocher de la dualité entre les rôles complémentaires du schéma TDD, entre la personne qui écrit les tests et celle qui implémente les fonctionnalités. On s'attend bien sûr à ce qu'un développeur maîtrise suffisamment les deux rôles, de même que les deux compétences considérées doivent être développées dans tout apprentissage de la programmation. On donc peut émettre l'hypothèse que la formalisation d'une séparation entre ces deux rôles favorise la mise en œuvre et le développement des deux types de compétences associées.

2.4 Transposition didactique

Selon Chevallard (1991), les savoirs académiques ou issus de pratiques professionnelles doivent être reconstruits pour devenir enseignables, ce processus impliquant une transformation de leur organisation, de leur statut et de leur finalité. C'est ce que l'on désigne sous le nom de *transposition didactique*.

La pratique du test en programmation offre un exemple intéressant de cette dynamique : elle constitue un élément majeur de l'activité professionnelle, ainsi qu'un type de tâche récurrent dans l'apprentissage de la programmation (Jolivet et al., 2024) mais est dans l'enseignement un savoir souvent implicite, suggéré comme une « bonne pratique » plutôt que véritablement institutionnalisé, là où la majorité des tâches explicites dont de l'ordre de la production de programmes. C'est notamment le cas dans les premiers apprentissages où la visée professionnelle n'est pas une préoccupation. Cette rareté de l'institutionnalisation interroge les types de techniques auxquelles les étudiants ont réellement accès et les significations qu'ils attribuent aux tests.

Un point important dans ce mécanisme de transposition concerne les objectifs et justifications de la pratique du test. Comme évoqué plus haut, les professionnels emploient le test systématique dans une variété de buts : formalisation exécutable des spécifications, documentation, validation et détection précoce d'erreurs, prévention des régressions lors de l'évolution d'un projet. Les objectifs dans un contexte d'enseignement de la programmation sont d'une nature différente, en lien avec les objectifs d'apprentissage : si la validation est un point commun, les autres aspects ne sont pas forcément mobilisés. Par exemple, les rôles de documentation et de prévention des régressions peuvent être pertinents dans une pédagogie de projet, qui implique un travail en groupe sur un temps long, mais pas dans un dispositif par exercices où le code produit est généralement isolé et à usage unique. À l'inverse, l'intention donnée aux rétroactions du test comme aides à la conceptualisation des notions est fondamentale dans une optique d'apprentissage mais ne s'applique pas dans un contexte ou l'écriture de code sert à répondre à la commande d'un client.

3 Problématique et méthode

Notre questionnement porte donc sur l'influence que peut avoir la pratique du test sur le développement des compétences en programmation, dans un contexte d'enseignement où le test n'est pas l'objet d'étude et donc n'est pas institutionnalisé. Pour cela, nous élaborons une comparaison entre les scores des étudiants, selon qu'une séquence de travaux pratiques aura été instrumentée par une batterie de tests unitaires fournis par l'enseignant ou que l'activité de test aura été laissée à la charge des étudiants.

L'idée sous-jacente est que la forme instrumentée par des tests fournis constitue une transposition de la séparation des rôles entre implémenteur et testeur, alors que la forme non instrumentée mèle les deux groupes de compétences, tout en mettant l'accent sur la production de code, qui est le rendu attendu par lequel l'apprentissage est évalué.

3.1 Contexte d'étude et population

L'expérimentation a été menée dans le cadre d'une unité d'enseignement d'*Algorithmique et programmation fonctionnelle* en second semestre de première année de licence scientifique générale, dans des parcours « Informatique » et « Mathématiques et informatique ». L'unité porte sur trois notions clés : la récursivité, les fonctions d'ordre supérieur et les arbres. Les travaux pratiques utilisent le langage de programmation OCaml. Ils utilisent la plateforme Caséine, dérivée de Moodle, et s'appuient notamment sur le module VPL (*Virtual Programming Lab*) qui permet d'évaluer automatiquement les programmes soumis par les étudiants.

L'effectif de l'étude est de 270 étudiants actifs, répartis en plusieurs groupes de travaux pratiques. Les séances concernées par l'intervention sont en milieu de semestre.

3.2 Protocole expérimental

Le protocole comprend un pré-test, inclus dans un élément de contrôle continu, permettant de mesurer les acquis en récursivité avant la séquence visée, suivi de deux séances de travaux pratiques instrumentés (les 6ème et 7ème du semestre dans l'unité concernée), puis un post-test inclus dans l'examen de fin de semestre. Ce post-test comporte deux parties : l'une (exercice 1) porte sur la définition de types de données et de fonctions récursives, ce qui est le sujet des travaux pratiques observés, l'autre (problème 2) porte sur les fonctions d'ordre supérieur, qui sont le sujet de la séquence suivante.

Deux groupes ont été constitués (chacun comportant plusieurs groupes d'étudiants, au sens de l'organisation pédagogique). Le groupe expérimental a fonctionné dans un environnement conçu pour l'étude : un formulaire en ligne avec dépôt de code, équipé d'une série de tests fournis pour chaque tâche de programmation de chaque séance. Les étudiants on peu faire évaluer leur code, avec un retour immédiat sur la fonctionnalité des fonctions demandées (nombre de tests réussis et premier test échoué, avec entrée fournie, sortie obtenue et sortie attendue), sans limitation du nombre d'essais, avec des éléments de correction proposés lorsque les solutions demeurent incorrectes après plusieurs tentatives. Le groupe contrôle a fonctionné sans ces tests, et donc sans retour immédiate sur leur production. En dehors de cette différence de dispositif, les deux groupes ont reçu les mêmes énoncés de travaux pratiques, avec la possibilité d'élaborer leur code dans un environnement classique, accompagnés d'un enseignant. Tous ont rendu leur code en fin de séquence et tous les codes rendus ont fait l'objet des mêmes évaluations.

3.3 Mesures et méthode d'analyse

Les mesures principales sont les scores au test avant séquence et aux deux éléments considérés dans l'examen final, à savoir l'exercice portant sur la récursivité (exercice 1) et le problème mobilisant l'ordre supérieur (problème 2). Ces deux derniers scores constituent les indicateurs de performance finale et seront corrélés aux scores initiaux.

L'analyse quantitative repose sur des modèles de régression linéaire multiple, conformément au cadre méthodologique proposé par Bressoux (2010). Cette approche

permet d'estimer l'effet propre de la participation à l'expérimentation tout en contrôlant les différences initiales entre étudiants. La variable qualitative « expérimentation » est introduite sous forme de variable dichotomique et les interactions entre le score initial et la participation permettent d'examiner les effets propres de l'expérimentation tout en contrôlant le niveau initial des étudiants .

Ces analyses quantitatives sont complétées par des observations qualitatives empiriques sur le comportement des étudiants en séance. Dans le cadre de cette étude, nous n'avons pas établi de grille d'observation critériée a priori sur les comportements.

4 Résultats et analyses

Comme l'indique Bressoux (2010), la variance expliquée par les modèles en sciences sociales est souvent modérée. C'est également le cas ici. Les interactions restent néanmoins informatives et mettent en lumière une dynamique d'apprentissage différenciée selon le niveau initial estimé des étudiants.

4.1 Effets sur l'apprentissage de la récursivité

Le premier modèle vise à déterminer comment la participation à l'expérimentation influence la performance sur la récursivité, une fois contrôlé le score initial (voir Figure 1). L'ajout d'un terme d'interaction révèle que les étudiants ayant un niveau initial plus faible tirent davantage profit de l'environnement instrumenté (avec une hausse observable de 1 point), alors que ceux ayant un niveau initial plus élevé semblent légèrement pénalisés (avec une baisse de 1,5 point observée en moyenne)[1]. Le modèle a néanmoins un pouvoir explicatif modéré ($p = 0{,}272$).

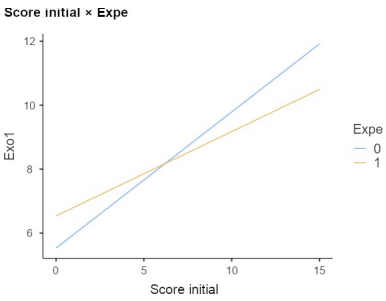

Figure 1: Interaction des variables Score initial et Participation ou non à l'expérimentation

Ces données suggèrent que l'environnement instrumenté a facilité l'apprentissage immédiat de la récursivité, notamment pour les étudiants ayant des difficultés initiales. Les tests automatisés jouent alors un rôle stabilisateur en réduisant les erreurs structurelles.

Dans le contexte d'observation, la répartition des scores des étudiants de l'expérimentation (voir Figure 2) suggère que l'environnement instrumenté n'a pas permis aux étudiants à niveau initial élevé de pousser leurs capacités jusqu'au

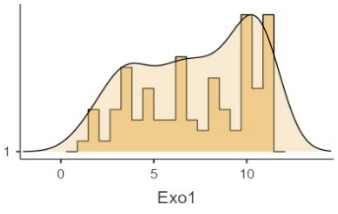

Figure 2: Répartition des scores à l'exercice 1 pour le groupe expérimental

[1]La régression linéaire donne $E = 5.531 + 0.426 \cdot I + 1.006 + D \cdot (-0.162\, x\, I)$ avec E = note à l'exercice 1, I = score initial, D = participation au dispositif (0 ou 1).

bout la nature du problème posé leur étant favorable. Le groupe contrôle présente la même tendance.

4.2 Difficultés de transfert

Le second modèle examine le transfert des compétences vers le sujet de l'ordre supérieur. Il prend en compte le score initial, la performance à l'exercice 1 et la participation au dispositif avec un $p = 0{,}050$ qui permet de conclure à un bon pouvoir explicatif du modèle. Les résultats, illustrés en Figure 3, montrent une inversion de tendance : les étudiants du groupe contrôle obtiennent de meilleures performances.[2]

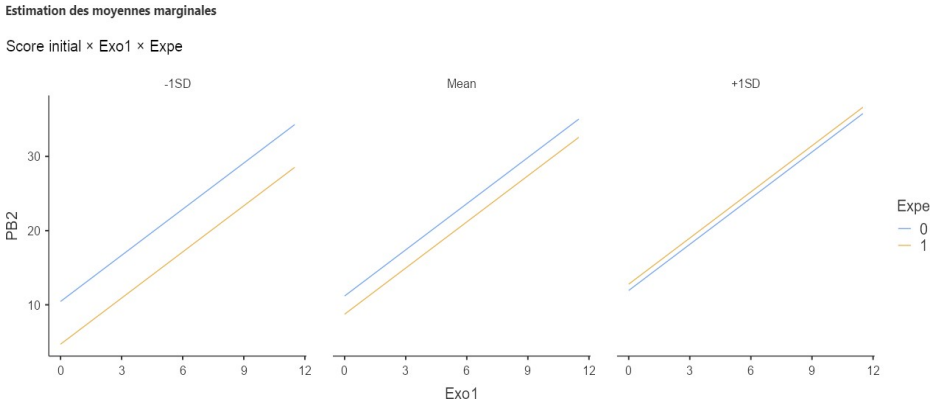

Figure 3: Interaction des variables Score initial, Score à l'exercice 1 et Participation ou non à l'expérimentation. Observations à ±1 écart type du score initial.

Novick et Holyoak (1991) observent les faibles capacités d'élaboration de schémas d'adaptation par les étudiants lorsque les problèmes sources se révèlent trop éloignés des problèmes traités. La comparaison entre l'exercice 1 et le problème 2 conforte cette observation : tiré directement d'un exercice traité en TP, l'exercice 1 représente une faible distance de transfert avec un problème source très proche du problème à traiter et des indices de résolution immédiats et se trouve naturellement plus réussi que le problème 2 qui impose une plus grande distance de transfert.

La baisse de performance entre l'exercice et le problème est néanmoins plus marquée dans le cas du groupe expérimental. Cette observation suggère que le transfert vers l'ordre supérieur se révèle plus difficile pour les étudiants de ce groupe, ce qui mène à formuler l'hypothèse que la démarche de construction des cas de tests – expérimentée par les étudiants du groupe contrôle – participe de la construction du schéma d'adaptation et conduit lesdits étudiants à développer leur capacité de transfert. L'automatisation excessive de la validation semble donc limiter la construction d'un schéma conceptuel transférable.

[2]La régression linéaire donne $P = 10{,}043 + 2{,}074 \cdot E + 0{,}195 \cdot I + D \cdot (-7{,}552 + 0{,}87 \cdot I)$ avec P = note au problème, E = note à l'exercice 1, I = score initial, D = participation au dispositif (0 ou 1).

4.3 Analyse qualitative des biais pédagogiques

Certaines caractéristiques générales des environnements informatiques pour l'apprentissage humain (EIAH) sont observables dans notre étude. Ils présentent l'avantage, par leur rôle d'automatisation, de renforcer la disponibilité des enseignants qui ne sont pas mobilisés pour la validation immédiate des productions des étudiants. Parmi les points de vigilance, les risques de biais pédagogiques et d'installation de verrous pédagogiques sont documentés (Cleuziou et al., 2021). En effet, l'observation du comportement des étudiants met en évidence des difficultés de plusieurs natures :

- Des effets de blocage ont pu être observés, avec chez certains étudiants une tendance à multiplier les tentatives sans stratégie organisée, dans une pratique d'essai-erreur improductive.
- Un comportement de « gamer » est aussi parfois observé : un étudiant qui adopte cette attitude recherchera la validation plutôt que l'apprentissage. Il se satisfera de voir réussir la série de tests fournis par l'enseignant même si sa façon d'aboutir à cette réussite ne repose pas sur une compréhension solide de la notion visée. Dans certains cas, cela a pu mener à un risque de désapprentissage, du fait de la validation possible de solutions qui sont incorrectes du point de vue conceptuel.
- Le « dialogue pédagogique » entre l'étudiant et la machine ne porte que sur des programmes entièrement écrits, contrairement à un enseignant qui peut accompagner l'étudiant pendant l'écriture de son code.
- L'analyse des programmes validés par la plateforme révèle qu'une part significative des solutions ne répondent pas aux attentes du point de vue des pratiques de programmation, même si elles valident les tests.

Ces difficultés confortent le besoin de prendre en compte d'autres dimensions de l'apprentissage au moyen de rétroactions plus fines que la seule validation d'une base de tests, afin de pallier ces risques de dérive.

5 Conclusion et perspectives

Les tests automatisés peuvent être un levier pédagogique. Ils offrent un soutien efficace aux étudiants les plus fragiles en fournissant une rétroaction immédiate. En ce sens, ils améliorent l'apprentissage technique et pour des techniques précises. Cette observation est cohérente avec les analyses de Bressoux sur les effets propres d'un dispositif où l'on contrôle les caractéristiques individuelles initiales .

À l'inverse, en déchargeant la responsabilité de l'étudiant dans la validation, ils créent un risque de fragilité dans l'apprentissage. Ils peuvent en effet conduire à une sorte de dépendance à l'outil qui risque de réduire l'activité de réflexion et d'appauvrir les techniques mobilisées. En ce sens, le manque d'élaboration personnelle des tests limite la construction de connaissances transférables.

L'enjeu est donc de didactiser l'usage des tests, afin qu'ils deviennent des instruments cognitifs permettant une compréhension profonde, et non uniquement un moyen de validation. L'enseignant – dont le travail a été allégé par les rétroactions immédiates – pourra ainsi consacrer une part de son enseignement à cet objet didactique.

Bibliographie

1. Bouhineau, D., & Puitg, F. (2011). Évaluations de solutions d'exercices d'algorithmique « à la main » versus « automatiques par jeux d'essai ». In M. Bétrancourt, C. Depover, V. Luengo, B. D. Lièvre, & G. Temperman (Éds.), *Actes de la 4e édition de la conférence EIAH* (p. 273-285). Editions de l'UMONS. https://hal.science/hal-00667472
2. Bressoux, P. (2010). *Modélisation statistique appliquée aux sciences sociales*. De Boeck Supérieur. https://doi.org/10.3917/dbu.bress.2010.01
3. Chevallard, Y. (1991). *La transposition didactique : Du savoir savant au savoir enseigné* (2ème). La Pensée Sauvage.
4. Cleuziou, G., Flouvat, F., Exbrayat, M., Robert, J., & Thion, R. (2021). Apprentissage de représentations pour l'enseignement de la programmation : Une approche centrée enseignant. *10e Conférence sur les Environnements Informatiques pour l'Apprentissage Humain*, 58-69. https://hal.science/hal-03292743
5. Combéfis, S. (2022). Automated Code Assessment for Education : Review, Classification and Perspectives on Techniques and Tools. *Software*, *1*(1), 3-30. https://doi.org/10.3390/software1010002
6. Declercq, C. (2021). Didactique de l'informatique : Une formation nécessaire. *STICEF*, *28*(3). https://doi.org/10.23709/STICEF.28.3.8
7. Denoo, O., Hage Chahine, M., Legeard, B., & Riou du Cosquer, É. (2019). *Les tests logiciels en Agile*. CFTL.
8. Jolivet, S., Dechaux, E., Gobard, A.-C., & Wang, P. (2024). Vers l'analyse de ressources d'apprentissage de la programmation à l'aide d'un référentiel de types de tâches. In K. Mens & O. Goletti (Éds.), *Actes du colloque Didapro 10 sur la Didactique de l'informatique et des STIC. Volume 1* (p. 87-98). https://hal.science/hal-04482123
9. Martin, R. C. (2007). Professionalism and Test-Driven Development. *IEEE Software*, *24*(3), 32-36. https://doi.org/10.1109/MS.2007.85
10. Novick, L. R., & Holyoak, K. J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *17*(3), 398-415. https://doi.org/10.1037/0278-7393.17.3.398
11. Rabardel, P. (1995). *Les hommes et les technologies; approche cognitive des instruments contemporains*. Armand Colin. https://hal.archives-ouvertes.fr/hal-01017462
12. Selby, C., & Woollard, J. (2013). *Computational thinking : The developing definition* [Monograph]. University of Southampton. https://eprints.soton.ac.uk/356481/