B-007

IDE の自動修正機能を用いた周辺コード補完ツール
A Peripheral Code Completion Tool with IDE Auto Correction

上村 勇太† Yuta Kamimura
橋浦 弘明† Hiroaki Hashiura

1. はじめに

今日、プログラミング言語には多種多様なものが存在しており、実行前にコンパイルが必要な言語におけるコンパイルエラーの発生とその解決に開発者は多大な時間と労力を費やしている[1]。本研究では Language Server を用いた IDE と依存関係解決ツールを組み合わせる事で、外部ライブラリに依存したコード片を実行可能なツールを開発し、その評価を行なった結果について述べる。

2. 研究目的

本研究の目的は、開発者が普段から行っているコード片の再利用の作業を支援することである。著者らは、先行研究[2]においては標準的なコード片に対して IDE の機能を用いた自動補完を行う機構を提案した。しかしながら、この機構は 外部ライブラリに依存しているコード片を補完する事が不可能であった。加えて、コード片には 依存ライブラリがコード上に明記されていないコード片も存在する。

例えば、プログラミング向け質問サイトである Stack Overflow にて “java json modify” というキーワードで検索して最も上位に表示される内容に記載されているコード片[3]を図 1 に示す。

```
JsonNode blablas = mapper.readTree(parser).get("blablas");
for (JsonNode jsonNode : blablas) {
    String elementId = jsonNode.get("element").asText();
    String value = jsonNode.get("value").asText();
    if (StringUtils.equalsIgnoreCase(elementId, "blabla")) {
        if(value != null && value.equals("YES")){
            // I need to change the node to NO then save it
into the JSON
        }
    }
}
```

図 1 依存ライブラリが明記されていないコード片

このようなコード片は、コードがどのようなライブラリに依存しているか等の情報が記載されていないため、このようなコードを実際にコンパイルし、実行するためには、質問サイトに掲載されている質問の内容等から開発者が依存ライブラリを推測し、入手しなければならない。

今日のソフトウェア開発において、ソフトウェアの全てを記述することはまれであり、高品質な既存のライブラリを用いて開発を行なうことが多い。このため、図 1 の事例のようなコード片に対処することができない先行研究[2]は実用上の大きな問題があった。

本研究では、上記のような典型的なコード片に対処できるよう、先行研究[2]とは別の新たな手段を用いて問題を解決する手法を提案する。また、その効果を既存の研究結果と比較し、有効性を確認する。

† 日本工業大学 Nippon Institute of Technology

3. 提案手法

外部ライブラリに依存したコード片を補完するには、そのコード片が直接依存している外部ライブラリに加えて、その外部ライブラリが依存しているまた別の外部ライブラリが必要である。よって、これを自動的に解決する依存関係解決ツールが必要になる。

先行研究[2]では、ツールを IDE 拡張機能として実装していた。本研究では新たに依存関係解決ツールを開発し、既存のツールから依存関係解決ツールを呼び出す事で外部ライブラリに依存したコード片の補完を行なう。

本手法におけるツールの利用方法を以下に示す。

1. IDE を起動し、ワーキングディレクトリを指定する
2. 新規ファイルを作成し、コード片を保存する
3. コマンド画面を開き、拡張機能の起動コマンドを入力する
4. 補完完了メッセージが表示されるまで待機する

これにより、ユーザーは補完済みのコード片とその動作に必要な外部ライブラリを入手することができる。また、自動補完ツールを実装するに当たってコード片の補完中に発生する主な問題は以下のように分類することができる。

1. コード片が Java として正しくない
2. `class`、`main` 宣言がない
3. `import` 文がない
4. 変数、関数の定義がない
5. 外部ライブラリを `import` している
6. 不明な外部ライブラリに依存している

これらへの対処方法を以下に述べる。

3.1 文法違反チェックとヒューリスティックなスニペット補完

コード片は正しく文法に則ったものであるとは限らない。したがって、コードを動作させたい場合、多くは何らかの補完が必要になる。このため、まずコード片に構文解析を行い、文法違反が含まれるかどうかのチェックを行なう。コード片に文法違反が含まれる場合には、ヒューリスティックな補完を試すことで補完の成功率を向上させる[6]。具体的に本研究で用いている手法は、コード片全体を `class` 宣言で囲うこと、`class` と `main` の宣言で囲うことの 2 種類である。これらの処理により、文法違反が解消された場合、後続の補完処理を行うこととした。

これにより「`class`、`main` 宣言がない」問題への対処が可能となった。

3.2 Quick Fix

現代の IDE は Quick Fix という機能を備えている。これは、IDE が現在発生しているエラーとその修正候補を提示し、開発者は対話式で提示された修正をコードに適用するものである。どのエラーをどのように解決するかは、開発者自らが判断しなければならない。一方、本研究ではツー

ルがルールベースで Quick Fix を連続的に自動で適用を行なう。

これにより「import 文がない」及び「変数、関数の定義がない」問題については Quick Fix を自動的に適用する事で自動補完が可能となった。

3.3 外部ライブラリへの依存関係自動解決機能

前節で述べた Quick Fix は、通常、外部ライブラリのような未知のクラスやメソッドを用いた補完を行うことはできない。本研究ではこのような問題を解決するために、依存関係解決ツールを用いる。

これにより、import 文の定義から必要とする外部ライブラリを自動的に判定し、「外部ライブラリを import している」場合においても補完が可能となった。

3.4 不明な外部ライブラリの予測

例えば、図 1 に示したような「不明な外部ライブラリに依存している」コード片については、エラーの原因になっているクラス名やメソッド名から外部ライブラリを予測できる Tabnine[7]を用いる。

Tabnine は Mishne ら[8]を元にしたコード補完サービスであり、機械学習を用いる事で図 2 のようにメソッド名やクラス名から依存している外部ライブラリを推測することができる。

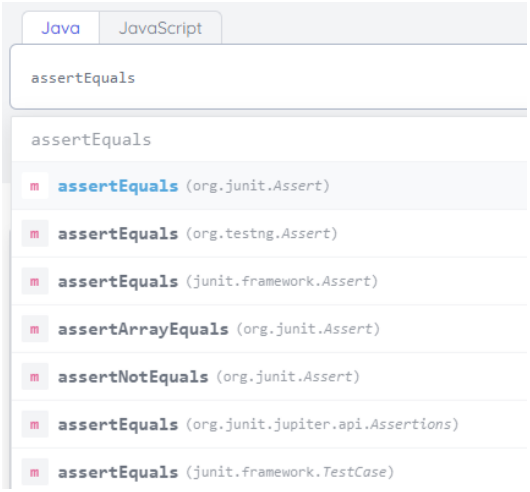

図 2 Tabnine[7]による依存ライブラリの推測例

4. 実現手法

4.1 Visual Studio Code によるコード補完

本ツールは当該コードエディタの拡張機能として実装し、コマンドによって起動するものとした。ツールの利用者は事前に、依存関係解決ツールと IDE 及び、その IDE 上で動作する拡張機能をインストールする必要がある。具体的には、本研究は Java 及び Visual Studio Code[4]を対象としたツールとしたため、ビルドツールとして Maven[5]、IDE として Visual Studio Code、これに加えて Visual Studio Code 用の Java 言語サポート拡張機能と本ツールをインストールする必要がある。Java 言語サポートには vscode-java[9] を用いる。なお、vscode-java は内部で Eclipse[10] の Language Server[11] を用いている。構文解析及び抽象構文木の作成には antlr4ts[12] を用いる。ツール全体の構成を図 3 に示す。

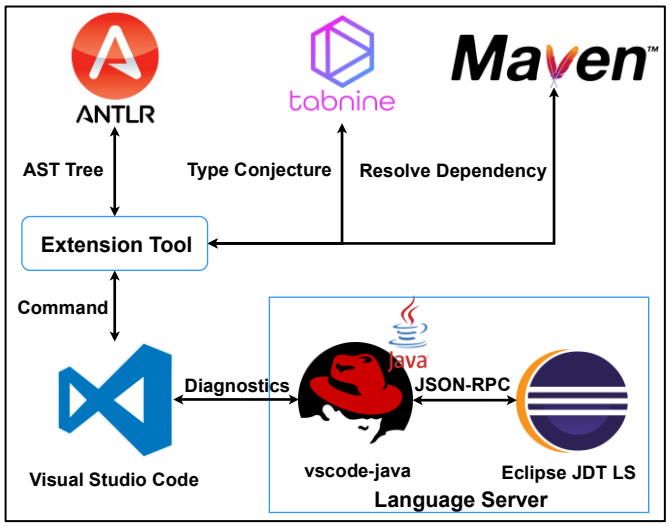

図 3 提案ツールの構成

この拡張機能の動作の流れは以下の通りである。

1. コード片に Java の文法違反が含まれるかどうかを構文解析によってチェックする
2. 文法違反が存在する場合、補完（3.1 節）によってこれが解決できるかをチェックし、解決しない場合は処理を中断する
3. 優先順位が最も高い Quick Fix （3.2 節）を 1 つ適用する
4. 解決可能なエラーが残っている場合、3 に戻る

4.1.1 エラーの処理における優先順位

Quick Fix は前述のとおり、通常、開発者が適用の要否や順序などを判断する。したがって、Quick Fix が提示する補完候補を機械的に適用することは想定されていない。本研究ではこの Quick Fix を自動的適用するために、優先順位を付与し、その順序にしたがって Quick Fix の適用を行なうこととした。本研究で用いる優先順位を以下に示す。

1. import 文の追加で解決可能な型未解決エラー、インスタンス化不能な型のインスタンス化エラー
2. 未初期化又は未定義変数参照エラー、不明な import 先エラー
3. import 文の追加では解決不能な型未解決エラー
4. その他

このような優先順位になった理由としては、エラー同士には依存関係があり、処理順によっては最終的な補完結果が変化してしまうためである。

4.1.2 未初期化変数への初期化子追加

さらに、未初期化変数へのアクセスを起因とするコンパイルエラーへの対処として初期化子「new 型名()」を補完する処理を行なう。ただし、初期化子がインスタンス化不能な型であった場合は null で初期化することとした。このような補完を実施すると、実行時エラー(例えば NullPointerException)を引き起こす可能性がある。しかしながら、後述する理由により、コンパイルエラーを抑制する必要があったため、本手法においてはこのようなエラーが発生することは許容することとした。

このような処理は未初期化変数に初期化子を追加する Quick Fix が存在しなかったため独自に実装を行なった。

4.2 Maven 向けのコード補完

本研究では前述のとおり Java のビルドツールである Maven を用いる。4.1 節までの補完を行ったコードに対して Maven によるビルドを行った場合、主に発生するエラーは「不明な `import` 先エラー」や「`import` 文の追加では解決不能な型未解決エラー」である。

4.2.1 外部ライブラリへの依存関係自動解決

「不明な `import` 先エラー」に対しては、依存するライブラリがプロジェクトのビルドパスに存在しないことがエラーの原因である。これを解決するためには、依存しているライブラリを特定し、ビルドパスに追加する必要がある。

本研究ではエラーを起こしているライブラリ名を Maven Central Repository Search[13]の API を用いて検索し、その候補毎に Maven のプロジェクトをテンポラリディレクトリ内に生成する。Maven の定義ファイル(pom.xml)内には候補となるライブラリを追記し、Maven によるビルドを行うことにより依存関係自動解決することができる。

4.2.2 コード片の型名から依存ライブラリの推定

「`import` 文の追加によって解決不能な型未解決エラー」に対しては、判明しているのはエラーの原因となっている型名のみであり、名前空間や依存ライブラリが不明な場合が多い。このようなコード片に対する補完を実現するために本研究では Tabnine[7] を用いる。これにより、依存ライブラリが自明でないコード片に対する依存関係の解決を行うことができる。

5. 評価と考察

評価には Stack Overflow の Java タグが付けられた約 150 万個の質問の中から使用頻度の高いキーワード 5 種(csv、dictionary、json、list、xml)で検索したものの内 android タグを除いた vote 上位 10 個に絞り、質問とベストアンサー又は最も評価の高い解答に記載されたコード片 81 個を対象とした。

本研究の評価項目は以下の 4 つである。

RQ1. 補完はどの程度成功したか
RQ2. 補完によって記述量は減少するか
RQ3. 減少した記述はどのようなものか
RQ4. キーワードによって減少した記述はどのように異なるのか

5.1 RQ1:補完はどの程度成功したか

補完の成功率を表 1、表 2 に示す。全体として、コンパイル可能数と補完成功数共に成功率はおおよそ 50%程度であった。

ここで、コンパイル可能とは実行時エラーを考慮せず純粋にコード片がコンパイル可能かどうかを表している。また、補完成功とは、コンパイル可能、かつ、コード片が本来の意図に沿った挙動を示すことが確認できたことを表している。ただし、外部リソース（例えばファイルなど）の読み込みが必要なため、実行時エラーが発生する一部のコード片については、本研究が対象としているコード片の補完自体に問題があるわけではないため補完成功したものと見なすこととした。

先行研究[2]ではおよそ 2/3 のコンパイルが可能になっていたのに対しコンパイル可能数が平均 54.3%と低下している。これは外部ライブラリに依存しているコード片の割合が多かったためであると考えられる。

表 1 キーワード毎のコンパイル可能率

キーワード	数	割合
csv	10/15	66.6%
dictionary	8/14	57.1%
json	8/20	40.0%
list	12/21	57.1%
xml	6/11	54.5%
	合計 44/81	**平均 54.3%**

表 2 キーワード毎の補完成功率

キーワード	数	割合
csv	9/15	60.0%
dictionary	7/14	50.0%
json	8/20	40.0%
list	8/21	38.0%
xml	3/11	27.2%
	合計 35/81	**平均 43.2%**

5.2 RQ2: 補完によって記述量は減少するか

前節で明らかになったとおり、本ツールは全てのコード片に対してコード補完を行うことはできない。したがって、本ツールを開発に適用する際には、ツールによる自動補完と開発者の手動による補完を併用する必要が生じる。このような場合を想定し、補完を全て手動で行った場合と、ツールを併用した場合を比較して、ツールが目標達成にどの程度寄与しているかについて調査した。

調査方法は、まずコード片が以下の 3 つの状態を考える。

(1). 初期状態
(2). ツールによる補完後
(3). 補完目標

これらに対して、補完を全て手動で行った場合の記述量(1)→(3)と、ツールを併用した場合の記述量(2)→(3)を求めると、(1) →(2)の部分がツールによって減少した記述量であると定義できる。本研究では diff コマンドによってコードの差分の算出を行った。結果を図 4 に示す。

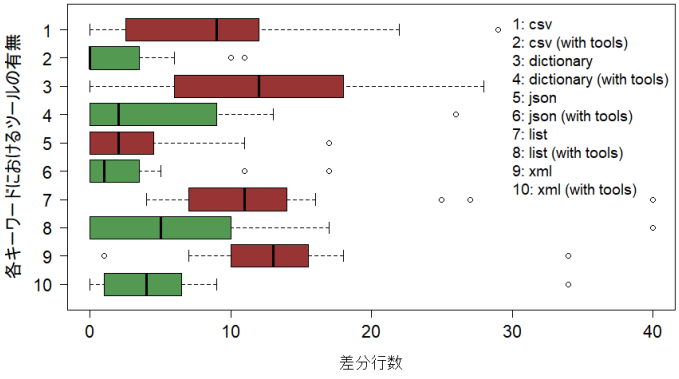

図 4 キーワード毎の差分行数の比較

全てのキーワードにおいて、記述量の減少がみられた。ここで差分行数はツールの有無で変化しないという帰無仮

説 H_0 を立て、対応のある t 検定を行った結果、json 以外は $p<0.01$、json のみ $p<0.05$ となり帰無仮説が棄却された。

5.3 RQ3: 減少した記述はどのようなものか

5.2 節で述べたツールによって減少した記述について、さらに詳細に調査するために減少した記述について手動で分類を行った。その結果を表 3 に示す。

表 3 より、`import` 文の不足や 3.1 節で述べた文法違反チェックとヒューリスティックなスニペット補完に対してツールが有効に作用していることが明らかになった。また、多くのコード片ではこれらの記述が省略されていることが明らかになった。

表 3 キーワード毎の補完結果

分類	csv	dictionary	json	list	xml
`import` 文	10	9	4	15	8
add `class`	1	1	1	2	1
add `class&main`	9	7	1	13	6
変数宣言	3	3	0	7	4
関数宣言	1	1	0	1	0

5.4 RQ4:キーワードによって減少した記述はどのように異なるのか

ここで、キーワード毎によるツールの有効性の違いについて考察を行う。表 3 に示す通り、今回の実験では json に関するコード片については、変数及び関数の宣言を補完したケースが存在しなかった。これは json がデータバインディングに用いられる事が多く、コード片がデータオブジェクトの定義に終始しているパターンが多かったためと考えられる。また、list は変数の宣言の補完が必要である場合が多い。これは頻出の質問がリストの走査や結合であり、操作対象のリストを例示する事が省略される事が多いためである。これらを除くと、本ツールの有効性はキーワードに依存しないことが明らかになった。

6. 関連研究

Yang ら[6]は Stack Overflow におけるコード片のパース及びコンパイル可能率を、そのままの場合と必要に応じてスニペットを用いて補完した場合でそれぞれ調査している。Java のコード片の場合、914,974 個のうちパース可能なものは 3.89%、コンパイル可能なものは 1.00%であった。また、補完後はパース可能なものは 19.24%、コンパイル可能なものは 3.02% まで上昇したと述べている。

Yang らがヒューリスティックな補完に用いたスニペットは `class` 宣言で囲む 1 種類であったが、本研究ではそれに加えて `class` と `main` 宣言で囲む計 2 種類のスニペットを用いている。表 3 の結果が示す通りスニペットによる補完が必要なケースのほとんどは `class` 宣言で囲むものではなく `class` と `main` 宣言で囲むケースであり、この事がパース不能で全く補完処理が行えないパターンの減少に寄与していると考えられる。

Zhang ら[14]は Stack Overflow と GitHub から生成したデータセットを用いて、Stack Overflow に掲載されているコード片からホットスポットを抽象化した状態で提示する Chrome 拡張機能 Example Stack を作成した。これにより、開発者はコードの安全性とロジックのカスタマイズに集中でき、有意に品質が向上すると述べている。

Zhang らの手法はブラウザ上のプルダウンメニューによって手動で加工し、コード片内のロジックを流用するのに特化しているのに対し、本手法では IDE 上でコマンドを入力する事で拡張機能を動作させ自動的に補完することで、コード片に含まれるロジックを改造せずにそのまま動かすことができる。

7. まとめ

本研究では IDE の機能と依存関係解決ツールを組み合わせたコード片の自動補完ツールを開発し、外部ライブラリに依存したコード片を含むコード片の補完を行った。その結果、コード片に対する周辺コード補完を一定程度自動的に行うことができることが明らかになった。

参考文献

[1] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, Edward Aftandilian, “Deepdelta: learning to repair compilation errors,” Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 925–936, Aug. 2019.

[2] 上村勇太, 橋浦弘明, “コード片の再利用に向けた周辺コード補完手法の提案,” ソフトウェア工学の基礎 XXVII 第 27 回ソフトウェア工学の基礎ワークショップ, pp. 145-146, Nov. 2020.

[3] Stack Overflow, “How to modify JsonNode in Java?,” https://stackoverflow.com/questions/30997362/how-to-modify-jsonnode-in-java, (accessed 2022-06-17).

[4] Microsoft, “Visual Studio Code - Code Editing. Redefined,” https://code.visualstudio.com/, (accessed 2022-01-02).

[5] The Apache Software Foundation, “Maven - Welcome to Apache Maven,” https://maven.apache.org/, (accessed 2022-01-02).

[6] Di Yang, Aftab Hussain, Cristina Videira Lopes, “From query to usable code: An analysis of stack overflow code snippets,” 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories, pp. 391–401, May. 2016.

[7] TabNine, Inc., “Code Faster with AI Code Completions − Tabnine,” https://www.tabnine.com/, (accessed 2022-01-02).

[8] Alon Mishne, Sharon Shoham, Eran Yahav, “Typestate-based semantic code search over partial programs,” OOPSLA, pp. 997–1016, Oct. 2012.

[9] Red Hat Developer, “Language support for Java ™ for Visual Studio Code,” https://github.com/redhat-developer/vscode-java/, (accessed 2022-01-02).

[10] Eclipse Foundation, “Eclipse IDE,” http://www.eclipse.org/ide/, (accessed 2022-01-02).

[11] Eclipse Foundation, “Eclipse JDT Language Server,” https://github.com/eclipse/eclipse.jdt.ls, (accessed 2022-01-02).

[12] Tunnel Vision Laboratories, LLC, “Optimized TypeScript target for ANTLR 4, ” https://github.com/tunnelvisionlabs/antlr4ts, (accessed 2022-01-02).

[13] Sonatype, Inc., “Maven Central Repository Search,” https://search.maven.org/, (accessed 2022-01-02).

[14] Tianyi Zhang, Di Yang, Crista Lopes, Miryung Kim, “Analyzing and supporting adaptation of online code examples,” Proc. of the 41st International Conference on Software Engineering (ICSE ’19), pp. 316–327, May. 2019.