Gestion de la co-évolution des logiciels partiellement générés pendant la phase d'évolution et de maintenance

Djamel E. Khelladi, Olivier Barais, Benoît Combemale, Mathieu Acher,
Arnaud Blouin, Johann Bourcier, Noel Plouzeau, Jean-Marc Jézéquel

Univ Rennes, INRIA, IRISA Laboratory CNRS, Univ Rennes, DiverSE Team, IRISA Laboratory
{firstname.lastname}@irisa.fr

1 Introduction et contexte

Une des visions poussées par une partie du GDR il y a maintenant 20 ans est une réalité : les utilisateurs définissent leurs propres abstractions au travers de langages dédiés qu'ils outillent (exemple, des générateurs de code) dans le but d'augmenter leur productivité. Cette vision prend notamment forme via les applications cloud native [4] construites à l'aide d'un ensemble de micro-services et mises en oeuvre à l'aide de différents langages de programmation. Les développeurs utilisent de plus en plus de générateurs de code pour initialiser de telles applications complexes à partir d'abstractions définies à haut niveau (abstraction réifiée dans des DSL, dans des wizards, ...). Le débat n'est pas pour un développeur de savoir avec quel langage ou quel framework il en utilisera plusieurs en fonction des contraintes des plate-formes d'exécution, ses compétences, etc.

Cette tendance liée à l'utilisation de générateurs ou template de code est clairement mise en avant dans les lessons apprises mis en avant par [2, 5]. Et c'est même explicitement mis en avant par Balalaie et al. [1] dans la citation suivante :

"creating service development templates is important. Polyglot persistence and the use of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in chaos in the system and even make it unmaintainable. As a solution, after architectural refactoring began, we started to create service development templates. We have different templates for creating microservices in Java using different data stores ; these templates include a simple sample of a correct implementation. We're also creating templates for Node.js. One simple rule is that a senior developer should first examine each new template to identify potential challenges."

On peut citer par exemple, les générateurs de code : définis sous la forme d'un archetype Maven ; mis en oeuvre dans la plupart des outils en ligne de commande associés à un framework technique particulier tel que AngularCLI et WordPress tool ; associés à un ensemble de langages dédiés (e.g., modèles JDL) qui permettent de générer les souches et les squelettes liés aux protocoles de communications distribués que l'on trouve entre ces services (générateur openapi, asyncAPI, etc.). Une des différences notables que l'on peut faire entre le monde des générateurs de code sur ce type d'application et le monde des compilateurs et qu'il est attendu que le développeur modifie le code généré afin de raffiner la mise en oeuvre de son microservice.

Cet état de la pratique montre clairement son intérêt afin de laisser les différents membres d'un projet travailler avec le bon niveau d'abstraction tout en leur permettant de raffiner la mise en oeuvre dans des langages de programmation efficace, bien outillé. Cependant, cet état de la pratique lève aussi un challenge pour la communauté SLE (Software Language Engineering), la communauté du versionning de code et la communauté de l'évolution et de la maintenance de code au sens large. Comment faire co-évoluer automatiquement du code généré et du code produit manuallement ? Comment co-évoluer les données produites et stockées dans les bases de données ? Qu'en est-il de la documentation ? etc.

La complexité de cette démarche est telle que dans la plupart des projets industriels, on observe soit une approche de type "generated once" visant à utiliser les générateurs une seule fois. Les évolutions futures ne sont alors plus capturées aux bons niveaux d'abstractions. Or, penser et raisonner l'évolution à un haut niveau d'abstraction permet entre autres de bien la spécifier et de garder traces des décisions et de leurs implications dans le système. Soit avec des approches du type compilateur pour lequel les utilisateurs essaient de ne jamais modifier le code généré tant qu'il y a un risque d'évolution à décrire au niveau des modèles de haut niveau génant fortement le besoin d'agilité dans un projet de développement moderne. En pratique, ce scénario est peu fréquent au vu des besoins d'évolution et de maintenance pour rester compétitif sur le marché.

Un cas d'étude intéressant pour comprendre ce phénomène est JHipster [6]. Projet opensource dynamique visant à faciliter la mise en oeuvre d'un service ou d'un micro-service en laissant un certain nombre de points de variation. Pour montrer cette pratique, nous pouvons regarder quelques chiffres liées à ce projet.

— Ils ont atteint 15 000 étoiles (de popularité) le mois dernier.
— Le nombre d'installations continue de croître au même rythme que les années précédentes.
— Ils ont maintenant plus de 120 000 installations par mois.
— Les téléchargements de nos artefacts Maven suivent la même tendance. Le mois dernier, ils ont atteint 272 000 téléchargements sur Maven Central.
— Ils ont eu des utilisateurs du monde entier. En fait, seulement trois pays n'ont pas utilisé JHipster !

JHipster est intéressant, car il cumule différents types de générateur de code. Un générateur de code pour initialiser le projet, un générateur construit par dessus un langage dédié de modélisation des entités métiers (.JDL), l'utilisation de générateur de code issu d'Angular CLI, l'utilisation de générateur de code (talon/squelette) pour mettre en oeuvre ou être client d'un service dont l'interface est définie à l'aide d'OpenAPI. Enfin JHipster génère du code vers un ensemble de plateformes, (code SQL ou liquibase pour la création et l'évolution du stockage des données), code Java pour la partie serveur, code TypeScript ou JavaScript pour la partie cliente, descripteur de projet (pom.xml, package.json), de déploiement. (yml).

Si l'on peut voir ce type de projet comme une vitrine pour vanter l'utilisation d'abstraction et d'informatique générative intégrée à un écosystème technique à l'état de l'art, il reste plusieurs points noirs associés à l'utilisation de JHipster. La principale concerne ce que l'on nomme la co-évolution abstractions mise en oeuvre en présence du code généré modifié. Ce problème n'est pas pleinement nouveau et la communauté IDM (Ingénierie Dirigée par les Modèles) [7, 3] l'a relevé depuis longtemps, pour autant ce problème n'a pas encore trouvé de réponse cohérente pleinement utilisée d'un point de vue industriel. Ceci s'explique probablement en partie, car ce problème touche de nombreuses communautés :

— Sans aucun doute, la communauté IDM et SLE doivent sans doute réfléchir à de nouveaux moyens de structurer ses générateurs de code.
— la communauté langage afin de mieux mettre en évidence dans les langages de programmation les éléments générés et les éléments manuellement décrits.
— La communauté liée au software repository afin de fournir des outils de gestion de source conscient de ses différences entre artefacts générés et artefacts modifiés.
— La construction des IDEs afin de nouveau de mieux s'adapter à ces méthodes de développement polyglots au sein desquels de nombreux générateurs de code sont utilisés.
— La communautés SPL qui permet très généralement de définir l'orchestration et la paramétrisation de ces générateurs de code.

Si ce problème de co-évolution est complexe, nous disposons maintenant d'une base de connaissances riches (repository de code, de dépendances, ...), nous disposons d'une capacité de calcul plus importante permettant d'évaluer plusieurs solutions de co-évolution automatique afin d'en proposer aux développeurs, nous disposons d'IDE plus riches. Nous proposons sur ce défi de réfléchir à l'automatisation de cette tâche de co-évolution en utilisant au mieux ses bases de connaissances disponibles, la puissance de calcul disponible, de nouvelles constructions de langages, etc. Par exemple, la construction d'un langage d'annotation pour le code qui doit être co-évolué ou même des annotations de co-évolution spécifiques. Par ailleurs, les lignes de produits pourraient aussi être utiles pour remonter des évolutions similaires faites sur des codes de produits dérivés d'un même "feature model".

Références

[1] Balalaie, A., Heydarnoori, A., Jamshidi Dermani, P. : Microservices architecture enables devops : An experience report on migration to a cloud-native architecture

[2] Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S. : Empirical assessment of mde in industry. In : Proceedings of the 33rd International Conference on Software Engineering. pp. 471–480. ACM (2011)

[3] Kleppe, A.G., Warmer, J., Warmer, J.B., Bast, W. : MDA explained : the model driven architecture : practice and promise. Addison-Wesley Professional (2003)

[4] Kratzke, N., Quint, P.C. : Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software **126**, 1–16 (2017)

[5] Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J. : Assessing the state-of-practice of model-based engineering in the embedded systems domain. In : Model-Driven Engineering Languages and Systems, pp. 166–182. Springer (2014)

[6] Raible, M. : The JHipster mini-book. Lulu. com (2016)

[7] Schmidt, D.C. : Model-driven engineering. COMPUTER-IEEE COMPUTER SOCIETY- **39**(2), 25 (2006)