DOI : https://doi.org/10.26034/cm.iptic.2025.11754

DÉVELOPPER UNE PENSÉE ALGORITHMIQUE UNE APPROCHE BASÉE SUR DES DESIGN PATTERNS ALGORITHMIQUES ET DES ATELIERS DE KATAS

Patrick Etcheverry, Université de Pau et des Pays de l'Adour, France,
patrick.etcheverry@iutbayonne.univ-pau.fr

Thierry Nodenot, Université de Pau et des Pays de l'Adour, France,
thierry.nodenot@iutbayonne.univ-pau.fr

Pantxika Dagorret, Université de Pau et des Pays de l'Adour, France,
pantxika.dagorret@iutbayonne.univ-pau.fr

Marta Toribio-Fontenla, Université de Pau et des Pays de l'Adour, France,
marta.toribio@iutbayonne.univ-pau.fr

Jennifer Lesbats, Université de Pau et des Pays de l'Adour, France,
jennifer.lesbats@iutbayonne.univ-pau.fr

Christophe Marquesuzaà, Université de Pau et des Pays de l'Adour, France,
christophe.marquesuzaa@iutbayonne.univ-pau.fr

Cet article présente un retour d'expérience sur une approche pédagogique visant à développer la pensée algorithmique chez des étudiants de première année du Bachelor Universitaire de Technologie (B.U.T.) en informatique. L'expérimentation repose sur l'introduction de design patterns algorithmiques, utilisés pour guider la réflexion lors de la résolution de problèmes algorithmiques récurrents. Afin de favoriser leur appropriation, l'approche s'appuie sur un atelier de katas, combiné à des activités de défense collective, d'évaluation par les pairs et de défi individuel. L'expérimentation explore également l'impact de la collaboration en groupe et de la ludification sur l'engagement et la progression des étudiants. Les résultats montrent une amélioration des compétences algorithmiques, bien que des difficultés persistent dans l'adaptation des design patterns à des contextes variés. L'article propose enfin des perspectives visant à accompagner plus finement la structuration de la pensée algorithmique des étudiants, notamment à travers le développement d'outils d'analyse de code permettant un retour réflexif sur leurs productions algorithmiques.

Mots-clés : Design patterns algorithmiques, Katas, Analyse réflexive, Ludification, Collaboration.

1. Introduction

En France, le BUT qui a remplacé le Diplôme Universitaire de Technologie (DUT) en 2021, est une formation professionnalisante en trois ans, visant à préparer les étudiants à des métiers spécifiques. Parmi les 24 spécialités proposées, le BUT Informatique joue un rôle central dans la formation au métier de développeur. Ce cursus, alliant théorie et pratique, met particulièrement l'accent sur des compétences fondamentales, parmi lesquelles figurent la programmation et l'algorithmique. Selon le programme national en vigueur (MESRI, 2022), l'enseignement de l'algorithmique constitue une étape clé pour aider les étudiants à développer une pensée structurée et critique, essentielle à la résolution de problèmes complexes en informatique. Cependant, plusieurs évolutions récentes rendent cet enseignement plus difficile.

Tout d'abord, les programmes ont considérablement réduit le volume horaire alloué à l'algorithmique. Ainsi, dans le programme de 2005 (MESRI, 2005), cette discipline était enseignée tout au long de la première année durant environ 150 heures de cours et travaux dirigés. Cette répartition offrait aux étudiants le temps nécessaire pour assimiler progressivement les notions fondamentales. En revanche, dans le programme actuel (2022), ces notions sont concentrées sur environ 100 heures au premier semestre. Cette réduction des temps d'apprentissage limite les opportunités pour les étudiants de réfléchir en profondeur aux concepts algorithmiques et d'acquérir une pensée algorithmique solide.

Par ailleurs, le profil des étudiants entrant en BUT a également évolué. Depuis l'introduction, en 2011-2012, des spécialités au lycée telles que *Informatique et Sciences du Numérique* (ISN) - devenue *Numérique et Sciences Informatiques* (NSI) - pour les Bacs Généraux et *Systèmes d'Information et Numérique* (SIN) pour les Bacs Technologiques, davantage d'élèves sont sensibilisés à l'informatique dès le secondaire. Toutefois, cette sensibilisation privilégie souvent le volet programmation, avec un accent sur la manipulation concrète du code, au détriment de la réflexion algorithmique, nécessaire en amont.

Ainsi, nous observons en BUT Informatique une réelle difficulté à susciter l'intérêt des étudiants pour la réflexion algorithmique, jugée souvent peu utile car :

- ils réussissent déjà à écrire des programmes, seuls ou au lycée, sans phase de réflexion algorithmique ;
- ils peinent à évaluer la viabilité d'une solution algorithmique sans la programmer ;
- ils ne sont pas sensibilisés à l'analyse de la solution produite, pour en assurer la robustesse.

Les étudiants restent ainsi attirés par l'écriture de code, car la machine, par sa capacité à exécuter rapidement des solutions, offre une validation immédiate : soit le code fonctionne, soit il produit une erreur. Cependant, cette approche par essai-erreur occulte souvent des failles dans les solutions algorithmiques sous-jacentes, comme la gestion inadéquate des cas limites, des duplications inutiles de code ou encore des performances non optimales.

Face à cette situation, le défi est de valoriser la pensée algorithmique dans l'enseignement de l'algorithmique et de la programmation. Il s'agit de faire comprendre aux étudiants l'intérêt d'une réflexion algorithmique pour améliorer une solution et réduire le temps de sa mise au point. La démarche ne cherche pas à dissocier l'algorithmique de la programmation, car les étudiants ressentent un besoin fort de concrétiser leurs idées via du code. Il s'agit plutôt de montrer que des temps investis à la réflexion algorithmique peuvent enrichir et optimiser leurs productions.

L'article propose un retour d'expérience sur une activité réalisée avec des étudiants de première année de BUT Informatique, menée en complément des enseignements classiques (Cours Magistraux, Travaux Dirigés et Pratiques). Elle vise à ancrer des raisonnements algorithmiques appliqués à des problèmes fréquemment rencontrés en programmation, comme l'échange de deux valeurs, le parcours de tableaux ou la recherche d'un élément dans une liste.

Nous nous sommes appuyés sur le concept des katas de programmation, initialement conçus pour perfectionner des compétences de codage par la répétition d'exercices. Nous avons adapté cette méthode en y intégrant des temps de réflexion sur la dimension algorithmique au travers de design patterns algorithmiques. Pour maintenir l'engagement des étudiants lors de ces phases d'apprentissage, souvent perçues comme rébarbatives, nous avons ajouté une dimension collaborative et ludique.

L'article est organisé de la façon suivante : la section 2 présente les fondements pédagogiques de la pensée algorithmique ainsi que les travaux en lien avec l'usage des design patterns et des katas en pédagogie. La section 3 détaille le dispositif mis en place, incluant les activités pédagogiques et la méthodologie employée.

La section 4 décrit l'expérimentation menée et les indicateurs analysés. La section 5 expose les résultats obtenus et discute leur portée. Enfin, la section 6 conclut en soulignant les contributions et les perspectives d'amélioration du dispositif.

2. Cadre théorique

Nous présentons ici les fondements théoriques liés à la pensée algorithmique, à ses composantes essentielles et aux stratégies pédagogiques susceptibles d'en favoriser l'apprentissage.

2.1 Pensée informatique et pensée algorithmique

La pensée **informatique** regroupe un ensemble de compétences permettant d'aborder la résolution de problèmes de manière efficace. Parmi elles, *l'abstraction* joue un rôle fondamental en permettant de représenter les données et processus sous une forme simplifiée et manipulable (Wing, 2006 ; Barr et Stephenson, 2011). La *modélisation* constitue une étape clé de cette démarche, en structurant un problème sous une forme exploitable pour un traitement informatique (Denning, 2009 ; Selby et Woollard, 2013).

Au-delà de la mise en œuvre d'une solution, la pensée informatique intègre également des mécanismes d'*amélioration continue* et *d'évaluation* qui permettent d'ajuster, d'optimiser et d'évaluer les solutions en fonction des contraintes rencontrées (Wing, 2006 ; Brennan et Resnick, 2012).

La pensée **algorithmique**, bien que faisant partie intégrante de la pensée informatique, se distingue par son focus sur les stratégies de résolution de problèmes à travers des séquences d'instructions bien définies. Un élément central de cette approche est la *décomposition*, qui consiste à diviser un problème complexe en sous-problèmes plus faciles à résoudre (Wing, 2006 ; Grover et Pea, 2013). Cette capacité de fragmentation est complétée par le *séquençage*, qui organise ces étapes dans un ordre logique garantissant la cohérence et l'efficacité de la solution finale (Brennan & Resnick, 2012).

Une fois la structure définie, la *modélisation logique* permet de formaliser la solution d'un problème à l'aide de structures de contrôle comme des itérations ou des conditions (Selby et Woollard, 2013). L'*analyse* de la solution évalue sa robustesse face aux cas limites et sa capacité d'adaptation à de nouvelles contraintes (Grover et Pea, 2013).

Ainsi, la pensée algorithmique constitue un levier essentiel dans la formation des étudiants, leur permettant d'acquérir des compétences transférables dans divers contextes informatiques et au-delà (Drot-Delange et al., 2019). Son intégration dans l'enseignement, notamment via des approches pédagogiques adaptées, apparaît donc comme une nécessité (Brennan et Resnick, 2012).

Les contributions proposées dans cet article s'inscrivent précisément à l'intersection de la pensée informatique et de la pensée algorithmique.

2.2 Deux grandes familles de problèmes algorithmiques

L'apprentissage de l'algorithmique se structure autour de deux familles de problèmes, chacune jouant un rôle clé dans le développement des compétences des étudiants.

Problèmes classiques et récurrents. Ces problèmes, tels que l'échange de deux valeurs ou le parcours d'une collection, apparaissent fréquemment et s'appuient sur des méthodes de résolution établies. Grâce à leur récurrence, les étudiants identifient des schémas de raisonnement réutilisables et structurent progressivement leur approche algorithmique (Fluckiger, 2019). L'introduction de modèles de solutions éprouvés constitue un cadre structurant, évitant aux étudiants de repartir de zéro face à chaque nouveau problème (Resnick, 2013).

Problèmes complexes et solutions originales. Ces problèmes nécessitent une analyse approfondie et mobilisent une réflexion algorithmique avancée. Wing (2006) et Grover et Pea (2013) soulignent leur importance pour développer une pensée critique et créative, essentielle dans la résolution de problématiques ouvertes.

Ces deux catégories de problèmes ne s'opposent pas, mais se complètent. Les problèmes pouvant être résolus avec des modèles de solution réutilisables offrent un socle solide, aidant les étudiants à affronter des défis plus complexes (Brennan et Resnick, 2012).

2.3 Dissociation entre réflexion algorithmique et syntaxe

Un des défis majeurs dans l'enseignement de l'algorithmique réside dans la dissociation entre réflexion abstraite et contraintes syntaxiques. Cette séparation permet aux débutants de se focaliser sur le raisonnement algorithmique sans être freinés par les erreurs de syntaxe ou d'exécution (Fluckiger, 2019). Le lecteur trouvera dans Branthôme (2023) des travaux complémentaires centrés sur les questions de syntaxe lors du passage d'une représentation graphique d'algorithme (ou de programme) en lignes de code.

Selby et Woollard (2013) insistent sur l'importance d'une telle dissociation pour aider les étudiants à structurer leur raisonnement et renforcer une compréhension durable des concepts algorithmiques. Ils apprennent à structurer leur réflexion de manière progressive tout en allégeant leur charge cognitive (Grover et Pea, 2013 ; Labouche, 2024).

De son côté, Wing (2006) souligne qu'en appréhendant d'abord les concepts abstraits, les étudiants développent des compétences transférables à divers langages de programmation. Dans cette optique, des outils visuels comme Scratch ou Blockly offrent un environnement propice à l'apprentissage des fondements de l'algorithmique (Brennan et Resnick, 2012).

2.4 Des temps dédiés à la réflexion algorithmique

2.4.1 L'importance de l'analyse préalable

Une analyse préalable approfondie permet non seulement de mieux cerner les exigences et contraintes d'un problème, mais aussi d'anticiper les difficultés potentielles et d'optimiser la conception de la solution. Décomposer les objectifs, identifier les contraintes et anticiper les cas limites favorisent une approche méthodique et rigoureuse (Jonassen et al., 2006).

Papadakis et Kalogiannakis (2020) soulignent que cette analyse préalable joue un rôle essentiel dans la consolidation des apprentissages et le transfert des compétences vers de nouveaux contextes. Encourager les étudiants à formaliser explicitement leur réflexion avant de coder leur permet de structurer leurs idées et de mieux en comprendre les enjeux.

2.4.2 La confrontation des idées pour affiner les solutions

L'échange avec les pairs ou les enseignants stimule la pensée algorithmique et renforce l'autorégulation cognitive. Vygotsky (1978) met en avant le rôle des interactions sociales dans l'apprentissage, en particulier via la zone proximale de développement. Domínguez et al. (2022) montrent que la collaboration améliore la performance des étudiants en programmation en favorisant l'analyse critique et la diversification des stratégies.

2.4.3 La réflexion post-solution pour renforcer les apprentissages

Analyser *a posteriori* une solution permet d'en évaluer la robustesse et d'identifier des améliorations possibles. Schön (1983) parle de « réflexion sur l'action » comme un levier pour développer la pensée critique et la transférabilité des connaissances. Borge et White (2016) soulignent que le feedback post-solution renforce l'apprentissage en engageant les étudiants dans un questionnement sur leurs propres raisonnements.

2.4.4 La valeur des schémas de solutions réutilisables

Les design patterns popularisés par Gamma et al. (1995) formalisent des solutions éprouvées et réutilisables, facilitant la structuration des raisonnements. Leur intérêt réside dans leur réutilisabilité (adaptables à divers contextes), leur structuration de la pensée (aide à l'analyse des problèmes) et leur indépendance du langage (applicables à différents paradigmes de programmation).

Sweller et al. (2011) expliquent, via la théorie de la charge cognitive, que l'apprentissage est plus efficace lorsque les étudiants s'appuient sur des exemples structurés plutôt que de créer une solution *ex nihilo*. Ce processus favorise la reconnaissance de schémas et la transposition des solutions à de nouveaux contextes, réduisant ainsi la complexité cognitive de la résolution de problèmes.

Grover et Pea (2013) insistent sur la nécessité d'identifier et d'exploiter des structures récurrentes en algorithmique. Cela inclut :

- les structures de contrôle fondamentales (itérations, conditions, modularisation) ;
- les stratégies de raisonnement algorithmique, permettant de transposer des solutions apprises à de nouvelles variantes d'un problème.

Brennan et Resnick (2012) soulignent l'importance des motifs récurrents (*computational patterns*) qui facilitent une compréhension transférable. Par exemple, l'échange de deux valeurs suit une séquence standardisée d'affectations applicable à divers types de données, illustrant ainsi l'intérêt d'un enseignement fondé sur des modèles réutilisables.

2.5 Le kata comme approche d'apprentissage par la répétition

L'acquisition des compétences algorithmiques, basée sur la capacité à reconnaître et mobiliser des schémas de solution génériques répondant à des problèmes récurrents, n'est pas une activité spontanée, elle nécessite un entraînement structuré.

2.5.1 L'apprentissage par la répétition : un levier cognitif reconnu

En sciences cognitives, les travaux d'Anderson (1983) sur l'acquisition des connaissances procédurales montrent que la répétition favorise l'apprentissage.

Selon l'auteur, l'apprentissage débute par une application consciente des règles avant d'aboutir à une automatisation progressive. Ericsson (2007) montre également que la maîtrise d'une compétence repose sur un entraînement délibéré où les erreurs sont progressivement corrigées. Enfin, Logan (1988) explique que la répétition favorise un accès rapide aux solutions mémorisées, facilitant ainsi la transition entre une réflexion analytique et une application automatisée.

2.5.2 Katas de programmation : un dispositif structurant pour l'apprentissage actif

Un kata de programmation se compose d'une série d'exercices courts et ciblés, répétés sous des variations progressives. Introduits en programmation par Thomas et Hunt (1999), il reprend le principe des arts martiaux où la répétition systématique permet une assimilation progressive. Appliqués à l'algorithmique, les katas peuvent renforcer la capacité à résoudre des problèmes récurrents pour lesquels il existe des solutions génériques pré-établies.

En dépassant une approche purement empirique, les katas peuvent favoriser une appropriation durable des concepts et offrent un entraînement adapté pour apprendre à maîtriser la résolution de problèmes récurrents, tout en renforçant l'autonomie et la flexibilité des étudiants face à des situations algorithmiques variées.

2.6 Ludification appliquée à la pensée algorithmique

La ludification consiste à intégrer des mécanismes inspirés du jeu pour favoriser l'engagement des apprenants. Cependant, si ces approches peuvent renforcer l'implication, elles ne garantissent pas une motivation universelle et doivent être appliquées avec discernement. Viau (1997) identifie trois leviers fondamentaux de la motivation en contexte scolaire : la perception de valeur (« *cette activité est-elle utile pour moi ?* »), la perception de compétence (« *suis-je capable de réaliser cette activité ?* ») et la perception de contrôlabilité (« *le résultat dépend-il de moi ?* »). Le dispositif que nous présentons en section 3 tente d'agir sur ces trois axes, par des défis progressifs et un système d'évaluation adaptatif. Il met en œuvre les leviers suivants :

1. *Introduction d'éléments narratifs* (Dicheva et al., 2015) pour renforcer l'engagement des étudiants dans les exercices en plus de leur expliquer les étapes clés d'une réflexion algorithmique.
2. *Défis progressifs* (Kapp, 2012) pour que les exercices proposés maintiennent un équilibre entre challenge et sentiment de compétence. La complexification des problèmes algorithmiques ne doit pas engendrer une démotivation des étudiants.

3. *Limitation temporelle* pour stimuler la concentration et l'engagement (Landers et Landers, 2014), mais pas de stress contre-productif. En aucun cas la difficulté des exercices conjuguée à la pression temporelle ne doit constituer un obstacle insurmontable pour l'étudiant.
4. *Compétition mais aussi entraide entre groupes* pour encourager les étudiants à se dépasser (Huang et Soman, 2013). Au-delà de la compétition, la mise en avant d'objectifs collectifs sera un levier pour renforcer la dynamique de collaboration et l'apprentissage entre pairs.
5. *Récompenses et reconnaissance* pour matérialiser la progression des étudiants et maintenir la motivation (Hamari et Koivisto 2015). Il s'agira plus de tenir compte des progrès individuels que de la seule réussite/performance (Deci et Ryan, 2000).

3. Dispositif pédagogique proposé

Le dispositif propose un ensemble d'activités amenant les étudiants à se mettre en situation de répondre à un besoin formulé à l'aide d'une série d'exercices sous forme de jeux de katas de programmation. Ces activités et exercices sont conçus pour aider les étudiants à maîtriser les design patterns algorithmiques. La scénarisation des activités est basée sur les principes de la ludification, l'ensemble du dispositif étant implémenté sur la plateforme Moodle.

3.1. Des design patterns algorithmiques pour structurer la réflexion

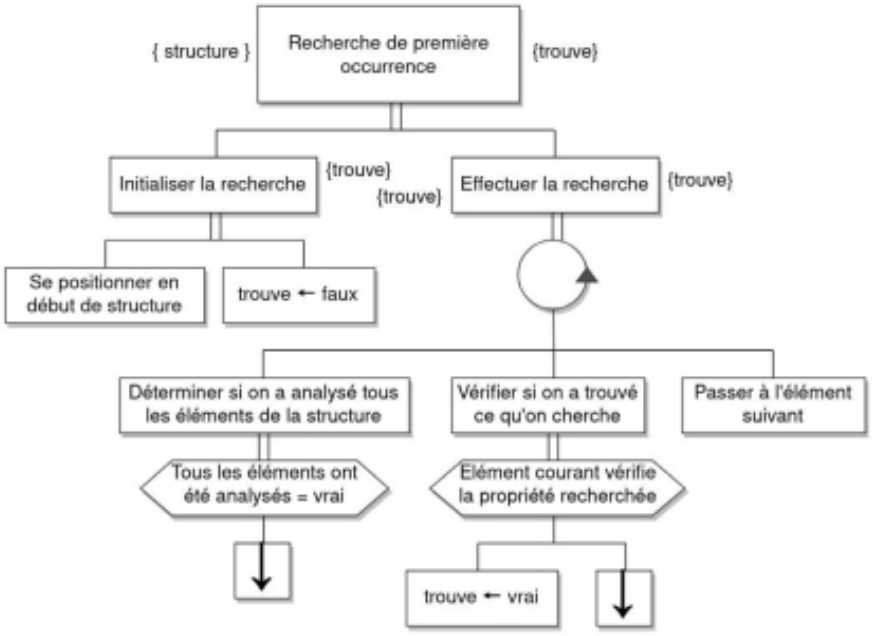
Afin d'intégrer les principes énoncés dans la section précédente dans un cadre pédagogique structuré, nous avons formalisé des design patterns algorithmiques adaptés aux problèmes classiques et récurrents en informatique. Dans la lignée des design patterns en génie logiciel (Gamma et al., 1995), ces modèles fournissent des solutions génériques applicables à des familles de problèmes tels que le parcours de collections d'éléments, la recherche d'une valeur dans un ensemble, ou encore la saisie sécurisée d'une donnée.
Parmi les design-patterns enseignés, citons notamment :

- *Échange de deux valeurs* : décrit les étapes génériques pour intervertir les valeurs de deux variables.
- *Saisie-vérification* : permet d'assurer qu'une valeur saisie respecte une contrainte (ex. : un numéro de mois compris entre 1 et 12).
- *Parcours complet avec traitement systématique* : applique une action à tous les éléments d'une collection (ex. : mettre en majuscule tous les caractères d'un mot).
- *Parcours complet avec traitement conditionné* : applique une action uniquement aux éléments vérifiant une propriété donnée (ex. : incrémenter de 1 toutes les valeurs négatives d'un tableau).
- *Recherche de première occurrence* : identifie la présence d'un élément correspondant à un critère dans une collection (ex. : détecter si une liste contient au moins un nombre négatif).

Le formalisme utilisé pour représenter ces design patterns repose sur des éléments visuels clés qui reprennent les concepts fondamentaux de l'algorithmique (*cf.* section 2.1) :

- Décomposition du problème (| |) pour diviser la tâche en hauteur en sous-problèmes.
- Séquençage des étapes (—) pour structurer en largeur la progression logique de la résolution.
- Modélisation des structures de contrôle (conditions et itérations) sous forme de symboles spécifiques.

Figure 1 : design pattern algorithmique pour détecter la présence d'un élément vérifiant une propriété particulière dans une collection d'éléments.



La Figure 1 donne l'aperçu d'un design pattern applicable pour résoudre un problème visant à détecter, dans un ensemble d'éléments, la présence d'au moins un élément vérifiant une propriété donnée : vérifier la présence d'un nombre négatif dans un tableau, vérifier la présence d'une majuscule dans une chaîne de caractères, etc.

Ce design pattern propose des étapes de résolution qui sont toujours les mêmes pour résoudre ce type de problème : se positionner au début de la structure (au début du tableau / au début de la liste / au début du fichier…), supposer que la recherche ne va pas aboutir puis répéter les actions suivantes :

- déterminer si on a analysé tous les éléments (du tableau/de la liste/du fichier…) en vérifiant si on est en fin de structure. Si c'est le cas la recherche prend fin ;
- déterminer si l'élément courant vérifie la propriété recherchée (le nombre est négatif/le caractère est une majuscule). Si c'est le cas, on infirme l'hypothèse de départ en indiquant que la recherche a abouti et on stoppe la recherche ;
- tenter de passer à l'élément suivant (à la cellule suivante dans le tableau, à l'élément suivant dans la liste, à la ligne suivante dans le fichier…).

Les étapes de résolution doivent donc être adaptées en fonction de la structure dans laquelle on effectue la recherche (une chaîne de caractères, un tableau d'entiers, un fichier…) et de la propriété recherchée (un caractère en majuscule, un nombre négatif, un mot de 5 lettres…).

La Figure 2 illustre la manière d'exploiter le design pattern de la figure 1 pour produire un algorithme (en pseudo-code) permettant de déterminer si une valeur négative se trouve :

- dans une structure de type tableau (algorithme de gauche) ;
- dans une structure de type liste (algorithme de droite).

Figure 2 : exploitation du design pattern algorithmique de la figure 1 pour rechercher la présence d'une valeur négative dans un tableau (à gauche) ou dans une liste (à droite).

```
// Initialiser la recherche
1.1 Se positionner en début de structure
    i ← 0
1.2 Initialiser la variable résultat
    trouve ← Faux

// Effectuer la recherche
2. REPETER
    2.1 Déterminer si on a analysé
        tous les éléments de la structure
        SI i ≥ nbCases
            ARRETER
        FIN SI

    2.2 Vérifier si on a trouvé ce qu'on
        cherche :
        SI tab[i] < 0
            trouve ← Vrai
            ARRETER
        FIN SI

    2.3 Passer à l'élément suivant :
        i ← i + 1
FIN REPETER

// La variable "trouve" contient le
résultat
```

```
// Initialiser la recherche
1.1 Se positionner en début de structure
    allerDebut(liste)
1.2 Initialiser la variable résultat
    trouve ← Faux

// Effectuer la recherche
2. REPETER
    2.1 Déterminer si on a analysé
        tous les éléments de la structure
        SI parcoursEnCours(liste) = faux
            ARRETER
        FIN SI

    2.2 Vérifier si on a trouvé ce qu'on
        cherche :
        SI elementCourant(liste) < 0
            trouve ← Vrai
            ARRETER
        FIN SI

    2.3 Passer à l'élément suivant :
        avancer(liste)
FIN REPETER

// La variable "trouve" contient le
résultat
```

3.2 Pratique des design patterns algorithmiques par le jeu des katas

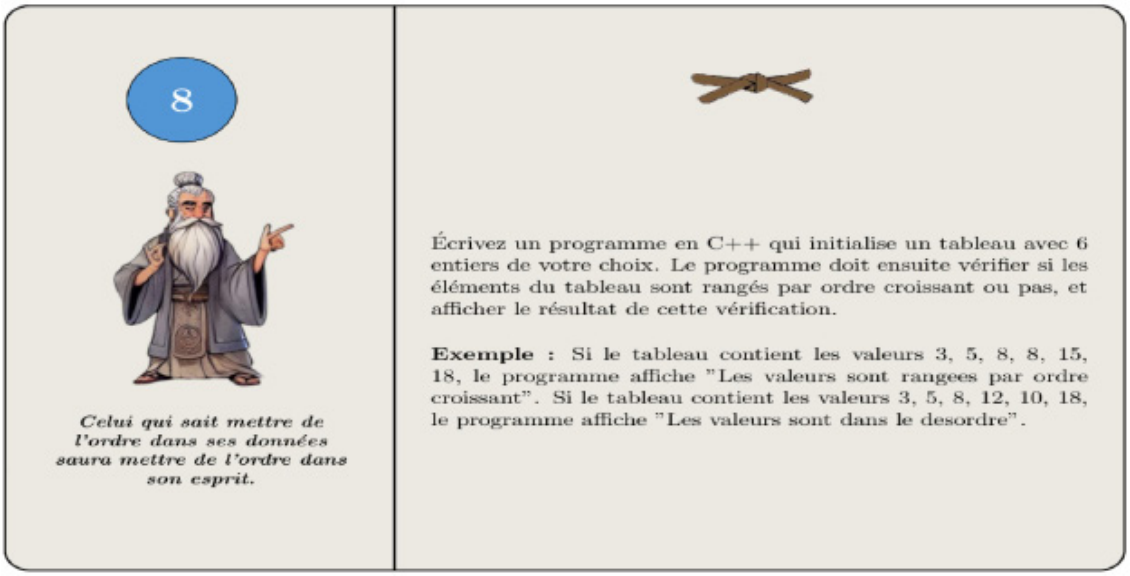
Les katas de programmation offrent un cadre structuré favorisant l'assimilation des design patterns par la répétition ciblée et la progression graduelle (*cf.* section 2.5). En s'exerçant de manière répétée sur des problèmes similaires mais sous différentes variantes, les étudiants intègrent progressivement des schémas de résolution proposés par ces design patterns.

Dans notre dispositif, un kata est une séquence d'exercices algorithmiques progressifs visant à ancrer une compétence algorithmique spécifique. Nous avons choisi d'utiliser les katas pour renforcer l'appropriation des design patterns algorithmiques, mais d'autres utilisations des katas auraient pu être faites, par exemple pour cibler la décomposition itérative ou conditionnelle qui sont d'autres compétences algorithmiques fondamentales.

L'atelier de katas reprend plusieurs mécanismes inspirés du jeu avec l'objectif d'engager les étudiants dans leur apprentissage.

L'activité est décrite par des éléments narratifs immersifs qui se concrétisent par un mentor fictif qui accompagne les étudiants à travers des proverbes inspirants ou humoristiques.

Figure 3 : un exemple d'exercice présenté par Maître Algoshi dans un kata



L'atelier de katas a par ailleurs une durée limitée pour inciter les étudiants à structurer leur réflexion et à prendre des décisions rapidement démontrant ainsi un certain niveau de maîtrise. Bien que potentiellement stressante, cette contrainte peut aussi stimuler l'engagement (*cf.* analyse des résultats en section 5).

Les exercices de katas sont aussi structurés par niveaux de difficulté croissante ; chaque exercice résolu en débloque un plus complexe, encourageant l'amélioration continue. Pour chaque atelier de katas, la progression des étudiants se matérialise par des badges attestant un niveau de maîtrise croissante des design pattern ; ces badges se matérialisent par un système de ceintures et de niveaux inspiré des arts martiaux (Tableau 1).

Tableau 1 : progression dans les niveaux de maîtrise

Niveau	Couleur de ceinture	Badge	Note max atteignable
1	Ceinture blanche-jaune		7/20
2	Ceinture jaune		8/20
3	Ceinture jaune-orange		9/20
4	Ceinture orange		11/20
5	Ceinture orange-verte		13/20
6	Ceinture verte		15/20
7	Ceinture bleue		16/20
8	Ceinture marron		18/20
9	Ceinture noire		20/20

Les exercices de katas sont réalisés par des groupes auto-constitués pour encourager les étudiants à s'entraider et à se dépasser au sein d'un groupe. Dans l'esprit de la ludification, il y a cependant un aspect compétitif entre groupes au travers des badges/ceintures.

Figure 4 : groupes d'étudiants et niveau de maîtrise correspondant

Niveau de maîtrise	Groupe	
Ceinture marron		Groupe 01 - Les Champions du Kata
		Groupe 09 - Les Elus du Grand Esprit Algorithmique
		Groupe 13 - Les Protecteurs des Secrets Algorithmique
		Groupe 18 - Les Tigres du Code Structuré
Ceinture bleue		Groupe 07 - Les Guerriers du Katana Algorithmique
		Groupe 16 - Les Sentinelles du Code Sacré

3.3 Design patterns algorithmiques : évaluation du niveau acquis

L'apprentissage des design patterns algorithmiques ne repose pas uniquement sur leur mise en pratique intensive au sein d'un kata. Comme vu en section 2.4.3, la réflexion post-solution joue un rôle clé dans la consolidation et l'ancrage durable des apprentissages. L'évaluation de la progression de chaque étudiant sur le niveau de compétences acquis pendant l'atelier de katas prend différentes formes.

1. **Défense collective** : chaque groupe justifie et explicite ses solutions en mobilisant les principes des design patterns algorithmiques étudiés. Cet exercice favorise la prise de recul sur les raisonnements algorithmiques menés durant les exercices de katas. Il incite les étudiants à verbaliser et argumenter leurs décisions (*cf*. sections 2.4.2 et 2.4.3).
2. **Évaluation par les pairs** : en parallèle, les étudiants analysent les productions d'autres groupes et évaluent/critiquent les choix algorithmiques effectués. Comme souligné en section 2.4.3, cette analyse favorise le développement de compétences métacognitives et renforce la capacité à structurer un raisonnement algorithmique.
3. **Défi individuel** basé sur trois principes :

a. Évaluation de la mobilisation d'un design pattern : chaque étudiant résout un problème similaire à ceux abordés durant l'atelier de katas, mais avec une variante. Il s'agit de vérifier s'il identifie et adapte le design pattern approprié sans assistance.

b. Développement de l'autonomie : Contrairement aux exercices de katas réalisés en groupe, cet exercice impose une analyse individuelle de l'énoncé, une identification de la structure du problème et une adaptation du design pattern.

c. Validation de la compréhension approfondie : l'assimilation des design patterns passe par la capacité à reconnaître leur champ d'application et à ajuster leur mise en œuvre en fonction des spécificités d'un problème donné.

Ces différentes évaluations donnent lieu à une note et aident l'enseignant à mesurer le niveau des acquis.

4. Expérimentation

4.1 Objectif de l'expérimentation

Nos travaux de recherche visent à explorer des stratégies pédagogiques permettant d'aider les étudiants débutant en informatique à surmonter leurs difficultés à développer une pensée algorithmique et à la mobiliser efficacement dans la production de programmes informatiques.

L'objectif est de tirer des leçons d'une expérience que nous avons menée avec des étudiants pour améliorer leur réflexion avant et après la phase de codage.

L'expérimentation menée vise à répondre à la question suivante : la mise en place d'un atelier de katas, combinée à une évaluation variée (défense collective, évaluation par les pairs, défi individuel), permet-elle d'améliorer les compétences en algorithmique des étudiants tout en renforçant leur engagement grâce au côté ludique de l'atelier ?

4.2 Profil des étudiants participant à l'expérimentation

L'expérimentation a été menée auprès de 62 étudiants en semestre 1 du BUT Informatique, représentatifs de la diversité des profils rencontrés en première année d'enseignement supérieur (promotion de 56 hommes et 6 femmes, issus de parcours variés) :

- Baccalauréats généraux scientifiques : 45 étudiants (mention TB : 4, B : 6, AB : 30, sans mention : 5).
- Baccalauréats technologiques : 12 étudiants (mention B : 5, AB : 5, sans mention : 2).
- Baccalauréats professionnels : 2 étudiants (mention B : 1, AB : 1).

Ces parcours hétérogènes se traduisent par des différences de bagage informatique et de raisonnement algorithmique. Sur les 4 dernières promotions, 80 à 90 % des étudiants ont fait de l'algorithmique via l'option NSI (Numérique et Sciences Informatiques) pour les bacs généraux et SIN (Systèmes d'Information et Numérique) pour les bacs technologiques. Mais l'hétérogénéité des enseignements au lycée entraîne une compréhension variable des distinctions entre algorithmique et programmation en début d'année.

4.3 Niveau des étudiants avant l'expérimentation

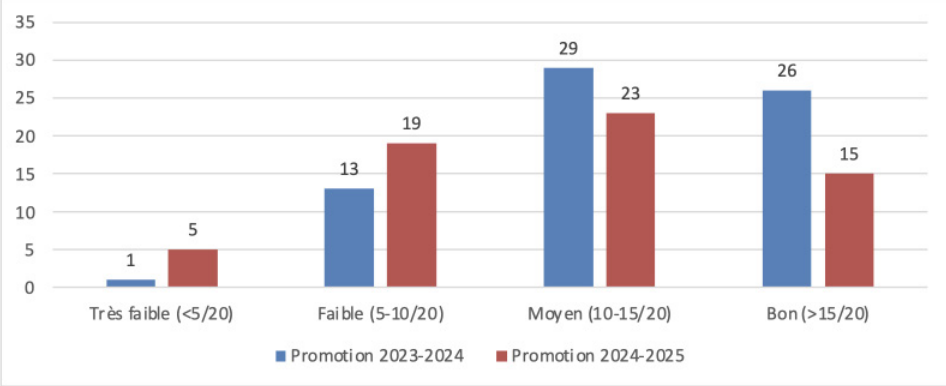
Avant l'expérimentation, un examen de diagnostic a été mené pour évaluer les compétences algorithmiques des étudiants sur des problèmes simples impliquant des structures de contrôle élémentaires (conditions, boucles) et des types de données de base (entiers, flottants, booléens, caractères). Cet examen de diagnostic est réalisé chaque année quelques semaines après le début du module d'enseignement de l'algorithmique en semestre 1.

L'analyse des résultats traduit une forte disparité des niveaux dans la promotion actuelle, avec près de 40 % d'étudiants ayant un niveau faible ou très faible. Pour situer ces résultats, nous avons comparé les performances avec celles de la promotion précédente (Tableau 2 et Figure 5).

Tableau 2 : résultats de l'examen de diagnostic

Indicateurs	Promotion 2023-2024	Promotion 2024-2025
Effectif total	69	62
Note minimale	3,5/20	2,0/20
Note maximale	19/20	17,5/20
Moyenne	12,53/20	10,56/20
Médiane	13/20	11,5/20
Écart-type	3,82	4,31

Figure 5 : examen diagnostic – Répartition des notes et nombre d'étudiants correspondants



Tendances observées :

- Baisse du niveau moyen : 10,56 contre 12,53 pour la promotion précédente, avec une médiane en recul (11,5 contre 13).
- Hausse du nombre d'étudiants en difficulté : 39% des étudiants affichent un niveau faible ou très faible (contre 20% auparavant).
- Réduction du nombre d'excellents profils : 24% des étudiants ont obtenu plus de 15/20, contre 38% pour la promotion précédente.
- Hétérogénéité plus marquée : l'écart-type est plus élevé (4,31 contre 3,82), soulignant un fossé plus important entre les étudiants en difficulté et les meilleurs éléments.

Ces écarts doivent bien sûr être interprétés avec prudence. Bien que l'examen ait été conçu pour évaluer des compétences équivalentes d'une année sur l'autre et ait été corrigé par le même enseignant, plusieurs facteurs externes peuvent expliquer ces différences, notamment la composition de la promotion (parcours scolaires, filières du baccalauréat) ou des variations dans les programmes d'enseignement secondaire. Ainsi, ces résultats offrent un contexte essentiel pour analyser l'expérimentation, sans constituer une base absolue de comparaison entre promotions.

4.4 Déroulé de l'expérimentation

L'expérimentation s'est déroulée sur trois semaines, alternant travail en autonomie et séances encadrées. Après une phase de préparation, les étudiants ont participé à un atelier de katas en groupe, suivi d'une défense collective et d'une évaluation critique. L'expérimentation s'est conclue par un défi individuel permettant d'évaluer la progression de chacun.

4.4.1 Semaine 1 : constitution des groupes et préparation de l'atelier de katas

La première semaine a débuté par une séance encadrée (1h30) présentant les objectifs, critères d'évaluation et organisation de l'expérimentation. Les étudiants ont formé librement des groupes de trois à quatre, puis se sont inscrits sur Moodle, utilisé comme support central.

Une séance en autonomie (1h30) a été consacrée à la révision des design patterns algorithmiques et des exercices associés. Chaque groupe a également défini une répartition des rôles et une stratégie de travail pour optimiser son travail lors de l'atelier de katas.

4.4.2 Semaine 2 : atelier de katas et préparation à la défense collective

Une séance encadrée de 3h a été consacrée à l'**atelier de katas**, en salle informatique. Tous les groupes d'étudiants ont été confrontés à un même kata composé de neuf exercices algorithmiques de difficulté croissante dont la résolution nécessitait de mobiliser un ou plusieurs design patterns algorithmiques. Pour chaque exercice, la démarche suivie par les étudiants est toujours la même :

- Analyse du problème : identification des caractéristiques clés.
- Choix et adaptation du design pattern algorithmique : discussion collective et réflexion sur papier.
- Codage : implémentation en C++ sous Visual Studio Code, avec compilateur et débogueur.
- Validation et dépôt : vérification collective avant soumission sur Moodle pour débloquer l'exercice suivant.

Le lendemain, une séance en autonomie (1h30) a été dédiée à la préparation de la défense collective et de l'évaluation critique :

- relecture et analyse des productions pour structurer l'argumentation ;
- répartition des prises de parole pour assurer une participation équilibrée ;
- première analyse des productions des autres groupes disponibles sur Moodle, en vue de l'évaluation critique.

4.4.3 Semaine 3 : défense collective, évaluation critique et défi individuel

Une séance encadrée de 1h30 a été consacrée à la défense collective et à l'évaluation critique, menées simultanément.

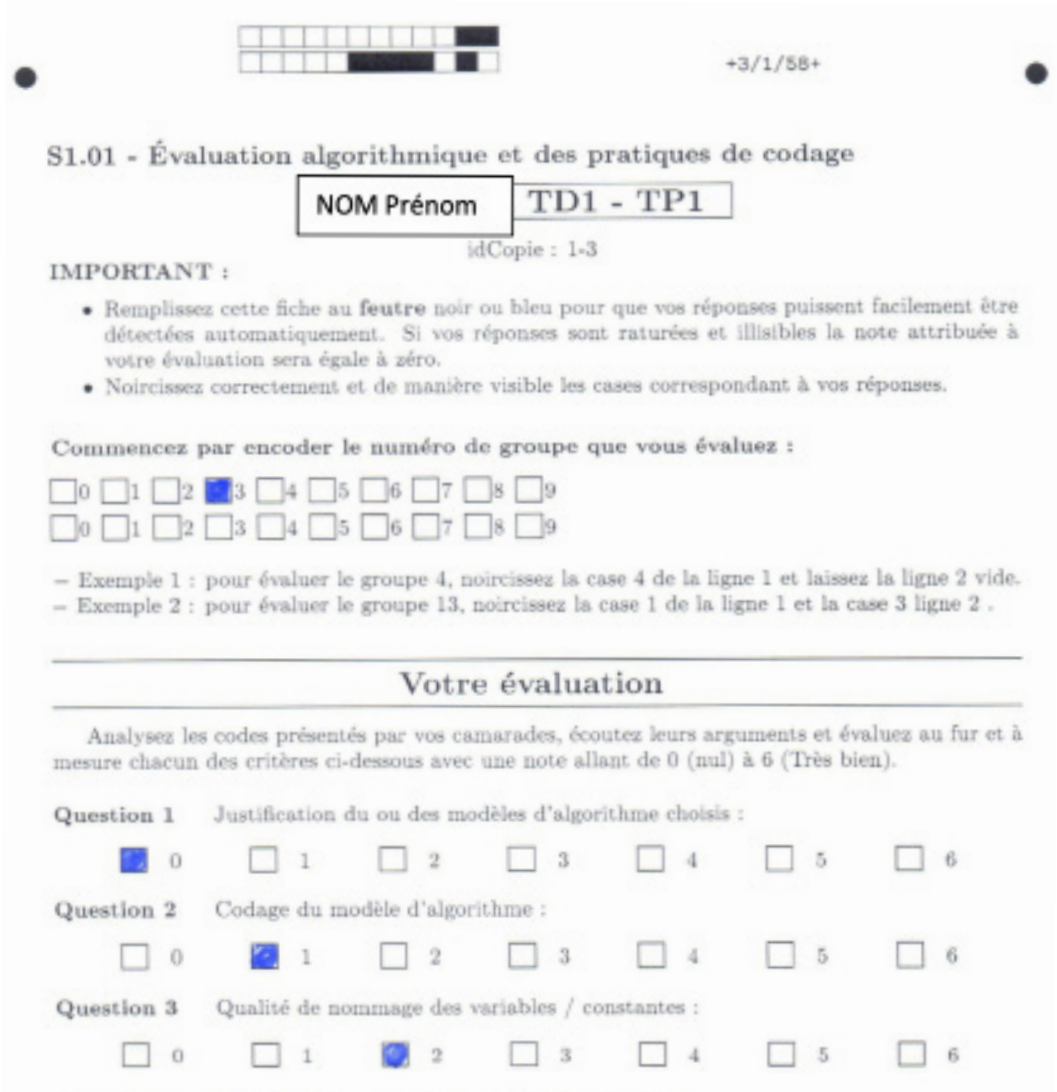
Défense collective. Chaque groupe a présenté et justifié deux solutions algorithmiques :

- L'exercice le plus avancé traité lors de l'atelier de katas.
- Un exercice imposé par l'enseignant pour tester leur capacité à argumenter une solution imparfaite.

Les groupes ont exposé leur raisonnement en projetant leur code depuis Moodle et en explicitant le design pattern algorithmique utilisé.

Évaluation par les pairs. En parallèle, les étudiants ont évalué les justifications algorithmiques de leurs pairs à l'aide de fiches *Auto Multiple Choice* (AMC) afin de faciliter le recueil et l'analyse des réponses (Figure 6).

Figure 6 : extrait d'une analyse critique réalisée par les pairs via des fiches AMC



+3/1/58+

S1.01 - Évaluation algorithmique et des pratiques de codage

NOM Prénom | TD1 - TP1

idCopie : 1-3

IMPORTANT :

- Remplissez cette fiche au **feutre** noir ou bleu pour que vos réponses puissent facilement être détectées automatiquement. Si vos réponses sont raturées et illisibles la note attribuée à votre évaluation sera égale à zéro.
- Noircissez correctement et de manière visible les cases correspondant à vos réponses.

Commencez par encoder le numéro de groupe que vous évaluez :

☐0 ☐1 ☐2 ☑3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9

– Exemple 1 : pour évaluer le groupe 4, noircissez la case 4 de la ligne 1 et laissez la ligne 2 vide.
– Exemple 2 : pour évaluer le groupe 13, noircissez la case 1 de la ligne 1 et la case 3 ligne 2 .

Votre évaluation

Analysez les codes présentés par vos camarades, écoutez leurs arguments et évaluez au fur et à mesure chacun des critères ci-dessous avec une note allant de 0 (nul) à 6 (Très bien).

Question 1 Justification du ou des modèles d'algorithme choisis :
☑ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6

Question 2 Codage du modèle d'algorithme :
☐ 0 ☑ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6

Question 3 Qualité de nommage des variables / constantes :
☐ 0 ☐ 1 ☑ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6

Cette évaluation portait sur la pertinence des choix algorithmiques et sur la qualité du code produit. L'enseignant a également noté les productions selon les mêmes critères pour comparer les évaluations des étudiants avec une référence experte. Pour limiter la variabilité des jugements, et du fait que cet exercice d'évaluation est difficile pour des étudiants débutants, les évaluations ont été réalisées sur une échelle de 0 à 6 (plutôt que 0 à 20).
L'expérimentation s'est conclue par un **défi individuel** (1h30, sur papier).

- Objectif : produire un algorithme (sans code) en mobilisant un design pattern algorithmique de manière autonome.
- Niveau d'exercice : l'exercice était choisi entre le niveau 1 et le niveau (max +1) atteint par le groupe de l'étudiant lors de l'atelier de katas.

Le défi individuel poursuivait ainsi un double objectif. D'une part, il permettait d'évaluer le niveau acquis par chaque étudiant en matière de design patterns algorithmiques, en vérifiant sa capacité à identifier et mobiliser le schéma de résolution adapté à un problème donné. D'autre part, il constituait un exercice d'auto-évaluation, où chaque étudiant devait choisir un exercice correspondant à son niveau de maîtrise. Ce choix impliquait un équilibre stratégique : opter pour un exercice plus simple augmentait les chances de réussite mais limitait la note maximale atteignable, tandis qu'un exercice plus ambitieux offrait une meilleure valorisation des compétences mais exposait à un risque d'échec plus élevé.

5. Résultats et discussion

5.1 Impact potentiel de l'atelier de katas

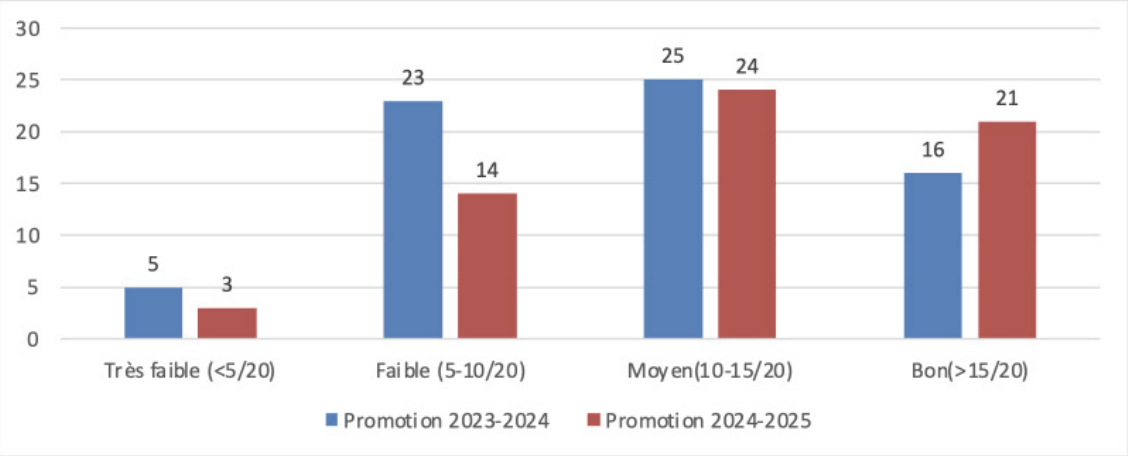
L'évaluation des compétences algorithmiques des étudiants a été réalisée à travers un examen final, couvrant l'ensemble des compétences du module d'algorithmique, y compris celles travaillées dans l'atelier de katas. Pour mesurer l'impact du dispositif, nous avons comparé les résultats de la promotion ayant suivi l'expérimentation (avec atelier de katas) à ceux de la promotion 2023-2024 (sans atelier de katas).

Les données (Tableau 3 et Figure 7) révèlent une légère amélioration des résultats de la promotion actuelle, initialement plus faible en début de semestre (+0,60 point de moyenne).

Tableau 3 : résultats de l'examen final

Indicateurs	Promotion 2023-2024	Promotion 2024-2025
Effectif total	69	62
Note minimale	2,0/20	3,5/20
Note maximale	19,75/20	20/20
Moyenne	12,01/20	12,61/20
Médiane	12,75/20	13/20
Écart-type	4,35	4,27

Figure 7 : examen final – Répartition des notes et nombre d'étudiants correspondants



La proportion d'étudiants ayant obtenu une note supérieure à 15/20 est plus élevée (+10 points de pourcentage), tandis que celle des étudiants en difficulté (<10/20) a diminué (de 41% à 28%). Ces résultats suggèrent que l'atelier de katas et ses trois activités associées (défense collective, évaluation critique, défi individuel) ont pu contribuer à une meilleure assimilation des compétences algorithmiques. Toutefois, ces effets doivent être interprétés avec prudence car d'autres facteurs, tels que la dynamique de groupe et les ajustements pédagogiques, ont pu influencer ces résultats.

5.2 Retours des étudiants

Une enquête anonyme a été menée en fin d'expérimentation pour recueillir les perceptions des étudiants sur les design-patterns algorithmiques et les activités associées. Les résultats détaillés de cette enquête sont consultables à l'adresse suivante :

https://anonym-guest.github.io/student-algo-skills-analysis/

Retours sur l'atelier de katas. L'atelier de katas a été jugé pertinent par 78% des étudiants, confirmant une forte adhésion au dispositif, bien qu'une minorité (5%) reste sceptique. Par ailleurs, 67% estiment que cet atelier a amélioré leur compréhension des design patterns algorithmiques, et autant reconnaissent l'importance de la répétition dans l'apprentissage. 69% souhaiteraient voir ce type d'atelier étendu à d'autres parties du programme, soulignant son potentiel pédagogique.

Retours sur l'intérêt des design patterns algorithmiques. Les design patterns algorithmiques sont perçus comme efficaces pour résoudre des problèmes classiques par 64% des étudiants, bien que 18% les jugent rigides ou inutiles. Un quart des répondants (25%) éprouve des difficultés à les adapter à des contextes spécifiques, mettant en évidence la nécessité d'un accompagnement supplémentaire sur cette phase d'adaptation. Par ailleurs, 76% des étudiants déclarent se sentir plus confiants pour identifier et utiliser le design pattern approprié, traduisant une meilleure compréhension des concepts et une capacité accrue à analyser les caractéristiques des problèmes rencontrés.

Retours sur la défense collective, l'évaluation par les pairs et le défi individuel. La défense collective a été jugée bénéfique par 55% des étudiants pour analyser et justifier leurs choix algorithmiques, tandis que 31% n'ont pas perçu cet apport et 14% n'ont pas donné d'avis. L'évaluation par les pairs a renforcé la compréhension de l'importance du choix du bon design pattern pour 49% des étudiants, mais 31% n'ont pas ressenti cet impact et 20% n'ont pas répondu, laissant planer un doute sur leur ressenti. Le défi individuel a été perçu comme un bon moyen de tester sa maîtrise algorithmique par 69% des étudiants. En revanche, 14% ont éprouvé des difficultés, probablement liées à la contrainte temporelle, et 17% n'ont pas exprimé d'opinion sur son intérêt.

Retours sur la dimension ludique du dispositif. L'ajout d'éléments narratifs (Maître Algoshi) et du système de ceintures a été perçu positivement par 64% des étudiants, renforçant leur engagement. La contrainte temporelle a stimulé 58% des étudiants, mais 31% la jugent problématique ou neutre, suggérant un ajustement pour éviter de pénaliser les moins rapides. Le travail en groupe a favorisé la compréhension pour 64% des étudiants, mais 20% ont trouvé l'approche collaborative peu efficace pour des questions sur la dynamique de groupe.

5.3 Contributions à la pensée algorithmique

Les résultats de l'expérimentation permettent d'identifier plusieurs leviers ayant contribué au développement de la pensée algorithmique chez les étudiants. L'analyse des performances des étudiants ainsi que leurs retours dans l'enquête montrent que certains dispositifs mis en place ont eu un impact plus marqué que d'autres.

L'intérêt des design patterns algorithmiques comme cadre structurant. Les design patterns algorithmiques ont aidé les étudiants à structurer leur réflexion, selon 78% des répondants. Leur usage a favorisé l'élaboration rapide de solutions mieux organisées, en incitant à raisonner en termes de concepts réutilisables plutôt que de traiter chaque problème individuellement. Cependant, certains étudiants ont rencontré des difficultés d'adaptation face à des problèmes nécessitant des ajustements spécifiques. Cette difficulté suggère que, bien que ces patterns offrent un cadre de réflexion pertinent, une pratique répétée est essentielle pour développer une capacité d'analyse et d'adaptation plus fine.

L'atelier de katas : un catalyseur pour la mise en pratique rapide des concepts algorithmique. L'atelier de katas a permis aux étudiants d'appliquer rapidement les design patterns algorithmiques dans le cadre des exercices en groupe. 73% des répondants ont jugé cet atelier bénéfique pour renforcer leur maîtrise des concepts. Le travail collaboratif et le rythme soutenu ont favorisé le développement d'une agilité dans le choix et l'adaptation des patterns. La verbalisation des raisonnements a également joué un rôle clé dans l'approfondissement de la pensée algorithmique. Les groupes les plus performants ont discuté et justifié collectivement leurs choix avant de coder, tandis que ceux ayant privilégié une approche plus intuitive ont parfois eu du mal à structurer leur solution.

Défense collective et évaluation critique : ancrer la réflexion par la justification et l'analyse. La défense collective a été perçue comme bénéfique par 55% des étudiants, leur permettant de prendre du recul sur leurs choix algorithmiques. De même, 49% des répondants ont trouvé que l'évaluation critique les aidait à mieux comprendre l'importance de choisir un design pattern adapté. Ces résultats montrent que ces activités ont contribué à structurer la réflexion algorithmique, en incitant les étudiants à analyser et justifier leurs décisions. Toutefois, la complexité de l'exercice souligne le besoin d'un accompagnement plus explicite sur les critères d'évaluation d'un algorithme pour renforcer ces compétences analytiques.

Le défi individuel : une mise à l'épreuve des compétences acquises. Le défi individuel a constitué un test d'auto-évaluation essentiel, permettant aux étudiants de mesurer leur maîtrise réelle des concepts algorithmiques. 69% des participants ont jugé cet exercice utile pour évaluer leurs compétences de manière autonome. Sans le soutien du groupe ni l'appui d'outils numériques, les étudiants ont dû raisonner uniquement sur la logique algorithmique. Ce défi introduisait aussi une dimension d'auto-évaluation, chaque étudiant devant choisir un niveau d'exercice adapté à son niveau. Cette approche les a incités à réfléchir sur leur progression, en trouvant un équilibre entre prise de risque et sécurisation de la réussite.

Vers un apprentissage algorithmique plus structuré et progressif. L'analyse des résultats met en évidence l'utilité des quatre piliers clés (design patterns algorithmiques, exercices de katas, défense collective incluant une analyse critique, défi individuel) choisis pour développer la pensée algorithmique des étudiants. L'articulation de ces dispositifs favorise un raisonnement algorithmique structuré et progressif. Toutefois, des améliorations restent possibles, notamment en renforçant l'accompagnement pédagogique sur l'évaluation des solutions et l'analyse critique des raisonnements algorithmiques.

6. Conclusion et perspectives

L'expérimentation menée a mis en évidence plusieurs bénéfices significatifs dans le développement de la pensée algorithmique des étudiants. L'introduction des design patterns algorithmiques, associée à des activités favorisant la réflexion avant et après la phase de codage, a structuré leur apprentissage et renforcé leur compréhension des concepts fondamentaux. Les résultats montrent une amélioration de leur capacité à raisonner et à produire des solutions algorithmiques viables.

Cependant, des lacunes persistantes ont été relevées, notamment en matière d'identification et d'adaptation des design patterns, en particulier dans la gestion des cas limites. Pour aller plus loin, nous envisageons d'accompagner davantage les étudiants dans leur réflexion algorithmique pendant l'activité de codage. Dans cette optique, nous développons Code Explorer (https://github.com/patrick-etcheverry/code-explorer), un projet basé sur Tree-sitter qui permet de représenter un code sous forme d'arbre syntaxique (*Abstract Syntax Tree*) puis de faire des requêtes pour l'analyser (https://tree-sitter.github.io/tree-sitter). L'objectif est d'offrir une représentation des codes produits par les étudiants en les décrivant avec un vocabulaire algorithmique : séquences, structures conditionnelles ou itératives, conditions d'arrêt, accumulateurs, compteurs, etc.

À terme, ce système pourrait devenir un compagnon pédagogique, capable de fournir des retours personnalisés aux étudiants, notamment pour :

- Analyser la pertinence des choix algorithmiques : repérer les oublis (initialisation, condition d'arrêt mal positionnée…) et guider l'étudiant vers de meilleures pratiques.
- Reformuler une solution problématique : permettre aux étudiants de prendre du recul en visualisant leur code sous un angle algorithmique avant de corriger leur conception.
- Proposer des améliorations : suggérer des modèles d'algorithmes plus adaptés (ex. remplacement d'une recherche séquentielle par une recherche dichotomique lorsque pertinent).

Ces travaux partent du constat que les étudiants demeurent fortement attirés par l'écriture de code, parfois au détriment de la réflexion algorithmique. L'enjeu est donc de leur fournir des outils favorisant une démarche d'analyse critique, les incitant à concevoir d'abord un raisonnement algorithmique structuré avant de passer à l'implémentation. Ainsi, Code Explorer s'inscrit dans une démarche visant à réconcilier algorithmique et programmation en ramenant régulièrement l'étudiant vers une réflexion algorithmique et une démarche d'analyse critique lui permettant de progresser sur sa réflexion algorithmique.

7. Bibliographie

Anderson, J. R. (1983). *The architecture of cognition*. Harvard University Press.

Barr, V., et Stephenson, C. (2011). Bringing computational thinking to K-12. *Communications of the ACM, 54*(6), 60–65. https://doi.org/10.1145/1929887.1929905

Borge, M., & White, B. (2016). Toward the development of socio-metacognitive expertise: An approach to developing collaborative competence. *Cognition and Instruction, 34*(4), 323–360. https://doi.org/10.1080/07370008.2016.1215722

Branthôme, M. (2023). *Apprentissage de la programmation informatique : analyses et ressources pour accompagner la transition collège-lycée* [Thèse de doctorat, Université de Bretagne occidentale - Brest]. https://theses.hal.science/tel-04397329

Brennan, K., et Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*. http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf

Denning, P. J. (2009). The profession of IT: Beyond computational thinking. *Computer, 52*(6), 28–30. https://doi.org/10.1145/1516046.1516054

Dicheva, D., Dichev, C., Agre, G., et Angelova, G. (2015). Gamification in education: A systematic mapping study. *Educational Technology & Society, 18*(3), 75-88. http://www.jstor.org/stable/jeductechsoci.18.3.75

Drot-Delange, B., Pellet, J.-P., Delmas-Rigoutsos, Y., et Bruillard, É. (2019). Pensée informatique : points de vue contrastés. *STICEF, 26*(1), 1-24. https://doi.org/10.23709/sticef.26.1.1

Ericsson, K. A. (2007). Deliberate practice and the modifiability of body and mind: Toward a science of the structure and acquisition of expert and elite performance. *International Journal of Sport Psychology, 38*(1), 4–34.

Fluckiger, C. (2019). *Une approche didactique de l'informatique scolaire*. Presses Universitaires de Rennes. https://journals.openedition.org/rfp/8523

Grover, S., et Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43. https://doi.org/10.3102/0013189X12463051

Hamari, J., et Koivisto, J. (2015). Why do people use gamification services? *International Journal of Information Management, 35*(4), 419-431. https://doi.org/10.1016/j.ijinfomgt.2015.04.006

Huang, W. H., & Soman, D. (2013). *Gamification of education*. Research Report Series: Behavioural Economics in Action. Rotman School of Management, University of Toronto. https://mybrainware.com/wp-content/uploads/2017/11/Gamification-in-Education-Huang.pdf

Kapp, K. M. (2012). *The gamification of learning and instruction: Game-based methods and strategies for training and education*. Pfeiffer.

Labouche, J.-Y. (2024). Algorithmique débranchée pour le cycle 3. *Les nouvelles technologies pour l'enseignement des mathématiques, (92)*. https://revue.sesamath.net/spip.php?article1661

Landers, R. N., et Landers, A. K. (2014). An empirical test of the theory of gamified learning: The effect of leaderboards on time-on-task and academic performance. *Simulation & Gaming, 45*(6), 769-785. https://doi.org/10.1177/1046878114563662

Logan, G. D. (1988). Toward an instance theory of automatization. *Psychological Review, 95*(4), 492–527. https://doi.org/10.1037/0033-295X.95.4.492

Ministère de l'Enseignement supérieur, de la Recherche et de l'Innovation (MESRI). (2005). *Programme national - DUT Informatique*. https://www.enseignementsup-recherche.gouv.fr/sites/default/files/content_migration/document/informatique_157826.pdf

Ministère de l'Enseignement supérieur, de la Recherche et de l'Innovation (MESRI). (2022). *Annexe 2 - Licence professionnelle BUT Informatique*. https://www.enseignementsup-recherche.gouv.fr/sites/default/files/annexe-2-licence-professionnelle-bachelor-universitaire-de-technologie-informatique-29016.pdf

Papadakis, S., et Kalogiannakis, M. (Eds.). (2020). *Mobile learning applications in early childhood education*. IGI Global. https://doi.org/10.4018/978-1-7998-1486-3

Selby, C., et Woollard, J. (2013). Computational thinking: The developing definition. *University of Southampton*. https://eprints.soton.ac.uk/356481/

Sweller, J., Ayres, P., et Kalyuga, S. (2011). *Cognitive load theory*. Springer New York. https://doi.org/10.1007/978-1-4419-8126-4

Thomas, D., & Hunt, A. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley Professional.

Viau, R. (1997). *La motivation en contexte scolaire*. De Boeck Supérieur.

Vuorikari, R., Kluzer, S. et Punie, Y. (2022). *DigComp 2.2 : The Digital Competence Framework for Citizens - With new examples of knowledge, skills and attitudes*. Office of the European Union. doi:10.2760/490274, JRC128415. C.

Vygotsky, L. S. (1978). *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press. https://doi.org/10.2307/j.ctvjf9vz4

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35. https://doi.org/10.1145/1118178.1118215