MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

Exploiter le Language Server Protocol pour créer un éditeur de code nomade et ergonomique

LOIR, Simon; ROCHET, Johan

Award date:
2024

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2023-2024

Exploiter le Language Server Protocol pour créer un éditeur de code nomade et ergonomique

LOIR Simon
ROCHET Johan

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : VANDEROSE Benoît

Co-promoteur : DEVROEY Xavier

Mémoire présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à finalité spécialisée en Software engineering

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Remerciements

Nos profonds remerciements s'adressent à notre promoteur, Pr. Benoît Vanderose, et à notre co-promoteur, Pr. Xavier Devroey, pour leurs précieux conseils et leurs relectures attentives.

Nous exprimons également notre gratitude envers les doctorants qui se sont montrés disponibles et réactifs pour répondre à nos interrogations.

Nous tenons aussi à remercier chaleureusement les étudiants qui ont accepté de prendre part à notre expérience.

Enfin, nous souhaitons témoigner notre reconnaissance à toutes les personnes qui nous ont apporté leur soutien, de manière directe ou indirecte, dans la réalisation de ce mémoire.

Résumé

Ce mémoire vise à évaluer l'utilisation du Language Server Protocol dans le cadre du développement d'un éditeur de code nomade et ergonomique. La popularité des appareils mobiles a considérablement augmenté au cours de la dernière décennie, renforçant la création de solutions mobiles. Cependant, les activités d'édition de code, traditionnellement effectuées sur un ordinateur, n'ont pas encore trouvé de véritables alternatives pour fournir un environnement de développement adapté et permettre un support multilangage sur les appareils mobiles. Les travaux antérieurs se concentrent sur la recherche de solutions d'interaction pour permettre une meilleure productivité lors de l'édition de code, en adaptant principalement l'éditeur de code à un seul langage de programmation. En intégrant l'utilisation des serveurs de langage au moyen du LSP, nous développons de nouvelles solutions d'interaction pour permettre le support multilangage dans un seul éditeur de code mobile. Dans ce mémoire, nous présentons un prototype de cet éditeur combinant des solutions d'interaction trouvées dans la littérature avec des fonctionnalités du LSP et l'évaluons en termes de productivité et d'utilisabilité. Ce travail vise à fournir une solution alternative à l'environnement de développement desktop traditionnel sur des appareils mobiles, s'adaptant ainsi aux changements technologiques et transformant la façon dont les développeurs pourraient se voir coder à l'avenir.

Abstract

This master thesis investigates how the Language Server Protocol (LSP) can be used to develop a nomad and ergonomic code editor. Mobile devices popularity has significantly increased in the past decade, strengthening the transformation of desktop solutions to mobile ones. However, code editing activities, traditionally carried out on a computer, have not yet found real alternatives to provide a suitable development environment and allow multilanguage support on mobile devices. Previous works focus on finding interaction solution to allow better code editing productivity, mainly adapting the code editor to one single programming language. By integrating the use of language servers through the LSP, we develop new design and interaction solutions to allow multilanguage support in a single mobile code editor. In this thesis, we present a prototype code editor combining interaction solutions found in the literature with LSP functionalities and evaluate it in terms of productivity and usability. This work aims to provide an alternative solution to the traditional desktop development environment on mobile devices, addressing the technological shifts and transforming the way developers may be coding in the future.

Keywords *: Language Server Protocol, code editor, touch gestures, mobile development, Programmer eXperience, programmer productivity*

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte

À l'heure de l'informatique ubiquitaire, la technologie ne fait qu'évoluer en permanence et se fond directement dans l'environnement de tout un chacun. Les smartphones, les appareils connectés, les payements électroniques et toutes les récentes avancées technologiques font maintenant partie intégrante de notre quotidien. Dans un contexte dans lequel toutes les interactions avec l'informatique deviennent de plus en plus mobiles, la plupart des solutions proposées anciennement sur ordinateur ont évolué pour devenir nomades et plus adaptées à l'utilisation actuelle de la technologie. Cependant, dans le secteur du développement informatique, les développeurs semblent avoir manqué le train des avancées technologiques récentes et continuent à développer des logiciels essentiellement avec leur ordinateur.

En 2023, selon les chiffres récupérés par Stack Overflow dans le *Stack Overflow Developer Survey* (annexe A.1) auquel 87.222 personnes ont participé, 46.91 % des développeurs utilisent *Windows* dans le cadre professionnel, 33 % utilisent *macOS* et 26,69 % des développeurs utilisent *Ubuntu*. Pour ce qui est des systèmes d'exploitation mobiles, seulement 8,23 % utilisent Android, 7,37 % iOS et 2.77 % iPadOS pour leurs activités de développement professionnelles. Ces chiffres, bien que plus élevés dans le cadre d'une utilisation personnelle (17,59 % pour Android) démontrent bien que les développeurs n'ont pas encore pris le pas d'adapter leurs habitudes de travail à la technologie mobile omniprésente.

Mais, à quoi bon vouloir changer une machine qui est déjà bien huilée en y intégrant de nouvelles technologies de développement nomades ? Les habitudes d'utilisation de la technologie évoluent avec leur temps. Aujourd'hui, la nouvelle génération est plongée dès la tendre enfance dans un univers rempli de technologie dans lequel les smartphones et tablettes prennent une place de plus en plus importante par rapport aux ordinateurs. En Belgique, comme le confirme StatBel (l'office belge de statistique), 92 % des Belges surfent sur internet avec un smartphone, 69% avec un laptop, 35 % avec une tablette et 32 % avec un PC. L'ordinateur fixe devient de moins en moins utilisé au profit de technologies mobiles comme le smartphone et la tablette. Concevoir des technologies de développement sur ces appareils apparait de plus en plus judicieux pour s'adapter aux habitudes numériques de la nouvelle génération de développeurs qui est en

train de se former. Avec des utilisations tant professionnelles qu'éducatives, des environnements de création de code sur appareils mobiles pourraient venir s'imposer comme une solution plus abordable que certains laptops.

Si on examine également la situation des pays les plus défavorisés, les données de GitHub dans leur enquête *Octoverse : The state of open source and rise of AI in 2023* (annexe A.2) sont sans équivoque : très peu de comptes utilisateurs sont créés en Afrique par rapport aux autres régions du monde, et ce, malgré une population d'environ 1.5 milliard de personnes. Une possible raison de ce manque de développeurs serait que les appareils technologiques les plus utilisées en Afrique aujourd'hui ne sont pas les ordinateurs, mais bel et bien les smartphones et tablettes, comme le confirment les statistiques de Statcounter GlobalStats (annexe A.3).

Concevoir des environnements de développements nomades et ergonomiques garantissant une productivité équivalente à celle disponible sur ordinateur devient dès lors de plus en plus nécessaire. C'est aux développeurs et chercheurs d'aujourd'hui de remonter dans le train déjà en marche pour proposer des environnements de développement adaptés à la nouvelle génération technologique.

1.2 Contributions

Ce mémoire vient avec l'objectif d'apporter les contributions suivantes dans le domaine du développement d'éditeurs de code nomades, disponibles sur tablette :

1. La création d'un éditeur de code adapté à l'utilisation tactile et prenant en charge le support multilangage.
2. La mise en place de solutions d'interactions propres à l'écriture de code dans différents langages sur tablettes de manière productive et fluide.
3. La gestion de la manipulation de code écrit dans différents langages au sein du même éditeur de code via l'utilisation de gestes.
4. Le développement d'un prototype disponible sur GitHub : `https://github.com/snail-unamur/EdgeRunner`
5. L'évaluation des solutions mises en place dans le prototype.

Chapitre 2

État de l'art

Dans ce chapitre, nous commençons par examiner l'état actuel de la recherche concernant les environnements de développement sur mobile dans le contexte de l'écriture et de la manipulation de code. Nous nous intéressons aussi à la navigation dans le code et nous présentons des éditeurs de code existants. Ensuite, nous discutons du support des langages de programmation et de leurs fonctionnalités au travers d'une présentation du *Language Server Protocol* (LSP) utilisé notamment dans Visual Studio Code [1] (VSCode), l'éditeur de code développé par Microsoft. Nous analysons par après l'impact qu'un éditeur de code peut avoir sur l'expérience développeur. Pour finir, nous évaluons la recherche propre à l'exécution de code et, plus spécifiquement, à l'exécution de code en ligne.

2.1 Développement sur appareil mobile

Concevoir des applications de développement sur mobile n'est pas une idée qui date d'aujourd'hui. Avec l'accroissement récent de la popularité des appareils mobiles, de nombreuses recherches se sont intéressées à la mise en place d'un moyen permettant de transposer le développement d'applications sur ces appareils mobiles. Dans la littérature, des solutions à plusieurs challenges ont été recherchées. Que ce soit celui de pouvoir transposer les facilités d'écriture de code sur tablette, ou celui de pouvoir mettre à profit la tactilité pour manipuler du code et/ou naviguer dans l'éditeur, ces challenges trouvent déjà une réponse, du moins partielle dans l'état de l'art.

Dans cette section, nous nous intéressons dès lors à toutes ces recherches se focalisant sur :

- l'écriture de code
- la manipulation de code
- la navigation au sein d'un IDE

dans le contexte d'application de développement sur appareils mobiles. Nous analysons en fin de section quelques outils qui ont déjà vu le jour, tentant de faire face à ces différents challenges.

1. https ://code.visualstudio.com/

2.1.1 Écriture de code

L'écriture du code est une activité quotidienne pour tout développeur. Transposer une habitude aussi ancrée dans le quotidien des développeurs sur un appareil mobile sans utiliser de clavier physique apparait ainsi comme un réel défi. De ce fait, plusieurs approches ont été envisagées afin d'améliorer l'expérience utilisateur liée au clavier virtuel classique qui n'est pas toujours adapté aux besoins de la programmation.

Utilisation de gestes

Dans un premier temps, on pourrait envisager d'utiliser les spécificités tactiles des appareils mobiles afin de pouvoir produire du code à l'aide de gestes prédéfinis. Cette piste a été explorée dans [Baćíková et al., 2015] ainsi que dans [Costagliola et al., 2018].

[Baćíková et al., 2015] se penche sur la création d'un *Domain Specific Language* (DSL) (Figure 2.1) afin de permettre la détection et la mise en correspondance des gestes de base (*General-purpose gestures*) et ceux dits *dessinés* avec un mot clé du langage ou une instruction particulière.

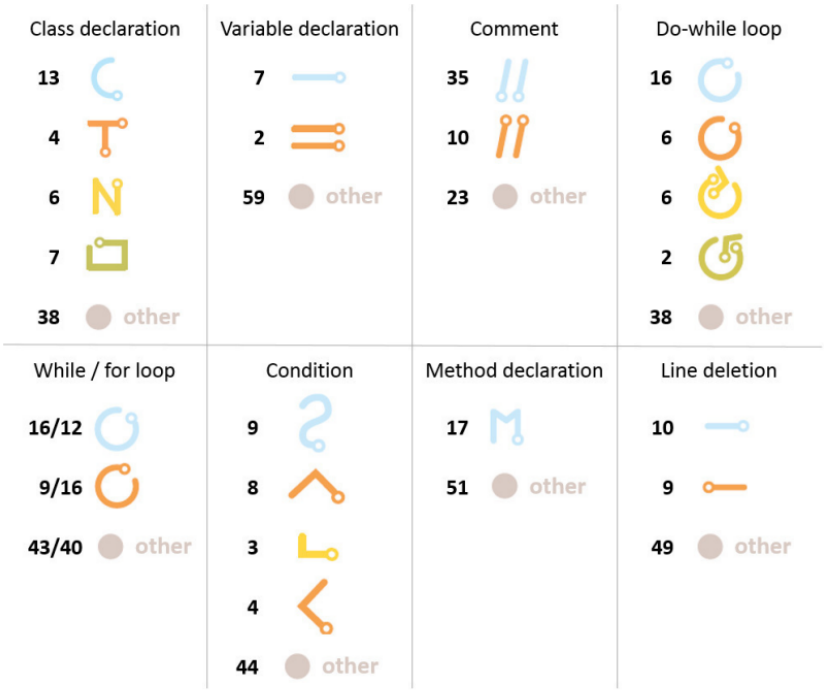


Figure 2.1 – Résultats de l'enquête de découverte de gestes (Baćíková et al., 2015)

Leur approche n'avait cependant pas pour objectif la programmation essentiellement sur un appareil mobile, mais plutôt de viser un système hybride dans lequel un appareil tactique pourrait venir assister un ordinateur classique.

[Costagliola et al., 2018] suit une approche assez similaire (Figure 2.2) en permettant aux utilisateurs d'effectuer des gestes directement sur le clavier virtuel afin de pouvoir produire du code ou effectuer une action dans le code.

Number	Gesture	Type	Produced code/action
1		C	Comment delimiter
2		C	Class stub
3		S	'Greater than' symbol
4		S	'Lower than' symbol
5		C	main() method
6		C	println() method
7		C	Square brackets
8		C	Parentheses
9		SL	Access to *Iteration* sub-layout
10		C;SL	Function code; Access to *Function* sub-layout
11		SL	Access to *Control Operators* sub-layout
12		SL	Access to *Exception* sub-layout
13		SL	Access to *Structure* sub-layout
14		SL	Access to *Variable* sub-layout

FIGURE 2.2 – Mapping gestes-actions à produire (Costagliola et al., 2018)

Clavier adapté

Une autre piste explorée est celle de la mise en place d'un clavier adapté afin de pallier les inconvénients d'un clavier virtuel classique. [Romanoa et al., 2014] et [Fennedy et al., 2022] présentent un clavier virtuel adapté à tout type de saisie de texte sur appareil mobile, tandis que [Almusaly and Metoyer, 2015] ainsi que [Costagliola et al., 2018] se concentrent sur la mise en place d'un clavier virtuel augmenté par des mots clés propres à des langages de programmation. Finalement, [Raab, 2016] présente un clavier mettant en place des mécanismes d'autocomplétion et de navigation au sein du code directement dans son design.

Dans [Romanoa et al., 2014], les auteurs ont mis en place et évalué le *Tap and Slide Keyboard (TaS)* (Figure 2.3) permettant de minimiser la place prise par le clavier sur l'écran. Après évaluation, les résultats montrent que le clavier permet non seulement d'avoir une meilleure

précision lors de la frappe, mais permet aussi, avec un usage répété du clavier, de réduire le nombre d'erreurs et le temps d'exécution de l'écriture.

FIG. 3. (a) The TaS Interface customized for a right hand

Figure 2.3 – *Tap and Slide Keyboard* (TaS) (Romanoa et al., 2014)

[Fennedy et al., 2022] propose d'adapter le clavier virtuel classique (QWERTY-AZERTY) afin de pouvoir y intégrer la gestion des raccourcis claviers propres à l'usage d'un clavier physique. Dans leur papier, les auteurs évaluent 4 designs différents pour leur clavier *SoftCuts* (Figure 2.4) qui se révèlent être tantôt adapté à un public, tantôt à un autre.

Figure 2.4 – SoftCuts (Fennedy et al., 2022)

Les 2 premiers reposent sur une connaissance préalable des raccourcis clavier par les utilisateurs permettant d'être utilisés facilement dans tous les contextes sous réserve de connaître les raccourcis. Les 2 derniers sont plus spécialisés et affichent sous forme de texte ou d'icône la commande exécutée par le raccourci avec l'inconvénient de difficilement pouvoir s'adapter au contexte d'utilisation.

[Almusaly and Metoyer, 2015] et [Costagliola et al., 2018] se sont focalisés sur la création d'un *Syntax-Directed Keyboard* (Figure 2.5), un clavier adapté à la syntaxe propre au langage de programmation *Java*.

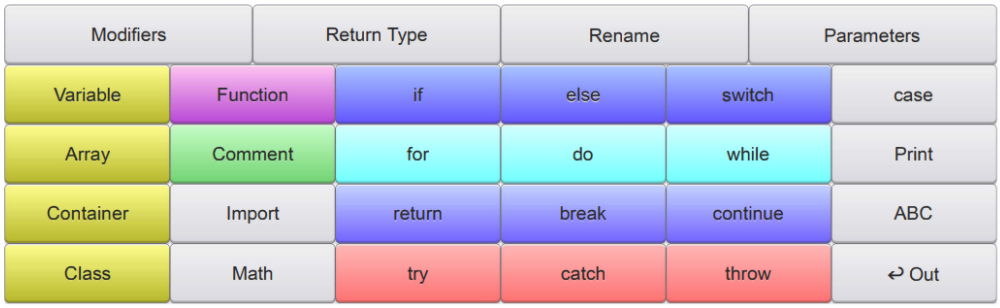
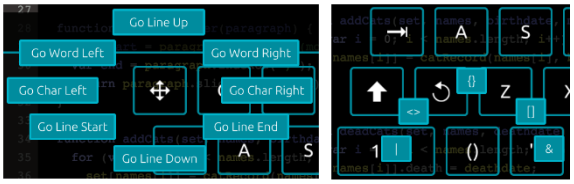


FIGURE 2.5 – Syntax-Directed Keyboard pour Java (Almusaly and Metoyer, 2015)

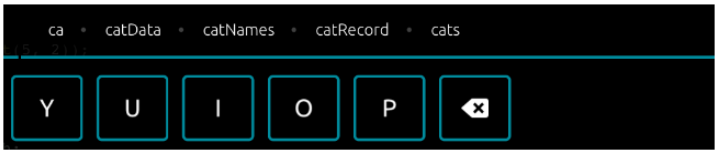
Ce clavier permet ainsi de réduire le nombre d'erreurs de syntaxe lors de l'écriture de code et d'augmenter la vitesse d'écriture de l'utilisateur.

[Raab, 2016] présente un clavier adapté à la programmation et à l'écriture de code source (Figure 2.6). Le clavier a la particularité d'être personnalisable par l'utilisateur tant en termes de taille, de position sur l'écran et de configuration des touches. De plus, le clavier permet d'accéder facilement aux caractères spéciaux nécessaires à la programmation et à des actions plus macro à l'aide d'un appui prolongé sur une touche de base du clavier. Finalement, le clavier permet aussi l'autocomplétion en fonction de l'état courant du fichier modifié.

(a) Clavier : design classique



(b) Gestes pour opérations avancées



(c) Clavier avec autocomplétion

FIGURE 2.6 – Clavier adapté avec autocomplétion et opérations spécifiques (Raab, 2016)

Speech-to-text

Une autre option envisagée pour faciliter l'écriture de code est de chercher à profiter des avancées en termes de reconnaissance vocale. Ainsi, [Feldman et al., 2015] et [Garcia et al., 2022] ont étudié l'utilisation d'une approche centrée sur la programmation vocale. Il ressort de [Garcia et al., 2022] que l'utilisation de la voix rend la programmation plus rapide dans le cas de problèmes simples à résoudre tout en réduisant la négativité du programmeur. Toutefois, le sentiment d'auto-efficacité [2] en est affecté en raison d'un manque de contrôle sur le code généré. L'approche devient également peu adaptée lorsque la codebase commence à s'élargir fortement et que les activités de programmation deviennent difficiles, demandant plus de concentration.

2.1.2 Manipulation de code

La nature tactile de l'interface d'un appareil mobile offre un panel de possibilités afin d'effectuer des activités de manipulation de code remplaçant les habituels menus déroulants des IDEs traditionnels et les combinaisons de touches du clavier physique. Cette particularité tactile a inspiré les auteurs de [Raab et al., 2013] pour développer un éditeur de code permettant de manipuler et d'éditer du code source sur des écrans tactiles. Après avoir répertorié une série d'opérations de base, ils ont mis en place une expérience pour détecter les gestes les plus adaptés à chaque opération de refactoring détaillés sur la figure 2.7.

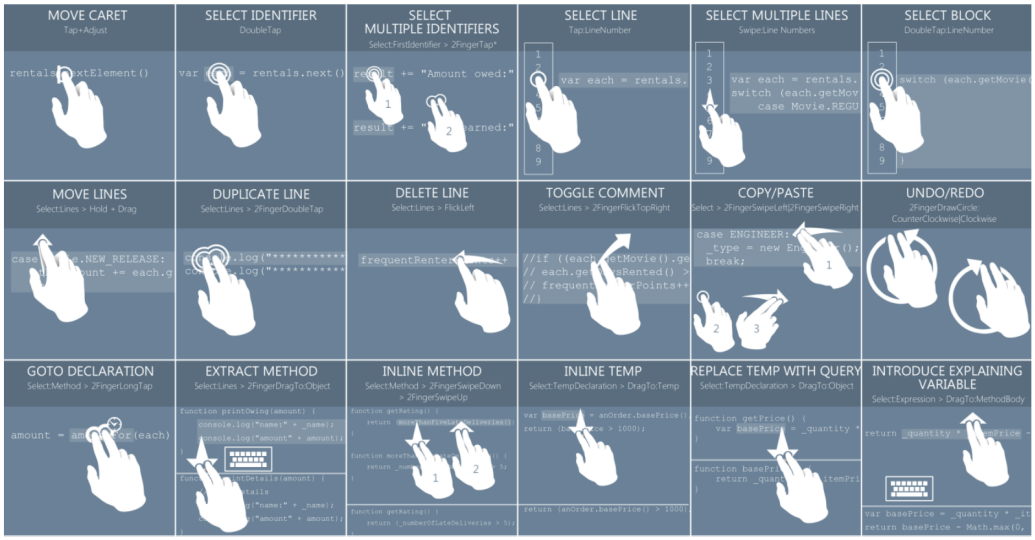


Figure 2.7 – RefactorPad : gestes de manipulation de code (Raab et al., 2013)

2. « croyances des individus quant à leurs capacités à réaliser des performances particulières », (Rondier, 2004)

Manipuler du code requiert aussi de pouvoir le sélectionner préalablement. [Raab, 2016] introduit ainsi le mécanisme de *syntax-aware code selection* (Figure 2.8) permettant la sélection de code sur base des limites structurelles du code, améliorant ainsi la précision de la sélection. Ce mécanisme permet sur base d'un simple touché de sélectionner la structure la plus générale dans laquelle se trouve l'élément ciblé en inspectant les nœuds de l'AST. Cette *syntax-aware code selection* combinée avec un mécanisme d'adaptation de la sélection permet d'obtenir une sélection efficace et précise.

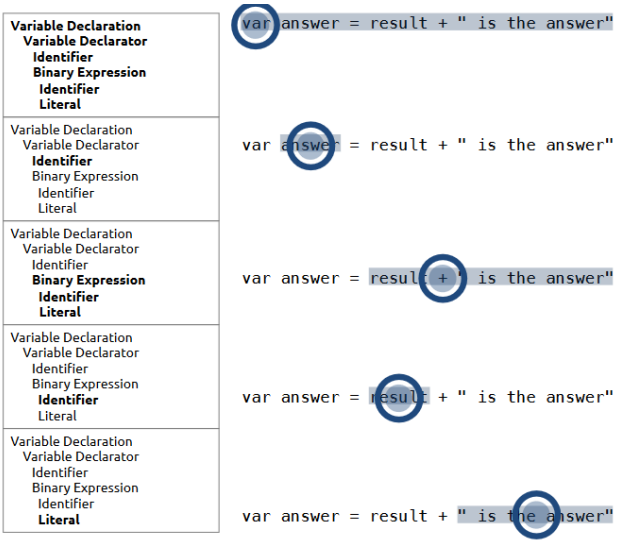
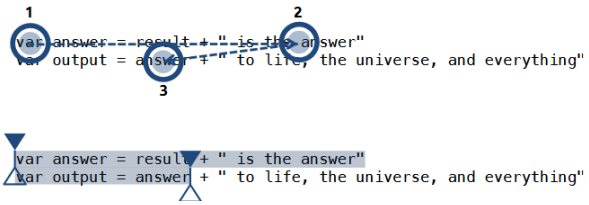
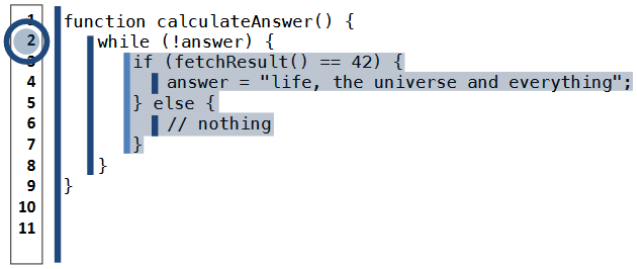


FIGURE 2.8 – Mécanisme de sélection de code basé sur l'AST (Raab, 2016)

D'autres techniques sont également développées comme celle des *selection spans* (Figure 2.9a) permettant la sélection à un ou 2 doigts et celle des *Selection Panning* et *Selection Rails* (Figure 2.9b) permettant de sélectionner le code par bloc.



(a) Selecting Spans



(b) Selecting Rails

FIGURE 2.9 – Mécanismes de sélection de code basés sur l'interaction (Raab, 2016)

Dans [Biegel et al., 2014], les auteurs ont imaginé un plugin pour Eclipse afin d'y intégrer des gestes permettant de manipuler le code et de modifier l'état de l'interface. D'une part, ils ont évalué la pertinence de certaines alternatives aux menus déroulants traditionnels avec des menus tactiles de nature circulaire, horizontale, etc. D'autre part, ils ont défini une liste de gestes permettant le refactoring de code sur appareil tactile présentée sur la figure 2.10.

Table 1: Interaction concept for triggering refactoring tools with common touch gestures.

Gesture	Context	Refactoring	Inverse Refactoring	Context	Gesture
	statements	Extract Method	Inline Method	method call/declar.	
	expression	Extract Local Variable	Inline Local Variable	local variable use	
	methods, fields, nested classes	Extract Class	*undo*	—	
	—	*undo*	Introduce Parameter Object	parameters	
	local variable declaration	Convert Local Var. to Field	*undo*	—	
	anonymous class declaration	Conv. Anon. Class to Nested	*undo*	—	
	class declaration	Move Type to New File	*undo*	—	
	identifier	Rename	Rename	identifier	
	modifier	Decrease Visibility	Increase Visibility	modifier	
	method parameter	Reorder Method Parameter	Reorder Method Parameter	method parameter	
	method declaration	Reorder Method	Reorder Method	method declaration	
	field declaration	Reorder Field	Reorder Field	field declaration	

Figure 2.10 – U can touch this : Inventaire des gestes de refactoring (Biegel et al., 2014)

2.1.3 Navigation dans le code

Une autre problématique importante lorsqu'on examine la possibilité de pouvoir coder sur un appareil mobile, est le fait de pouvoir naviguer de manière efficace au sein du système de fichiers, des fichiers ouverts et des portions de code en cours d'édition. La question n'a cependant pas spécifiquement été largement étudiée pour l'instant, mais d'autres techniques de navigation évaluées sur un ordinateur classique pourraient facilement être transposées à une interface mobile.

C'est le cas du *Patchworks Code Editor* (Figure 2.11) développé dans [Henley and Fleming, 2014]. Cet éditeur introduit avec lui un nouveau paradigme de navigation différent du traditionnel *file-base editor* (Eclipse, VSCode, etc.) et inspiré du *canvas-based editor* (Code Bubbles). Il se base sur deux éléments principaux : la grille et le ruban.

La grille est composée de six *patches* représentant chacun un onglet d'un IDE classique tandis que le ruban permet de naviguer vers d'autres *patches* de manière horizontale tout en en gardant six à l'écran. Ce système permet ainsi de réduire le temps de navigation global au sein de l'éditeur de code et pourrait facilement venir s'intégrer dans un éditeur de code mobile, notamment grâce aux gestes qui pourraient permettre de manipuler facilement le ruban.

Un autre problème rencontré lors de la navigation sur des éditeurs mobiles est la taille de l'écran. Il devient ainsi parfois difficile de pouvoir parcourir facilement le code afin de pouvoir y trouver une fonction, une classe, une variable, etc. Pour résoudre ce problème, [Mbogo et al.,

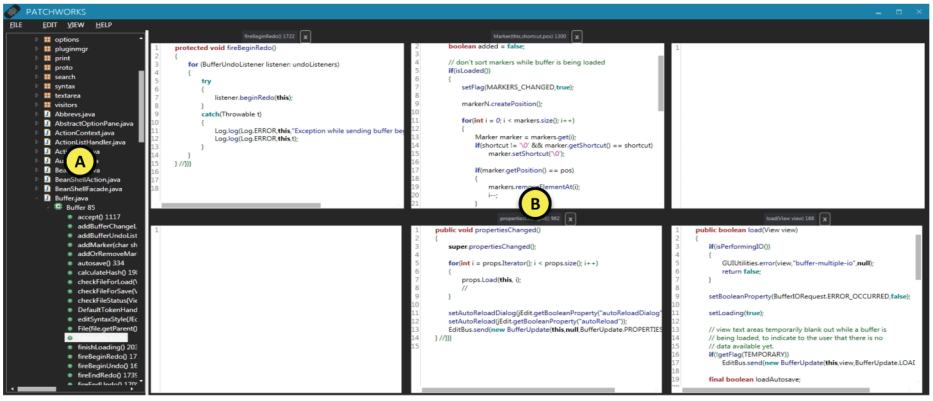


Figure 3. The Patchworks editor, including (A) a package explorer and (B) a 3×2 patch grid. Four of the patches contain code fragments and two are empty.

(a) Grille de patches

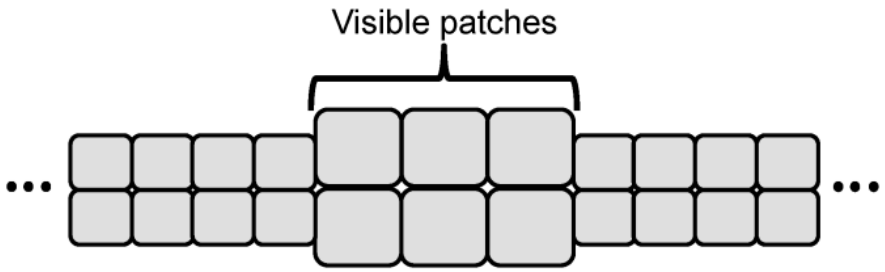


Figure 5. The conceptual ribbon of patches.

(b) Ruban

FIGURE 2.11 – La grille de patches et le ruban (Henley and Fleming, 2014)

2016] propose de mettre en place des techniques de *Scaffolding* statiques permettant, au moyen de vues dépliables, de naviguer au sein d'un fichier sans avoir à en traverser tout le code dans le contexte de programme écrit en *Java* (Figure 2.12).

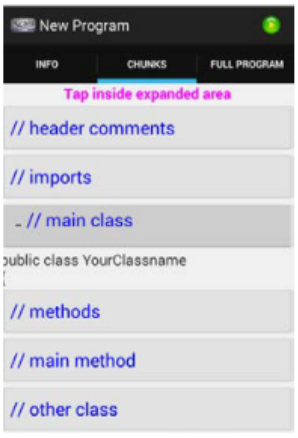


Figure 1. Main interface showing program overview with only the main class parts activated

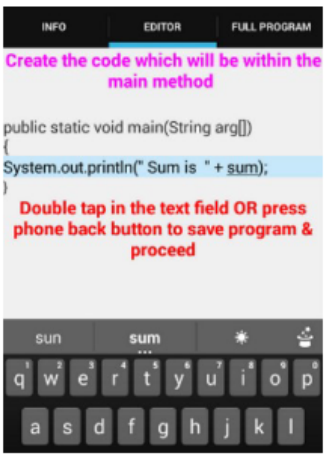


Figure 2. Editor interface showing construction of only the main method

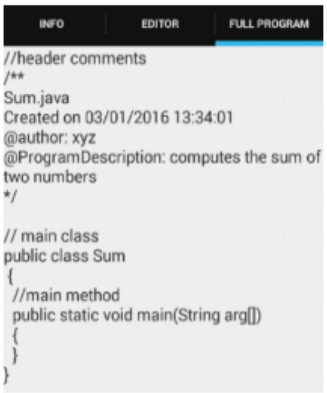


Figure 3. Full program as was last saved

FIGURE 2.12 – Techniques de Scaffolding avec vues dépliables (Mbogo et al., 2016)

Une réponse à ce problème peut aussi être trouvée dans la recherche sur le développement d'outils permettant de rendre les IDEs plus accessibles pour les personnes malvoyantes. [Potluri et al., 2018] répertorie les challenges d'accessibilité au sein des IDEs afin de développer le plugin *CodeTalk* pour Visual Studio permettant de rajouter des fenêtres telles que le résumé de fichier (Figure 2.13), la liste des fonctions, la liste des erreurs, etc. à la vue principale de l'IDE. Ces solutions pourraient ainsi être également mises en place sur un IDE mobile pour faciliter la navigation, surtout dans le cas d'un appareil ayant une taille d'écran réduite.

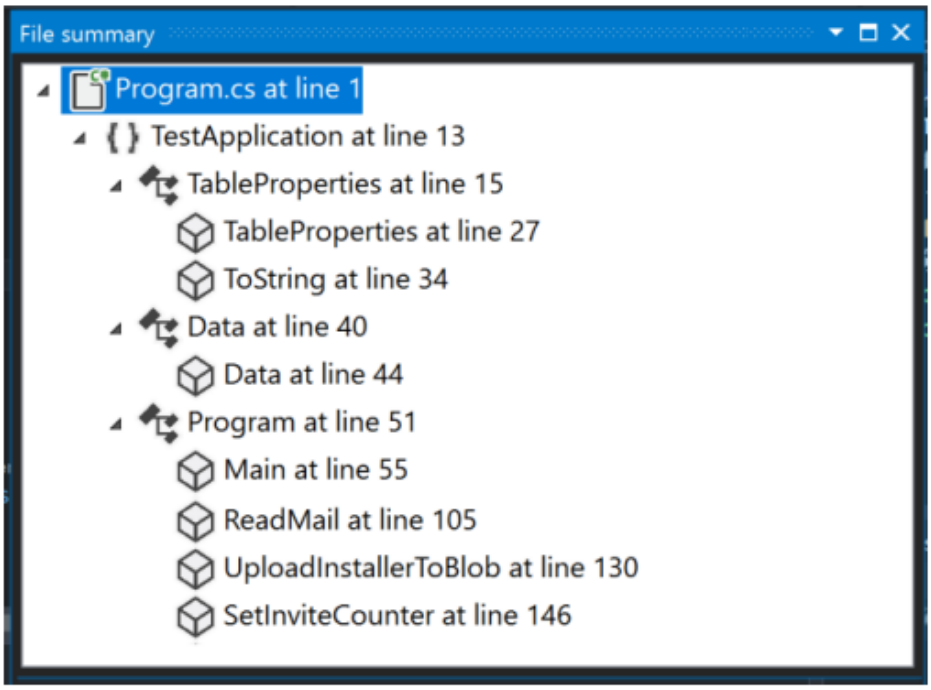


Figure 2.13 – Fenêtre de résumé d'un fichier (Potluri et al., 2018)

2.1.4 Outils concrets

Outre les différentes recherches examinant des points particuliers propres au développement sur des appareils mobiles, de réelles applications ont déjà été créées et utilisées par des programmeurs sur mobile. [Sukumar and Metoyer, 2017, 2019] répertorient ainsi un bon nombre d'applications destinées à la programmation sur appareil mobile tactile, certaines déjà présentées ci-avant. Les cas de TouchDevelop et de Touching Factor sont particulièrement intéressants dans le cadre de cette recherche.

TouchDevelop

TouchDevelop, développé par *Microsoft Research* entre 2011 et 2012, est une application de programmation anciennement disponible sur Windows phone. Dans [Tillmann et al., 2011,

2012], les auteurs décrivent la solution mise en place et détaillent leurs choix concernant l'interaction.

TouchDevelop présente une interface graphique essentiellement tactile permettant la création d'applications mobiles directement installables sur Windows Phone. Pour ce faire, les chercheurs de *Microsoft Research* ont mis en place un langage de programmation typé propre à l'application qui constitue une sorte de mélange entre un langage impératif, fonctionnel et orienté objet. Le principal objectif de ce langage est de faciliter l'interaction avec les senseurs du smartphone (GPS, caméra, gyroscope, etc.) afin de créer des applications à des fins de personnalisation de son smartphone.

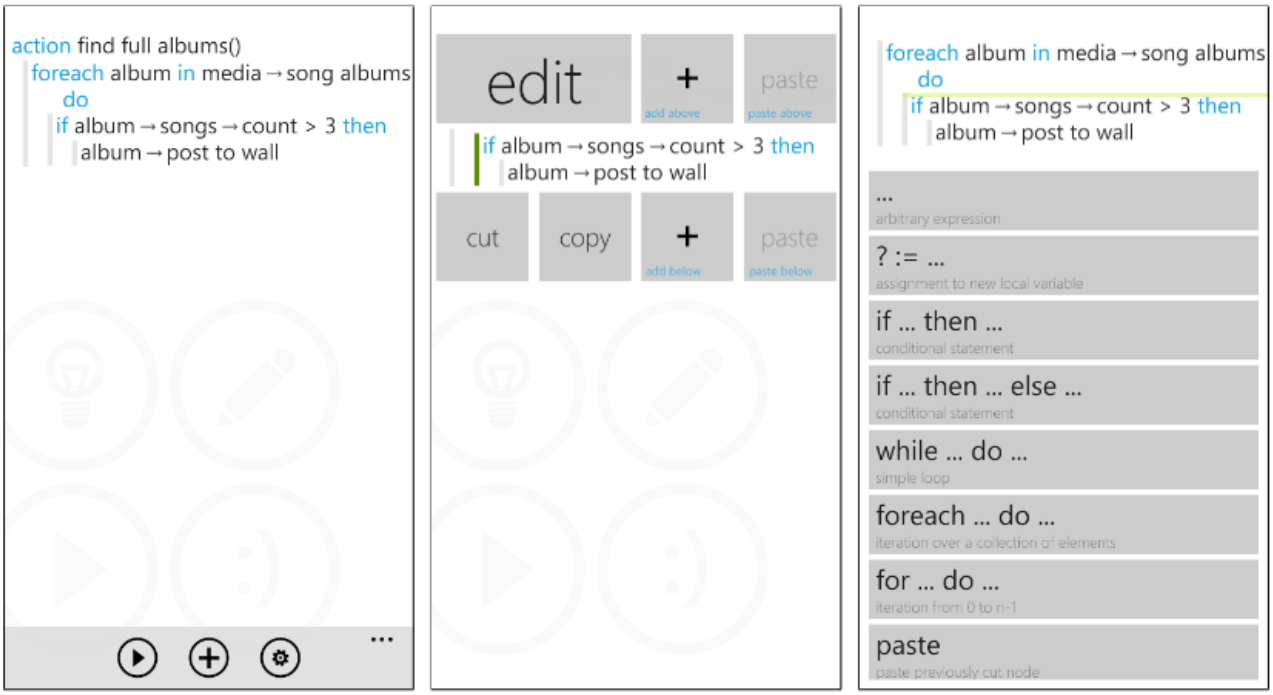


FIGURE 2.14 – TouchDevelop : éditeur de code (Tillmann et al., 2011)

Au niveau de la gestion de l'écriture de code, TouchDevelop met en place l'autocomplétion par défaut dans un éditeur de code semi-structuré : structuré au niveau des instructions et non structuré au niveau des expressions (Figure 2.14). Ainsi, il devient impossible pour l'utilisateur de commettre une erreur syntaxique dans une instruction, mais bel et bien dans des expressions. Le clavier virtuel n'est ainsi utilisé que pour le renommage de variable ou pour l'écriture de chaine de caractère.

Deux grands principes règlent les choix de design de l'application :

— « All programming and relevant content authoring must be done on a mobile device », (Tillmann et al., 2011)

— « The resulting program should run on a mobile device, leveraging the computing capabilities, sensors and cloud connection », (Tillmann et al., 2011)

Ces deux principes imposent ainsi que :

— le smartphone fonctionne de manière indépendante sans que l'utilisateur ait recours à un PC auxiliaire
— Tout programme créé pourra être exécuté sur le smartphone où il a été rédigé (langage de programmation Turing complet)

Depuis, le projet a été abandonné par Microsoft et n'a plus été mis à jour depuis le 26 octobre 2012. En 2019, le projet a définitivement été enterré et retiré du téléchargement.

Touching Factor

[Hesenius et al., 2012], à l'instar de TouchDevelop, présente un environnement de développement sur tablette pour concevoir des programmes à l'aide l'un langage de programmation spécifique (Figure 2.16).

Les chercheurs ont fait le choix de mettre en place un langage concaténatif ne nécessitant pas, comme les langages prédominants (Java, C, etc.), d'un IDE surpuissant. Ainsi, le code de la figure 2.15 écrit en *Factor* permet d'exprimer de manière brève et lisible l'algorithme de *Quicksort*.

Program 2. Quicksort in Factor[a]

```
: qsort ( seq -- seq )
    dup empty? [
      unclip [ [ < ] curry partition [ qsort ] bi@ ] keep
      prefix append
    ] unless ;
```

[a] http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Factor

Figure 2.15 – Quicksort écrit en Factor (Hesenius et al., 2012)

Afin de pouvoir remédier au problème de la taille de l'écran restreint et de l'écriture du code en *Factor*, les auteurs présentent une interface graphique basée sur 4 éléments principaux :

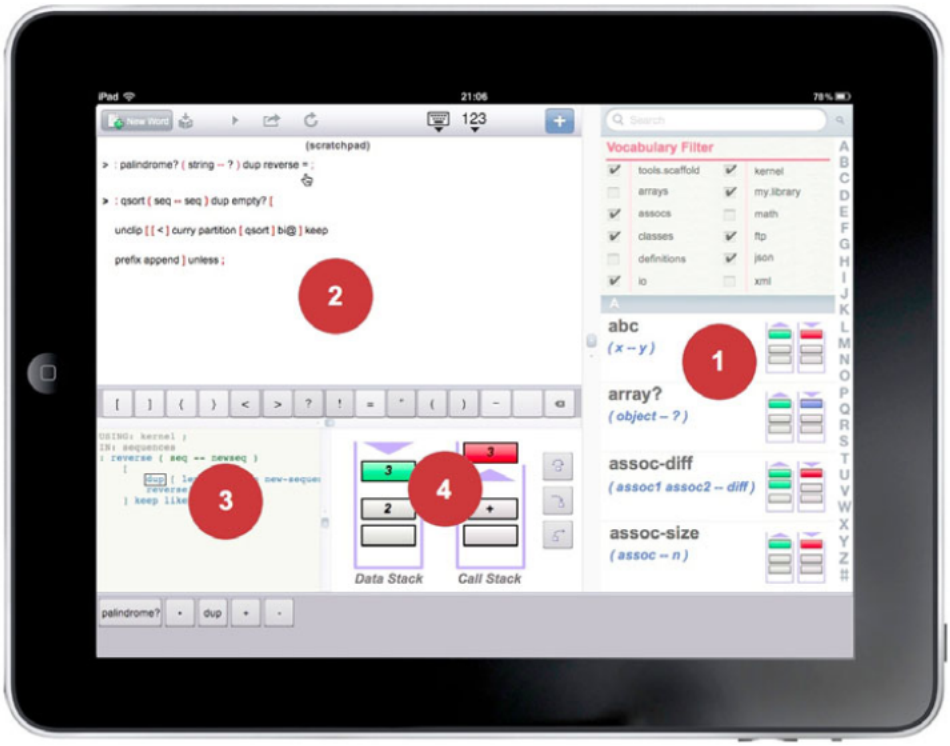


FIGURE 2.16 – Touching Factor : interface graphique (Hesenius et al., 2012)

1. **Le dictionnaire** de mots permettant aux utilisateurs d'organiser les mots à écrire au sein d'une liste triée alphabétiquement qui soit filtrable. Une opération de *drag-and-drop* permet ainsi de déplacer un mot du dictionnaire vers le scratchPad ;
2. **Le scratchPad** ou interface de création de logiciel. Il correspond à l'espace d'édition de code classique à tout IDE muni de fonctionnalités propres au langage *Factor*
3. **Le traveller** qui supporte la navigation au sein des mots composés d'autres mots (fonctionnalité spécifique au langage *Factor*).
4. **Le stack viewer** qui affiche l'état de la stack après exécution du code.

2.2 Language Server Protocol (LSP)

Le *Language Server Protocol* (LSP) (Microsoft, 2024), rendu disponible en 2015, est un protocole mis en place pour permettre la communication entre un éditeur de code et un serveur de langage. Comme le détaille [Bork and Langer, 2023], le LSP a été initié et porté par Microsoft afin de permettre à tout éditeur de code de pouvoir interagir facilement avec n'importe quel langage sans en connaître les spécificités. Avant le LSP, pour chaque éditeur de code E et pour chaque langage de programmation L, un nombre $E \times L$ d'intégrations de langage étaient nécessaires. Avec le LSP, mettre en place le support de langages dans un éditeur (le client de langage) repose uniquement sur la mise en place de la communication avec des serveurs de langage au moyen du protocole et de l'interprétation des résultats à matérialiser dans l'interface utilisateur. Du point de vue des serveurs de langage, ils peuvent dès lors se concentrer uniquement sur la mise en place d'un support pour le langage respectant le protocole sans se soucier de l'éditeur sur lequel le support sera utilisé.

Depuis qu'il a été rendu public en 2015, le nombre de serveurs de langage créés n'a fait qu'augmenter (Figure 2.17), permettant à des éditeurs, notamment le désormais très populaire VSCode [3], de pouvoir supporter de plus en plus de langages au sein du même éditeur.

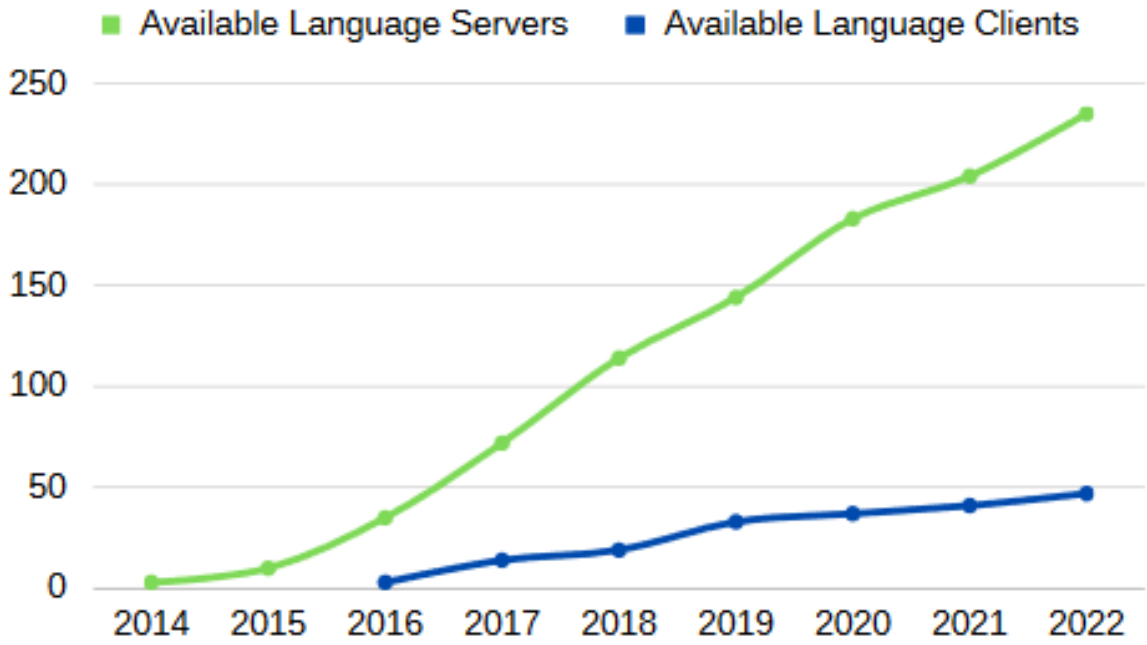


Figure 2.17 – Évolution du nombre de serveurs de langage et de clients depuis 2014 (Bork and Langer, 2023)

3. https ://code.visualstudio.com/

2.2.1 Description du protocole

Comme décrit dans la documentation du protocole (Microsoft, 2024) et dans [Bork and Langer, 2023], le protocole est basé sur le *JSON Remote Procedure Call* (JSON-RPC) permettant d'échanger des requêtes, réponses et notifications. Un serveur de langage, exécuté dans un processus séparé, reçoit des notifications et des requêtes du client et envoie des notifications et des réponses aux requêtes au client. L'exemple de la figure 2.18 illustre les interactions entre un client et un serveur de langage.

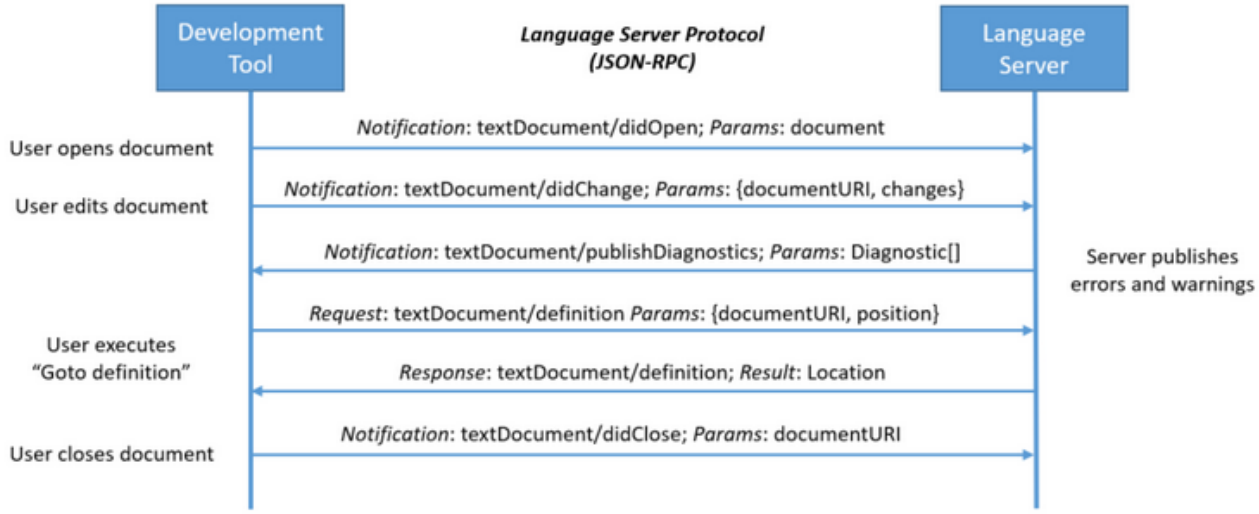


FIGURE 2.18 – Exemple d'échange via le LSP entre un serveur de langage et un client (Microsoft, 2024)

Dans cet exemple, l'utilisateur commence par ouvrir un fichier et notifie au serveur de langage qu'un document a été ouvert. Ensuite, l'utilisateur modifie le fichier et l'éditeur notifie ainsi le serveur de langage qui met à jour sa représentation du fichier et renvoie via une notification au client la présence d'erreur ou de warning dans le code. L'utilisateur veut ensuite exécuter une action de type "Go to definition" sur un symbole. Pour ce faire, le client envoie une requête au serveur afin d'obtenir la localisation de la définition du symbole. Finalement, le client notifie au serveur que le fichier a été fermé.

Cette interaction permet ainsi au serveur de langage d'avoir une vue à jour de tous les fichiers modifiés par l'utilisateur afin d'avoir un support le plus adéquat possible. À un niveau plus élevé que celui d'un simple fichier, il sera important de tenir le serveur de langage au courant de toutes modifications pour qu'il puisse permettre des actions nécessitant la recherche simultanée dans plusieurs fichiers.

2.2.2 Exemple de requête d'autocomplétion

[Bork and Langer, 2023] présente aussi un exemple de requête de recherche d'autocomplétion à un serveur de langage (Figure 2.19).

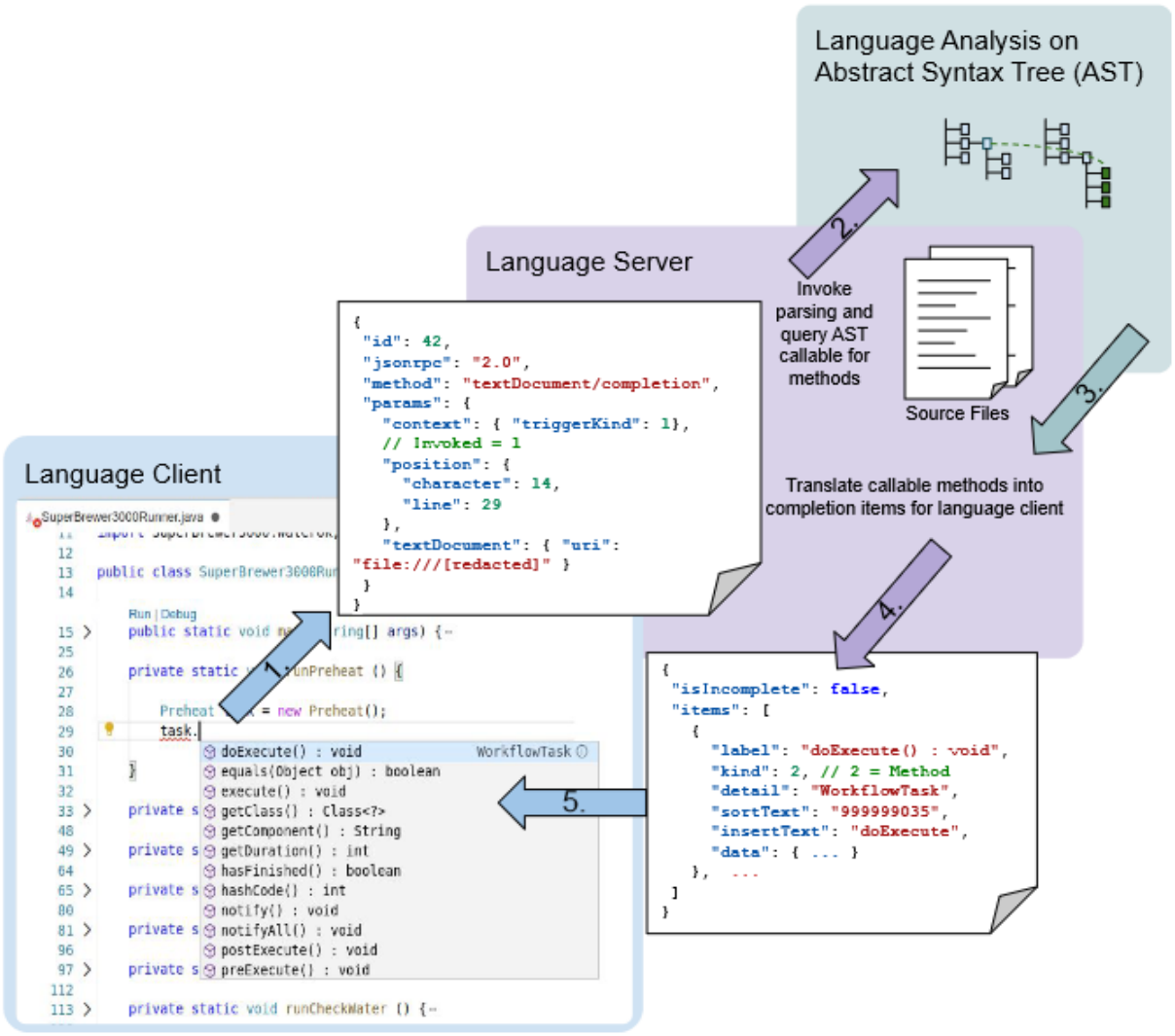


Figure 2.19 – Exemple de requête d'autocomplétion à un serveur de langage (Bork and Langer, 2023)

Ici, l'éditeur de code envoie une requête de type *textDocument/completion* au serveur de langage à une position précise d'un fichier. Une fois que le serveur reçoit la requête, il va pouvoir invoquer des mécanismes de *parsing* et faire des requêtes à l'AST afin de récupérer les méthodes qui peuvent être appelées sur l'objet *task*. Une fois les méthodes récupérées, il va transmettre sa réponse au client en formatant le résultat en tant que *completion items* à travers le LSP pour rendre la réponse interprétable par le client. Le client va ainsi pouvoir proposer des différentes méthodes à l'utilisateur dans son interface.

2.2.3 Les *Capabilities*

Le mécanisme de *Capabilities* permet également de gérer la différence qui peut exister entre les langages : une action disponible pour un langage ne le sera pas spécialement dans un autre. [Bünder, 2019] explique que les serveurs de langages sont configurés pour ignorer les *Capabilities* non supportées par le client. Lors de la requête d'initialisation du serveur de langage, le client peut lui fournir ses *Capabilities* (fonctionnalités du LSP implémentées), ce à quoi le serveur de langage répond avec le sous-ensemble des *Capabilities* du client que lui supporte. Ainsi, bien qu'il puisse supporter certaines d'entre elles, le serveur de langage ignorera toutes les requêtes ne respectant pas les *Capabilities* négociées.

2.2.4 Limitations

Finalement, [Bünder, 2019] met en exergue trois grandes limitations inhérentes au protocole lui-même :

1. Le LSP part du principe que le client et le serveur ont un accès partagé au système de fichier
2. Un serveur de langage ne communique qu'avec un seul environnement de développement à la fois
3. Le support actuel se concentre sur l'édition de fichiers et pas sur l'exécution, la compilation ou le débogage de ceux-ci.

2.3 Impact des éditeurs de code sur la productivité du programmeur

Chercher à concevoir un éditeur de code sur des appareils mobiles renvoie aussi à la préoccupation initiale de la nécessité d'utiliser ou non un IDE plus intelligent qu'un simple éditeur de texte basique. Autrement dit, en quoi l'utilisation d'un éditeur de code permet d'améliorer la *Programmer Experience* (PX) et la productivité des utilisateurs ?

Dans leur *Systematic Litterature Review* ([Morales et al., 2019]), les auteurs définissent le concept de *Programmer Experience* (PX) comme suit : « The PX is the result of the intrinsic motivations and perceptions of programmers regarding the use of development artifacts » (Morales et al., 2019). Ils estiment que le concept de PX serait lié à la motivation intrinsèque et extrinsèque du développeur à réaliser une activité de production de code ainsi que son sentiment de contrôle et de performance. Cette PX est ainsi influencée par l'éditeur de code utilisé par le programmeur : un éditeur mal organisé, trop fantaisiste ou non adapté à l'objectif du développeur affectera de ce fait radicalement sa motivation et son expérience d'utilisation.

L'éditeur de code a donc bel et bien une influence sur la PX mais aussi sur la productivité réelle du programmeur. [Zayour and Hajjdiab, 2013] identifie le manque d'expérience avec un environnement de développement comme un des principaux facteurs de manque de productivité. Rendre l'éditeur accessible et pragmatique est ainsi essentiel afin de pouvoir réduire l'impact que celui-ci peut avoir sur la productivité de l'utilisateur. [Zayour and Hajjdiab, 2013] souligne également que des fonctionnalités proposées par la plupart des environnements de développement permettent d'améliorer la productivité des programmeurs. Par exemple, l'*IntelliSense* permet de limiter le temps nécessaire à déboguer un programme comportant une erreur de syntaxe.

2.4 Exécution de code en ligne et virtualisation

Une fois le code écrit, les développeurs l'exécutent parfois afin de vérifier qu'il fonctionne correctement. Exécuter du code sur son ordinateur est une tâche aisée. Cependant, quid de l'exécution de code dans des environnements qui ne sont pas propices tels qu'un navigateur web ou une application mobile ? Au cours des années, des solutions ont émergées pour permettre d'exécuter du code en ligne.

2.4.1 Machines virtuelles

Une première approche consiste en l'exécution d'une ou plusieurs instances d'un système d'exploitation dans des machines virtuelles en utilisant un hyperviseur pour les gérer. Cette méthode permet de limiter les risques en termes de sécurité, mais introduit un overhead (Kamod and Jadhav, 2023). [Abbasi et al., 2010] présente XYLUS, un environnement de développement virtualisé qui utilise des machines virtuelles fonctionnant sur un serveur Hyper-V pour exécuter du code dans le navigateur sans devoir installer d'environnement de développement et d'exécution localement. Pour ce faire, ils utilisent le Remote Desktop Protocol pour se connecter à des machines virtuelles tournant sous une version allégée de Windows.

2.4.2 Containers

Une seconde approche est l'utilisation de containers (Mares and Blackham, 2012). En utilisant des containers, la compartimentation se fait au niveau de l'OS. Tous les containers présents sur une machine partagent le même noyau, celui de l'hôte. Les containers ont un overhead plus faible que celui des machines virtuelles, mais le partage d'un seul noyau les rend plus vulnérables aux attaques. Les containers ont été utilisés en 2020 dans [Došilović and Mekterović, 2020] dans Judge0, un environnement d'exécution de code en ligne et open source qui les utilise pour exécuter du code de manière sécurisée.

Chapitre 3

Approche

3.1 Problématique étudiée

L'examen de la littérature concernant les avancées en termes de développement sur appareils mobiles et la récente apparition du LSP nous amène à nous poser la question suivante :

RQ1. Comment mettre en place un éditeur de code mobile supportant et facilitant l'écriture et la manipulation de code dans des projets multilangages ?

Comment permettre, au moyen du LSP, d'avoir un éditeur de code essentiellement disponible sur tablette permettant la rédaction et la modification élégante de code source écrit dans différents langages au sein d'un même projet ? Cette question met ainsi en exergue certains défis et problèmes avec lesquelles il nous a fallu composer. Ainsi, pour répondre à cette problématique, une réponse aux questions suivantes a été trouvée :

RQ1.1. Comment supporter l'édition de plusieurs langages de programmation dans une application mobile ?

Quelle(s) solution(s) apporter pour permettre d'avoir un support pour une multitude de langages de programmation au sein du même éditeur de code mobile et tactile. Le *Language Server Protocol* (LSP), utilisé dans certains IDE disponibles sur ordinateur, viendra apporter une réponse à cette question adaptée aux contraintes propres à la nature de l'interaction et aux limites techniques de la technologie tant sur le plan logiciel que matériel.

RQ1.2. Comment adapter les outils d'écriture de code sur tablette en utilisant les fonctionnalités proposées par le *Language Server Protocol* ?

Comment intégrer les différentes fonctionnalités fournies par le LSP au sein de l'interface afin de faciliter les activités d'écriture de code de manière à viser l'efficacité et la productivité du développeur ?

RQ1.3. Comment adapter les outils de manipulation de code sur tablette en utilisant

les fonctionnalités proposées par le *Language Server Protocol* ?

Comment intégrer les différentes fonctionnalités fournies par le LSP au sein de l'interface afin de faciliter les activités de manipulation de code de manière à viser l'efficacité et la productivité du développeur ?

Ainsi, les questions relatives à l'exécution et la navigation dans le code, bien qu'essentielles pour imaginer un éditeur de code mobile complet, sont ici laissées de côté pour des travaux futurs et ne seront pas ou très peu explorées dans ce mémoire.

3.2 Méthodologie

Pour répondre à ces questions, nous avons dans un premier temps développé (1) un éditeur de code pour tablettes que nous avons par la suite soumis à la (2) validation d'utilisateurs-test.

1. L'éditeur de code a été conçu de manière à pouvoir intégrer facilement et rapidement le support d'un nombre conséquent de langages, de paradigmes parfois différents via l'utilisation des serveurs de langage et du protocole LSP. En combinant les fonctionnalités du LSP avec certains artefacts et concepts développés par la littérature, nous avons mis en place un IDE mobile, tactile et agnostique par rapport au langage utilisé.
2. La validation avec des utilisateurs-test visait à évaluer la qualité de l'interaction avec l'éditeur en termes de *Programmer Experience* et de productivité. L'objectif était ici d'évaluer si l'éditeur de code, grâce à son interface, permettait vraiment d'apporter une plus-value réelle aux programmeurs dans leur activité de développement sur des appareils mobiles.

Dans la suite, nous détaillons cet éditeur de code, l'expérience mise en place pour l'évaluer et les résultats obtenus après évaluation.

Chapitre 4

Implémentation

Dans ce chapitre, nous présentons l'éditeur de code conçu dans le cadre de ce mémoire. Cet éditeur a pour but de :

- Faciliter le support multilangage au sein d'un même projet grâce à l'utilisation des serveurs de langages ;
- Présenter une interface à la fois proche des éditeurs de code traditionnels, mais adaptée au contexte d'une utilisation uniquement tactile d'une tablette.

Pour ce faire, nous présentons dans un premier temps l'architecture logicielle permettant l'utilisation des serveurs de langage sur tablette et apportant des possibilités d'extension variées. Ensuite, nous précisons et justifions les choix technologiques faits pour implémenter cette architecture. Par après, nous détaillons comment l'interaction avec le serveur de langage est gérée et nous présentons les fonctionnalités implémentées dans l'éditeur. Finalement, l'interface utilisateur et ses adaptations propres à une utilisation tactile sont exposés.

4.1 Architecture logicielle

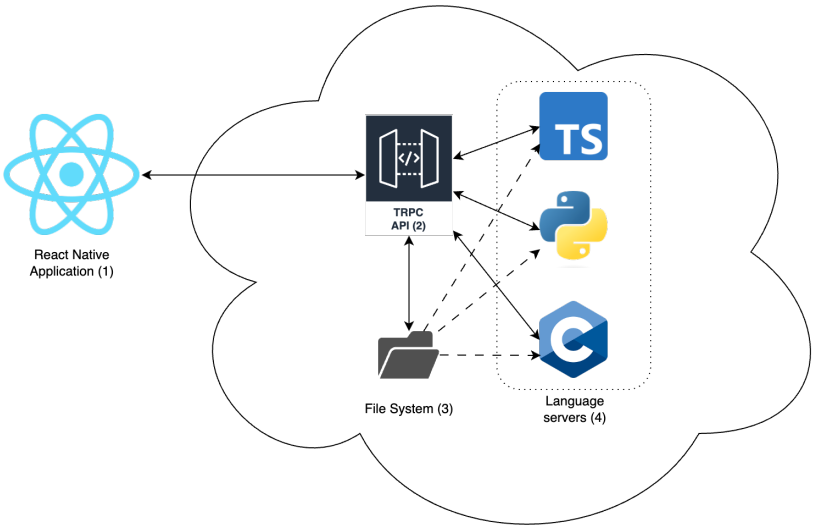


Figure 4.1 – Architecture de la solution logicielle

L'architecture logicielle, comme présentée dans la figure 4.1, est constituée de deux composantes principales. Il y a d'une part l'application mobile (1) et d'autre part l'API (2) à laquelle l'application mobile peut se connecter. L'API permet d'interagir avec le système de fichiers du serveur (3) et communique avec les différents serveurs de langage (4) installés sur celui-ci. Les serveurs de langage peuvent, eux aussi, accéder au système de fichiers du serveur.

4.1.1 API

L'API a un accès direct au système de fichiers de l'ordinateur distant et permet à l'utilisateur de créer, renommer et supprimer des dossiers et des fichiers. Elle permet aussi au client d'exécuter des commandes sur les différents serveurs de langage supportés. Soit le serveur retransmet directement le résultat de l'exécution au client, soit le serveur transforme le résultat et/ou exécute des actions à partir de celui-ci avant de le renvoyer. Puisque cette architecture fonctionne en réseau, la capacité du serveur à manipuler le résultat des exécutions peut être utilisée pour limiter la bande passante requise lors de certaines actions. En effet, certaines actions ont comme résultat une liste de manipulations à effectuer sur le code. Le client exécute ensuite ces manipulations en informant le serveur de langage du résultat de chacune d'entre elles. Une ou plusieurs requêtes sont donc envoyées dans ce processus. Dans un contexte classique où le serveur de langage se trouve sur la même machine que le client, il n'est pas dérangeant d'exécuter plusieurs requêtes pour une seule action. Cependant, lorsque ces requêtes sont effectuées en réseau et parfois sur des connexions peu rapides, il devient important de limiter le nombre d'interactions avec le réseau. La figure 4.2 illustre la différence entre l'utilisation de l'API et l'interaction directe avec le serveur de langage lors de l'appel de la procédure *textDocument/rename*.

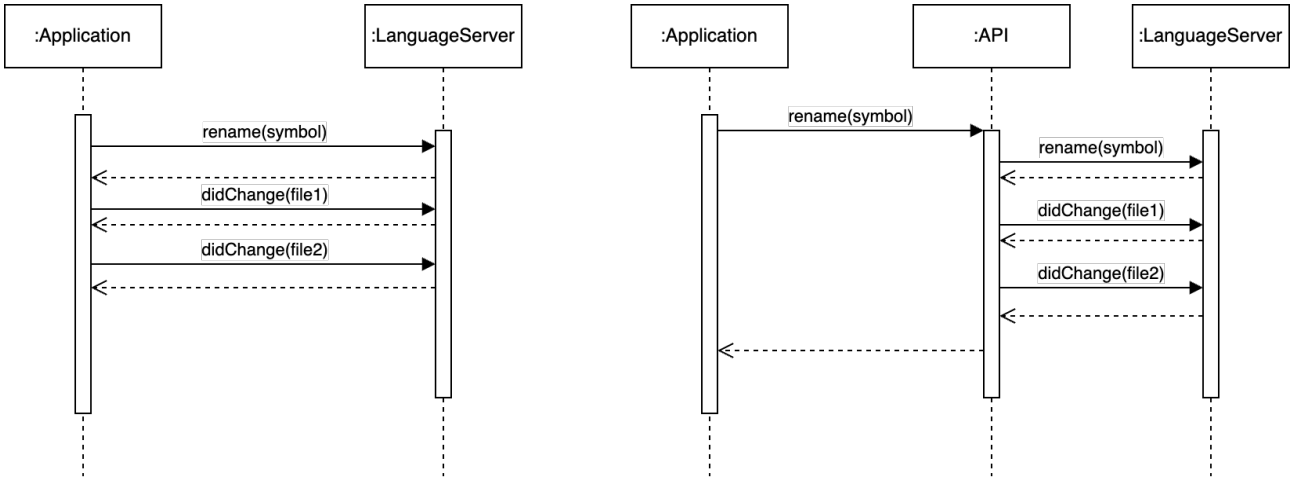


FIGURE 4.2 – Comparaison de la méthode classique par rapport à l'utilisation d'une API

Il se pourrait aussi que le serveur de langage ne supporte pas une fonctionnalité spécifique. Grâce à cette architecture, il serait néanmoins possible de retourner le résultat voulu en l'obtenant par un autre moyen que par le serveur de langage.

4.1.2 Application mobile

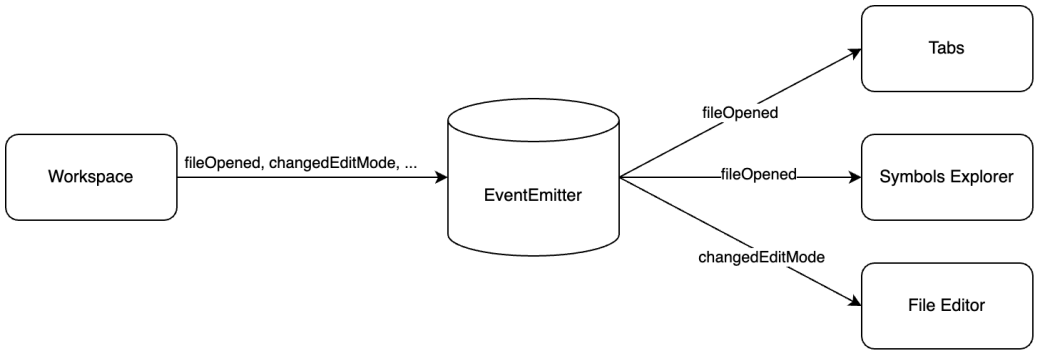


FIGURE 4.3 – Envoi d'évènements via l'EventEmitter

L'application comporte un composant principal appelé **Workspace**. Celui-ci a pour rôle d'initialiser les différents serveurs de langage au démarrage de l'application. Il permet aussi de savoir quels fichiers et dossiers sont ouverts dans l'application. Il sert de cache pour les fichiers ouverts par l'utilisateur. Comme le montre la figure 4.3, le pattern "Publish/Subscribe" est utilisé pour mettre à jour les différents composants de l'application. Prenons l'exemple de l'ouverture d'un fichier. Lorsque l'utilisateur clique sur le nom d'un fichier, le workspace commence le téléchargement de son contenu depuis le serveur. Lorsqu'il est téléchargé, la liste des fichiers ouverts est mise à jour et un évènement est envoyé au travers de l'application via un **EventEmitter**. En recevant l'évènement, les composants ayant besoin de cette liste sont mis à jour. Ces composants peuvent s'abonner aux sujets qui les intéressent via l'**EventEmitter**.

4.2 Choix de technologies

4.2.1 React Native

React Native [1] a été choisie comme technologie pour construire l'application mobile parce qu'elle permet de compiler l'application sur les deux plateformes majeures : Android et iOS (et iPadOS). Puisque les gestes sont utilisés dans le logiciel pour effectuer des manipulations sur le code, il était nécessaire d'utiliser des fonctionnalités natives qui ne sont pas disponibles sur des plateformes telles que les navigateurs web. En effet, bien que ces derniers supportent certains gestes, ils ne permettent pas de capturer aisément des gestes plus complexes. React Native semblait être plus pertinent que Flutter dans ce contexte puisqu'il permet d'utiliser le même langage sur le serveur et dans l'application mobile, et ainsi partager certains types, notamment ceux des requêtes et des réponses avec l'API.

4.2.2 TypeScript

Sa capacité à être utilisé autant dans l'application que dans le serveur ainsi que son utilisation dans la spécification du LSP a fait de TypeScript [2] un langage de choix pour ce projet. En effet, la spécification du LSP (Microsoft, 2024) fournit des types écrits dans la syntaxe de TypeScript, ce qui a facilité l'intégration du LSP dans la solution logicielle.

4.2.3 tRPC

tRPC[3] a été utilisé pour construire l'API. Il permet la création d'API fortement typées de bout en bout. Pour assurer la validité des entrées et des sorties, tRPC utilise des librairies de validation de schémas qui permettent de vérifier que les données qui transitent par l'API adhèrent aux règles de structure définies dans les schémas. Ici, la librairie *zod*[4] a été choisie parce qu'elle est utilisée dans les exemples fournis dans la documentation de tRPC.

Pour créer les schémas des entrées et sorties de l'API, un outil a été conçu pour extraire les types depuis la spécification du LSP. Une fois le moissonnage terminé, un traitement manuel a été effectué pour supprimer les types paramétriques qui ne peuvent pas être convertis en schémas zod. En retirant ces types problématiques, il est possible de convertir automatiquement les types Typescript en schémas zod via un utilitaire comme *ts-to-zod*[5].

4.2.4 Docker

Une image Docker a été créée pour le serveur de l'API. Cela permet d'installer les différents serveurs de langage utilisés dans un environnement contrôlé. Ainsi, dans cette image, nous retrouvons

1. `https://reactnative.dev/`
2. `https://www.typescriptlang.org/`
3. `https://trpc.io/`
4. `https://zod.dev/`
5. `https://www.npmjs.com/package/ts-to-zod`

un *langage server* pour *Swift, C, C++, JavaScript/Typescript* et *prolog*. À terme, Docker permettrait aussi de fournir un environnement sécurisé dans lequel exécuter le code qui est écrit par l'utilisateur de l'application.

4.3 Interaction avec les serveurs de langage

Par souci d'efficacité, les serveurs de langage ne sont instanciés que lorsqu'une première requête leur est destinée. Pour ce faire, le pattern Singleton a été utilisé pour créer une instance à la volée lorsqu'une requête est reçue. Dans le cadre de notre implémentation, l'application initialise tous les serveurs de langage lorsque qu'un nouveau workspace est ouvert, mais il serait tout à fait possible d'initialiser les serveurs en fonction des types de fichiers présents dans le workspace.

4.4 Fonctionnalités du LSP implémentées

Seule une petite partie des fonctionnalités proposées par le LSP a été implémentée. Bien qu'il ne s'agisse que d'une sélection des fonctionnalités disponibles, celles-ci nous permettent de justifier certains choix faits lors de l'intégration du LSP dans l'API.

4.4.1 textDocument/initialize

Cette procédure est utilisée pour créer une nouvelle session avec le serveur de langage. Elle permet de spécifier les options relatives à cette session telles que les fonctionnalités supportées et le workspace dans lequel le serveur de langage doit opérer. Certains serveurs de langage peuvent implémenter des options qui ne font pas partie de la spécification du LSP. Celles-ci peuvent néanmoins être passées via cette procédure.

4.4.2 textDocument/didChange

Cette procédure est appelée après un changement dans le contenu d'un fichier. Elle sert à prévenir le serveur de langage d'un changement effectué dans un fichier. Il peut ainsi utiliser le nouveau contenu dans les prochains appels de procédures.

4.4.3 textDocument/prepareRename

Lorsque l'utilisateur pousse sur un symbole à renommer, la procédure *textDocument / prepareRename* est appelée pour vérifier la validité de la demande de renommage à l'endroit spécifié.

4.4.4 textDocument/rename

Cette méthode est appelée lorsque l'utilisateur valide le renommage d'un symbole. Au niveau de l'API, cette procédure est implémentée en suivant la spécification du LSP pour les entrées, mais pas

pour les sorties. En effet, comme mentionné précédemment, *rename* fait partie des procédures qui peuvent être optimisées au niveau du serveur. Son fonctionnement peut être représenté par le pseudo-code suivant :

```
function renameSymbol
    changes ← textDocument/rename()
    updatedFileContents ← [ ]
    for change in changes do
        newFileContent ← apply(change)
        save(newFileContent)
        textDocument/didChange()
        updatedFileContents.push(newFileContent)
    end for
    return updatedFileContents
end function
```

Sans cette optimisation, pour une requête de renommage qui modifie 5 fichiers, nous aurions eu besoin de faire 11 requêtes HTTP. En effet, lors d'un renommage, *textDocument/rename* est appelée, suivie d'une sauvegarde de chaque fichier sur le serveur, puis un appel à *textDocument/didChange* pour chaque document. Avec l'optimisation, une seule requête HTTP est exécutée.

4.4.5 textDocument/formatting

Lorsque l'on développe sur tablette, il est plus complexe de s'assurer que le code que l'on écrit respecte les conventions de style mises en place. Le formatage des documents a donc été une des premières opérations de manipulation de code implémentée au sein de l'application. Ainsi, l'utilisateur peut se concentrer sur le code et utiliser la fonctionnalité de formatage lorsqu'il souhaite être assisté par son éditeur pour rendre le code plus lisible. Seul le formateur fourni par le serveur de langage est utilisé dans cette version de l'application. Cependant, il est important de noter que les fonctionnalités de formatage fournies par le serveur de langage sont relativement limitées, voire inexistantes en fonction des langages utilisés.

L'architecture permettrait l'utilisation d'un linter ou d'un formateur en complément ou à la place du formateur fourni par le serveur de langage. Dans un projet TypeScript, par exemple, il serait possible de charger la configuration ESLint (linteur pour ECMAScript) et prettier (formateur) depuis le workspace et les exécuter sur le code précédemment formaté par le serveur de langage TypeScript. Ainsi, on pourrait étendre les fonctionnalités du serveur de langage sans pour autant changer le protocole utilisé pour communiquer avec.

4.4.6 textDocument/completion

Cette procédure permet de récupérer une liste de propositions pour l'autocomplétion à un endroit donné. Certains serveurs de langage renvoient tous les symboles possibles s'ils ne savent pas produire des propositions pertinentes. Il est donc nécessaire de limiter le nombre de résultats envoyés afin d'éviter

que les requêtes en réseau ne prennent trop de temps. Par souci de simplicité, il a été décidé de ne renvoyer que les 10 premiers résultats de la requête. Cependant, l'architecture choisie permettrait de filtrer les résultats obtenus en fonction du contexte et donc de montrer les résultats les plus probables en premier.

4.4.7 textDocument/documentSymbol

Cette procédure permet de récupérer la liste des symboles présents dans un fichier. Dans l'application, cette requête est utilisée pour afficher les symboles présents dans les fichiers ouverts.

4.4.8 textDocument/didOpen et textDocument/didClose

Ces procédures sont utilisées pour respectivement dire au serveur de langage qu'un fichier a été ouvert ou fermé dans l'éditeur de code.

4.4.9 Autres fonctionnalités

Ces fonctionnalités n'ont pas été implémentées au niveau de l'application, mais sont disponibles au sein de l'API :

- **textDocument/codeActions** : liste des actions que l'utilisateur peut effectuer à l'aide du serveur de langage pour un endroit donné dans le code
- **textDocument/declaration** : donne des informations sur l'endroit où un symbole est déclaré
- **textDocument/definition** : donne des informations sur l'endroit où un symbole est défini
- **textDocument/hover** : donne des informations sur un élément se situant à un endroit spécifié dans le code
- **textDocument/implementation** : donne des informations sur l'endroit où un symbole est implémenté
- **textDocument/references** : liste toutes les références d'un symbole dans le workspace
- **textDocument/selectionRange** : liste des zones de sélections possibles autour d'un ensemble de points dans le code
- **textDocument/semanticTokens** : permet à un éditeur d'afficher des couleurs particulières pour certains symboles en fonction d'informations spécifiques à ceux-ci.
- **textDocument/typeDefinition** : donne des informations sur l'endroit de définition du type d'un symbole
- **window/showMessage** : permet au serveur de langage de demander au client d'afficher un message à l'utilisateur
- **window/logMessage** : permet au serveur de langage d'envoyer des logs au client

4.5 Interface adaptée

L'interface a été adaptée pour proposer 2 modes d'interaction avec le code. Le premier vise à fournir une interface plus classique où l'utilisateur peut écrire du code au moyen d'un clavier virtuel. Il peut aussi faire toutes les actions que l'on peut attendre d'une zone de texte telles que les sélections, les copier-coller, etc. Le deuxième mode vise à pouvoir manipuler le code : les actions de refactor et de formatage. Ces actions peuvent avoir leur propre bouton ou être activées par des gestes effectués sur le code.

4.5.1 Zone d'édition de fichier

La zone d'édition de fichier est semblable à celle présentée dans des éditeurs de code classiques comme Eclipse ou VSCode. Celle-ci est présente tant dans le mode d'édition que le mode de manipulation pour interagir avec le contenu du fichier.

Figure 4.4 – Zone d'édition de code

Dans les deux modes d'interaction, le contenu du fichier est décoré avec de la coloration syntaxique obtenue après transformation du résultat fourni par la librairie highlight.js. Cette librairie renvoie en effet une coloration syntaxique en situant les portions de code directement dans des balises de type *...* non directement utilisable en React Native. Ce mécanisme de transformation du résultat permettrait, par la suite, de combiner le résultat de la coloration syntaxique avec la coloration sémantique[6] qui serait apportée par les serveurs de langages afin de colorer le code de manière plus

6. `https://github.com/microsoft/vscode/wiki/Semantic-Highlighting-Overview`

précise.

4.5.2 Clavier

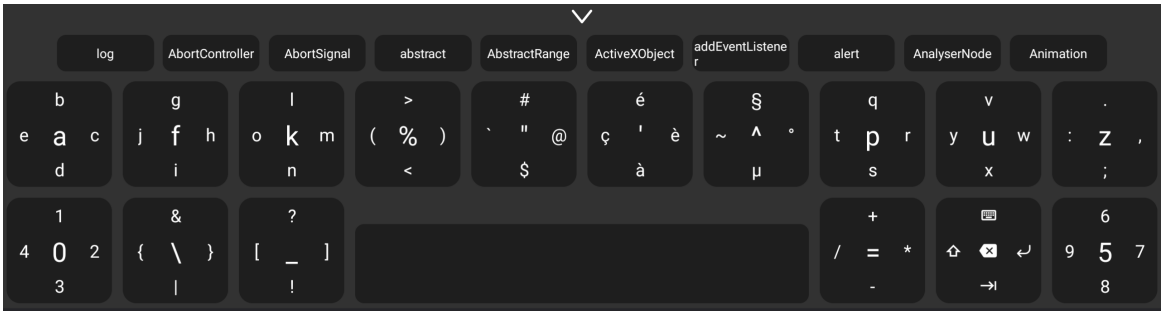


FIGURE 4.5 – Clavier adapté pour afficher plus de symboles et de l'autocomplétion

Écrire du code sur mobile n'est pas une tâche aisée et le clavier logiciel en est partiellement la cause. En effet, bien qu'il soit relativement adapté pour écrire du texte simple, il est peu adapté pour écrire du code contenant parfois des symboles qui ne sont accessibles qu'en entrant dans plusieurs menus.

C'est pourquoi, nous avons mis en place un clavier composé de 2 parties clés. Il y a d'une part la zone réservée à l'autocomplétion et d'autre part un clavier avec des touches inspirées de celles du *Tap and Slide Keyboard (TaS)* présenté dans **[Romanoa et al., 2014]**.

AUTOCOMPLÉTION

L'autocomplétion est fournie par les serveurs de langage au travers de l'appel à la procédure *textDocument/completion*. Seuls quelques résultats sont affichés sur le clavier. Lorsque l'utilisateur écrit de nouveaux caractères, des demandes d'autocomplétion sont envoyées au serveur de langage pour la nouvelle position du curseur. Lorsque les résultats sont disponibles, ils viennent remplacer les résultats précédents. En poussant sur un résultat, celui-ci est inséré dans le texte et le curseur est positionné à la fin d'un texte inséré. L'implémentation de la fonctionnalité d'autocomplétion est propre à chaque serveur de langage et il se peut donc que certains langages aient une autocomplétion de moindre qualité. C'est notamment le cas du serveur de *prolog* ou encore le serveur pour le langage C. Cette méthode permet néanmoins de créer un *syntax oriented keyboard* (Almusaly and Metoyer, 2015; Costagliola et al., 2018) fonctionnant avec tous les langages qui proposent les mots clés réservés dans leur autocomplétion. Par ailleurs, il se veut plus précis que le *syntax oriented keyboard* parce qu'il peut adapter les résultats en fonction du contexte et proposer, en fonction du serveur de langage, des tokens sémantiques directement dans les suggestions.

TOUCHES

Contrairement au *TaS* de [Romanoa et al., 2014], le clavier se veut plus orienté vers l'écriture de code. Il est par conséquent nécessaire de pouvoir accéder aux caractères spéciaux qui sont fort utilisés en programmation sans devoir naviguer dans un menu différent. Une autre différence vient du fait

que l'utilisateur n'est pas obligé de commencer son geste au centre de la touche. En effet, dans le *TaS*, l'utilisateur devait commencer son mouvement au centre de la touche. Pour limiter ce besoin de précision, nous pouvons capturer le geste à partir de n'importe quel endroit sur la touche et capturer la destination. De cette manière, il est possible d'activer la lettre "b" en faisant un simple swipe vers le haut. Cela permettrait aux utilisateurs ayant l'habitude d'utiliser le clavier d'aller plus vite en retirant la nécessité de se concentrer pour toucher le milieu de la touche.

Au niveau de la position des touches sur le clavier, nous avons fait le choix de positionner la moitié des lettres et des chiffres à gauche et l'autre à droite du clavier pour faciliter une écriture à deux mains du code. De plus, les caractères spéciaux ont été regroupés par famille en fonction de leur utilité dans un contexte de développement. Ainsi, les opérateurs mathématiques et logiques ainsi que la ponctuation de base ont toutes été isolées sur une touche en particulier. Une organisation alphabétique des lettres, comme celle du *TaS* (Romanoa et al., 2014), a aussi été préférée à une autre pour faciliter l'utilisation du clavier.

Gestion de l'interaction tactile avec les zones d'édition

React Native ne permet pas de générer des évènements système comme l'occurrence d'une touche sur un clavier natif. Afin de pouvoir contourner ce problème et pouvoir interagir avec tout type de zone d'édition, un pattern semblable à un *observer* a été mis en place et est présenté dans la figure 4.6. Le clavier expose un gestionnaire d'évènements (*KeyboardEventManager*) sur lequel n'importe quelle zone d'édition peut venir s'attacher afin de recevoir la touche pressé par l'utilisateur. Comme il est nécessaire de ne pas avoir plusieurs zones d'édition appairées simultanément au clavier, chaque fois qu'une nouvelle zone vient se greffer sur le clavier, l'ancienne zone est immédiatement déconnectée du clavier. Une zone d'édition a également la possibilité de se débrancher par elle-même du clavier, par exemple quand l'utilisateur ferme un fichier ouvert.

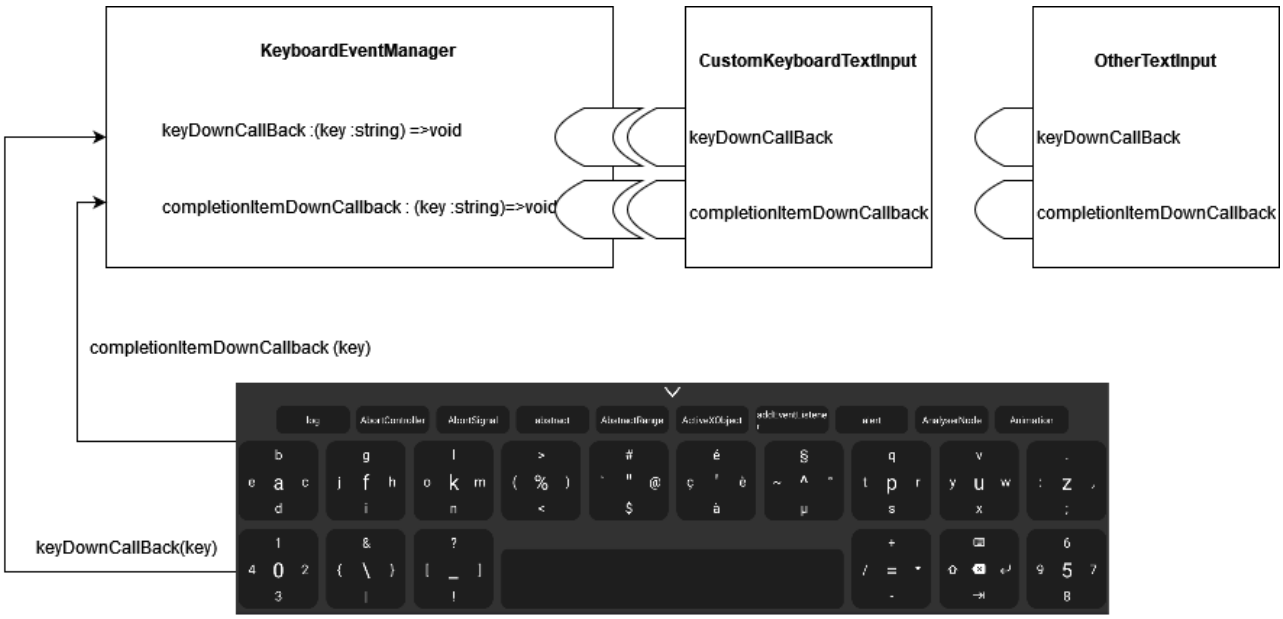


Figure 4.6 – Gestion de l'interaction du clavier avec les zones d'éditions

4.5.3 Manipulation de code

Les gestes sont utilisés dans le mode de manipulation de code afin de fournir des actions rapides sans surcharger l'interface dont la superficie est limitée sur tablette. Ces gestes ne peuvent pas être implémentés sur les zones de textes classiques en raison de contraintes techniques. Il est donc nécessaire d'utiliser une vue à part sur laquelle il est possible d'ajouter de la reconnaissance de gestes.

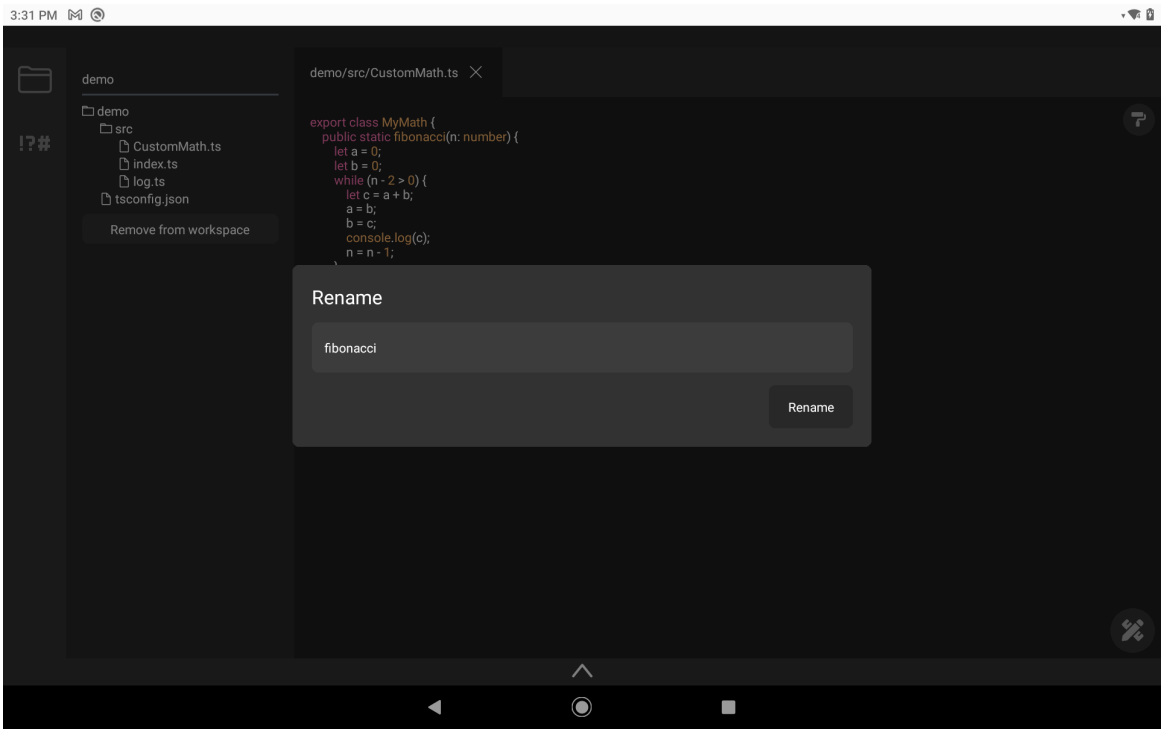


FIGURE 4.7 – Renommage d'un symbole

Ainsi, il est possible d'invoquer la fonctionnalité de renommage de symboles depuis cette vue. En tapant 2 fois sur un nom de symbole, une requête est faite au serveur de langage pour savoir s'il peut être renommé. Si tel est le cas, une boite de dialogue s'ouvre alors et propose à l'utilisateur de fournir un nouveau nom pour le symbole. Il peut alors décider de renommer celui-ci ou annuler. En validant le renommage, une requête est envoyée au serveur, qui se chargera de renommer le symbole dans tout le workspace grâce au serveur de langage.

L'idée ainsi de ce mode de manipulation est de pouvoir intégrer les gestes de refactoring découverts dans [Raab et al., 2013] et [Biegel et al., 2014] compatibles avec le LSP. Ces fonctionnalités restent encore à être implémentées plus en profondeur en utilisant le mécanisme de reconnaissance de gestes déjà mis en place.

Le mode de manipulation de code apporte aussi la possibilité d'exécuter des actions rapides comme le formatage au travers d'un bouton accessible depuis cette interface.

4.5.4 Navigation

La navigation au sein de l'application est faite via un menu sur la gauche de l'écran contenant une arborescence de dossiers et fichiers présents dans le workspace. Lorsque l'utilisateur pousse sur le nom d'un fichier, ce fichier est ouvert dans un système d'onglets. Pour rendre les onglets plus faciles à manipuler, leur taille et la taille de la croix permettant de les fermer ont été adaptées pour qu'il soit facile de les utiliser sur un écran tactile. Depuis ce menu, il est aussi possible de créer, renommer et supprimer des fichiers.

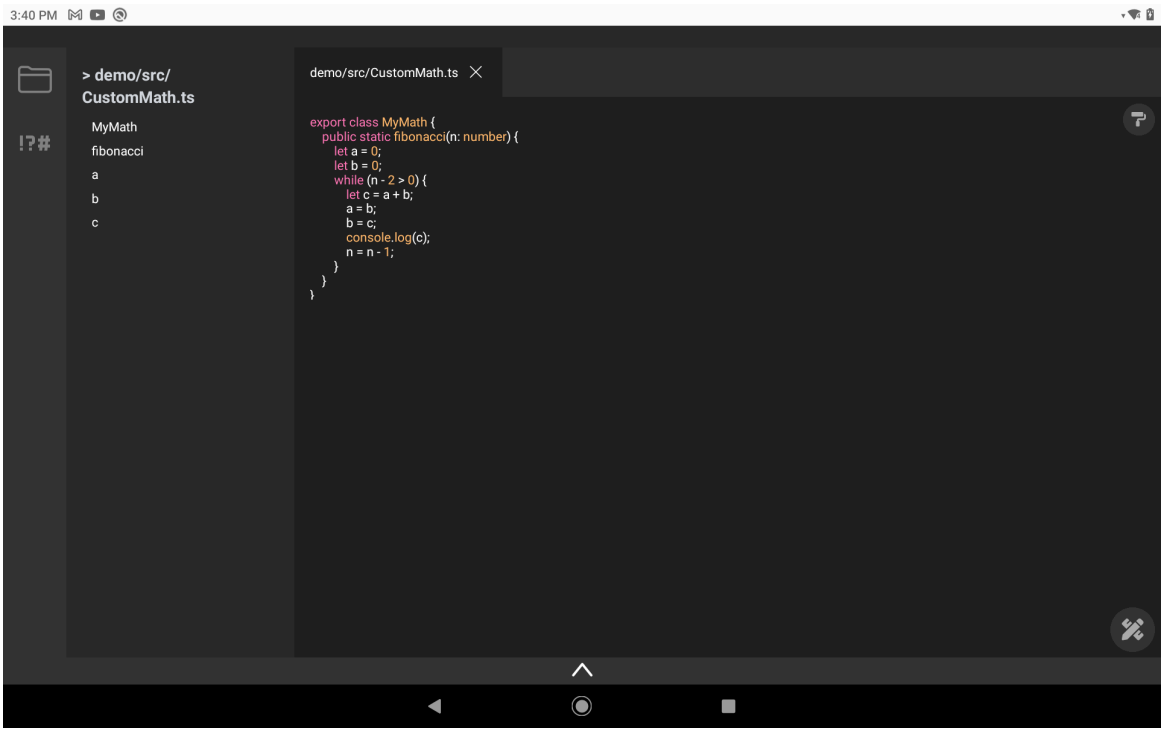


Figure 4.8 – Explorateur de symboles

Une deuxième méthode de navigation est l'utilisation de l'explorateur de symboles. Cette vue permet d'afficher la liste des symboles présents dans les fichiers ouverts. Toutes les classes, méthodes, fonctions, propriétés et variables s'y retrouvent dans leur ordre d'apparition dans le document. Lorsque l'utilisateur pousse sur le nom d'un symbole, l'onglet relatif au fichier concerné est ouvert et le curseur est placé à la position du symbole dans l'éditeur. Ainsi, il est possible de naviguer aisément dans les fichiers au travers d'une vue plus condensée. Cela permet aussi de retirer le besoin de scroller dans des fichiers de plus grandes tailles puisque la zone de texte s'adapte à la position du curseur.

4.6 Prototype

Un prototype de l'application est publié sur un dépôt disponible sur GitHub [7]. Il est accompagné d'une documentation permettant à d'autres développeurs de pouvoir reprendre le travail déjà effectué. La documentation explique aussi comment mettre l'API en production au travers d'un reverse proxy tel que Treafik [8].

L'application React Native compte environ 3600 lignes de code réparties sur 76 fichiers. Elle contient 10 hooks et 33 composants React. Le serveur, quant à lui, compte environ 3200 lignes de code réparties sur 58 fichiers. Parmi ces 3200 lignes, 1900 sont issues de la documentation du *Language Server Protocol*.

Afin d'assurer une certaine qualité du code écrit, ESLint [9] a été utilisé pour forcer le respect d'un ensemble de règles qui ont été définies au début du projet. Cela permet ainsi de s'assurer que le code produit respecte un ensemble de règles stylistiques, mais cela permet aussi d'éviter certains bugs. C'est le cas des règles concernant l'utilisation des promesses, par exemple. SonarQube [10] a aussi été utilisé pour mettre en avant les éventuels code smells pouvant être trouvés dans un projet de cette taille.

7. `https://github.com/snail-unamur/EdgeRunner`
8. `https://traefik.io/`
9. `https://eslint.org/`
10. `https://www.sonarsource.com/products/sonarqube/`

Chapitre 5

Évaluation

Des tests utilisateurs ont été effectués avec des étudiants allant de la deuxième année de bachelier à la deuxième année de master. Dans ces tests, il est demandé aux utilisateurs de construire un petit programme en utilisant l'éditeur de code sur tablette.

5.1 Méthodologie

5.1.1 Environnement

Les tests utilisateur sont effectués sur un iPad de 8^e génération avec un écran de 10.2 pouces. Les participants sont assis devant la tablette qui est mise sur un support vertical, dans une disposition paysage. Il est mentionné aux participants qu'ils peuvent positionner la tablette comme ils le souhaitent. L'application est déjà ouverte et un dossier est ouvert dans l'environnement de travail. La structure d'un projet TypeScript est déjà présente :

- src/
  - *log.ts* fonction de logging importée dans index
  - *index.ts* point d'entrée de l'application
- *tsconfig.json* fichier de configuration utilisé par le serveur de langage

Une liste d'instructions est donnée aux utilisateurs. Les étapes suivantes sont suivies :

- Créer un fichier CustomMath.ts dans *src/*
- Créer une classe *CustomMath* dans ce fichier
- Ajouter une méthode statique appelée fib
- Ajouter le paramètre n de type number à la méthode fib
- Implémenter Fibonacci dans fib
- Exporter la classe *CustomMath*
- Importer *CustomMath* dans le index.ts
- Appeler *CustomMath.fib(5)*
- Renommer la méthode fib
- Renommer la classe *CustomMath*
- Appliquer un formatage dans le fichier CustomMath.ts

De cette manière, l'utilisateur est amené à utiliser le clavier pour écrire le code et l'autocomplétion pour les mots clés du langage, pour les symboles et pour les chemins des imports. Il est ensuite amené

à utiliser des fonctionnalités plus rares sur des éditeurs mobiles comme le renommage de symboles ou le formatage de code.

5.1.2 Questionnaire

À l'issue de ce test utilisateur, il est demandé aux participants de remplir un *User Experience Questionnaire* (UEQ) (Andreas et al., 2018) via la tablette. Ce questionnaire est conçu pour mesurer l'expérience utilisateur de produits interactifs. Il propose 6 échelles de mesures au travers de 26 items :

- **Attractivité :** Impression générale du produit. Les utilisateurs aiment-ils ou n'aiment-ils pas le produit ?
  6 items : agaçant/agréable, bien/médiocre, repoussant/attractif, désagréable/agréable, attrayant/rébarbatif, sympathique/inamical
- **Compréhensibilité :** Est-il facile de se familiariser avec le produit ? Est-il facile d'apprendre à comment utiliser le produit ?
  4 items : incompréhensible/compréhensible, appropriation simple/appropriation compliquée, compliqué/simple, clair/déroutant
- **Efficacité :** Les utilisateurs peuvent-ils accomplir leurs tâches sans effort inutile ?
  4 items : rapide/lent, efficace/inefficace, pragmatique/non pragmatique, sobre/surchargé
- **Contrôlabilité :** L'utilisateur a-t-il le sentiment de maîtriser l'interaction ?
  4 items : prévisible/imprévisible, rigide/facilitant, sécurisant/insécurisant, répond aux attentes/ne répond pas aux attentes
- **Stimulation :** L'utilisation du produit est-elle excitante et motivante ?
  4 items : apporte de la valeur/peu de valeur ajoutée, inintéressant/intéressant, stimulant/soporifique, ennuyeux/captivant
- **Originalité :** le produit est-il innovant et créatif ? Le produit suscite-t-il l'intérêt des utilisateurs ?
  4 items : moderne/sans fantaisie, original/conventionnel, habituel/avant-gardiste, conservateur/innovant

Dans le questionnaire, les items sont représentés par des paires de termes ayant des significations opposées. Pour chaque échelle, la moitié des items commencent avec le terme positif et l'autre moitié avec le terme négatif. L'UEQ utilise une échelle de -3 à 3 où le 0 est une réponse neutre.

Une fois l'UEQ rempli, des questions ouvertes sont posées afin de récolter plus d'informations :

- Qu'avez-vous le plus aimé dans le logiciel présenté ?
- Qu'avez-vous le moins aimé dans le logiciel présenté ?
- Qu'avez-vous pensé de la coloration syntaxique ?

Cette dernière question survenait souvent, car la plupart des utilisateurs tests, habitué à avoir la coloration dans tous leurs éditeurs, n'en faisait souvent pas mention.

	1	2	3	4	5	6	7		
Agaçant	○	○	○	○	○	○	○	Agréable	1
Incompréhensible	○	○	○	○	○	○	○	Compréhensible	2
Moderne	○	○	○	○	○	○	○	Sans fantaisie	3
Appropriation simple	○	○	○	○	○	○	○	Appropriation compliquée	4
Apporte de la valeur	○	○	○	○	○	○	○	Peu de valeur ajoutée	5
Ennuyeux	○	○	○	○	○	○	○	Captivant	6
Inintéressant	○	○	○	○	○	○	○	Intéressant	7
Imprévisible	○	○	○	○	○	○	○	Prévisible	8
Rapide	○	○	○	○	○	○	○	Lent	9
Original	○	○	○	○	○	○	○	Conventionnel	10
Rigide	○	○	○	○	○	○	○	Facilitant	11
Bien	○	○	○	○	○	○	○	Médiocre	12
Compliqué	○	○	○	○	○	○	○	Simple	13
Repoussant	○	○	○	○	○	○	○	Attractif	14
Habituel	○	○	○	○	○	○	○	Avant-gardiste	15
Désagréable	○	○	○	○	○	○	○	Agréable	16
Sécurisant	○	○	○	○	○	○	○	Insécurisant	17
Stimulant	○	○	○	○	○	○	○	Soporifique	18
Répond aux attentes	○	○	○	○	○	○	○	Ne répond pas aux attentes	19
Inefficace	○	○	○	○	○	○	○	Efficace	20
Clair	○	○	○	○	○	○	○	Déroutant	21
Non pragmatique	○	○	○	○	○	○	○	Pragmatique	22
Sobre	○	○	○	○	○	○	○	Surchargé	23
Attrayant	○	○	○	○	○	○	○	Rébarbatif	24
Sympathique	○	○	○	○	○	○	○	Inamical	25
Conservateur	○	○	○	○	○	○	○	Innovant	26

FIGURE 5.1 – *User Experience Questionnaire* (UEQ) (Andreas et al. 2018)

5.2 Résultats

Les résultats ont été obtenus auprès d'étudiants de deuxième année de bachelier (1), de troisième année de bachelier (1), de première année de master (2) et de deuxième année de master (6), soit un total de 10 étudiants. L'âge des participants varie de 20 à 36 ans avec un âge moyen de 23 ans. Trois d'entre eux ont déjà une expérience professionnelle dans le domaine de la programmation.

5.2.1 Résultats bruts

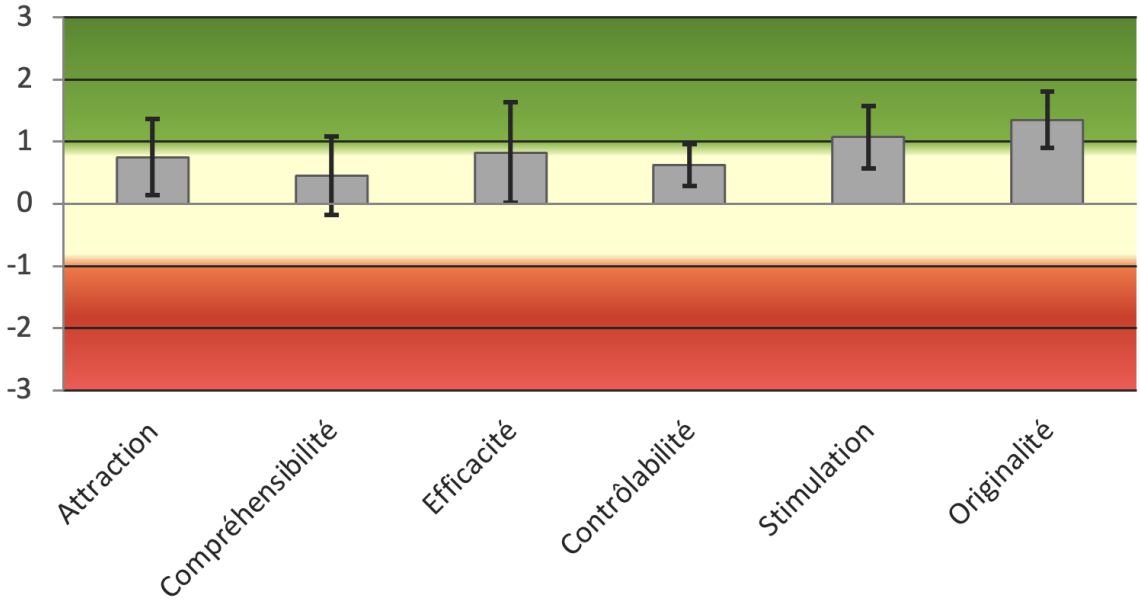


Figure 5.2 – Résultats de l'évaluation en utilisant l'UEQ

Le graphique précédent est généré par les outils d'analyse de l'UEQ et permet une analyse visuelle des résultats obtenus. Les valeurs situées entre -0,8 et 0,8 sont considérées comme des résultats neutres. Les valeurs strictement supérieures à 0,8 sont des résultats positifs et, inversement, des valeurs strictement inférieures à -0,8 sont considérées comme des valeurs négatives (Schrepp, 2023).

Sur base de ce graphique et malgré les limites de la collecte de ces résultats (voir section 5.3), on peut remarquer que les participants ont trouvé que l'application était originale, stimulante et tendait à être efficace. Leur avis concernant l'attraction, la compréhensibilité et la contrôlabilité était relativement neutre. Les résultats complets des réponses à l'UEQ sont disponibles dans l'annexe B.

5.2.2 Écart-type

L'écart-type peut être utilisé pour mesurer à quel point les participants sont d'accord entre eux. Au plus l'écart-type est faible, au plus les participants sont unanimes (Schrepp, 2023).

Les écart-types peuvent être interprétés de la manière suivante :

— Fort accord : écart-type de l'échelle inférieur à 0,83.

Échelles UEQ	Moyenne	Variance	Écart-type
Attraction	0,750	0,97	0,985230436
Compréhensibilité	0,450	1,04	1,01925899
Efficacité	0,825	1,71	1,307297892
Contrôlabilité	0,625	0,30	0,543266867
Stimulation	1,075	0,65	0,808376288
Originalité	1,350	0,53	0,728392446

TABLE 5.1 – Échelles UEQ avec Moyenne, Variance et Écart-type

— Accord moyen : écart-type de l'échelle compris entre 0,83 et 1,01.

— Faible accord : écart-type de l'échelle supérieur à 1,01.

On peut donc en conclure que les participants sont fortement d'accord concernant le résultat positif de la stimulation et l'originalité ainsi que pour le résultat neutre de la contrôlabilité. Ils sont moyennement d'accord concernant l'attraction et ils ne sont qu'en faible accord pour la compréhensibilité et l'efficacité.

5.2.3 Benchmark

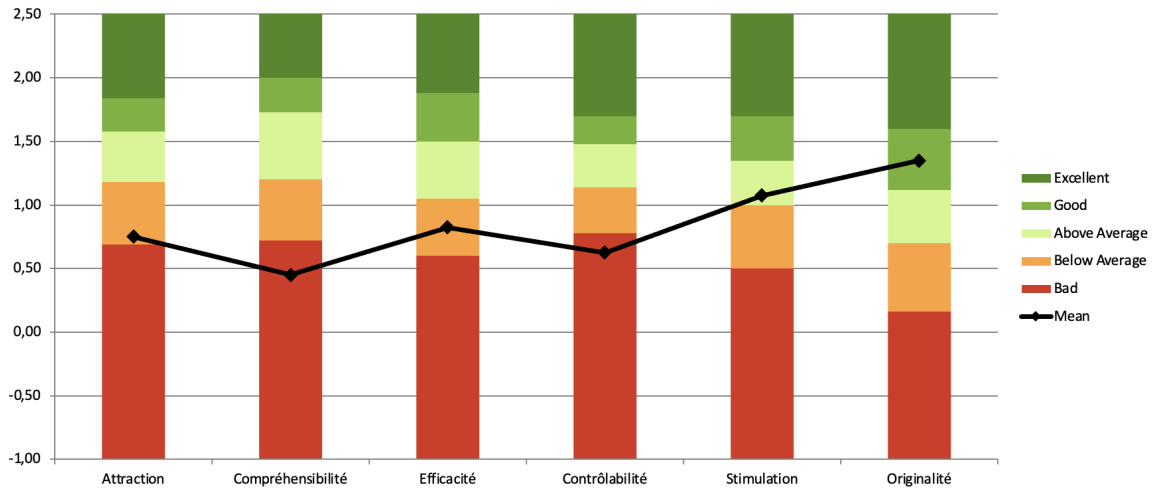


FIGURE 5.3 – Comparaison de l'application avec le benchmark

Il peut être intéressant de comparer l'expérience des utilisateurs par rapport à d'autres produits existants. C'est pour cette raison que les outils d'analyse de l'UEQ viennent avec un benchmark permettant de comparer les résultats obtenus aux résultats obtenus par d'autres applications. Ainsi, le graphique ci-dessus compare les résultats obtenus par l'application aux résultats de 452 autres produits évalués avec l'UEQ. Les résultats sont classés dans 5 catégories (Schrepp, 2023) :

— **Excellent** si le résultat du produit est dans la fourchette des 10% des meilleurs résultats

— **Bon** si 10% des résultats dans le dataset sont meilleurs et 75% des résultats sont moins bons.

— **Au-dessus de la moyenne** si 25% des résultats sont meilleurs et 50% des résultats sont moins bons.

— **En dessous de la moyenne** si 50% des résultats sont meilleurs et 25% des résultats sont moins bons.
— **Mauvais** si le résultat du produit se situe dans la fourchette des 25% des pires résultats

L'application obtient donc une bonne note par rapport à l'originalité et une note légèrement au-dessus de la moyenne en ce qui concerne la stimulation. En termes d'attraction et d'efficacité, elle se situe en dessous de la moyenne. Sa contrôlabilité et sa compréhensibilité sont, quant à elles, mauvaises dans ce benchmark.

5.2.4 Questions ouvertes et remarques

Des notes ont été prises durant les expériences et lorsque les participants remplissaient l'UEQ. Les réponses aux questions ouvertes ont aussi été notées. Les différentes remarques et pistes d'amélioration sont reprises ci-dessous. Les réponses et notes complètes sont disponibles dans l'annexe C.

Renommage

Concernant le popup de renommage de fichiers, plusieurs remarques ont été émises. Le clavier ne devrait pas recouvrir le popup et le popup devrait s'adapter à la position du clavier. Par ailleurs, certains utilisateurs ont mentionné le fait qu'ils auraient préféré que le clavier utilisé pour le renommage soit le même clavier que celui utilisé dans l'éditeur de code et non le clavier du système.

Clavier

De manière générale, les utilisateurs trouvent que l'utilisation du clavier est assez lente, mais ils précisent qu'avec l'habitude, ils seraient probablement capables d'écrire plus vite. Beaucoup ont rencontré des difficultés avec le choix de disposition des touches triées par ordre alphabétique. Certains auraient préféré pouvoir choisir la disposition de clavier et trouvent cela perturbant que l'AZERTY n'ait pas été choisi.

Autocomplétion

Certains participants ont utilisé l'autocomplétion directement, d'autres ne l'ont pas remarquée avant qu'on ne leur signale. Beaucoup ont trouvé qu'elle manquait de pertinence dans les résultats affichés et il aurait été souhaitable qu'elle affiche les résultats les plus probables en premier. Même après avoir remarqué la présence de l'autocomplétion, certains utilisateurs ne pensaient pas à l'utiliser parce qu'ils étaient captivés par le clavier. Un utilisateur a mentionné le fait qu'il aurait préféré avoir l'autocomplétion au niveau du texte et non au niveau du clavier. Un autre utilisateur mentionne qu'en l'absence de résultat pour l'autocomplétion, les symboles mathématiques pourraient être affichés à la place des résultats.

Smart typing

Le manque de smart typing semble poser problème à de nombreux utilisateurs. En effet, ils sont habitués à l'indentation automatique lors des retours à la ligne et à la fermeture automatique des parenthèses et autres symboles similaires. Dans les éditeurs de code modernes, les nouvelles lignes insérées possèdent la même indentation que la ligne précédente. Aussi, lorsque l'on insère des symboles allant par paire comme les parenthèses, les accolades ou encore les crochets, l'insertion d'un premier symbole entraîne l'insertion du second.

Curseur de sélection

Certains participants ont mentionné le fait que l'ajout de flèches sur le clavier permettrait de se déplacer plus facilement dans le code. La sélection au doigt n'étant pas assez précise. Un participant a mentionné le fait qu'il aurait apprécié pouvoir naviguer dans le code en swipant sur la barre espace, comme avec le clavier natif sur iPadOS.

Intelisense

Certains utilisateurs ont mis en avant qu'il manquait l'affichage des erreurs dans le code. Il est plus difficile de voir les erreurs comme les fermetures des accolades ou des parenthèses.

Découvrabilité

Concernant le mode de manipulation de code, certains utilisateurs ont éprouvé des difficultés à trouver le geste de renommage du fichier. Certains auraient préféré un menu contextuel pour remplacer les gestes comme mis en place dans la fenêtre avec l'arborescence de fichiers. D'autres encore ont trouvé que le bouton de formatage  n'était pas particulièrement intuitif.

IA générative

Pour finir, certains utilisateurs ont mentionné le fait que l'IA générative pourrait être utilisée pour accélérer l'écriture de code et proposer une autocomplétion encore plus adaptée au contexte.

5.3 Menaces à la validité

Comme dans tout travail scientifique, l'expérience présentée ici est sujette à un ensemble de menaces à la validité tant interne qu'externe. Dans cette partie, nous décrivons ces menaces.

5.3.1 Biais de l'expérimentateur

Les participants sélectionnés étaient tous des personnes que nous connaissions de près ou de loin puisqu'il s'agissait d'autres étudiants de la faculté d'informatique. Il est donc possible que nous ayons agi différemment d'une expérience à l'autre, ce qui pourrait avoir un impact sur les résultats. Toutefois, pour limiter cet impact, nous avons créé une liste d'étapes à suivre et les discussions sur l'application n'ont eu lieu qu'une fois le questionnaire rempli.

5.3.2 Biais d'échantillonnage

Les participants sélectionnés sont tous des étudiants en faculté d'informatique. Nous avons essayé d'avoir des participants de chaque année. Cependant, nous n'avons pas eu de participants en première année de bachelier. Ces étudiants n'auraient potentiellement pas eu les mêmes attentes que les étudiants représentés dans cet échantillon puisqu'ils n'ont pas la même expérience avec des éditeurs de code classiques.

5.3.3 Taille de l'échantillon

Seules 10 personnes ont participé à l'expérience. Il serait donc possible que certains résultats ne soient pas totalement généralisables compte tenu de la petite taille de l'échantillon. À noter que, au niveau des commentaires généraux sur l'application, les dernières interviews tendaient à ne plus apporter de nouveaux éléments.

5.3.4 Langage de programmation utilisé

Une partie des étudiants ayant participé à l'expérience n'étaient pas familiers avec TypeScript, mais tous avaient des bases en programmation et étaient familiers avec le langage JavaScript. Tout code JavaScript valide étant aussi valide en TypeScript, les étudiants pouvaient utiliser leurs connaissances en JavaScript pour réaliser l'expérience. Des exemples de code TypeScript étaient aussi donnés dans la liste d'instructions.

Chapitre 6

Discussion

Dans ce chapitre, nous discutons et mettons en perspective l'implémentation et les résultats obtenus lors de l'évaluation vis-à-vis de l'état de l'art et de notre question de recherche : **RQ1. Comment mettre en place un éditeur de code mobile supportant et facilitant l'écriture et la manipulation de code dans des projets multilangages ?** Dans cette idée, nous analysons ici autant les solutions d'implémentations mises en place pour répondre à la problématique que les résultats obtenus par l'expérimentation liés à ces solutions (pour celles liées à l'expérience utilisateur).

6.1 Évaluation de la mise en place de support multilangage

Permettre l'écriture dans plusieurs langages de programmation est une préoccupation principale de ce mémoire, matérialisée par la question de recherche : **RQ1.1. Comment supporter l'édition de plusieurs langages de programmation dans une application mobile ?**

6.1.1 Évaluation de la solution logicielle proposée

L'architecture logicielle semble convenir à un éditeur de code mobile, car elle :

- permet l'intégration sans difficulté de nouveaux supports de langage au sein de l'éditeur, sans avoir à modifier l'interface graphique
- autorise, via l'API, de venir brancher des modules supplémentaires venant supplanter l'implémentation de base d'un serveur de langage afin de permettre de fournir un support plus perfectionné pour certains langages.
- n'impacte pas les performances d'utilisation (peu de latence) malgré l'utilisation d'une API externe à l'application mobile.

De plus, l'éditeur, via des fonctionnalités comme la coloration syntaxique adaptée à chaque langage et l'ensemble des fonctionnalités récupérées depuis les serveurs de langages, propose un réel support complètement agnostique du langage utilisé à l'utilisateur.

La gestion de support multilangage au sein d'un même projet n'a pas été formellement évaluée avec des utilisateurs, mais bel et bien testée lors du développement et est disponible dans l'application développée.

6.1.2 Architecture logicielle alternative

L'approche la plus souhaitable aurait été de pouvoir utiliser les serveurs de langage directement au sein de l'application mobile. De cette manière, il aurait été possible d'utiliser la tablette dans des endroits où la connectivité cellulaire n'est pas optimale tels que dans le train, par exemple. Cependant, cette approche vient avec quelques inconvénients.

Contrainte technique

En effet, la contrainte technique majeure lorsque l'on développe sur mobile est le manque de flexibilité donnée par les systèmes d'exploitation. Leur mode de fonctionnement fait qu'il est impossible d'exécuter un serveur de langage en lançant un nouveau processus comme on le ferait sur un ordinateur classique.

Au cours des dernières années, le web a vu arriver un système permettant d'exécuter du code écrit dans des langages comme le C ou le Rust directement depuis le navigateur. Il consiste en l'exécution d'un bytecode dans une *sandbox*. Il est alors possible d'exécuter des programmes tels que FFMPEG ou encore SWI-Prolog dans le navigateur. Mais le WebAssembly ne se limite pas qu'aux navigateurs web. En effet, il est possible d'exécuter du WebAssembly dans une application mobile grâce à des implémentations telles que WasmKit ou react-native-wasm.

Il serait alors techniquement possible de compiler les serveurs de langages vers WebAssembly pour pouvoir les exécuter dans des applications mobiles.

Il faudra toutefois garder en tête qu'il serait nécessaire d'adapter certains serveurs de langage pour qu'ils fonctionnent dans l'environnement WebAssembly. En effet, tous les programmes, même s'ils sont écrits dans des langages supportés comme target du WebAssembly, ne fonctionneraient peut-être pas tel quel. D'autres encore ne sont pas écrits dans des langages compilables vers WebAssembly. Il faudrait alors créer un nouveau serveur de langage, ce qui va à l'encontre de l'idée initiale de faire en sorte de supporter un maximum de langages avec un minimum d'adaptations.

Contraintes matérielles

En exécutant les serveurs de langage directement sur l'appareil, il serait possible d'utiliser les fonctionnalités relatives aux langages directement dans l'application sans avoir besoin d'utiliser une connexion à internet. Cependant, cela vient aussi avec des inconvénients majeurs qui compromettent l'efficacité de sa mise en place.

D'une part, la taille de l'application peut devenir assez volumineuse. En effet, certains serveurs de langage peuvent peser près d'un giga octet lorsque l'on prend en considération les dépendances avec lesquelles ils sont fournis. Par conséquent, intégrer plusieurs serveurs dans le même bundle pose problème lorsque l'on considère le fait que l'espace de stockage présent sur les tablettes est d'environ 64 Go pour les modèles d'entrée de gamme. Pour contrer partiellement ce problème, il serait possible de ne pas inclure les fichiers WASM dans le bundle mais de les télécharger par la suite. In fine, au plus l'utilisateur utilisera de langages, au plus la taille de l'application risque d'être volumineuse.

D'autre part, les serveurs de langage sont assez gourmands en termes de ressources comme la RAM et le CPU lorsque les projets prennent de l'ampleur. Il semble alors compliqué d'exécuter de tels serveurs sur des appareils parfois dotés de 8GB de RAM ou moins. Il est évident que certaines tablettes sont équipées de processeurs équivalents ou supérieurs à ceux des ordinateurs. C'est, par exemple, le cas des iPad PRO d'Apple qui, en mai 2024, sont dotés d'une puce plus performante que les puces présentes dans certains ordinateurs de la marque. Cependant, une partie des personnes susceptibles d'utiliser une tablette comme outil de développement principal sont les personnes qui l'utilisent parce qu'elles sont un moyen plus abordable d'accéder à la technologie. C'est, par exemple, le cas des pays africains, où les appareils mobiles sont bien plus intégrés dans le quotidien de la population.

6.2 Évaluation des fonctionnalités d'écriture de code

Écrire du code, de manière intelligente et de manière similaire pour plusieurs langages, c'est la question posée dans notre deuxième question de recherche : **RQ1.2. Comment adapter les outils d'écriture de code sur tablette en utilisant les fonctionnalités proposées par le *Language Server Protocol* ?**Afin de permettre l'écriture de code sur tablette, c'est le clavier virtuel qui a été principalement adapté et enrichi.

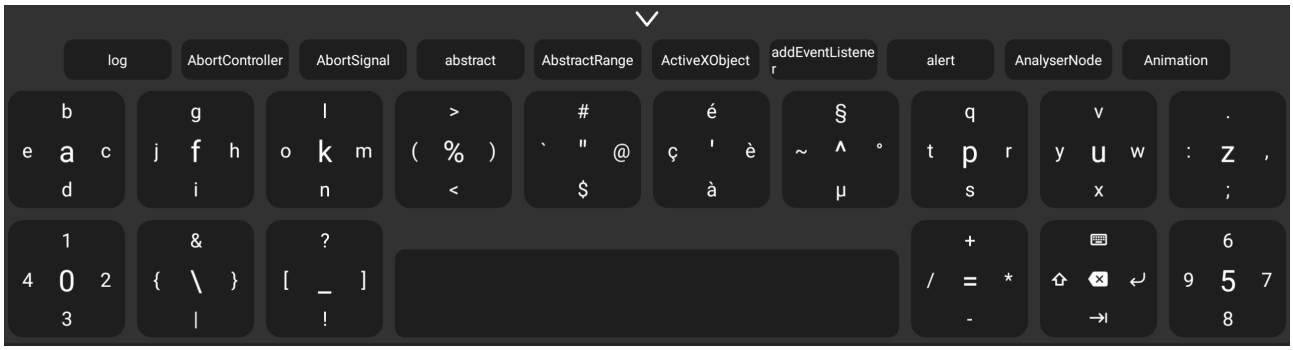


FIGURE 6.1 – Clavier adapté proposé dans l'application

6.2.1 Touches *TaS* du clavier

Certains résultats obtenus dans **[Romanoa et al., 2014]** ont pu être répliqués lors de l'expérience menée. En effet, la plupart des participants ont mentionné le fait que le clavier était difficile à prendre en main, mais qu'ils pensaient pouvoir l'utiliser de manière efficace avec le temps. De manière générale, les participants ne faisaient pas ou peu de fautes de frappe. Les principales erreurs venaient du fait qu'ils poussaient directement sur la lettre au lieu de swiper. Certains participants étaient déjà capables d'écrire aisément après une quinzaine de minutes d'utilisation du clavier.

Cette solution pour le clavier a ainsi été préférée à celle présentée dans **[Raab, 2016]** car le besoin d'appuyer longtemps sur une touche avant de pouvoir swiper sur un élément rendait l'interaction plus lente que si l'élément était déjà visible sur le clavier. La solution de **[Fennedy et al., 2022]** a aussi été laissée de côté, car elle permettait surtout de gérer les raccourcis clavier, pas encore mis en place dans l'application, mais qui pourraient l'être dans des travaux futurs.

6.2.2 Disposition des touches sur le clavier

Le clavier proposé a le grand avantage d'afficher tous les caractères spéciaux utiles lors de la programmation en permanence sur le clavier. Toutefois, la disposition des lettres de manière alphabétique a perturbé plus d'un utilisateur habitué à l'AZERTY du clavier physique français. Il serait tout à fait possible de garder un clavier AZERTY tout en conservant le principe inhérent aux touches en entourant chaque lettre de caractères spéciaux. Cette solution pourrait être adaptée à une utilisation sur smartphone, mais peut-être moins sur tablettes, car le clavier proposé a l'avantage d'avoir tous les caractères principaux (lettres, chiffres, symboles mathématiques, logiques, etc.) sur le bord du clavier, facilitant ainsi la prise de la tablette à deux mains pour écrire du code. Il serait intéressant de pouvoir évaluer les différences en termes de productivité et de PX des deux propositions tant sur smartphone que sur tablette.

6.2.3 Position de l'autocomplétion dans l'éditeur

L'autocomplétion présente sur le clavier était intuitive pour certains utilisateurs habitués des suggestions disposées au-dessus du clavier classiques des appareils mobiles. Par contre, certains utilisateurs moins habitués à utiliser ce genre de technologies auraient préféré que l'autocomplétion apparaisse dans le code comme dans d'autres éditeurs de codes. La solution de suggestions au-dessus du clavier semble néanmoins être la plus adaptée à cause de la taille de l'interface : mettre les suggestions sur le code impliquerait qu'un menu ou une icône cliquable faisant apparaitre un menu soit constamment présent sur la zone d'édition dont la taille est déjà assez réduite.

6.2.4 Position du curseur

Le positionnement du curseur n'est pas actuellement géré de manière personnalisée par l'éditeur de code. Des mécanismes de sélection comme une touche avec des flèches pour naviguer sur le clavier ou bien un mécanisme similaire à celui d'iPadOS permettant la navigation en se dirigeant sur la barre espace s'avèreraient sûrement plus adaptées pour gérer le positionnement du curseur dans le code.

6.3 Évaluation du mode de manipulation de code

Finalement, la manipulation de code faisait également partie intégrante de la problématique, comme en témoigne la dernière sous-question de recherche : **RQ1.3. Comment adapter les outils de manipulation de code sur tablette en utilisant les fonctionnalités proposées par le *Language Server Protocol* ?**

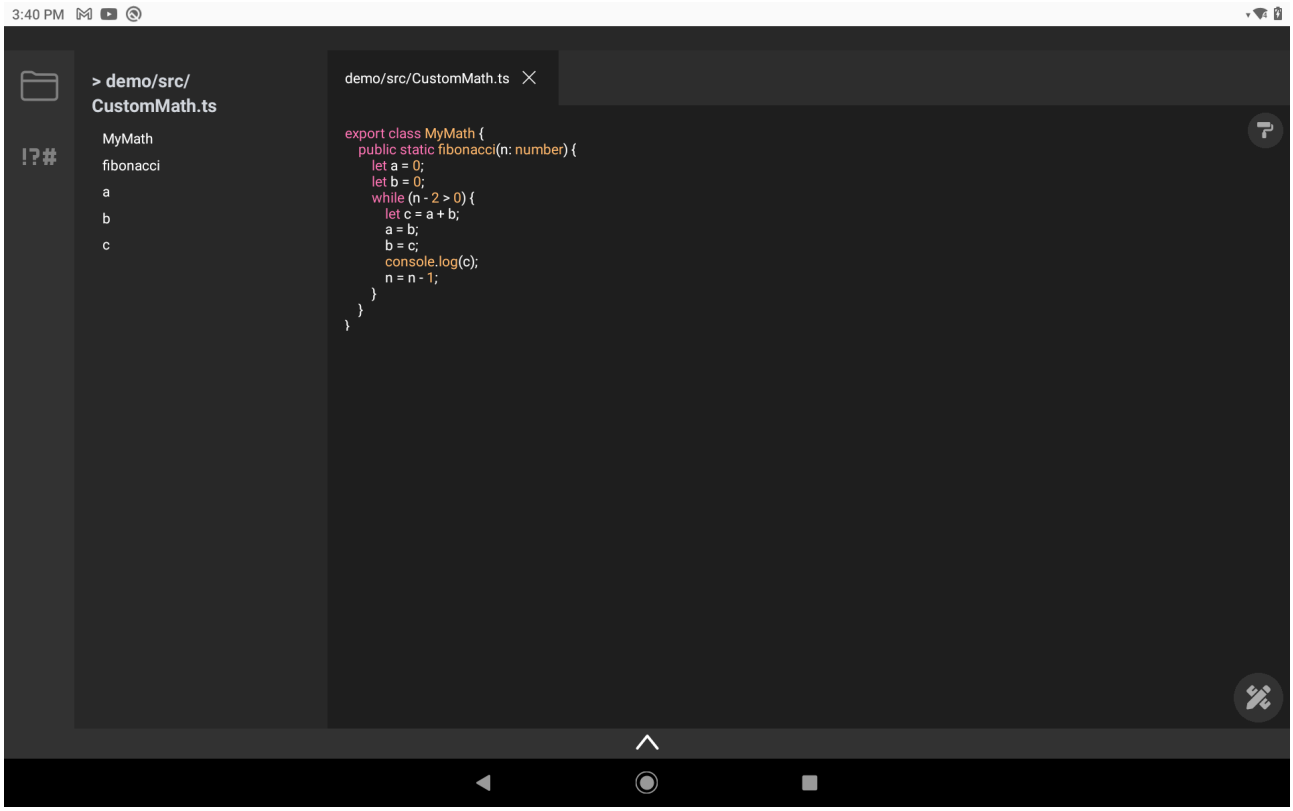


FIGURE 6.2 – Mode de manipulation de code

Le mode de manipulation de code a été mis en place de manière à reconnaître des gestes directement dans le code et pouvoir effectuer des actions via des boutons. Le mécanisme de reconnaissance de gestes a été mis en place, mais seulement un geste a pu être pour le moment implémenté et lié à une fonctionnalité du LSP.

Il aurait été intéressant de mettre plus de gestes en place, en se basant sur **[Raab et al., 2013]** et **[Biegel et al., 2014]**. Ainsi, le sous-ensemble des actions de refactoring présenté dans ces papiers qui pourrait trouver une équivalence dans le LSP pourrait être implémenté en utilisant le mécanisme de reconnaissance de gestes et de communication avec les serveurs de langage pour permettre une uniformité des actions de refactoring dans différents langages.

Interagir avec le code via des gestes présente cependant un inconvénient majeur. Si, par exemple, une interaction avec un symbole d'une seule lettre veut être faite, alors la précision des doigts ne sera pas toujours suffisante pour faciliter l'interaction. Ainsi, pendant l'expérience, il est souvent arrivé qu'un utilisateur ait à tenter plusieurs fois sa chance avant de pouvoir effectuer le geste voulu sur le token sémantique voulu. Une solution à ce problème serait, par exemple, de permettre d'effectuer certains gestes également dans l'explorateur de symbole (panneau situé sur la gauche dans la figure 6.2), utilisé actuellement pour la navigation entre les fichiers ouverts.

De plus, des mécanismes de découvrabilité des gestes devraient être mis en place pour permettre à l'utilisateur de prendre connaissance de la possibilité d'effectuer une action sur le code via un geste dans la zone d'édition.

Le bouton de formatage de code, bien que l'icône n'ait pas toujours été considérée comme adaptée,

est une fonctionnalité que la plupart des utilisateurs ont fortement appréciée, car elle permet facilement au programmeur de ne pas se soucier d'avoir un code parfaitement formaté à la main. D'autres fonctionnalités, non reprises dans les gestes, pourraient aussi être disponibles sous forme de bouton le long du bord droit de l'éditeur pour permettre d'intégrer encore plus d'action sur le code.

Chapitre 7

Travaux futurs

En ce qui concerne les travaux futurs, la recherche réalisée laisse place à un large horizon de sujets sur lesquels des recherches futures pourraient être menées. L'éditeur de code pourrait, en effet, être amélioré en y rajoutant d'autres fonctionnalités du LSP, des fonctionnalités d'assistance lors de l'écriture de code et des améliorations côté serveur permettant ainsi de faciliter son utilisation dans des contextes réels. La question d'un fonctionnement hors-ligne pourrait aussi être abordée ainsi que d'autres possibilités pour améliorer l'expérience d'utilisation des développeurs.

7.1 Utilisation d'autres fonctionnalités du LSP

7.1.1 Amélioration de la sélection de texte

La sélection de texte pourrait être améliorée par l'utilisation des serveurs de langage. En effet, la procédure *textDocument/selectionRange* pourrait être utilisée par l'application pour définir les zones de sélection par rapport à un point donné et ainsi pouvoir implémenter une version abstraite de la *syntax-aware code selection* présentée dans [Raab, 2016].

7.1.2 Affichage des erreurs

Lorsque le serveur de langage a analysé le contenu d'un fichier, il peut envoyer des notifications au client afin de lui signaler que des erreurs ont été trouvées dans le code fourni. Le client pourrait ainsi utiliser ces informations pour montrer au développeur les endroits problématiques en les mettant en évidence.

7.1.3 Coloration sémantique

Comme mentionné précédemment, la coloration sémantique [1] peut être ajoutée grâce à la procédure *textDocument/semanticTokens* des serveurs de langages. Le résultat de cette méthode pourrait être utilisé en combinaison avec la coloration syntaxique pour améliorer sa pertinence. Il est nécessaire de combiner les deux parce que la coloration sémantique seule n'est pas suffisante. En effet, elle ne permet pas, par exemple, de mettre en couleur les mots clés des langages. Par ailleurs, certains serveurs

1. `https://github.com/microsoft/vscode/wiki/Semantic-Highlighting-Overview`

de langage ne supportent pas la fonctionnalité *textDocument/semanticTokens*, et seule la coloration syntaxique serait disponible.

7.1.4 Intellisense

Les fonctionnalités d'autocomplétion sont, pour le moment, assez limitées. La précision de l'autocomplétion pourrait être améliorée via des statistiques sur l'utilisation des différents symboles au sein de la code base. Ainsi, lorsque l'utilisateur écrit "console.", la première méthode proposée devrait être "log" et non "assert" si "log" a une plus grande probabilité d'être utilisée.

7.1.5 Navigation dans le code

Actuellement, l'explorateur de symboles permet de naviguer un peu plus facilement entre les symboles des fichiers ouverts. Cependant, pour améliorer la navigation dans le workspace, les symboles affichés dans l'explorateur pourraient venir de l'entièreté du workspace. Il faudrait alors adapter l'affichage pour rendre la navigation dans la liste plus efficace.

Par ailleurs, l'explorateur de symboles pourrait être revu pour permettre le renommage de ceux-ci directement dans l'explorateur. Ainsi, une méthode pourrait être renommée sans même avoir à ouvrir un seul fichier.

7.2 Assistance lors de l'écriture de code

7.2.1 Ajout de feedbacks lors des interactions

La solution manque de feedbacks visuels pour indiquer à l'utilisateur l'action qui est faite lorsqu'il utilise certains gestes. C'est le cas du clavier, par exemple. Sur le clavier natif, lorsque l'utilisateur appuie sur une touche, celle-ci devient plus sombre afin de lui indiquer qu'elle a été pressée. Cependant, puisque le clavier proposé ne fonctionne pas uniquement en touchant la touche, mais en effectuant un geste dessus, le feedback classique n'est plus approprié. Il faudrait alors proposer une solution pour fournir un feedback lorsque la lettre du milieu est pressée et un autre pour indiquer quelle lettre est sélectionnée en fonction du geste commencé.

7.2.2 IA générative

L'intelligence artificielle est devenue un outil précieux pour de nombreux développeurs. En effet, depuis l'arrivée de GitHub Copilot et d'autres outils similaires, bon nombre de développeurs utilisent ces outils au quotidien pour les assister dans leurs tâches de programmation. Ces outils peuvent s'avérer utiles pour écrire automatiquement du code qui peut être répétitif comme les déclarations de classes, de méthodes, etc.

Dans le contexte de la programmation sur tablette, il semble important de limiter les interactions avec le clavier. Utiliser des outils de la sorte permettrait aux développeurs d'écrire moins de code et de pouvoir se concentrer sur la logique de celui-ci au lieu de se concentrer sur le clavier.

7.2.3 Smart typing

Il semble important d'ajouter des fonctionnalités de smart typing telles que l'indentation automatique ou encore la fermeture automatique des accolades ou des parenthèses afin de limiter le nombre d'interactions avec le clavier. L'ajout de la fermeture automatique des parenthèses nécessiterait l'intégration d'une fonctionnalité permettant de naviguer facilement dans le code comme les flèches directionnelles d'un clavier traditionnel. En fonction du contexte, l'éditeur pourrait aussi ajouter des parenthèses automatiquement lorsqu'une méthode/fonction est choisie par l'utilisateur dans l'autocomplétion.

7.2.4 Raccourcis claviers

Le clavier pourrait être modifié afin d'ajouter le support de raccourcis pour des actions comme le copier-coller, undo/redo, etc. Pour ce faire, il faudrait adapter le clavier SoftCuts de [Fennedy et al., 2022] à la nouvelle disposition de clavier.

7.3 Améliorations côté serveur

7.3.1 Gestion des utilisateurs et authentification

La solution logicielle actuelle est dépourvue d'un système d'authentification. Toute personne connaissant l'URL du serveur peut interagir avec. Une prochaine évolution de l'application serait l'ajout d'un système de comptes pour les utilisateurs afin de pouvoir uniquement avoir accès à leurs projets. La fonctionnalité pourrait ensuite évoluer pour permettre le partage de projets entre utilisateurs et la collaboration en temps réel.

7.3.2 Réduction de la taille de l'image Docker

Dans la solution actuelle, tous les serveurs de langage supportés sont installés à la création de l'image Docker. De ce fait, cette image pèse près de 4 Go pour seulement 5 langages supportés, ce qui laisse place à des optimisations.

Une alternative plus viable consisterait en l'installation des serveurs de langage uniquement quand il est nécessaire de les installer. On pourrait, par exemple, les installer à l'ouverture d'un workspace. Le serveur détecterait alors les différents langages de programmation utilisés dans les fichiers et installerait les serveurs de langage manquants en arrière-plan.

7.3.3 Exécution de code et débogage

L'exécution du code sur la tablette semble impossible étant donné les contraintes imposées par les systèmes d'exploitation. Cependant, il serait possible d'exécuter le code sur le même serveur que celui utilisé par les serveurs de langage. Évidemment, exécuter du code non fiable n'est pas sans risques. Cependant, ceux-ci peuvent être limités grâce, notamment, à la conteneurisation ou l'utilisation de machines virtuelles comme mentionné précédemment.

Une fois que l'exécution de code sera implémentée, il sera possible d'ajouter des fonctionnalités plus avancées telles que le débogueur.

7.3.4 Version control

La solution actuelle ne permet pas l'importation de projets et ne permet pas non plus d'exporter les projets créés. Les éditeurs de code actuels incluent souvent un outil de version control qui permet de cloner des dépôts, faire des commits, faire des merges, etc.

L'éditeur devrait offrir aux développeurs la possibilité de cloner un dépôt git en spécifiant l'URL de celui-ci. Il devrait aussi leur permettre de sélectionner les modifications qu'ils souhaitent ajouter dans leurs commits et donner à l'utilisateur d'encoder un message. Pour limiter l'utilisation du clavier, celui-ci pourrait être généré automatiquement. La collaboration à plusieurs implique qu'il y aura, par moments, des merges à faire. Il serait pertinent de s'intéresser à l'utilisation de gestes pour manipuler le code dans de tels contextes.

7.4 Fonctionnement hors connexion

Bien qu'il ne soit pas possible d'intégrer les serveurs de langage actuels directement dans l'application, d'autres pistes pourraient être explorées. Par exemple, il existe des projets permettant d'émuler Windows ou Linux depuis une application mobile. Il serait alors possible d'exécuter Docker dessus et cela permettrait ainsi d'utiliser le serveur en local. Cependant, ces projets dépendent généralement de failles de sécurité dans les systèmes d'exploitation mobiles ou nécessitent que ceux-ci soient débridés ("root" sur Android et "jailbreak" pour iPadOS).

7.5 Autres possibilités d'amélioration de l'expérience développeur

7.5.1 S'affranchir du clavier

Sur un ordinateur de bureau, tous les développeurs n'utilisent pas les mêmes types d'éditeurs. Certains préfèrent utiliser des éditeurs graphiques comme VSCode mais d'autres préfèrent utiliser des éditeurs comme Vim qui utilisent le clavier comme outil pour naviguer dans le code et interagir avec.

Ce mémoire a principalement étudié l'intégration du *Language Server Protocol* et l'adaptation du clavier pour accélérer la production de code sur tablette. Cependant, comme sur les ordinateurs classiques, il n'y a probablement pas qu'une façon d'écrire du code. Il serait peut-être intéressant d'explorer comment nous aurions pu améliorer l'écriture de code sans changer aussi radicalement le clavier utilisé.

Dans *TouchDevelop*, les chercheurs de Microsoft tentaient de s'affranchir au maximum du clavier, sauf pour les noms de symboles et les chaînes de caractère. Pour ce faire, ils avaient créé un langage de programmation spécifique au développement pour mobile. L'idée pourrait être adaptée en permettant la transformation du code produit vers d'autres langages. Ainsi, nous aurions un langage de haut niveau facile à manipuler sur mobile qui pourrait être transpilé vers des langages plus difficiles à manipuler sur tablette.

7.5.2 Navigation tactile

Des mécanismes de navigation semblables à ceux présentés dans [Henley and Fleming, 2014] pourraient également permettre d'améliorer la PX en exploitant la tactilité de l'interface. Des mécanismes des scaffolding semblables à ceux présentés dans [Mbogo et al., 2016] permettraient également de naviguer au sein d'un fichier plus rapidement sur une interface tactile à taille réduite.

Chapitre 8

Conclusion

Le développement sur mobile intéresse les développeurs et chercheurs depuis plusieurs années. Quelques projets ont déjà permis d'atteindre cet objectif, mais, bien souvent, l'expérience de développement ne permettait pas d'utiliser ces applications avec le niveau de productivité auquel on s'attend lorsque l'on a l'habitude de développer sur un ordinateur. D'autres applications, bien que productives, sont spécifiques à un langage de programmation en particulier, limitant ainsi les possibilités de diversifications des utilisations.

Pour proposer un éditeur de code sur appareils mobiles garantissant la productivité, nous avons mis en place un clavier spécialement conçu pour développer sur mobile. En affichant tous les caractères nécessaires à la programmation, le nombre d'actions nécessaires pour insérer un caractère spécial est considérablement réduit. De plus, l'utilisation du LSP pour ajouter le support de l'autocomplétion rend l'écriture de code encore plus simple. Évidemment, un temps d'adaptation est nécessaire pour pouvoir utiliser le clavier de manière productive.

En ajoutant le support pour plusieurs langages de programmation au sein de la même application, les usages de l'éditeur de code sur mobile changent. Précédemment, les éditeurs sur mobiles permettaient surtout d'écrire des scripts ou des programmes de petite taille. Intégrer des fonctionnalités comme le renommage de symboles dans le workspace ou le formatage rend l'outil plus proche des éditeurs de code dont nous avons l'habitude sur ordinateur. Sa capacité à supporter plusieurs langages répond aussi à une problématique moderne où un projet utilise bien souvent plusieurs langages de programmation en son sein.

L'outil présenté dans ce travail est encore au stade de prototype. Comme l'ont montré les résultats de l'expérience, le logiciel n'est pas parfait et manque de certaines fonctionnalités auxquelles les développeurs sont habitués. Les éditeurs de codes classiques, sur ordinateur, présentent, en effet, une panoplie de fonctionnalités auxquels les programmeurs se sont accoutumés. La perspective de pouvoir fournir des fonctionnalités équivalentes sur des appareils mobiles pourrait venir répondre à un besoin des développeurs, s'intégrant ainsi dans le contexte actuel d'utilisation des technologies.

Alan Kay, célèbre chercheur en informatique, affirme que *"La meilleure façon de prédire l'avenir est de l'inventer"*. Notre mémoire, rassemblant en un seul outil plusieurs années de recherche, pourrait ainsi servir de point de départ pour construire un éditeur aux fonctionnalités plus étoffées qui permettrait de satisfaire aux exigences propres à son contexte d'utilisation et favoriser son adoption.

Glossaire

API

Une API, ou interface de programmation d'application, est un ensemble de définitions et de protocoles qui facilite la création et l'intégration des applications [1]. 4, 8, 34–37, 39

AST

Abstract Syntax Tree. 19, 28

DSL

Domain Specific Language. 14

Flutter

Flutter est un kit de développement logiciel d'interface utilisateur open-source créé par Google. 36

JSON-RPC

JSON Remote Procedure Call. 27

LSP

Language Server Protocol. 6, 8, 13, 26–29, 31, 32, 36, 37, 43, 45, 57–59, 61, 65, 67

PX

Programmer Experience. 29, 30, 32, 58, 65

React Native

Framework d'applications mobiles open source créé par Facebook. 36

TaS

Tap and Slide Keyboard. 8, 15, 16, 41, 42

UEQ

User Experience Questionnaire. 9, 48–52

1. `https://www.redhat.com/fr/topics/api/what-are-application-programming-interfaces`

Bibliographie

Abbasi, A. Z., Shaikh, Z. A., and Pervez, S. A. (2010). Xylus : A virtualized programming environment. In *2010 International Conference on Information and Emerging Technologies*, pages 1–5.

Almusaly, I. and Metoyer, R. (2015). A syntax-directed keyboard extension for writing source code on touchscreen devices. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 195–202.

Andreas, H., Martin, S., and Jörg, T. (2018). Ueq user experience questionnaire. https ://www.ueq-online.org/. (Date de consultation : 10/04/2024).

Baćíková, M., Maricák, M., and Vancík, M. (2015). Usability of a domain-specific language for a gesture-driven ide. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 909–914.

Biegel, B., Hoffmann, J., Lipinski, A., and Diehl, S. (2014). U can touch this : touchifying an ide. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, page 8–15, New York, NY, USA. Association for Computing Machinery.

Bork, D. and Langer, P. (2023). Language server protocol : An introduction to the protocol, its use, and adoption for web modeling tools. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 18 :9–1.

Bünder, H. (2019). Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. In *International Conference on Model-Driven Engineering and Software Development*.

Costagliola, G., Fuccella, V., Leo, A., Lomasto, L., and Romano, S. (2018). The design and evaluation of a gestural keyboard for entering programming code on mobile devices. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 49–56.

Došilović, H. Z. and Mekterović, I. (2020). Robust and scalable online code execution system. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1627–1632, Opatija, Croatia. IEEE.

Feldman, Y. A., Gam, A., Tilkin, A., and Tyszberowicz, S. (2015). Deverywhere : Develop software everywhere. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 121–124.

Fennedy, K., Srivastava, A., Malacria, S., and Perrault, S. T. (2022). Towards a unified and efficient command selection mechanism for touch-based devices using soft keyboard hotkeys. *ACM Transactions on Computer-Human Interaction*.

Garcia, M. B., Benedic R. Enriquez, J., Adao, R. T., and Happonen, A. (2022). Hey ide, display hello world" : Integrating a voice coding approach in hands-on computer programming activities. In *2022 IEEE 14th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM)*, pages 1–6, Boracay Island, Philippines. IEEE.

Henley, A. Z. and Fleming, S. D. (2014). The patchworks code editor : toward faster navigation with less code arranging and fewer navigation mistakes. In *CHI '14 : Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 2511–2520, Toronto Ontario Canada. Association for Computing Machinery, New York, NY, United States.

Hesenius, M., Orozco Medina, C. D., and Herzberg, D. (2012). Touching factor : Software development on tablets. In Gschwind, T., De Paoli, F., Gruhn, V., and Book, M., editors, *Software Composition*, pages 148–161, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kamod, N. D. and Jadhav, R. N. (2023). A secure and scalable system for online code execution and evaluation using containerization and kubernetes. *Journal of Emerging Technologies and Innovative Research*, 10(2) :c151–c159.

Mares, M. and Blackham, B. (2012). A new contest sandbox. *Olympiads in Informatics*, 6 :100–109.

Mbogo, C., Blake, E., and Suleman, H. (2016). Design and use of static scaffolding techniques to support java programming on a mobile phone. In *ITiCSE '16 : Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 314–319, Arequipa Peru. Association for Computing Machinery, New York, NY, United States.

Microsoft (2024). Language server protocol. https ://microsoft.github.io/language-server-protocol/. (Date de consultation : 05/05/2024).

Morales, J., Rusu, C., Botella, F., and Quiñones, D. (2019). Programmer experience : A systematic literature review. *IEEE Access*, 7 :71079–71094.

Potluri, V., Vaithilingam, P., Iyengar, S., Vidya, Y., Swaminathan, M., and Srinivasa, G. (2018). Codetalk : Improving programming environment accessibility for visually impaired developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*, pages 1–11, Montreal QC Canada. Association for Computing Machinery, New York, NY, United States.

Raab, F. (2016). Source code interaction on touchscreens.

Raab, F., Wolff, C., and Echtler, F. (2013). Refactorpad : editing source code on touchscreens. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 223–228, New York, NY, USA. Association for Computing Machinery.

Romanoa, M., Paolinob, L., Tortorab, G., and Vitiello, G. (2014). The tap and slide keyboard : A new interaction method for mobile device text entry. *International Journal of Human–Computer Interaction*, 30(12) :935–945.

Rondier, M. (2004). A. bandura. auto-efficacité. le sentiment d'efficacité personnelle. *L'orientation scolaire et professionnelle*, (33/3) :475–476.

Schrepp, D. M. (Version 11 (12.09.2023)). User experience questionnaire handbook. https ://www.ueq-online.org/Material/Handbook.pdf. (Date de consultation : 10/05/2024).

Sukumar, P. T. and Metoyer, R. (2019). Mobile devices in programming contexts : A review of the design space and processes. In *DIS '19 : Proceedings of the 2019 on Designing Interactive Systems Conference*, page 1109–1122, San Diego CA USA. Association for Computing Machinery, New York, NY, United States.

Sukumar, P. T. and Metoyer, R. A. (2017). Design space of programming tools on mobile touchscreen devices. *ArXiv*, abs/1708.05805.

Tillmann, N., Moskal, M., de Halleux, J., and Fahndrich, M. (2011). Touchdevelop : programming cloud-connected mobile devices via touchscreen. In *Onward ! 2011 : Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60, Portland Oregon USA. Association for Computing Machinery, New York, NY, United States.

Tillmann, N., Moskal, M., de Halleux, J., Fahndrich, M., and Burckhardt, S. (2012). Touchdevelop : app development on mobile devices. In *FSE '12 : Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–2, Cary North Carolina. Association for Computing Machinery, New York, NY, United States.

Zayour, I. and Hajjdiab, H. (2013). How much integrated development environments (ides) improve productivity ? *JOURNAL OF SOFTWARE*, 8(10) :2425–2431.

Annexes

Annexe A

Statistiques d'utilisation des OS mobiles

A.1 *Stack Overflow Developer Survey*

Les statistiques de Stack Overflow dans le *Stack Overflow Developer Survey*[1].

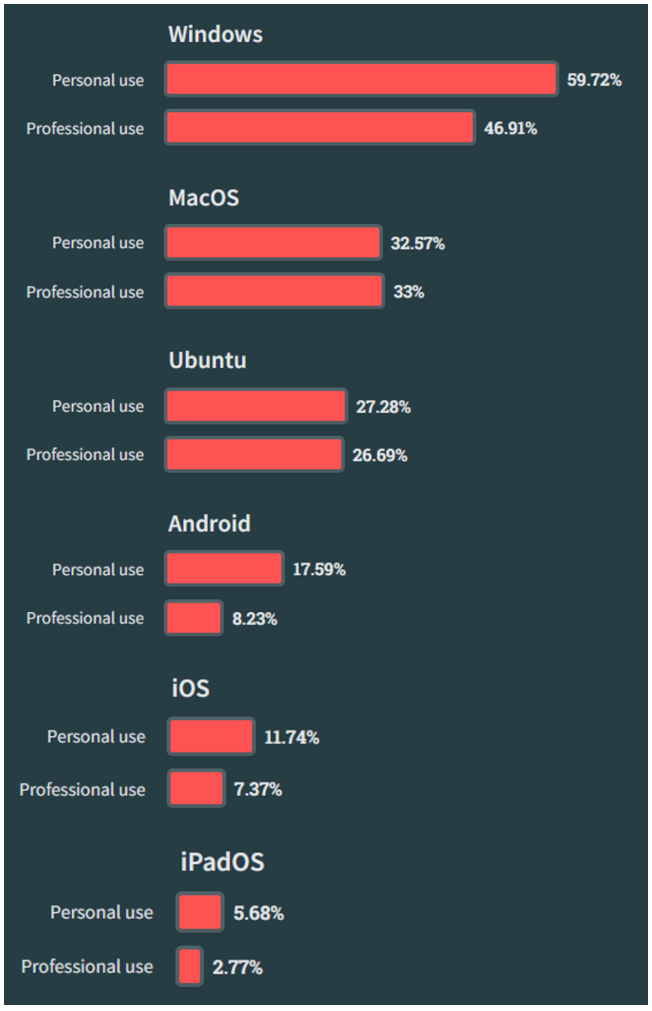


FIGURE A.1 – Statistiques d'utilisation des OS par les développeurs en 2023

1. https://survey.stackoverflow.co/2023/#section-most-popular-technologies-operating-system

A.2 *Octoverse : The state of open source and rise of AI in 2023*

Les données de GitHub dans leur enquête : *Octoverse : The state of open source and rise of AI in 2023*[2].

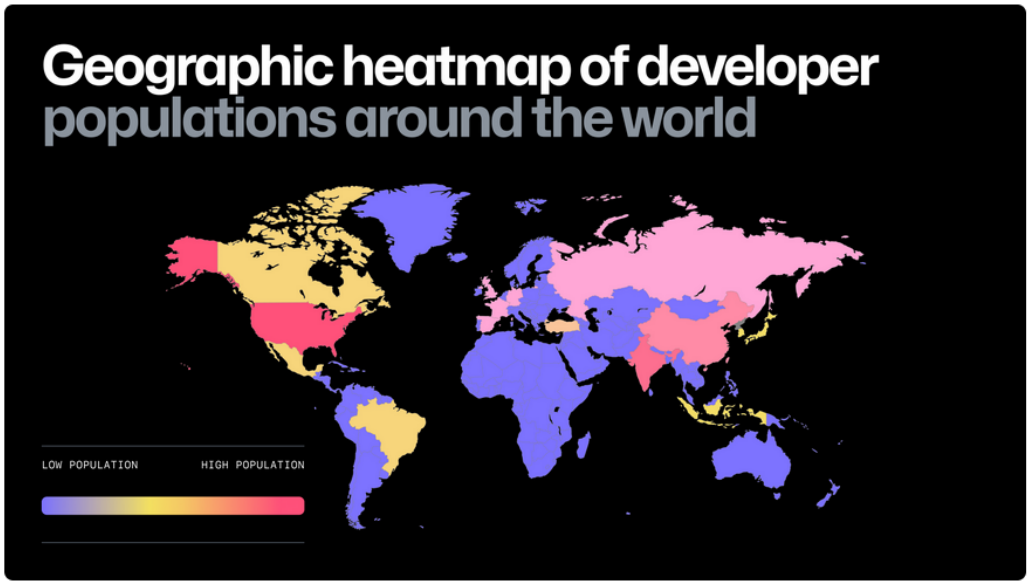


Figure A.2 – Heatmap de la population de développeur dans le monde en 2023

A.3 Statcounter GlobalStats

Les statistiques sur le marché d'OS en Afrique de Statcounter GlobalStats[3].

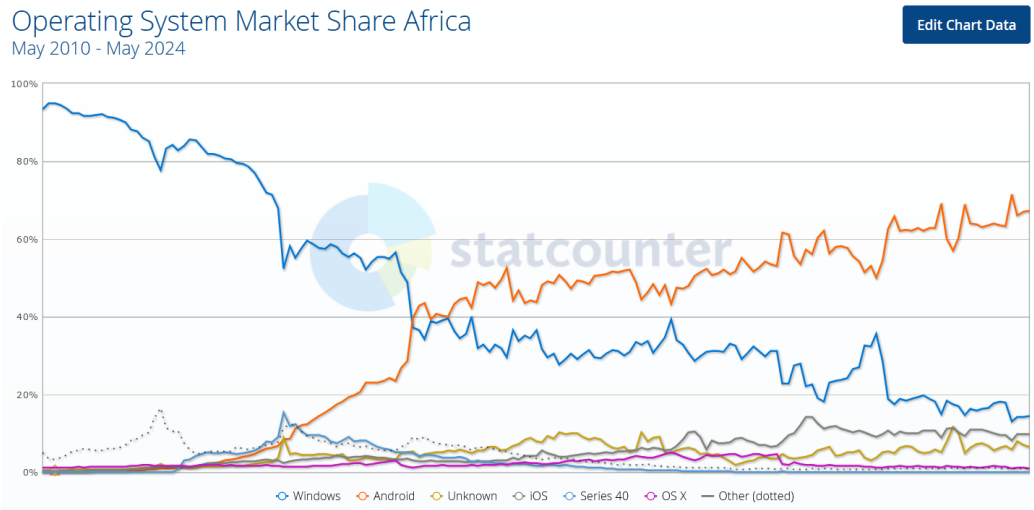


Figure A.3 – Marché des OS en Afrique entre 2010 et 2024

2. https://github.blog/2023-11-08-the-state-of-open-source-and-ai/
3. https://gs.statcounter.com/os-market-share/all/africa/#monthly-201006-202404

Annexe B

Évaluation : Résultats de l'UEQ

Items																								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
3	6	2	2	6	4	6	1	1	1	5	2	3	4	7	6	3	1	3	7	5	5	3	2	3	6
2	6	3	5	5	4	5	6	7	3	2	3	5	5	5	3	4	4	2	5	3	2	1	4	4	6
5	5	1	3	2	6	7	5	1	5	5	1	5	6	3	6	2	2	1	6	2	4	1	1	1	5
4	5	2	2	2	6	6	6	2	3	6	2	6	6	4	6	2	3	5	6	2	4	1	2	2	5
1	3	1	5	6	6	4	4	7	3	4	4	2	2	6	2	4	3	2	1	5	5	5	3	4	5
5	6	3	4	3	5	5	7	3	3	6	2	7	4	6	7	6	2	4	4	1	6	1	3	4	5
3	5	4	5	3	4	6	6	7	2	3	3	3	3	6	4	5	3	2	3	3	2	2	5	3	6
5	3	6	5	2	6	6	3	4	3	5	3	6	7	5	4	2	1	1	6	5	6	2	1	1	4
4	6	2	6	2	5	6	6	1	2	3	2	3	5	6	4	4	5	2	6	4	4	2	2	4	6
5	6	3	6	2	4	5	6	1	2	2	4	6	5	6	5	4	6	4	3	5	6	2	4	4	6

FIGURE B.1 – Résultats des expériences

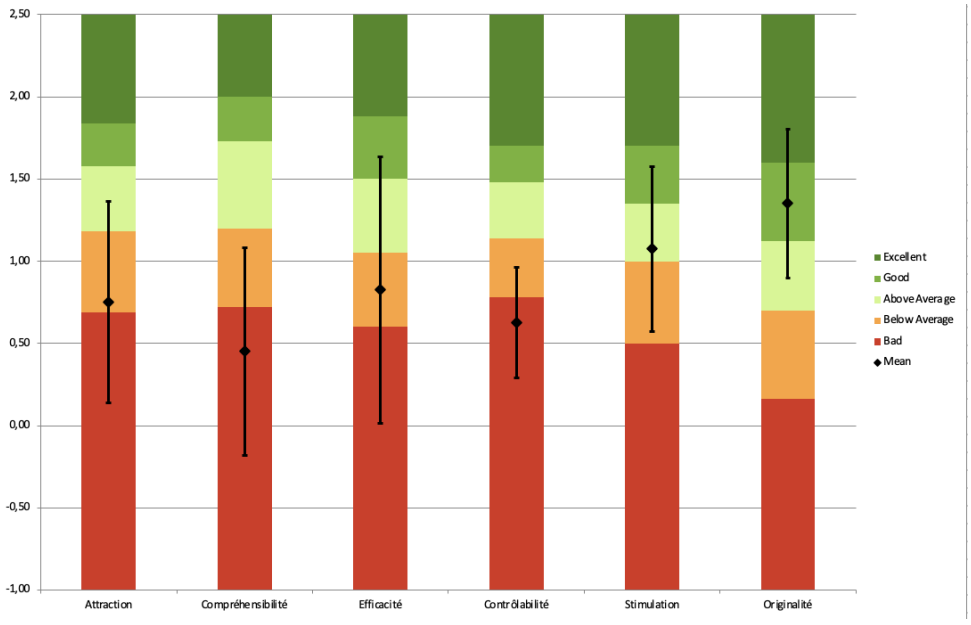


FIGURE B.2 – Résultats du benchmark

Item	Mean	Variance	Std. Dev.	No.	Left	Right	Scale
1	-0,3	2,0	1,4	10	Agaçant	Agréable	Attraction
2	1,1	1,4	1,2	10	Incompréhensible	Compréhensible	Compréhensibilité
3	1,3	2,2	1,5	10	Moderne	Sans fantaisie	Originalité
4	-0,3	2,2	1,5	10	Appropriation simple	Appropriation compliquée	Compréhensibilité
5	0,7	2,9	1,7	10	Apporte de la valeur	Peu de valeur ajoutée	Stimulation
6	1,0	0,9	0,9	10	Ennuyeux	Captivant	Stimulation
7	1,6	0,7	0,8	10	Inintéressant	Intéressant	Stimulation
8	1,0	3,3	1,8	10	Imprévisible	Prévisible	Contrôlabilité
9	0,6	7,2	2,7	10	Rapide	Lent	Efficacité
10	1,3	1,1	1,1	10	Original	Conventionnel	Originalité
11	0,1	2,3	1,5	10	Rigide	Facilitant	Contrôlabilité
12	1,4	0,9	1,0	10	Bien	Médiocre	Attraction
13	0,6	2,9	1,7	10	Compliqué	Simple	Compréhensibilité
14	0,7	2,2	1,5	10	Repoussant	Attractif	Attraction
15	1,4	1,4	1,2	10	Habituel	Avant-gardiste	Originalité
16	0,7	2,5	1,6	10	Désagréable	Agréable	Attraction
17	0,5	2,3	1,5	10	Sécurisant	Insécurisant	Contrôlabilité
18	1,0	2,7	1,6	10	Stimulant	Soporifique	Stimulation
19	0,9	2,5	1,6	10	Répond aux attentes	Ne répond pas aux attentes	Contrôlabilité
20	0,6	3,4	1,8	10	Inefficace	Efficace	Efficacité
21	0,4	2,7	1,6	10	Clair	Déroutant	Compréhensibilité
22	0,0	2,4	1,6	10	Non pragmatique	Pragmatique	Efficacité
23	2,1	1,7	1,3	10	Sobre	Surchargé	Efficacité
24	1,3	1,8	1,3	10	Attrayant	Rébarbatif	Attraction
25	0,7	1,1	1,1	10	Sympathique	Inamical	Attraction
26	1,4	0,5	0,7	10	Conservateur	Innovant	Originalité

Figure B.3 – Moyenne, variance et écart-type pour chaque item de l'UEQ

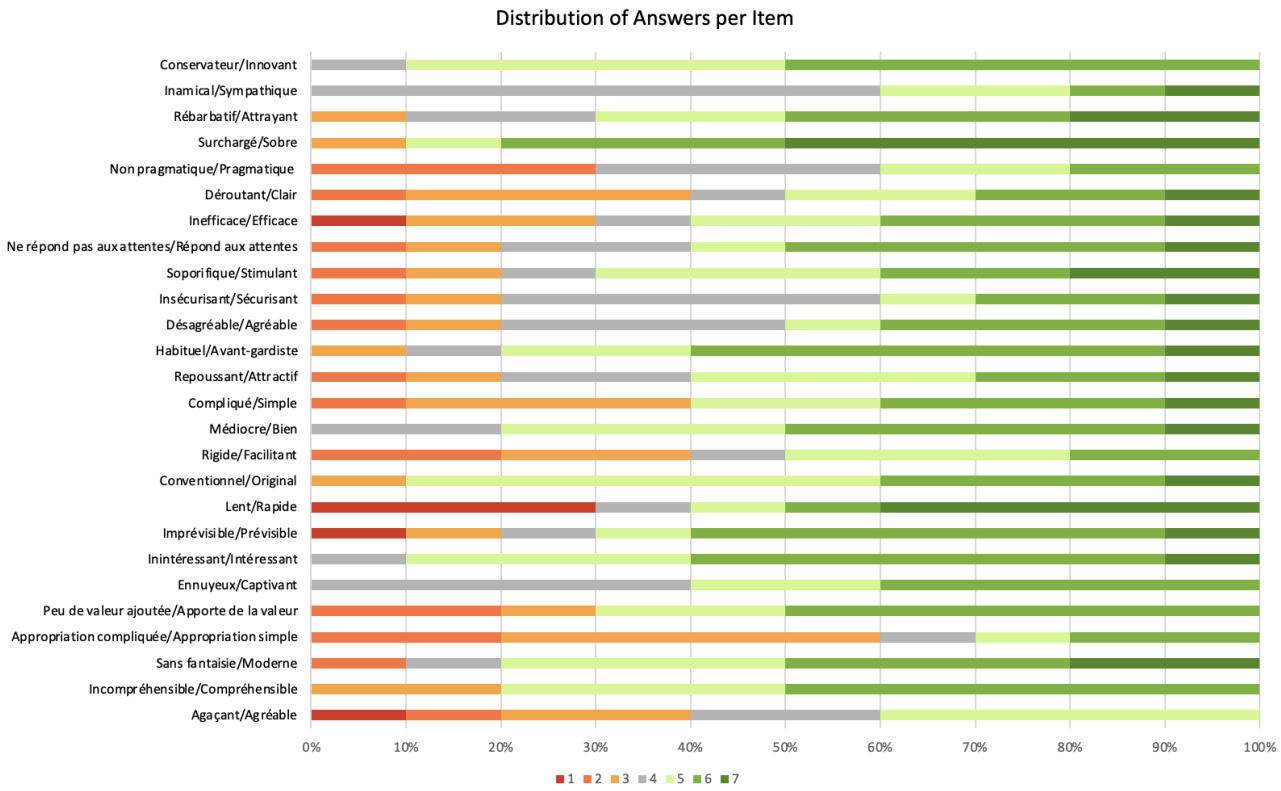


Figure B.4 – Distribution des réponses par item

Annexe C

Évaluation : notes et réponses aux questions

Sujet d'expérience n°1	
Commentaires/notes durant l'interaction • Difficile de découvrir par soi-même qu'il faut appuyer longtemps sur le répertoire parent pour créer un fichier. • Le clavier natif recouvre en partie la modal de renommage. • N'utilise pas naturellement l'autocomplétion • N'utilise pas les onglets • "Le clavier rend l'interaction lente, mais sans doute qu'avec l'habitude, on irait plus vite".	
Le plus appréciable ✓ Coder sur tablette ✓ Support multi-langage	**Le moins appréciable** ✗ Difficulté d'utilisation du clavier

Table C.1 – Compte-rendu de l'expérience du sujet n°1

Sujet d'expérience n°2	
Commentaires/notes durant l'interaction • La création d'un fichier est intuitive • Le sujet slide intuitivement sur le clavier • N'utilise pas naturellement l'autocomplétion • Le sujet est perturbé par la position des touches sur le clavier • "L'indentation ne se fait pas automatiquement quand j'appuie sur *Enter*" • Bizarre que le clavier ne soit pas le même tout le temps (clavier natif dans les modal) • "Je ne coderais de toute façon pas sur tablette"	
Le plus appréciable ✓ Accès aux caractères spéciaux comme les accolades aisé ✓ Renommage de token dans plusieurs fichiers ✓ Pas de latence	**Le moins appréciable** ✗ Pas habitué à utiliser un clavier où il faut faire des mouvements ✗ Position des accolades pas adaptée sur le clavier ✗ Pas de formatage en temps réel ✗ Pas possible de renommer le nom du fichier tout en renommant l'import ✗ L'autocomplétion ne fournit pas de structure plus précise que simplement du texte

Table C.2 – Compte-rendu de l'expérience du sujet n°2

Sujet d'expérience n°3	
Commentaires/notes durant l'interaction • Pas de smart typing • Interagis plutôt facilement avec l'interface • Autocomplétion assez irrégulière	
Le plus appréciable ✓ Agréable à utiliser ✓ La gestion des imports depuis d'autres fichiers	**Le moins appréciable** ✗ Clavier inhabituel ✗ Le mode de manipulation de code est bizarre et pas assez fourni ✗ La nature de la fonction de certains boutons nécessiterait un tutoriel

Table C.3 – Compte-rendu de l'expérience du sujet n°3

Sujet d'expérience n°4	
Commentaires/notes durant l'interaction • Utilise directement bien le clavier • "C'est perturbant que ce ne soit pas AZERTY" • "L'application pourrait proposer des symboles quand l'autocomplétion ne renvoie pas de résultats pertinents" • Aurait aimé que ça puisse s'exécuter	
Le plus appréciable ✓ Expérience intéressante	**Le moins appréciable** ✗ Très peu d'information sur ce qui peut être fait ✗ Pas de smart typing

Table C.4 – Compte-rendu de l'expérience du sujet n°4

Sujet d'expérience n°5	
Commentaires/notes durant l'interaction • Pas emballé par le design du clavier • Création de fichier intuitive • Difficile de trouver les lettres parfois • Sélection au sein du fichier parfois difficile à gérer • "Il aurait été intéressant d'ajouter dès flèches permettant de naviguer dans le code" • Les propositions sont parfois très bizarres • Le clavier n'est pas idéal pour la productivité (lenteur) • Bonne rapidité et réactivité de l'interface • Clavier assez petit finalement • "C'est difficile de rivaliser avec le clavier physique"	
Le plus appréciable	**Le moins appréciable**
✓ Rien de choquant ✓ Pas dépaysant, cela ressemble à un IDE traditionnel	✗ Clavier fort chargé et utilisation demandant apprentissage ✗ Autocomplétion avec propositions bizarres

Table C.5 – Compte-rendu de l'expérience du sujet n°5

Sujet d'expérience n°6	
Commentaires/notes durant l'interaction • "Dommage qu'il n'y ait pas de flèche pour gérer la sélection" • "J'aime bien l'idée du clavier, mais il faut s'y habituer" • Positionnement de l'autocomplétion pas le plus adapté • Gestes pour renommage pas intuitif • Il manque la gestion des erreurs	
Le plus appréciable ✓ Autocomplétion ✓ La position des touches des lettres est bonne pour permettre l'écriture rapide	**Le moins appréciable** ✗ Bouton sur l'arborescence de fichiers un peu petit ✗ Autocomplétion parfois boguée ✗ Manque de gestion des erreurs ✗ Mode manipulation assez pauvre ✗ Manque coloration sémantique

Table C.6 – Compte-rendu de l'expérience du sujet n°6

Sujet d'expérience n°7	
Commentaires/notes durant l'interaction • "Il faut des doigts de fée pour utiliser le logiciel" (sélection) • Aurait préféré que l'autocomplétion apparaisse dans le code • Symboles dans l'autocomplétion serait pertinent • Pas d'autocomplétion sur la langue d'écriture • Pas de smart-typing	
Le plus appréciable ✓ Formatage de code ✓ Interface épurée ✓ Navigation instinctive	**Le moins appréciable** ✗ Productivité impactée ✗ Autocomplétion sur le clavier et pas dans le code

Table C.7 – Compte-rendu de l'expérience du sujet n°7

Sujet d'expérience n°8

Commentaires/notes durant l'interaction

- Aimerait pouvoir zoomer sur l'arborescence de fichiers
- Pas l'habitude que ce ne soit pas AZERTY
- Autocomplétion parfois boguée
- Pas de smart-typing
- Perte de la sélection avec le changement de clavier
- "L'autocomplétion propose les variables, c'est cool"
- Aimerait que les propositions de l'autocomplétion soient filtrés par fréquence d'utilisation
- Les groupements de caractères spéciaux sont sympas (opérateurs logique, ensemblistes...)
- Aurait aimé pouvoir naviguer dans le code avec le clavier (clavier apple barre espace)
- Pas même interaction pour renommage fichier et renommage variable
- Pas toujours le même clavier
- Pas de *save onclose* sur la modal
- Petit symbole rend les cliques parfois difficiles
- Pas habituer à un clavier alphabétique
- Curieux de voir ce que ça peut donner sur un gros projet
- Manque Copilot/Chat GPT

Le plus appréciable	**Le moins appréciable**
✓ Renommage modifie toutes les références	✗ Pas d'autoindenation
✓ Positionnement des symboles sur le clavier	✗ pas de smart-typing
✓ Autocomplétion	✗ Incohérence pour le renommage
	✗ Manque un tri des propositions
	✗ Manque coloration sémantique
	✗ Manque de pouvoir exécuter le code

Table C.8 – Compte-rendu de l'expérience du sujet n°8

Sujet d'expérience n°9	
Commentaires/notes durant l'interaction • Trouve comment créer un fichier plutôt aisément • Utilise directement le clavier correctement • "Selon moi, au premier abord, c'est compliqué de prendre en main le clavier, mais sûrement qu'au fur à mesure, ce sera plus simple" • Utilise directement l'autocomplétion • Manque tabulation automatique • "Je vais déjà plus vite pour me repérer avec les lettres" • Aurait aimé que ça propose les autocomplétions par probabilité d'apparition • N'utilise pas toujours instinctivement l'autocomplétion • Interaction avec les touches "joystick" marche très bien (jamais d'erreur) • Pas de gestion des erreurs • Modal pas bien alignée par rapport au clavier • Agréable, car nouveau, mais prise en main difficile avec le clavier • "Ce serait cool d'avoir un éditeur de code sur tablette pour pouvoir coder un peu quand on le veut" • Suggestions sont cool, mais rigide à cause de la tabulation et manque de précision des autocomplétions • Manque des informations sur comment utiliser l'application • La coloration syntaxique manque parfois de différence entre les natures des token	
Le plus appréciable ✓ Le clavier est très efficace et l'idée est vraiment bonne	**Le moins appréciable** ✗ Manque la tabulation automatique ✗ Autocomplétion ne propose pas toujours la bonne complétion

Table C.9 – Compte-rendu de l'expérience du sujet n°9

Sujet d'expérience n°10	
Commentaires/notes durant l'interaction • Ne trouve pas où cliquer pour écrire facilement • Utilise le clavier bien directement • Utilise l'autocomplétion • Pas d'autoindentation • Aurait aimé des flèches dans le clavier pour la sélection • "Aurait préféré avoir un bouton retour à la ligne séparé" • Inconstance des propositions d'autocomplétion • Plus facile d'interagir avec 2 doigts • Difficile de deviner les gestes par soi-même pour la manipulation • Compliqué d'interagir avec le code en manipulation quand il est petit • Plutôt agréable pour du mobile • Clavier différent quand on renomme, c'est perturbant • Icône de formatage pas hyper adaptée • Aurait préféré de juste devoir appuyer pour avoir un menu pour les gestes • Manque la coloration sémantique pour avoir des couleurs plus spécialisées	
Le plus appréciable	**Le moins appréciable**
✓ Le clavier est assez sympa, "je me vois le prendre en main"	✗ Difficile de découvrir comment réaliser les actions.

Table C.10 – Compte-rendu de l'expérience du sujet n°10