UNIVERSITÉ DU QUÉBEC À MONTRÉAL

APPLICATIONS MOBILES PORTABLES ET DE HAUTE QUALITÉ : DU PROTOTYPE À LA LIGNE DE PRODUITS PAR LE RAFFINEMENT DE CLASSES ET LA PROGRAMMATION POLYGLOTTE

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

ALEXIS LAFERRIÈRE

MARS 2018

REMERCIEMENTS

Avant de commencer, j'aimerais remercier tous ceux qui m'ont appuyé durant mon doctorat. D'abord ma famille, Arabelle pour son amour et sa confiance, Christiane et Jacques pour leur support et leurs encouragements continus.

Je désire remercier particulièrement Jean Privat, mon directeur de recherche et professeur au département d'informatique de l'UQAM, pour sa logique contagieuse, la création du langage Nit et sa machine à café. Je me dois aussi de remercier Lucas Bajolet, diplômé de la maîtrise, pour ses chaînes de caractères, les débats sur `CString` et ses talents de barista.

Ce projet aurait été beaucoup plus long et définitivement plus solitaire sans l'aide de stagiaires. Merci à Matthieu Lucas, pour avoir été le premier cobaye de la FFI de Nit et avoir pointé le besoin en patrons de conception polyglotte. Romain Chanoir, maintenant étudiant à la maîtrise, pour sa contribution au support de la JVM, les premiers services pour Android et nos séances de travail traditionnelles. Frédéric Vachon, maintenant une personne responsable, pour la persistance des données en Android et la création de *jwrapper*. Mehdi Ait Younes et Arthur Delamare pour leur contribution à *jwrapper* et la création de *objcwrapper*.

Un merci spécial à Jean-Philippe Caissy et à Guillaume Auger pour s'être aventurés avec la FFI de Nit lorsqu'elle était expérimentale, ainsi que pour avoir posé les premières briques de *nitcorn* et de la sérialisation de Nit. Il en est de même pour Philippe Pépos-Petitclerc, maintenant étudiant à la maîtrise, pour l'expérimentation de la FFI avec C++.

AVANT-PROPOS

L'inspiration de ce projet de recherche est née lors de la visite d'une *startup* Montréalaise où deux équipes réalisaient la même application séparément pour Android et pour iOS. À ce moment, nous avions déjà une ébauche de solution pour réaliser des applications portables à plusieurs plateformes, et nous y avons vu une occasion pour l'étendre afin de couvrir davantage les besoins des applications mobiles.

Cette thèse propose une nouvelle solution complète pour réaliser des applications de haute qualité qui sont portables à Android et à iOS tout en étant adaptées à chaque plateforme. Notre solution profite de fonctionnalités du langage Nit, mais les concepts fondamentaux peuvent être appliqués à d'autres langages. Ce document sert également de manuel d'utilisation pour notre solution en général et pour l'interface de fonctions étrangères de Nit.

Pour évaluer notre proposition, nous avons réalisé quatre API et leurs implémentations pour Android et iOS. Nous avons étudié la réalisation de lignes de produits en Nit à l'aide du raffinement de classes. Nous avons conçu et implémenté l'interface de fonctions étrangères de Nit avec le support de C, Objective-C et Java. Et finalement, nous avons réalisé trois applications pour mettre à l'épreuve tous les aspects de notre solution.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ACRONYMES

AOSP *Android Open Source Project*, projet libre Android

API *Application Programming Interface*, interface de programmation

FFI *Foreign Function Interface*, interface de fonctions étrangères

GPS *Global Positionning System*, système mondial de positionnement

IDL *Interface Description Language*, langage de description d'interface

J2ME *Java 2 Micro Edition*, Java 2 version micro

JNI *Java Native Interface*, interface native de Java

JSON *JavaScript Object Notation*, notation objet JavaScript

JVM *Java Virtual Machine*, machine virtuelle Java

NDK *Native Development Kit*, trousse de développement natif

NFC *Near Field Communication*, communication en champ proche

SDK *Software Development Kit*, trousse de développement

UI *User Interface*, interface utilisateur

URL *Uniform Resource Locator*, localisateur uniforme de ressource

XAML *Extensible Application Markup Language*, langage de balisage extensible pour applications

XML *Extensible Markup Language*, langage de balisage extensible

RÉSUMÉ

Les appareils mobiles sont une plateforme populaire pour le développement de nouvelles applications. Toutefois, le développement d'applications mobiles est entravé par la division du marché entre Android et iOS, ainsi que par la fragmentation à l'intérieur même des plateformes qui empêchent une même application de fonctionner sur tous les appareils mobiles.

L'industrie et le milieu scientifique proposent quelques solutions pour réaliser des applications portables. Par contre, aucune ne donne un accès entier, simple et sûr aux fonctionnalités propres à chaque plateforme.

Cette thèse introduit une nouvelle solution pour réaliser des applications portables et pour les adapter à Android et à iOS. Notre solution combine trois approches. (i) Des API portables servent à réaliser un prototype rapidement. (ii) Une organisation en ligne de produits, réalisée par le raffinement de classes, permet une évolution graduelle du prototype en deux applications adaptées à Android et à iOS. (iii) La programmation polyglotte, via l'interface de fonctions étrangères (FFI) de Nit, donne accès aux API natives dans leur entièreté et leur langage natif.

Pour mettre sur pied notre solution, nous avons bonifié le langage Nit par l'ajout de fonctionnalités et de services. D'abord, nous avons conçu la FFI de Nit et implémenté le support des langages C, Objective-C et Java. Ensuite, nous avons étendu le compilateur Nit pour qu'il génère les applications Android et iOS. Puis, nous avons conçu et réalisé quatre API dédiées aux applications mobiles : interface utilisateur, cycle de vie, requêtes HTTP et persistance des données.

Finalement, nous avons validé notre solution en réalisant trois projets d'applications mobiles. Deux petites applications, une calculatrice et un client de microblogue, nous ont permis d'évaluer les API portables, des variations simples dans de petites lignes de produits, et l'intégration de code étranger avec la FFI de Nit. Un projet d'application réelle, la ligne de produits Tenenit, a mis à l'épreuve tous les aspects de notre solution. De plus, au travers de ces travaux, nous avons identifié et défini trois patrons de conception polyglotte.

INTRODUCTION

Les appareils mobiles, tels que les téléphones intelligents et les tablettes, sont des plateformes de choix pour les applications en tout genre. L'utilisateur moyen passe maintenant plus de temps sur les appareils mobiles que sur les ordinateurs de bureau, surtout pour consulter les réseaux sociaux et accéder à des informations lorsqu'en déplacement (comScore, 2015). Certaines entreprises se contentent même de ne viser que les appareils mobiles ; un exemple notable est le populaire WhatsApp, lancé en 2009 sur iOS, en 2010 sur Android et en 2016 seulement sur ordinateurs de bureau.

L'avantage des applications pour appareils mobiles est la connectivité et la disponibilité de l'appareil qui accompagne presque toujours son utilisateur. Par contre, elles sont aussi soumises à des restrictions particulières, elles doivent être économes en énergie et elles sont limitées à de petits écrans tactiles. De plus, une grande embûche pour les développeurs est la fragmentation du marché des appareils mobiles qui est divisé entre deux principaux systèmes d'exploitation : Android de Google et iOS d'Apple.[1] Ces deux systèmes d'exploitation sont incompatibles, une application native à Android ne fonctionne pas sous iOS et vice versa.

Plusieurs solutions surmontent le problème de fragmentation en permettant aux développeurs de réaliser des applications entièrement portables, toutefois, celles-ci sont limitées aux fonctionnalités communes à Android et à iOS. Plus récemment, de nouvelles solutions ouvrent l'accès aux fonctionnalités propres à chaque plateforme. Par contre, ces solutions ne profitent pas des avancées récentes favorisant le passage entre différents langages de programmation, l'accès aux fonctionnalités natives est donc limité, difficile et peu sûr. Tout de même, il y a un intérêt palpable, plusieurs grands joueurs de l'industrie

1. Le système d'exploitation iOS de Apple est utilisé sur tous les iPhone, iPad et iPod.

s'attaquent au problème dont Microsoft, Google, Facebook et Dropbox. Notamment, Microsoft a acquis Xamarin, une des solutions dominantes du marché, pour un montant de 400 M$US en 2016 (Greene, 2016).

Contributions scientifiques

Cette thèse apporte une contribution scientifique centrale, une solution complète pour le développement d'applications mobiles de haute qualité pour Android et iOS. Notre solution, nommée *app.nit*, combine trois approches :

- Des API portables offrent des services communs à Android et à iOS pour réaliser rapidement un prototype portable, une première version de l'application fonctionnelle sur les deux plateformes.
- L'organisation en ligne de produits permet de réaliser une famille d'applications semblables, dont des applications adaptées à Android et à iOS. Cette organisation est réalisée par le raffinement de classes de Nit qui sépare proprement les préoccupations dans le code. Les lignes de produits surmontent la fragmentation entre les plateformes en permettant d'adapter graduellement le prototype à chaque plateforme.
- La programmation polyglotte favorise l'utilisation de plusieurs langages selon leurs spécialités pour réaliser une même application. Elle est rendue possible par l'interface de fonctions étrangères (FFI) de Nit, qui donne accès aux fonctionnalités propres à Android et à iOS via les API natives, et ce, dans leur langage natif : Objective-C et Java.

De plus, cette thèse apporte quatre autres contributions scientifiques complémentaires :

- L'analyse du raffinement de classes pour réaliser les artéfacts de code des lignes de produits offre un nouveau point de vue sur le raffinement de classes de Nit et un nouveau langage pour en décrire l'application.
- La spécification de la FFI de Nit qui innove sur plusieurs points, elle s'intègre au paradigme à objets, supporte plusieurs langages étrangers à objets et génère sur mesure des services pour le code étranger de façon à préserver leur sûreté statique.

— La définition de trois patrons de conception pour résoudre des problèmes de programmation polyglotte : structurer les échanges de données entre les langages, reproduire une API étrangère à objets en Nit, et spécialiser une classe étrangère depuis le code Nit.
— L'étude de cas du projet Tenenit explore l'utilisation pratique de notre solution dans une application mobile complexe. L'application Tenenit profite des différents aspects de notre solution, depuis le prototype entièrement portable à des applications adaptées aux plateformes et à différents commerces.

Démarche et contributions techniques

Nous avons débuté ce projet en observant le processus de développement d'applications mobiles pour en déterminer les besoins réels. Ensuite, nous avons formulé notre solution composée des trois approches. Pour la mettre à l'épreuve, nous avons étendu le langage Nit, ses outils et sa bibliothèque, et nous avons réalisé trois applications mobiles pour valider notre solution.

Notre effort se divise en cinq grandes contributions techniques ;

— La conception de la FFI de Nit ainsi que son implémentation entière dans le compilateur et partielle dans l'interpréteur. Ce travail inclut la modification de la grammaire Nit pour qu'elle accepte le code étranger imbriqué parmi le code Nit, ainsi que le support des langages C, Objective-C et Java.
— Le développement de la bibliothèque *app.nit* qui offre, sous forme de quatre API, des services spécialisés pour les applications mobiles. En plus de l'implémentation de tous les services séparément pour Android et pour iOS, nous avons révisé la bibliothèque standard de Nit pour en assurer la portabilité aux deux plateformes.
— La modification du compilateur Nit pour qu'il produise des applications Android et iOS. Pour ce faire, nous avons ajouté des modules spécialisés pour chaque plateforme qui génèrent des projets d'applications natives comme attendu par les suites d'outils officielles d'Android et d'iOS.

— Nous avons réalisé trois applications mobiles portables à Android et iOS en Nit. Deux petites applications valident des parties spécifiques de notre solution et une application de taille réelle nous a servi à valider notre solution en entier.
— En parallèle à ce projet de thèse, nous avons développé plusieurs outils et bibliothèques pour le langage Nit. Notamment, les services de sérialisation intégrés dans le compilateur Nit et deux moteurs : JSON et binaire. La reproduction de la bibliothèque GTK+ en Nit qui permet de réaliser des applications graphiques sur ordinateurs de bureau. L'implémentation des *threads* POSIX, le modèle d'exécution parallèle, en Nit. Et même un engin de jeu, *gamnit*, qui vise la portabilité du code et l'accès direct aux services d'OpenGL ES.

Contexte de recherche

Cette thèse a été réalisée dans le cadre d'un projet de recherche interuniversitaire articulé autour du langage Nit. Le projet est une collaboration entre le groupe de recherche GRÉSIL de l'UQAM, et le LIRMM de l'Université de Montpellier.

Nit est un langage de programmation à objets servant à expérimenter avec de nouvelles fonctionnalités de langage et techniques de compilation. Il conserve plusieurs fonctionnalités expérimentales qui se sont avérées avantageuses : le raffinement de classes (Ducournau, Morandat et Privat, 2008) ; les types nullables (Gélinas, J.S., Gagnon, E. et Privat, J., 2009) ; le typage covariant (Terrasa et Privat, 2013) ; des chaines de caractères performantes (Bajolet, 2016) ; et une interface native statique (Laferrière, 2012). Le langage Nit supporte également la spécialisation multiple, le typage adaptatif et des constructeurs intelligents. [2]

Le langage Nit est central à ce projet de recherche. Nous réalisons les lignes de produits par le raffinement de classes, une fonctionnalité propre à Nit. De plus, nous avons étendu le langage en y ajoutant une FFI originale et nous avons modifié le compilateur pour qu'il produise des applications Android et iOS. Toutefois, notre solution pourrait aussi être appliquée avec d'autres langages et technologies, nous présenterons brièvement des technologies alternatives lorsque ce sera pertinent.

2. Pour en apprendre davantage sur le langage Nit, consultez le manuel disponible en ligne à `https://nitlanguage.org/manual/`.

Structure du document

La suite de ce document est organisée comme suit :

— Le chapitre 1 détaille la problématique visée, le développement d'applications mobiles et portables. Nous y définissons l'application mobile typique et précisons le problème de fragmentation du marché. Ensuite, nous établissons les critères de qualité d'une solution en développement d'applications portables et évaluons les solutions existantes du milieu scientifique et de l'industrie. Finalement, nous présentons notre solution et nos choix technologiques.

— Le chapitre 2 discute du rôle du prototype dans le processus de développement d'une application mobile et comment il est réalisé à l'aide d'API portables dans notre solution. Nous y discutons de la bibliothèque de Nit et surtout des quatre API de la bibliothèque *app.nit* qui offrent des services spécialisés pour les appareils mobiles. Nous introduisons les projets de validation, la calculatrice *app.nit* et le client mobile au service de microblogue Tnitter. De plus, nous présentons des services de support : les métadonnées et les ressources.

— Le chapitre 3 présente l'ingénierie de lignes de produits agiles et comment elle permet le développement d'une famille d'applications semblables avec des variations adaptées à Android et à iOS. Nous y présentons des solutions techniques pour réaliser les artéfacts de code, dont le raffinement de classes de Nit. Finalement, nous revisitons la calculatrice *app.nit* pour discuter de son organisation en ligne de produits.

— Le chapitre 4 présente la programmation polyglotte et comment elle est rendue possible par la FFI de Nit. Nous y présentons la FFI de Nit de façon générale, les critères ayant guidé sa conception et les fonctionnalités communes à tous les langages étrangers. Puis, nous présentons le support de C, Objective-C et Java, sous la forme de manuel de référence.

— Le chapitre 5 introduit trois patrons de conception polyglotte. Ces patrons proposent des solutions abstraites à des problèmes que nous avons observés à répétition au cours de ce projet de recherche. Nous y présentons également deux générateurs de code qui assistent les programmeurs utilisant nos patrons de conception et la FFI de Nit en général.

— Le chapitre 6 présente une étude de cas au sujet de la ligne de produits Tenenit. Cette étude sert d'exemple complet d'application de notre solution dans un contexte réel. Un lecteur intéressé par l'utilisation de notre solution en pratique peut commencer par ce chapitre pour en voir un exemple concret.

Nous concluons avec un retour sur notre solution, nos contributions et en listant des possibilités de futurs travaux de recherche.

CHAPITRE I

PROBLÉMATIQUE EN DÉVELOPPEMENT D'APPLICATIONS MOBILES

Ce chapitre délimite le domaine d'intérêt de cette thèse, le développement d'applications mobiles. Il s'adresse à toute personne désirant connaître les problèmes particuliers aux appareils mobiles ainsi que les solutions disponibles.

Nous commençons par définir l'application mobile typique ainsi que la problématique principale, la fragmentation du marché des appareils mobiles. Ensuite, nous établissons les critères de qualité d'une solution en développement d'applications mobiles, et nous les utilisons pour évaluer des solutions existantes du milieu scientifique et de l'industrie. Finalement, nous introduisons notre solution, survolons ses aspects principaux et comment ils répondent à nos critères.

1.1 L'application mobile typique

Avant tout, qu'est-ce qu'une application mobile, et qu'est-ce qui la différencie d'une application classique pour ordinateurs de bureau ? L'application mobile typique est communément définie comme suit :

— Elle est basée sur une interface utilisateur tactile, sans clavier physique ni souris. L'interface utilisateur doit adhérer au style de la plateforme pour profiter de l'intuition des utilisateurs.

— Elle est économe en énergie, car elle fonctionne sur un appareil avec une batterie limitée.

— Elle est compatible avec différents appareils, s'adapte à la taille des écrans et aux différents capteurs.
— Elle est connectée à un serveur distant pour la persistance des données, pour accéder à un réseau social ou pour télécharger des nouvelles. L'application peut compter sur la présence d'une connexion internet sans-fil, mais les connexions ne sont pas stables.
— Elle est utilisée fréquemment et rapidement. Par exemple, une application de clavardage est ouverte pour lire et envoyer les messages, et fermée entre les utilisations.

Sans être une liste exhaustive, nous utilisons ces particularités pour guider notre effort de recherche, viser des applications de validation représentatives et prioriser les fonctionnalités à offrir dans les API portables.

1.2 Fragmentation du marché

La principale entrave au développement d'applications mobiles est la fragmentation du marché. Tel qu'illustré à la figure 1.1, le marché est partagé entre les systèmes d'exploitation Android et iOS. Chacun ayant une part substantielle du marché, aucun des deux ne peut être ignoré par un programmeur visant un grand public. Les outils de développement standard de chaque plateforme sont incompatibles, non seulement les API natives sont différentes, mais leurs langages sont différents. Les API natives à Android sont en Java, et celles de iOS en Objective-C et en Swift.

Il y a aussi fragmentation à l'intérieur même des plateformes ; comme illustré dans la figure 1.2, plusieurs versions différentes d'Android sont encore en utilisation. Les applications Android visant une plateforme sont incompatibles avec les plateformes précédentes.

De plus, les périphériques matériels varient grandement entre les appareils. Par exemple, les écrans ont des tailles variables, en ce qui concerne iOS, la diagonale des écrans varie entre 4.0 pouces sur le iPhone SE et 12.9 pouces sur le iPad Pro. Les applications doivent être adaptées à ces différentes tailles d'écran. Beaucoup d'autres périphériques

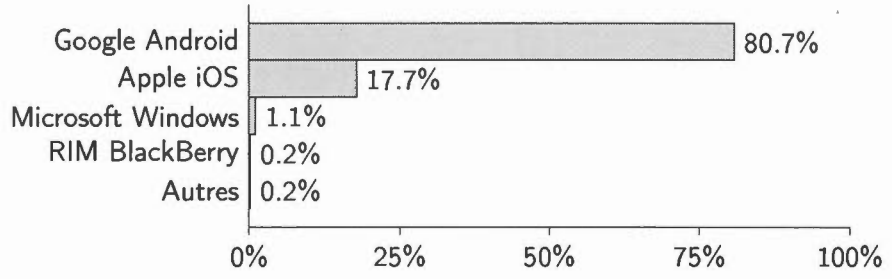


Figure 1.1: Parts de marché des systèmes d'exploitation sur téléphones intelligents, mondialement, à la fin de 2015 (Gartner, 2016).

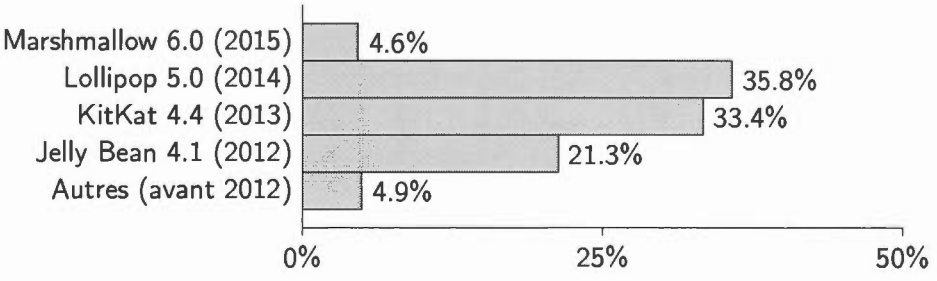


Figure 1.2: Versions d'Android installées sur les appareils actifs, mondialement, en avril 2016 (AOSP, 2016a).

sont optionnels, certains appareils ont une antenne cellulaire, un GPS, le support pour NFC, la vibration, une caméra ou même plusieurs caméras.

La fragmentation du marché et l'incompatibilité des outils de développement officiels d'Android et iOS obligent le programmeur à choisir les plateformes cibles dès le début du développement. La solution simple est de développer des applications distinctes pour chaque plateforme. C'est une solution encore populaire, une étude (Joorabchi, Mesbah et Kruchten, 2013) a rapporté que 63% des développeurs mobiles consultés ont réalisé la même application pour plus d'une plateforme.

Toutefois, il y a maintenant des solutions pour réaliser des applications portables aux deux plateformes. Les différentes solutions appliquent des stratégies variées pour obtenir des résultats tout autant variés.

1.3 Critères de qualité

Pour comparer les solutions en développement d'applications portables, nous avons établi une liste de huit critères. Nous avons constitué cette liste en analysant les critères mis de l'avant par les solutions existantes. Ces critères ont aussi guidé les choix de conception de notre solution.

1.3.1 Critère *prototype rapide*

Le prototype, une version de l'application non finale regroupant les fonctionnalités principales du projet, doit être produit rapidement. Il doit être portable à Android et iOS, et être réalisable avec des ressources limitées. À cette étape du projet, la solution ne doit pas ralentir le développement pour des préoccupations dépassant le prototype.

1.3.2 Critère *portabilité*

Le code réalisant l'application doit, autant que possible, être portable entre Android et iOS, mais aussi entre un client mobile et son serveur distant. Pour ce faire, la solution doit favoriser l'utilisation de technologies portables entre les plateformes mobiles ainsi qu'à toute autre plateforme pertinente.

1.3.3 Critère *fragmentation*

La solution doit surmonter la fragmentation du marché par un équilibre entre le code portable et l'adaptation de l'application à Android et à iOS. Elle doit donner accès aux fonctionnalités propres à chaque plateforme, même lorsque celles-ci ne sont pas disponibles sur l'autre plateforme.

1.3.4 Critère *écosystème*

La solution doit favoriser l'intégration de l'application à l'écosystème de chaque plateforme. Android et iOS définissent des contrôles natifs pour réaliser des interfaces utilisateur constantes entre les applications. Cette constance permet à l'utilisateur de développer une intuition qui peut être mise à profit pour alléger l'interface utilisateur avec des raccourcis.

1.3.5 Critère *API natives*

Android et iOS offrent chacun des services qui leur sont propres. Pour bien adapter l'application à chaque plateforme, le programmeur doit avoir un accès à tous les services offerts par les API natives. De plus, il doit avoir accès aux fonctionnalités offertes par des bibliothèques de tierces parties.

1.3.6 Critère *expertise*

La solution doit favoriser le transfert de l'expertise préalable sur les plateformes mobiles ; que ce soit l'expérience pratique d'un expert, la documentation officielle des plateformes ou les solutions aux problèmes communs présentés par les sites de questions et réponses.

1.3.7 Critère *évolution*

La solution doit prendre en compte l'évolution et la maintenance du projet. Un projet débute généralement par un prototype, qui évolue en (ou est remplacé par) des applications natives et distinctes par plateforme. Sur le long terme, les applications sont

maintenues pour compatibilité avec les nouvelles versions d'Android et iOS, et pour l'ajout des fonctionnalités.

1.3.8 Critère *assistance*

La solution doit assister le programmeur pour lui éviter de faire des erreurs, notamment en détectant les erreurs potentielles à la compilation ou en facilitant les tests automatisés. Toutefois, les tests automatisés sur les appareils mobiles sont difficiles à réaliser parce que, entre autres, les appareils sont variés, les ressources limitées sont difficiles à simuler et l'interface utilisateur est importante (Wasserman, 2010; Muccini, Di Francesco et Esposito, 2012). Détecter les erreurs tôt réduit le temps de développement et limite la dépendance aux tests automatisés.

1.4 État de l'art

Avec la montée en importance des applications mobiles, il y a un grand intérêt à trouver une solution aux problèmes de fragmentation. Dans cette section, nous présentons des solutions représentatives de l'offre de l'industrie et certaines solutions complètes du milieu scientifique. Les solutions partielles du milieu scientifique, qui visent une partie précise de la problématique, seront présentées plus loin dans ce document lorsqu'approprié. Sans discuter de chaque critère pour chaque solution, nous explorons ici comment les techniques mises de l'avant par chaque solution affectent certains critères. Cette approche nous permet de présenter les techniques ayant le plus grand effet sur chaque critère, qu'il soit positif ou négatif.

1.4.1 Android Native Development Kit – API alternative en C

La portabilité d'applications réalisées en C et C++ vers Android est un problème connu par Google. Les jeux, couramment écrits dans ces langages pour leur portabilité, sont difficiles à porter à Android avec seulement l'API native Java. La solution de Google a été de reproduire en C une partie de l'API native Java. L'API C est offerte par le

Android Native Development Kit (NDK), il expose des services spécialisés pour les jeux ainsi que des services de bas niveau.

Cette solution vise précisément le critère *portabilité*. Elle permet de partager la logique d'un jeu avec iOS, ainsi que l'affichage graphique avec OpenGL ES, mais elle nécessite du code spécifique à chaque plateforme pour traiter les entrées de l'utilisateur. Toutefois, l'API C ne permet pas de réaliser des interfaces utilisateur avec les contrôles natifs à Android, limitant ainsi le critère *écosystème*.

1.4.2 Bibliothèques de compatibilité Android

La fragmentation du marché Android est aussi un problème connu par Google. Chaque nouvelle version d'Android ajoute des fonctionnalités à l'API native Java. Une application Android utilisant une API récente, et ses nouvelles fonctionnalités, n'est pas compatible avec les versions précédentes d'Android.

Pour contourner ce problème, Google offre des bibliothèques de compatibilité (AOSP, 2016b). Elles étendent le support des nouvelles fonctionnalités aux versions précédentes d'Android. La première version publique a été annoncée en 2011, elle permettait aux anciens appareils d'utiliser les fragments, une API pour créer des interfaces utilisateur indépendantes de la taille de l'écran (Ducrohet, 2011). Lors de l'appel à une fonctionnalité, la bibliothèque détermine dynamiquement s'il faut le déléguer à l'API native, reproduire le comportement voulu ou alors traiter l'appel comme un *no-op*. Le principal désavantage pour le programmeur est que l'application doit inclure la bibliothèque de compatibilité, ce qui requiert plus d'espace disque.

En pratique, cette stratégie duplique partiellement l'API native. Ceci est visible dans la documentation officielle d'Android où certains exemples utilisent l'API native et d'autres, les bibliothèques de compatibilité.

Ces bibliothèques répondent au critère *fragmentation*, mais pour Android seulement. Elles sont compatibles avec notre solution et peuvent aider à adapter les applications à Android.

1.4.3 Apportable – API natives à iOS sous Android

L'entreprise Apportable offre une solution pour recompiler une application native à iOS en application Android (Appportable, 2016; Apportable, 2013). Leur solution est privée et l'entreprise publie peu de détails techniques. Elle semble utiliser une couche de compatibilité pour implémenter les API natives à iOS en Android.

L'approche de Apportable vise le critère de *portabilité*, sans tenir compte des différences entre les plateformes. Les applications Android résultantes sont limitées aux services d'iOS, ne respectant pas les critères *écosystème* ni *API natives*. Le principal cas d'utilisation est pour convertir des jeux d'iOS vers Android, car ceux-ci sont souvent basés sur des API portables tel que OpenGL ES.

1.4.4 XMLVM – Compilation croisée

La recompilation de code entre plateformes est la principale solution proposée dans la littérature scientifique. Le projet XMLVM (Puder et Lee, 2009) recompile le code octet Java des applications Android en un format XML, pour ensuite le recompiler vers Objective-C (Puder et Antebi, 2013). Ils ont notamment utilisé ce système pour recompiler des jeux d'Android vers iOS (Puder et Yoon, 2010).

De façon semblable à Apportable, cette solution ne permet pas d'adapter une application à iOS, ne respectant pas les critères *écosystème* ni *API natives*.

1.4.5 Cordova et Appcelerator – Technologies du web

La solution libre Apache Cordova (Apache Software Foundation, 2013) et la solution privée Appcelerator (Appcelerator, 2016) mettent à profit les technologies du web pour réaliser des applications portables en JavaScript, HTML et CSS. Une API JavaScript permet d'accéder à des fonctionnalités communes à Android et iOS, et l'interface utilisateur est déclarée en HTML et CSS. L'application prend alors la forme d'une coquille native englobant un navigateur qui affiche une application web.

Le principal attrait de ces solutions est qu'elles comblent bien le critère *prototype rapide* et, le code pouvant être partagé avec une application web classique, elles répondent également au critère *portabilité*. Toutefois, sans adaptation aux plateformes et sans contrôles natifs, ces solutions ne comblent pas les critères *écosystème* ni *API natives*.

1.4.6 Kivy et Unity3D – OpenGL ES

La solution libre Kivy propose de réaliser des applications graphiques en Python à l'aide de OpenGL ES. OpenGL ES est un API graphique portable compatible à Android, iOS, Windows et GNU/Linux. Par le fait que Kivy est basé sur OpenGL ES, il est portable aux mêmes plateformes. Toutefois, Kivy ne donne pas accès aux contrôles natifs, ne respectant pas les critères *écosystème* ni *API natives*.

Kivy est aussi utilisé pour réaliser des jeux, tout comme la solution privée Unity3D qui offre une API C# pour développer des jeux et des applications multimédias. Unity3D est aussi basé sur OpenGL ES et ne donne pas accès aux contrôles natifs.

1.4.7 Djinni – Logique d'affaires portable et interface utilisateur native

La compagnie Dropbox publie une application du même nom pour Android, iOS, Windows, macOS, GNU/Linux et autres. Sur les plateformes mobiles, la logique d'affaires en C++ est portable. Par contre, l'interface utilisateur est distincte, réalisée en Java pour Android et en Objective-C pour iOS. La logique d'affaires et l'interface utilisateur sont connectées à l'aide de l'outil Djinni, développé par la même compagnie.

L'outil Djinni est un générateur de code et d'API. Il génère le code réalisant les échanges entre les langages, ainsi que les API en C++, Objective-C et Java qui exposent les services des autres langages. Les services de chaque langage et les interfaces à générer sont décrits explicitement dans un langage de description d'interface (IDL).

Cette solution répond au critère de *portabilité* pour la logique d'affaires seulement, le code des interfaces utilisateur est propre à chaque plateforme. Ce compromis permet

d'adaptater l'application à chaque plateforme, dans le langage natif à la plateforme, et de répondre aux critères de *fragmentation*, *API natives*, *écosystème* et *expertise*. De plus, la génération des API et la sûreté statique des langages C++, Objective-C et Java répondent bien au critère *assistance* en rapportant des erreurs à la compilation. Par contre, Djinni ne facilite aucunement le développement d'un prototype portable, allant à l'encontre du critère *prototype rapide*.

1.4.8 Xamarin – API portables en C#

Xamarin est une solution populaire pour réaliser des applications portables en C#. Nous l'avons déjà mentionné dans l'introduction pour son achat récent par Microsoft.

L'interface utilisateur d'une application Xamarin est réalisée programmativement en C# ou déclarativement en XAML. Les interfaces utilisateur résultantes prennent la forme de contrôles natifs à chaque plateforme. Cette fonctionnalité contribue au critère *écosystème* qui est d'ailleurs mis de l'avant par Xamarin.

Pour donner accès aux fonctionnalités propres à chaque plateforme, Xamarin réplique les API d'Android et d'iOS en C# . Cette approche répond partiellement au critère *API natives*, les API officielles sont accessibles, mais celles offertes par une tierce partie ne le sont pas. L'accès limité restreint également le critère *écosystème* qui n'est que partiellement comblé par les contrôles natifs. De plus, l'expertise est difficile à transférer aux API répliquées, car celles-ci ne peuvent pas reproduire fidèlement les API natives de par les différences entre les langages, une embûche au critère *expertise*.

Xamarin est une solution intéressante, permettant de réaliser un prototype portable rapidement et de l'adapter de façon limitée aux plateformes. De plus, l'utilisation du langage C# répond bien au critère *assistance*.

1.4.9 React Native – API portable en JavaScript

La compagnie Facebook a également proposé une solution de développement mobile en publiant React Native (Facebook Inc., 2016a). React Native vise à amener les outils du web au développement d'applications mobiles, reprenant les bases de leur solution ReactJS utilisée dans l'implémentation de *facebook.com* (Occhino, 2015). Cette logique met de l'avant le critère *portabilité* en favorisant le partage du code entre l'application mobile, une application web et même avec le serveur. Toutefois, alors que ReactNative vise à préserver l'expertise en développement web, il ne préserve pas l'expertise en développement mobile.

React Native repose sur des API JavaScript offrant les services communs à Android et iOS pour réaliser des applications portables et répondre au critère *prototype rapide.* Les applications peuvent aussi être adaptées à Android et à iOS via des API natives reproduites en JavaScript. Tout comme Xamarin, cette approche ne répond que partiellement au critère *API natives.*

Toutefois, React Native met de l'avant les modules natifs qui donnent accès aux langages natifs à Android et iOS ainsi qu'à leur API en entier (Facebook Inc., 2016b). Ils permettent de créer des fichiers source Java ou Objective-C, et d'invoquer leurs fonctions depuis le code JavaScript. Par contre, les modules natifs ne mettent pas à profit les récentes avancées dans le domaine de la programmation polyglotte. Il s'agit d'une interface dynamique qui demande beaucoup de *code colle* de la part du programmeur et ne profite pas de la sûreté statique de Java et de Objective-C. Ces limites permettent aux modules natifs de ne répondre que partiellement aux critères *API natives* et *expertise.*

Pour faciliter l'évolution de l'application, React Native organise le code en composantes, ce qui en facilite la maintenance et la réutilisation. Les composantes sont un premier pas dans les lignes de produits, mais la solution ne facilite pas les petites variations ni une organisation globale du code.

Malgré les problèmes déjà soulevés, selon nos critères, React Native est la meilleure solution sur le marché. Par contre, sa grande limite est le choix de JavaScript comme langage hôte. JavaScript n'est pas statiquement typé, ce qui nuit à la détection d'erreurs et qui permet d'implémenter peu d'analyses, allant à l'encontre du critère *assistance.*

1.4.10 Deux applications natives distinctes

Une dernière option donne un différent point de vue sur les solutions précédentes : simplement réaliser deux applications distinctes, une pour Android et une pour iOS.

Au besoin, le programmeur peut d'abord réaliser un prototype temporaire avec une solution comme Cordova, répondant au critère *prototype rapide*. Le prototype peut être suffisant comme preuve de concept, mais aucune adaptation n'étant possible, il est abandonné en faveur d'applications distinctes dès que les ressources le permettent.

Ensuite, le programmeur réalise deux applications natives distinctes qui ne partagent aucune ligne de code. Il s'agit de la solution idéale pour les critères *écosystème*, *API natives*, *expertise* et *assistance*. Toutefois, ignorant le critère *portabilité*, cette solution nécessite d'écrire approximativement deux fois plus de code qu'avec une solution portable. Et encore davantage si un prototype portable a été réalisé par une autre solution avant d'être abandonné.

1.4.11 Comparaison des solutions et intérêt pour une nouvelle solution

Les solutions existantes sont variées, appliquent différentes techniques, visent différents cas d'utilisation et chacune comble différents critères de qualité. Pour en faciliter la comparaison, le tableau 1.1 associe les solutions avec les techniques appliquées et les cas d'utilisation. En plus des techniques mentionnées directement plus haut — la recompilation entre les plateformes, l'utilisation d'API portables et la reproduction des API natives en un autre langage — on y ajoute des approches plus générales : permettre l'écriture de code dédié à une plateforme, favoriser la séparation des préoccupations, encourager la programmation polyglotte et offrir une FFI moderne – au niveau de la littérature scientifique actuelle.

Quoique certaines s'en approchent, aucune solution ne comble tous nos critères de qualité. Le tableau 1.2 classifie les solutions selon nos huit critères de qualité. On remarque que quatre critères ne sont pas comblés simultanément de façon satisfaisante par les solutions déjà existantes : *API natives*, *expertise*, *évolution* et *assistance*.

Tableau 1.1: Comparaison des solutions selon les techniques appliquées.

Solution	Recompilation entre plateformes	API portable	Reproduction des API natives	Code dédié à une plateforme	Séparation des préoccupations	Programmation polyglotte	FFI moderne	Cas d'utilisation
Apportable	oui	-	-	-	-	-	-	Projet iOS existant
XMLVM	oui	-	-	-	-	-	-	Projet Android existant
Cordova	-	oui	-	-	-	-	-	Projet web
Kivy	-	oui	-	-	-	-	-	Nouveau projet
Djinni	-	-	-	oui	-	-	-	Nouveau projet
Xamarin	-	oui	oui	oui	-	-	-	Nouveau projet
React Native	-	oui	oui	oui	oui	oui	-	Nouveau projet
app.nit	-	oui	-	oui	oui	oui	oui	Nouveau projet

Considérant cet état des choses, nous croyons qu'il y a un intérêt pour une nouvelle solution de développement mobile. En fait, les solutions ont progressé au fil des ans, passant d'applications entièrement portables à des applications de plus en plus adaptées aux plateformes. Notre solution poursuit ce progrès en simplifiant l'adaptation aux plateformes tout en facilitant l'évolution et la maintenance du projet. Pour ce faire, nous réutilisons les points forts des techniques existantes et en proposons de nouvelles pour pallier aux faiblesses.

1.5 Notre solution – *app.nit*

Notre solution en développement d'applications portables à Android et à iOS est nommée *app.nit*, elle vise à surmonter le problème de fragmentation du marché par un compromis sur la portabilité du code. La majorité du code est portable entre les deux plateformes, mais l'application peut aussi être adaptée à une plateforme en accédant aux

Tableau 1.2: Comparaison des solutions selon nos huit critères.

Solution	1. Prototype rapide	2. Portabilité	3. Fragmentation	4. Écosystème	5. API natives	6. Expertise	7. Évolution	8. Assistance
Apportable		++				+		
XMLVM		++				+		
Cordova	++	++						
Kivy	++	++						
Djinni		+	+	++	+	++		++
Xamarin	++	+	+	+	+			++
React Native	++	+	++	++	+	+	+	
app.nit	++	+	++	++	++	++	++	++

Chaque solution est associée à un critère par + si le critère est adressé minimalement et par ++ s'il est comblé de façon satisfaisante.

API natives via le langage natif à la plateforme. Notre solution rassemble trois concepts fondamentaux accompagnés de solutions techniques :

1. Des API portables permettent de réaliser rapidement un prototype fonctionnel sous Android et iOS. Ils sont offerts par la bibliothèque standard de Nit, certains modules portables de la bibliothèque étendue de Nit, et surtout par la bibliothèque *app.nit* qui offre des services communs à Android et iOS. La bibliothèque *app.nit* définit quatre API pour répondre aux besoins des applications mobiles : interface utilisateur, cycle de vie, persistance des données et requêtes HTTP.

   Les API portables sont une fondation solide pour réaliser des applications portables à plusieurs plateformes. L'offre de la bibliothèque de Nit et de *app.nit* est suffisante pour réaliser des applications réelles entièrement portables.

   Les quatre API de la bibliothèque *app.nit* sont implémentées différemment sur chaque plateforme en utilisant les API natives. Notamment, l'API d'interface utilisateur est réalisée par les contrôles natifs à Android et à iOS pour intégrer l'application à l'écosystème de chaque plateforme.

2. Les projets d'applications *app.nit* sont organisés en ligne de produits pour une évolution graduelle du prototype vers des applications adaptées à Android et à iOS. Cette organisation structure les différences entre les deux plateformes et les variations entre les applications finales. Ceci surmonte la fragmentation du marché tout en facilitant la maintenance de l'application.

   Les lignes de produits sont réalisées au niveau du code par le raffinement de classes du langage Nit. Le raffinement de classes permet d'ouvrir des classes et méthodes existantes pour ajouter ou modifier des services. Il sert également à déclarer les points de variation dans le code et à joindre les variations en différents produits.

3. La programmation polyglotte consiste à utiliser plusieurs langages selon leurs spécialités pour réaliser une même application. Le langage Nit est idéal pour le code portable, c'est un langage généraliste ayant accès à la bibliothèque *app.nit* et à des services pour échanger avec les langages natifs. Pour le code spécifique à chaque

plateforme, Objective-C et Java donnent accès aux API natives et permettent d'appliquer l'expertise préalable sur les deux plateformes.

La programmation polyglotte est facilitée par la FFI de Nit. La FFI s'intègre au paradigme de programmation orientée objet en offrant une forme alternative aux méthodes, classes et appels de méthodes. Elle supporte plusieurs langages étrangers dont C, Objective-C et Java, elle utilise une syntaxe imbriquée pour une utilisation agréable et elle préserve la sûreté statique des langages étrangers pour assister le programmeur. De cette façon, elle ouvre un accès sûr aux API natives à Android et à iOS, et ce, dans leur entièreté et leur langage natif.

1.6 Portée et choix des technologies

Pour réaliser ce projet dans un temps raisonnable, nous avons eu à limiter son ampleur et à choisir tôt les technologies sur lesquelles bâtir notre expérimentation.

1.6.1 Plateformes visées

Ce projet de recherche vise les plateformes Android et iOS, des systèmes d'exploitation utilisés sur téléphones et tablettes. Ce choix a été motivé par leur popularité ; combinés, Android et iOS dominent le marché. De plus, la plateforme Android, qui expose une API en Java uniquement, s'avère être un défi notable à relever.

Pour faciliter le développement, notre solution supporte partiellement la plateforme GNU/Linux. Cette plateforme simplifie le débogage du code Nit portable, car elle partage plusieurs similarités avec Android et iOS. Toutefois, GNU/Linux n'étant pas concernée par les problématiques de développement mobile, nous n'y ferons référence que lorsque ce sera pertinent.

1.6.2 Langage hôte

Le raffinement de classes et l'ancienne interface native de Nit ayant inspiré ce projet de recherche, il a été naturel d'utiliser Nit comme langage portable et hôte à la programmation polyglotte. D'autant que nous avons une expertise et un accès incomparable au compilateur Nit et à sa bibliothèque.

Nous avons tout de même considéré d'autres langages. Dans tous les cas, on aurait eu à modifier le langage et ses outils pour ajouter les fonctionnalités sur lesquelles repose cette thèse. Avec Nit, le travail à réaliser était prévisible et réaliste dans le cadre de cette thèse. Nous présenterons les différents langages considérés ainsi que leurs aspects pertinents à ce projet, lorsqu'appropriés tout au long de ce document.

1.6.3 Langages natifs

Nous avons retenu un langage natif par plateforme visée : Java pour Android et Objective-C pour iOS. Ces langages sont utilisés par le programmeur pour accéder aux API natives.

Les API natives d'Android sont compatibles avec tous les langages compilant vers du code octet Java. Nous avons préféré Java à Scala (et autres) parce que les exemples dans la documentation officielle d'Android sont en Java. Java a également été préféré à C, car l'API native d'Android en Java est plus complète que l'API C offerte par le NDK. Notamment, l'API C ne donne pas accès aux contrôles d'interfaces utilisateur natifs à Android. Par contre, la compilation de Nit pour Android est basée sur des outils du NDK qui permettent de compiler le code C généré en binaire pour Android.

Pour iOS, nous avons préféré Objective-C à Swift comme langage natif. Au début du projet de recherche, l'API native d'iOS était uniquement disponible en Objective-C. Apple a publié le langage Swift avec son API native pour iOS en 2014. À ce moment, nous avons décidé de poursuivre avec Objective-C car Swift n'offrait pas d'avantages significatifs. De plus, l'utilisation de Objective-C simplifie le développement de la bibliothèque, car il a une compatibilité binaire avec le C généré par le compilateur Nit.

1.6.4 API portables

En support à notre expérimentation, nous avons conçu et implémenté la bibliothèque *app.nit* qui regroupe quatre API portables spécialisées pour les applications mobiles. Nous avons eu à limiter les services offerts pour que l'effort de développement soit réaliste dans le cadre de ce projet de recherche. Les services retenus priorisent les besoins de l'application mobile typique présentée à la section 1.1.

1.7 Validation

Pour valider notre solution, nous avons réalisé trois projets d'applications mobiles. Chaque application vise différentes API et profite de différents avantages de notre solution.

La calculatrice *app.nit* est réalisée avec les API interface utilisateur, cycle de vie et persistance des données. L'application Tnitter, un client mobile au microblogue du même nom, repose sur l'API requêtes HTTP. Ces deux applications seront présentées avec plus de détails pour illustrer différents aspects de notre solution aux chapitres 2 et 3.

La ligne de produits Tenenit offre une famille d'applications mobiles pour les clients de brasseries artisanales. Tenenit est un projet réel et complexe, il sera sujet d'une étude de cas présentée au chapitre 6.

1.8 Conclusion

Notre solution en développement d'applications portables retient les points forts des solutions existantes et, de plus, elle comble les critères manquants. Les API portables offrent les fonctionnalités communes aux différentes plateformes et permettent de réaliser rapidement un prototype portable. L'organisation en ligne de produits et sa réalisation avec le raffinement de classes permettent de faire évoluer graduellement le prototype en applications distinctes pour Android et iOS, ainsi que de structurer les variations. Et finalement, la programmation polyglotte ouvre l'accès aux API natives et aux fonctionnalités propres à chaque plateforme, permettant aux applications d'atteindre le niveau de qualité des applications natives.

CHAPITRE II

PROTOTYPE ET API PORTABLES

Un projet d'application mobile utilisant notre solution *app.nit* débute par la réalisation du prototype portable, une version non finale de l'application qui est fonctionnelle sous Android et iOS. Sans être adapté aux plateformes, le prototype rassemble les fonctionnalités principales du projet.

Le prototype est réalisé à l'aide des services de la bibliothèque standard de Nit ainsi que des API portables *app.nit*[1] qui offrent des services communs à Android et iOS. L'utilisation d'API portables est une approche éprouvée pour réaliser des applications portables, la portabilité des applications Java est assurée par une bibliothèque de classes portables qui agit d'interface abstraite aux fonctionnalités communes aux différents systèmes d'exploitation.

Ce chapitre s'adresse aux utilisateurs de notre solution *app.nit*. Il en décrit les services principaux et explique brièvement nos choix de conception. Toutefois, il ne donne pas la spécification précise des API, la documentation en ligne présente déjà cette information.

Dans ce chapitre, nous définissons le prototype et son rôle dans un projet d'application mobile. Nous présentons les API portables préexistantes du projet Nit et la bibliothèque *app.nit* conçue en support à ce projet de recherche. De plus, nous décrivons les services pour spécifier les métadonnées et les ressources des applications mobiles. Finalement, nous présentons comment compiler le code du prototype en application pour Android et iOS.

---

1. Nous utilisons le terme *app.nit* pour qualifier ce qui est dérivé de notre solution, dont la bibliothèque *app.nit*, les quatre API *app.nit*, les applications *app.nit* et la calculatrice *app.nit*.

2.1 Le prototype

Le prototype est la première version de l'application qui rassemble les fonctionnalités principales du projet. Sa réalisation est une étape importante d'un projet d'application mobile. Le prototype peut être diffusé au public cible pour avoir un retour de la clientèle. Il peut aussi servir de preuve de concept pour la recherche de financement. Pour ces raisons, il doit bien présenter les fonctionnalités importantes et innovatrices du projet.

Le prototype sert à évaluer le potentiel du projet en entier et déterminer si l'effort doit être poursuivi. Selon le retour de la clientèle et la disponibilité de financement, un afflux de nouvelles ressources permet de poursuivre le développement et alors, le projet peut évoluer vers des applications adaptées à Android et iOS.

Dans le cadre de ce projet de recherche, on s'intéresse surtout au prototype portable, une version de l'application entièrement compatible avec Android et iOS. Le prototype portable rassemble les fonctionnalités principales sans être adapté à une plateforme. Il est non final par définition, toutefois, nous discuterons au chapitre suivant comment il peut côtoyer des applications adaptées et servir au débogage.

2.2 Bibliothèques de Nit

Le langage Nit est accompagné de sa bibliothèque standard et d'une bibliothèque étendue. Les bibliothèques regroupent des paquets de modules offrant des services variés, allant de structures de données performantes jusqu'à des services de communication. Certains paquets sont portables à Android et iOS, ils peuvent alors être utilisés pour réaliser le prototype portable.

Dans cette section, nous définissions des critères pour évaluer la portabilité des paquets de la bibliothèque standard et de la bibliothèque étendue.

2.2.1 Bibliothèque standard

La bibliothèque standard de Nit offre des services généraux pour toutes sortes d'applications. Elle est implémentée en Nit et en C, à l'aide de la FFI de Nit qui sera présentée en détail au chapitre 4. Elle utilise des services offerts par les API portables *libc* et POSIX. Ces caractéristiques assurent la portabilité de la bibliothèque standard vers Android et iOS. Le langage C est natif aux deux plateformes, et le code Nit est compilé vers du code C par le compilateur officiel. De plus, *libc* et POSIX sont supportés par Android et iOS.

Nous avons modifié certains services de la bibliothèque standard pour les adapter davantage aux plateformes mobiles. Notamment, sous Android, la méthode `print` redirige les messages vers le journal de développement, un outil standard de la suite d'outils pour Android.

2.2.2 Bibliothèque étendue

Les paquets de la bibliothèque étendue de Nit offrent une grande variété de services. Ils sont implémentés en Nit pur ou alors ils incorporent du C, du Java ou de l'Objective-C. De plus, certains dépendent de bibliothèques natives ou de bibliothèques de classes Java.

Nous listons ici les critères qui déterminent si un paquet est portable ou non à Android et à iOS. Nous considérons le problème du point de vue du programmeur qui utilise la bibliothèque étendue.

— Les paquets réalisés en Nit pur sont portables à Android et iOS. Par exemple, les services de sérialisation d'objets Nit en JSON sont entièrement portables. Il s'agit d'une de nos contributions pour faciliter la communication entre les applications mobiles et les serveurs distants. Ces services sont implémentés en Nit et dépendent seulement de paquets portables, dont la bibliothèque standard.

— Les paquets réalisés partiellement en C, à l'aide de la FFI de Nit, dépendent de la portabilité du code C et de celle des bibliothèques natives utilisées. Par exemple, les services de sérialisation vers des flux binaires sont réalisés partiellement en C. Ils permettent d'échanger efficacement une grande quantité de données. Ces services sont portables, car le code C prend en compte les restrictions de différentes architectures, dont celles des processeurs ARM à 32 bits souvent utilisés par les appareils Android et iOS. De plus, ils ne dépendent d'aucune bibliothèque native particulière.

  Lorsqu'il y a une dépendance à une bibliothèque native, la portabilité du paquet dépend de celle de la bibliothèque. Par exemple, la bibliothèque native `xlib` offre des services pour communiquer avec le gestionnaire de fenêtre X11, principalement utilisé sous GNU/Linux. Cette bibliothèque n'est pas portable à Android ni à iOS, donc les paquets Nit qui en dépendent ne le sont pas non plus. Pour cette raison, elle est utilisée sous GNU/Linux seulement, d'autres services offrent un équivalent sous Android et iOS.

— Finalement, les paquets réalisés partiellement en Java, encore une fois avec la FFI de Nit, sont portables à Android et à GNU/Linux, alors que ceux réalisés en Objective-C sont portables à iOS, macOS et GNU/Linux. Ils dépendent aussi de la portabilité des bibliothèques étrangères, qu'elles soient en Java ou Objective-C.

2.3 Bibliothèque *app.nit*

En support au développement d'applications mobiles, nous avons bonifié l'offre de services portables de Nit avec la bibliothèque *app.nit*. Cette bibliothèque offre des fonctionnalités spécialisées pour les appareils mobiles, organisées sous quatre API : l'API interface utilisateur, l'API cycle de vie, l'API persistance des données et l'API requêtes HTTP.

Nous avons sélectionné ces quatre API pour répondre aux besoins d'une application mobile typique. Bien entendu, ces API peuvent toujours être étendues, mais elles sont suffisantes pour valider notre solution dans leur état actuel.

La conception et l'implémentation de ces API sont la première contribution technique importante de ce projet de thèse. Chaque API a été conçue suite à une analyse des services disponibles en Android et iOS. En plus de servir à réaliser des applications mobiles, ces API pourront servir à des expérimentations futures.

La bibliothèque *app.nit* et ses API sont réalisées par plusieurs paquets, ceux-ci sont illustrés à la figure 2.1. Le paquet app définit l'interface abstraite des API, il utilise la bibliothèque standard de Nit et il est importé par tous les prototypes d'applications *app.nit*. Deux paquets implémentent les API séparément pour Android et pour iOS. Ces deux paquets modifient les services abstraits par raffinement de classes (discuté au chapitre suivant) pour faire appel aux services natifs de chaque plateforme par programmation polyglotte (discuté au chapitre 4). Un autre paquet fait l'équivalent pour GNU/Linux, toutefois il ne sert qu'au débogage, les services spécialisés pour les appareils mobiles n'étant pas adaptés aux ordinateurs de bureaux.

Le reste de cette section présente les interfaces abstraites des quatre API avec un accent sur les choix de conception. Pour illustrer les besoins comblés par chaque API, nous introduisons deux applications ayant servi à les valider. De plus, nous présentons certaines alternatives appliquées par d'autres solutions en développement d'applications portables.

2.3.1 API interface utilisateur (API UI)

L'interface utilisateur est un aspect important des applications mobiles, elle contribue de façon notable à l'intégration des applications dans l'écosystème d'Android et d'iOS. D'autant plus que les applications mobiles dépendent d'une interface utilisateur tactile, puisque les appareils ont rarement un clavier et une souris physique.

L'API UI offre une suite de contrôles pour construire des interfaces utilisateur portables. Chaque contrôle (étiquette, bouton, boîte à cocher, etc.) est implémenté séparément selon la plateforme, par des contrôles natifs à Android et à iOS. Cette approche favorise

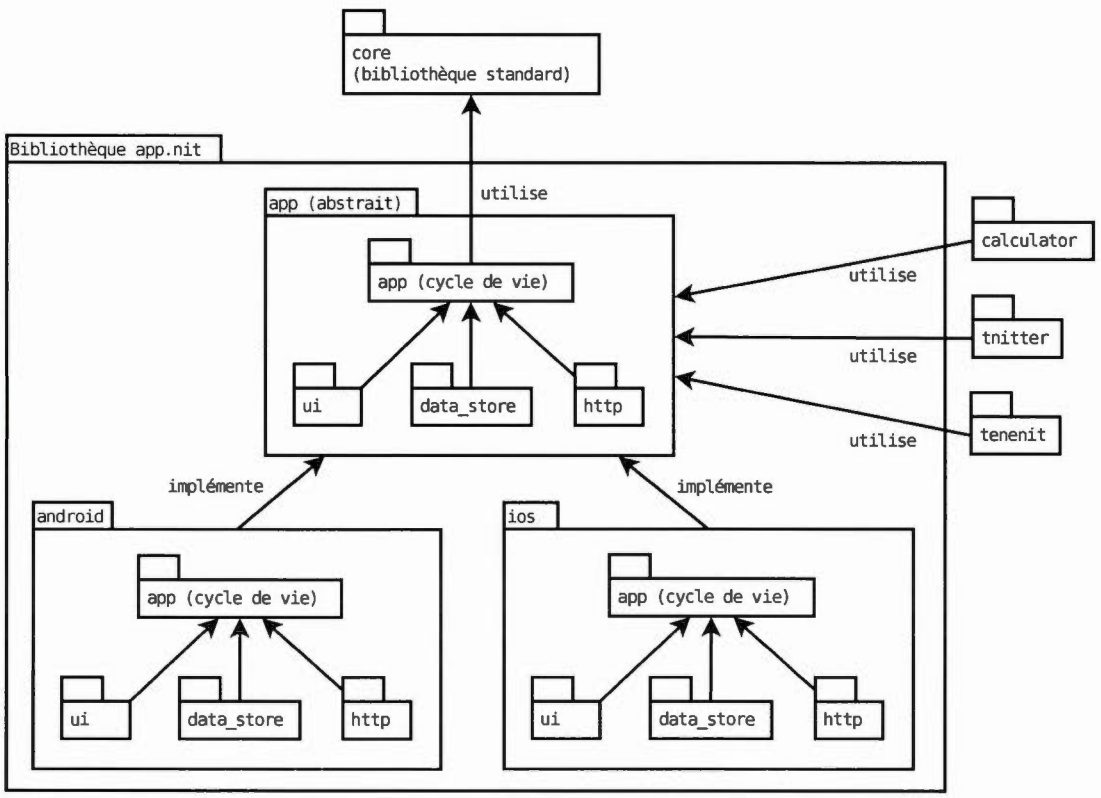


Figure 2.1: Structure de la bibliothèque *app.nit*.

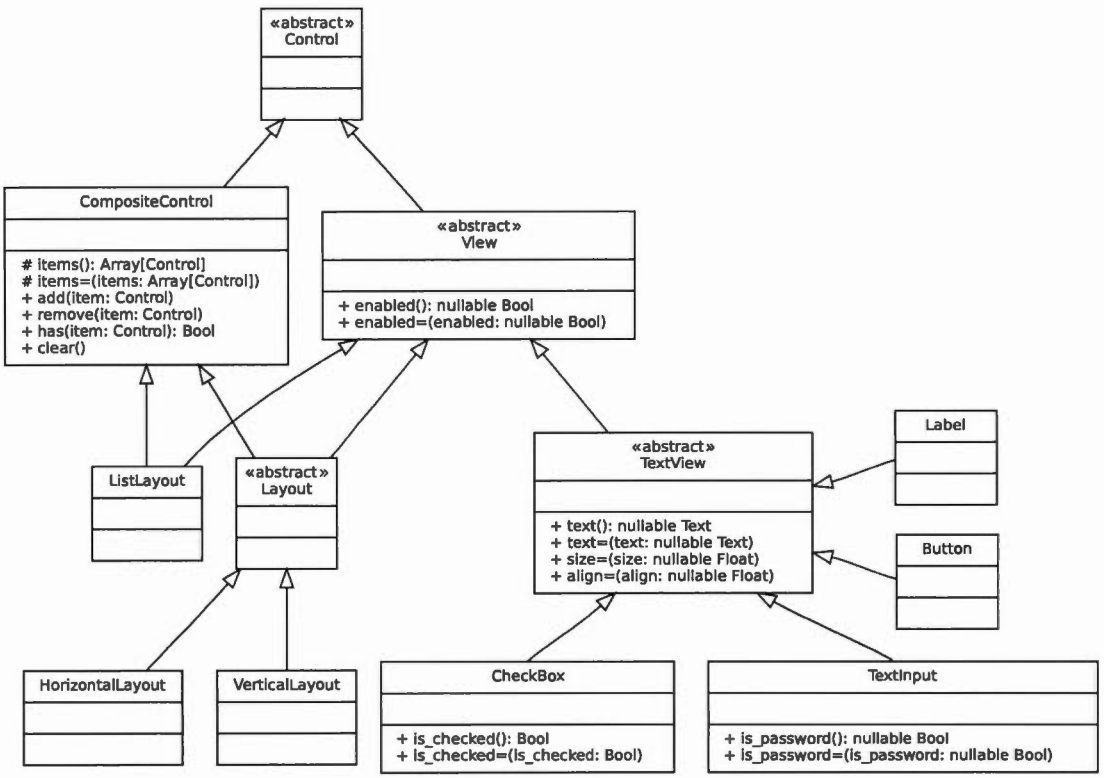


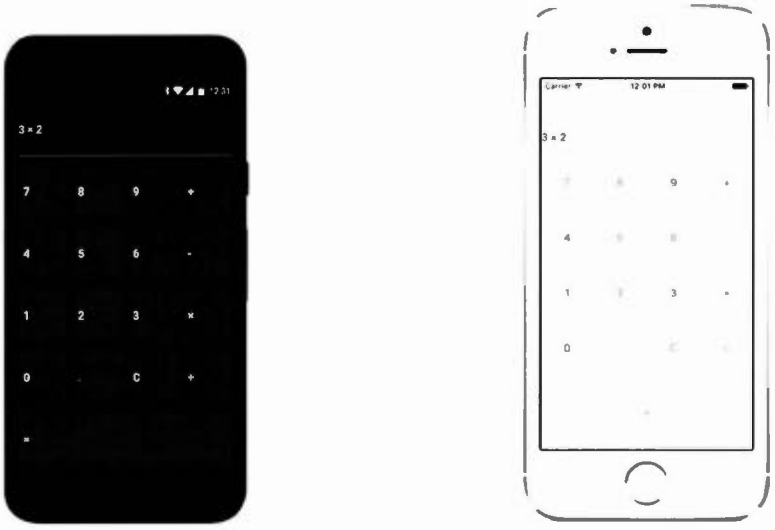
Figure 2.2: Diagramme de classes des principaux contrôles de l'API UI.

l'intégration de l'application à l'écosystème en lui donnant une apparence native, identique aux applications natives développées avec la suite d'outils officielle à chaque plateforme. D'autres solutions, dont React Native et Xamarin, utilisent aussi des contrôles natifs d'une façon similaire.

Pour concevoir l'API UI, nous avons analysé les API natives à Android et à iOS. La première étape consistait à identifier les services semblables entre les deux plateformes et à sélectionner les plus importants. Les services retenus sont listés dans le tableau 2.1. Ensuite, nous avons étudié la hiérarchie des classes des API natives pour en extraire un graphe d'héritage commun (ou presque). L'API UI de *app.nit* reproduit ce graphe dans sa hiérarchie de classes, telle que présentée à la figure 2.2.

Tableau 2.1: Contrôles Android et iOS représentés par l'API UI.

Android	iOS	app.nit	Description
android.app.Fragment	UIViewController	Window	Fenêtre contenant des contrôles.
android.view.View	UIView	View	Superclasse de tous les contrôles.
android.widget.TextView	UILabel	Label	Simple étiquette.
android.widget.Button	UIButton	Button	Bouton avec une étiquette.
android.widget.EditText	UITextField	TextInput	Champs de saisie de texte.
android.widget.CheckBox	UILabel et UISwitch	CheckBox	Boîte à cocher avec une étiquette.
android.widget.LinearLayout	UIStackView	Layout	Organisation de contrôles sur une ligne horizontale ou verticale.
android.widget.ListView	UIScrollView et UIStackView	ListLayout	Organisation de contrôles sur la verticale, avec défilement au besoin.



(a) Prototype sous Android (b) Prototype sous iOS

Figure 2.3: Prototype de la calculatrice sous Android et iOS.
En haut de la fenêtre, on y voit le `TextInput`, puis des `Button` organisés en grille.
La grille est réalisée par des `Layout`, un vertical et plusieurs horizontaux.

2.3.1.1 Application – Calculatrice *app.nit*

Pour évaluer notre solution, nous avons réalisé la calculatrice *app.nit*.[2] Son interface utilisateur repose sur l'API UI et utilise au moins cinq contrôles : `Window`, `View`, `Button`, `TextInput` et `Layout`. Le prototype, défini par environ 300 lignes de code, est entièrement portable entre Android et iOS. La figure 2.3 présente l'interface utilisateur du prototype sous les deux plateformes.

La calculatrice peut être comparée avec les calculatrices installées par défaut sur les appareils Android et iOS. Nous présentons une telle comparaison au prochain chapitre avec l'adaptation de la calculatrice aux deux plateformes.

---

2. Le code source de la calculatrice *app.nit* est disponible dans le dépôt Nit et sur le web à `https://github.com/xymus/nit/tree/these_alexis/examples/calculator/`

2.3.1.2 Alternative – Éviter les contrôles natifs

Une alternative à utiliser les contrôles natifs consisterait à définir des contrôles propres à notre solution qui seraient identiques d'une plateforme à l'autre. Cette alternative est appliquée par quelques solutions existantes, dont Cordova qui n'utilise pas les contrôles natifs mais plutôt une interface HTML, et Kivy qui utilise OpenGL ES. L'avantage est que les applications résultantes ont une apparence identique sur Android et sur iOS. Le développement est simplifié, car le programmeur n'a pas à se soucier des changements de style et de taille des contrôles entre les plateformes. Par contre, les applications ne s'intègrent pas à l'écosystème de chaque plateforme.

2.3.1.3 Alternative – Langage de description

L'API UI est, par définition, une interface de programmation qui construit l'interface utilisateur à l'exécution de l'application. Une alternative à l'API UI serait d'utiliser un langage de description pour déclarer statiquement l'interface utilisateur.

Notamment, Android permet de réaliser les interfaces utilisateur de deux façons, avec une API ou en utilisant XML comme langage de description. De façon semblable, la solution Xamarin utilise XAML pour déclarer une interface utilisateur portable, XAML est alors recompilé vers les langages de description propres à chaque plateforme.

Le principal avantage des langages de description est la séparation de la logique de l'interface utilisateur du reste du code. De plus, les langages de description sont généralement utilisés en parallèle avec des outils graphiques qui permettent de construire l'interface utilisateur et de voir le résultat immédiatement. Toutefois, les langages de description agissent seulement à la compilation, ils ne permettent pas de modifier l'interface à l'exécution.

Nous avons préféré réaliser une API parce qu'elle permet autant de construire une interface utilisateur complète que de la modifier à l'exécution. La réalisation d'un seul système était plus réaliste dans le cadre de ce projet de recherche. De plus, lors d'un travail futur, l'API pourra servir à appliquer un langage de description pour construire l'interface utilisateur, alors que l'inverse serait impossible.

2.3.2 API cycle de vie

L'API cycle de vie est centrale à la bibliothèque *app.nit*, elle est utilisée par les trois autres API. Elle abstrait le cycle de vie des applications Android et iOS. Elle envoie des messages à l'application lorsque, entre autres, l'application devient visible à l'utilisateur ou qu'elle est mise en pause.

Le cycle de vie des applications mobiles est plus complexe que celui des applications pour ordinateurs de bureaux qui ont un seul point d'entrée. De façon à économiser la batterie et d'autres ressources limitées de l'appareil, une application mobile peut être mise en pause à tout moment, pour ensuite être relancée ou terminée.

Pour illustrer le cycle de vie d'une application mobile, prenons l'exemple d'un utilisateur qui reçoit un appel pendant qu'il utilise l'application calculatrice sur son téléphone. L'application de téléphonie s'ouvre et la calculatrice passe en arrière-plan. Le système d'exploitation envoie un message à la calculatrice pour qu'elle se mette en pause. Pendant l'appel, si le système d'exploitation détecte qu'il manque les ressources nécessaires pour assurer une bonne qualité d'appel, il envoie un message de terminaison à la calculatrice. À la fin de l'appel, l'utilisateur ferme l'application de téléphonie et s'attend à retrouver la calculatrice dans l'état où il l'avait laissée, affichant le calcul en cours. Pour ce faire, le système d'exploitation relance la calculatrice et lui envoie un message pour qu'elle recharge son état et donne l'impression à l'utilisateur qu'elle n'a pas été interrompue. Cette apparence d'une application qui s'exécute de façon continue est difficile à préserver et il est fréquent de voir des applications échouer cet effet.

L'API cycle de vie est organisée autour d'une classe singleton `App` et définie par le module `app`. La classe `App` reçoit les messages des systèmes d'exploitation sous la forme d'appel à des méthodes qui peuvent être redéfinies dans chaque application. Il y a un total de sept méthodes :

`on_create` L'application a été créée, c'est le temps de construire l'interface utilisateur. Cette méthode est appelée une seule fois au cours de l'exécution de l'application.

`on_resume` L'application arrive au premier plan, elle devient interactive.

`on_pause` L'application a quitté le premier plan, elle n'est plus interactive, mais elle peut être encore visible.

`on_stop` L'application est suspendue et elle n'est plus visible, elle peut être détruite à tout moment.

`on_restart` L'application n'est plus suspendue, elle revient d'un précédent `on_stop`.

`on_save_state` L'application peut bientôt être détruite, c'est le temps d'en sauvegarder l'état. Les services de l'API persistance des données, présentée à la section suivante, permettent de conserver l'état de l'application.

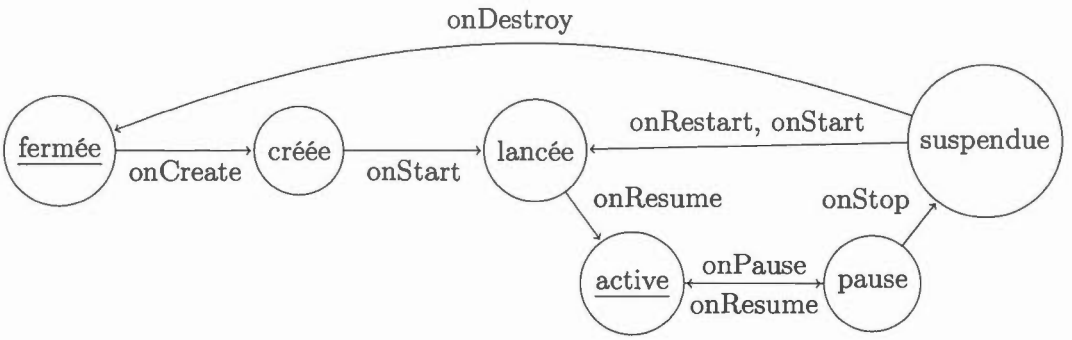
`on_restore_state` L'application est en cours de lancement, c'est le temps de charger l'état de l'application depuis une sauvegarde précédente.

Pour concevoir cet API, nous avons analysé les cycles de vie et les messages envoyés par Android et par iOS pour en abstraire les fonctionnalités communes. La figure 2.4 présente les cycles de vie des deux plateformes, ainsi que le cycle de vie retenu dans notre solution. Le cycle de vie *app.nit* reprend un nombre minimum d'états et de transitions communs à Android et iOS. Il conserve trois états communs : *fermée*, *active* et *suspendue*, en plus de l'état *inactive* qui regroupe l'état du même nom sous iOS et les états *créée* et *lancée* d'Android.

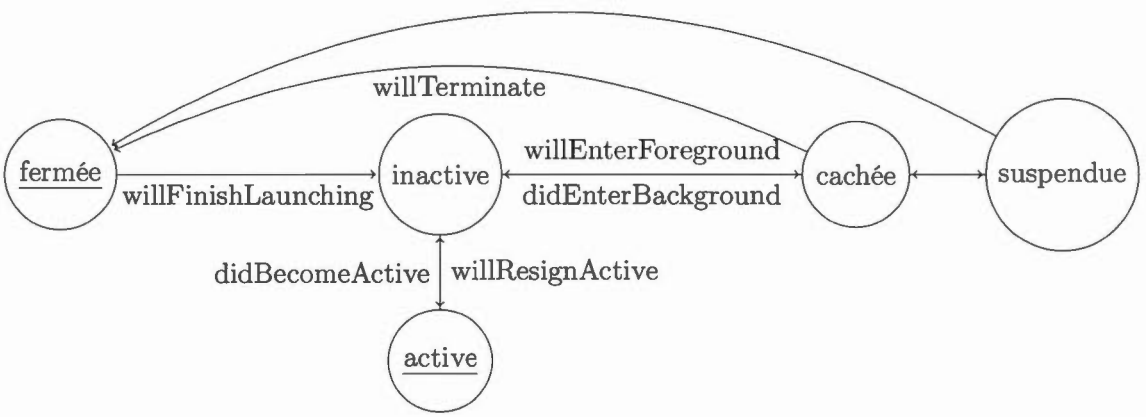
2.3.3 API persistance des données

Pour préserver l'état de l'application lorsqu'elle est interrompue et sauvegarder les préférences de l'utilisateur entre deux exécutions, la bibliothèque *app.nit* offre l'API persistance des données. Elle prend la forme d'un dictionnaire `data_store`, offert par le paquet du même nom, qui associe des chaînes de caractères à des valeurs d'un type sérialisable. Il utilise le paquet portable de sérialisation vers JSON que nous avons mentionné brièvement à la section 2.2.2, page 27.

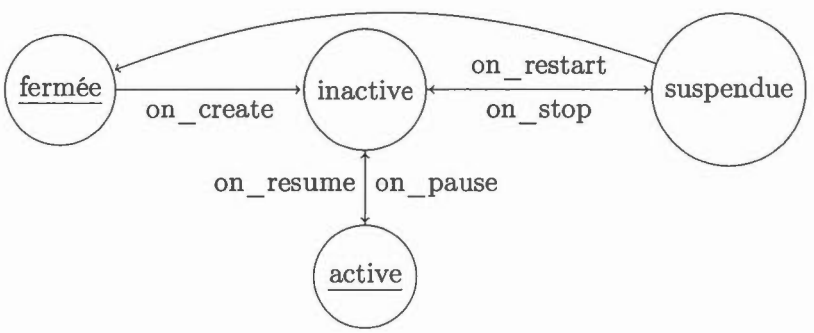
La figure 2.5 présente un exemple d'utilisation de l'API persistance des données, il s'agit d'une version simplifiée de la sauvegarde de l'état dans la calculatrice *app.nit*.



(a) Le cycle de vie d'une activité Android (AOSP, 2016c).



(b) Le cycle de vie d'une application iOS (Apple Inc., 2016). Le nom de certaines méthodes a été raccourci pour améliorer la lisibilité du graphe.



(c) Le cycle de vie d'une application *app.nit*.

Figure 2.4: Les cycles de vie des applications Android, iOS et *app.nit*.
Les arêtes représentent les transitions possibles entre deux états. Si le système d'exploitation envoie un message à l'application lors d'une transition, l'arête correspondante porte le nom de la méthode invoquée en étiquette.

```
1 redef class App
2   # Contexte incluant le calcul en cours
3   var context = new CalculatorContext is lazy
4
5   redef fun on_save_state
6   do
7     # Sauvegarde le contexte avec les chiffres entrés
8     app.data_store["context"] = context
9     super
10  end
11
12  redef fun on_restore_state
13  do
14    super
15
16    # Tentative de chargement du contexte
17    var context = app.data_store["context"]
18
19    # Vérifie si le contexte existe (si ce n'est pas
20    # la première exécution) et si il est d'un format valide.
21    if not context isa CalculatorContext then return
22
23    # Réinstalle le contexte
24    self.context = context
25
26    # Met à jour l'affichage avec le calcul en cours
27    display.text = context.display_text
28  end
29 end
```

Figure 2.5: Exemple d'utilisation de l'API cycle de vie et persistance des données.

Cette API est implémentée en utilisant des fonctionnalités natives à chaque plateforme. Elle est réalisée par `android.content.SharedPreferences` sous Android et par `NSUserDefaults` sous iOS. Malgré des API différentes, les deux services natifs répondent au même besoin de préserver des données simples dans un dictionnaire.

2.3.4 API requêtes HTTP

L'application mobile typique est connectée à un serveur distant pour déléguer la persistance des données, communiquer avec un réseau social ou télécharger des nouvelles. L'API de requêtes HTTP permet la communication avec le serveur et aide à résister aux

connexions instables.

Deux contraintes complexifient la réalisation des requêtes HTTP et la mise à jour de l'interface utilisateur selon la réponse du serveur. (i) Android et iOS, ainsi que les bonnes pratiques, imposent que les requêtes HTTP ne bloquent pas le *thread* de l'interface utilisateur (*UI thread*). (ii) Par contre, les deux plateformes restreignent les modifications à l'interface utilisateur au *UI thread* seulement. Le programmeur doit donc exécuter les requêtes sur un *thread* indépendant, mais mettre à jour l'interface utilisateur depuis la *UI thread.*

L'API de requêtes HTTP abstrait la solution au problème. L'API est centrée autour de la classe `AsyncHttpRequest` qui définit quatre méthodes qui agissent sur le *UI thread.* Le programmeur spécialise `AsyncHttpRequest` et implémente les méthodes de rappel sans avoir à se soucier de l'aspect concurrent de la requête. La classe est associée à un *thread* dédié qui exécute la requête, elle attend la réponse du serveur et déserialise les réponses au format JSON vers des objets Nit. Les appels entre les *threads*, illustrés dans la figure 2.6, sont donc gérés automatiquement de façon transparente au programmeur.

Les quatre méthodes ont différents rôles :

`before` est invoquée juste avant la communication avec le serveur. Elle peut être utilisée pour indiquer à l'utilisateur qu'une requête est en cours. Par exemple, le bouton ayant lancé la requête peut être désactivé en attendant la réponse.

`on_fail` est invoquée lorsqu'une erreur survient pendant le traitement de la requête. Il peut s'agir d'une interruption de la connexion avec le serveur, ou d'un problème de désérialisation des données reçues. Les erreurs réseau sont communes sur les appareils mobiles qui sont vulnérables à la qualité des signaux sans-fil. Par défaut, ce rappel affiche l'erreur à la console. Les erreurs côté serveur, dont le code HTTP 404, sont traitées par `on_load`, car il s'agit d'un comportement normal.

`on_load` est invoquée lorsque la communication avec le serveur et la désérialisation ont été un succès. C'est le moment de traiter les données reçues et de vérifier si la réponse n'est pas une erreur côté serveur.

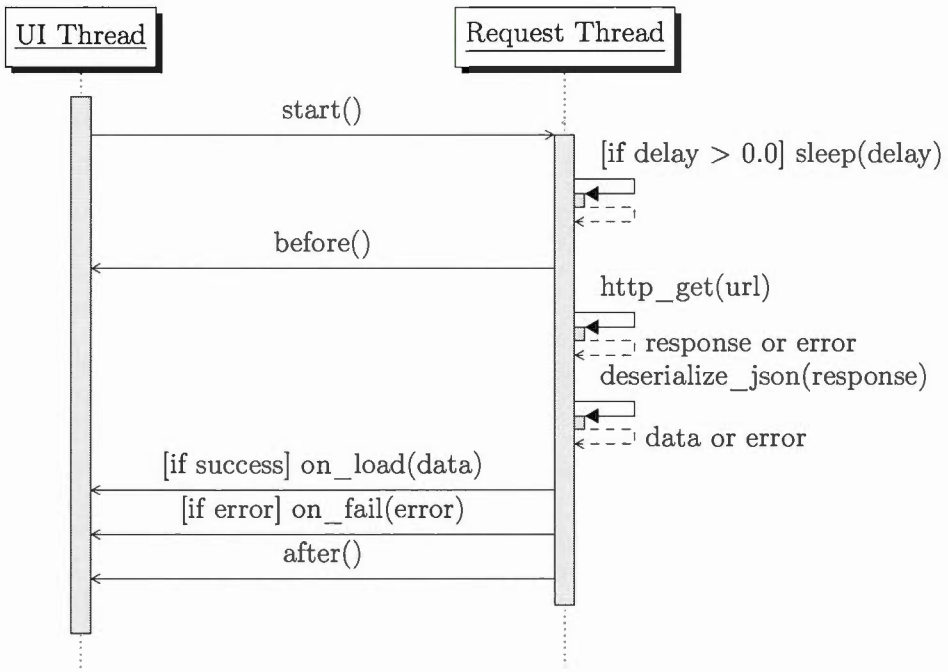


Figure 2.6: Diagramme de séquence de l'API de requêtes HTTP asynchrones.

`after` est invoquée après l'appel de `on_fail` ou `on_load`. Elle peut être utilisée pour mettre à jour l'interface utilisateur et indiquer la fin de la requête. Pour reprendre l'exemple précédent, le bouton ayant lancé la requête peut être réactivé.

En support à l'API de requêtes HTTP, nous avons implémenté le support pour les *threads* POSIX en Nit. Autant Android qu'iOS supportent ce modèle d'exécution et l'ajouter à Nit a permis de factoriser du code portable. C'était une préoccupation extérieure au sujet de cette thèse, mais ce fut tout de même un effort technique non négligeable.

2.3.4.1 Application – Client mobile Tnitter

Tnitter est un service web de microblogue, semblable à Twitter, que nous avons réalisé en Nit. Il permet à ses utilisateurs de se créer un compte et de publier de courts messages qui sont affichés sur la page d'accueil de l'interface web. [3]

3. L'interface web de Tnitter est publiée à http ://xymus.net/tnitter/.

(a) Tnitter sous Android

(b) Tnitter sous iOS

Figure 2.7: Prototype du client Tnitter sous Android et iOS.

En alternative à l'interface web, nous avons réalisé un client mobile pour Tnitter avec *app.nit.*[4] Le client mobile permet de lire les messages récents et les nouveaux messages au fur et à mesure de leur publication. La figure 2.7 en présente l'interface sous Android et iOS.

Cette expérimentation nous a servi à valider deux fonctionnalités absentes de la calculatrice. D'abord le contrôle `ListLayout` qui affiche les messages dans une liste déroulante, puis l'API de requêtes web qui est utilisée à deux endroits dans le client mobile : pour lister les messages récents, et pour recevoir des alertes du serveur lorsque de nouveaux messages sont publiés.

Lister les messages se fait par de simples requêtes où le serveur répond immédiatement au format JSON, le code source de la requête est présenté à la figure 2.8. Pour recevoir les alertes du serveur, les requêtes sont plus complexes, on applique une technique surnommée *push notification.* Le client envoie une requête au serveur et le serveur conserve

4. La source du client mobile et du serveur Tnitter est disponible dans le dépôt Nit, ou sur le web à **`https://github.com/xymus/nit/tree/these_alexis/contrib/tnitter/`**

la connexion ouverte, sans y répondre immédiatement. De cette façon, le serveur peut répondre à la requête dès qu'un message est publié par un autre utilisateur.

Les requêtes ouvertes sont un cas limite d'utilisation de l'API de requêtes HTTP. Les connexions peuvent être interrompues à tout moment selon la qualité des signaux sans-fil. Cette fonctionnalité nous a servi à valider l'API de requêtes HTTP même dans les cas plus difficiles.

2.3.5 Métadonnées

Indépendamment des API utilisées par le code de l'application, *app.nit* recueille des métadonnées pour configurer les binaires pour Android et iOS. Ces métadonnées servent aux systèmes d'exploitation pour identifier et classifier les applications. Certaines métadonnées sont communes à Android et iOS, alors que d'autres sont spécifiques à Android.

Trois annotations permettent au programmeur de déclarer les métadonnées communes : `app_name` déclare le nom de l'application affiché aux utilisateurs, `app_namespace` déclare l'identifiant unique de l'application, et `app_version` construit une chaîne de caractères pour identifier la version. La figure 2.9 présente un exemple d'utilisation de ces trois annotations et des annotations spécifiques à Android.

En ce qui concerne Android, les projets d'applications natives rassemblent toutes les métadonnées dans un fichier `Android.xml`. Ce fichier déclare, entre autres, la version de l'API native à utiliser et les permissions de sécurité demandées par l'application. Le compilateur Nit génère ce fichier et utilise des annotations supplémentaires pour recueillir les métadonnées propres à Android. Les annotations `android_api_target`, `android_api_min` et `android_api_max` définissent la version visée de l'API native et les limites de compatibilité. Les annotations `android_manifest` et `android_manifest_application` insèrent des lignes directement dans le fichier `Android.xml`

```
1 # Requête pour lister les derniers messages du serveur
2 class ListPostRequest
3   super AsyncHttpRequest
4
5   # Adresse pour la requête
6   redef fun rest_server_uri do return "http://tnitter.xymus.net/"
7   redef fun rest_action do return "rest/list?count=16"
8
9   # Le serveur a renvoyé un objet JSON qui a été désérialisé
10  redef fun on_load(posts)
11  do
12    # Traite les messages d'erreur côté serveur (404 ou autre)
13    if posts isa Error then
14      print_error "Server Error: '{posts.message}'"
15      return
16    end
17
18    # Vérification que les données sont dans le format attendu
19    if not posts isa Array[Post] then
20      print_error "Local Error: Got '{posts or else "null"}'"
21      return
22    end
23
24    # Affiche les messages sur l'interface utilisateur
25    window.apply_update posts
26  end
27
28  # Lors d'une erreur de connexion ou de désérialisation
29  redef fun on_fail(error)
30  do
31    print_error "Warning: Request failed with {error}"
32  end
33 end
```

Figure 2.8: Utilisation de `AsyncHttpRequest` dans le client Tnitter.

```
1 module tnitter_app is
2   app_name "Tnitter"
3   app_namespace "net.xymus.tnitter"
4   app_version(1, 0)
5
6   android_api_target 20
7   android_manifest """
8     <uses-permission android:name="android.permission.INTERNET" />
9   """
10 end
```

Figure 2.9: Métadonnées du client Tnitter.

2.3.6 Ressources

En plus du code exécutable et des métadonnées, les applications mobiles sont composées de ressources, telles que des images, des fichiers sonores et des fichiers textes. Le format de ces ressources dépend souvent de la plateforme et elles sont organisées différemment sous Android et iOS. Par exemple, les icônes qui identifient l'application sur l'écran d'accueil sont dans des formats incompatibles entre Android et iOS. L'icône doit être disponible dans les résolutions appropriées à chaque plateforme. Pour joindre facilement des ressources à une application tout en respectant les différences entre les plateformes, les projets *app.nit* déclarent séparément les ressources pour Android et pour iOS.

Par défaut, les icônes et les autres ressources sont recherchées dans le dossier racine du projet. Le compilateur y cherche les dossiers `android` et `ios`, leur contenu est utilisé comme base aux projets d'applications générés par le compilateur. Le programmeur peut y organiser toutes les ressources selon le format attendu par chaque plateforme.

L'annotation `app_files` indique au compilateur de rechercher les dossiers `android` et `ios` dans le dossier du module. Cette annotation peut être utilisée par des paquets de la bibliothèque pour associer des ressources à des modules et ainsi favoriser la réutilisation de ressources en plus de la réutilisation de code.

2.4 Comment compiler le prototype *app.nit* pour Android et iOS

Les projets d'applications mobiles *app.nit* sont à la base très simple. Un seul module Nit peut déclarer l'interface utilisateur et la structure de l'application. Notamment, le client mobile Tnitter est implémenté principalement par un seul module. Et il peut être compilé en application Android par un seul appel au compilateur Nit (`nitc`) :

```
nitc src/tnitter.nit -m android
```

Cette commande produit le binaire `tnitter.apk`, une archive APK, qui peut être installée sur n'importe quel appareil Android : les téléphones, les tablettes et même les montres. Pour iOS, un appel au compilateur génère un dossier `tnitter.app` prêt à être installé sur les appareils iOS :

```
nitc src/tnitter.nit -m ios
```

L'option `-m ios` (pour *mixin*) demande au compilateur d'importer le paquet `ios` de la bibliothèque de Nit. Le module `ios` et ses dépendances sont alors compilés avec le code local de l'application pour produire le binaire `tnitter.app`. L'équivalent peut être réalisé par le programmeur par un module Nit de la forme suivante :

```
1 module tnitter_ios
2
3 import tnitter # Code local
4 import ios     # Implémentation app.nit pour iOS
```

Ce module peut être étendu pour adapter davantage le prototype à iOS, cette possibilité sera discutée au prochain chapitre.

2.5 Conclusion

Les API portables permettent de réaliser un prototype fonctionnel sur Android et iOS. Le prototype est une première version de l'application qui rassemble les fonctionnalités principales du projet et qui est suffisante pour être publiée, aller chercher du financement et décider si le projet doit être poursuivi.

Les quatre API *app.nit* permettent au prototype de répondre aux besoins de l'application mobile typique. Le prototype peut offrir une interface utilisateur tactile utilisant les contrôles natifs à chaque plateforme. Il peut répondre aux messages des systèmes d'exploitation et il applique un cycle de vie qui regroupe les états communs à Android et iOS. Le prototype peut préserver son état lors d'interruptions ou entre les exécutions à l'aide de l'API persistance des données. Il peut aussi réaliser des requêtes HTTP de façon asynchrone sans bloquer l'interface utilisateur.

Au-delà du code, les métadonnées du prototype sont déclarées par des annotations et utilisées par le compilateur pour former les binaires. Et finalement, des ressources, dont l'icône de l'application, peuvent être jointes aux binaires en réutilisant la structure de fichiers des projets natifs d'Android et d'iOS.

Les API portables *app.nit* servent de fondation fiable à notre solution, il s'agit d'une approche éprouvée pour réaliser des applications portables. Par contre, restreindre les applications à du code portable seulement ne permet pas de les adapter à Android et à iOS. Le prochain chapitre présente l'organisation des projets d'applications *app.nit* en ligne de produits pour obtenir une famille d'applications semblables, chacune adaptée à une plateforme.

CHAPITRE III

LIGNE DE PRODUITS ET RAFFINEMENT DE CLASSES

La fragmentation du marché des appareils mobiles entre Android et iOS est une embûche importante au développement d'applications mobiles. Elle force les programmeurs à réaliser deux applications distinctes ou alors à construire des applications portables peu adaptées à chaque plateforme.

Notre solution *app.nit* surmonte le problème de fragmentation au-delà du prototype en facilitant l'adaptation des applications à chaque plateforme. Nous proposons d'organiser chaque projet *app.nit* en ligne de produits. Cette organisation partage une base de code commune à laquelle on applique des variations pour réaliser une famille d'applications semblables. En pratique, l'organisation en ligne de produits permet de faire évoluer graduellement le prototype en applications adaptées à Android et à iOS.

Pour réaliser les lignes de produits au niveau du code, nous utilisons le raffinement de classes du langage Nit. Le raffinement de classes permet une séparation propre des préoccupations et une détection statique des ambiguïtés entre les variations. Dans une ligne de produits, le raffinement de classes applique des variations par la réouverture de méthodes et de classes qui servent alors de points de variations.

Ce chapitre s'adresse aux utilisateurs du langage Nit et de son raffinement de classes. Nous utilisons le langage des lignes de produits pour décrire les effets du raffinement de classes et une organisation possible pour tout projet d'application en Nit.

Dans ce chapitre, nous discutons de l'intégration des lignes de produits et du raffinement de classes dans notre solution. Nous commençons par survoler le sujet des lignes de

produits et comment elles peuvent cohabiter avec le développement agile sous forme de lignes de produits agiles. Nous discutons de langages et d'outils permettant de réaliser des artéfacts de code pour lignes de produits en établissant trois axes de services à combler. Ensuite, nous présentons le raffinement de classes de Nit et comment il s'applique dans une ligne de produits. Finalement, nous illustrons les lignes de produits *app.nit* en reprenant la calculatrice introduite au chapitre précédent.

3.1 Lignes de produits

Les lignes de produits logiciels sont une technique éprouvée pour réaliser une famille d'applications semblables (les produits), de haute qualité et à faible coût (Clements et Northrop, 2002; Pohl, Böckle et van Der Linden, 2005). Une ligne de produits s'articule autour de fonctionnalités communes et de variations. Les fonctionnalités communes sont partagées par tous les produits et réalisées par des artéfacts communs : du code, des modèles et autres. Les variations sont des fonctionnalités réalisées par des artéfacts optionnels qui apportent les modifications qui différencient les produits entre eux.

Dans le cadre de ce projet de recherche, l'intérêt est au niveau des artéfacts de code. Le domaine des lignes de produits est populaire et bien documenté par le milieu scientifique, par contre la réalisation de lignes de produits avec le raffinement de classes n'a pas encore été explorée.

Dans cette section, nous survolons le processus de développement de lignes de produits et comment il peut cohabiter avec le développement agile. Ensuite, nous présentons différentes alternatives étudiées par le milieu scientifique pour réaliser des artéfacts de code.

3.1.1 Lignes de produits et développement agile

Le processus de développement classique d'une ligne de produits commence par la modélisation de toutes les fonctionnalités et les variations d'une famille de produits, pour ensuite réaliser les artéfacts nécessaires et finalement les lier en produits. L'effort initial

de modélisation est typique de la philosophie des lignes de produits qui vise le long terme et nécessite un travail important avant d'obtenir des produits fonctionnels.

Toutefois, ce processus est partiellement contradictoire avec le développement agile (Díaz et al., 2011). Ce dernier facilite la réalisation d'applications de haute qualité, à faible coût et vise une mise en marché rapide. Il repose sur un développement incrémental des fonctionnalités et l'implication du client dans le processus. La philosophie du développement agile favorise la création de valeur sur le court terme (Martin, 2003).

L'ingénierie de lignes de produits agiles (*Agile Product Line Engineering*) réconcilie les deux philosophies en rassemblant les aspects compatibles et en corrigeant les incompatibilités. La principale modification apportée au processus de développement d'une ligne de produits permet la modélisation des fonctionnalités et des variations en cours de développement, plutôt que de tout modéliser préalablement (Díaz et al., 2011).

Cette thèse propose de suivre les bonnes pratiques d'ingénierie de lignes de produits agiles pour réaliser une famille d'applications mobiles semblables, de haute qualité et à faible coût, tout en profitant d'une mise en marché rapide du prototype. Dans notre cas, les principales variations sont les adaptations à Android et à iOS, il y a donc au moins deux produits, un binaire Android et un binaire iOS. Selon les besoins d'un projet, il peut s'y ajouter d'autres produits.

3.1.2 Artéfacts de code

Les lignes de produits sont réalisées au niveau du code par des artéfacts de code, des fragments de code qui implémentent les fonctionnalités communes et les variations. Nous présentons dans cette section des langages et outils pour réaliser les artéfacts de code et les lier en produits.

Pour structurer la comparaison des différentes alternatives, nous avons établi trois axes auxquels doit répondre une solution de développement de lignes de produits : *séparation*, *dépendance* et *liaison*. Chaque axe représente une catégorie de services qui peut être réalisée de différentes façons par les langages et outils.

3.1.2.1 Axe *séparation*

Une séparation efficace des variations dans le code est centrale aux lignes de produits. De façon générale, on parle de séparation des préoccupations, une philosophie reconnue pour aider la modélisation de systèmes complexes (Dijkstra, 1976).

3.1.2.2 Axe *dépendance*

La solution doit permettre d'identifier les relations de dépendance entre les variations. Empruntant la terminologie de Cho et al. (Cho, Lee et Kang, 2008), deux formes de dépendances nous intéressent : les dépendances d'utilisation (tel que l'appel à une méthode d'une autre variation), et les dépendances de comportement (où une variation modifie un comportement introduit par une autre variation). Un point sensible de cet axe est sur les dépendances de comportement, à savoir quelle implémentation d'un même comportement a priorité sur les autres.

3.1.2.3 Axe *liaison*

Un outil permet de sélectionner des variations et de les lier en produits (ou binaires). Le programmeur sélectionne les variations par une interface qui peut prendre la forme de ligne de commande, d'un fichier de configuration ou d'une interface graphique. Cet outil est partiellement automatique, les relations de dépendance déclarent déjà certaines variations à lier.

3.1.3 Langages et outils répondant aux trois axes

Nous utilisons ici les trois axes pour analyser certains langages et outils permettant de réaliser des artéfacts de code pour des lignes de produits. Sans être une liste exhaustive, il s'agit de technologies utilisées dans le domaine mobile ou que nous considérons comme étant une alternative possible à Nit.

3.1.3.1 AspectJ

AspectJ est une extension au langage Java qui implémente la programmation orientée aspects (Kiczales et al., 1997). Elle a été utilisée à plusieurs reprises pour évaluer les lignes de produits en Java (Gacek et Anastasopoulos, 2001; Anastasopoulos et Muthig, 2004; Botterweck, Lee et Thiel, 2009; Cho, Lee et Kang, 2008).

La programmation orientée aspects vise à séparer le code par préoccupations, chacune dans un fichier distinct. Ce paradigme de programmation se base sur trois concepts : les aspects, les points de coupe et le tissage. (i) Chaque aspect rassemble le code ciblant une préoccupation qui affecte le logiciel à différents endroits. (ii) Les points de coupe sont les endroits dans le code source où est inséré le code des aspects. (iii) Le tissage est le processus qui traite le code source à l'entrée, y insère le code des aspects aux bons points de coupe et génère le code source final.

AspectJ sépare les variations en permettant d'implémenter des préoccupations transversales à plusieurs classes par des aspects distincts, chacun dans un fichier qui lui est propre. Chaque aspect est optionnel et étend des classes existantes.

Les dépendances d'utilisation sont déclarées par les importations habituelles à Java, qui sont aussi disponibles depuis un aspect. AspectJ y ajoute les dépendances de comportement par le mot `extends` utilisé sur un aspect pour lui donner priorité sur un autre.

Le tissage de AspectJ lie les variations en produits. Les aspects à lier sont sélectionnés par la ligne de commande ou listés dans un fichier XML.

Le mémoire de maîtrise *Using AspectJ to Build a Software Product Line for Mobile Devices* (Young, 2005) mérite une mention spéciale par sa proximité avec le sujet de la présente thèse. L'auteur évalue l'utilisation de AspectJ pour réaliser des lignes de produits surmontant le problème de fragmentation des API du marché des appareils mobiles. Il en conclut que AspectJ s'applique aux appareils mobiles et qu'il isole bien, dans des aspects dédiés, les dépendances aux API non portables. Toutefois, il s'agit d'une ancienne génération d'appareils mobiles, le mémoire a été publié en 2005, deux

ans avant le lancement du premier iPhone. Il visait seulement J2ME, une plateforme Java pour appareils mobiles. Il n'y avait donc pas encore les problèmes de fragmentation entre les plateformes, ni de langages natifs différents, ni les API natives distinctes qui nous intéressent dans cette thèse.

3.1.3.2 Ruby

En Ruby, les variations peuvent être séparées en fichiers. Tout fichier peut étendre des classes existantes avec des méthodes et attributs. De plus, les *mixins* agissent comme héritage multiple et, eux aussi, peuvent être ajoutés à une classe existante (Flanagan et Matsumoto, 2008).

Les relations de dépendance d'utilisation y prennent la forme d'importation, tout comme avec AspectJ, en fait, tous les langages présentés ici appliquent cette même approche. Les dépendances de comportement sont déterminées selon l'ordre de chargement du code. Si deux fichiers implémentent une même fonctionnalité, le dernier fichier chargé a préséance. Ceci peut causer des comportements imprévus et difficiles à résoudre en cas de conflits.

L'interpréteur officiel de Ruby est lancé sur un seul fichier, le point d'entrée, il ne permet donc pas de lier des variations au lancement. Par contre, les fichiers Ruby peuvent être importés de façon conditionnelle, ce qui permet de lier dynamiquement des variations supplémentaires à l'exécution.

Ruby a été utilisé pour mettre sur pied des lignes de produits qui joignent dynamiquement les variations (Günther et Sunkle, 2009; Günther et Sunkle, 2010). Dans une de ces études, l'application présente un menu graphique au lancement et l'utilisateur sélectionne les variations désirées.

3.1.3.3 Python et JavaScript

En Python et JavaScript, il est possible de réaliser des variations en modifiant le modèle du programme par métaprogrammation, une pratique surnommée *monkey patching*. De cette façon, les variations peuvent être séparées en modules qui modifient les classes et méthodes concernées par une préoccupation.

Il n'y a aucune gestion des dépendances de comportement, différentes variations peuvent entrer en conflit de façon imprévisible. Pour cette raison, cette technique est généralement non recommandée.

Tout comme Ruby, Python et JavaScript permettent l'importation dynamique de modules. Les variations peuvent donc être liées à l'exécution du logiciel.

3.1.3.4 Objective-C

Objective-C offre deux structures de langage qui facilitent la séparation des variations : les catégories et les classes d'extensions. Les catégories ajoutent des méthodes à toute classe existante, alors que les classes d'extensions ne permettent d'ajouter des méthodes et des attributs qu'à des classes compilées au même moment.

Ces deux structures ne permettent pas de modifier des comportements entre les variations, il n'y a donc pas de dépendances de comportement.

Tous les fichiers d'implémentation (les `.m`) doivent être passés en arguments au compilateur, un classique des langages de style C. Les variations à lier sont sélectionnées à ce moment.

3.1.3.5 C#

Le langage C# offre les classes partielles pour séparer les variations dans plusieurs fichiers sources (MSDN, 2016b), ainsi que les méthodes d'extension pour étendre des classes

existantes en leur ajoutant des services qui sont en réalité des fonctions statiques (MSDN, 2016a).

Tout comme Objective-C, il n'y a pas de dépendances de comportement entre les variations en C# et les variations sont liées à la compilation.

Certains outils utilisent les classes partielles pour implémenter des lignes de produits en C# (Kastner et al., 2009). Ces fonctionnalités sont toutefois limitées, car il est impossible de redéfinir une méthode, de déclarer la spécialisation d'une autre interface ou d'ajouter un attribut à une classe d'un autre espace de nom.

3.2 Raffinement de classes

Notre solution *app.nit* utilise le raffinement de classes pour réaliser les lignes de produits. Le raffinement de classes a été introduit par le langage PRM (Privat, 2002), l'ancêtre du langage Nit. Il bonifie le paradigme de programmation orientée objet en utilisant les objets pour encapsuler les données, et des modules pour séparer les préoccupations et les variations.

Le raffinement de classes permet à tout module de rouvrir des classes introduites par d'autres modules, tant qu'elles sont visibles (Ducournau, Morandat et Privat, 2008). Comme de fait, chaque module peut étendre les classes existantes en leur ajoutant des super-classes et des propriétés : attributs, méthodes et constructeurs. Un module peut aussi modifier le comportement de méthodes en redéfinissant leur implémentation, d'une façon semblable à la spécialisation de classe où les sous-classes peuvent redéfinir (*override*) l'implémentation d'une méthode introduite dans une super-classe.

Dans le code, le mot clé `redef` déclare le raffinement d'une classe et la redéfinition d'une méthode. Les exemples des figures 3.1 et 3.2 introduisent une classe et l'étendent par raffinement de classes dans plusieurs modules.

Le raffinement de classes est une fonctionnalité de premier niveau du langage Nit, au même titre que la spécialisation de classes. En fait, ces deux fonctionnalités, le raffinement et la spécialisation, partagent plusieurs caractéristiques. Le mot clé `redef` sert

```
1 # Fichier "mod1.nit"
2 module mod1
3
4 class A
5     fun foo do print "foo introduction"
6 end
7
8 var a = new A
9 a.foo         # Affiche "foo introduction"
```

(a) Introduction de la classe A et de la méthode foo.

```
1 # Fichier "mod2.nit"
2 module mod2
3
4 import mod1
5
6 redef class A
7     # Raffinement
8     redef fun foo do print "foo raffinement"
9 end
10
11 var a = new A
12 a.foo         # Affiche "foo raffinement"
```

(b) Raffinement de la classe A et de la méthode foo.

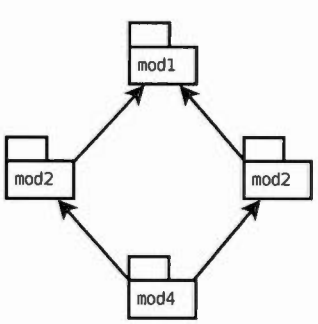
Figure 3.1: Introduction et raffinement d'une méthode.

```
1 # Fichier "mod3.nit"
2 module mod3
3
4 import mod1
5
6 class B
7     fun bar do print "bar"
8 end
9
10 redef class A
11     super B # Nouvelle super-classe
12
13     # Nouvel attribut
14     var attr = "attribut"
15
16     # Nouvelle méthode
17     fun baz do print "baz"
18 end
19
20 var a = new A
21 a.foo         # Affiche "foo introduction"
22 a.bar         # Affiche "bar"
23 print a.attr # Affiche "attribut"
24 a.baz         # Affiche "baz"
```

(a) Ajout d'une super-classe, d'un attribut et d'une méthode par raffinement.

```
1 # Fichier "mod4.nit"
2 module mod4
3
4 import mod2
5 import mod3
6
7 var a = new A
8 a.foo         # Affiche "foo raffinement"
9 a.bar         # Affiche "bar"
10 print a.attr # Affiche "attribut"
11 a.baz         # Affiche "baz"
```

(b) Combinaison des raffinements de `mod2` et `mod3`.



(c) Hiérarchie d'importation des modules.

Figure 3.2: Combinaison de plusieurs raffinements.

autant à déclarer les raffinements que la redéfinition de méthodes par spécialisation. De plus, la hiérarchie d'importation des modules est une structure d'ordre partiel qui permet le raffinement multiple, tout comme la hiérarchie d'héritage des classes qui, en Nit, permet l'héritage multiple.

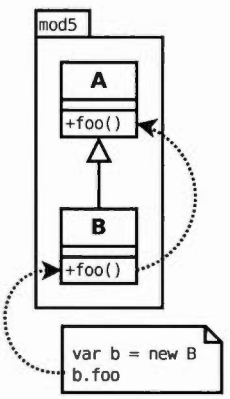
Du point de vue des lignes de produits, la caractéristique commune la plus importante est l'ordre de linéarisation qui combine le raffinement et la spécialisation. L'ordre de linéarisation détermine la priorité d'exécution des multiples implémentations d'une même méthode. C'est-à-dire, quelle implémentation doit être exécutée lors de l'invocation de la méthode et lors d'un appel à super. L'ordre est d'abord déterminé par les relations de spécialisation, les implémentations plus précises (déclarées par les sous-classes) ont priorité sur les implémentations générales (déclarées par les super-classes). L'ordre considère ensuite les implémentations ajoutées par raffinement, des modifications transversales à la spécialisation modifiant autant les sous-classes que les super-classes. Les implémentations déclarées par un module plus bas dans la hiérarchie d'importation (par raffinement) ont priorité sur les implémentations déclarées plus haut (dont l'introduction de la méthode).

Autrement dit, la spécialisation a priorité sur le raffinement. L'ordre de linéarisation priorise les implémentations des sous-classes et ensuite les implémentations déclarées par les modules au bas de la hiérarchie d'importation. La figure 3.3 présente l'ordre de linéarisation d'une méthode en cas de spécialisation et de raffinement.

Il y a ambiguïté entre deux raffinements lorsqu'il n'y a pas de priorité entre les deux implémentations d'une même méthode, c'est-à-dire que leur ordre d'exécution ne peut pas être déterminé par le compilateur. Un tel cas survient dans l'exemple de la figure 3.4 où deux modules, `mod8` et `mod9`, modifient la même méthode `foo`. Le module `mod10` importe `mod8` et `mod9`, et il invoque la méthode `foo`. Il y a alors ambiguïté sur quelle implémentation est à exécuter, celle de `mod8` ou celle de `mod9`. Les deux modules ne sont pas en relation directe d'importation, l'ordre de linéarisation ne peut pas déterminer quel module et quelle implémentation ont priorité. Le compilateur Nit détecte cette ambiguïté à la compilation et la rapporte au programmeur sous forme d'un avertissement sur le

```
1 module mod5
2
3 class A
4     fun foo do print "mod5:A"
5 end
6
7 class B
8     super A
9
10    # Spécialisation
11    redef fun foo do
12        printn "mod5:B "
13        super
14    end
15 end
16
17 var b = new B
18 b.foo # Affiche "mod5:B mod5:A"
```

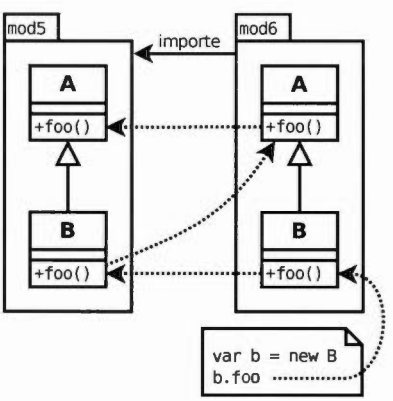
(a) Spécialisation de classes.



(b) Ordre de linéarisation avec spécialisation seulement.

```
1 module mod6
2 import mod5
3
4 redef class A
5     # Raffinement
6     redef fun foo do
7         printn "mod6:A "
8         super
9     end
10 end
11
12 redef class B
13    # Raffinement
14    redef fun foo do
15        printn "mod6:B "
16        super
17    end
18 end
19
20 var b = new B
21 b.foo # "mod6:B mod5:B mod6:A mod5:A"
```

(c) Raffinement de classes.

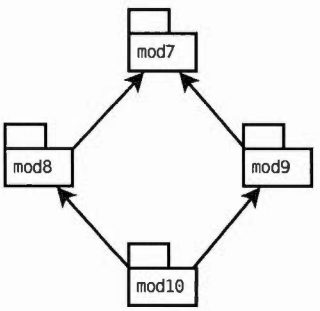


(d) Ordre de linéarisation avec spécialisation et raffinement.

Figure 3.3: Ordre de linéarisation en cas de spécialisation et de raffinement. Les lignes pointillées, dans (b) et (d), identifient l'ordre d'exécution des multiples implémentations de la méthode `foo` lors de son invocation et des appels à super.

```
1 module mod7
2
3 class A
4     fun foo do print "mod7"
5 end
```

(a) Introduction de la méthode `foo`.



(b) Ordre d'importation des modules.

```
1 module mod8
2
3 import mod7
4
5 redef class A
6     redef fun foo do print "mod8"
7 end
```

(c) Modification de la méthode `foo`.

```
1 module mod9
2
3 import mod7
4
5 redef class A
6     redef fun foo do print "mod9"
7 end
```

(d) Autre modification de la méthode `foo`.

```
1 module mod10
2
3 import mod8
4 import mod9
5
6 var a = new A
7 a.foo
```

(e) Combinaison des raffinement et appel ambigu à `foo`.

```
1 $ nitc mod10.nit
2 mod10.nit:7,3--5: Warning: conflicting property definitions for
      property `foo` in `A`: mod8$A$foo mod9$A$foo
3   a.foo
4     ^
```

(f) Appel au compilateur Nit et avertissement affiché au programmeur.

Figure 3.4: Cas d'ambiguïté entre deux raffinements d'une même méthode.

site d'appel. Le programmeur peut alors insérer une importation additionnelle pour déterminer l'ordre de linéarisation.

Selon la terminologie des sections précédentes, les variations d'une ligne de produits peuvent être réalisées par raffinement de classe et les dépendances de comportements sont représentées par l'ordre de linéarisation. Alors que le programmeur n'a pas un contrôle direct sur l'ordre de linéarisation, il peut l'affecter selon la hiérarchie de classes et les importations de modules. L'ordre de linéarisation est calculé à la compilation et les ambiguïtés sur la priorité entre deux implémentations d'une même méthode sont rapportées au programmeur selon le site d'appel. L'ordre de linéarisation et la détection statique des ambiguïtés est une force du raffinement de classes qui assure ainsi une dépendance de comportement prévisible et fiable. De plus, le binaire résultant peut être optimisé à la compilation car le résultat du raffinement au niveau du modèle est connu.

Un moteur d'exécution Nit, dont le compilateur officiel *nitc*, est le seul outil nécessaire pour lier les variations en produits, ou lier les raffinements apportés par différents modules en une application. De façon classique, *nitc* est lancé sur un module, le point d'entrée, dont il compile toutes les dépendances selon les modules importés. L'option `-m` (utilisée au chapitre précédent) permet d'y lier des modules supplémentaires. Le programmeur peut utiliser cette option pour lier les variations désirées à la compilation d'un produit.

Reprenant l'exemple de Tnitter, la commande `nitc src/tnitter.nit -m ios` lie le prototype avec le module `ios`, la variation pour iOS des API *app.nit*. Cette variation modifie les services abstraits par raffinement de classes pour utiliser les services natifs à iOS. Le module `android` est une variation équivalente qui implémente les services abstraits des API *app.nit* pour Android par raffinement.

Les modules `ios` et `android` utilisent l'annotation `platform` pour indiquer au compilateur comment compiler l'application. Nous présenterons l'annotation `platform` et ses effets plus en détails au chapitre suivant.

Alors que l'option `-m ios` suffit à compiler le prototype en application fonctionnelle, une ligne de produits pour un projet réel bénéficie d'adaptations supplémentaires pour

chaque plateforme. Le raffinement de classes permet d'ajouter des variations adaptant davantage l'application aux deux plateformes. La calculatrice *app.nit*, présentée à la section suivante, bénéficie de ce genre de variation. Au chapitre suivant, nous présenterons un outil pour perfectionner les adaptations, la programmation polyglotte via la FFI de Nit, qui peut agir en combinaison avec le raffinement de classes.

Cette thèse est la première étude à appliquer le raffinement de classes pour réaliser une ligne de produits. Toutefois, une étude précédente (Ducournau, Morandat et Privat, 2008) avait comparé les fonctionnalités du raffinement par rapport aux autres solutions de classes ouvertes, dont certaines sont utilisées pour réaliser des lignes de produits. La conclusion générale était que le raffinement de classes apportait une meilleure modélisation des classes et des propriétés en modules, tout en étant appuyé par une syntaxe simple et une vérification statique.

3.3 Calculatrice – la ligne de produits

Pour illustrer une ligne de produits réalisée avec le raffinement de classes, réutilisons notre projet de calculatrice. Alors que le chapitre précédent n'en présentait que le prototype, la calculatrice *app.nit* est en réalité une ligne de produits avec plusieurs variations. Cette organisation permet non seulement de surmonter la fragmentation entre les plateformes, mais aussi la fragmentation interne à la plateforme Android.

Dans cette section, nous classifions les fonctionnalités de la calculatrice en trois catégories : les fonctionnalités obligatoires qui sont communes à tous les produits, les variations d'adaptation aux plateformes, et une variation optionnelle ajoutant des opérations scientifiques. Les fonctionnalités sont également présentées sous forme de diagramme de fonctionnalités à la figure 3.5. Finalement, nous présentons les produits qui en résultent.

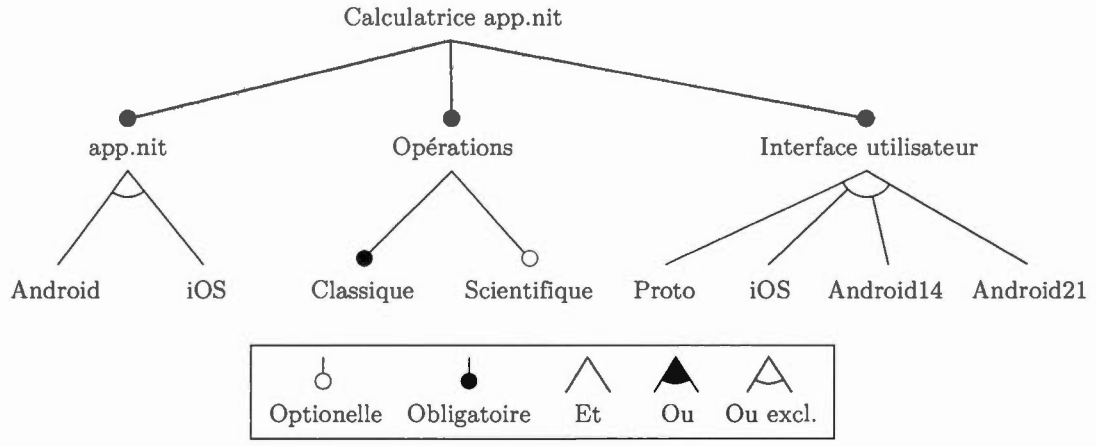


Figure 3.5: Diagramme de fonctionnalités de la calculatrice *app.nit*.

3.3.1 Fonctionnalités obligatoires

Les fonctionnalités obligatoires, présentes dans tous les produits, sont définies dans deux modules qui sont au sommet de la hiérarchie d'importation présentée dans la figure 3.6.

— Le module `calculator_logic` implémente la logique d'affaires et n'utilise que des services portables de la bibliothèque standard de Nit. Le fonctionnement de ce module est aisément vérifié par des tests en boîte noire.

— Le module `calculator` importe `calculator_logic` et définit l'interface utilisateur en faisant appel aux services de l'API UI de *app.nit*. De plus, il utilise l'API cycle de vie et persistance pour préserver le calcul en cours.

Dans ce projet de recherche, nous considérons que les fonctionnalités obligatoires composent le prototype. Le module `calculator` peut être compilé en binaire lorsque joint avec une variation de *app.nit* pour Android ou iOS. La commande suivante génère le binaire iOS du prototype sous forme du fichier `calculator.app` :

```
nitc src/calculator.nit -m ios
```

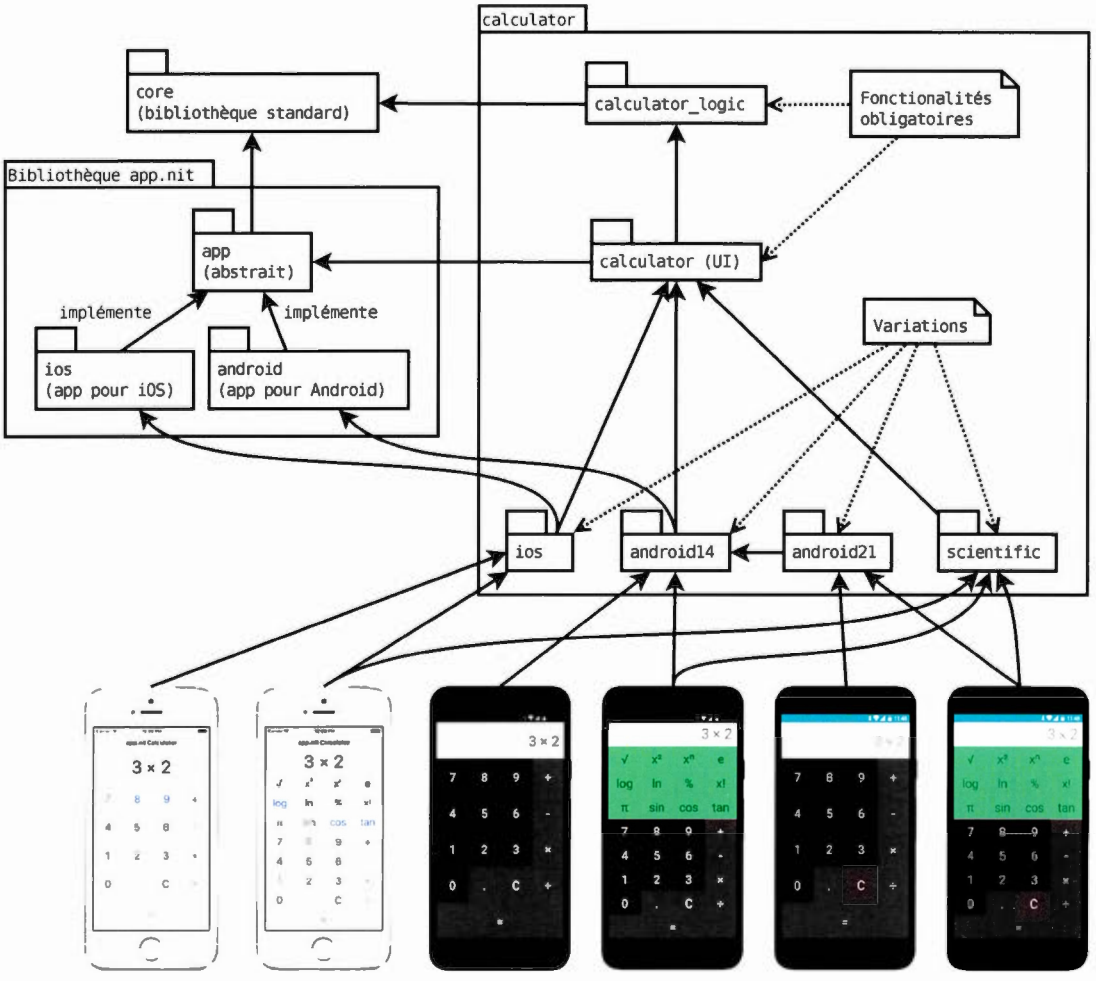


Figure 3.6: Dépendances entre les modules et produits de la calculatrice *app.nit*. La bibliothèque *app.nit* et ses paquets sont illustrés par la figure 2.1 à la page 30.

3.3.2 Variations d'adaptation aux plateformes

La calculatrice est adaptée à Android et à iOS par des variations à deux niveaux : dans la bibliothèque *app.nit* et localement dans la ligne de produits.

Les variations de la bibliothèque *app.nit* implémentent les API portables différemment pour chaque plateforme et permettent de compiler le prototype en binaires Android ou iOS. Ce sont des variations obligatoires ; pour produire un binaire viable, la calculatrice doit être jointe soit avec la variation Android, soit avec la variation iOS de *app.nit*.

Parallèlement, la ligne de produits définit localement trois variations optionnelles pour adapter davantage l'application à iOS et à différentes versions de Android. Il s'agit uniquement de variations esthétiques de l'interface utilisateur.

- La variation pour iOS applique quatre changements au prototype : (i) elle assigne un titre à la fenêtre, (ii) elle grossit le texte affiché, (iii) elle centre le contenu de l'entrée de texte en haut de la fenêtre et (iv) elle distribue les boutons pour qu'ils remplissent la fenêtre. Cette adaptation est réalisée en Nit pur, son code est présenté dans la figure 3.7. Le produit résultant de cette variation est présenté dans la figure 3.8.b, avec le prototype sous iOS.

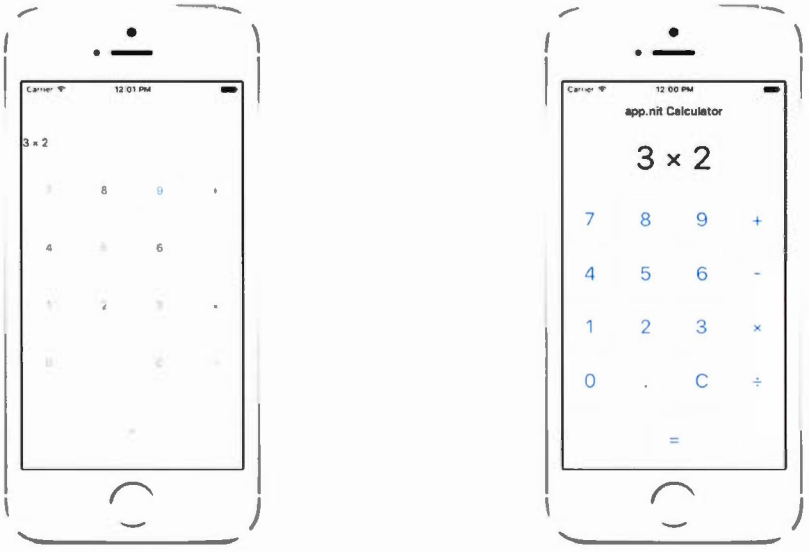
- La variation principale pour Android utilise la classe `Button` de l'API UI comme point de variation pour reproduire le style *matériel* recommandé sur Android. Le résultat est une interface épurée qui ressemble à la calculatrice Android publiée parmi le projet libre *Android Open Source Project* (AOSP).[1] En fait, le thème de couleur, défini par un fichier XML, provient directement de la base de code AOSP. Le produit résultant de cette variation est présenté par la figure 3.9.b, sur la droite du prototype portable.

---

1. Le code source et les autres ressources de la calculatrice AOSP utilisée comme référence sont archivés à `https://github.com/android/platform_packages_apps_calculator/tree/71d5b97`

```
1 # Fichier "calculator/src/ios.nit"
2 module ios
3
4 import calculator # Le prototype
5 import ::ios      # L'implémentation iOS de app.nit
6
7 redef class CalculatorWindow
8     # (i) Titre de la fenêtre
9     init do title = "app.nit Calculator"
10 end
11
12 redef class Button
13     # (ii) Grossit la police des boutons
14     init do size = 2.5
15 end
16
17 redef class TextInput
18     # (iii) Grossit et centre le texte de l'entrée de texte
19     init do
20         size = 5.0
21         align = 0.5
22     end
23 end
24
25 redef class VerticalLayout
26     # (iv) Ajuste la distribution des boutons
27     redef init do native.distribution = new
28         UIStackViewDistribution.fill_proportionally
29 end
```

Figure 3.7: Source annotée de l'adaptation iOS de la calculatrice *app.nit*.



(a) Prototype sous iOS

(b) Calculatrice adaptée à iOS

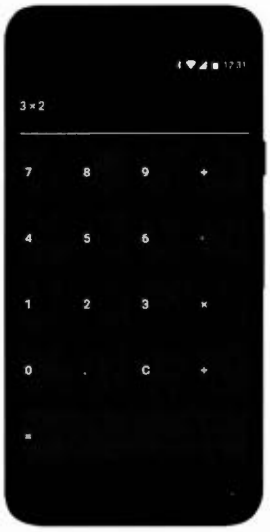
Figure 3.8: Prototype et calculatrice adaptée à iOS.

— Une variation supplémentaire vise spécifiquement les appareils Android 5.0 *Lollipop* (API 21)[2] et plus récent. Elle utilise deux fonctionnalités introduites par l'API 21 : le thème *matériel* officiel de Android qui offre un style constant avec le reste de l'écosystème, et un service désactivant le clavier virtuel pour éviter qu'il duplique les contrôles à l'écran. Le produit résultant de cette variation est présenté par la figure 3.9.c, sur la gauche de la calculatrice AOSP qui a servi d'inspiration stylistique.

L'étude de cas de la ligne de produits Tenenit, présentée au chapitre 6, développe davantage les adaptations à chaque plateforme. Ses adaptations dépassent l'esthétisme et ajoutent des fonctionnalités distinctes selon la plateforme, ce qui démontre davantage la force des lignes de produits pour surmonter la fragmentation du marché des appareils mobiles.

---

2. Le projet Android utilise trois versions parallèles : la version commerciale de Android (tel que 5.0), le nom (*Lollipop*) qui n'est pas toujours arrimé à une version commerciale majeure, et la version de l'API (21).



(a) Prototype *app.nit* sous Android.

(b) Calculatrice *app.nit* adaptée à Android.

(c) Calculatrice *app.nit* adaptée à l'API 21. Comparée à (b), la police est différente et la barre en haut de l'écran est colorée.

(d) Calculatrice AOSP pour API 21. La barre turquoise sur la droite active le mode scientifique.

Figure 3.9: Calculatrice *app.nit* et AOSP sous Android.

3.3.3 Variation optionnelle

Une variation perpendiculaire aux plateformes ajoute des fonctionnalités optionnelles à l'interface utilisateur. Le module `scientific` étend la calculatrice classique en calculatrice scientifique avec des opérations trigonométriques, logarithmiques et autres. Étant optionnel, ce module peut être lié à la compilation (ou non) pour former différents produits. Cette variation est indépendante mais compatible avec les variations d'adaptation à Android et à iOS. Les produits en résultant sont présentés dans la figure 3.10 et comparés avec la calculatrice AOSP.

Il est à noter que le mode scientifique de notre calculatrice est lié à la compilation seulement. C'est une différence avec la calculatrice AOSP où le mode scientifique est activé à l'exécution en tapant sur la barre colorée.

3.3.4 Produits

Les produits du projet de la calculatrice sont les binaires destinés aux appareils mobiles. Il y a un total de six produits, soit le résultat du produit cartésien entre les trois plateformes (iOS, Android et Android API 21) et les deux modes (classique et scientifique). On ne considère pas les prototypes comme faisant partie des produits car ils sont incomplets par définition. Les six produits sont rassemblés dans la figure 3.11.

Chaque produit est généré par le compilateur Nit. Par exemple, la commande suivante compile la calculatrice scientifique adaptée à Android *Lollipop* et produit un binaire Android `calculator.apk` :

```
nitc src/calculator.nit -m src/scientific -m src/android21
```

3.3.5 Travaux connexes

Les auteurs d'au moins trois autres études ont utilisé des calculatrices comme exemple de ligne de produits.

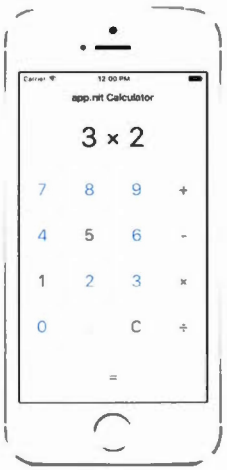
(a) Prototype *app.nit* scientifique.

(b) Calculatrice *app.nit* adaptée à Android.

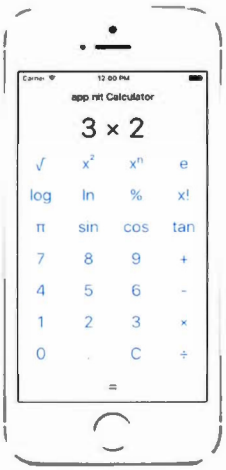
(c) Calculatrice *app.nit* adaptée à l'API 21. La variation scientifique est liée à la compilation avec l'option `-m`.

(d) Calculatrice AOSP en mode scientifique. Le mode scientifique est activé pendant l'exécution par l'utilisateur.

Figure 3.10: Calculatrice scientifique *app.nit* et AOSP sous Android.



(a) Classique pour iOS.



(b) Scientifique pour iOS.

(c) Classique pour Android.

(d) Scientifique pour Android.

(e) Classique pour Android API 21.

(f) Scientifique pour Android API 21.

Figure 3.11: Les six produits de la calculatrice *app.nit*.

— Certains utilisent AspectJ pour réusiner une calculatrice Java en ligne de produits (Botterweck, Lee et Thiel, 2009). Le résultat est semblable à notre ligne de produits, il y a des opérations optionnelles et des fonctionnalités qui bonifient l'interface utilisateur.

— D'autres utilisent une calculatrice réalisée avec AspectJ pour illustrer le besoin de dépendances de comportement entre les variations (Cho, Lee et Kang, 2008). En Nit, ce besoin est comblé par les relations d'importation entre les modules et l'ordre de linéarisation.

— Finalement, des auteurs ont réalisé une calculatrice en programmation orientée fonctionnalité avec Ruby et le *framework rbFeatures* (Günther et Sunkle, 2009). Leur ligne de produits affiche un menu pour sélectionner les variations (des opérations arithmétiques) à lier au lancement de la calculatrice. Les variations sont appliquées par des importations de modules conditionnelles. Le résultat est limité : si une opération n'est pas liée, le bouton associé est désactivé, mais toujours visible.

3.4 Conclusion

Les lignes de produits sont une technique éprouvée pour réaliser une famille d'applications semblables, de haute qualité et à faible coût. L'organisation des projets d'application *app.nit* en ligne de produits profite de ces avantages pour surmonter la fragmentation du marché des appareils mobiles et faire évoluer graduellement le prototype en applications adaptées à Android et à iOS. Combiné au raffinement de classes de Nit, cette organisation facilite la maintenance par une séparation propre des préoccupations dans le code. Les variations dédiées à chaque plateforme peuvent être développées de façon incrémentale pour améliorer l'apparence de l'application portable et s'approcher de l'apparence native.

Nous avons démontré en réalisant la calculatrice en ligne de produits que le raffinement de classes est une solution viable pour implémenter une ligne de produits. Toutefois, nous avons tenu sous silence certains détails de l'adaptation aux plateformes. Les adaptations

dépendent d'un accès entier aux API natives, les API portables sont généralement insuffisantes pour atteindre le niveau de qualité des applications natives. Le chapitre suivant présente les outils nécessaires pour accéder aux API natives tout en profitant de l'expertise préalable.

Il est à noter que malgré les adaptations aux plateformes, le prototype n'est pas perdu, il côtoie les variations d'adaptation à Android et à iOS. Le prototype peut aider au débogage, car il est plus simple que les adaptations. Il peut servir de version minimale pour une plateforme qui n'est pas prioritaire, par exemple pour produire un binaire GNU/Linux sans investir dans l'adaptation à la plateforme. De plus, il peut servir à prototyper de nouvelles fonctionnalités.

CHAPITRE IV

PROGRAMMATION POLYGLOTTE ET L'INTERFACE DE FONCTIONS ÉTRANGÈRES DE NIT

Une application native à Android ou à iOS s'intègre à l'écosystème de la plateforme en utilisant les services des API natives qui offrent une interface utilisateur ainsi que d'autres fonctionnalités propres à chaque plateforme. Cela dit, atteindre le niveau de qualité des applications natives est un des grands défis dans le développement d'applications portables qui visent le besoin à court terme de portabilité à l'encontre de la qualité et de l'évolution à long terme de l'application.

Pour réconcilier le développement portable et l'atteinte du niveau de qualité des applications natives, nous avons choisi de tirer profit de la programmation polyglotte, un concept général selon lequel une application est réalisée en plusieurs langages de programmation. Cette approche nous permet de conserver notre prototype portable réalisé en Nit pur et d'y ajouter des variations qui sont partiellement implémentées dans les langages natifs à Android et iOS. Ceci ouvre l'accès aux API natives de chaque plateforme dans leur entièreté et facilite le transfert de connaissances d'experts.

La programmation polyglotte joue deux rôles dans notre solution. Pour les programmeurs, elle sert d'alternative aux API portables lorsque celles-ci sont limitées. Pour nous, en tant que développeur de la solution, elle permet d'implémenter les API de *app.nit* pour Android et pour iOS.

Nous nous laissons guider par un principe simple : *en programmation polyglotte, utiliser chaque langage selon ses spécialités.* Alors que Nit permet de réaliser du code portable

et de structurer des applications polyglottes, les langages natifs à Android et iOS sont spécialisés pour les plateformes. Ainsi l'accès aux API natives via les langages natifs permet aux applications portables de s'intégrer à l'écosystème de la plateforme, et d'utiliser ses services natifs, son interface utilisateur et ses fonctionnalités propres.

Les langages de programmation assistent la programmation polyglotte en offrant des interfaces natives ou des interfaces de fonctions étrangères (FFI). Les interfaces natives permettent l'interaction avec du code natif écrit dans le langage sous-jacent au moteur d'exécution. En comparaison, les FFI permettent l'interaction avec d'autres langages, sans se limiter aux langages natifs. Même s'il s'agit de deux concepts semblables, on peut considérer les interfaces natives comme étant un sous-ensemble des FFI.

Dans ce chapitre, nous comparerons régulièrement notre solution à la la FFI du langage Java, la *Java Native Interface* (JNI) (Liang, 1997). La JNI est représentative de l'offre du milieu et ses forces et faiblesses sont bien documentées par la littérature scientifique. La JNI sert d'interface entre Java et les langages natifs tels que C et C++. En pratique, elle permet au code Java d'invoquer des fonctions de bibliothèques natives chargées dynamiquement. De plus, la JNI offre des API en C et en C++ pour manipuler la JVM depuis le code étranger et ainsi, invoquer du code Java depuis le code étranger. L'API C prend une forme de métaprogrammation qui permet de contourner plusieurs restrictions de sûreté du langage Java, donnant un grand pouvoir au programmeur avec lequel viennent de grandes responsabilités. En fait, la JNI est reconnue comme étant difficile d'utilisation et propice aux erreurs (Bubak, Kurzyniec et Luszczek, 2000; Kondoh et Onodera, 2008; Tan et al., 2006).

Pour intégrer la programmation polyglotte aux lignes de produits dans les projets *app.nit*, nous avons conçu et réalisé une FFI originale pour le langage Nit. Cette FFI, à laquelle nous référerons dorénavant simplement comme étant la FFI de Nit, se définit par la combinaison de quatre caractéristiques principales. (i) La FFI de Nit s'intègre au paradigme de programmation orienté objet en offrant une forme alternative aux objets, classes, méthodes et appels de méthodes. (ii) Elle imbrique le code étranger parmi le code Nit pour améliorer la lisibilité du code. (iii) Elle accepte plusieurs langages étrangers,

C, Objective-C, Java et C++, et profite de leurs aspects communs pour une meilleure expressivité. (iv) Elle génère sur mesure les services pour le code étranger de façon à préserver la sûreté statique des langages étrangers.

La FFI de Nit remplace l'interface native de Nit, en y ajoutant la syntaxe imbriquée et le support de plusieurs langages étrangers. Elle en conserve les méthodes et classes externes ainsi que les services générés sur mesure. L'interface native séparait le code d'un même module en plusieurs fichiers distincts, approximativement un fichier par langage. Cette organisation est réutilisée à l'interne comme phase intermédiaire avant d'invoquer les compilateurs de chaque langage étranger. Alors que ce document présente certains concepts introduits par l'interface native de Nit qui sont communs avec la FFI de Nit, le lecteur peut se référer à notre mémoire (Laferrière, 2012) pour une comparaison de chaque fonctionnalité avec les alternatives disponibles.

Dans ce chapitre, nous commençons par présenter un aperçu de haut niveau de la FFI de Nit et nous plongeons ensuite graduellement dans les aspects techniques de la FFI. Les premières sections visent d'abord les créateurs de langages et justifient les choix de conception de la FFI de Nit. Elles peuvent guider l'implémentation d'une FFI similaire dans tout autre langage. Les dernières sections du chapitre servent avant tout de manuel de référence pour les utilisateurs de la FFI de Nit. Elles présentent les détails pertinents pour le programmeur du support de chaque langage en plus de donner un aperçu de leur fonctionnement interne.

Ce chapitre est composé de huit sections :

1. Une présentation des critères de qualité qui ont guidé la conception de la FFI de Nit. Il s'agit des qualités visées par le langage Nit sous l'influence supplémentaire de la programmation polyglotte.
2. Une description générale des caractéristiques de la FFI de Nit, l'intégration au paradigme orienté objet, la syntaxe imbriquée, les multiples langages étrangers et les services générés sur mesure.

3. Une présentation générale des fonctionnalités communes à tous les langages étrangers supportés par la FFI de Nit. Cette section est centrale à la compréhension du chapitre, elle sert également d'introduction à la FFI de Nit pour les programmeurs.
4. La structure modulaire que prend l'implémentation de la FFI de Nit dans les engins d'exécution. Cette section peut servir de guide à la réalisation d'autres FFI qui supporteraient plusieurs langages étrangers et plusieurs moteurs d'exécution.
5. Une présentation de la FFI avec C, son rôle dans ce projet de recherche, ainsi que l'application des fonctionnalités communes et de ses fonctionnalités propres.
6. Une présentation de la FFI avec Objective-C, son rôle dans le support de la plateforme iOS, ainsi que l'application des fonctionnalités communes et de ses fonctionnalités uniques.
7. Une présentation de la FFI avec Java, son rôle dans le support de la plateforme Android, ainsi que l'application des fonctionnalités communes et de ses fonctionnalités uniques. Cette section commence par une comparaison avec les supports pour C et Objective-C.
8. La conclusion ramène la programmation polyglotte et la FFI de Nit dans le contexte du développement d'applications mobiles portables et discute de son rôle dans notre solution.

4.1 Critères de qualité d'une FFI pour le langage Nit

La FFI de Nit est le mécanisme que nous avons ajouté au langage Nit pour réaliser des applications polyglottes. Étant une partie intégrale du langage, la FFI priorise les mêmes six qualités que le langage : la facilité d'apprentissage, l'expressivité, la lisibilité, la souplesse, l'assistance et l'agrément (Privat, 2006). Celles-ci sont toutefois influencées par l'aspect fondamentalement polyglotte d'une FFI.

La facilité d'apprentissage d'un programmeur détermine la vitesse à laquelle un programmeur maîtrise la FFI, qu'il soit débutant ou avancé, et ce, en Nit ou dans un langage étranger. La FFI doit prendre une forme qui facilite le transfert des connaissances en Nit et en langages étrangers vers la programmation polyglotte.

L'expressivité détermine la capacité du programmeur à s'exprimer succinctement et directement avec la FFI, le langage Nit et les langages étrangers. La FFI doit s'intégrer aux langages en préservant leur expressivité.

La lisibilité du code source affecte la difficulté et la vitesse de lecture du programmeur. La FFI doit favoriser une uniformité entre ses services en évitant l'utilisation de mots clés et de symboles avec une signification différente entre les langages.

La souplesse de la FFI aide à l'évolution, la maintenance et la réutilisation du code polyglotte.

L'assistance qu'offre la FFI au programmeur pour prévenir les erreurs est favorisée par une détection, à la compilation, des erreurs d'utilisation de la FFI et des langages étrangers.

L'agrément est le niveau auquel la FFI est plaisante à utiliser pour le programmeur. Sans être très objectif, on peut évaluer ce critère par la tendance du programmeur à accepter d'utiliser la FFI, un phénomène rare chez les autres langages.

4.2 Caractéristiques de la FFI de Nit

Les six critères de qualité sont comblés par les quatre principales caractéristiques de la FFI de Nit. (i) Elle s'intègre aux concepts fondamentaux du paradigme orienté objet. (ii) Elle utilise une syntaxe imbriquée et étend la grammaire de Nit. (iii) Elle supporte plusieurs langages qui ont chacun leurs spécialités tout en profitant de leurs similarités. (iv) Elle génère sur mesure des services pour le code étranger qui sont, autant que possible, uniformes entre les langages.

4.2.1 Intégration au paradigme orienté objet

La FFI s'intègre au paradigme orienté objet en Nit en offrant une forme alternative à ses concepts fondamentaux : les objets, les classes, les méthodes et les envois de messages.

— Des objets externes agissent comme références aux données d'un langage étranger depuis le code Nit. Ils peuvent être manipulés naturellement depuis le code Nit, ce qui contribue à l'expressivité de la FFI.

— Des classes externes agissent comme catégories pour associer les objets externes à des types des langages étrangers. Elles supportent l'héritage multiple et servent de type statique aux variables et signatures Nit.
— Des méthodes externes sont implémentées dans un langage étranger. Elles peuvent être invoquées naturellement depuis Nit et elles sont organisées parmi les classes Nit (que celles-ci soient externes ou non).
— Des fonctions de rappel générées sur mesure permettent, depuis le code étranger, l'envoi de messages à des méthodes Nit, l'accès à des attributs d'objets Nit, l'appel à des constructeurs Nit ainsi que la vérification et la conversion des types.

Ces fonctionnalités servent de fondement à la FFI de Nit, nous les revisiterons en plus amples détails à la section 4.3.

L'extension aux concepts du paradigme orienté objet contribue à l'apprentissage de la FFI pour les programmeurs qui maîtrisent déjà le paradigme. Ils doivent uniquement apprendre les formes alternatives de chaque concept et ils peuvent appliquer leurs connaissances préalables en bonne pratique de programmation orientée objet.

4.2.2 Syntaxe imbriquée

La FFI étend la grammaire de Nit pour imbriquer le code étranger parmi le code Nit. Le code étranger est délimité par une structure grammaticale généraliste qui permet aussi d'identifier le langage utilisé. Cette structure sert, entre autres, à implémenter les méthodes externes et à associer des types étrangers aux classes externes.

La structure syntaxique est délimitée par un bloc `` `{ `} `` qui, dans le cas des méthodes externes, est une alternative au **do end** des méthodes ordinaires Nit. La structure `` `{ `} `` est suggérée par la littérature scientifique, les solutions comme Janet (Bubak, Kurzyniec et Luszczek, 2000) et Jeannie (Hirzel et Grimm, 2007) étendent la grammaire de Java avec cette structure pour y imbriquer du code C. Tout en étant distincte, cette structure syntaxique évoque les blocs de style C `{ }` et les fermetures d'Objective-C `^{ ^}`.

La syntaxe imbriquée favorise la lisibilité du code en conservant tout le code d'une préoccupation dans un seul fichier. L'alternative étant de séparer une préoccupation en au moins deux fichiers, un par langage. En comparaison, la syntaxe imbriquée contribue à la souplesse du code en limitant la duplication du code et en laissant le compilateur générer le *code colle* répétitif.

En plus de Janet et Jeannie, des FFI imbriquées existent dans d'autres langages, notamment C++ qui peut imbriquer du code C, et C lui-même qui permet d'imbriquer du code assembleur.

4.2.3 Plusieurs langages étrangers

La FFI de Nit supporte quatre langages étrangers : C, Objective-C, Java et C++. Chacun de ces langages a ses spécialités :

- Le langage C est près de la machine, il permet un contrôle précis du processeur et de la mémoire. Il a accès à plusieurs API de bibliothèques natives dont l'API C de POSIX qui est portable entre Android et iOS.
- Objective-C étend le langage C en y ajoutant le paradigme à objets. Il est utilisé dans l'implémentation de Darwin, le système d'exploitation sous-jacent à iOS et à macOS, de plus, il a accès à leurs API natives.
- Java est un langage à objets qui est généralement compilé en code octet et exécuté par une machine virtuelle. L'API native à Android prend la forme d'une bibliothèque de classes Java, et le système Android est accompagné de deux machines virtuelles, Dalvik et ART.
- C++ est un langage à objets qui est près de la machine. Certaines bibliothèques natives exposent des API en C++, dont LLVM.[1] Nous discutons peu de C++ dans cette thèse, car nous ne l'utilisons pas dans le contexte des applications mobiles.

1. LLVM est un *framework* de compilation dont l'API principal est en C++. L'accès à cet API a motivé la réalisation de la FFI avec C++.

Malgré leurs différentes spécialités, ces quatre langages sont généralistes et ils partagent des caractéristiques importantes : des fonctions semblables, des structures de données comparables, le typage statique et la compatibilité avec C.

— Les fonctions des différents langages sont d'une même famille, partageant des caractéristiques communes : le passage des arguments par référence, une seule valeur de retour, et le nom des paramètres est ignoré à l'appel. Ces caractéristiques sont utilisées pour structurer la transition entre les langages par les méthodes externes et les rappels à Nit.

— Les modèles de données comparables permettent d'associer les types Nit à des types des langages étrangers et d'en assurer le passage entre les langages. De plus, la forme semblable des pointeurs et des références à des objets permet de manipuler les objets externes depuis Nit.

— Le typage statique offert par chaque langage est utilisé pour préserver leur sûreté statique au travers de la FFI. La technique du typage statique signifie que les variables, les attributs et les signatures sont annotés par des types statiques qui contrôlent les objets acceptés. Le typage statique contribue à la sûreté statique de chaque langage et à l'assistance au programmeur en permettant la détection d'erreurs dans la manipulation de données à la compilation, même lors d'appels polyglottes.

— Finalement, les quatre langages sont compatibles avec C, directement ou via une FFI. La FFI de Nit utilise C comme langage intermédiaire entre le code Nit et chacun des langages étrangers, et ce, via les services propres à chaque langage.

Le support de plusieurs langages contribue avant tout à l'expressivité de la FFI, le programmeur ayant le choix entre Nit et quatre langages étrangers pour implémenter chaque méthode. Sélectionner un langage spécialisé pour le travail peut grandement simplifier la tâche du programmeur et lui permettre d'exprimer ses idées de façon optimale.

4.2.4 Services générés sur mesure

Le code étranger peut invoquer des méthodes Nit (ce que nous appelons des *rappels à Nit*) et accéder à d'autres services via des fonctions générées sur mesure. La FFI génère des fonctions naturelles aux langages étrangers pour donner un accès sur mesure à chaque service demandé par le programmeur. Les fonctions générées portent un nom conçu pour être prévisible, et des types statiques précis sont utilisés pour typer les arguments et le retour.

Le résultat est une API sur mesure selon les besoins de chaque module. En comparaison, la JNI offre seulement une API généraliste forçant le programmeur à réaliser les rappels à Java via une forme de métaprogrammation. Cette approche alourdit chaque rappel et perd la sûreté statique des langages natifs.

La génération des rappels et services sur mesure contribue avant tout à l'assistance offerte au programmeur. En générant des fonctions statiques pour le code étranger, les compilateurs étrangers peuvent rapporter toute erreur de manipulation au programmeur. Notamment, les erreurs de types sur les arguments ou les erreurs dans le nom des méthodes sont rapportées à la compilation. De plus, l'utilisation d'un format d'appel de fonction naturel aux langages étrangers facilite l'apprentissage et la lisibilité de la FFI.

4.3 Fonctionnalités communes de la FFI de Nit

Les quatre caractéristiques de la FFI de Nit présentées à la section précédente sont combinées dans la forme que prend la FFI dans le code. Les fonctionnalités centrales à la FFI de Nit sont communes à tous les langages supportés : les méthodes externes, la correspondance des types, les classes externes, les blocs d'entêtes, les rappels à Nit, la gestion de la mémoire et les services de vérification et conversion de types.

Dans cette section, nous présentons les fonctionnalités communes en détail, puis nous les visiterons à nouveau selon les spécificités de chaque langage dans les sections dédiées à Objective-C et Java. Cette section et les suivantes sont plus techniques et peuvent servir de manuel à l'utilisation de la FFI de Nit.

4.3.1 Méthodes externes

Le besoin fondamental du programmeur qui utilise la FFI de Nit est d'invoquer du code dans un langage étranger depuis du code Nit. La figure suivante illustre le rôle de la FFI pour répondre à ce besoin.

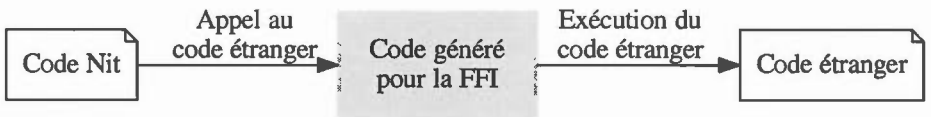


Figure 4.1: Invocation de code étranger depuis le code Nit.

Ce besoin est comblé par les méthodes externes, des méthodes Nit déclarées avec une signature Nit, mais implémentées dans un langage étranger. Elles sont invoquées comme toutes autres méthodes Nit, elles respectent entièrement le polymorphisme de Nit et elles peuvent être redéfinies normalement par une sous-classe ou un autre module.

Par exemple, la méthode externe suivante `print_c` est implémentée en C et affiche l'argument `message` via un appel à la fonction C `printf`.

```
1 fun print_c(message: CString) `{
2   printf("%s\n", message);
3 `}
```

La méthode `print_c` est invoquée naturellement depuis le code Nit.

```
1 print_c "Bonjour C!".to_cstring
```

Par défaut, le code étranger est traité comme étant du C. Au besoin, le langage peut être déclaré explicitement par l'ajout de `in "C"`, `in "ObjC"` ou `in "Java"` à la fin de la signature et avant le corps de la méthode.

Par exemple, la méthode externe `print_java` déclare une implémentation en Java et affiche `message` via les services Java classiques.

```
1 fun print_java(message: JavaString) in "Java" `{
2   System.out.println(message);
3 `}
4
5 print_java "Bonjour Java!".to_java_string
```

Les méthodes externes se comportent comme toute autre méthode Nit. Elles peuvent avoir un opérateur comme identifiant, tel que +. Leur signature peut contenir un nombre variable d'arguments (*vararg*) qui sont passés sous la forme d'un tableau Nit au code étranger.

La forme imbriquée des méthodes externes agit avant tout sur l'agrément à utiliser la FFI de Nit. Une méthode externe simple, implémentée en C, Objective-C ou Java, est aussi rapide à écrire que son équivalent en Nit. Même s'il ne s'agit que d'une anecdote, nous avons observé que les membres de notre groupe de recherche apprécient cette forme au point de parfois en abuser et implémenter des méthodes en code étranger alors que Nit aurait répondu aux mêmes besoins. La forme imbriquée encourage la réalisation d'une première méthode externe, réduisant la barrière ou la difficulté du premier pas en programmation polyglotte et facilitant ainsi l'apprentissage de la FFI.

Les arguments des méthodes externes sont passés aux implémentations en code étranger. Dans les deux exemples précédents, les arguments typés par `CString` et `JavaString` sont utilisés dans les langages étrangers en tant que `char*` et `java.lang.String`, respectivement. Les types étrangers sont déterminés par la FFI selon les règles de correspondance des types entre Nit et le langage étranger, discutées à la section suivante.

4.3.2 Correspondance des types

En plus de la simple invocation de code étranger, le programmeur a besoin de passer des données entre les langages : les arguments des méthodes externes et les valeurs de retour.

Pour permettre aux données Nit de passer aux langages étrangers et d'y être manipulées naturellement, la FFI de Nit associe chaque type Nit avec un type étranger. Cette

association sert à assigner un type statique et prévisible dans le code étranger aux données qui entrent et sortent du code Nit. De plus, elle permet de préserver la sûreté statique des langages au travers des appels polyglottes. Le tableau 4.1 donne un aperçu des types C, Objective-C et Java associés à différents types Nit.

Nous présentons ici les trois règles générales qui guident l'association des types. Toutefois, l'association précise dépend du langage étranger, donc nous en discuterons davantage dans les sections dédiées à C, Objective-C et Java.

— Les types primitifs de Nit (`Int`, `Float`, `Byte`, `Bool`, etc.) ont une association directe avec les types primitifs des langages étrangers. Ces valeurs peuvent être utilisées directement dans le code étranger.

  Par contre, tous les types primitifs des langages étrangers n'ont pas nécessairement d'équivalent en Nit. En général, le programmeur peut les convertir en un type compatible depuis le code étranger, ou alors, ils peuvent être représentés par des classes externes.

— Les classes externes, discutées en plus amples détails à la section suivante, permettent au programmeur de déclarer une association directe (mais opaque) à un type d'un langage étranger. Lorsque le type étranger est compatible avec un langage, il est utilisé comme type statique dans le code étranger. Dans ce cas, le type devrait être naturel au code étranger et la valeur peut être manipulée naturellement.

— Les autres types, tels que les interfaces, les classes ordinaires et les types nullables, sont associés à des types étrangers générés sur mesure servant de référence opaque à des objets Nit. Ils ne permettent pas au code étranger d'accéder directement aux données. Le code étranger doit utiliser les fonctions de rappel, elles aussi générées sur mesure, pour invoquer des méthodes ou accéder à des attributs des références opaques, nous les présentons à la section 4.3.5.

Les types étrangers générés portent un nom prévisible que nous qualifions de « nom plat ». Le nom plat est composé du nom de la classe, préfixé de `nullable_` (s'il est nullable) et suivi du nom plat des arguments formels (si le type est générique). Le

Tableau 4.1: Correspondance des types entre Nit, C, Objective-C et Java.

Type Nit	en C	en Objective-C	en Java	Note
`Int`	`long`	`long`	`long`	Type primitif
`Float`	`double`	`double`	`double`	Type primitif
`Byte`	`char`	`char`	`byte`	Type primitif
`Bool`	`int`	`int`	`boolean`	Type primitif
`Object`	`Object`	`Object`	`int` (opaque)	Classe ordinaire
`String`	`String`	`String`	`int` (opaque)	Classe ordinaire
`Comparable`	`Comparable`	`Comparable`	`int` (opaque)	Interface
`Array[String]`	`Array_of_String`	`Array_of_String`	`int` (opaque)	Type paramétré
`nullable String`	`nullable_String`	`nullable_String`	`int` (opaque)	Type nullable
`nullable Int`	`nullable_Int`	`nullable_Int`	`int` (opaque)	Type primitif nullable
`Pointer`	`void*`	`void*`	`int` (opaque)	Classe externe C
`CString`	`char*`	`char*`	`int` (opaque)	Classe externe C
`NSObject`	`void*`	`NSObject*`	`int` (opaque)	Classe externe Objective-C
`NSString`	`void*`	`NSString*`	`int` (opaque)	Classe externe Objective-C
`JavaObject`	`jobject`	`jobject`	`java.lang.Object`	Classe externe Java
`JavaString`	`jobject`	`jobject`	`java.lang.String`	Classe externe Java

Les règles par langage seront discutées aux sections dédiées à chaque langage.

tableau 4.2 associe des exemples de types Nit aux types statiques générés. Le nom plat est aussi utilisé pour composer le nom des fonctions de rappel.

Malgré les différences d'association entre les langages, le nom des types statiques générés reste le même entre les langages. Cette similarité favorise l'apprentissage de la FFI et le transfert de connaissances d'un langage étranger à l'autre.

4.3.3 Classes externes

En plus de manipuler des données Nit depuis le code étranger, le programmeur a besoin de l'inverse, manipuler des données étrangères depuis le code Nit. Les classes externes répondent à ce besoin, il s'agit de classes Nit spéciales qui sont associées à un type étranger : un pointeur C, C++ ou Objective-C, ou encore une classe ou un tableau Java. Chacune de leurs instances, que nous qualifions d'objets externes, est une référence à une valeur ou à un objet d'un langage étranger.

Par exemple, la classe externe `CString` est associée à un pointeur d'une chaîne de caractères C, `char*` :

```
1 extern class CString `{ char * `}
2 end
```

Tout comme les méthodes externes, les classes externes sont associées à un langage étranger, celui-ci étant C par défaut. Les classes `JavaObject` et `JavaString` sont associées respectivement aux objets et aux chaînes de caractères Java.

```
1 extern class JavaObject in "Java" `{ java.lang.Object `}
2 end
3
4 extern class JavaString in "Java" `{ java.lang.String `}
5   super JavaObject
6 end
```

La déclaration du type étranger correspondant est facultative ; une classe externe qui ne déclare pas de type étranger hérite du type associé à ses super-classes.

Une particularité des classes externes est qu'elles sont toutes implicitement sous-classes de la classe externe `Pointer`. C'est-à-dire que `Pointer` est au somment de la hiérarchie

Tableau 4.2: Exemples de types opaques générés par la FFI de Nit.

Type Nit	Type généré (Nom plat)	Note
`String`	`String`	Une chaîne de caractères Nit, une classe ordinaire.
`nullable String`	`nullable_String`	Un type nullable, toujours opaque.
`Array[String]`	`Array_of_String`	Un type générique doit être paramétré.
`Map[Int, String]`	`Map_of_Int_String`	Un type générique avec plusieurs paramètres.
`nullable Array[String]`	`nullable_Array_of_String`	Les noms plats peuvent devenir longs.

des classes externes, comme `Object` est à la racine de la hiérarchie des classes Nit. Cette politique renforce l'obligation des classes externes d'être associées uniquement à des pointeurs, ou encore à des références à un objet étranger.

Une instance d'une classe externe peut pointer vers une valeur nulle du langage étranger. Comme de fait, chaque classe externe correspond à un type étranger, donc si le type étranger accepte les valeurs nulles, l'instance peut pointer vers une valeur nulle. Cette fonctionnalité est indépendante des types nullables de Nit qui n'acceptent les valeurs nulles que lorsqu'explicitement déclarés comme tels (Gélinas, J.S., Gagnon, E. et Privat, J., 2009). Les deux fonctionnalités sont compatibles, une variable typée par `nullable CString` peut être nulle en Nit ou en C, ces deux états sont indépendants.

Le reste de cette section présente en détail l'intégration des classes externes au système d'héritage de Nit ainsi que le support des méthodes et constructeurs dans les classes externes.

4.3.3.1 Spécialisation

On remarque dans le dernier exemple que la classe externe `JavaString` spécialise `JavaObject`. Comme de fait, les classes externes peuvent spécialiser d'autres classes, mais uniquement si elles sont externes ou si elles sont des interfaces. Elles ne peuvent pas spécialiser des classes ordinaires ou des énumérations. Cette restriction provient de la différence entre les quatre grandes catégories de classes du langage Nit, qui se différencient par ce qui définit l'état de leurs instances :

- Les *interfaces* ne définissent que des méthodes et n'ont pas d'attributs. Seules les interfaces peuvent être spécialisées par d'autres catégories de classes.
- Les *classes ordinaires* encapsulent des attributs. L'état de leurs instances se compose de l'état des attributs.
- Les *classes externes* correspondent à un type étranger en Nit. Leurs instances ont comme seul état l'adresse pointée, et par extension, celui des valeurs pointées.

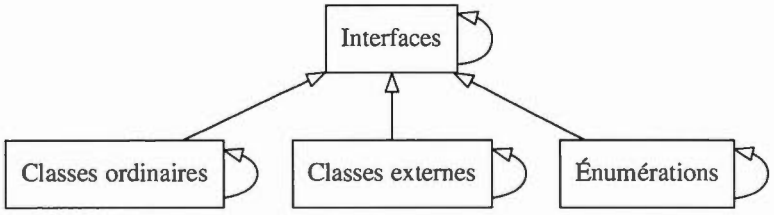


Figure 4.2: Relations de spécialisation possibles entre les catégories de classes Nit. Cette figure est modifiée depuis une version similaire présentée dans notre mémoire de maîtrise (Laferrière, 2012).

— Les *énumérations*, ou les types primitifs, ne définissent pas d'attributs. Leurs instances ont comme seul état leur valeur même.

À l'exception des interfaces, les états des trois autres catégories de classes sont incompatibles. Pour cette raison, les classes de différentes catégories ne peuvent pas se spécialiser entre elles. Les relations de spécialisation possibles sont illustrées par la figure 4.2.

Une attention particulière doit être portée aux relations de spécialisation entre deux classes externes. Les types associés à une classe et sa sous-classe doivent être compatibles et cohérents. Il n'y a pas de vérification statique sur la compatibilité des types externes par le compilateur Nit. Il y aura un comportement non défini si une instance est utilisée sous forme de pointeurs incompatibles. Par contre, les ambiguïtés sur le type étranger d'une classe sont rapportées. Ces ambiguïtés peuvent survenir si une classe externe ne déclare pas de type associé et qu'elle spécialise deux classes externes avec des types étrangers différents.

La spécialisation des classes externes, tout comme la spécialisation des autres classes Nit, favorise la factorisation du code et ainsi contribue à la souplesse de la FFI. Elle permet également de représenter fidèlement des classes étrangères en Nit en reproduisant leurs relations de spécialisation en Nit.

4.3.3.2 Méthodes

Les classes externes peuvent déclarer des méthodes, qu'elles soient externes ou non. Par exemple, on peut étendre la classe externe `JavaString`, présentée plus haut, avec les méthodes `length` et `info` :

```
1 extern class JavaString in "Java" `{ java.lang.String `}
2   super JavaObject
3
4   # Méthode externe implémentée en Java
5   fun length: Int in "Java" `{ return self.length(); `}
6
7   # Méthode ordinaire implémentée en Nit
8   fun info: String do return "Length: {length}"
```

La méthode externe `length` utilise le receveur `self`. Il s'agit du receveur de la méthode Nit qui est passé au code étranger comme un argument spécial. Dans l'exemple précédent, le receveur est une classe externe, `self` prend alors le type externe associé, `java.lang.String`.

En fait, toute classe (incluant les externes) peut déclarer des méthodes externes implémentées dans n'importe quel langage étranger. La classe `JavaString` pourrait définir une méthode en C et une autre en Objective-C.

4.3.3.3 Contructeurs

Les classes externes peuvent aussi déclarer des constructeurs externes qui prennent la forme de *factory*, des fonctions statiques sans receveur qui retournent une instance de la classe. Les *factory* sont parfaites pour réaliser un appel aux constructeurs des langages étrangers.

Par exemple, `JavaString` peut être étendu avec deux constructeurs externes, un anonyme et un nommé `copy` :[2]

14 ———————————

2. Les constructeurs nommés permettent d'accéder aux constructeurs surchargés statiquement de Java, une fonctionnalité qui n'a pas d'équivalent direct en Nit.

```
 9   # Constructeur anonyme, retournant une chaîne vide
10   new in "Java" `{ return new java.lang.String(); `}
11
12   # Constructeur nommé, copiant la chaîne `source`
13   new copy(source: JavaString) in "Java" `{
15     return new java.lang.String(source);
16   `}
```

Les instances des classes externes ne sont créées que lors de l'entrée d'un objet étranger dans le code Nit. Cette entrée peut se faire par le retour d'un constructeur externe, le retour d'une méthode externe ou encore un argument passé à un rappel à Nit (présenté à la section 4.3.5). Lors de l'entrée d'un objet étranger dans le code Nit, le système crée une nouvelle instance d'une classe externe pour encapsuler une référence à l'objet. L'instance portera le type qui annote l'entrée dans le code Nit : le receveur d'un appel à un constructeur externe, le type de retour d'une méthode externe ou le type de l'argument d'un rappel à Nit. L'instance agit alors d'une façon semblable à l'instance d'une classe ordinaire, elle conserve une référence à l'objet étranger tout comme l'instance ordinaire conserve les références aux attributs, et toutes deux portent des informations de types.

Il est important de noter que les objets qui entrent dans le code Nit, lorsque le type statique à l'entrée est une classe externe, se feront toujours assigner comme type dynamique le type statique de l'entrée. En fait, l'objet étranger a un type dynamique dans le code étranger, mais celui-ci n'est pas considéré par le système Nit.

4.3.3.4 Attributs

Les classes externes ne peuvent pas déclarer d'attributs. Par contre, elles peuvent offrir des méthodes externes agissant comme accesseurs et mutateurs des attributs de l'objet étranger.

4.3.4 Blocs d'entête

Le code étranger doit souvent être accompagné d'importations ou de fonctions de support. Les blocs d'entête répondent à ce besoin en permettant d'associer des fragments

de code étranger à un module Nit. Les blocs utilisent la structure imbriquée en entête du module, ils doivent être déclarés en dessous des importations, mais avant les classes et le reste du code.

Ils sont utilisés différemment pour chaque langage étranger, nous en discuterons plus en détail dans les sections dédiées au support de chaque langage. Encore une fois, le langage par défaut est C, mais il peut être précisé. De plus, un module peut déclarer plus d'un bloc :

```
1 `{
2   #include <stdio.h>
3 `}
4
5 in "Java" `{
6   import java.util.Log;
7 `}
```

Les blocs d'entête contribuent à la lisibilité d'un module utilisant la FFI en plaçant les importations des langages étrangers juste en dessous des importations Nit. De plus, ils permettent de déclarer des services de support au code étranger, tels que des fonctions et macros pour factoriser le code.

4.3.5 Rappels à Nit

Finalement, en plus d'invoquer du code étranger depuis Nit, le programmeur a besoin d'invoquer du code Nit depuis le code étranger. Nous utilisons l'expression « rappel à Nit » pour identifier ce besoin. Le rôle de la FFI dans les rappels à Nit est illustré par la figure suivante.

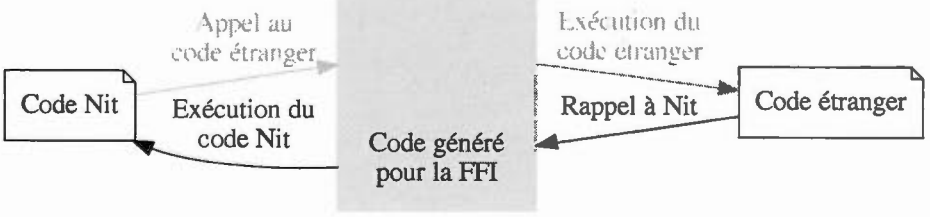


Figure 4.3: Invocation de code Nit depuis le code étranger.

Pour répondre à ce besoin, la FFI de Nit génère sur mesure des fonctions pour le code étranger. Chaque fonction représente un service précis, que ce soit une méthode, un constructeur ou un appel à super.

Les rappels doivent être déclarés par la méthode externe d'où ils sont exécutés. Cette information indique les fonctions à générer et permet au compilateur de préserver un graphe d'appel. Le graphe d'appel est notamment utilisé pour optimiser le logiciel à la compilation (Privat et Ducournau, 2005; Ducournau et al., 2009).

Une syntaxe spéciale permet leur déclaration après la signature et avant le corps d'une méthode externe. Il y a trois variantes pour les différentes sortes de rappels : l'appel à une méthode, à un constructeur et à super.

Le rappel à une méthode est déclaré en Nit par le nom court d'une méthode locale ou le type statique du receveur suivi du nom d'une méthode d'une autre classe. La fonction de rappel générée pour le langage étranger porte un nom composé du nom plat du type statique suivi d'une version textuelle de l'identifiant de la méthode. Le tableau 4.3 associe des déclarations de rappels aux fonctions de rappel générées.

Par exemple, la figure 4.4 présente une méthode externe en Objective-C qui réalise deux rappels à Nit à chaque tour de boucle. Les rappels sont déclarés par `import iter, NSString.to_s`.

Deux fonctions de rappel sont générées pour le code étranger, elles appliquent la correspondance des types d'Objective-C.

— Le rappel à la méthode `to_s` de `NSString` convertit une chaîne Objective-C en chaîne Nit. Son seul argument est la chaîne Objective-C, il retourne la chaîne Nit comme une référence opaque `String` :

```
String NSString_to_s(NSString *self);
```

— Le rappel à la méthode locale `iter` accepte le receveur (une référence opaque), un entier et une chaîne Nit (une autre référence opaque) :

```
void MaClasse_iter(MaClasse self, long i, String message);
```

Tableau 4.3: Correspondance des déclarations et des fonctions de rappel générées.

Déclaration en Nit	Nom de la fonction générée	Remarque
`foo`	`UneClasse_foo`	Rappel local à la méthode `foo`, déclaré depuis la classe fictive `UneClasse`.
`UneClasse.foo`	`UneClasse_foo`	Rappel à la méthode `foo` depuis une autre classe.
`attr`	`UneClasse_attr`	Accesseur local à l'attribut `attr`.
`attr=`	`UneClasse_attr__assign`	Mutateur local de l'attribut `attr`.
`UneClasse`	`new_UneClasse`	Appel au constructeur anonyme de `UneClasse`.
`UneClasse.bar`	`new_UneClasse_bar`	Appel au constructeur nommé `bar` de `UneClasse`.
`Int.+`	`Int__plus`	Appel à l'opérateur +.
`Int.-`	`Int__minus`	Les opérateurs portent un nom textuel.
`Array[String].add`	`Array_of_String_add`	Appel à une méthode sur un type paramétré.
`Array[String].[]`	`Array_of_String__index`	Accès indexé à un tableau.
`Array[String].[]=`	`Array_of_String__index_assign`	Assignation à un index dans un tableau.
`Ref[Int].item=`	`Ref_of_Int_item__assign`	Assignation d'un attribut dans un type paramétré.
`Map[String, Array[Int]].[]`	`Map_of_String_Array_of_Int__index`	Les types complexes entraînent une signature complexe. Il est préférable d'écrire ce code en Nit.
`super`	`UneClasse_foo__super`	Appel à super depuis la méthode `foo` de `UneClasse`.

```
1 class MaClasse
2
3   # Boucle implémentée en Objective-C
4   fun ma_boucle_objc import iter, NSString.to_s in "ObjC" `{
5     for (long i = 0; i < 4; i ++) {
6       // Rappel à NSString::to_s
7       String message = NSString_to_s(@"Message Objective-C");
8
9       // Rappel à MaClasse::iter
10      MaClasse_iter(self, i, message);
11    }
12  `}
13
14  # Méthode ordinaire invoquée par `ma_boucle_objc`
15  private fun iter(i: Int, message: String) do print "{i}: {message}"
16 end
17
18 var c = new MaClasse
19 c.ma_boucle_objc
```

Figure 4.4: Exemple de rappels à Nit depuis une méthode externe en Objective-C.

Les rappels à une méthode exécutent un envoi de message standard de Nit. Alors que les fonctions sont générées selon un type statique pour le receveur, le type dynamique du receveur est utilisé pour déterminer l'implémentation à invoquer par liaison tardive habituelle. La seule exception est lorsque le receveur est statiquement une classe externe, à ce moment, le type statique détermine l'implémentation à exécuter. Cette limitation provient du fait que, dans un tel cas, le receveur est dans un type naturel au langage étranger et non pas un type généré. À l'exécution, le système Nit n'a accès à aucune information de type plus précise que le type statique.

Un rappel à un constructeur alloue la mémoire pour l'objet à créer et l'initialise normalement. Il est déclaré en Nit par la classe à construire suivie, au besoin, du nom du constructeur. La fonction générée porte un nom composé de la chaîne `new_` suivie du nom plat du type à construire et du nom du constructeur lorsqu'il n'est pas anonyme.

Par exemple, pour créer une instance de `CString` via les services de Nit, de façon à ce que les données soient sous la responsabilité du ramasse-miettes, l'importation est simplement `CString`, et la fonction suivante est générée :

```
char* new_CString(long byte_length);
```

Un rappel à super exécute l'implémentation précédente de la méthode selon l'ordre de linéarisation de Nit résultant de l'application de la spécialisation et du raffinement de classes. Le rappel est déclaré par le mot clé `super`. Le nom de la fonction générée est composé du nom du type statique du receveur, une version textuelle de l'identifiant de la méthode suivi de `__super`. La fonction reprend tous les paramètres de la méthode.

Par exemple, pour réaliser un appel à super depuis la méthode `ma_boucle_objc` présentée plus haut, la fonction suivante est générée :

```
MaClasse_ma_boucle_objc__super(MaClasse self);
```

Le nom des fonctions générées est conçu de façon à éviter les collisions si les classes et les méthodes Nit respectent le style recommandé. C'est à dire, les noms des classes sont en *CamelCase*, et ceux des propriétés en *snake_case*. Dans tous les cas, une collision est possible si une fonction du même nom est définie dans le code étranger. La FFI ne détecte pas automatiquement les collisions et elle n'offre pas directement de services pour les éviter. Toutefois, il est généralement possible de changer le nom des entités Nit pour éviter les collisions, ou alors d'ajouter des méthodes d'indirection pour accéder à un service en particulier.

Il y a une certaine duplication de l'information entre la déclaration du rappel en Nit et l'appel à la fonction générée dans le code étranger. Toutefois, nous considérons que ce coût est contrebalancé par l'assistance apportée par le typage statique des fonctions générées. De plus, la forme naturelle des appels aux fonctions générées contribue à la lisibilité du code.

Une alternative serait de remplacer les appels aux fonctions générées par une syntaxe spéciale imbriquée dans le code étranger (Hirzel et Grimm, 2007). En pratique, la syntaxe de la déclaration du rappel, tel que `NSString::to_s`, remplacerait le nom de la fonction dans le code étranger. Par contre, cette forme ne supporte pas certains cas limites, comme un rappel à Nit depuis un bloc d'entête qui doit être déclaré depuis la méthode externe causant le rappel.

4.3.6 Gestion de la mémoire et ramasse-miettes de Nit

Le compilateur Nit utilise le ramasse-miettes conservateur Boehm-Demes-Weiser (Boehm et Weiser, 1988) pour gérer automatiquement l'espace mémoire où sont conservés les objets Nit. En fait, seulement les objets Nit et certaines instances de `CString` sont sous la responsabilité du ramasse-miettes de Nit.

Les instances de classes étrangères ne sont donc pas automatiquement gérées par le ramasse-miettes de Nit. Par contre, elles peuvent être sous la responsabilité d'un ramasse-miettes ou un autre service automatisé du langage étranger. Pour ces raisons, la gestion de la mémoire demeure une des difficultés importantes en programmation polyglotte, même avec la FFI de Nit, et elle requiert une attention particulière du programmeur. Dans le reste de ce chapitre et le prochain, lorsque pertinent, nous discuterons de différentes facettes de la gestion de la mémoire et nous décrirons alors des solutions pratiques.

Dans cette section, nous présentons d'abord le compteur de référence de la FFI de Nit qui permet de préserver des références à des objets Nit dans le code étranger. Ensuite, nous présenterons les services de finalisation pour ajouter un comportement à la libération d'un objet Nit.

4.3.6.1 Compteur de référence

Lorsque les objets Nit sont manipulés depuis le code étranger, le ramasse-miettes de Nit ne peut pas automatiquement détecter s'ils sont encore référencés. Il risque donc de libérer des objets qui sont encore utilisés depuis le code étranger. Pour prévenir cette situation, le programmeur peut explicitement déclarer l'utilisation d'une référence à un objet Nit à l'aide du compteur de référence de la FFI.

Deux fonctions permettent d'incrémenter et de décrémenter un compteur de référence. Elles sont générées sur mesure selon les types utilisés par les méthodes externes et les rappels. Offrir deux fonctions spécifiques à chaque type Nit permet de préserver la sûreté statique des langages étrangers. Pour incrémenter le compteur, la fonction porte

le nom plat du type statique suivi de `__incr_ref`, et pour décrémenter, le suffixe est `__decr_ref`.

Par exemple, les deux fonctions pour modifier le compteur de référence à un objet de type `Array[String]` depuis le code C ont les signatures suivantes :

```
1 void Array_of_String__incr_ref(Array_of_String);
2 void Array_of_String__decr_ref(Array_of_String);
```

Les arguments passés à une méthode externe sont préservés pour le temps de son exécution. Toutefois, lorsqu'un objet Nit est préservé à long terme, tel que dans une variable globale C, un attribut d'instance de Java ou une fermeture Objective-C, le compteur de référence doit être utilisé.

4.3.6.2 Services de finalisation

Pour personnaliser un comportement à la libération d'un objet Nit, le programmeur peut utiliser les services de finalisation. Sans être en lien direct avec la FFI de Nit, les services de finalisation servent souvent à libérer des ressources externes et ainsi ils sont utilisés dans un contexte de programmation polyglotte.

La bibliothèque standard offre une classe `Finalizable` qui est reconnue par le ramasse-miettes. Lors de la libération de l'une de ses instances, le ramasse-miettes invoque la méthode `finalize`. La méthode `finalize` doit être implémentée par les sous-classes et elle peut libérer des ressources limitées avant que l'objet Nit soit lui-même libéré. Par exemple, elle peut libérer un objet externe conservé en attribut.

La classe `FinalizableOnce` spécialise `Finalizable` et assure que sa méthode `finalize_once` n'est invoquée qu'une seule fois. Elle est généralement préférable à `Finalizable` où plusieurs invocations de `finalize` peuvent causer une double libération de ressources externes.

Les services de finalisation proviennent du ramasse-miettes de Nit et ainsi, ils sont soumis aux mêmes restrictions (Boehm, 2003). Nous dénotons trois limites auxquelles le programmeur polyglotte doit porter attention :

— L'appel automatique à `finalize` n'est pas fiable : à partir du moment où un objet n'est plus référencé, le ramasse-miettes peut invoquer `finalize` à tout moment, ou simplement jamais. La méthode est automatiquement invoquée lors du ramassage, mais celui-ci est imprévisible, il peut même n'y avoir aucun ramassage avant la fin d'exécution du programme. La méthode `finalize` peut donc servir à attraper les fuites, mais les clients devraient généralement invoquer `finalize` explicitement avant de perdre la dernière référence à l'objet.

— Les instances de `Finalizable` ne doivent pas avoir de références cycliques, ou alors elles ne seront jamais invoquées. Cette restriction provient de la visite dans un ordre topologique des objets à finaliser par le ramasse-miettes. Cet ordre assure la validité des attributs des objets à finaliser, mais il ne permet pas les cycles.

— La classe ordinaire `Finalizable` ne peut pas être spécialisée par une classe externe, donc les objets externes doivent être libérés depuis un objet Nit. Dans un tel cas, il est important que les objets étrangers soient bien encapsulés par l'objet Nit, et qu'ils ne soient pas référés ailleurs dans le programme. Nous présenterons le patron de conception *adaptateur polyglotte*, au chapitre 5, pour traiter ces cas.

4.3.7 Services de conversion de types

La correspondance de chaque type Nit à un type précis dans le code étranger apporte le besoin de manipuler les types Nit depuis le code étranger. Notamment lors de rappels à Nit, les arguments doivent être d'un type statique précis. Pour répondre à ce besoin, la FFI offre des services de vérification et de conversion de types.

D'une façon similaire aux rappels à Nit, le programmeur déclare les types manipulés par une méthode externe et la FFI génère sur mesure les services de types. La déclaration se fait avec les déclarations de rappel et prend une forme similaire à une conversion dans le code Nit pur, par exemple, `import Object.as(String)`.

La FFI génère deux fonctions pour le code étranger :

— La vérification de type, l'équivalent au `isa` de Nit, prend la forme d'une fonction `Object_isa_String(Object)`. Encore une fois, les noms plats sont utilisés pour composer le nom de la fonction. Cette fonction retourne vrai, ou l'équivalent du langage étranger, si l'objet passé en argument est du type Nit `String`.

— La conversion de type, l'équivalent au `as` de Nit, prend la forme d'une fonction `Object_as_String(Object): String`. Cette fonction change le type statique seulement, par contre, elle peut aussi convertir un type primitif en référence opaque, et vice versa. Si le type dynamique de l'argument ne correspond pas au type destination, cette fonction cause un *abort*.

En pratique, ces services sont peu utilisés, et pour une bonne raison : Nit est meilleur à manipuler les types Nit que les autres langages. Suivant notre principe *en programmation polyglotte, utiliser chaque langage selon ses spécialités*, il est généralement plus simple de manipuler les types Nit en Nit. Au chapitre suivant, nous présenterons le patron de conception *façade polyglotte* qui limite l'utilisation des services de conversion et des autres rappels en organisant ce travail avant et après l'appel au code étranger.

4.3.8 Bibliothèques de support

Alors que la FFI est intégrée à la grammaire de Nit et aux moteurs d'exécution, des services de support sont offerts par des bibliothèques Nit. Les bibliothèques sont dédiées à un langage étranger précis, elles définissent, entre autres, les classes externes fondamentales et des services pour convertir les chaînes Nit en chaînes étrangères. Nous les présenterons dans les sections dédiées à chaque langage.

4.4 Architecture modulaire de la FFI dans les moteurs d'exécution

Avant de poursuivre avec les détails du support de chaque langage par la FFI, nous survolons leur organisation dans les moteurs d'exécution de Nit. Deux principales contraintes ont guidé la conception architecturale de la FFI de Nit :

Le support d'Objective-C et Java ouvre l'accès aux API natives à Android et à iOS. Il s'agit d'un besoin fondamental pour ce projet de recherche.

L'indépendance envers le moteur d'exécution assure la portabilité des modules utilisant la FFI avec le compilateur, l'interpréteur et tout autre moteur futur. Cette contrainte n'est pas prioritaire dans le cadre de ce projet, nous utilisons uniquement le compilateur pour générer les applications mobiles.

Pour respecter les deux contraintes, la FFI de Nit a été divisée en plusieurs interfaces frontales, chacune dédiée au support d'un langage étranger. Autant le compilateur que l'interpréteur peuvent utiliser les mêmes interfaces frontales, mais tous deux gèrent différemment l'interaction entre Nit et le code étranger.

Le compilateur Nit intègre la FFI au processus de compilation du code Nit en code natif. Alors qu'il recompile normalement le code Nit en code C, il fait appel aux interfaces frontales pour faire le pont entre le code C généré et le code écrit par le programmeur, qu'il soit en C, Objective-C ou Java. Tout le code généré est compilé par les outils appropriés pour obtenir un binaire natif ou du code octet Java.

Pour déterminer quelle est la plateforme cible, le compilateur recherche l'annotation `platform` parmi les modules importés. Cette annotation accepte comme seul argument le nom de la plateforme cible. Il adapte alors la chaîne de compilation selon la plateforme déclarée par l'annotation, générant un projet Android, iOS ou, par défaut, un projet pour le système local. Chaque chaîne de compilation est réalisée par un module distinct dans le compilateur.

L'interpréteur de Nit ne supporte que la FFI avec le langage C. N'ayant normalement pas de phase de compilation, nous avons modifié l'interpréteur pour qu'il compile le code étranger à sa première utilisation. À l'exécution, lorsqu'une méthode étrangère est invoquée, l'interpréteur compile tout le code C présent dans le module en une bibliothèque native qu'il charge dynamiquement. Par la suite, il relaie les appels aux méthodes externes aux fonctions correspondantes de la bibliothèque native.

L'implémentation de la FFI dans l'interpréteur n'est qu'une preuve de concept. Elle pourrait être étendue avec toutes les fonctionnalités de la FFI de Nit par un effort de développement futur.

Le reste de ce chapitre présente les interfaces frontales : les FFI avec C, avec Objective-C et avec Java.

4.5 La FFI avec C

L'interface frontale avec C, surnommée simplement la FFI avec C, permet au programmeur d'implémenter des méthodes Nit en C et de rappeler des services Nit depuis le code C. Elle reprend toutes les fonctionnalités de la précédente interface native de Nit, avec la nouvelle forme imbriquée comme seul changement.

La FFI avec C est compatible avec Android, iOS, GNU/Linux, macOS et Windows. Elle donne accès à des API C telles que POSIX, OpenGL ES et GTK+.[3] De plus, une utilisation commune de la FFI avec C est l'optimisation de méthodes critiques avec des algorithmes de bas niveau et un contrôle précis de la mémoire. La FFI avec C nous a également servi à accéder à l'API C du NDK d'Android qui sert de base au fonctionnement de Nit sur Android, ainsi qu'à implémenter la FFI avec Java.

Cette section présente les particularités de la FFI avec C. Nous commençons par présenter le processus de compilation d'un module imbriquant du code C. Ensuite, nous revisitons les fonctionnalités communes et leur application en C : les méthodes externes, la correspondance des types, les classes externes, les blocs d'entête et les rappels à Nit. Finalement, nous introduisons les bibliothèques de support à la FFI avec C ainsi que ses annotations.

4.5.1 Processus de compilation

La FFI avec C est intégrée aux moteurs d'exécution de Nit, elle agit à la compilation d'un module Nit qui imbrique du code C.

Comme pour tout module Nit, le compilateur analyse le code Nit et le recompile en code C. De plus, le compilateur extrait le code C de l'utilisateur des méthodes externes et des blocs d'entête pour le réorganiser en deux fichiers par modules, un fichier corps (`.c`) et une entête (`.h`). Finalement, il compile le code C généré et le code C utilisateur pour les joindre en un seul binaire.

Ce processus est illustré par la figure 4.5.

---

3. GTK+ permet de réaliser des applications graphiques portables entre GNU/Linux, macOS et Windows. Nous l'avons utilisé pour l'implémentation GNU/Linux de l'API UI de *app.nit*.

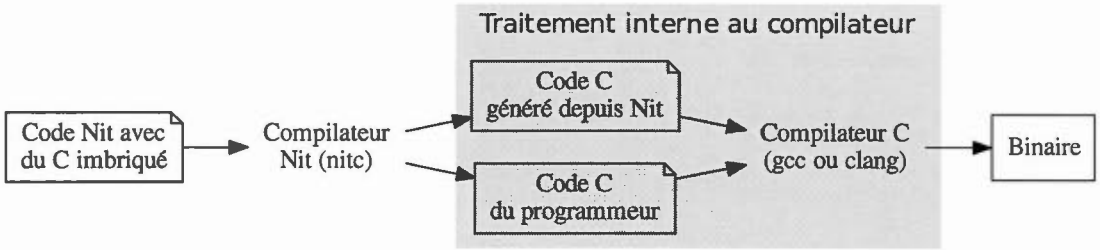


Figure 4.5: Processus de compilation d'un module utilisant la FFI avec C.

4.5.2 Méthodes externes

Le corps des méthodes externes implémentées en C prend la forme de fonctions C. Par exemple, la méthode suivante est implémentée en C dans le but d'utiliser l'API C de GTK+, elle affiche un message dans une fenêtre GTK+.

```
1 fun show_gtk_window(message: CString) `{
2
3     // Construit la fenêtre
4     gtk_init(0, NULL);
5     GtkWindow *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
6     g_signal_connect(window, "destroy", gtk_main_quit, NULL);
7
8     // Prépare le message
9     GtkWidget *label = gtk_label_new(message);
10    gtk_container_add((GtkContainer*)window, label);
11
12    // Affiche la fenêtre
13    gtk_widget_show_all((GtkWidget*)window);
14    gtk_main();
15 `}
16
17 show_gtk_window("Message à afficher".to_cstring)
```

4.5.3 Correspondance des types

La correspondance des types entre Nit et C est guidée par les trois règles générales communes au support de tous les langages, présentées à la section 4.3.2. Tel qu'attendu, les classes externes C portent le type associé en C et les classes externes Objective-C portent un type opaque. Par contre, les classes externes Java portent le type `jobject`

dans le code C. Cette particularité provient de l'implémentation de la FFI avec Java et sera discutée à la section 4.7.9.

4.5.4 Classes externes

Les classes externes C sont associées à des types pointeurs du langage C. L'exemple suivant définit `Pointer`, la classe racine à toutes les classes externes, et `CString` qui est associée à une chaîne de caractères C.

```
1 extern class Pointer `{ void* `}
2 end
3
4 extern class CString `{ char* `}
5 end
```

Certaines bibliothèques Nit contournent les restrictions sur les pointeurs et associent des classes externes à des entiers ou des énumérations. En pratique, ces bibliothèques conservent des entiers dans la valeur même du pointeur. Cette approche n'est pas encouragée, mais elle est tolérée tant qu'elle respecte les restrictions du langage C. Il s'agit d'une solution temporaire à une limitation du langage Nit qui n'offre pas d'énumérations au moment de l'écriture.

4.5.4.1 Spécialisation

Les classes externes C peuvent spécialiser des interfaces, ou d'autres classes externes C, mais seulement lorsque les types associés sont compatibles. Notamment, `CString`, comme toutes les classes externes, est implicitement sous-classe de `Pointer` et comme de fait, il y a une relation similaire entre les types C `char*` et `void*`.

De plus, dans certains cas, le code C peut simuler une hiérarchie de classes avec des structures. Nous recommandons alors de reproduire le même héritage sur les classes externes.

Par exemple, une alternative à la méthode externe `show_gtk_window`, présentée par la figure 4.6, consiste à représenter l'héritage de classe simulé par la bibliothèque GTK+ 3.0 par des classes externes. La hiérarchie peut être reproduite fidèlement en Nit pour factoriser le code accédant à leurs services.

4.5.4.2 Méthodes et constructeurs

Les classes externes C peuvent déclarer des méthodes, mais pas d'attributs. Elles peuvent aussi déclarer des constructeurs externes C pour faire appel à un `malloc` ou à un service d'une bibliothèque C pour allouer l'objet étranger.

4.5.5 Blocs d'entête

Les blocs d'entête C permettent d'insérer du code à deux endroits.

- Les blocs `in "C"` sont destinés au début du corps C (le fichier `.c`). On y déclare les importations privées ainsi que des services C locaux au module.
- Les blocs `in "C Header"` sont destinés à l'entête (le fichier `.h`) de l'unité de compilation. On y déclare les importations et les types qui sont associés aux classes externes publiques du module. Ces fragments de code sont automatiquement importés par les modules clients qui utilisent aussi la FFI avec C. De cette façon, la correspondance de type des classes externes peut être appliquée dans les autres modules clients. L'exemple de la figure 4.6 utilise un tel bloc pour importer l'entête GTK+.

4.5.6 Rappels à Nit

Les rappels à Nit depuis C sont générés sous la forme de fonctions statiques. Les règles de correspondance des types sont appliquées pour déterminer le type des paramètres et du retour de la méthode.

Les rappels ne sont disponibles que pour les méthodes externes, ils ne sont pas directement disponibles dans les blocs d'entête. Pour les invoquer depuis un bloc `in "C"`, leur prototype doit être déclaré explicitement dans le bloc.

```
1 module gtk3 is pkgconfig("gtk+-3.0")
2
3 in "C Header" `{
4     #include <gtk/gtk.h>
5 `}
6
7 # Contrôle racine
8 extern class GtkWidget `{ GtkWidget* `}
9     # Affiche ce contrôle et tous ses enfants
10    fun show_all `{ gtk_widget_show_all(self); `}
11 end
12
13 # Étiquette
14 extern class GtkLabel `{ GtkLabel* `}
15    super GtkWidget
16
17    # Contruit une étiquette
18    new (text: CString) `{ return gtk_label_new(text); `}
19 end
20
21 # Fenêtre
22 extern class GtkWindow `{ GtkWindow* `}
23    super GtkWidget
24
25    # Initialise GTK+ et construit la fenêtre
26    new `{
27        gtk_init(0, NULL);
28        GtkWindow *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
29        g_signal_connect(window, "destroy", gtk_main_quit, NULL);
30        return window;
31    `}
32
33    # Ajoute `widget` à la fenêtre
34    fun add(widget: GtkWidget) `{
35        gtk_container_add((GtkContainer*)self, widget);
36    `}
37
38    # Affiche la fenêtre
39    fun main `{ gtk_main(); `}
40 end
41
42 # Code client
43 var win = new GtkWindow
44 var lbl = new GtkLabel("Message à afficher".to_cstring)
45 win.add lbl
46 win.show_all
47 win.main
```

Figure 4.6: Module utilisant les services de l'API C de GTK+ 3.0.

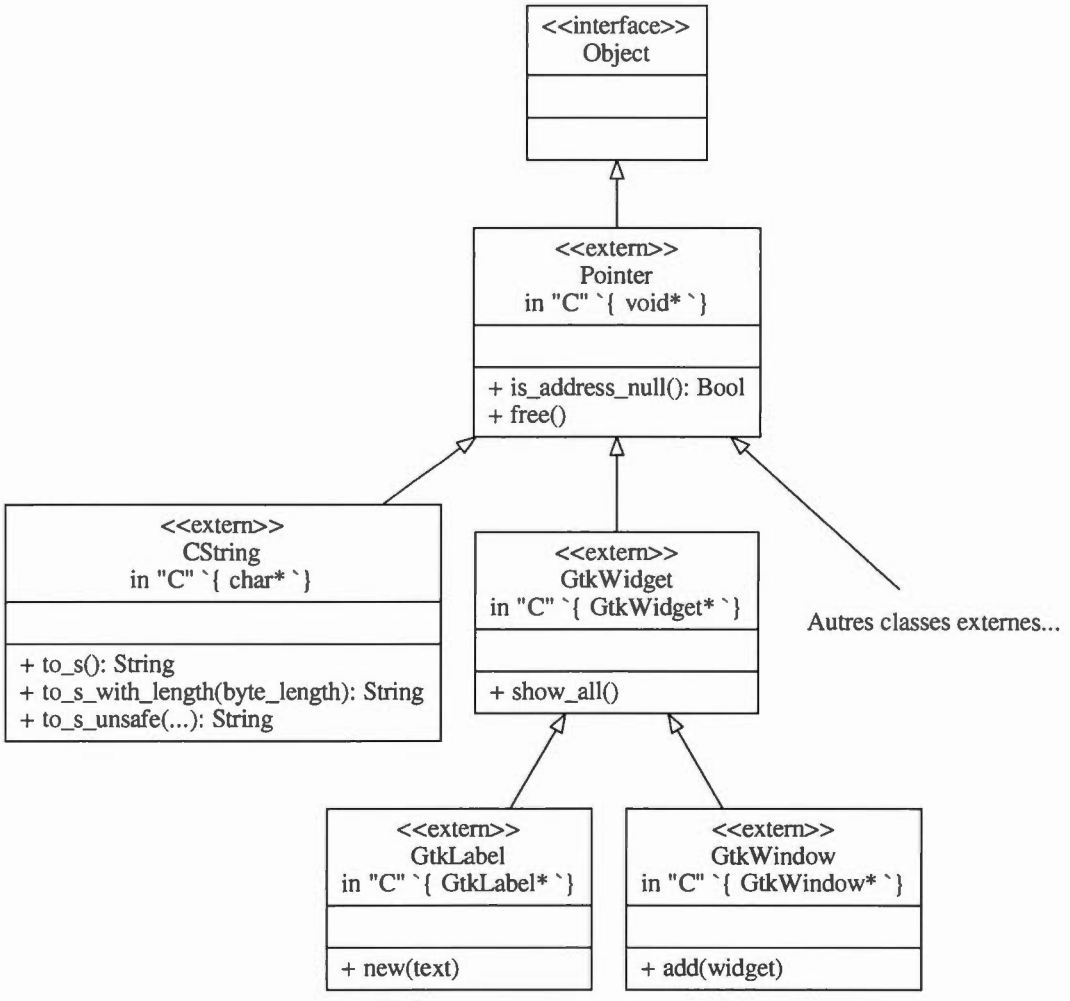


Figure 4.7: La classe `Pointer` et d'autres classes externes C en relation d'héritage.

4.5.7 Bibliothèques de support

La bibliothèque standard de Nit offre des services en support à la FFI avec C, ils sont centrés autour des classes `Pointer` et `CString`. La figure 4.7 en présente les deux classes en relation d'héritage, leurs principaux services et certaines sous-classes représentatives des classes externes C.

La classe externe `Pointer` associée au type C `void*` définit deux méthodes principales :

— La méthode `is_address_null` détecte si l'adresse du pointeur est nulle. Elle peut

être invoquée sur toute instance d'une classe externe.

— La méthode `free` fait appel à la fonction C du même nom pour libérer l'espace mémoire référencé.

La classe externe `CString` définit plusieurs méthodes qui convertissent des chaînes C en chaînes Nit.

— Les méthodes `CString::to_s` et `to_s_with_length` convertissent une chaîne C en chaîne Nit. Alors que `to_s` recherche le premier caractère nul pour déterminer la taille de la chaîne, la taille (en octets) doit être spécifiée explicitement lors de l'appel à `to_s_with_length`. Ces services recopient la chaîne dans un espace sous le contrôle du ramasse-miettes de Nit et élaguent les caractères invalides en UTF-8.

— La méthode `CString::to_s_unsafe` convertit, elle aussi, une chaîne C en chaîne Nit. Toutefois, elle offre plus d'options qui sont potentiellement dangereuses. Elle permet de spécifier si la chaîne doit être copiée sous le contrôle du ramasse-miettes et si elle doit être élaguée des caractères invalides en UTF-8. Ces deux options peuvent causer une corruption de la mémoire si la chaîne d'origine est libérée par un autre service ou si elle n'était pas dans un format UTF-8 valide.

À l'inverse, la méthode `String::to_cstring` produit une chaîne C depuis une chaîne Nit. Elle peut retourner une référence aux données originales, ou allouer un nouvel espace mémoire et y recopier son contenu. La chaîne C ne doit pas être modifiée, au risque de corrompre la mémoire de l'application Nit.

4.5.8 Gestion de la mémoire

Lorsque des objets Nit sont conservés depuis le code C ou que des données C sont utilisées depuis Nit, une attention particulière doit être portée à la gestion de la mémoire. Dans cette section, nous présentons les solutions pratiques aux problèmes de gestion de mémoire avec la FFI de C en utilisant les services généraux de la FFI présentés à la section 4.3.6.1.

Les références opaques aux objets Nit passés au code C sont valides jusqu'à la fin de l'exécution de la méthode externe. Pour conserver des références à long terme, tel que dans une variable globale C, le programmeur doit utiliser le compteur de référence de la FFI. Incrémenter le compteur d'une référence à un objet assure que la référence et que l'objet restent valides jusqu'à ce que le compteur soit décrémenté.

Les instances des classes externes C pointent généralement vers un espace mémoire en dehors de la responsabilité du ramasse-miettes de Nit. L'espace mémoire est alors alloué par `malloc` ou un équivalent. Lorsque l'objet n'est plus utilisé, pour éviter les fuites mémorielles, il doit généralement être libéré par un appel à la méthode `free`, présenté à la section précédente. Une alternative est d'automatiser la libération en utilisant les services de finalisation. Dans tous les cas, le programmeur doit s'assurer de respecter les spécificités des services C utilisés, notamment ne pas libérer d'espace mémoire qui pourraient être en utilisation ailleurs dans le logiciel, en Nit ou en C.

4.5.9 Options du compilateur et de l'éditeur de lien

Trois annotations permettent de passer des options à la chaîne d'outils C : `cflags`, `ldflags` et `pkgconfig`. Elles sont déclarées avec le nom du mobile.

L'annotation `cflags` passe des options au compilateur C. Elle permet, entre autres, de préciser des dossiers où sont situés les entêtes importés par le code C :

```
module jvm is cflags "-I/usr/lib/jvm/default-java/include/"
```

L'annotation `ldflags` passe des options à l'éditeur de liens (*linker*) qui forme le binaire final. Elle permet de lier des fichiers supplémentaires, dont les bibliothèques à chargement dynamique :

```
module egl is ldflags "-lEGL"
```

Autant `cflags` que `ldflags` acceptent un argument spécial, l'expression `exec` qui exécute une commande dans un *shell*. La sortie standard de la commande est utilisée comme option pour la compilation ou l'édition de lien. L'expression `exec` permet de faire appel à des programmes qui retournent les options pour le compilateur et l'édition de liens :

```
1 module sdl is
2     cflags exec("sdl-config", "--cflags")
3     ldflags(exec("sdl-config", "--ldflags"), "-lSDL_image")
4 end
```

De plus, les deux annotations peuvent elles-mêmes être annotées par `@android` si elles sont spécifiques à la plateforme Android. Les options sont alors ignorées par les autres plateformes. Par exemple, l'annotation permet de lier le code C à une bibliothèque dynamique sous Android seulement :

```
module glesv2 is ldflags("-lGLESv2")@android
```

Finalement, l'annotation `pkgconfig` fait appel à *pkg-config*, un programme qui fournit des informations sur les paquets systèmes installés sur la machine. Cette annotation récupère automatiquement les options pour le compilateur et l'édition de liens. Elle accepte au plus un seul argument, le nom du paquet qui peut contenir, au besoin, un numéro de version. Par défaut, le nom du module est utilisé comme nom du paquet. L'annotation n'est utilisée que pour créer des binaires pour GNU/Linux et macOS. Par exemple, elle était utilisée dans l'exemple de la figure 4.6 pour configurer le module utilisant GTK+ 3.0 avec la ligne suivante :

```
module gtk3 is pkgconfig "gtk+-3.0"
```

Cette annotation détecte d'abord si le paquet est connu par *pkg-config*, sinon, un message invite l'utilisateur à installer le paquet système correspondant (avec *aptitude*, *brew*, ou autres). Ensuite, un appel à *pkg-config* obtient automatiquement les options à passer au compilateur et à l'édition de liens.

4.6 La FFI avec Objective-C

La FFI avec Objective-C traite les méthodes externes, les classes externes et les blocs d'entête identifiés par l'expression clé `in "ObjC"`. Elle est compatible avec iOS, macOS et GNU/Linux. De plus, le code généré est standard et compatible avec au moins deux compilateurs Objective-C : clang et gcc.

Cette section présente les particularités de l'utilisation de la FFI avec Objective-C. Nous commençons par présenter les détails techniques du processus de compilation qui aident à comprendre le fonctionnement de la FFI. Ensuite, nous revisitons les fonctionnalités communes et leur application en Objective-C : les méthodes externes, la correspondance des types, les classes externes, les blocs d'entête et les rappels à Nit. Nous introduisons les services propres à la FFI avec Objective-C : les bibliothèques de support et la gestion de la mémoire.

4.6.1 Processus de compilation

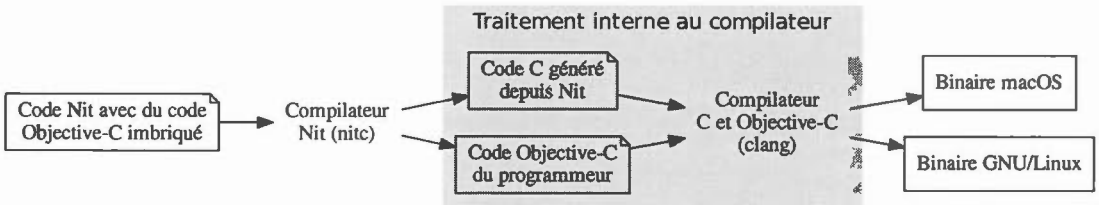
Le processus de compilation de la FFI avec Objective-C est relativement simple, car il profite de la compatibilité syntaxique et binaire d'Objective-C avec C. Le processus est illustré par la figure 4.8.

Le compilateur Nit extrait le code Objective-C du programmeur et le réorganise dans une unité de compilation associée au module, un fichier corps (`.m`) et un fichier d'entête (`.h`). Il y insère d'autres services, dont les fonctions de rappel. Finalement, le code Objective-C est compilé parallèlement au code C et est lié avec le code C pour former un exécutable pour macOS ou GNU/Linux.

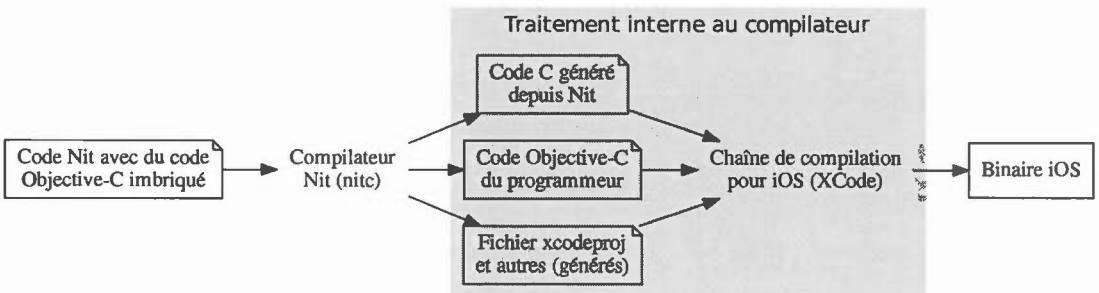
Alternativement, pour créer une application iOS, le compilateur Nit délègue une partie importante du travail à XCode, la suite d'outils officielle d'Apple pour le développement iOS. Le compilateur Nit génère une structure de projet XCode, il y organise le code C et Objective-C, puis il délègue la compilation de l'application iOS à XCode qui utilise le compilateur clang en arrière-plan. Le résultat est un dossier avec l'extension `.app` qui contient l'exécutable, l'icône et les autres ressources de l'application.

4.6.2 Méthodes externes

Le corps des méthodes externes implémentées en Objective-C prend la forme d'un corps de fonction Objective-C. La méthode `show_alert_box` ouvre une fenêtre de dialogue sous macOS et y affiche le message passé en argument :



(a) Processus de compilation sous macOS et GNU/Linux



(b) Processus de compilation pour une application iOS

Figure 4.8: Processus de compilation d'un module utilisant la FFI avec Objective-C.

```
1 fun show_alert_box(message: NSString) in "ObjC" `{
2   NSAlert *alert = [[[NSAlert alloc] init] autorelease];
3   [alert setMessageText: message];
4   [alert runModal];
5 `}
```

4.6.3 Correspondance des types

La correspondance des types entre Nit et Objective-C est guidée par les trois règles générales. Il y a toutefois deux particularités : d'abord, les classes externes C portent leur type associé en Objective-C, un avantage de la compatibilité entre les deux langages ; de plus, comme pour la FFI avec C, les classes externes Java portent le type `jobject` en Objective-C.

4.6.4 Classes externes

Les classes externes Objective-C doivent être associées à un pointeur (de classes Objective-C, de structures ou autres). L'exemple suivant définit la classe externe

`NSObject` associée à la classe racine de la hiérarchie de classes Objective-C et `NSString` associée à une chaîne de caractères Objective-C.

```
1 extern class NSObject in "ObjC" `{ NSObject* `}
2 end
3
4 extern class NSString in "ObjC" `{ NSString* `}
5   super NSObject
6 end
```

Nous recommandons de nommer les classes externes selon la classe Objective-C associée, en ignorant le pointeur. Le style des noms de classes Objective-C est compatible avec celui de Nit et le fait de préserver des noms similaires favorise le transfert de connaissance d'Objective-C à Nit.

4.6.4.1 Spécialisation

Les classes externes qui pointent sur une classe Objective-C devraient généralement spécialiser les classes externes Nit qui correspondent aux super-classes Objective-C. Cette approche permet de reproduire le graphe d'héritage d'Objective-C en Nit, et ainsi préserver l'accès aux services hérités sans duplication de code. Si le graphe n'est pas fidèlement reproduit, les méthodes externes pour accéder à un même service peuvent être dupliquées.

La figure 4.9 présente `NSObject`, sa super-classe `Pointer`, et certaines classes externes Objective-C représentatives.

4.6.4.2 Méthodes et constructeurs

Les classes externes Objective-C peuvent déclarer des méthodes (externes ou non) ainsi que des constructeurs externes, mais pas d'attributs. L'exemple suivant offre une alternative pour créer et manipuler un `NSAlert` depuis Nit. Une classe externe correspond à la classe Objective-C `NSAlert`, elle définit un constructeur et deux méthodes externes donnant accès aux services minimaux.

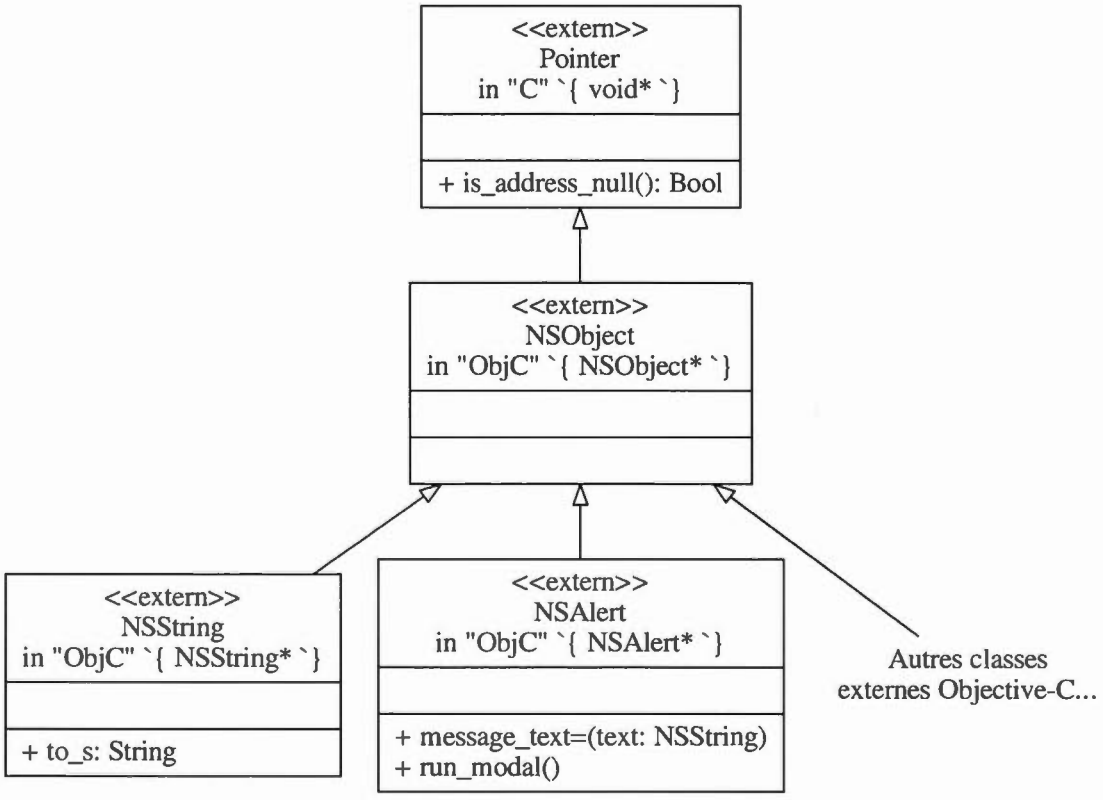


Figure 4.9: Relation d'héritage entre la classe `NSObject` et autres.

```
1 # Fenêtre de dialogue sous macOS
2 extern class NSAlert in "ObjC" `{ NSAlert * `}
3   super NSObject
4
5   # Alouer l'espace et initialiser une nouvelle fenêtre
6   new in "ObjC" `{ return [[NSAlert alloc] init]; `}
7
8   # Assigner le text à afficher
9   fun message_text=(text: NSString) in "ObjC" `{
10    [self setMessageText: text];
11  `}
12
13  # Afficher la fenêtre
14  fun run_modal in "ObjC" `{ [self runModal]; `}
15 end
```

Cette classe peut être instanciée et manipulée naturellement depuis le code Nit.

```
1 var alert = new NSAlert
2 alert.message_text = "Bonjour!".to_nsstring
3 alert.run_modal
```

4.6.5 Blocs d'entête

Les blocs d'entête Objective-C sont similaires à ceux de C, ils permettent d'insérer du code à deux endroits dans l'unité de compilation générée par le compilateur pour le module :

- Les blocs `in "ObjC"` sont destinés au début du corps Objective-C (le fichier `.m`). On y déclare les importations privées ainsi que des services Objective-C locaux au module.
- Les blocs `in "ObjC Header"` sont destinés à l'entête (le fichier `.h`) de l'unité de compilation. On y déclare les importations et les types qui sont associés aux classes externes publiques du module. Ces fragments de code sont automatiquement importés par les modules clients qui utilisent aussi la FFI avec Objective-C.

L'exemple suivant importe l'entête de la bibliothèque AppKit qui déclare la classe `NSAlert` utilisée plus tôt :

```
1 in "ObjC Header" `{
2   #import <AppKit/AppKit.h>
3 `}
```

```
# Exécute `task` depuis la thread UI
fun run_on_ui_thread(task: Task) import Task.main in "ObjC" `{

  // Incrémente le compteur de référence
  Task_incr_ref(task);

  // Place la fermeture en queue pour exécution sur thread UI
  dispatch_async(dispatch_get_main_queue(), ^{

    // Rappel à `Task::main`
    Task_main(task);

    // Décrémente le compteur de référence
    Task_decr_ref(task);
  });
`}
```

Figure 4.10: Rappel à Nit dans l'implémentation iOS de l'API UI, avec de légères modifications pour la lisibilité. Cette méthode permet de passer un objet Nit via le système d'iOS pour exécuter du code sur la *thread UI*.

4.6.6 Rappels à Nit

Les rappels à Nit depuis le code Objective-C prennent la forme d'appels de fonctions de style C. Nous avons préféré ce format à celui des appels de méthodes d'Objective-C pour conserver le style partagé par tous les langages étrangers et éviter l'incompatibilité avec le format des appels de méthodes où apparaît le nom des paramètres.

Les figures 4.10 et 4.11 présentent des exemples réels de méthodes externes Objective-C avec des rappels à Nit.

4.6.7 Bibliothèques de support

Il n'y a pas de bibliothèque standard officielle au langage Objective-C, toutefois les API natives à iOS et à macOS dépendent de bibliothèques généralistes, dont Core Foundation, Core Data, AppKit et UIKit.

La bibliothèque Core Foundation définit les classes et services de base, dont la racine de la hiérarchie de classe `NSObject` et la chaîne de caractère `NSString`. Pour en assurer

```
redef class NSString
  # Lance une requête HTTP à l'URL représentée par `self`
  #
  # Retourne `null` lors d'erreur et stocke le message d'erreur
  # dans `error_ref`.
  fun http_get(timeout: Float, error_ref: Ref[NSString]): NSData
  import Ref[NSString].item= in "ObjC" `{

    // Prépare les arguments
    NSURL *url = [NSURL URLWithString:self];
    NSURLRequest *request = [NSURLRequest requestWithURL:url
      cachePolicy:NSURLRequestUseProtocolCachePolicy
      timeoutInterval:timeout];

    NSURLResponse *response = nil;
    NSError *error = nil;

    // Exécute la requête
    NSData *data = [NSURLConnection sendSynchronousRequest:request
      returningResponse:&response
      error:&error];

    // Traite les cas d'erreur
    if (data == nil) {
      NSString *message = [error localizedDescription];

      // Rappel à Nit pour assigner l'erreur à `error_ref`
      Ref_of_NSString_item__assign(error_ref, message);
      return nil;
    }

    return data;
  `}
end
```

Figure 4.11: Rappel à Nit dans l'implémentation iOS de l'API HTTP, avec de légères modifications pour la lisibilité. Cette méthode lance une requête HTTP avec les services natifs à iOS et retourne les messages d'erreur via un rappel en stockant le message dans un des arguments. Il s'agit d'une façon de réaliser des retours multiples.

la compatibilité avec Nit, le module `cocoa::foundation` de la bibliothèque de Nit offre des classes externes, dont `NSObject` et `NSString`, et les services de conversion de chaînes de caractères :

— La méthode `String::to_nsstring` convertit une chaîne Nit en chaîne Objective-C. La nouvelle chaîne copie les données de façon à ce qu'elles passent sous la responsabilité du compteur de référence d'Objective-C.
— La méthode `NSString::to_s` fait l'opération inverse, elle convertit une chaîne Objective-C en chaîne Nit. Elle copie les données sous la responsabilité du ramasse-miettes de Nit et élague les caractères invalides en UTF-8.

La bibliothèque Core Data est portable entre iOS et macOS. Elle offre notamment `NSUserDefaults` que nous avons utilisé pour implémenter l'API persistance des données de *app.nit*.

Les bibliothèques d'interface utilisateur sont spécialisées selon la plateforme. AppKit offre les contrôles graphiques pour macOS alors que UIKit offre ceux de iOS. Nous avons donc utilisé les services de UIKit pour implémenter l'API UI *app.nit*.

4.6.8 Gestion de la mémoire

Pour le programmeur, une des principales difficultés à faire cohabiter Nit et Objective-C se trouve au niveau de la gestion de la mémoire. Les objets Nit sont alloués et libérés par un ramasse-miettes, alors que les objets Objective-C sont gérés par un compteur de référence.

Un objet Objective-C qui est conservé à long terme dans l'environnement Nit doit être protégé pour qu'il ne soit pas libéré par le compteur de référence automatique (ARC). Pour ce faire, le programmeur doit utiliser les services `CFBridgingRetain` pour extraire l'objet de ARC et en obtenir une référence. L'objet passe sous la responsabilité du programmeur qui doit le libérer manuellement avec `CFRelease`. Ces deux services sont offerts par la bibliothèque Core Foundation avec différentes alternatives qui donnent plus de contrôle sur le compteur de référence.

À l'inverse, pour préserver une référence à un objet Nit en Objective-C, le programmeur doit utiliser les services de la FFI. Les services d'incrémentation et de décrémentation du compteur de référence, présentés à la section 4.3.6.1, permettent de préserver une référence externe vers un objet Nit pour éviter qu'elle soit libérée par le ramasse-miettes.

4.7 La FFI avec Java

La FFI avec Java traite les méthodes externes, les classes externes et les blocs d'entête identifiés par l'expression clé `in "Java"`. Nous l'avons utilisé principalement pour accéder à l'API native d'Android, mais elle est aussi fonctionnelle sous GNU/Linux. La FFI avec Java partage la structure syntaxique des autres FFI, mais son implémentation dans le compilateur est plus complexe.

Dans cette section, nous présentons la FFI avec Java, ses particularités et certains détails d'implémentation. L'interface visible au programmeur est similaire aux FFI avec C et avec Objective-C, et ce, par choix. Cette similarité aide au transfert de connaissances lors de l'utilisation de plusieurs langages étrangers avec Nit. Pour simplifier la lecture de cette section, le tableau 4.4 regroupe les différences entre les FFI avec C, avec Objective-C et avec Java.

L'importance de ce projet de recherche est tangible lorsqu'on compare la FFI avec Java aux équivalents offerts par les autres solutions en développement d'applications portables. Plusieurs alternatives populaires n'utilisent pas de FFI moderne et relèguent encore au programmeur la tâche d'écrire le code complexe d'utilisation de la JNI. Cette section sert donc d'argument en faveur d'une FFI expressive qui offre une interface simple et statique au programmeur, et qui génère le code répétitif à la compilation.

Cette section présente les particularités à l'utilisation de la FFI avec Java. Nous commençons par présenter l'implémentation sur la JNI et les détails techniques du processus de compilation. Ensuite, nous revisitons les fonctionnalités communes pour voir leur application en Java : les méthodes externes, la correspondance des types, les classes externes (ainsi que le rôle de `JavaObject` avec C), les blocs d'entête et les rappels à Nit. Nous introduisons les services propres à la FFI avec Java : les bibliothèques de support, la gestion de la mémoire, les exceptions, les *threads*, l'annotation `extra_java_file` et l'intégration d'outils d'analyse de code.

Tableau 4.4: Comparaison des FFI avec C, avec Objective-C et avec Java.

Caractéristique	C	Objective-C	Java
Type associé aux classes externes	Pointeurs	Pointeurs à objets et autres	Références à objets et tableaux
Racine des classes externes	`Pointer` implicitement	`NSObject` suggéré	`JavaObject` obligatoirement
Sortes de blocs d'entête	`C` et `C Header`	`ObjC` et `ObjC Header`	`Java` et `Java Inner`
Forme des rappels à Nit	Fonctions ordinaires	Fonctions de style C	Fonctions natives
Forme des références opaques	Types générés	Types générés	Entiers (`int`)
Gestion de la mémoire	`malloc` et `free`	Compteur de référence ARC	Cadres et références globales
Bibliothèques de support	`core` (la bibliothèque standard de Nit)	`cocoa`	`java`, `jvm` et `android::dalvik`
Plateformes mobiles	iOS et Android	iOS	Android
Autres plateformes	GNU/Linux, macOS et Windows	GNU/Linux et macOS	GNU/Linux
Outils utilisés par nitc	gcc ou clang	clang et XCode	javac, ant et ndk-build
Passage entre les langages	Compatiblité avec le code C généré par nitc	Compatiblité binaire	JNI

4.7.1 Implémentation sur la JNI

La FFI avec Java repose sur la JNI pour exécuter le code Java écrit par le programmeur et réaliser les rappels à Nit. La JNI est une interface standard, indépendante de la structure interne des machines virtuelles Java. Son utilisation assure la compatibilité de la FFI de Nit avec les machines virtuelles Java populaires, dont HotSpot et celles d'Android : Dalvik et ART.

La JNI prend deux formes principales : les fonctions natives et son API C.

— La JNI permet la déclaration de fonctions natives dans le code Java, c'est-à-dire, des méthodes Java implémentées par du code natif. Les fonctions natives sont l'équivalent Java aux méthodes externes de Nit, mais sans la forme imbriquée.
L'implémentation native d'une méthode y est associée par une des deux façons suivantes. Elle est recherchée automatiquement parmi les fonctions d'une bibliothèque native chargée dynamiquement par un appel à `System.loadLibrary` depuis le code Java. Ou alors, la fonction native est associée explicitement à un pointeur de fonction depuis le code natif par un appel à `RegisterNatives`.
La FFI de Nit utilise `RegisterNatives` pour lier des fonctions natives Java à des services du binaire Nit pour réaliser les rappels à Nit. Cette technique s'adapte bien aux applications Nit dont le code natif peut agir comme exécutable indépendant sous GNU/Linux, ou être chargé en tant que bibliothèque native sous Android.

— L'API C de la JNI offre des services pour manipuler l'environnement Java et la JVM depuis le code natif. Elle permet au programmeur de créer des objets Java, d'invoquer des méthodes et de traiter les exceptions.
La FFI de Nit utilise l'API C pour exécuter le code Java des méthodes externes. L'utilisation de l'API C requiert beaucoup de *code colle* répétitif, même pour de simples rappels au code Java, mais ce code est généré par la FFI de Nit et la complexité d'utilisation de la JNI est cachée au programmeur.

Il est important de noter que, par rapport à la JNI, Nit agit comme langage natif. Tel qu'illustré par la figure 4.12, une méthode externe Nit implémentée en Java est réalisée par un rappel à Java via l'API C de la JNI. Et un rappel à Nit depuis Java est une fonction native Java dont l'implémentation en C redirige chaque appel au système Nit.

Figure 4.12: Utilisation de la JNI par la FFI avec Java.

4.7.2 Processus de compilation et d'exécution

Le processus de compilation d'un module utilisant la FFI avec Java, illustré par la figure 4.13, est plus complexe que l'équivalent avec Objective-C. Cette section décrit le processus étape par étape pour mettre de l'avant les comportements alternatifs lorsque l'application cible Android ou GNU/Linux.

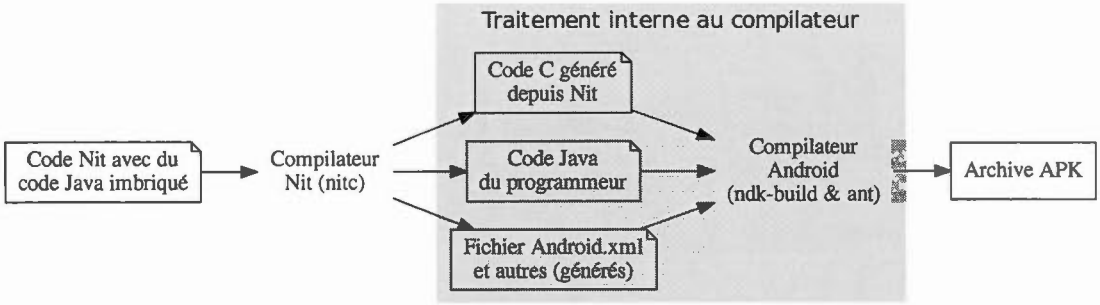
1. Le compilateur génère le code C qui manipule la JNI. Il génère aussi une classe Java par module Nit et y insère le code Java imbriqué parmi le code Nit.

   Pour Android seulement, le compilateur génère aussi la structure standard d'un projet Android, dont le fichier *Android.xml* qui définit les métadonnées de l'application. Les fichiers C et Java sont générés dans le projet, dans les dossiers `jni/` et `src/` respectivement.

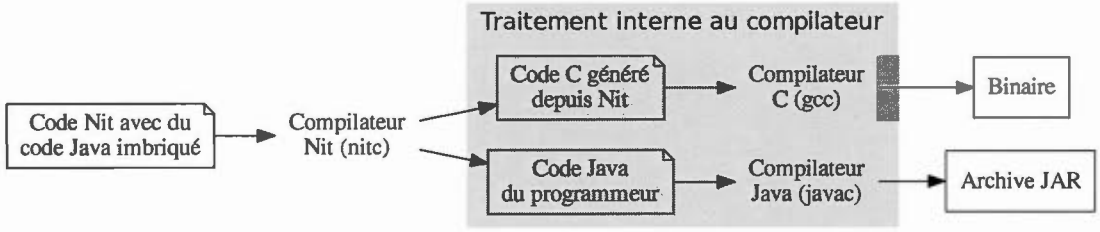
2. Le compilateur Nit invoque les outils de la plateforme visée pour compiler le code C et Java.

   a. Pour Android, le code C est compilé via l'outil `ndk-build`, un outil du NDK, et le code Java est compilé via un appel à `ant`. Le résultat est l'application Android, une archive APK qui contient le binaire natif et le code octet Java.

   b. Pour GNU/Linux, le code C est compilé par `gcc` ou `clang` pour produire le binaire natif. Le code Java est compilé avec *javac* puis organisé dans une archive JAR. L'archive est générée dans le même dossier que l'exécutable et elle porte le même nom auquel on ajoute l'extension `.jar`.



(a) Processus de compilation pour Android



(b) Processus de compilation pour GNU/Linux

Figure 4.13: Processus de compilation d'un module utilisant la FFI avec Java.

3. À l'exécution, le binaire natif manipule une instance de la JVM et charge dynamiquement les classes Java via la JNI. Les détails de la relation avec la JVM varient selon la plateforme.

   a. En Android, les applications sont lancées par une JVM associée au processus par le système d'exploitation. Le point d'entrée de l'application est un morceau de code Java de la bibliothèque `app.nit` qui délègue le contrôle au système Nit et au code natif. La FFI avec Java réutilise la JVM existante qui est exposée au code C par les services du NDK.

   b. En GNU/Linux, le point d'entrée est le binaire natif. Au premier appel à une méthode externe Java, une JVM est instanciée automatiquement via les services de la JNI. La JVM est configurée de façon à chercher les classes Java dans l'archive JAR qui doit être dans le même dossier ou identifiée par la variable d'environnement `CLASS_PATH`.

4.7.3 Méthodes externes

Le corps des méthodes externes Java prend la forme d'une fonction statique Java. De cette façon, le code Java n'a pas le receveur implicite `this` habituel à une méthode d'instance Java. Par contre, la FFI de Nit insère le receveur Nit parmi les paramètres, sous le nom `self`.

Par exemple, la méthode externe `show_toast` affiche un petit message au bas de l'écran sous Android.

```
1 fun show_toast(context: NativeContext, message: JavaString)
2 in "Java" `{
3     Toast toast = new Toast(context);
4     toast.setText(message);
5     toast.show();
6 `}
```

La complexité réelle du passage entre Nit et Java est cachée au programmeur. La figure 4.14 présente le code C généré pour manipuler la JNI et invoquer l'implémentation Java de la méthode `show_toast`.

4.7.4 Correspondance des types

La conversion des types entre Nit et Java est guidée par les trois règles générales. La seule particularité est qu'il n'y a pas de types générés pour les références opaques à des objets Nit qui sont simplement typées par des entiers (`int`) en Java. Chacun de ces entiers agit comme une référence opaque, ou un identifiant, à un objet Nit. Ces entiers ne doivent pas être modifiés, toute modification pouvant entraîner une corruption de la mémoire.

L'utilisation d'entiers est une limitation de la FFI avec Java, elle n'étend pas la sûreté statique des types Nit au code Java comme les types statiques générés. Tout de même, la FFI avec Java bénéficie du typage statique sur les types primitifs et les classes externes Java. Il y a des avenues intéressantes à explorer pour représenter les types Nit en Java, mais nous les laissons à un effort de recherche futur.

```
1 // Fonction C généré pour exécuter le code Java de `show_toast`
2 void toast___Sys_show_toast___impl(Sys self, jobject context, jobject message)
3 {
4     // Récupère une référence à la JVM via les services
5     // du module `jvm` et la FFI avec C.
6     Sys sys = Pointer_sys(NULL);
7     JNIEnv *jni_env = Sys_jni_env(sys);
8
9     // Recherche de la classe générée pour le module `toast`
10    // via les services du module `jvm` et la FFI avec C.
11    jclass java_class = Sys_load_jclass(sys, "Nit_toast");
12
13    if (java_class == NULL) {
14        (*jni_env)->ExceptionDescribe(jni_env);
15        exit(1);
16    }
17
18    // Recherche de la fonction statique qui est l'implémentation de
19    // la méthode `show_toast` depuis son nom et sa signature.
20    jmethodID = (*jni_env)->GetStaticMethodID(
21        jni_env, java_class,
22        "toast___Sys_show_toast___java_impl",
23        "(ILandroid/content/Context;Ljava/lang/String;)V");
24
25    if (java_meth_id == NULL) {
26        (*jni_env)->ExceptionDescribe(jni_env);
27        exit(1);
28    }
29
30    // Invocation du code du programmeur.
31    (*jni_env)->CallStaticVoidMethod(
32        jni_env, java_class,
33        java_meth_id, self, context, message);
34
35    // Nettoyage
36    (*jni_env)->DeleteLocalRef(jni_env, java_class);
37 }
```

Figure 4.14: Code C généré pour exécuter la méthode externe `show_toast`, annoté et légèrement modifié pour la lisibilité. Cette fonction est invoquée par le système Nit et déclenche l'exécution du code Java qui implémente la méthode externe `show_toast`.

4.7.5 Classes externes

Les classes externes Java sont associées à une interface Java, une classe Java (paramétrée ou non) ou à un tableau Java. Le type externe doit être déclaré avec son nom qualifié complet, lequel est utilisé pour le passage des instances de la classe via la JNI.

Voici des exemples représentatifs de classes externes associées à différents types Java :

```
1 # Chaîne de caractères Java
2 extern class JavaString in "Java" `{ java.lang.String `}
3   super JavaObject
4 end
5
6 # L'interface `Comparable`
7 extern class JavaComparable in "Java" `{ java.lang.Comparable `}
8   super JavaObject
9 end
10
11 # Une liste de chaînes de caractères
12 extern class StringList in "Java" `{ java.util.List<java.lang.String> `}
13   super JavaObject
14 end
15
16 # Un tableau d'objets
17 extern class JavaArray in "Java" `{ java.lang.Object[] `}
18   super JavaObject
19 end
```

4.7.5.1 Spécialisation

Les classes externes Java peuvent spécialiser des interfaces Nit et d'autres classes externes Java. De plus, elles doivent obligatoirement spécialiser la classe externe `JavaObject` pour être correctement manipulées par la FFI de Nit.

Par exemple, les trois classes externes suivantes sont associées à des contrôles de l'interface utilisateur d'Android et en reproduisent la spécialisation :

```
1 # Racine de la hiérarchie des contrôles sous Android
2 extern class AndroidView in "Java" `{ android.view.View `}
3   super JavaObject
4 end
5
6 # Contrôle affichant du texte
```

```
7 extern class AndroidTextView in "Java" `{ android.view.TextView `}
8   super AndroidView
9 end
10
11 # Bouton interactif
12 extern class AndroidButton in "Java" `{ android.widget.Button `}
13   super AndroidTextView
14 end
```

Tout comme en Objective-C, les classes externes Java devraient reproduire autant que possible l'héritage des classes Java correspondantes pour que les services soient hérités correctement. Le système d'héritage de Java peut être représenté entièrement en Nit, même l'héritage multiple des interfaces Java peut être représenté par un héritage multiple entre les classes externes de Nit.

4.7.5.2 Méthodes et constructeurs

Les classes externes Java peuvent déclarer des méthodes (externes ou non) ainsi que des constructeurs externes, mais pas d'attributs. L'exemple suivant offre une alternative a `show_toast` pour obtenir le même résultat, une classe externe associée à la classe Java `android.widget.Toast` en reproduit les services par un constructeur et des méthodes externes :

```
1 # Court message pour l'utilisateur
2 extern class AndroidToast in "Java" `{ android.widget.Toast `}
3     super AndroidView
4
5     new (context: NativeContext) in "Java" `{
6         return new Toast(context);
7     `}
8
9     # Assigne le message à afficher
10    fun text=(message: JavaString) in "Java" `{
11        self.setText(message);
12    `}
13
14    # Affiche le message
15    fun show in "Java" `{ self.show(); `}
16 end
```

Cette classe peut être instanciée et manipulée naturellement depuis le code Nit :

```
17 var toast = new AndroidToast(native_activity)
18 toast.text = "message".to_java_string
19 toast.show
```

4.7.6 Blocs d'entête

Les blocs d'entête Java permettent d'insérer du code à deux endroits dans la classe Java générée par le compilateur. La figure 4.15 présente un module Nit avec des deux blocs et la classe Java générée par le compilateur regroupant le contenu des blocs.

- Les blocs `in "Java"` sont insérés en haut du fichier de classe, avant la déclaration de la classe. Les importations de classes Java sont déclarées dans ce bloc.
- Les blocs `in "Java Inner"` sont insérés à l'intérieur de la classe Java générée. À cet endroit, le programmeur peut déclarer des fonctions et attributs de classe Java. La classe générée n'est normalement pas instanciée, donc tous ses services doivent être statiques, c'est-à-dire associés à la classe et non pas à ses instances. Les fonctions statiques peuvent servir à factoriser du code Java utilisé à répétition dans un même module Nit. De plus, ce bloc peut déclarer des classes imbriquées, lesquelles peuvent être instanciées au besoin. Le principal cas d'utilisation de classes imbriquées est pour implémenter des interfaces.

L'exemple suivant importe les classes nécessaires pour réaliser la classe externe `AndroidToast` présentée plus haut :

```
1 in "Java" `{
2     import android.view.View;
3     import android.widget.Toast;
4 `}
```

4.7.7 Rappels à Nit

Les rappels à Nit depuis Java prennent la forme d'appels de fonctions Java. Tel que discuté précédemment à la section 4.7.1, pour chaque rappel, le compilateur génère une fonction native Java dont l'implémentation redirige les appels au système Nit.

La figure 4.16 présente un exemple réel de méthode externe Java avec un rappel à Nit.

```
module mon_module

in "Java" `{
// Bloc "Java"
import android.widget.Toast;
`}

in "Java Inner" `{
  // Bloc "Java Inner"
  private void show_toast(message: String) {
    Toast toast = new Toast(context);
    toast.setText(message);
    toast.show();
  }
`}

class MaClasse
  fun foo(message: JavaString) in "Java" `{
    // Méthode externe
    show_toast(message);
  `}
end
```

(a) Module Nit utilisant la FFI avec Java.

```
// Bloc "Java"
import android.widget.Toast;

public class Nit_mon_module {
  // Bloc "Java Inner"
  private void show_toast(message: String) {
    Toast toast = new Toast(context);
    toast.setText(message);
    toast.show();
  }

  public static void mon_module___foo_impl(int self, String message) {
    // Méthode externe
    show_toast(message);
  }
}
```

(b) Classe Java générée par le compilateur Nit.

Figure 4.15: Module Nit utilisant la FFI avec Java et la classe Java générée.

```
1 extern class NativeActivity in "Java" `{ android.app.Activity `}
2   super NativeContextWrapper
3
4   # Exécute `task` depuis la thread UI
5   fun run_on_ui_thread(task: Task) import Task.main in "Java" `{
6
7     // Incrémente le compteur de référence
8     Task_incr_ref(task);
9
10    final int final_task = task;
11    self.runOnUiThread(new Runnable() {
12      @Override
13      public void run() {
14
15        // Rappel à `Task::main`
16        Task_main(final_task);
17
18        // décrémente le compteur de référence
19        Task_decr_ref(final_task);
20      }
21    });
22  `}
23 end
```

Figure 4.16: Rappel à Nit dans l'implémentation Android de l'API UI, avec de légères modifications pour la lisibilité. Cette méthode est équivalente à celle du même nom pour iOS présentée dans la figure 4.10, page 116.

4.7.8 Bibliothèques de support

La bibliothèque de Nit offre trois paquets en support à la FFI avec Java : `java`, `jvm` et `android::dalvik`.

Le paquet `java` offre des services de compatibilité entre Nit et Java. Ces services desservent autant l'implémentation de la FFI que le code client. Ce paquet doit être importé par tout module utilisant la FFI avec Java, car il définit la classe `JavaObject` qui est obligatoirement spécialisée par toutes les classes externes Java. La classe `JavaObject` offre des services pertinents à toutes classes externes Java, dont `is_java_null` qui retourne vrai si la valeur Java est nulle. La figure 4.17 présente `JavaObject` et certaines classes externes Java représentatives.

Le paquet `java` définit aussi les services de conversion de chaînes de caractères :

- La méthode `String::to_java_string` convertit les chaînes Nit en chaînes Java UTF-16 et en copie les données sous la responsabilité du ramasse-miettes Java.
- La méthode `JavaString::to_s` convertit les chaînes Java en chaînes Nit UTF-8 et copie les données sous la responsabilité du ramasse-miettes de Nit.

Le paquet `jvm` reproduit en Nit un sous-ensemble des services de l'API C de la JNI. Il permet de détecter la machine virtuelle Java, de la manipuler et de la démarrer au besoin. Ce paquet utilise la FFI avec C seulement et il agit dans l'implémentation de la FFI avec Java. Le paquet `jvm` est rarement utilisé par le programmeur moyen qui peut se contenter du paquet `java` et de l'abstraction offerte par la FFI.

À la compilation pour GNU/Linux, le paquet `jvm`, et donc la FFI avec Java, utilise deux variables d'environnement pour lier le binaire avec une JVM.

- `JAVA_HOME` identifie le dossier d'installation de la JVM.
- `JNI_LIB_PATH` identifie le dossier contenant la bibliothèque `libjvm.so`.

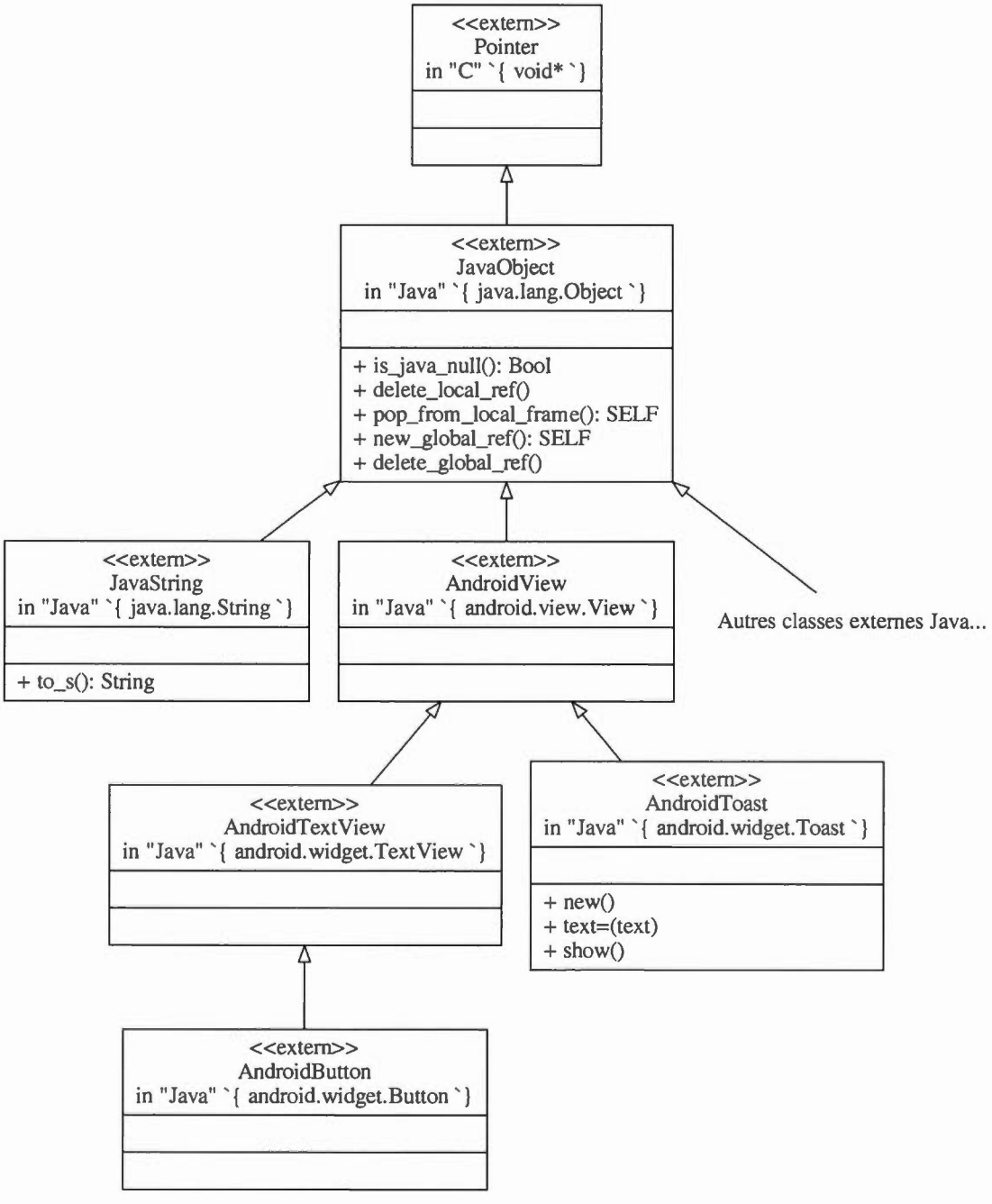


Figure 4.17: Relation d'héritage entre la classe `JavaObject` et autres.

De plus, à l'exécution, la variable `LD_LIBRARY_PATH` doit contenir le dossier avec `libjvm.so` pour que la bibliothèque puisse être chargée dynamiquement.

Finalement, le paquet `android::dalvik` adapte les services du paquet `jvm` pour les machines virtuelles d'Android. Sous Android, la JVM est lancée par le système d'exploitation et il est impossible de démarrer une autre JVM. Ce paquet raffine aussi le service de chargement d'une classe Java pour rechercher parmi les classes de l'APK.

4.7.9 `JavaObject` et la FFI avec C

La classe `JavaObject` est associée à la classe Java `java.lang.Object`. Tout comme son équivalent Java, `JavaObject` est la racine de la hiérarchie des classes externes Java. De façon spéciale, la FFI associe `JavaObject` au type C `jobject`, un type défini par l'API C de la JNI. Le type `jobject` représente les références à des objets Java depuis le code C.

Le résultat est que `JavaObject` est associé à deux types étrangers, `jobject` en C et Objective-C, et `java.lang.Object` en Java. Il en est de même pour toutes les autres classes externes Java, elles héritent de la correspondance automatique avec `jobject` en C et elles ont une correspondance précisée par le programmeur à un type Java.

Cette double correspondance permet de passer des instances de classes externes Java autant à des méthodes externes Java, C que Objective-C. Alors que les méthodes externes Java peuvent manipuler directement les objets Java passés en arguments, les méthodes C et Objective-C peuvent les manipuler via les services de la JNI. Cette particularité est notamment utilisée dans la bibliothèque pour transformer les chaînes Java en chaînes Nit.

Dans l'exemple de la figure 4.18, la classe externe `JavaNioBuffer` est associée à la classe Java `java.nio.Buffer` qui est un service de la JNI pour partager des données efficacement entre le code Java et le code natif. La classe externe déclare des méthodes externes en Java et en C. Elle utilise la FFI avec Java pour accéder à `isReadOnly` et la FFI avec C pour accéder aux données via `GetDirectBufferAddress`.

```
1 # Tampon accessible efficacement depuis Nit, C et Java
2 extern class JavaNioBuffer in "Java" `{ java.nio.Buffer `}
3   super JavaObject
4
5   # Est-ce que le tampon est en lecture seule?
6   fun is_read_only: Bool in "Java" `{
7     return self.isReadOnly();
8   `}
9
10  # Adresse du tampon
11  fun direct_buffer_address(jni_env: JniEnv): Pointer `{
12    return (*jni_env)->GetDirectBufferAddress(jni_env, self);
13  `}
14
15  # Capacité du tampon
16  fun direct_buffer_capacity(jni_env: JniEnv): Int `{
17    return (*jni_env)->GetDirectBufferCapacity(jni_env, self);
18  `}
19 end
```

Figure 4.18: Classe externe `JavaNioBuffer` qui utilise les FFI avec Java et C.

4.7.10 Gestion de la mémoire

Le langage Java utilise un ramasse-miettes pour libérer les objets Java qui ne sont plus référencés depuis le code Java. Une différence notable avec le compteur de référence automatique d'Objective-C est que le ramasse-miettes Java peut déplacer les objets en mémoire à tout moment, et ainsi invalider des références depuis un langage étranger (dont Nit).

Une attention particulière doit être portée aux objets Java conservés depuis le code Nit. Le programmeur peut utiliser deux mécanismes de la JNI pour assurer la viabilité des références aux objets Java : les cadres et les références.

4.7.10.1 Pile d'exécution

Le ramasse-miettes de Java préserve, entre autres, les objets Java référencés par les variables locales aux méthodes en cours d'exécution. En fait, pour être plus précis, le

ramasse-miettes préserve les objets Java référencés par un cadre (*frame*) sur la pile d'exécution Java.

Cette protection s'étend aussi aux objets qui quittent l'environnement Java. Elle assure ainsi une certaine protection aux objets Java passés au code Nit comme arguments d'un rappel à Nit ou par un retour d'une méthode externe. Cette politique est documentée par la spécification de la JNI, toutefois, son application du côté Nit se fait d'un point de vue différent et mérite une attention particulière.

— Un objet Java passé comme argument d'un rappel à une méthode Nit est valide jusqu'à ce que la méthode Nit se termine. Par après, il peut être libéré à tout moment. Pour préserver l'objet plus longtemps en Nit, il faut utiliser les services de références globales.
— De plus, un objet Java retourné par une méthode externe Java est associé au cadre précédent, qui est le dernier rappel à Nit depuis Java. Donc, un objet récupéré par un appel à une méthode externe Java est valide dans la méthode du site d'appel. Par contre, dans certains cas, un grand nombre d'objets Java peuvent être associés à un même cadre. Cette situation peut mener à un débordement du cadre qui cause le lancement d'une exception par la JNI.

La JNI offre plusieurs services pour manipuler les cadres et éviter les débordements. Nous les présentons ici tels qu'exposés en Nit par les modules `jvm` et `java` :

— La méthode `JavaObject::delete_local_ref` retire une référence à un objet Java de son cadre. Elle peut être utilisée par le programmeur pour éviter les débordements lorsqu'une référence à un objet Java n'est plus utilisée.
— La méthode `JniEnv::push_local_frame` alloue un nouveau cadre d'exécution Java sur le dessus de la pile d'exécution. Ce cadre accueille les objets retournés par les appels à des méthodes externes. La méthode accepte un seul argument, le nombre maximum de références que conservera le cadre.

— Les méthodes `JniEnv::pop_local_frame` et `JavaObject::pop_from_local_frame` libèrent le cadre sur le dessus de la pile d'exécution Java. Les références locales au cadre sont alors invalidées, à l'exception du receveur de `pop_from_local_frame` qui est traité comme le retour d'une méthode et dont la référence est déplacée au cadre précédent.

Les services `push_local_frame` et `pop_local_frame` permettent d'allouer manuellement des cadres Java depuis le code Nit. Une bonne pratique est d'associer un cadre Java aux méthodes Nit qui font plusieurs appels à des méthodes externes Java. Allouer un cadre avec `push_local_frame` en début de la méthode Nit et le libérer avec `pop_local_frame` avant la fin de la méthode assure qu'il n'y aura pas de fuite de références. La figure 4.19 présente une telle utilisation d'un cadre dans l'implémentation de l'API HTTP sous Android. Alternativement, la valeur de retour d'une méthode peut être préservée avec `pop_from_local_frame`, ou encore, toute valeur peut être conservée en référence globale.

4.7.10.2 Références globales

La JNI offre également des services pour maintenir des références globales à un objet Java. Ces références ne sont pas associées à un cadre et elles peuvent être conservées à long terme dans l'environnement Nit. La méthode Nit `JavaObject::new_global_ref` retourne une nouvelle référence globale depuis une référence locale à un cadre, et `JavaObject::delete_global_ref` libère une référence globale (son receveur). La responsabilité de libérer la référence globale revient au programmeur pour éviter les fuites mémorielles.

La figure 4.20 présente un exemple d'un objet Java préservé dans un attribut comme référence globale et libéré avec l'objet l'encapsulant.

```
redef class Text

    # Lance une requête HTTP GET synchrone à l'URL `self`
    #
    # ~~~
    # var response = "http://example.org/".http_get
    # assert not response.is_error
    # print "HTTP status code: {response.code}"
    # print response.value
    # ~~~
    redef fun http_get
    do
        # Réserve un cadre pour 4 références
        jni_env.push_local_frame 4

        var juri = self.to_java_string # -> JavaString
        var jrep = java_http_get(juri) # -> JavaObject

        var res
        if jrep.is_exception then
            jrep = jrep.as_exception # -> JavaException
            var jmsg = jrep.message # -> JavaString
            var error = new IOError(jmsg.to_s)
            res = new HttpRequestResult(null, error)
        else if jrep.is_http_response then
            jrep = jrep.as_http_response # -> JavaHttpResponse
            var content = jrep.content # -> JavaString
            var status = jrep.status
            res = new HttpRequestResult(content.to_s, null, status)
        else abort

        # Libère le cadre
        jni_env.pop_local_frame
        return res
    end
end
```

Figure 4.19: Utilisation d'un cadre Java depuis Nit dans l'API HTTP pour Android. Un cadre Java est créé au début et libéré à la fin de la méthode `http_get`. Les références à des objets Java retournés par les appels à des méthodes externes (identifiées par leur type en commentaire) sont associées au cadre et libérées avec lui. Les appels à des méthodes Java qui ne retournent que des types primitifs, tel que `is_exception`, n'ajoutent pas d'objets au cadre.

```
1 class AndroidButton
2     super Finalizable
3
4     # Référence globale conservée à long terme
5     private var native: NativeAndroidButton
6
7     init do
8         # Création de l'objet Java `android.widget.Button`
9         var local = new NativeAndroidButton(app.native_activity)
10
11        # Convertit la référence locale en référence globale
12        self.native = local.new_global_ref
13    end
14
15    # Libère la référence globale avec l'objet Nit
16    redef fun finalize do native.delete_global_ref
17 end
```

Figure 4.20: Référence globale à un objet Java depuis Nit dans l'API UI sous Android. La classe Nit `Button` est réalisée par un bouton natif à Android. Une référence globale au bouton Android est conservée par l'attribut `native`, et elle est libérée avec l'objet Nit lors de l'appel à `finalize`.

4.7.11 Exceptions

Le langage Java utilise des exceptions pour rapporter et propager les erreurs. Cette fonctionnalité est incompatible avec Nit qui n'offre pas d'équivalent direct. Pour remédier à cette différence, nous appliquons notre principe, *en programmation polyglotte, utiliser chaque langage selon ses spécialités* et laissons au code Java le traitement des exceptions.

La FFI avec Java impose une restriction aux méthodes externes, elles doivent capturer toutes les exceptions et ne laisser passer que les cas dont il est impossible de récupérer. Cette restriction est appliquée statiquement et dynamiquement :

— Le compilateur Nit réorganise l'implémentation en Java des méthodes externes dans des fonctions Java qui ne déclarent pas d'exceptions (donc pas de `throws`). Ceci délègue la responsabilité au compilateur Java de détecter statiquement les exceptions qui ne sont pas capturées par les méthodes externes.
Toutefois, ceci ne concerne pas les exceptions sérieuses (`Error`) ni les exceptions normales (`RuntimeException`) de la machine virtuelle qui n'ont pas à être déclarées statiquement. De plus, il y a des cas documentés où des exceptions peuvent être lancées même si elles ne sont pas déclarées statiquement, notamment en abusant de la JNI (Li et Tan, 2014).

— Pour détecter les autres cas, à l'exécution, après chaque appel à une méthode externe Java, le système Nit vérifie qu'il n'y a pas d'exceptions levées via l'API C de la JNI. Donc, si une exception n'est pas capturée par une méthode externe, elle sera détectée à ce moment. Dans un tel cas, le programme quitte en affichant, à la console, la trace d'appel de la JVM.

Il en revient à la responsabilité du programmeur de capturer et de traiter les exceptions à l'intérieur des méthodes externes. Lorsqu'une exception est levée, la solution est souvent de retourner l'exception elle-même ou de l'assigner à un attribut. La figure 4.21 présente comment les exceptions sont traitées dans l'implémentation Android de l'API requêtes HTTP.

```
1 import java
2
3 in "Java" `{
4     import org.apache.http.client.methods.HttpGet;
5     import org.apache.http.impl.client.DefaultHttpClient;
6 `}
7
8 # Lance une requête HTTP à l'URL représentée par `self`
9 #
10 # Implémentée avec les services Apache HTTP de l'API Java sous Android.
11 #
12 # Retourne soit une instance de `JavaException` ou de `JavaHttpResponse`.
13 private fun java_http_get(url: JavaString): JavaObject
14 in "Java" `{
15     try {
16         // Construit la requête
17         DefaultHttpClient client = new DefaultHttpClient();
18         HttpGet get = new HttpGet(url);
19
20         // Retourne la réponse
21         return client.execute(get);
22     }
23     catch (Exception ex) {
24         // Retourne l'exception elle-même
25         return ex;
26     }
27 `}
```

Figure 4.21: Gestion des exceptions dans l'API requêtes HTTP sous Android.

Cette méthode est invoquée par le code présenté à la figure 4.19, page 137.

4.7.12 *Threads*

L'API native à Android restreint les requêtes HTTP à certains *threads* pour ne pas bloquer l'interface utilisateur. Tel que discuté au chapitre 2, cette restriction est respectée par les API UI et requêtes HTTP de *app.nit* dont l'implémentation dépend fortement du support des *threads*.

Nous avons implémenté le support pour les *threads* POSIX en Nit de façon à en assurer la compatibilité avec les *threads* Java au travers de la FFI. Pour ce faire, la FFI fait appel aux services de la JNI pour attacher et détacher des *threads* POSIX à la JVM.

Le programmeur est responsable de détacher les *threads* de la JVM en fin d'exécution avant de les joindre par un appel à la méthode `JavaVM::detach_current_thread`. En pratique, cette responsabilité est souvent prise en charge par un service de haut niveau. Notamment, les *threads* de l'API requêtes HTTP de *app.nit* sont détachés automatiquement.

4.7.13 Annotation `extra_java_file`

L'annotation `extra_java_file` associe un fichier source Java externe à un module Nit, permettant d'implémenter une classe Java dans un fichier dédié. Le fichier source Java sera ainsi compilé avec l'application et le fichier de classe résultant sera joint à l'archive JAR ou à l'APK.

Cette annotation permet de contourner une limitation de la FFI avec Java qui associe chaque module Nit à une classe Java générée. Alors que le programmeur peut y ajouter des classes imbriquées par un bloc d'entête, il ne peut pas ajouter des classes Java non imbriquées.

Nous utilisons notamment cette annotation pour déclarer le point d'entrée des applications *app.nit* sous Android. Le module Nit `nit_activity` est associé à la classe Java `NitActivity` qui relaie tous les messages du système d'exploitation au code Nit et aux services de *app.nit*.

```
module nit_activity is extra_java_files "NitActivity.java"
```

4.7.14 Intégration d'outils d'analyse de code

Plusieurs auteurs ont étudié l'analyse et la vérification du code utilisant la JNI :

- SafeJNI (Furr et Foster, 2006) et le *framework* ILEA (Tan et Morrisett, 2007) analysent statiquement le code utilisant la JNI pour y détecter les erreurs de programmation.
- L'outil Jinn (Lee et al., 2010) analyse les programmes à l'exécution avec une machine à état pour détecter les erreurs de manipulation de la JNI.
- Le *framework* JET (Li et Tan, 2011) analyse statiquement les exceptions lancées depuis le code C afin d'assurer qu'elles sont correctement déclarées en Java.
- Deux projets visent à assurer la sûreté de la mémoire, l'un utilise un modèle formel (Tan, 2010) alors que l'autre, SafeJNI (Tan et al., 2006), vérifie les pointeurs avec CCured (Necula, McPeak et Weimer, 2002).
- L'outil CheckJNI (CheckJNI, 2016) détecte des erreurs de manipulation de la JNI à l'exécution, dont l'utilisation de références invalides à des objets Java et les rappels à Java alors qu'une exception est levée. Cet outil est inclus directement dans les JVM HotSpot, Dalvik et ART.

Ces travaux pourraient être appliqués avec la FFI de Nit car le code généré utilise la JNI de façon standard. Pour l'instant, nous avons seulement utilisé CheckJNI qui nous a assisté lors du débogage de l'implémentation de la FFI avec Java pour Android. Nous laissons à des travaux futurs d'intégrer davantage d'analyses.

4.8 Conclusion

La programmation polyglotte favorise l'utilisation de plusieurs langages selon leurs spécialités. Pour l'intégrer à notre solution de développement d'applications portables et en faciliter l'utilisation, nous avons conçu et implémenté la FFI de Nit.

La FFI de Nit innove en combinant plusieurs points : (i) Elle s'intègre au paradigme de programmation orientée objet et en étend les concepts principaux avec les méthodes et

classes externes, ainsi que les rappels à Nit. (ii) Elle imbrique le code étranger parmi le code Nit. (iii) Elle accepte plusieurs langages étrangers, chacun ayant des spécialités, mais elle profite aussi de leurs similarités. (iv) Elle génère sur mesure les services pour le code étranger de façon à préserver leur sûreté statique.

Les trois interfaces frontales partagent la même structure syntaxique générale tout en donnant chacune accès à un langage étranger différent :

— La FFI avec C permet d'invoquer du code C depuis Nit et d'accéder à des bibliothèques natives. Elle agit dans l'implémentation de la FFI avec Java et le support de Nit sous Android via le NDK.
— La FFI avec Objective-C ouvre l'accès aux API natives à iOS et permet l'adaptation des applications *app.nit* pour la plateforme. Elle est compatible avec iOS, macOS et GNU/Linux, ainsi qu'avec les compilateurs Objective-C clang et gcc. Son implémentation profite de la compatibilité d'Objective-C avec le langage C.
— La FFI avec Java ouvre l'accès aux API natives à Android. Basée sur la JNI, la FFI avec Java est compatible avec les JVM populaires. Son implémentation est complexe, mais le code répétitif est généré par le compilateur pour en simplifier l'utilisation par le programmeur client.

La programmation polyglotte est le dernier composant de notre solution *app.nit*, mais elle y prend un rôle central.

D'abord, la programmation polyglotte agit dans l'implémentation des API abstraites *app.nit* qui permettent de réaliser un prototype entièrement portable à Android et à iOS. Nous, en tant que développeurs de la bibliothèque, avons utilisé la FFI de Nit pour implémenter chaque API sous chaque plateforme. L'expressivité de la FFI nous a permis de réaliser nos intentions directement, sa souplesse a aidé à l'évolution des API, et le rapport des erreurs à la compilation a non seulement simplifié le développement, mais assure aussi une certaine sûreté à son utilisation.

Pour les programmeurs utilisant *app.nit*, la programmation polyglotte donne davantage de pouvoir à l'organisation en ligne de produits. Alors que des variations permettent

d'adapter l'application à chaque plateforme, la programmation polyglotte ouvre l'accès aux API natives, dans leur entièreté et leur langage natif, pour améliorer les adaptations. L'accès entier permet l'utilisation de toutes les fonctionnalités propres à chaque plateforme et celles offertes par des bibliothèques tierces parties. L'utilisation du langage natif favorise le transfert de l'expertise préalable avec chaque plateforme et, de plus, elle permet au programmeur de suivre la documentation officielle d'Android et d'iOS. La forme imbriquée de la FFI facilite son apprentissage et l'agrément à son utilisation. Enfin, combinant tous ces avantages, la FFI de Nit favorise l'adaptation à la plateforme de tout projet Nit, de la petite application *app.nit* à des bibliothèques entières.

CHAPITRE V

PATRONS DE CONCEPTION POLYGLOTTE

Au cours de ce projet de recherche, les membres de notre groupe de recherche ont écrit une quantité notable de code polyglotte.[1] Nous avons été confrontés à des problèmes récurrents d'architecture logicielle. Alors que la FFI de Nit facilite la programmation polyglotte, ce n'est qu'un mécanisme du langage, elle n'aide pas à la conception du logiciel. Il s'agit d'une situation où les patrons de conception sont bénéfiques, ils proposent une solution abstraite à des problèmes semblables (Gamma et al., 1994). Toutefois, aucun patron de la littérature scientifique n'offrait de solutions à nos problèmes fondamentalement polyglottes. Après avoir expérimenté avec différentes architectures logicielles, nous en avons extrait de nouveaux patrons de conception polyglotte.

Ce chapitre s'adresse aux programmeurs polyglottes qui, comme nous, font face à des dilemmes en matière d'architecture logicielle. Ce chapitre peut aussi être considéré comme un guide des bonnes pratiques pour l'utilisation de la FFI de Nit.

Dans ce chapitre, nous proposons trois nouveaux patrons de conception polyglotte qui visent des problèmes réels, que nous avons observés à répétition. Ces patrons nous ont servi pour implémenter les quatre API *app.nit* par du code polyglotte sous Android et

1. Nous estimons qu'environ 15 000 lignes de code polyglotte utilisant la FFI de Nit ont été contribuées au dépôt du projet Nit depuis le début de ce projet de recherche. Nous comptons les lignes avec du code, ignorant les commentaires, composant 115 modules qui sont fondamentalement polyglottes, dont les implémentations de *app.nit*, les API reproduites (telles que GTK+) et les autres services basés sur des méthodes externes.

iOS, ainsi que pour réaliser les trois applications de validation. Nous présentons aussi deux générateurs de code qui assistent le programmeur dans l'application des patrons.

Nous présentons les trois patrons en ordre de complexité selon les problèmes visés :

Façade polyglotte Échanger des données entre Nit et un langage étranger.
Adaptateur polyglotte Reproduire une API étrangère à objets en Nit.
Spécialisation polyglotte Spécialiser une classe Objective-C ou Java en Nit.

Nous décrivons chaque patron selon un format similaire à celui du livre *Design Patterns* (Gamma et al., 1994) :

Intention Description courte de la raison d'être du patron.
Motivation Exemple pratique du problème et de la solution proposée.
Applicabilité Indices guidant le moment où le patron doit être appliqué.
Structure Diagramme de séquence ou de classe qui représente la relation entre les participants du patron.
Participants Description des classes ou méthodes composant le patron, incluant leurs responsabilités.
Collaborations Les relations entre les participants du patron qui n'ont pas déjà été couvertes.
Conséquences Effets positifs et négatifs de l'application du patron.
Implémentation Détails de l'implémentation du patron, avec des pistes de solution pour des variations au patron.
Exemple de code Exemple pratique d'application du patron, découpant le code selon les aspects importants.
Utilisations connues Liste de projets où le patron est appliqué.
Patrons connexes Patrons similaires et ce qui les différencient.

5.1 Patron – Façade polyglotte

5.1.1 Intention

Échanger des données entre Nit et un langage étranger.

5.1.2 Motivation

Le besoin fondamental en programmation polyglotte est l'invocation de code étranger. Alors que la FFI de Nit répond à ce besoin, elle n'en établit pas les bonnes pratiques. Notamment, quand utiliser un langage ou un autre. Cette question prend de l'importance lorsque des données sont passées entre les langages.

Par exemple, pour changer le texte affiché et l'alignement d'un bouton sous Android, un programmeur doit faire appel à deux méthodes Java : `setText` et `setGravity`. Nous pouvons regrouper ces deux appels dans une seule méthode externe `set_style_native` à invoquer depuis Nit, tel qu'illustré à la figure 5.1.

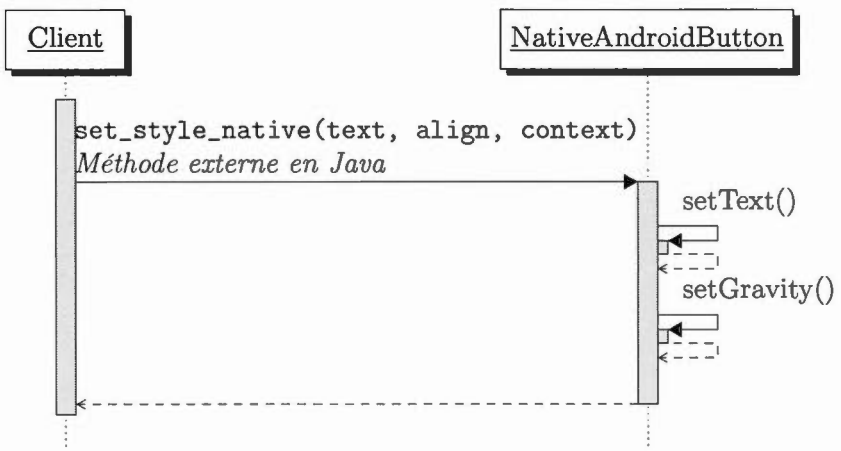


Figure 5.1: Appel à deux services natifs via une méthode externe.

Deux valeurs sont partagées entre Nit et Java dans un appel à `set_style_native` : le texte à afficher et l'alignement à utiliser. Il en résulte quelques questions : Quand convertir le texte d'une chaîne Nit à une chaîne Java, dans le code Nit ou dans le code Java ? Comment passer l'information d'alignement, par un type Nit ou une valeur de l'énumération Java ? Les deux langages peuvent répondre à ces besoins. De même qu'à la question, comment cacher la complexité de la FFI aux clients de `set_style_native` ?

La solution consiste toujours à appliquer notre principe fondamental : *en programmation polyglotte, utiliser chaque langage selon ses spécialités*, mais nous complétons maintenant

la formule en ajoutant : *et en cas de doute, préférer Nit comme langage généraliste.* Et donc séparer le traitement des données en deux méthodes, une en Nit et l'autre dans le langage étranger.

Dans notre exemple, nous avons déjà une méthode externe, `set_style_native`, qui peut traiter les données de la spécialité de Java et exécuter le travail désiré. Nous y ajoutons une méthode ordinaire, `set_style`, qui sert d'indirection pour traiter les données de la spécialité de Nit en Nit. Le résultat est une séparation du traitement en deux phases, une première partie en Nit et le reste dans le code Java.

Le langage Nit offrant des services pour la programmation polyglotte, dont la conversion des chaînes de caractères, l'argument `text` peut être converti d'une chaîne Nit à une chaîne Java depuis le code Nit. Pour sa part, `align` peut être passé à Java sous forme d'un point flottant, un type primitif aux deux langages, et transformé en une énumération Java dans le code Java. Cette organisation cache les préoccupations polyglottes aux clients et limite les rappels à Nit depuis le code étranger.

Le diagramme présenté à la figure 5.2 étend le diagramme précédent pour que le site d'appel invoque d'abord la méthode d'indirection `set_style`.

5.1.3 Applicabilité

Utilisez le patron *façade polyglotte* lorsque :

- Le code Nit invoque le code étranger.
- Des arguments ou le retour doivent être transformés entre les langages.
- Des arguments sont conservés de façon globale en Nit ou dans le code étranger.

5.1.4 Structure

De façon générale, le traitement et le travail voulu sont divisés entre les deux méthodes invoquées en chaîne. La structure est présentée à la figure 5.3.

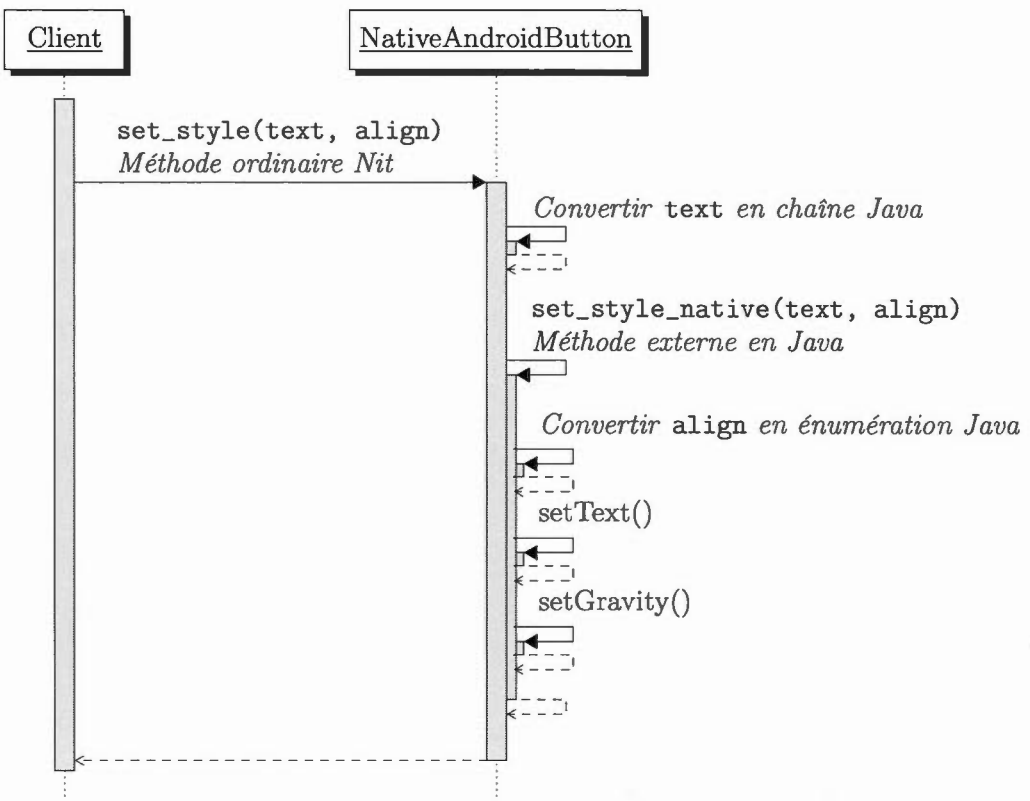


Figure 5.2: Application du patron *façade polyglotte* pour invoquer deux services.

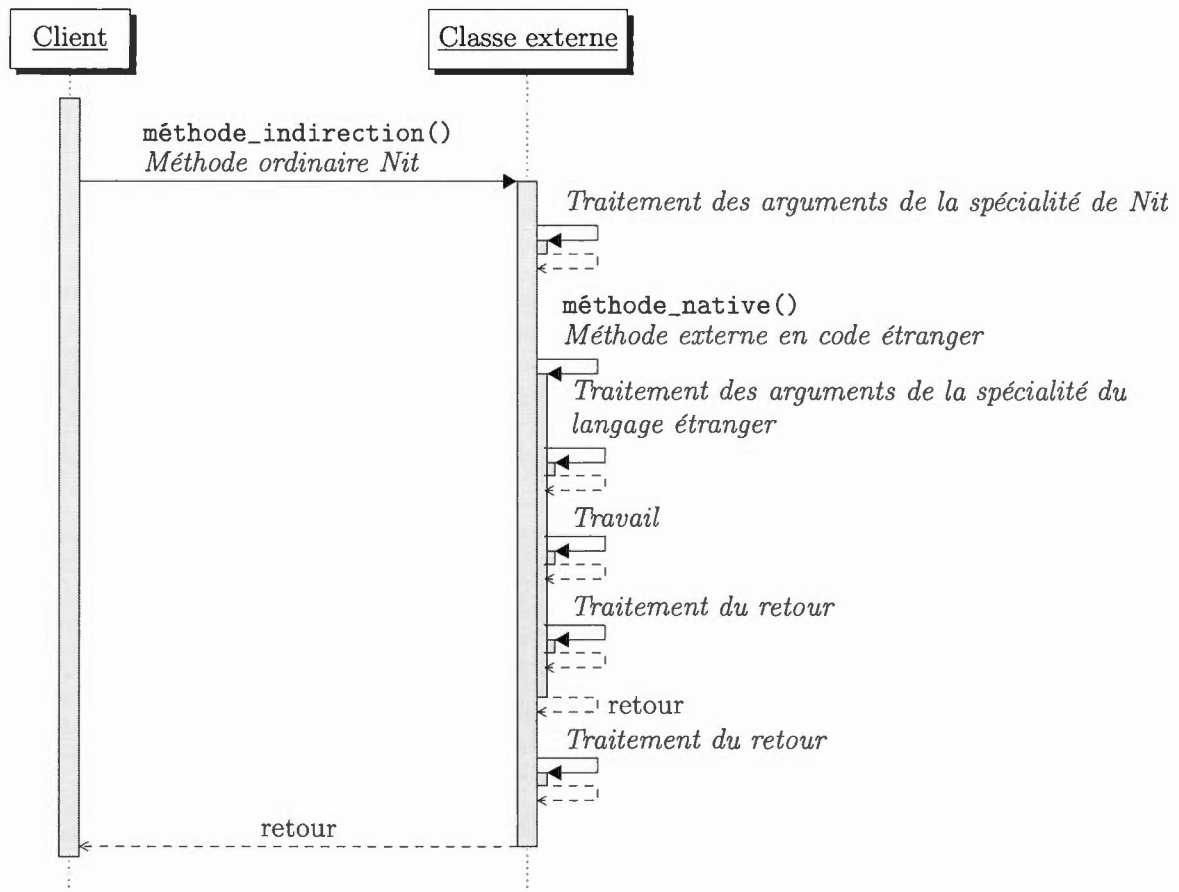


Figure 5.3: Structure du patron *façade polyglotte.*

5.1.5 Participants

Le client et une seule classe avec deux méthodes composent le patron.

Le client (l'appel à `set_style()`) invoque la méthode d'indirection.

Une classe externe (`NativeAndroidButton`) sert de receveur et définit les deux méthodes, l'indirection et l'externe. Cette classe est associée au type receveur dans le langage étranger, pour cette raison elle est généralement externe. Toutefois, si le receveur étranger est d'un type primitif correspondant à une énumération Nit (`Int`, `Float`, etc.), il est possible de raffiner l'énumération pour y ajouter les

méthodes. Ou encore, si il n'y a pas de receveur étranger, cette classe peut être ordinaire ou être une interface.

La méthode d'indirection (`set_style`) agit comme API publique, sa signature se compose uniquement de types Nit ordinaires : des types primitifs et des classes ordinaires. Elle invoque la méthode externe.

Elle traite les données qui sont de la spécialité de Nit. Le traitement appliqué aux données dépend du contexte, par exemple :

— Elle transforme les chaînes Nit en chaînes du langage étranger.

— Elle récupère des variables globales conservées dans l'environnement Nit.

— Elle traite les types nullables de Nit, au besoin les convertissant en types nullables du langage étranger.

La méthode externe (`set_style_native`) est privée et implémentée en C, Objective-C ou Java. Elle sert de frontière (ou d'interface) sur mesure entre Nit et le langage étranger. Elle n'est paramétrée que par des types ordinaires au langage étranger : des types primitifs ou des classes externes.

Elle traite les données qui sont de la spécialité du langage étranger, par exemple :

— Elle récupère des données depuis l'environnement étranger.

— Elle applique des opérations sur des énumérations Java.

— Elle traite les exceptions Java.

5.1.6 Collaboration

Le client invoque la méthode d'indirection qui, à son tour, invoque la méthode externe. La méthode externe exécute le travail et peut retourner une valeur qui remonte la chaîne d'appel.

5.1.7 Conséquences

Ce patron apporte plusieurs avantages :

- *La complexité d'utilisation de la FFI est cachée aux clients de l'API publique.* Les clients n'utilisent que la méthode d'indirection, ils ne manipulent que des types de la spécialité de Nit. Ils évitent le traitement des arguments, dont la conversion des chaînes entre les langages.
- *La frontière entre Nit et le langage étranger est isolée de l'API publique.* La signature de la méthode externe et l'unique site d'appel ne sont pas exposés aux clients de l'API. Cette séparation facilite la maintenance du code en permettant l'évolution de la frontière et du code étranger sans modifier l'API publique.
- *Les rappels à Nit depuis le code étranger sont limités.* Traiter les objets Nit du côté Nit seulement réduit l'utilisation des rappels à Nit qui complexifient le code et nécessitent une déclaration explicite.
- *Forte cohésion.* La séparation du traitement des données en deux méthodes améliore la cohésion des deux méthodes qui ont une relation claire et un rôle précis. De plus, la séparation réduit le couplage de la méthode externe avec d'autres services Nit.

Le principal désavantage est qu'au lieu d'utiliser une seule méthode externe, on ajoute la méthode d'indirection, doublant ainsi le nombre de méthodes nécessaires pour exécuter un fragment de code étranger.

5.1.8 Implémentation

- *Identifiants des méthodes.* Pour différencier la méthode d'indirection de la méthode externe, nous suggérons d'utiliser un nom semblable dont la seule différence est l'ajout du préfixe ou suffixe `native` à la méthode externe. Dans notre exemple, la méthode d'indirection `set_style` est accompagnée de la méthode externe `set_style_native`.

— *Où traiter une donnée ?* Pour chaque argument et valeur de retour qui doit être converti entre les langages, le programmeur doit choisir de le faire en Nit ou dans le langage étranger.

Préférez traiter une donnée en Nit lorsque :

— Elle est une chaîne de caractères, la bibliothèque de Nit offre plusieurs services pour convertir les chaînes entre les langages.

— Elle est conservée en attribut d'un objet Nit, extraire l'attribut de l'objet est plus simple en Nit.

— Elle est conservée globalement en Nit.

— Elle est d'un type complexe Nit, tel qu'un `Array`, la donnée peut être convertie en une classe externe équivalente.

— Elle nécessite un test de type Nit, comme les types nullables.

— Elle doit être convertie en un objet Nit, fréquent lors du retour d'un type primitif par la méthode externe.

— Elle est conservée en cache pour optimiser la performance.

Préférez traiter une donnée dans le code étranger lorsque :

— Elle est d'un type primitif étranger, sans équivalent en Nit. Tel que le `short` de Java, qui peut être passé par un `Int` Nit, et converti en `short` du côté Java.

— Elle est conservée dans un attribut ou globalement dans le code étranger.

— Il s'agit d'une énumération ou d'un autre type de donnée spécifique au code étranger.

— Elle sera conservée à long terme dans le code étranger. Si c'est un objet Nit, il doit être marqué comme tel auprès du ramasse-miettes de Nit.

— *Allouer un espace mémoire.* Il est commun pour les API C de déléguer la responsabilité aux clients d'allouer l'espace mémoire pour des chaînes de caractères, leurs services ne font alors que remplir l'espace avec des caractères. Dans ce cas, il est souvent préférable d'allouer l'espace avec le ramasse-miettes de Nit, depuis le code Nit, en instanciant `CString`. L'instance peut ensuite être passée au code C et au service qui y copiera les données. Il est important de ne pas déborder de l'espace alloué par le ramasse-miettes pour éviter une corruption de sa mémoire. L'espace mémoire sera libéré automatiquement par le ramasse-miettes lorsqu'il ne sera plus référencé depuis le code Nit.

— *Constructeur nity/natif.* Ce patron peut être appliqué par un constructeur d'une classe Nit ordinaire. Le constructeur sert alors de méthode d'indirection, il traite les arguments de la spécialité de Nit et invoque une méthode externe.

5.1.9 Exemple de code

Nous réutilisons l'exemple de la méthode `set_style` de `NativeAndroidButton` pour illustrer l'implémentation de ce patron.

1. La classe `NativeAndroidButton` définit la méthode d'indirection et la méthode externe. Elle a un constructeur externe qui prend un seul argument, le contexte de l'application attendu par le constructeur Java. Le constructeur aussi pourrait bénéficier du patron, mais par simplicité nous le définissons comme un simple constructeur externe.

```
1 extern class NativeAndroidButton `{ android.widget.button `}
2   super NativeAndroidTextView
3
4   new (context: NativeContext)
5     return new android.widget.Button(context);
6   `}
```

2. La méthode d'indirection traite l'argument `text` qui est converti en chaîne Java à l'aide d'un service de la bibliothèque de Nit, et fait appel à la méthode externe.

```
7   # Assigne le texte à afficher et l'alignement
8   fun set_style(text: String, align: Float)
9   do
10    # Convertit l'argument `text` en chaîne Java
11    var java_text = text.to_java_string
12
13    # Appelle la méthode externe
14    native_set_style(java_text, align)
15  end
```

3. La méthode externe est privée. Elle traite l'argument `align` qui est passé à Java en tant que point flottant pour être ensuite converti en une énumération Java. Puis, elle invoque les deux services visés de l'API Android.

```
17   private fun native_set_style(text: JavaString, align: Float)
18   in "Java" `{
19     // Traite l'argument `align`
20     int gravity;
21     if (align == 0.5d)
22       gravity = android.view.Gravity.CENTER_HORIZONTAL;
23     else if (align < 0.5d)
24       gravity = android.view.Gravity.LEFT;
25     else
26       gravity = android.view.Gravity.RIGHT;
27
28     // Exécute le travail
29     self.setText(text);
30     self.setGravity(gravity);
31   `}
32 end
```

4. Finalement, le site d'appel instancie un bouton. (L'argument du constructeur aurait pu être récupéré par une méthode d'indirection si on y avait appliqué le patron.) Ensuite, le site d'appel invoque la méthode d'indirection en ne lui passant que des types ordinaires à Nit.

```
33 var button = new NativeAndroidButton(app.native_activity)
34 button.set_style("Étiquette", 1.0)
```

5.1.10 Utilisations connues

— Le patron *façade polyglotte* est utilisé dans les applications *app.nit* pour exécuter des fragments de code étranger. L'exemple `set_style` est dérivé de l'adaptation pour Android de la calculatrice *app.nit*, dans le code original, les constructeurs des boutons et des entrées de texte remplacent la méthode d'indirection.

— Dans l'implémentation de l'API requêtes HTTP pour Android, le patron apparaît entre les méthodes `http_get` et `java_http_get`. Ces deux méthodes ont déjà été présentées par les figures 4.19 et 4.21 (aux pages 137 et 140) pour illustrer la gestion des cadres et des exceptions Java. La méthode `http_get` sert d'indirection, elle appelle `java_http_get` et traite son retour par d'autres appels à des méthodes externes, `java_http_get` fait appel aux services de requêtes HTTP offerts par l'API Android.

5.1.11 Patrons connexes

— Le patron *façade* cache la complexité d'une interface derrière une nouvelle interface plus simple. De façon similaire, le patron *façade polyglotte* camoufle la complexité d'une interface étrangère et la complexité de la FFI de Nit derrière une interface plus simple.

— Le patron *chaîne de responsabilité* partage le traitement d'une seule requête par plusieurs méthodes. Dans le cas de *façade polyglotte*, deux méthodes traitent les arguments et la valeur de retour selon la spécialité de leur langage d'implémentation. Alors que *façade polyglotte* n'implique qu'un objet avec deux méthodes qui sont toujours exécutées, *chaîne de responsabilité* accepte plusieurs objets et toutes les méthodes de la chaîne ne sont pas exécutées.

— Le patron *adaptateur polyglotte* applique le patron *façade polyglotte* pour chaque méthode d'une API à objets.

5.2 Patron – Adaptateur polyglotte

5.2.1 Intention

Reproduire une API étrangère à objets en Nit.

5.2.2 Motivation

Les méthodes externes permettent d'accéder directement aux API en C, Objective-C ou Java. Toutefois, lorsque l'API étrangère est à objets, le code client en Nit peut avoir à conserver des objets étrangers pour des appels subséquents. Ce besoin entraîne l'utilisation des fonctionnalités complexes de la FFI de Nit, dont la gestion de la mémoire. Lorsque les API étrangères sont utilisées fréquemment, il peut être bénéfique de les reproduire en Nit et de camoufler l'utilisation de la FFI aux clients pour qu'ils ne manipulent que des données ordinaires en Nit.

Par exemple, pour adapter une application mobile à Android, le programmeur doit utiliser l'API UI native d'Android. Il peut le faire directement via des méthodes externes ou alors reproduire l'API en Nit. Dans cet exemple, nous nous intéressons à un sous-ensemble de l'API UI d'Android composé de trois classes et de quelques méthodes. Les trois classes, `View`, `TextView` et `Button`, sont présentées en relation d'héritage à la figure 5.4.

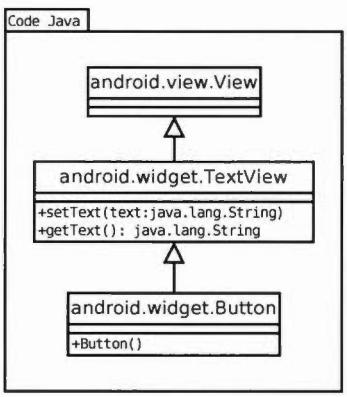


Figure 5.4: Diagramme de classe d'un sous-ensemble de l'API UI d'Android.

Alors que le patron *façade polyglotte* indique comment gérer le passage des données, il ne répond pas à des questions spécifiques aux API à objets. Comment conserver une référence à un bouton depuis le code Nit à long terme ? Quand libérer la référence ?

La solution consiste à utiliser la FFI pour accéder à l'API native et en camoufler la complexité derrière une API publique. Pour chaque classe étrangère (telle que `android.widget.Button`), ce patron y associe deux classes Nit : une classe externe et une classe ordinaire Nit.

— La classe externe `NativeAndroidButton` est associée à la classe Java `android.widget.Button`. Elle en reproduit l'API aussi fidèlement que possible par des méthodes externes.

— La classe ordinaire `AndroidButton` encapsule une instance de `android.widget.Button` dans un attribut typé par `NativeAndroidButton` Elle reproduit l'API native avec des méthodes d'indirection et elle en adapte les services pour qu'ils soient plus naturels en Nit.

Pour reproduire seulement la classe Java `android.widget.Button` et une seule de ses méthodes, nous définissons les deux classes Nit en gris à la figure 5.5.

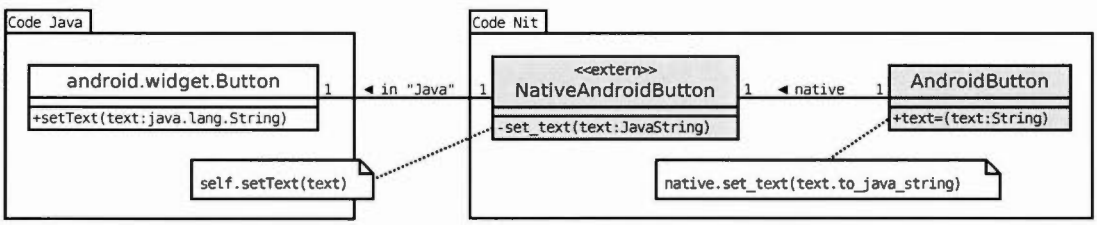


Figure 5.5: Reproduction de la classe `android.widget.Button` en Nit.
La classe blanche provient de l'API originale et les classes grisées réalisent le patron. On remarque que la méthode externe `set_text` est implémentée en Java, alors que la méthode d'indirection `text=` est implémentée en Nit.

Pour reproduire l'API entière, chaque classe Java nécessite deux classes Nit et chaque service Java deux méthodes Nit, tel qu'illustré à la figure 5.6. Seule la classe à la racine de la hiérarchie doit déclarer un attribut, et les méthodes `set_text` et `text=` peuvent être remontées dans les classes associées à `android.widget.TextView`.

5.2.3 Applicabilité

Utilisez le patron *adaptateur polyglotte* lorsque :

— L'API étrangère est articulée autour d'objets ou de structures C.
— Des objets de l'API étrangère doivent être conservés dans le code Nit.
— L'API étrangère est utilisée à plusieurs endroits dans le code.

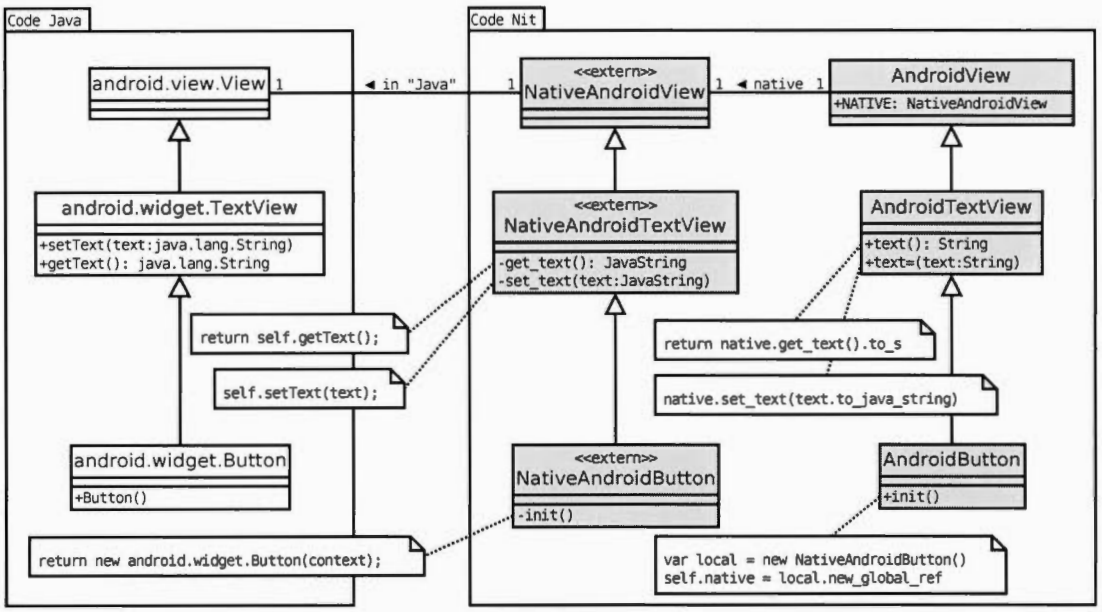


Figure 5.6: Reproduction d'un sous-ensemble de l'API UI d'Android en Nit.

5.2.4 Structure

Le diagramme de la figure 5.7 présente la structure générale du patron *adaptateur polyglotte*.

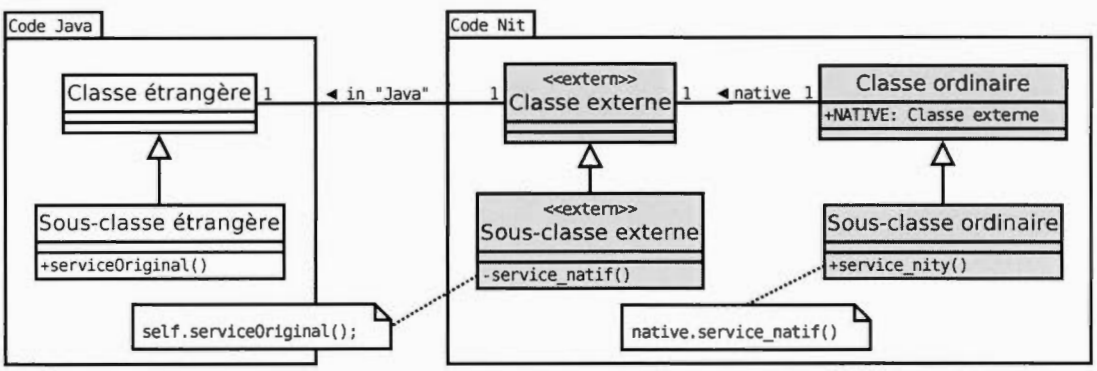


Figure 5.7: Structure du patron *adaptateur polyglotte*.

5.2.5 Participants

Deux classes Nit pour chaque classe étrangère composent le patron.

La classe étrangère (`android.widget.Button`) offre les services de l'API originale sous forme de constructeurs, méthodes et attributs. Cette classe est en pur Objective-C ou Java, ou alors il peut s'agir d'une structure C. Elle est offerte par une API native ou par une API tierce partie.

La classe externe (`NativeAndroidButton`) est associée à la classe étrangère.

Elle regroupe des méthodes externes pour accéder aux services de la classe étrangère dont elle reproduit aussi fidèlement que possible les identifiants et la forme. Tout comme avec le patron *façade polyglotte*, la signature des méthodes externes est composée uniquement de types naturels au langage étranger.

La classe externe ne fait pas partie de l'API publique. Elle est soit déclarée comme étant privée, soit son importation est privée de façon à ne pas être visible aux clients de l'API publique.

La classe ordinaire (`AndroidButton`) reproduit les services natifs par des méthodes Nit ordinaires. Elle agit comme API publique et expose uniquement des types ordinaires en Nit.

Ses méthodes ont les mêmes responsabilités que les méthodes d'indirection du patron *façade polyglotte*. Elles servent d'indirection vers les méthodes externes de la classe externe, elles convertissent les types ordinaires Nit en types ordinaires au langage étranger. Entre autres, elles transforment les chaînes Nit en chaînes étrangères.

La classe ordinaire encapsule une instance de la classe étrangère dans un attribut privé. De plus, elle a la responsabilité de libérer l'attribut automatiquement.

5.2.6 Collaboration

— Le seul attribut de la classe ordinaire est typé par la classe externe et il contient une instance de la classe étrangère.

— Autant la classe externe que la classe ordinaire reproduisent le graphe d'héritage de l'API originale et de la classe étrangère.

5.2.7 Conséquences

Ce patron partage les mêmes avantages que le patron façade polyglotte, en plus des avantages suivants :

— *Les clients ne manipulent que des classes ordinaires.* La classe ordinaire peut être manipulée naturellement depuis le code Nit et ainsi, une référence à l'objet de l'API étrangère être conservée en Nit.
— *Libération automatique des objets étrangers.* La classe externe étant encapsulée, ses instances peuvent être libérées automatiquement par la classe ordinaire.
— *Faible couplage de la classe externe.* L'encapsulation de la classe externe en réduit le couplage avec d'autres services Nit.

On dénote deux désavantages :

— *Les classes externes ne sont pas visibles aux clients.* Les clients ne peuvent pas utiliser les classes externes pour accéder à d'autres services qui ne seraient pas déjà reproduits en Nit. Cette limitation peut être contournée par une importation optionnelle des classes externes.
— *Le dédoublement des classes Nit nécessite beaucoup de code.* En fait, ce patron peut être inutilement complexe pour des accès simples à des services natifs.

5.2.8 Implémentation

— *Graphe d'héritage.* Autant les classes ordinaires que les classes externes doivent reproduire le graphe d'héritage des classes étrangères de l'API originale. Ceci permet de bien reproduire l'héritage des services de l'API originale par les méthodes Nit et d'éviter la duplication de code.
— *Type virtuel* `NATIVE`. Le langage Nit offre les types virtuels, des types définis comme des propriétés de classes. Ils agissent de façon similaire aux types paramètres des classes génériques, mais ils sont définis par chaque classe au lieu d'être défini à l'instanciation.

Un type virtuel peut servir à typer l'attribut de la classe ordinaire de façon à faire évoluer le type de l'attribut avec les sous-classes. Nous lui donnons habituellement le nom de `NATIVE` et il sert de type à l'attribut `native`. L'utilisation du type virtuel facilite la reproduction du graphe d'héritage et l'association de chaque classe ordinaire avec la classe externe correspondante.

— *Adapter l'API à Nit.* Le programmeur peut prendre plus de liberté dans la forme de l'API publique, c'est-à-dire la signature des méthodes d'indirection et la forme des classes ordinaires, que dans la forme des classes externes. Il peut définir une API de plus haut niveau, par exemple, en supportant les itérateurs de Nit. Ou encore, il peut offrir une approche différente, telle qu'une API fluide.

— *Identifiants des classes.* Pour différencier les classes ordinaires des classes externes, nous suggérons d'ajouter le préfixe `Native` au nom de la classe externe. Ceci rappelle le patron *façade polyglotte* et distingue clairement les classes dans le code Nit. Il est alors superflu d'ajouter un tel préfixe ou suffixe aux méthodes externes.

— *Cohésion des modules.* Nous recommandons de réaliser chaque couche dans son propre module pour une bonne cohésion. C'est-à-dire, regrouper les classes externes dans un même module, et les classes ordinaires dans un autre module. Cette séparation permet d'importer de façon privée le module des classes externes depuis le module des classes ordinaires. Ceci n'expose pas les classes externes aux clients de l'API, tout en leur donnant l'alternative d'importer le module directement pour avoir accès aux classes externes.

— *Attributs et champs.* L'accès à un attribut d'une classe étrangère est réalisé en Nit par des mutateurs et accesseurs seulement, l'attribut étant toujours conservé dans l'API étrangère. Sur la classe externe, une méthode externe obtient et retourne l'attribut de l'objet étranger, une autre assigne une valeur à l'attribut. Sur la classe ordinaire, deux méthodes servent d'indirection.

— *Libérer l'objet natif avec l'objet Nit.* Si les objets étrangers sont bien encapsulés par la classe ordinaire, il est possible d'en automatiser la libération. Pour ce faire, la classe ordinaire doit spécialiser `Finalizable` et implémenter la méthode `finalize` de façon à ce qu'elle libère l'objet étranger. Cette méthode est invoquée lorsque l'objet Nit est libéré par le ramasse-miettes. Toutefois, ces services ont des limites, tel que précédemment discuté à la section 4.3.6.2.

— *Couche d'abstraction.* Il y a souvent une troisième couche qui s'ajoute à ce patron, la couche d'abstraction. Par exemple, le patron *adaptateur polyglotte* peut servir à reproduire les services d'Android en Nit et à implémenter l'API abstraite de *app.nit*. Dans ce cas et lorsque possible, nous recommandons d'implémenter l'API abstraite par les classes ordinaires. À ce moment, les classes externes peuvent être partagées avec une autre abstraction ou une API publique.

— *Répondre à des événements.* Au besoin, la classe ordinaire peut supporter des rappels lors d'événements. Par exemple, pour répondre à la pression d'un bouton de l'interface utilisateur. Le patron *spécialisation polyglotte* offrira une solution à ce problème.

— *Génération de code.* Les classes externes sont dérivées directement des classes étrangères et leur réalisation est répétitive. Heureusement, ce travail répétitif peut parfois être automatisé. Nous présenterons deux générateurs de code à la section 5.4, pour les API Java et Objective-C

5.2.9 Exemple de code

Le sous-ensemble de l'API UI d'Android peut être reproduit par trois classes ordinaires et trois classes externes.

1. Une classe externe est associée à chaque classe Java de l'API native. Dans ce cas, la classe `NativeAndroidView` agit comme racine de le hiérarchie de classes.

```
1 module native_android_ui
2
3 extern class NativeAndroidView in "Java" `{ android.view.View `}
4   super JavaObject
5 end
```

2. La classe `NativeAndroidTextView` reproduit la hiérarchie des classes Java et importe `NativeAndroidView`. Elle définit deux méthodes externes en conservant le style de l'API originale autant que possible. Les méthodes externes peuvent appliquer des traitements aux types naturels à Java, dans ce cas, `NativeAndroidTextView::text` traite la valeur de retour de façon à convertir un `java.lang.CharSequence` en `java.lang.String`.

```
6 extern class NativeAndroidTextView in "Java" `{android.widget.TextView`}
7   super NativeAndroidView
8
9   fun get_text: JavaString in "Java" `{
10    return self.getText().toString();
11  `}
12
13  fun set_text(value: JavaString) in "Java" `{
14    self.setText(value);
15  `}
16 end
```

3. La dernière classe externe est la seule qui est instanciée, elle déclare donc un constructeur externe. Elle hérite des méthodes `get_text` et `set_text` de `NativeAndroidTextView`.

```
12 extern class NativeAndroidButton in "Java" `{android.widget.Button`}
13   super NativeAndroidTextView
14
15   new (context: NativeActivity)
16     return new android.widget.Button(context);
17   `}
18 end
```

4. Le module déclarant les classes ordinaires débute par l'importation privée des classes externes. De cette façon, elles ne seront pas visibles aux clients de l'API publique qui ne verront que les classes ordinaires.

```
1 module android_ui
2
3 private import native_android_ui
```

5. Les classes ordinaires reproduisent chaque classe de l'API et leur héritage. La racine de l'héritage (`AndroidView`) définit un attribut `native` qui conserve une instance à l'objet étranger. L'attribut `native` est typé par le type virtuel `NATIVE` qui est associé à la classe externe.

```
4 class AndroidView
5   type NATIVE: NativeAndroidView
6   var native: NATIVE
7 end
```

6. Les autres classes ordinaires spécialisent `AndroidView` et redéfinissent `NATIVE` avec le type précis de la classe externe correspondante. Elles reproduisent les services de l'API originale avec plus de liberté que les classes externes, par exemple, `getText` et `setText` peuvent être remplacés par des accesseurs de style Nit. Ceux-ci convertissent les données d'une chaîne Java à une chaîne Nit, et vice versa.

```
 8 class AndroidTextView
 9   super View
10   redef type NATIVE: NativeTextView
11
12   fun text: String do return native.get_text.to_s
13   fun text=(value: String) do native.set_text(value.to_java_string)
14 end
```

7. À sa construction, la classe `Button` crée une instance de la classe Java `android.widget.Button`. Elle en conserve une référence globale dans l'attribut `native`.

```
14 class AndroidButton
15   super AndroidTextView
16
17   redef type NATIVE: NativeAndroidButton
18
19   init do
20     var local = new NativeAndroidButton(app.native_activity)
21     self.native = local.new_global_ref
22   end
23 end
```

8. Le client peut créer et manipuler une instance de la classe `Button` naturellement.

```
1 var button = new Button
2 button.text = "Mon bouton"
```

9. Pour libérer automatiquement la référence globale conservée par l'attribut `native`, la classe `AndroidView` peut spécialiser `Finalizable` et redéfinir `finalize`. Le fragment de code suivant étend la définition de `AndroidView` présentée plus haut.

```
 5 class AndroidView
 6   super Finalizable
 7
 8   type NATIVE: NativeAndroidView
 9   var native: NATIVE
10
11   # Libère la référence Java avec l'objet Nit
12   redef fun finalize do native.delete_global_ref
13 end
```

5.2.10 Utilisations connues

— Les exemples précédents sont fortement inspirés de l'implémentation de l'API UI sous Android, la principale différence est que le code original implémente l'abstraction de *app.nit*.

— Le paquet `sqlite` de la bibliothèque de Nit reproduit les services de l'API C du gestionnaire de base de données SQLite. Il applique le patron en deux modules : `native_sqlite` et `sqlite`.

Le module `native_sqlite` reproduit fidèlement les services de l'API C par des classes et des méthodes externes. Il est réalisé de façon neutre pour supporter plusieurs clients, le module `sqlite` ainsi qu'une abstraction future des gestionnaires de base de données.

Le module `sqlite` définit l'API publique du paquet. Il convertit les chaînes de caractères et remplace les pointeurs C nuls par des nullables de Nit. Il offre d'autres services de plus haut niveau, comme des itérateurs pour parcourir les résultats d'une requête. Il en profite pour automatiser la libération des structures C à la fin de l'exécution de l'itérateur.

5.2.11 Patrons connexes

— Le patron *adaptateur* modifie l'interface d'une classe de façon à correspondre à l'interface attendue par un client. De façon similaire, *adaptateur polyglotte* reproduit une API Java en une API Nit pour des clients Nit.

— Ce patron applique la même stratégie que *façade polyglotte* pour gérer le passage de données entre Nit et le langage étranger. Par contre, alors que *façade polyglotte* permet l'usage de grosses méthodes externes qui réalisent plusieurs appels à des services étrangers, *adaptateur polyglotte* encourage d'associer chaque service étranger à une seule méthode Nit.

— Le patron *adaptateur polyglotte* ne répond pas aux besoins des API étrangères dont les clients doivent spécialiser des interfaces pour personnaliser un comportement. Dans un tel cas, il est préférable d'utiliser le patron *spécialisation polyglotte.*

5.3 Patron – Spécialisation polyglotte

5.3.1 Intention

Spécialiser une classe Objective-C ou Java en Nit.

5.3.2 Motivation

Les API à objets de Objective-C et de Java utilisent des interfaces qui peuvent être spécialisées par les clients pour personnaliser un comportement ou répondre à des événements. Cette approche évite au client d'aller chercher les événements, c'est le système qui invoque le client lors d'événements.

Par exemple, pour répondre à la pression d'un bouton sous Android, on doit implémenter l'interface `OnClickListener` et sa méthode `onClick`. En Java pur, le programmeur n'aurait qu'à créer une sous-classe `ClientListener` et personnaliser `onClick`, tel que présenté dans la figure 5.8.

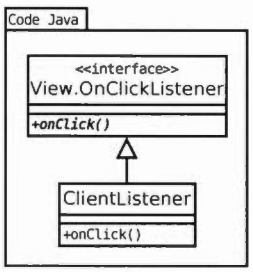


Figure 5.8: Spécialisation de `View.OnClickListener` en Java pur.

On aimerait faire l'équivalent depuis Nit, créer une sous-classe Nit à une classe Java. Malheureusement, simplement sous-classer une classe externe associée à `OnClickListener` ne suffit pas. Les appels à `OnClickListener` proviennent du code Java et sont reçus par l'objet Java, ils ne sont pas automatiquement transférés au code Nit.

Le patron *spécialisation polyglotte* propose une solution pour recevoir les appels en Java et les transmettre au code Nit. On ajoute une sous-classe Java à `View.OnClickListener`, nommons-la `NitOnClickListener`. Cette dernière relaie tous les appels à la méthode `onClick` à un objet Nit. Le passage de Java à Nit prend alors la forme de rappels à Nit générés pas la FFI de Nit. La sous-classe `NitOnClickListener` contient un seul attribut, une référence à l'objet Nit.

L'objet Nit est une instance de l'interface `AndroidOnClickListener` qui définit une seule méthode, `on_click`, le rappel attendu de Java. De cette façon, les appels de méthode depuis Java sont d'abord traités par le polymorphisme de Java, et ensuite par celui de Nit.

Après l'application du patron, les clients en Nit n'ont qu'à sous-classer `AndroidOnClickListener` et redéfinir la méthode `on_click` pour en personnaliser le comportement. Le diagramme de classe de la figure 5.9 représente l'application du patron dans notre exemple.

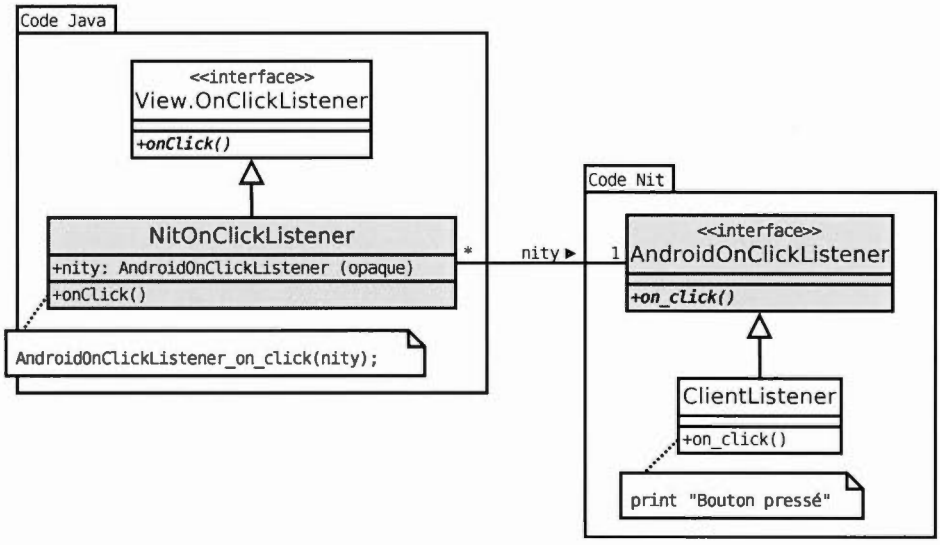


Figure 5.9: Spécialisation de `View.OnClickListener` en Nit.

Encore une fois, les classes grisées ont été ajoutées pour réaliser le patron. Les deux classes blanches sont celles de l'API originale en Java (`View.OnClickListener`) et celle du client de l'API Nit (`ClientListener`).

5.3.3 Applicabilité

Utilisez le patron *spécialisation polyglotte* lorsqu'une API étrangère emploie des interfaces à spécialiser par le client, ou des méthodes à redéfinir pour modifier un comportement.

5.3.4 Structure

Le diagramme de la figure 5.10 présente la structure générale du patron *spécialisation polyglotte.*

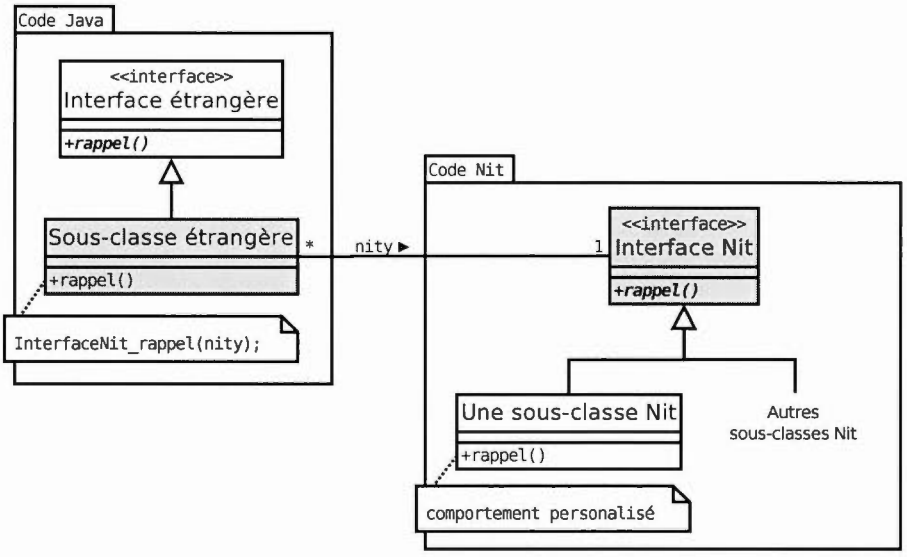


Figure 5.10: Structure du patron *spécialisation polyglotte.*

5.3.5 Participants

L'interface étrangère (`View.OnClickListener`) provient de l'API à objets originale. Elle définit des méthodes qui sont à implémenter par les sous-classes.

La sous-classe étrangère (`NitOnClickListener`) spécialise l'interface étrangère. Elle a un seul attribut, un objet Nit typé par l'interface Nit. La sous-classe étrangère implémente les méthodes de l'interface étrangère de façon à relayer tous les appels à l'objet Nit.

L'interface Nit (`AndroidOnClickListener`) déclare les mêmes méthodes que l'interface étrangère. Il peut s'agir aussi d'une classe ordinaire, mais pas d'une classe externe. Ses méthodes peuvent être abstraites ou avoir un comportement par défaut.

Les sous-classes Nit (`ClientOnClickListener`) sont créées par les clients de l'API Nit pour spécialiser l'interface Nit. Elles implémentent les méthodes de l'interface Nit pour en personnaliser le comportement.

5.3.6 Collaboration

— Un appel depuis Java à l'interface Java est traité par la sous-classe étrangère qui le relaie à l'objet Nit conservé en attribut. L'appel à l'objet Nit est traité par le polymorphisme ordinaire de Nit pour exécuter l'implémentation la plus précise de la méthode visée.
— Seuls les clients définissent les sous-classes Nit et le comportement exécuté.

5.3.7 Conséquences

Le patron *spécialisation polyglotte* apporte deux avantages :

— *Réaliser des sous-classes en Nit pur.* Les clients de l'API Nit peuvent spécialiser les interfaces d'une API étrangère et en personnaliser le comportement sans toucher aux rappels à Nit de la FFI.
— *Libérer l'objet Nit.* La sous-classe étrangère peut libérer l'objet Nit automatiquement lorsqu'il ne sera plus utilisé par l'API étrangère.

Le principal désavantage, encore une fois, est que ce patron nécessite d'implémenter plusieurs classes pour chaque classe étrangère concernée.

5.3.8 Implémentation

— *Conversion des données.* S'il y a un traitement à appliquer sur les données lors des rappels à Nit, on peut appliquer une variation du patron *façade polyglotte.* Dans ce cas, la méthode dans la sous-classe Java peut traiter les données naturelles à Java. On ajoute une méthode d'indirection supplémentaire à l'interface Nit pour traiter les données naturelles à Nit et invoquer la méthode abstraite redéfinie par les sous-classes Nit. De cette façon, les sous-classes ne manipulent que des données naturelles à Nit.

— *Ramasse-miettes Nit.* Pour s'assurer que l'objet Nit préservé par la sous-classe étrangère ne soit pas libéré, son compteur de référence doit être incrémenté. Le compteur doit être décrémenté lorsque la sous-classe étrangère n'est plus utilisée. En général, il est possible de profiter d'un rappel de l'API étrangère indiquant la libération de l'objet étranger.

— *Classe anonyme Java.* En Java, la sous-classe étrangère peut être réalisée de deux façons. Le plus simple est d'en faire une classe anonyme qui est donc déclarée et instanciée du même coup. L'alternative, lorsque la classe doit être initialisé depuis plusieurs endroits, est d'utiliser un bloc d'entête `in "Java Inner"` pour déclarer une classe interne.

— *Fermeture Objective-C.* Lorsqu'un API Objective-C utilise des fermetures pour exécuter les rappels, il se peut que la sous-classe externe soit superflue. La fermeture peut exécuter le rappel à Nit et libérer l'objet au besoin.

— *Pointeur de fonction C.* Il n'y a pas d'interfaces ou de méthodes à redéfinir en C, les API C utilisent plutôt des pointeurs de fonctions. Les clients peuvent alors passer une référence à une fonction locale pour personnaliser le comportement d'un API.

  Le patron *spécialisation polyglotte* s'applique différemment dans un tel cas. On conserve l'interface Nit et ses sous-classes, par contre, on remplace la sous-classe étrangère par une seule fonction C. Celle-ci porte la signature attendue par l'API C, et relaie les appels à un objet Nit typé par l'interface Nit.

  En général, l'objet Nit peut être conservé par l'API C dans un champ *user data*. L'objet sera passé à la fonction C à chaque rappel. Sinon, l'objet Nit peut être conservé globalement, mais dans ce cas il n'y a qu'un receveur par fonction C.

5.3.9 Exemple de code

Notre exemple de réponse à la pression d'un bouton sous Android peut être implémenté par un seul module Nit.

1. Les importations Java sont déclarées dans un bloc `in "Java"`.

```
1 module button_event
2
3 in "Java" `{
4   import android.view.View;
5   import android.widget.Button;
6 `}
```

2. La sous-classe Java est déclarée dans un bloc `in "Java Inner"`. Elle conserve une référence à l'objet Nit.

```
7 in "Java Inner" `{
8
9   private class NitOnClickListener
10    implements View.OnClickListener {
11
12    // Objet Nit de type `AndroidOnClickListener`
13    private int nit_listener;
14
15    // Constructeur
16    NitOnClickListener(int nit_listener) {
17      self.nit_listener = nit_listener;
18    }
```

3. On y implémente le rappel `onClick` qui est relayé à l'objet `nit_listener` par un rappel à Nit.

```
19    @Override
20    void onClick(View view) {
21      // Rappel à `on_click` sur l'objet Nit `nit_listener`
22      AndroidOnClickListener_on_click(nit_listener);
23    }
24 `}
```

4. De retour en Nit, on définit l'interface Nit avec une seule méthode `on_click`.

```
25 # Receveur d'événement lors de pression d'un bouton
26 interface AndroidOnClickListener
27   fun on_click is abstract
28 end
```

5. Une méthode externe `set_on_click_listener` instancie la sous-classe Java et l'associe au bouton. Elle peut être déclarée à plusieurs endroits, mais nous la plaçons dans `NativeAndroidButton` pour correspondre à l'API originale. Cette méthode doit aussi déclarer les rappels à Nit.

```
29 extern class NativeAndroidButton `{ android.widget.Button `}
30   super NativeAndroidTextView
31
32   new (context: NativeActivity)
33     return new android.widget.Button(context);
34   `}
35
36   fun set_on_click_listener(nit_listener: AndroidOnClickListener)
37   import AndroidOnClickListener.on_click in "Java" `{
38
39     // Création d'une instance de la sous-classe Java
40     NitOnClickListener java_listener = new NitOnClickListener(nit_listener);
41
42     // Assigne notre instance pour répondre aux événements
43     self.setOnClickListener(java_listener);
44   `}
45 end
```

6. Un client spécialise l'interface Nit et personnalise le comportement de `on_click`. Ensuite, il instancie la classe et l'associe à un bouton.

```
1 module client
2
3 import button_event
4
5 class ClientOnClickListener
6   super AndroidOnClickListener
7
8   redef fun on_click do print "Bouton pressé!"
9 end
10
11 var button = new NativeAndroidButton(context)
12 button.set_on_click_listener = new ClientOnClickListener
```

7. Finalement, il y a deux façons de libérer l'objet Nit préservé par les instances Java de `NitOnClickListener`. Il peut être libéré au premier rappel ou avec l'objet étranger.

   Si le bouton n'est cliqué qu'une seule fois (tel que pour fermer la fenêtre), le compteur de référence peut être incrémenté dans `set_on_click_listener` et décrémenté dans `onClick` avec les deux fonctions :

```
1 AndroidOnClickListener_incr_ref(nit_listener);
2 AndroidOnClickListener_decr_ref(nit_listener);
```

Dans cet exemple, l'alternative est d'utiliser un second rappel de l'API Android, la méthode `onDetachedFromWindow` est invoquée lorsque le bouton n'est plus visible. Le rappel peut être personnalisé de façon à libérer la référence à `nit_listener`. Malheureusement, la forme de l'API Android nous force à appliquer le patron deux fois. La figure 5.11 présente une implémentation alternative utilisant des classes anonymes comme sous-classes étrangères à `View.OnClickListener` et à `Button`, ainsi qu'une variable `final` au lieu d'un attribut. Elle remplace plusieurs étapes (2, 3 et 5) par une seule classe externe et un seul constructeur, toutefois, elle utilise encore la classe Nit `AndroidOnClickListener`.

5.3.10 Utilisations connues

— Encore une fois, l'exemple de la pression de boutons sous Android est fortement inspiré de l'implémentation de l'API UI de *app.nit* sous Android où elle prend une forme semblable à l'alternative présentée par la figure 5.11.

— L'API cycle de vie sous Android et iOS applique ce patron pour répondre aux messages des systèmes d'exploitation. Les messages de Android et de iOS sont reçus par une classe étrangère et transférés à l'application Nit. Ces messages sont traités par les clients qui spécialisent les méthodes `on_create`, `on_resume`, etc.

— Le module `libevent` de la bibliothèque Nit couvre l'API C de la bibliothèque native du même nom. Il offre des services pour répondre à des événements réseau et autres, il est utilisé notamment pour implémenter un serveur HTTP. Ce module applique la variation utilisant des pointeurs de fonctions C.

5.3.11 Patrons connexes

— Dans certains cas, le patron *spécialisation polyglotte* est similaire au patron de conception *patron de méthode*. Ce dernier laisse les sous-classes personnaliser certaines étapes d'un algorithme. Il vise surtout les algorithmes avec plusieurs rappels, mais la logique est similaire pour de simples rappels. Quant au patron *spécialisation polyglotte*, il permet à des sous-classes Nit de personnaliser certaines étapes d'un algorithme d'une API étrangère.

```
1 extern class NativeAndroidButton `{ android.widget.Button `}
2   super NativeAndroidTextView
3
4   new (context: NativeActivity, nit_listener: AndroidOnClickListener)
5   import AndroidOnClickListener.on_click in "Java" `{
6
7     // Conserve la référence à long terme
8     final int final_nit_listener = nit_listener;
9
10    // Incrémente le compteur de référence
11    AndroidOnClickListener_incr_ref(final_sender_object);
12
13    // Sous-classe anonyme pour répondre à `onClick`
14    View.OnClickListener java_listener = new View.OnClickListener() {
15      @Override
16      void onClick(View view) {
17        // Rappel à `on_click` sur l'objet Nit
18        AndroidOnClickListener_on_click(final_nit_listener);
19      }
20    };
21
22    // Sous-classe anonyme pour la libération
23    Button java_button = new Button(context) {
24      @Override
25      void onDetachedFromWindow() {
26        // Décrémente le compteur de référence
27        AndroidOnClickListener_decr_ref(final_nit_listener);
28      }
29    };
30
31    // Assigne notre instance pour répondre à
32    // la pression du bouton
33    java_button.setOnClickListener(java_listener);
34  `}
35 end
```

Figure 5.11: Implémentation alternative utilisant des classes anonymes Java.

— Le patron *adaptateur* convertit l'interface d'une classe de façon à correspondre à l'interface attendue par un client. De façon similaire, *spécialisation polyglotte* reproduit une interface Nit en une interface Java pour qu'elle soit invoquée depuis Java.
— Ce patron est l'inverse du patron *adaptateur polyglotte*, alors que ce dernier permet d'invoquer les services étrangers depuis Nit, *spécialisation polyglotte* permet d'invoquer des services Nit depuis le code étranger.

5.3.12 Travaux futurs – Patron unique

Nous croyons qu'il serait possible de concevoir un patron unique pour lier *adaptateur polyglotte* et *spécialisation polyglotte*. Pour ce faire, la sous-classe externe devrait conserver une référence à l'interface Nit et l'interface Nit devrait conserver une référence à la sous-classe externe. Ce patron unique répondrait aux mêmes problèmes et apporterait les mêmes avantages que les deux patrons présentés plus haut, en plus de supporter les appels aux méthodes depuis les deux langages.

Nous laissons l'exploration de ce patron à un travail futur, car il est inutilement complexe pour les cas que nous avons rencontrés en pratique. Toutefois, si un même patron s'appliquait à toutes les API étrangères, il pourrait être généré automatiquement.

5.4 Générateurs de code

Nous avons déjà noté que la réalisation des classes externes du patron *adaptateur polyglotte* pourrait être automatisée. En fait, dès qu'il s'agit de reproduire mécaniquement des API étrangères en Nit, le code peut être généré par un outil car il n'y a pas de choix stylistique. Par contre, donner un style naturel aux API en Nit est plus difficile à automatiser et une touche humaine demeure nécessaire.

De plus, la reproduction des API natives est une tâche de taille, l'API Java d'Android à la version 10 est composée de 1 285 classes Java et 14 365 méthodes, fonctions, constructeurs et champs. [2] Dans nos expérimentations, nous avons reproduit un sous-ensemble de

2. Nous considérons les services de l'API 10 en nous limitant au paquet `android` seulement.

ces classes en Nit. D'abord, nous avons confié la tâche de programmation à des stagiaires qui ont travaillé à reproduire manuellement certaines API. Mais cette approche n'était pas réaliste, puisque couvrir chaque classe nécessitait trop de temps, donc nous sommes passés à des générateurs de code. Nous avons conçu deux générateurs de code, d'abord *jwrapper* pour reproduire les API Java d'Android, et puis *objcwrapper* pour Objective-C et iOS.

Avec du recul, nous constatons que l'utilisation de générateurs de code, au moins en tant que support, est la seule façon réaliste pour reproduire les API natives de taille réelle. Alors que le code est relativement facile à générer automatiquement, le défi est l'analyse des API (depuis le code octet Java ou les entêtes Objective-C) et la transformation du modèle de l'API native vers un modèle Nit.

Les deux sections suivantes présentent plus en détail chaque générateur de code. Nous en profitons pour donner une idée sur la forme que prennent les API reproduites.

5.4.1 *jwrapper* – Générateur pour reproduire les API Java

Le générateur *jwrapper* reproduit les API Java en Nit.[3] Il est entièrement fonctionnel, nous l'avons utilisé avec succès sur *android.jar*, l'archive entière de l'API native d'Android ainsi que sur *cardboard.jar*, l'API de réalité virtuelle de Google Cardboard.

Le générateur *jwrapper* prend en entrée des archives JAR, dont il extrait le code octet qu'il analyse à l'aide du désassembleur *javap*. La sortie de *javap* prend la forme d'un sous-ensemble du langage Java avec seulement la structure des classes et la signature des méthodes, mais sans les détails de leur implémentation. Le générateur *jwrapper* en extrait le modèle des classes. Finalement, *jwrapper* génère le code Nit, incluant l'utilisation de la FFI, pour reproduire tous les services en Nit.

Le générateur *jwrapper* reproduit les API Java en Nit au niveau des classes, méthodes, attributs, constructeurs et fonctions statiques :

---

3. Le code source de *jwrapper* est disponible dans le dépôt Nit et sur le web à `https://github.com/xymus/nit/tree/these_alexis/contrib/jwrapper/`

— Pour chaque classe Java (incluant les classes internes), par exemple `android.widget.Button`, *jwrapper* génère une classe externe Nit correspondante. La classe Nit est nommée de façon prévisible selon le nom du paquet et de la classe Java, tel que `Android_widget_Button`. De plus, il génère une autre classe externe pour représenter le tableau Java primitif de cette classe, tel que `Array_Android_widget_Button` pour `android.widget.Button[]`.

— Pour chaque méthode Java, *jwrapper* génère une méthode externe Nit portant un nom modifié de façon à respecter la grammaire et le style du langage Nit. En Nit, le nom des méthodes, et des autres propriétés de classes, commencent par une lettre minuscule et le style suggère d'utiliser la convention *snake_case*. Donc une méthode Java `setOnClickListener` est reproduite par `set_on_click_listener` en Nit.

  De plus, si la méthode Java est surchargée statiquement, une fonctionnalité n'existant pas en Nit, le type des paramètres est utilisé en suffixe pour distinguer les différentes implémentations. Le résultat est que toutes les différentes implémentations sont accessibles depuis Nit, malgré l'incompatibilité entre les langages. Par contre, le nom des méthodes est moins prévisible par le programmeur, il doit consulter la documentation ou le code généré pour les utiliser.

— Pour chaque attribut Java, *jwrapper* génère deux méthodes Nit, un accesseur et un mutateur. Par exemple, un champ Java `java.lang.String text` est reproduit par les méthodes externes `fun text: JavaString` et `fun text=(value: JavaString)`.

— Pour chaque constructeur Java, *jwrapper* ajoute un constructeur externe à la classe externe Nit. Si la classe Java a un seul constructeur, le constructeur externe est anonyme et il reproduit la signature du constructeur Java. S'il y a plusieurs constructeurs, ils sont reproduits avec des noms dérivés du type des paramètres. Encore une fois, les incompatibilités entre les langages rendent les noms générés moins prévisibles. Au moins, les constructeurs nommés, une fonctionnalité de Nit n'existant pas en Java, permettent de contourner la différence.

— Pour chaque fonction statique Java, *jwrapper* génère l'équivalent en Nit, une méthode sur la classe `Objet`. Le nom de la méthode est composé du paquet Java suivi du nom de la fonction modifiée pour la grammaire et le style Nit.
Par exemple, la fonction statique `makeText` de `android.widget.Toast` est reproduite par `android_widget_Toast_make_text`.

Le générateur *jwrapper* utilise un dictionnaire de correspondance des types entre Java et Nit. Le dictionnaire permet de typer les signatures des méthodes reproduites en Nit et de bénéficier de la correspondance des types offerte par la FFI. Le générateur analyse la bibliothèque Nit pour réutiliser les classes externes existantes. De plus, il sauvegarde le dictionnaire de correspondance pour une réutilisation entre deux générations.

Il y a une certaine incompatibilité entre les modules Nit et les classes Java. En Java, il peut y avoir des dépendances cycliques entre les classes, alors qu'en Nit, les dépendances entre les modules sont d'un ordre partiel, donc sans cycle. Le résultat est que chaque classe Java ne peut pas être reproduite dans son propre module Nit. De même que deux paquets Java couplés ne peuvent pas être reproduits dans des modules distincts. Pour éviter les dépendances cycliques, l'utilisateur de *jwrapper* doit s'assurer de générer toutes les classes couplées dans un même module Nit. Par exemple, dans nos expérimentations, nous avons reproduit la bibliothèque standard de Java dans un module, et l'API native d'Android dans plusieurs modules regroupant des services spécialisés.

En pratique, *jwrapper* est capable d'analyser la bibliothèque standard Java et les API natives d'Android. Pour notre expérimentation, nous avons visé un sous-ensemble de la bibliothèque standard Java, soit les paquets `java`, `javax`, `junit` et `org`, ainsi que `android` de l'API native d'Android (version 10). Nous avons lancé *jwrapper* sur les 2 760 classes composant ces paquets par deux appels successifs présentés dans la figure 5.12. Sur le total de 24 382 méthodes, fonctions, constructeurs et champs, seulement 54 propriétés ne sont pas reproduites en Nit par *jwrapper* avec la configuration par défaut. Ces 54 propriétés dépendent de tableaux à deux dimensions qui peuvent être générés automatiquement en activant une option de *jwrapper*.

```
1 # Reproduit l'API de la bibliothèque standard de Java (et plus)
2 jwrapper android.jar -o java_api.nit -r "^(java|javax|junit|org)" \
3   -u comment --save-model
4
5 # Reproduit l'API de Android, réutilisant la passe précédente
6 jwrapper android.jar -o android_api.nit -r "^(android|com.android)" \
7   -u comment -i java_api.nit -m java_api.jwrapper.bin
```

Figure 5.12: Appels à *jwrapper* pour reproduire l'API de Android en Nit.

5.4.1.1 Limitations

Le générateur *jwrapper* ne supporte pas les méthodes avec un nombre variable d'arguments (*vararg*). Alors que Nit supporte aussi les *vararg*, il faudrait leur appliquer un traitement supplémentaire pour les passer aux méthodes Java. Chaque *vararg* est passé sous la forme d'un tableau Nit qui pourrait être converti en tableau Java. Ce traitement pourrait être généré par *jwrapper*, ou alors directement par la FFI avec Java.

De plus, *jwrapper* ne considère pas les exceptions lancées par les méthodes Java, le programmeur doit donc modifier le code généré pour traiter les exceptions. Le langage Nit n'offrant pas un équivalent aux exceptions Java, il n'y a pas de solution unique à ce problème. Le programmeur doit choisir une stratégie pour préserver les informations des exceptions et les passer au code Nit.

5.4.2 *objcwrapper* – Générateur pour reproduire les API Objective-C

Le générateur *objcwrapper* reproduit les API Objective-C en Nit.[4] Il est semblable à *jwrapper*, mais *objcwrapper* est encore à l'état de preuve de concept et il ne supporte qu'un sous-ensemble des classes Objective-C de l'API native d'iOS.

L'analyse du code Objective-C est plus difficile que l'analyse des classes Java via *javap*. Objective-C est une extension au langage C, il en hérite beaucoup de complexité, en plus

4. Le code source de *objcwrapper* est disponible dans le dépôt Nit et sur le web à `https://github.com/xymus/nit/tree/these_alexis/contrib/objcwrapper/`

de ses structures grammaticales propres. Pour simplifier le travail, *objcwrapper* ne lit que les fichiers d'entête (`.h`), et ce, après l'application du préprocesseur C pour étendre les macros.

Plusieurs structures du langage Objective-C ne sont pas encore supportées. Au moment de l'écriture de ce document, *objcwrapper* est capable d'analyser 32 des 168 classes qui composent l'API d'interface utilisateur UIKit d'iOS 9.2. L'outil pourrait être complété par un travail futur.

5.4.3 Travaux connexes

Le milieu scientifique a aussi étudié la génération de code, souvent dans le but d'éviter au programmeur la complexité de la JNI.

Nous avons déjà mentionné Djinni, à la section 1.4.7, un générateur d'API C++, Java et Objective-C. Il utilise un langage de description d'interface (IDL) pour déclarer explicitement l'interface à générer. De façon similaire, GIWS (GIWS, 2012) génère des classes C++ agissant de *proxy* à des classes Java. Il utilise un IDL en XML pour décrire les classes Java.

Comparativement à l'analyse de la sortie de *javap*, l'utilisation d'un IDL demande plus de travail manuel au programmeur, mais l'analyse de l'IDL est plus simple que l'analyse de *javap*. De plus, utiliser un IDL permet de personnaliser la génération de code.

D'une certaine façon, on peut considérer que le code généré par *jwrapper* agit comme IDL expressif. D'abord, *jwrapper* analyse une API et génère le code pour la FFI de Nit, incluant le Java. Le programmeur peut modifier le code généré pour y ajouter des traitements particuliers, par exemple pour convertir des chaînes de caractères ou modifier l'API en Nit. Ensuite, le compilateur et la FFI de Nit appliquent un traitement semblable à Djinni et à GIWS en générant le code utilisant la JNI, et en le compilant en code binaire.

5.5 API reproduites et *app.nit*

En développement d'applications portables, certaines solutions reproduisent les API natives à Android et à iOS dans un autre langage, tel que React Native qui les reproduit en JavaScript. La motivation de cette approche est qu'en reproduisant les deux API, le programmeur peut réaliser une application en JavaScript seulement, tout en accédant aux fonctionnalités propres à chaque plateforme.

En fait, reproduire les API natives cache la programmation polyglotte et la FFI au programmeur. Il s'agit d'un avantage lorsque la FFI est complexe d'utilisation et propice aux erreurs. Mais cacher la programmation polyglotte n'est pas favorable avec la solution *app.nit*, car la FFI de Nit est expressive et permet au programmeur d'atteindre ses objectifs facilement. Une telle FFI apporte plus d'avantages que d'inconvénients.

De plus, les API reproduites ne répondent pas à tous les besoins. Notamment, il est impossible de reproduire toutes les API à temps pour leur utilisation. Il y a de nouvelles versions des API d'Android et iOS qui sont publiées au moins une fois par année et, en plus, il y a des bibliothèques de tierces parties qui offrent des fonctionnalités supplémentaires.

Tout de même, nous avons utilisé des API reproduites à différents endroits dans ce projet de recherche, et nous en recommandons un usage limité aux clients de *app.nit*. Nous différencions la reproduction des classes de la reproduction de leurs services. Reproduire les classes étrangères par des classes externes permet de typer les objets passés au code étranger et d'en conserver une référence en Nit. Toutefois, reproduire chaque service par une méthode externe Nit est généralement plus restrictif que de regrouper plusieurs appels aux mêmes services dans une seule méthode externe en appliquant le patron *façade polyglotte*. Nous recommandons donc de reproduire les classes étrangères en classes externes Nit dès leur première utilisation, mais de n'en reproduire les services que s'ils sont utilisés plus d'une fois.

5.6 Conclusion

Dans ce chapitre, nous avons proposé trois patrons de conception visant des problèmes de programmation polyglotte. Nous avons observé ces problèmes et appliqué les patrons lors de l'implémentation des API *app.nit* pour Android et iOS, ainsi que lors de la conception des applications de validation. Chacun des trois patrons répond à des problèmes fondamentalement polyglottes et à des besoins différents :

— Le patron *façade polyglotte* structure le passage de données du code Nit au code étranger en divisant le travail en deux méthodes. Le résultat est une API publique n'exposant que des types naturels à Nit et cachant la complexité de la FFI à ses clients.
— Le patron *adaptateur polyglotte* reproduit des API à objets en Nit, encapsulant les classes externes de façon à en automatiser le cycle de vie. Encore une fois, le résultat est une API publique naturelle en Nit.
— Le patron *spécialisation polyglotte* permet de spécialiser une interface étrangère en Nit pour personnaliser la réponse aux événements d'une API étrangère.

Pour assister l'application du patron *adaptateur polyglotte*, nous avons réalisé deux générateurs de code. Le générateur *jwrapper* analyse les classes Java dans une archive JAR et génère le code Nit déclarant des classes externes associées aux classes Java. Pour Objective-C, *objcwrapper* analyse les fichiers d'entête après avoir appliqué le préprocesseur pour ensuite générer le code des classes externes. Alors que *objcwrapper* n'est qu'une preuve de concept, *jwrapper* est entièrement fonctionnel et nous l'avons utilisé dans l'implémentation des API *app.nit* pour Android.

Des travaux futurs pourraient étendre ces générateurs pour générer le code pour implémenter les autres patrons. Les classes ordinaires de *adaptateur polyglotte* pourraient être partiellement générées automatiquement, toutefois une touche humaine est nécessaire pour les adapter au style Nit. De plus, le traitement des données du patron *façade polyglotte* pourrait être divisé automatiquement entre la méthode d'indirection et la méthode externe.

Dans ce chapitre, et le suivant, nous appliquons les patrons de conception polyglotte dans le domaine des applications mobiles pour accéder à des API natives. Ces patrons établissent les bonnes pratiques à l'utilisation de la FFI de Nit pour tout programme polyglotte. Ils peuvent aussi guider la conception d'un programme utilisant les méthodes externes pour optimiser un goulot d'étranglement des performances. Ou alors, pour reproduire un ensemble de classes et de services étrangers créés par le programmeur en même temps que le code Nit.

CHAPITRE VI

ÉTUDE DE CAS – LIGNE DE PRODUITS TENENIT

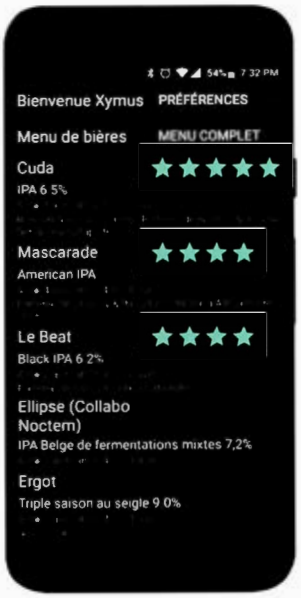
Afin de valider l'application de notre solution dans un contexte réel, nous avons réalisé une application mobile pour les clients de la brasserie artisanale montréalaise Benelux. L'application affiche le menu de bières et sert d'interface à un petit réseau social. Elle est indépendante du commerce et peut être configurée pour n'importe quel commerce offrant un menu de bières.

Nous avons conçu et développé l'application mobile sous forme d'une ligne de produits nommée Tenenit, en l'honneur de Tjenenet, une déesse égyptienne de la bière. Nous avons aussi développé le logiciel serveur, il peut paraître moins pertinent relativement au sujet de cette thèse, mais il apporte un besoin supplémentaire en portabilité du code, donc nous en discuterons lorsque ce sera approprié. [1]

Pour illustrer le projet, les figures 6.1 et 6.2 présentent deux produits Tenenit, une application pour Android et une pour iOS.

Ce projet a été retenu parce que l'application est représentative des besoins des applications mobiles typiques. L'application est fondamentalement mobile, elle peut être utilisée dans le commerce ou dans tout autre lieu pour consulter rapidement le menu. Fonctionnant sur un appareil mobile, son interface utilisateur est tactile et elle doit être économe en énergie. Comme on désire rejoindre toute la clientèle, l'application doit aussi

1. Le code source de la ligne de produits Tenenit et du serveur est publié à **https://github.com/xymus/nit/tree/these_alexis/contrib/tenenit/**.



(a) Haut de la fenêtre d'accueil

(b) Bas de la fenêtre d'accueil

Figure 6.1: Produit adapté à Android.

(a) Haut de la fenêtre d'accueil

(b) Bas de la fenêtre d'accueil

Figure 6.2: Produit adapté à iOS.

être portable à Android et à iOS. Finalement, elle est connectée à un serveur distant pour obtenir le menu et pour joindre le réseau social.

L'application mobile profite des trois aspects de notre solution en plus des patrons de conception polyglotte.

— Un prototype regroupant les fonctionnalités principales a été réalisé rapidement à l'aide des quatre API portables de la bibliothèque *app.nit*. Les quatre API agissent dans ce projet pour construire l'interface utilisateur, gérer le cycle de vie de l'application, exécuter les requêtes HTTP et préserver les préférences de l'utilisateur entre deux exécutions.
— Le projet est organisé en une ligne de produits réalisée par raffinement de classes pour obtenir une famille d'applications semblables avec des variations pour Android, pour iOS et pour différents commerces. Cette organisation permet de partager une base de code commune et de sélectionner des fonctionnalités optionnelles pour former chaque produit.
— L'application utilise des fonctionnalités natives à Android et à iOS par programmation polyglotte via la FFI de Nit. Les méthodes externes donnent un accès direct aux API natives dans leur entièreté et dans leur langage natif, ce qui permet d'appliquer l'expérience préalable sur chaque plateforme.
— Finalement, la conception de l'application a été guidée par les patrons de conception polyglotte, qui apparaissent localement dans l'application ou dans la bibliothèque *app.nit*.

Dans ce chapitre, nous présentons certaines fonctionnalités représentatives de la ligne de produits Tenenit. Ensuite, nous discutons du moment d'application des variations et du rôle de la solution *app.nit* dans le projet en la comparant avec deux solutions alternatives. Finalement, nous concluons sur une évaluation du rôle de notre solution *app.nit* dans le projet Tenenit.

Nous classifions les fonctionnalités selon la taxonomie de variabilité dans les lignes de produits telle que définie en quatre catégories par Svahnberg et al. (Svahnberg, Van Gurp et Bosch, 2005) :

— les fonctionnalités obligatoires présentes dans le prototype et tous les produits,

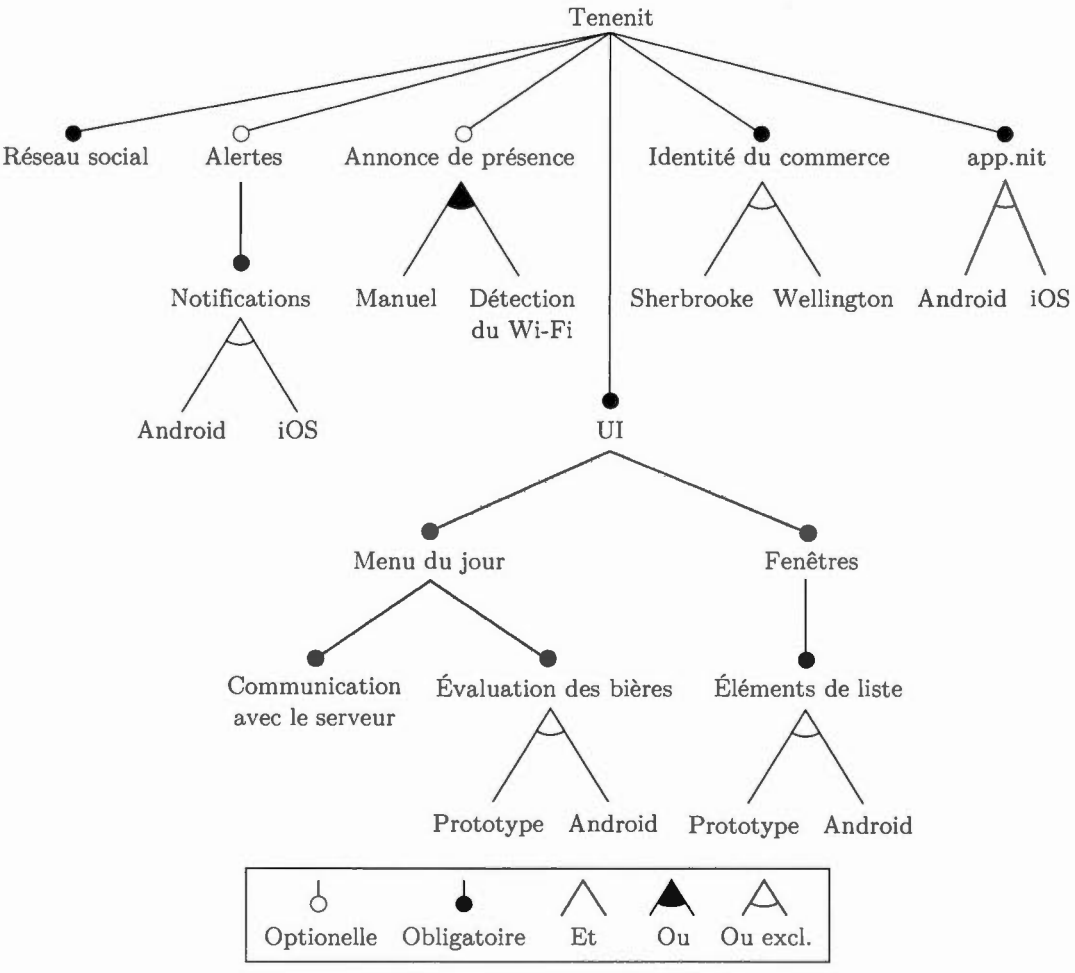


Figure 6.3: Diagramme de fonctionnalités de la ligne de produits Tenenit.

— les fonctionnalités optionnelles non exclusives (OU),

— les fonctionnalités optionnelles exclusives (XOU),

— les fonctionnalités externes dont l'implémentation dépend de la plateforme.

Nous illustrons également les fonctionnalités sous deux formes. La figure 6.3 les présente sous forme de diagramme de fonctionnalités avec leurs relations, alors que la figure 6.4 présente les relations d'importation entre les modules qui composent le projet — notamment, on y observe la séparation des variations entre les modules Nit.

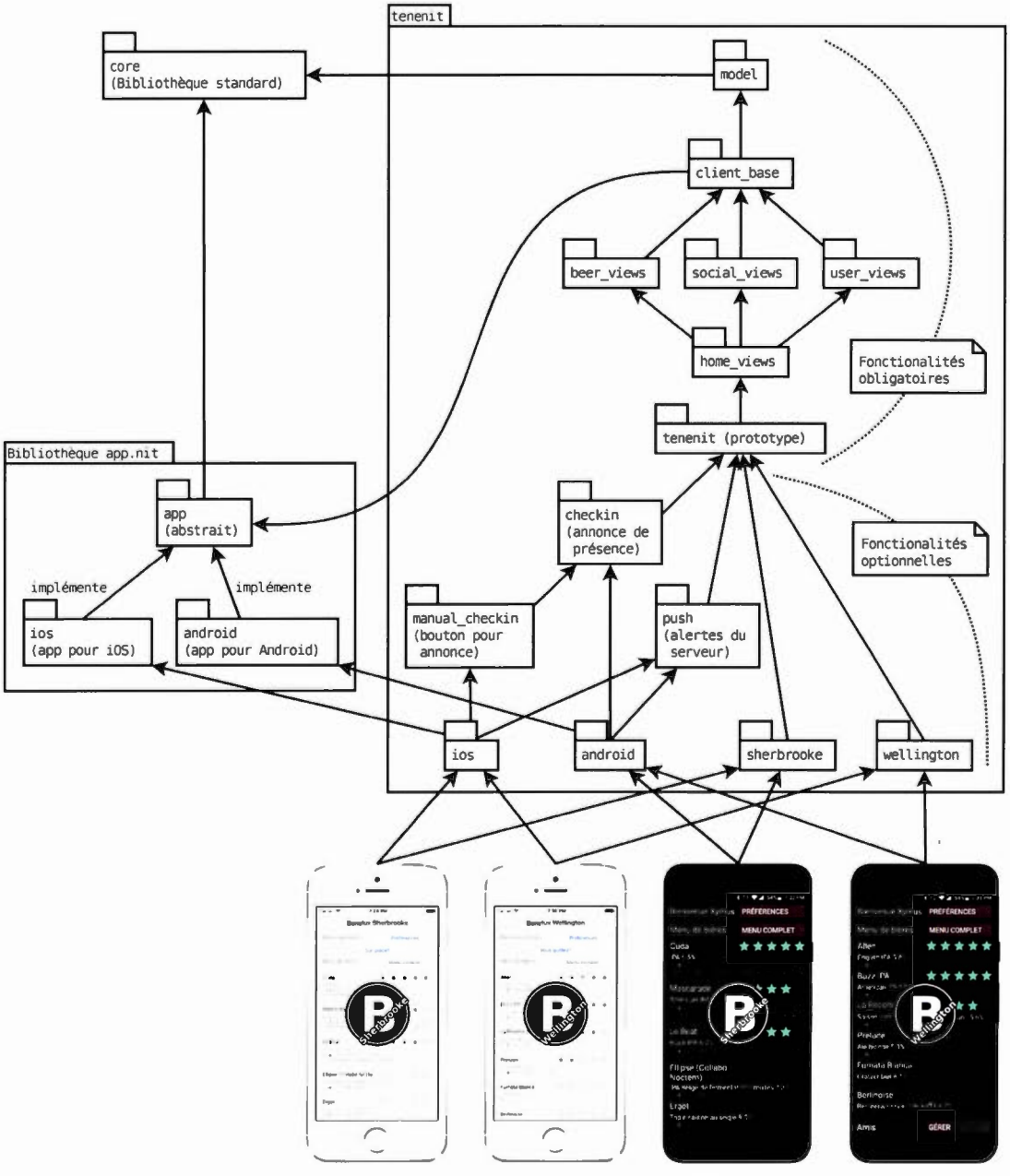


Figure 6.4: Dépendances entre les modules et produits du projet Tenenit.

6.1 Fonctionnalités obligatoires et prototype

Les fonctionnalités obligatoires définissent le projet Tenenit et sont vitales au bon fonctionnement des produits. Elles sont déclarées dans le prototype et sont présentes dans tous les produits.

En fait, toutes les fonctionnalités du prototype sont obligatoires, même si certaines peuvent varier. Cette section présente des fonctionnalités obligatoires centrales : le menu du jour, le réseau social ainsi que les fenêtres de support.

6.1.1 Menu du jour

La premier besoin auquel répond l'application mobile est l'affichage du menu journalier de bières. La brasserie Benelux offre un menu variable, il y a une douzaine de bières sur le menu chaque jour sur un total d'environ 60 bières en rotation annuellement. Le menu est donc la première chose recherchée par les utilisateurs, que ce soit sur place ou avant de se déplacer.

Le menu est réalisé avec les contrôles de l'API UI de *app.nit*, il prend donc la forme de contrôles natifs autant sous Android que sous iOS. Chaque bière est présentée par un contrôle personnalisé, `BeerView`, qui en affiche le nom, une courte description, l'évaluation de la communauté et des recommandations personnalisées. Ce contrôle est interactif, il permet à l'utilisateur d'entrer une évaluation pour chaque bière.

Le menu et le contrôle `BeerView` sont visibles dans les figures 6.1.a et 6.2.a, page 188. Le menu est affiché à deux endroits : un résumé est présenté dans la fenêtre d'accueil et le menu complet est affiché dans une fenêtre dédiée au menu de bières.

Le menu est mis à jour en réponse à l'événement `on_resume` de l'API cycle de vie. À ce moment, l'application communique avec le serveur à l'aide de l'API de requêtes HTTP.

La majorité du code pour afficher et obtenir le menu est portable, car il utilise seulement les API de *app.nit* et d'autres services portables de la bibliothèque Nit. Le menu est donc entièrement fonctionnel dans le prototype. Toutefois, le contrôle `BeerView` sert de point de variation, il est utilisé par les adaptations aux plateformes et il sera revisité à la section 6.3.2.

6.1.2 Réseau social

Pour agrémenter le menu de recommandations personnalisées, nous avons ajouté un petit réseau social au projet Tenenit. Le réseau social permet à l'utilisateur de se créer un identifiant et de suivre d'autres utilisateurs. Le fait de suivre un utilisateur est l'équivalent de s'abonner à ses évaluations. Le réseau social donne le caractère aux applications Tenenit, pour cette raison, nous le considérons comme une fonctionnalité obligatoire.

L'application mobile n'est que l'interface utilisateur au réseau social, le serveur gère la persistance des comptes utilisateurs, des évaluations et autres. Le client et le serveur partagent plusieurs fichiers de code, un bénéfice de la portabilité de Nit, non seulement entre Android et iOS, mais aussi avec GNU/Linux. Encore une fois, les échanges avec le serveur sont exécutés à l'aide de l'API requêtes HTTP.

Dans l'application mobile, le réseau social est d'abord accessible par une liste d'utilisateurs au bas de la fenêtre d'accueil, visible dans les figures 6.1.b et 6.2.b, page 188. La liste présente des utilisateurs déjà suivis et des utilisateurs recommandés. Le nom de chaque utilisateur est accompagné de ses bières favorites et d'un bouton pour le suivre. De plus, une fenêtre dédiée à la recherche d'utilisateurs permet aussi de consulter les utilisateurs déjà suivis.

6.1.3 Fenêtres de support

En plus des fenêtres d'accueil, du menu de bières et de recherche d'utilisateurs, deux fenêtres permettent à l'utilisateur de se connecter au réseau social et de personnaliser le comportement de l'application.

La fenêtre d'authentification permet à l'utilisateur de se connecter auprès du réseau social. Elle est entièrement portable, car elle utilise les API UI et requêtes HTTP. À la création d'un compte, elle vérifie localement la validité du nom d'utilisateur et du mot de passe. Par la suite, elle envoie une requête au serveur et affiche la réponse à l'utilisateur.

La fenêtre de préférences rassemble les options de l'application qui permettent à l'utilisateur d'en personnaliser le comportement. En plus de l'API UI, elle utilise l'API persistance des données pour préserver les préférences entre deux exécutions. La version obligatoire offre une seule option, se déconnecter du réseau social. Les autres options sont introduites par des fonctionnalités optionnelles. La fenêtre est donc étendue par d'autres modules et variations de la ligne de produits.

6.2 Fonctionnalités optionnelles non exclusives

Les fonctionnalités optionnelles sont partagées par certains produits, sans être obligatoirement présentes dans tous les produits. Les fonctionnalités optionnelles non exclusives sont une sous-catégorie, celles-ci sont facultatives et leurs différentes variations n'entrent pas en conflit entre elles. La ligne de produits Tenenit définit deux principales fonctionnalités optionnelles non exclusives, les alertes du serveur et la fonctionnalité d'annonce de présence.

6.2.1 Alertes du serveur

Pour tenir l'utilisateur informé des changements au menu, l'application peut recevoir des alertes du serveur. Le serveur envoie alors un message, tel que la liste des nouvelles bières, pour qu'il soit affiché à l'utilisateur de l'application mobile.

Cette fonctionnalité est réalisée avec l'API de requêtes HTTP et par la technique *push notification* qui conserve une connexion HTTP ouverte avec le serveur, permettant au serveur de répondre à tout moment. Cette utilisation valide un cas limite de l'API de requêtes HTTP où les requêtes restent ouvertes longtemps et peuvent être interrompues à tout moment par un problème de connexion. Il s'agit de la même technique utilisée par le client mobile à Tnitter, discutée à la section 2.3.4.1.

La fonctionnalité d'alertes est majoritairement portable entre Android et iOS, à l'exception de l'affichage à l'utilisateur du contenu de l'alerte, le message comme tel. L'affichage

est une fonctionnalité externe adaptée à chaque plateforme, elle sera discutée à la section 6.4.

De plus, la fonctionnalité d'alertes ajoute des options à la fenêtre des préférences de l'utilisateur. Elle profite donc de l'organisation en ligne de produits et du point de variation pour modifier une classe existante. Les options ajoutées prennent la forme de boîtes à cocher de l'API UI, elles permettent à l'utilisateur de choisir quelles alertes recevoir.

6.2.2 Annonce de présence

La deuxième fonctionnalité optionnelle est partiellement complémentaire aux alertes du serveur. Elle permet à un utilisateur d'annoncer lorsqu'il est présent sur place. Cette information est utilisée à deux fins : informer l'utilisateur des nouvelles bières apparues au menu depuis sa dernière visite, ainsi qu'inviter ses amis à le rejoindre sur place, et ce, via les alertes du serveur.

La détection de la présence sur place est une fonctionnalité variante réalisée différemment selon la plateforme. Il y a deux variations pour réaliser la même fonctionnalité, elles sont implémentées par programmation polyglotte.

Sous Android, la présence est détectée automatiquement selon les réseaux Wi-Fi dans la portée de l'appareil. Il s'agit d'une fonctionnalité disponible uniquement en Android, car iOS ne donne accès qu'au réseau auquel l'appareil est actuellement connecté. L'implémentation accède aux API natives d'Android via la FFI de Nit, et applique le patron *spécialisation polyglotte* pour répondre à la détection de nouveaux réseaux.

Sous iOS, la présence sur place doit être déclarée manuellement par l'utilisateur et ce, par la pression d'un bouton qui porte l'étiquette «Sur place ?». Ce bouton est visible dans la capture d'écran du produit adapté à iOS, présentée dans la figure 6.5 qui compare le prototype avec le produit pour iOS. Cette variation est réalisée simplement par des contrôles de l'API UI, elle est donc aussi portable à Android. Comme de fait, ces deux

(a) Prototype pour iOS. (b) Produit adapté à iOS.

Figure 6.5: Prototype et produit d'un même commerce pour iOS.

variations ne sont pas mutuellement exclusives, il est possible d'activer la version manuelle sous Android en plus de la détection automatique, ou même de n'activer aucune des deux variations.

Encore une fois, cette fonctionnalité ajoute des options à la fenêtre des préférences de l'utilisateur. L'utilisateur a le choix de partager ou non sa présence avec le serveur et avec à ses amis.

6.3 Fonctionnalités optionnelles exclusives

Les fonctionnalités optionnelles exclusives sont des fonctionnalités dont une seule variation peut être appliquée à la fois. Il peut s'agir de fonctionnalités obligatoires, mais dont l'implémentation varie selon les produits.

Dans le projet Tenenit, la majorité des fonctionnalités optionnelles exclusives modifient des fonctionnalités obligatoires en utilisant des services propres à une plateforme. Une seule fonctionnalité dépend du commerce, nous commençons par présenter celle-ci.

6.3.1 Identité du commerce

Le projet Tenenit a été conçu pour desservir n'importe quelle brasserie, ou tout autre commerce offrant une sélection de bières. Pour ce faire, l'identité du commerce, le logo, le nom de l'application et les informations du serveur distant sont des points de variation.

Avec l'organisation en ligne de produits, un seul module Nit peut appliquer cette variation. Les métadonnées *app.nit* servent à configurer le nom de l'application et son *namespace*. Le logo peut servir d'icône s'il est placé parmi les ressources recherchées par le compilateur. De plus, le raffinement de classes permet de modifier des points de variation qui définissent le nom du commerce à afficher dans l'application et l'adresse du serveur distant.

Au moment de l'écriture, il y a deux variations, une pour chaque succursale de la brasserie artisanale Benelux : Sherbrooke et Wellington. La figure 6.6 présente la source de la variation de Tenenit pour la succursale Sherbrooke.

6.3.2 Évaluation des bières

Le menu de bières étant central au projet, nous avons décidé d'en améliorer l'apparence selon la plateforme. Donc, même si le menu de bières est obligatoire, nous l'utilisons comme point de variation.

Dans le prototype, le contrôle pour évaluer une bière est réalisé par cinq boutons distincts avec des étoiles Unicode comme étiquette. Sans être très élégants, les cinq boutons agissent comme version par défaut du contrôle et ils sont portables à Android et à iOS.

L'adaptation pour Android utilise le raffinement de classes pour modifier le point de variation `BeerView`. Elle remplace le contrôle du prototype par un contrôle de l'API

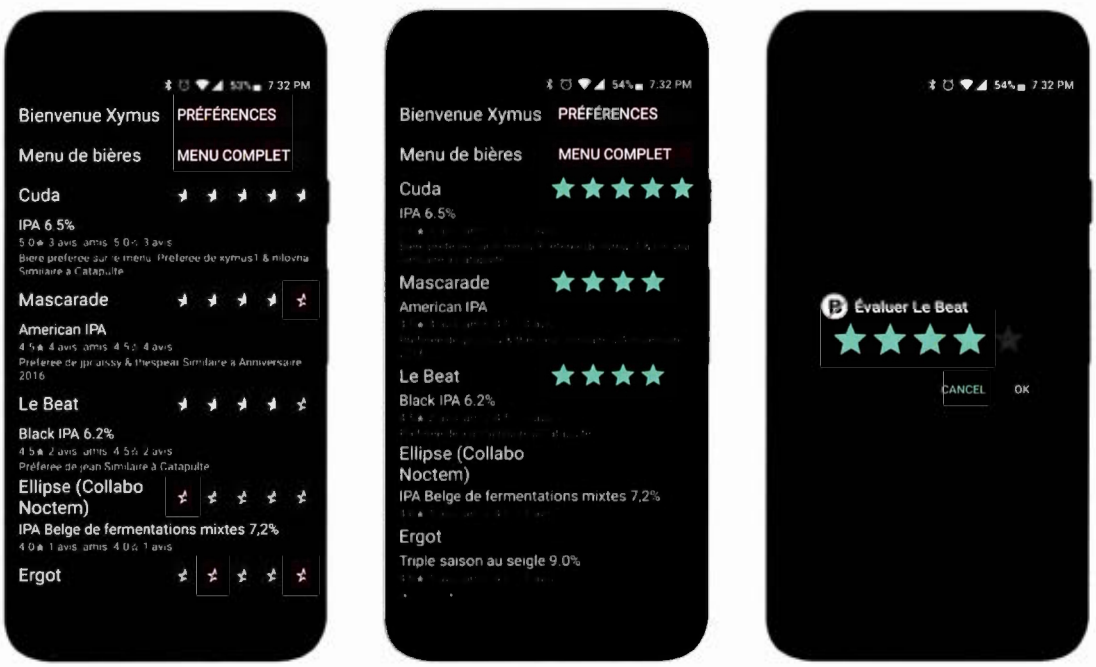
```
1 module benelux_sherbrooke is
2     app_name "Benelux Sherbrooke"
3     app_namespace "net.xymus.beneluxsherbrooke"
4     app_files # L'icône est local au dossier
5 end
6
7 import tenenit::client
8
9 # Nom utilisé comme titre de fenêtre sous iOS
10 redef fun in_app_name do return "Benelux Sherbrooke"
11
12 # Adresse du serveur REST
13 redef fun tenenit_rest_server_uri
14 do return "http://xymus.net/benelux-sherbrooke/"
15
16 # Adresse du site web affichant les événements
17 redef fun event_website
18 do return "http://brasseriebenelux.com/sherbrooke"
```

Figure 6.6: Code source d'une variation sur l'identité du commerce.

native à Android, un `android.widget.RatingView`, auquel elle accède via une méthode étrangère en appliquant le patron *façade polyglotte*. Ce contrôle natif affiche cinq étoiles qui respectent le thème de couleur de l'appareil. La différence apportée par cette variation est démontrée dans la figure 6.7 qui compare le prototype avec le produit adapté à Android.

Pour ajouter une fonctionnalité propre à Android, nous avons étendu la méthode étrangère. Lorsque l'utilisateur tape sur le contrôle d'évaluation, une fenêtre s'ouvre et l'invite à confirmer l'évaluation, cette fenêtre est présentée dans la figure 6.7.c. L'implémentation est réalisée par un gros bloc de code Java qui utilise plusieurs fonctionnalités de l'API native. De plus, l'application répond au choix de l'utilisateur, et envoie l'évaluation au serveur, en appliquant le patron *spécialisation polyglotte*.

Pour iOS, le contrôle `BeerView` est raffiné uniquement pour en améliorer l'apparence. La taille et la couleur des étiquettes sont modifiées selon le style de la plateforme, et une marge autour du contrôle allège l'interface. Ces modifications utilisent les fonctionnalités de l'API UI pour les modifications simples, ainsi que du code Objective-C via la FFI pour ajuster la marge. Le résultat est visible dans la figure 6.5, page 196.



(a) Prototype pour Android. (b) Produit adapté à Android. (c) Fenêtre d'évaluation.

Figure 6.7: Prototype et produit d'un même commerce pour Android.

```
1 redef class ItemView
2   # Constructeur agissant d'indirection et de point de variation
3   init do set_background(native, app.native_context)
4
5   # Méthode externe
6   private fun set_background(view: NativeView, context: NativeContext)
7    in "Java" `{
8     int color = context.getResources().getIdentifier(
9       "item_background", "color", context.getPackageName());
10    view.setBackgroundResource(color);
11  `}
12 end
```

Figure 6.8: Code source adaptant les éléments de liste à Android.

6.3.3 Éléments de liste

Presque toutes les fenêtres de l'application utilisent des listes pour organiser l'information présentée à l'utilisateur, dont les détails sur les bières et sur les utilisateurs. Pour mieux présenter les sections et différencier les titres des autres éléments de listes, nous en avons modifié l'apparence différemment pour Android et pour iOS. En fait, ce sont les éléments de listes qui agissent comme points de variation modifiables par raffinement de classes.

Pour Android, la seule modification est la couleur de l'arrière-plan. L'implémentation de cette variation, dont le code est présenté à la figure 6.8, applique le patron *façade polyglotte* et utilise les ressources Android. Nous avons appliqué une pratique recommandée pour Android, qui consiste à définir les couleurs des contrôles dans un fichier XML. Pour y arriver, un fichier XML est ajouté dans la base de projet Android à `android/res/values/styles.xml`. Ensuite, une méthode externe en Java récupère la couleur par une requête à l'API des ressources d'Android et l'assigne comme arrière-plan au contrôle.

Pour iOS, le raffinement permet d'assigner un arrière-plan gris aux titres de section et un blanc aux éléments de liste. Le changement est fait par du code polyglotte et la couleur grise, standard au système, est obtenue par un appel à l'API native.

6.4 Fonctionnalités externes

Les fonctionnalités externes sont définies par une API abstraite et elles sont réalisées par des services externes à l'application. Elles se différencient des fonctionnalités optionnelles par l'API abstraite indépendante de l'implémentation.

La majeure partie des fonctionnalités externes de Tenenit proviennent de la bibliothèque *app.nit*. Quelques autres sont définies localement et accompagnées d'une implémentation distincte pour Android et pour iOS.

6.4.1 Bibliothèque *app.nit*

La bibliothèque *app.nit* définit les quatre API portables qui sont utilisées pour réaliser le prototype et certaines autres variations. Presque toutes les fonctionnalités de la bibliothèque *app.nit* sont externes. Les quatre API sont abstraites et elles sont implémentées par raffinement de classes selon chaque plateforme.

L'API UI en est un bon exemple, les classes `Button` et `CheckBox` exposent la même API sur toutes les plateformes, mais leur implémentation est complètement différente selon la plateforme. La classe `Button` est réalisée par un `android.widget.Button` sous Android et par un `UIButton` de UIKit sous iOS. On accède à ces services via des méthodes externes et conservées en Nit à l'aide de classes externes. La classe `CheckBox` sous Android est réalisée par un simple `android.widget.CheckBox` alors que sous iOS, il est réalisé par la combinaison d'un `UIToggle` et d'un `UILabel`. Ces deux contrôles utilisent le système d'événement spécifique à leur plateforme pour répondre aux interactions de l'utilisateur en appliquant le patron *spécialisation polyglotte.*

Plusieurs parties des API *app.nit* suivent le patron *adaptateur polyglotte.* L'API UI utilise la variation avec abstraction, donc les classes externes reproduisent fidèlement l'API native à Android et à iOS, mais les classes ordinaires de la couche *nity* servent d'abstraction aux deux plateformes. Elles s'éloignent donc beaucoup des API natives originales pour offrir une API commune et *nity.* De plus, l'API UI gère automatiquement le cycle de vie des objets étrangers.

6.4.2 Navigateur

L'application Tenenit présente un événement régulier de la brasserie au bas de la fenêtre d'accueil. Une brève description de l'événement est accompagnée d'un bouton pour ouvrir le site web du commerce et obtenir plus d'informations. Dans le code, l'ouverture du site web est implémentée par un appel à `open_in_browser`, une autre fonctionnalité externe de *app.nit*.

La méthode `open_in_browser` de l'API UI est abstraite et implémentée différemment selon la plateforme. De plus, sous Android, le système offre à l'utilisateur un choix de navigateur. Cette fonctionnalité est donc indépendante de la plateforme et du navigateur.

Il s'agit d'un exemple simple mais complet d'un service abstrait implémenté par raffinement et programmation polyglotte. En plus, il applique le patron *façade polyglotte*. La déclaration abstraite de `open_in_browser` et les implémentations pour Android et iOS sont présentées dans la figure 6.9.

6.4.3 Notifications

La fonctionnalité optionnelle pour recevoir des alertes du serveur ne définissait qu'une API abstraite pour afficher les messages à l'utilisateur. L'affichage des messages est une fonctionnalité externe dont l'implémentation varie par plateforme. Le service prend la forme d'une seule méthode abstraite, `notify`, présentée à la figure 6.10.a.

Les implémentations du service appliquent le patron *façade polyglotte*. Dans l'implémentation pour iOS, présentée à la figure 6.10.b, la méthode `notify` est raffinée et elle transforme les arguments vers des types Objective-C. Une méthode `native_notify` est implémentée en Objective-C et réalise l'affichage de la notification avec l'API native à iOS.

Nous utilisons la même stratégie sous Android, mais l'implémentation Java est alourdie par la complexité de l'API native. Le code Java, présenté à la figure 6.11, fait appel à des fonctionnalités natives et réutilise des fragments de code de la documentation officielle

```
redef class Text
  # Ouvre l'URL `self` dans un navigateur web
  #     "http://nitlanguage.org/".open_in_browser
  fun open_in_browser is abstract
end
```

(a) Déclaration abstraite dans l'API UI.

```
redef class Text
  redef fun open_in_browser
  do to_java_string.native_open_in_browser(app.native_activity)
end

redef class JavaString
  private fun native_open_in_browser(context: NativeContext)
  in "Java" `{
    android.content.Intent intent = new android.content.Intent(
      android.content.Intent.ACTION_VIEW, android.net.Uri.parse(self));
    context.startActivity(intent);
  `}
end
```

(b) Implémentation pour Android.

```
redef class Text
  redef fun open_in_browser do to_nsstring.native_open_in_browser
end

redef class NSString
  private fun native_open_in_browser in "ObjC" `{
    NSURL *nsurl = [NSURL URLWithString: self];
    [[UIApplication sharedApplication] openURL: nsurl];
  `}
end
```

(c) Implémentation pour iOS.

Figure 6.9: Fonctionnalité externe `open_in_browser`.

```
# Affiche une notification à l'utilisateur
#
# Chaque catégorie de notification devrait définir un `unique_id`
# distinct pour réutiliser les notifications existantes lorsque
# possible selon la plateforme.
fun notify(title, content: Text, unique_id: Int) is abstract
```

(a) Déclaration abstraite.

```
redef fun notify(title, content, uique_id)
do native_notify(title.to_nsstring, content.to_nsstring)

private fun native_notify(title, content: NSString) in "ObjC" `{
  // Construit la notification
  UILocalNotification* notif = [[UILocalNotification alloc] init];
  notif.alertTitle = title;
  notif.alertBody = content;
  notif.timeZone = [NSTimeZone defaultTimeZone];

  // Lance la notification
  [[UIApplication sharedApplication] presentLocalNotificationNow: notif];
`}
```

(b) Implémentation pour iOS.

Figure 6.10: Déclaration abstraite de `notify` et implémentation.

d'Android. L'icône de la notification est définie dans le dossier de ressources propres à Android. Selon les exigences de la plateforme, l'icône est composée de blanc seulement avec un fond transparent. La notification est associée à une instance de `Intent` pour ouvrir l'application lorsque l'utilisateur sélectionne la notification.

6.5 Application des variations

Le moment où les variations sont liées est un aspect important dans la classification des fonctionnalités d'une ligne de produits, car il détermine qui sélectionne les variations, le programmeur ou l'utilisateur de l'application. Dans le contexte de Tenenit, les fonctionnalités peuvent être classées en deux catégories :

— La majorité des fonctionnalités sont liées à la compilation selon les importations (incluant l'option `-m` de `nitc`) et les raffinements présents dans le code. Elles sont donc sélectionnées par le programmeur selon la plateforme visée et les besoins du commerce.

— Les autres fonctionnalités sont liées à l'exécution. Le navigateur sous Android est sélectionné par l'utilisateur via un menu du système d'exploitation. Les alertes du serveur et le partage de présence sur place sont activés et désactivés à partir de la fenêtre des préférences.

6.6 Discussion

Dans cette section, nous analysons le projet d'application mobile Tenenit de façon à différencier le code portable du code non portable et à évaluer le rôle qu'y a pris notre solution *app.nit*.

Le projet de l'application mobile Tenenit est composé de 1 585 lignes de code[2] dont 79% réalisent le prototype portable et les fonctionnalités optionnelles non exclusives, 12% adaptent l'application à Android, 8% l'adaptent à iOS et 1% définissent l'identité

```
1 redef fun notify(title, content, unique_id)
2 do
3   var service = app.service
4   assert service != null
5   native_notify(service.native, unique_id, title.to_java_string, content.to_java_string)
6 end
7
8 private fun native_notify(context: NativeService, id: Int, title, content: JavaString)
9 in "Java" `{
10  // Récupère l'icône depuis les ressources
11  int icon = context.getResources().getIdentifier(
12    "notif", "drawable", context.getPackageName());
13
14  android.app.Notification.BigTextStyle style =
15    new android.app.Notification.BigTextStyle();
16  style.bigText(content);
17
18  // Prépare la réponse à la pression de la notification
19  android.content.Intent intent = new android.content.Intent(
20    context, nit.app.NitActivity.class);
21  android.app.PendingIntent pendingIntent = android.app.PendingIntent.getActivity(
22    context, 0, intent, android.app.PendingIntent.FLAG_UPDATE_CURRENT);
23
24  // Construit la notification comme telle
25  android.app.Notification notif = new android.app.Notification.Builder(context)
26    .setContentTitle(title)
27    .setContentText(content)
28    .setSmallIcon(icon)
29    .setStyle(style)
30    // ...
31    .build();
32
33  // Lance la notification
34  android.app.NotificationManager notificationManager =
35    (android.app.NotificationManager)context.getSystemService(
36      android.content.Context.NOTIFICATION_SERVICE);
37  notificationManager.notify((int)id, notif);
38 `}
```

Figure 6.11: Implémentation de `notify` pour Android.

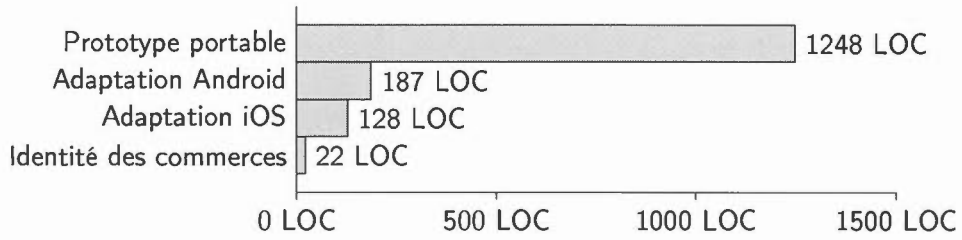


Figure 6.12: Lignes de code selon leur rôle sur un total de 1 585 lignes.

des deux succursales. L'histogramme de la figure 6.12 présente le nombre de lignes de code selon leur rôle.

Parmi le code pour Android, 96 lignes sont en Java, soit 51% du code, regroupées dans cinq méthodes externes. Deux des méthodes externes apportent de simples modifications esthétiques en une seule ligne de code. Dans le cas de iOS, 17 lignes sont en Objective-C, soit 13% du code, encore une fois dans cinq méthodes externes. Nous attribuons la faible proportion de code étranger pour iOS au fait que l'API d'iOS est moins verbeuse que celle d'Android ; de plus, les variations pour iOS sont plus simples, elles ont surtout étés réalisées via les services de l'API UI.

Dans la suite de cette section, pour mettre en lumière l'apport de notre solution en développement d'applications portables, nous la comparons à deux solutions alternatives disponibles sur le marché. Nous estimons par simple déduction les différences que ces autres solutions apporteraient à l'effort de développement. Les deux solutions alternatives consistent à réaliser deux applications natives distinctes et à réaliser une application portable avec React Native.

6.6.1 Solution alternative – Deux applications distinctes

Nous commençons par la solution alternative la plus simple, réaliser deux applications distinctes, une native à Android en Java, et l'autre native à iOS en Objective-C. Pour réaliser le projet Tenenit de cette façon, il faudrait doubler le code portable, 79% du

2. Nous comptons uniquement les lignes avec du code, ignorant les commentaires.

code actuel, de façon à en avoir une version pour les deux langages et plateformes. Les variations sur l'identité de chaque commerce seraient à doubler aussi.

Alors que cette solution aurait tout autant permis d'utiliser les fonctionnalités natives à chaque plateforme et d'appliquer l'expertise préalable, la complexité supplémentaire et le dédoublement de la structure du projet auraient prolongé le temps de développement. Cette solution n'aurait pas permis d'obtenir rapidement un prototype pour chaque plateforme avec aussi peu de ressources que la solution *app.nit*. De plus, toute évolution future du projet, tel que l'ajout d'un commerce ou d'une fonctionnalité partagée entre Android et iOS, nécessiterait deux fois le travail par rapport au code portable utilisé dans notre solution.

6.6.2 Solution alternative – React Native

La meilleure solution en développement d'applications portables, React Native, que nous avions identifiée à la section 1.4, dans l'étude de l'état de l'art, pourrait produire des applications semblables à ce que nous avons obtenu. Par contre, le développement aurait été plus difficile. Pour faciliter la comparaison avec notre solution, nous présentons les différences qu'aurait entraînées React Native selon trois aspects : réaliser un prototype portable, surmonter la fragmentation, et faciliter la programmation polyglotte.

React Native offre des API portables permettant de réaliser rapidement un prototype portable à Android et à iOS. Toutefois, les API et le code portable sont en JavaScript, un langage dynamique sans typage statique. Cette caractéristique limite l'assistance au programmeur en détectant les erreurs de manipulation de données à l'exécution plutôt qu'à la compilation. Le programmeur dépend alors davantage des tests qui sont difficiles à automatiser sur les appareils mobiles.

Pour surmonter la fragmentation du marché et adapter l'application à Android et à iOS, React Native offre deux alternatives. Des variations peuvent être appliquées par des blocs conditionnels à Android ou à iOS avec un simple `if`, regroupant ces différentes préoccupations dans un même fichier. Ou alors, les variations peuvent être apportées

par des fichiers distincts à chaque plateforme, l'outil accepte deux plateformes selon l'extension du fichier `.android.js` ou `.ios.js`. Cette dernière fonctionnalité ne supporte pas de fragmentation plus fine, telle que celle entre les versions d'Android. Alors qu'il n'y avait pas de telle fragmentation fine en Tenenit, nous en avons observé le besoin dans la calculatrice *app.nit*. Les autres sortes de variations, telles que l'identité des commerces, ne sont pas supportées non plus. La seule solution serait l'utilisation du *monkey patching*, qui n'est pas recommandée car les ambiguïtés sur les dépendances de comportement ne sont pas détectées automatiquement.

Le support de la programmation polyglotte en React Native prend la forme de modules natifs, des fichiers Objective-C ou Java dont les fonctions peuvent être invoquées depuis JavaScript. Toutefois, comme leur nom l'indique, pour le moindre fragment de code étranger, tel que les modifications esthétiques de Tenenit, un fichier natif entier doit être ajouté avec du *code colle*. Un tel fichier réalisé en Java doit s'enregistrer auprès du système JavaScript avec plus de 20 lignes de *code colle*, sans compter le code des fonctions effectuant le travail réel. Dans Tenenit, en regroupant tout le code Java dans un même fichier, ces 20 lignes se seraient ajoutées aux 95 lignes effectuant le travail réel. La forme des modules natifs limite l'expressivité en imposant un travail supplémentaire pour accéder au langage étranger et elle nuit à l'apprentissage de la programmation polyglotte par une barrière de complexité à l'entrée.

De plus, les rappels à JavaScript ne sont pas définis par les fonctions natives, mais plutôt par les sites d'appels aux fonctions natives. Les appels à une fonction native doivent passer une fermeture en argument pour chaque rappel à JavaScript. Le code natif, en Objective-C ou Java, exécute les rappels en invoquant la fermeture qui accepte un nombre variable d'arguments (*vararg*). Cette forme correspond au style du langage JavaScript, mais elle ne profite pas de la sûreté statique de Java et d'Objective-C, et ainsi les rappels à JavaScript offrent moins de sûreté statique que les fonctions de rappel générées sur mesure par la FFI de Nit.

En résumé, en tant que développeurs, avec React Native nous aurions, dès le départ, réalisé un prototype portable avec un langage moins sûr que Nit. La séparation du

code des différentes préoccupations et variations aurait été limitée à deux plateformes seulement. Les autres variations, telle que l'identité des commerces, n'auraient pas pu être appliquées de façon fiable. Nous aurions eu à écrire le *code colle* répétitif pour accéder aux langages natifs et nous aurions perdu la sûreté statique des langages natifs lors des rappels à JavaScript.

Alors que la forme actuelle de React Native a de fortes limitations au niveau de la sûreté statique, de la fiabilité des lignes de produits et de la programmation polyglotte, celles-ci ne sont pas insurmontables. Dans la conclusion de ce document, nous présenterons des pistes de solution pour améliorer React Native et y apporter les mêmes avantages que ceux de notre solution *app.nit*.

6.7 Conclusion

Nous avons réalisé le projet Tenenit en appliquant les trois composants de notre solution *app.nit*, ainsi que les trois patrons de conception polyglotte.

Le prototype rassemble les fonctionnalités principales du projet et il est compilable pour Android et pour iOS. Il utilise les quatre API de la bibliothèque *app.nit*, soit l'API UI pour construire l'interface graphique, l'API cycle de vie pour déclencher des événements au lancement de l'application, l'API requêtes HTTP pour communiquer avec le serveur, et l'API persistance des données pour conserver les préférences de l'utilisateur. De plus, le prototype nous sert encore sous GNU/Linux, car il aide à l'implémentation de nouvelles fonctionnalités et au débogage.

Le projet est organisé en ligne de produits pour surmonter la fragmentation du marché des appareils mobiles et produire une famille d'applications semblables. Une base de code commune réalise le prototype portable et certaines fonctionnalités optionnelles dont les alertes du serveur et l'annonce de présence. Des fonctionnalités optionnelles exclusives modifient le prototype en applications adaptées à Android et à iOS, en plus, elles configurent les applications pour les différents commerces. Des fonctionnalités externes

offrent une interface abstraite implémentée différemment sous Android et sous iOS, autant la bibliothèque *app.nit* que les notifications sont réalisées par des services des API natives. Les produits sont les applications mobiles, chacune adaptée à une plateforme et configurée pour un commerce précis.

La programmation polyglotte tient un rôle important dans le projet Tenenit. Autant les variations locales que la bibliothèque *app.nit* accèdent aux fonctionnalités propres à chaque plateforme via leur API native et dans leur langage natif. Ceci est rendu possible par la FFI de Nit, dont les trois interfaces frontales agissent à différents endroits dans le code. La FFI avec C ouvre l'accès à l'API C du NDK d'Android qui est fondamental au fonctionnement de Nit sous Android. La FFI avec Objective-C permet les adaptations esthétiques sous iOS ainsi que l'implémentation des notifications. La FFI avec Java permet en plus d'ajouter une fenêtre pour évaluer les bières et de détecter les réseaux Wi-Fi sous Android.

Finalement, Tenenit applique aussi les trois patrons de conception polyglotte. Le patron *façade polyglotte* guide le passage des données pour les modifications esthétiques, les notifications et la fenêtre d'évaluation des bières. La détection des réseaux Wi-Fi applique le patron *spécialisation polyglotte* en utilisant les rappels à Nit pour spécialiser une interface Java en Nit et personnaliser la réponse à la détection d'un nouveau réseau. Finalement, le patron *adaptateur polyglotte* agit dans les implémentations des API *app.nit* pour reproduire les API natives en Nit.

CONCLUSION GÉNÉRALE

Les appareils mobiles sont maintenant une plateforme de choix pour toutes sortes d'applications. Toutefois, lorsqu'un grand public est visé, la fragmentation du marché entre Android et iOS est une embûche importante. Bien que l'industrie et la littérature scientifique proposent quelques solutions pour réaliser des applications portables à Android et à iOS, elles ne comblent pas tous les critères que nous jugeons prioritaires : produire un prototype portable rapidement, favoriser la portabilité du code entre les plateformes, surmonter la fragmentation du marché, intégrer l'application à l'écosystème, donner accès aux API natives, favoriser le transfert de l'expertise, simplifier l'évolution du projet, et finalement, assister le programmeur en lui évitant de faire des erreurs.

La principale contribution de cette thèse est une nouvelle solution en développement d'applications mobiles et portables. Surnommée *app.nit*, elle fait un compromis sur la portabilité du code pour répondre à tous nos critères et favoriser l'adaptation des applications à Android et à iOS. Elle combine trois approches théoriques avec une solution pratique : des API portables offertes par la bibliothèque *app.nit*, l'organisation en ligne de produits réalisée par le raffinement de classes, et la programmation polyglotte facilitée par la FFI de Nit.

API portables et bibliothèque *app.nit*

Des API portables permettent de réaliser rapidement un prototype portable à Android et à iOS, et ce, en Nit pur. Le prototype est une version de l'application non finale par définition, mais qui rassemble les fonctionnalités principales du projet. Il peut être publié ou servir à trouver du financement et ainsi décider de l'avenir du projet. Les API portables servent de fondation fiable à notre solution, il s'agit d'une approche éprouvée pour développer des applications portables.

La bibliothèque *app.nit* offre quatre API pour répondre aux besoins typiques des applications mobiles. L'API UI définit des contrôles pour réaliser une interface utilisateur tactile portable, ces contrôles de l'API sont implémentés par les contrôles natifs à chaque plateforme. L'API cycle de vie permet à l'application de répondre aux messages des systèmes d'exploitation lorsqu'elle change d'état. L'API persistance des données offre des services simples pour conserver l'état de l'application lors d'interruptions et pour préserver les préférences de l'utilisateur entre deux exécutions. Finalement, l'API de requêtes HTTP permet de communiquer avec un serveur distant et automatise la logique asynchrone de façon transparente. Alors que la bibliothèque *app.nit* peut toujours être étendue avec plus de services et d'API, elle a été suffisante pour répondre aux besoins en code portable de nos applications de validation dans son état actuel.

De plus, plusieurs paquets de la bibliothèque étendue de Nit sont portables à Android, iOS, macOS et GNU/Linux. Ils peuvent être utilisés autant par les applications mobiles que les logiciels serveur.

Ligne de produits et raffinement de classes

L'organisation en ligne de produits permet de surmonter la fragmentation du marché et de réaliser une famille d'applications mobiles semblables de haute qualité et à faible coût. Des fonctionnalités obligatoires sont partagées par tous les produits alors que des fonctionnalités optionnelles différencient les produits entre eux. En pratique, cette organisation permet de faire évoluer graduellement le prototype en applications adaptées à Android et à iOS qui s'intègrent à leur écosystème.

Dans un projet d'application mobile *app.nit*, les lignes de produits sont réalisées par le raffinement de classes de Nit, une bonification du paradigme de programmation orientée objet qui utilise les objets pour encapsuler les données et des modules pour séparer les préoccupations et les variations dans le code. Chaque module peut ouvrir toute classe existante pour en modifier les propriétés : attributs, méthodes et constructeurs. De plus, un module peut leur ajouter des propriétés et même des super-classes.

Tout engin d'exécution Nit, dont le compilateur *nitc*, permet de lier les variations en produits. En assistance au programmeur, le compilateur rapporte toute ambiguïté sur les dépendances de comportement. Celles-ci surviennent lorsque le compilateur ne peut pas déterminer l'ordre de linéarisation d'une méthode, c'est-à-dire la priorité d'exécution des multiples implémentations d'une même méthode.

Programmation polyglotte et FFI de Nit

La programmation polyglotte consiste à utiliser plusieurs langages pour réaliser une même application. Un projet *app.nit* rassemble généralement quatre langages, chacun ayant ses spécialités. Le langage Nit et ses bibliothèques servent à réaliser le code et le prototype portable. Le langage C permet d'écrire du code optimisé et près de la machine, et il implémente de nombreuses fonctionnalités de bas niveau de la bibliothèque Nit. Le langage Java donne accès aux API natives à Android dans leur entièreté, et Objective-C à celles d'iOS. L'accès aux API entières dans leur langage natif facilite le transfert de connaissances et, ayant accès aux mêmes services de la même façon, permet d'atteindre le niveau de qualité des applications natives.

La programmation polyglotte est facilitée par la FFI de Nit, une FFI qui innove par la combinaison de quatre caractéristiques :

— Elle s'intègre au paradigme de programmation orientée objet par l'offre de méthodes externes, de classes externes et de rappels à Nit.
— Elle imbrique le code étranger parmi le code Nit pour en faciliter la lisibilité et l'apprentissage en évitant au programmeur d'écrire du code colle répétitif.
— Elle accepte plusieurs langages étrangers en profitant de leurs aspects communs pour exposer une interface semblable entre les différents langages.
— Elle génère sur mesure les services pour le code étranger de façon à préserver la sûreté statique des langages étrangers et assister le programmeur en détectant des erreurs de manipulation des données à la compilation.

De plus, nous avons introduit trois patrons de conception polyglotte qui donnent des solutions abstraites à des problèmes concrets que nous avons observés au cours de ce projet de recherche. Les patrons définissent aussi de bonnes pratiques pour l'utilisation de la FFI de Nit.

— Le patron *façade polyglotte* structure le passage de données lors de l'invocation de code étranger depuis Nit en traitant chaque donnée dans le langage spécialisé.
— Le patron *adaptateur polyglotte* reproduit des API étrangères à objets en Nit et automatise la libération des objets étrangers en utilisant des classes ordinaires Nit pour encapsuler des objets étrangers.
— Le patron *spécialisation polyglotte* permet de personnaliser la réponse aux événements d'une API étrangère en spécialisant des interfaces étrangères en Nit.

Combinaison des trois approches

La combinaison des trois approches est une solution complète pour réaliser un prototype portable rapidement et le faire évoluer graduellement vers des applications adaptées à Android et à iOS. Les applications adaptées, ayant accès à toutes les fonctionnalités natives dans les langages natifs, peuvent atteindre le même niveau de qualité que les applications natives. L'expressivité de la FFI de Nit facilite l'utilisation des langages natifs de façon à en encourager la lisibilité et l'apprentissage, et ce, pour les cas simples comme pour les cas complexes.

Validation

Nous avons d'abord validé différents aspects de notre solution en réalisant deux petites applications, la calculatrice *app.nit* et le client mobile Tnitter. Ensuite nous avons réalisé un projet réel, la ligne de produits Tenenit, qui applique entièrement notre solution *app.nit*.

La calculatrice *app.nit* utilise l'API UI, l'API cycle de vie et l'API persistance des données. Elle est organisée en ligne de produits avec des variations pour un mode scientifique et pour l'adapter aux plateformes iOS, Android API 14 et Android API 21.

Le client mobile au microblogue Tnitter utilise surtout l'API UI et l'API requêtes HTTP. Il a servi à valider certains contrôles graphiques qui ne sont pas utilisés par la calculatrice, et à tester des cas limites de l'API requêtes HTTP lors des *push notifications*.

La ligne de produits Tenenit est un projet complexe d'une taille réelle. Il s'agit d'une application mobile pour les clients de brasseries artisanales. Elle affiche le menu journalier de bières et sert d'interface à un petit réseau social. Le prototype portable utilise les quatre API *app.nit* en plus de services portables de la bibliothèque de Nit.

Les applications mobiles produites par la ligne de produits Tenenit partagent des fonctionnalités obligatoires qui composent aussi le prototype. Les applications se différencient par des fonctionnalités optionnelles qui sont exclusives ou non. Les fonctionnalités exclusives adaptent les applications à chaque plateforme ou à un commerce précis. Les fonctionnalités non exclusives offrent des services facultatifs telles que recevoir les alertes du serveur et annoncer sa présence sur place.

Les adaptations à chaque plateforme appliquent la programmation polyglotte pour accéder aux services propres à Android et à iOS via la FFI de Nit. Pour Android, des méthodes externes et des rappels à Nit permettent d'ouvrir une fenêtre pour évaluer les bières et de détecter les réseaux Wi-Fi. Pour iOS, les modifications sont plus simples, améliorant l'apparence des contrôles pour une meilleure intégration à l'écosystème.

Contributions

Cette thèse apporte cinq contributions scientifiques :

— La combinaison des trois approches pour offrir une solution complète en développement d'applications mobiles et portables, telle qu'appliquée par la solution *app.nit*.

— L'analyse du raffinement de classes pour réaliser les artéfacts de code des lignes de produits apporte un nouveau point de vue sur le raffinement de classes et un nouveau langage pour en décrire l'application.

— La spécification de la FFI de Nit qui innove sur plusieurs points : elle s'intègre au paradigme à objets, elle supporte plusieurs langages étrangers à objets et elle génère sur mesure des services pour le code étranger de façon à préserver leur sûreté statique. De plus, elle étend la syntaxe imbriquée, proposée à l'origine dans la littérature scientifique, de façon à déclarer le langage étranger et les rappels à Nit.

— La définition de trois patrons de conception polyglotte donne des solutions abstraites à des problèmes de programmation polyglotte concrets. En plus de définir de bonnes pratiques à l'utilisation de la FFI de Nit, les trois patrons établissent une structure et un langage sur lesquels une étude future pourrait développer.

— L'étude de cas du projet Tenenit explore une utilisation pratique de notre solution pour réaliser une application mobile complexe.

Cette thèse apporte quatre principales contributions techniques :

— La conception de la FFI de Nit ainsi que son implémentation entière dans le compilateur et partielle dans l'interpréteur. Cet effort inclut la modification de la grammaire Nit pour y imbriquer les langages étrangers, ainsi que le support des langages C, Objective-C et Java.

— Le développement de la bibliothèque *app.nit* qui offre, via quatre API, des services spécialisés pour les applications mobiles. Ainsi que l'implémentation de tous les services séparément pour Android et pour iOS. De plus, nous avons mis à jour la bibliothèque standard de Nit pour en assurer la portabilité aux deux plateformes.

— La modification du compilateur Nit pour qu'il produise des applications Android et iOS. Pour ce faire, nous avons ajouté des modules spécialisés pour chaque plateforme qui génèrent des projets d'applications natives attendus par les outils standard de développement Android et iOS.

— Le développement des trois applications de validations, d'abord la calculatrice et le client mobile Tnitter, qui ont servi à valider des parties spécifiques de notre solution. Ensuite, le projet Tenenit, sujet à l'étude de cas, qui a servi à valider notre solution en entier.

Travaux futurs

Ce projet de recherche peut servir de base à des travaux et expérimentations futurs. Nous avons déjà mentionné trois travaux possibles pour améliorer des composants de notre solution :

- Pour combler une limitation de la FFI avec Java, mentionnée à la section 4.7.4, la FFI pourrait générer sur mesure des types Java pour les références opaques aux objets Nit. De cette façon, la sûreté statique de Java améliorerait l'assistance au programmeur. Le défi est de supporter la compilation séparée et de ne pas encourir un surcoût en performance par l'utilisation d'un grand nombre de classes. De plus, il serait intéressant d'étudier la reproduction de la hiérarchie de classes Nit en Java.
- Intégrer les analyses proposées par la littérature scientifique à la FFI avec Java, listées à la section 4.7.14, apporterait une assistance supplémentaire au programmeur. De plus, certaines analyses pourraient être adaptées au langage Nit où les appels à Java sont plus restrictifs que ceux faits via l'API C de la JNI.
- Des expérimentations pratiques seraient nécessaires pour concevoir et évaluer le patron de conception unique, mentionné à la section 5.3.12, qui rassemblerait les patrons *adaptateur polyglotte* et *spécialisation polyglotte*. Il permettrait de répliquer des API étrangères à objets en Nit et de spécialiser des interfaces étrangères en Nit. De plus, répondant à la majorité des besoins, ce patron pourrait être généré automatiquement.

Nous présentons ici d'autres sujets intéressants pour des travaux futurs qui élargiraient la portée de notre solution :

- Notre solution, *app.nit*, pourrait être étendue pour supporter les nouveaux appareils qui s'ajoutent à la catégorie des appareils mobiles. Notamment, les montres *Android Wear* et *Apple Watch* apportent de nouvelles restrictions, dont des écrans encore plus petits qui forcent l'application à dépendre davantage de l'intuition de

l'utilisateur et des commandes vocales. Déterminer la limite de ce qui est portable aux téléphones et aux montres est en soi un défi important. De plus, les différences avec les téléphones mobiles causeraient une différente fragmentation du marché, menant à des lignes de produits plus complexes avec un rôle encore plus important.

— La FFI de Nit pourrait être étendue pour supporter d'autres langages étrangers avec de nouvelles spécialités. Le support du langage Swift offrirait une alternative de plus haut niveau à Objective-C pour accéder aux API natives d'iOS. Pour Android, Scala serait une alternative à Java ajoutant le paradigme de programmation fonctionnelle, son implémentation serait simplifiée par le code octet commun aux deux langages. De plus, le support de C# donnerait accès aux API natives du *Universal Windows Platform* pour réaliser des applications Windows 10 compatibles avec les appareils mobiles.

Au-delà du langage Nit

Tout au long de ce document, nous avons discuté des fonctionnalités de Nit qui supportent le développement d'applications mobiles et portables. Nous avons profité de caractéristiques préexistantes de Nit, telles que le raffinement de classes, la compilation en code C portable et sa bibliothèque standard. Nous y avons aussi ajouté des fonctionnalités pour répondre à nos besoins, dont la FFI de Nit avec le support de C, Objective-C et Java, ainsi que la bibliothèque *app.nit*. Toutefois, nous croyons que notre solution peut aussi être appliquée par d'autres langages de programmation et technologies, en utilisant des outils externes et avec certaines modifications.

Une solution pour offrir un équivalent en Java consisterait à combiner la programmation orientée aspects d'AspectJ avec une FFI comme Janet ou Jeanie. Ces FFI sont présentement limitées au langage C, mais il serait possible de les bonifier pour qu'elles supportent les langages Objective-C et Java. Ensuite, le principal obstacle est l'exécution du code Java sous iOS. Pour ce faire, le code Java peut être soit recompilé en code natif avec XMLVM, soit exécuté par une machine virtuelle pour iOS. Il faudrait aussi bonifier

l'offre de services de la bibliothèque standard de Java par une bibliothèque spécialisée pour applications mobiles offrant les fonctionnalités communes à Android et à iOS.

Ou encore, React Native pourrait être amélioré pour assister davantage le programmeur, supporter les lignes de produits et faciliter la programmation polyglotte avec une FFI imbriquée. Le langage JavaScript n'offre pas une grande l'assistance à l'utilisateur et le manque d'information statique limite le code qui peut être généré automatiquement. Le langage pourrait être remplacé par un langage recompilé en JavaScript avec une meilleure sûreté statique, tel que TypeScript (Bierman, Abadi et Torgersen, 2014). Pour supporter des lignes de produits complètes, il serait possible d'ajouter le raffinement de classes à TypeScript, ou encore de développer un outil de recompilation source à source, d'une façon semblable à ce qu'offre AspectJ. L'ajout d'une FFI imbriquée encouragerait la programmation polyglotte en réduisant la barrière, ou la difficulté, à implémenter des services avec des langages natifs. La FFI pourrait prendre la forme d'un compilateur source à source qui produirait du code JavaScript (ou TypeScript) standard et les modules natifs attendus par React Native.

BIBLIOGRAPHIE

Anastasopoulos, M., et D. Muthig. 2004. « An evaluation of aspect-oriented programming as a product line implementation technology ». In *International Conference on Software Reuse (ICSR)*. Springer.

AOSP. 2016a. Android Developer Dashboards. <http://developer.android.com/about/dashboards/>.

———. 2016b. Support Library. <https://developer.android.com/topic/libraries/support-library/index.html>.

———. 2016c. The Activity Lifecycle. <https://developer.android.com/guide/components/activities/activity-lifecycle.html>.

Apache Software Foundation. 2013. Apache Cordova. <http://cordova.apache.org/>.

Appcelerator. 2016. Appcelerator. <http://www.appcelerator.com/>.

Apple Inc. 2016. The App Life Cycle. <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html>.

Apportable. 2013. Apportable Raises $2.4M Led by Google Ventures to 'Translate' Apps from iOS to Android. <http://pevc.dowjones.com/Article?an=DJFVW000201307l7e97haongz>.

Appportable. 2016. Multi-platform mobile apps the smart way. <https://web.archive.org/web/20180117180529/http://www.apportable.com/>.

Bajolet, L. 2016. *Des chaînes de caractères efficaces et résistantes au passage à l'échelle, une proposition de modélisation pour les langages de programmation à objets.* Mémoire de maîtrise, Université du Québec à Montréal.

Bierman, G., M. Abadi et M. Torgersen. 2014. « Understanding typescript ». In *European Conference on Object-Oriented Programming*. Springer.

Boehm, H.-J. 2003. « Destructors, finalizers, and synchronization ». In *SIGPLAN Notices*. T. 38. ACM.

Boehm, H.-J., et M. Weiser. 1988. « Garbage collection in an uncooperative environment ». *Software : Practice and Experience*, vol. 18, no. 9.

Botterweck, G., K. Lee et S. Thiel. 2009. « Automating product derivation in software product line engineering. ».

Bubak, M., D. Kurzyniec et P. Luszczek. 2000. « Creating Java to native code interfaces with Janet extension ». In *Worldwide SGI Users' Conference.*

CheckJNI. 2016. Java 8 other command-line options. <`https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/clopts002.html`>.

Cho, H., K. Lee et K. C. Kang. 2008. « Feature relation and dependency management : an aspect-oriented approach ». In *Software Product Line Conference (SPLC).* IEEE.

Clements, P., et L. Northrop. 2002. « Software product lines : practices and patterns ».

comScore. 2015. 2015 U.S. Digital Future in Focus. <`http://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/2015-US-Digital-Future-in-Focus`>.

Díaz, J., J. Pérez, P. P. Alarcón et J. Garbajosa. 2011. « Agile product line engineering—a systematic literature review ». *Software : Practice and experience*, vol. 41, no. 8.

Dijkstra, E. W. 1976. *A discipline of programming.* Prentice Hall.

Ducournau, R., F. Morandat et J. Privat. 2008. Modules and class refinement — a meta-modeling approach to object-oriented languages. Rapport no. 07021, LIRMM – Université de Montpellier II.

Ducournau, R., F. Morandat, J. Privat *et al.* 2009. « Empirical assessment of object-oriented implementations with multiple inheritance and static typing ». In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Ducrohet, X. 2011. Fragments for all. <`http://android-developers.blogspot.com/2011/03/fragments-for-all.html`>.

Facebook Inc. 2016a. React native. <`http://facebook.github.io/react-native/`>.

———. 2016b. React native : Native modules. <`http://facebook.github.io/react-native/docs/native-modules-android.html`>.

Flanagan, D., et Y. Matsumoto. 2008. *The Ruby programming language.* O'Reilly Media.

Furr, M., et J. Foster. 2006. « Polymorphic type inference for the JNI ». *European Symposium on Programming (ESOP).*

Gacek, C., et M. Anastasopoulos. 2001. « Implementing product line variabilities ». In *SIGSOFT Software Engineering Notes*. T. 26. ACM.

Gamma, E., R. Helm, R. Johnson et J. Vlissides. 1994. *Design Patterns : Elements of Reusable Object-Oriented Software*. Pearson Education.

Gartner. 2016. Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015. <http://www.gartner.com/newsroom/id/3215217>.

Gélinas, J.S., Gagnon, E. et Privat, J. 2009. « Prévention de déréférencement de références nulles dans un langage à objets ». *Langages et Modèles à Objets (LMO)*, vol. 3.

GIWS. 2012. « A wrapper generator to generater C++ mapping Java classes ». <https://github.com/opencollab/giws>.

Greene, J. 2016. Microsoft to buy app-development startup Xamarin. <http://www.marketwatch.com/story/microsoft-to-buy-app-development-startup-xamarin-2016-02-24>.

Günther, S., et S. Sunkle. 2009. « Feature-oriented programming with Ruby ». In *International Workshop on Feature-Oriented Software Development*. ACM.

———. 2010. « Dynamically adaptable software product lines using Ruby metaprogramming ». In *International Workshop on Feature-Oriented Software Development*. ACM.

Hirzel, M., et R. Grimm. 2007. « Jeannie : Granting Java Native Interface developers their wishes ». In *Object-Oriented Programming Systems and Applications (OOPSLA)*.

Joorabchi, M. E., A. Mesbah et P. Kruchten. 2013. « Real challenges in mobile app development ». In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE.

Kastner, C., T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz et S. Apel. 2009. « Featureide : A tool framework for feature-oriented software development ». In *International Conference on Software Engineering (ICSE)*. IEEE.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin. 1997. « Aspect-oriented programming ». In *European Conference on Object-Oriented Programming (ECOOP)*.

Kondoh, G., et T. Onodera. 2008. « Finding bugs in Java Native Interface programs ». In *International Symposium on Software Testing and Analysis*, p. 109–118.

Laferrière, A. 2012. *L'interface native de Nit, un langage de programmation à objets*. Mémoire de maîtrise, Université du Québec à Montréal.

Lee, B., B. Wiedermann, M. Hirzel, R. Grimm et K. McKinley. 2010. « Jinn : synthesizing dynamic bug detectors for foreign language interfaces ». In *Programming Language Design and Implementation (PLDI)*.

Li, S., et G. Tan. 2011. « JET : exception checking in the Java Native Interface ». In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

Li, S., et G. Tan. 2014. « Exception analysis in the Java Native Interface ». *Science of Computer Programming*, vol. 89.

Liang, S. 1997. Java Native Interface specification. Rapport, JavaSoft.

Martin, R. C. 2003. *Agile software development : principles, patterns, and practices*. Prentice Hall.

MSDN. 2016a. Extension methods (C# programming guide). <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>.

———. 2016b. Partial classes and methods (C# programming guide). <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods>.

Muccini, H., A. Di Francesco et P. Esposito. 2012. « Software testing of mobile applications : Challenges and future research directions ». In *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE.

Necula, G., S. McPeak et W. Weimer. 2002. « CCured : type-safe retrofitting of legacy code ». In *Symposium on Principles of Programming Languages (POPL)*.

Occhino, T. 2015. React native : Bringing modern web techniques to mobile. <https://code.facebook.com/posts/1014532261909640/>.

Pohl, K., G. Böckle et F. J. van Der Linden. 2005. *Software product line engineering : foundations, principles and techniques*. Springer Science & Business Media.

Privat, J. 2002. « Analyse de types et graphe d'appels en compilation séparée ». *Mémoire de DEA, Université Montpellier II*.

Privat, J. 2006. « De l'expressivité à l'efficacité : une approche modulaire des langages à objets : le langage prm et le compilateur prmc ». Thèse de Doctorat, Université Montpellier II.

Privat, J., et R. Ducournau. 2005. « Link-time static analysis for efficient separate compilation of object-oriented languages ». In *Program Analysis for Software Tools and Engineering (PASTE)*.

Puder, A., et O. Antebi. 2013. « Cross-compiling Android applications to iOS and Windows Phone 7 ». *Mobile Networks and Applications*, vol. 18, no. 1.

Puder, A., et J. Lee. 2009. « Towards an XML-based bytecode level transformation framework ». *Electronic Notes in Theoretical Computer Science (ENTCS)*.

Puder, A., et I. Yoon. 2010. « Smartphone cross-compilation framework for multiplayer online games ». In *Conference on Mobile, Hybrid, and On-line Learning (eLmL)*.

Svahnberg, M., J. Van Gurp et J. Bosch. 2005. « A taxonomy of variability realization techniques ». *Software : Practice and Experience*, vol. 35, no. 8.

Tan, G. 2010. « JNI light : An operational model for the core JNI ». *Asian conference on Programming Languages and Systems (APLAS)*.

Tan, G., A. Appel, S. Chakradhar, A. Raghunathan, S. Ravi et D. Wang. 2006. « Safe Java Native Interface ». In *International Symposium on Software Reliability Engineering (ISSRE)*.

Tan, G., et G. Morrisett. 2007. « ILEA : Inter-language analysis across Java and C ». In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Terrasa, A., et J. Privat. 2013. « Efficiency of subtype test in object oriented languages with generics ». In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ACM.

Wasserman, A. I. 2010. « Software engineering issues for mobile application development ». In *FSE/SDP workshop on Future of software engineering research*. ACM.

Young, T. J. 2005. « Using AspectJ to build a software product line for mobile devices ». Mémoire de maîtrise, University of British Columbia.