Fluent API 生成の現実世界での利用に向けて

中丸 智貴　千葉 滋

Silverchain は “利用者に優しい fluent API” を生成するライブラリ開発者支援ツールである. 既にいくつかの論文において Silverchain に実装された技術の中核は説明しているものの, 現実世界で利用することを想定して取り組んだ様々な工夫・対策についてはほとんど議論してこなかった. そこで本論文では, Silverchain 開発の背景と中核技術を簡単に説明した上で, 取り組んだ工夫・対策を紹介する.

1 はじめに

SQL クエリを以下のように記述できる Java ライブラリを作ろう:

```
// SELECT name FROM users WHERE id = 1
Result r = new SQL()
    .select("name").from("users")
    .where("id = 1").execute();
```

このようにメソッド呼び出しを連鎖させる記法はメソッドチェインスタイルと呼ばれ, 近年広く利用されているプログラミングスタイルである [5]. メソッドチェインスタイルでの利用を想定してインターフェイスが設計されているライブラリは fluent interface [1] と呼ばれている.

上で述べた SQL ライブラリの最も単純な作り方は, クラスを一つ定義し, 全てのメソッドをそのクラス内に定義する方法である. 具体的には

```
class SQL {
  SQL() { ... }
  SQL select(String columns) {
    ... ; return this;  }
  SQL from(String table) {
    ... ; return this;  }
  SQL where(String expression) {
    ... ; return this;  }
  Result execute() {
    ...; return ... ;  }
}
```

という単一の Java のクラス定義によって実現できる.

この単純な作り方で, 確かに冒頭で挙げたようなメソッドチェイン形式での SQL クエリ記述が行える. しかし, 以下のような SQL として成り立っていない “不正な” メソッドチェインの記述も許してしまうという問題がある:

```
// Missing "from(...)"
new SQL().select("name")
    .where("id = 1").execute();
```

このような不正なメソッドチェインは, 次に連鎖させることができるメソッドに応じて, 各メソッドが返却する型を変えれば防ぐことができる. 例の SQL ライブラリの場合, 具体的には

```
class SQL {
  SQL() { ... }
  SQL1 select() { ... }
}
class SQL1 {
  SQL2 from() { ... }
}
class SQL2 {
  SQL3 where() { ... }
  Result execute() { ... }
}
class SQL3 {
  Result execute() { ... }
}
```

Tomoki Nakamaru, The University of Tokyo.
Shigeru Chiba, The University of Tokyo.

図 1　メソッド補完候補の比較

という 4 つのクラス定義を使って構成すればよい．このように複数クラスを使って作るすることで, 誤ったクエリ記述が型エラーを引き起こすようになり, 不正なクエリ記述を実行前に発見することができる:

```
new SQL()
  .select("name") // Returns "SQL1"
  .where("id = 1");
  // "SQL1" does not have "where()"
```

複数クラスを使った作り方は, 統合開発環境のメソッド補完機能との相性も良い．複数クラスを使った作り方の場合, 次に呼び出すことが許されているメソッドのみが候補に現れるようになる．逆に単純な作り方の場合, 次に呼び出しても実行時エラーとなることが明らかなメソッドまで候補に入ってしまっている．図 1 はこの補完候補の違いを示しているものである．

上述したように, 複数クラスを使った作り方は確かに“ライブラリ利用者に優しい” 作りである．しかしライブラリ開発者にとってはどうだろうか? ライブラリ開発者は多くのクラスを定義し, どのメソッドがどの順で呼び出せるのかに注意しながらメソッドを定義していく必要がある．冒頭の例のように SELECT 文を書くだけであればまだ現実的な規模の手間で済むが, INSERT/UPDATE/DELETE 文も対応しようとすればライブラリの作成はさらに困難になる．つまり “ライブラリ利用者に優しい” 作り方は “開発者にとって全く優しくない” 作り方なのである．

本論文で紹介する Silverchain はこの開発者視点の問題を解決するためのツールである．Silverchain は「どのようなメソッドチェインの記述が許されているのか」を示すルール記述を入力に受け取る．図 2 は,

```
SQL:
  select(String columns)
  from(String table)
  where(String expression)?
  execute() Result;
```

図 2　SQL ライブラリのルール記述

例の SQL のライブラリを生成するための入力ルール記述である．入力を受け取った Silverchain は, ルールに従っているメソッドチェインのみが書けるようなライブラリのコードを生成する．

以下ではまず Silverchain に実装されているコード生成処理を簡単に説明する (第 2 節). 次に, 現実世界で Silverchain を利用することを想定して取り組んだ工夫・対策について説明する．(第 3 節). そして最後に, 今後の課題と結論を述べる (第 4 節).

2　Silverchain

まず Silverchain が入力からクラス生成を行うまでの処理の概要を解説する．具体的にどのようなクラス定義が生成されるのか等の詳細は, 第 3 節で述べるため, 本節では省略している．

まず Silverchain は入力されたルール記述から決定性有限状態オートマトン (Deterministic finite-state automaton, DFA) を構築する．図 2 が入力の場合, 図 3 のような DFA を構築する．ここで行う DFA 構築は, メソッドの定義 (メソッド名, 引数型, 引数名) やクラス名を一つの文字とみなせば, 単なる正規表現からの DFA 構築である．図中の状態番号は次節での

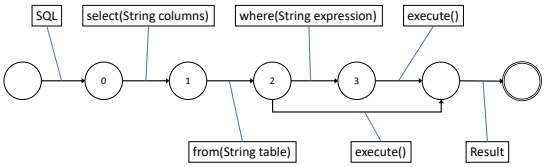

図 3　図 2 が入力から構築される DFA

説明のための番号である.

次に各状態をクラス定義へ, 各遷移をメソッド定義へと変換することで, 構築された DFA を Java コードへ変換する. ただし例外的に, 開始状態, 終了状態, クラス名のラベルを持つ遷移の始状態はクラス定義へ変換しない. つまり図 3 の DFA からは, 4 つのクラス定義が生成される. また, クラス名でラベル付された (クラス名を消費する) 遷移もメソッド定義へと変換しない. (クラス名ラベルは, そのラベルが付いた遷移の始状態/終状態から生成されるクラスのクラス名として利用される.)

処理概要として上の 2 段落で解説した DFA 構築を行うという技術は Silverchain の開発において発明されたものではない. Silverchain の新規性は型パラメータを取り扱うことができるようにした点にある. しかし本論文の趣旨としてはその新規性のある点は重要でないため, 詳細は文献 [2] に譲る. (文献 [2] ではツール名が異なるが, 実装内容は本稿で紹介する Silverchain と同様である. また文献 [3] と文献 [4] では Silverchain という名前ツールを紹介しているが, 手法が古く, 本稿で紹介する最新の Silverchain とは実装内容が異なるので注意されたい.)

3　現実世界での利用に向けた取り組み

3.1　コード生成

前節では「DFA の状態に対応する Java クラス定義を生成する」と述べたが, 実際には Silverchain は様々なクラス/インターフェイスを生成する. 表 1 は生成される Java ファイルの一覧である. 各 Java ファイル内では 1 つの型定義のみを行っており, パッケージ名はパスに対応する形で指定されている.

3.1.1　様々なファイルを生成する意義

表 1 に示したような様々なファイルを生成するのは, 単に Java の習慣に従ったからではない. 機械生成されたライブラリ API のためのコードに対し, 人間がライブラリの挙動 (セマンティクス) を可能な限り手軽にできるようにするためである.

まず Silverchain が状態に対応するインターフェイスのみを生成, つまりライブラリの挙動定義は完全に人間に委ねるとしよう. その場合, 実際にライブラリを使えるようにするにはインターフェイスに対応するクラスを状態の数だけ定義しなければならない. 特に `execute()` のように, 同じ挙動であるべきメソッドがルールの都合上複数のクラスに現れていたりする場合には, 状態数分のクラス定義はかなり退屈な作業である. この退屈な作業を避けるためには, クラス定義も生成しなければならない.

前段落での考察を踏まえ, Silverchain が状態に関するインターフェイスとクラスを生成し, 挙動定義用のインターフェイス定義 (ISQLAction.java) は生成しないとする. この場合, ライブラリの挙動定義のためには生成されたコードを人間が編集しなければならない. 生成されたコードを人間が編集することは不可能ではないが, ルール変更に伴い再生成を行う場合に非常に不便である. 最悪の場合, 記述した挙動定義を全て失ってしまいかねない. そのため挙動定義用のインターフェイス定義を用意し, 挙動定義を生成コードの外に記述できるようにする必要がある †1.

以上のように考えていくと, 人間がライブラリの挙動を手軽に定義できるようにするには, 状態に対応するクラス/インターフェイスと挙動定義用のインターフェイス定義を生成する必要がある. ここまでの考察ではクラスとインターフェイスは分離しないという選択肢もあり得るように思える. しかし, 後述するクラス名の工夫と「ライブラリ利用者に不必要なクラスはアクセス不可能にする」という方針のためには, クラスとインターフェイスは分離しておく必要がある.

†1　そもそも図 2 の定義中に挙動定義 (Java コード) も合わせて記述できるようにすれば, 挙動定義を失うことはない. しかしその場合, Java コードの記述を統合開発環境の支援なしで行わなければならず, 実用上不便である.

表 1　生成されるファイル一覧

ファイル名	型の種類	概要
ISQL.java	package-private interface	状態 0 に対応するインターフェイス
state1/SQL.java	public interface	状態 1 に対応するインターフェイス
state2/SQL.java	public interface	状態 2 に対応するインターフェイス
state3/SQL.java	public interface	状態 3 に対応するインターフェイス
SQL0.java	package-private class	ISQL を実装したクラス
SQL1.java	package-private class	state1.SQL を実装したクラス
SQL2.java	package-private class	state2.SQL を実装したクラス
SQL3.java	package-private class	state3.SQL を実装したクラス
ISQLAction.java	package-private interface	メソッドの挙動を定義するためのインターフェイス

(当然ながら, ソフトウェア構成の原則の一つである「インターフェイスと実装の分離」に従うという意図もある.)

3.1.2　インターフェイス名に一貫性がない理由

表 1 を見るとわかるように, インターフェイス名には一貫性がなく, 状態 0 に対応するものだけ特殊な命名がされている. これは, メソッドチェイン記述のためのエントリポイントになるクラス `SQL` を, 人間が定義することを想定した命名をしているからである.

ライブラリの作成者は, 自身が手書きで用意した挙動定義のクラス (`ISQLAction` を実装したクラス) を生成されたコードと結び付ける必要がある. それらを結び付ける場所として `SQL` というエントリポイントになるクラスを定義することになる. しかし, もし SQL.java 既にが生成されてしまっていれば, 別の名前を付けなければならない. 代わりに `state0.SQL` という名前の状態 0 に対応するクラスの生成する選択肢はあり得るが, この場合後述する「ライブラリ利用者に不必要なクラスはアクセス不可能にする」という方針を維持できない.

状態 1 以降のファイル配置は必須ではないが, ここにもライブラリ利用者を意識した工夫がある. 状態 1 以降は state1.SQL.java ではなく, ISQL1.java というインターフェイスに変換しても良い. しかしながらその場合, メソッド補完機能で表示の中に SQL1 や SQL2 という好ましくない名前が現れることになる. このような不快感を軽減するために, 状態番号をクラス名ではなくパッケージ名に変換するという工夫を行った.

3.1.3　アクセス修飾子に一貫性がない理由

生成される型のアクセス修飾子は「ライブラリ利用者に不必要なクラスはアクセス不可能にする」という方針で決めている. これは単なるコードへの美学的な要素もあるが, 生成されるコードは名前が似通ったものが多く, 不必要なアクセスが可能になるコード補完の選択肢が増えてしまい, その中から本当に必要なコードを見つけるのがそう簡単ではなかったという理由がある.

各型定義の役割から明らかであるように, public であるべき (ライブラリ利用者が自由にアクセスできるべき) なのは状態 1 以降に対応するインターフェイスのみである. 他にもエントリポイントとなるクラスは public でなければならないが, 先述したようにこれは挙動定義を記述する人間が作るものであるため, コード生成では気にする必要はない.

第 3.1.1 節で, クラスとインターフェイスは分離しなければならないと述べた. これは上述のアクセスは最大限に制限する方針と第 3.1.2 の末尾で述べた命名の工夫が関連している. クラスとインターフェイスの分割はせず, 単にクラスのみを生成するとしよう. その場合, 命名の工夫の都合から, 各状態に対応するクラスは異なるパッケージに配置される. 各状態クラスは次の状態クラスのインスタンスを構築する必要があるため, 各状態クラスのコンストラクタは public

にしなければならない. しかし public にしてしまうと, ライブラリ利用者がエントリポイントとして用意したクラス以外のインスタンス作成も可能になってしまうという問題が発生する. 結果的に, 様々な工夫の整合性を取るにはクラスとインターフェイスの分割が必須であると言える.

3.2 利用を促すための取り組み

Silverchain の役割について, 冒頭でおよそ 2 ページに渡って説明した (第 1 節). しかし現実問題として, そのような長い説明文を読まなければ「何のためのツールであるか」さえ分からないツールはなかなか利用されない. 以下では, この状況を打開するために行った対策について述べる.

3.2.1 デモ動画の作成

Silverchain を利用することでどのような恩恵があるのかを分かりやすくするため, Silverchain で生成したライブラリを使ってデモ動画 †2 の作成を行った. デモ動画では特に統合開発環境との親和性を強調することで, 他の静的検査ツールとの差を強調した.

動画では, 簡単なメロディ作曲のためのライブラリ MelodyChain を使ってデモを行っている. MelodyChain は音楽知識が一切ない人でも, メソッド補完機能が示す候補から音を選んでメソッドチェインを喜寿していくだけで “それなりの” メロディが作れるようになっている.

3.2.2 チュートリアルの作成

デモ動画では「Silverchain で何ができるか」を説明したが, 興味を持ったとしても, 試しに一つライブラリを作ってみるというのは敷居が高い. 試しにライブラリを作るとは言っても, 何を作るべきかわからず, 利用しないまま終わってしまうと考えられる.

そこで何もない状態からデモ動画で紹介した MelodyChain の作成を行うチュートリアル †3 の作成を行った. チュートリアルではルールの記述から各メソッドの挙動の定義まで行い, 最終的には動画と同じようにメロディ作曲が行えるようになっている.

3.2.3 Silverchain の動作環境

Silverchain は Java を用いて作成されている. そのため基本的には Java の実行ができる環境でしか動作しない. しかしこれでは環境の制約上 Silverchain を試すことができないプログラマもいる.

そこで Silverchain では Docker イメージ †4 を DockerHub にて配布している †5. これにより, Docker を利用可能な環境であれば以下のコマンド一つで実行ができるようになる:

```
docker run -v $(pwd):/workdir --rm \
  -it tomokinakamaru/silverchain \
  -i <input-file> \
  -o <output-directory>
```

これに加えて Graal †6 を用いた実行可能なバイナリの生成も行えるようにしている. しかし本稿執筆の段階ではバイナリ配布を行っていないため, Java の利用できる環境で一度コンパイルを行わなければならない. 今後バイナリの配布も行い, さらに利用を開始しやすくする予定である.

4 まとめ

本論文では, 現実世界での利用を想定して Silverchain に施した工夫と対策について解説した. しかし現在も取り組むべきだが実装に至っていない課題は複数あるため, 今後もさらなる開発を進める予定である. また, ここで述べた内容と今後の開発状況をソフトウェア論文として改めてまとめることを目指して, 関連ツールとの比較も進めていく必要がある.

参考文献

[1] Fowler, M.: FluentInterface, December 2005.

[2] Nakamaru, T. and Chiba, S.: Generating a Generic Fluent API in Java, *The Art, Science, and Engineering of Programming*, Vol. 4, No. 3(2020).

[3] Nakamaru, T., Ichikawa, K., Yamazaki, T., and Chiba, S.: Silverchain: A Fluent API Generator, *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017, pp. 199–211.

[4] Nakamaru, T., Ichikawa, K., Yamazaki, T., and

†2 https://youtu.be/3fOn8cbhFZU

†3 https://github.com/tomokinakamaru/silverchain/blob/master/doc/tutorial.md

†4 https://www.docker.com

†5 https://hub.docker.com/r/tomokinakamaru/silverchain

†6 https://github.com/oracle/graal

Chiba, S.: Generating fluent embedded domain-specific languages with subchaining, *Journal of Computer Languages*, Vol. 50(2019), pp. 70 – 83.

[5] Nakamaru, T., Matsunaga, T., Yamazaki, T., Akiyama, S., and Chiba, S.: An Empirical Study of Method Chaining in Java, *Proceedings of the 17th IEEE/ACM International Conference on Mining Software Repositories*, GPCE 2017, 2020, pp. 93–102.