LL(1) 文法から flat-chaining および sub-chaining の fluent interface を自動生成する DSL の提案にむけて

山隈 由衣　山崎 徹郎　千葉 滋

本論文では，LL(1) 文法から flat-chaining および sub-chaining の fluent interface を生成するツールの提案をおこなう．fluent interface は，method chain と呼ばれるメソッド呼び出しの連鎖からなるインターフェースである．fluent interface の表現方法には，flat-chaining と sub-chaining の 2 種類がある．本論文は，Python 上の埋め込み DSL を使って開発者が BNF 風の文法規則を記述すると Java で書かれた fluent interface の雛形を生成する生成器の設計とその生成アルゴリズムについて述べる．提案する生成アルゴリズムは，Greibach 標準形の LL(1) 文法から flat-chaining の fluent interface を生成する手法を，sub-chaining に拡張したものである．ε ルールを除去できる LL(1) 文法からであれば，flat-chaining および sub-chaining の fluent interface の生成が可能になる．

1　はじめに

Fluent interface は，メソッド呼び出しの連鎖からなるインターフェースである．近年，高機能なライブラリの利用インタフェースとして採用されることが増えている [1] [2]．ホスト言語の型システムを利用することでメソッドの並び順の間違いを静的に検出することも可能だが，インタフェースの実装が複雑になる．このため，複雑な実装を支援するために，インタフェースの実装のひな形を出力するコード生成器の開発が重要である．

Fluent interface の表現方法には，flat-chaining と sub-chaining の 2 種類がある．Flat-chaining の fluent interface 生成器を，sub-chaining の生成器に応用する手法が研究されている．本論文では，LL(1) のより大きい部分集合から sub-と flat-の fluent interface を生成するツールを提案する．このツールを用いると Graibach 標準形 (GNF) で書ける LL(1) から flat-chaining の静的型検査付き fluent interface を生成する手法があれば，ε ルールを除去できる LL(1) 文法から flat-chaining および sub-chaining の fluent interface の生成が可能になる．

2　Subchaining に対応した fluent interface 生成器

Fluent interface は，method chain と呼ばれるメソッド呼び出しの連鎖からなるインターフェースである．method chaining の各メソッドはオブジェクトを返し，そのオブジェクトから次のメソッドが呼び出される．fluent interface は埋め込み型領域特化言語 (embedded domain specific languages, EDSLs) のようなライブラリの設計に有用である．例えば，JOOQ は SQL 文を表現する Java のコードを生成する．以下のような SQL クエリに対して，

```
SELECT TITLE
FROM BOOK
WHERE BOOK.PUBLISHED_IN = 2011
ORDER BY BOOK.TITLE
```

以下のクエリが Java で生成される．

```
create.select(BOOK.TITLE)
        .from(BOOK)
        .where(BOOK.PUBLISHED_IN.eq(2011))
        .orderBy(BOOK.TITLE);
```

Toward a proposal for a DSL that automatically generates fluent interfaces for flat-chaining and sub-chaining from LL(1) grammar.

Yui Yamaguma, Testuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

Fluent interface の利用者は誤ったメソッドの並びを書いてしまうことがあるが，そのような誤った並びは文法エラーと見なすことができる．この fluent interface の文法エラーはホスト言語の静的な型エラーとしてコンパイル時にエラーとすることができる．

しかし，このような fluent inferaface の文法エラーを型検査で判別するようなインターフェースを手作業で開発するのは難しい．なぜなら，ホスト言語上でたくさんの型を定義する必要があるからである．静的型検査を行う仕組みは，文法に対応するプッシュダウンオートマトンのスタックを表現する型を定義して構文検査を行う．文法の規則が多いと，使うスタックの種類も多くなる．fluent interface の表現方法には，flat-chaining と sub-chaining の 2 種類がある．flat-chaining は，以下のように一つの method chaining からなる．

```
i(9).add().i(2).multiple().i(3)
```

flat-chaining は，短いあるいは単純な連鎖の表現に有用である．sub-chaining は複数の method chaining で表現し，ある連鎖を他の連鎖に引数として渡す．sub-chaining では，以下のように，あるメソッド呼び出し連鎖を他の連鎖に引数として渡す．

```
add(i(9), i(2).multiple().i(3))
```

sub-chaining は長いあるいは複雑な連鎖の表現に有用である．

文法エラー検出機能つきの fluent interface の開発を支援するために自動生成器が開発されている．しかしながら，flat-chainging と sub-chaining の両方に生成する自動生成器で，対応可能な fluent interface の文法クラスが明らかになっているものはない．

Yamazaki らによって，LL(1) から flat-chaining のインターフェースを生成するアルゴリズムがあれば，LL(1) から flat-chaining と sub-chaining の両方を生成するアルゴリズムが提案されている [3]．しかし LL(1) から flat-chaining のインターフェースを生成するアルゴリズムは知られていない．知られているのは GNF(グライバッハ標準形，Graibach Normal Form) [4] の LL(1) 文法に対応する flat-chaining のインターフェースを生成するアルゴリズム [5] だけである．GNF は全ての生成規則の右辺が終端記号から始まる文法の形である．

LL(1) 文法の中には等価な GNF が存在しないものがある [6] ので，GNF の LL(1) 文法から flat-chaining のインタフェースを生成できても，任意の LL(1) から falt-chaining のインタフェースを生成できるとは限らない．例えば，言語 $a^n(b^kd+b+cc)^n|n >= 1(k >= 1)$ を認識する以下の文法は，ε ルールを除去できない [6]．

S → a D A

D → a D A | ε A → cc |b B

B → b^{k-1}d | ε

GNF の文法は ε ルールを含まないことから，この文法は等価な GNF が存在しない．

3 GreenChain の提案

本論文では，ε ルールを除去可能な LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの静的検査付きで生成するツール GreenChain を提案する．GreenChain は Python 上の EDSL(embedded domain specific language) である．本論文は GreenChain のために fluent interface を自動生成するための新しいアルゴリズムを示す．このアルゴリズムは，ε ルールを除去可能という制限があるものの LL(1) 文法から flat-chaining および sub-chaining の fluent interface を文法エラーの検出機能つきで生成する現実的なアルゴリズムである．

3.1 利用方法

GreenChain の EDSL は BNF 風の文法を記述するための言語で，GreenChain はこの記述から Java で書かれた fluent interface の雛形を生成する．例えば次のような加算と乗算の前置記法を表現する BNF の文法

終端記号：i, add, multiple

非終端記号：E, T, F

生成規則：

E → add T E |T

T → multiple F t |F

F → i

```
fi.Lang
.rule().nt('E').arrow().t('add').nt('T').nt('E')
.rule().nt('E').arrow().nt('T')
.rule().nt('T').arrow().t('multiple').nt('F').nt('T')
.rule().nt('T').arrow().nt('F')
.rule().nt('F').arrow().t('i')
.terminal('i').arg_type('Integer').action_type('boolean')
.start_from('E').end()
```

図 1　Python 上での入力文法の記述

```
static boolean action_i(Integer arg0) {
    Data.numbers.add(arg0);
    return true;
}

static void action_multiple() {
    Data.operators.add("multiple");
    return;
}

static void action_add() {
    Data.operators.add("add");
    return;
}
```

図 2　終端記号が受け取った引数を処理するメソッド

開始記号：E

を，GreenChain では Python で図 1 のように記述する．

各生成規則は `rule` から始まり，`nt` は非終端記号，`t` は終端記号を表す．各終端記号に対応するメソッドの引数の型は `arg_type` で指定する．受け取った引数をどのように処理するかは，GreenChain が生成したクラスの中にある図 2 のようなメソッドの雛形を修正して記述する．このメソッドの戻り値の型は `action_type` で指定した型になる．`arg_type` を指定しない場合は引数はなし，`action_type` を指定しない場合は戻り値の型は void になる．開始記号は `start_from` で与える．

3.2　生成アルゴリズム

GreenChain は与えられた LL(1) 文法からまず ε ルールを除去し，これを GNF の LL(1) に変換し，sub-chaining と flat-chaining の fluent interface を生成する．以下では Python の EDSL で記述した文法を変換して得られた GNF の LL(1) 文法を G = (T, N, δ, S) とする．T は終端記号，N は非終端記号，δ は生成規則，S は開始記号である．

3.2.1　ε ルールの除去

L(A)⊇ ε となる記号を nullable な記号と呼ぶ [6]．ここで L(A) は，A を展開して生成される言語である．nullable な記号とその後ろの non-nullable な記号を組み合わせて，新しい記号を作る．例えば以下の規則に対して，

S → a B C

B → b D | ε

C → c D |e

D → d

nullable な記号 B とその後ろに出てくる記号 C を組み合わせた新しい記号 B' を以下のように定義する．

B' → b D C |C

記号 B が含まれる規則 S → a B C を以下の規則に置き換える．

S → a B'

これにより ε ルールが除去できる．ただし，規則の右辺が nullable な記号で終わる場合には適用できない．

3.2.2　ε ルールなしの LL(1) 文法から GNF の LL(1) への変換

この変換では規則の右辺の始めの記号を，終端記号になるまで展開する．$\forall A \in N, \forall X \in \delta$ (A) について，X を $aY_1...Y_n (n \geq 1, Y_i \in N \cup T (1 \leq i \leq \mathrm{n}))$ とする．a ∈ N の場合，δ（A）から X を削除して，以下の規則を δ (A) に追加する．

$\forall Z \in \delta$ (a).

$ZY_1...Y_n$

さらに X を Z に置き換えて，a $\notin$ N となるまで繰り返す．左再帰があるとこの繰り返しは停止しないが，LL(1) は左再帰を許さないので，繰り返しは停止する．このようにして ε ルールを含まない全ての LL(1) 文法を GNF に変換できる．

3.2.3 GNF の LL(1) から flat-chaining と sub-chaining の両方の fluent interface の生成

ここでは Yamazaki アルゴリズム [3] を応用したアルゴリズムで生成をおこなう．アルゴリズムは，GNF の LL(1) から flat-chaining の fluent interface を生成するツールが存在することを仮定する．まず 3.2.2 の変換で得られた文法 G から sub-chaining の表現を埋め込んだ次のような文法 G ’ = (T’, N, δ’, S) を得る．

T’ = T $\cup$ $\{A_{subchain}$ |A $\in$ N$\}$

δ’(A) = $\{A_{subchain}\}$ $\cup$ δ (A)

$A_{subchain}$ は，A に対応する sub-chaining を表現する記号である．次に GNF の LL(1) から flat-chaining の fluent interface 生成器で，G’ から flat-chaining の fluent interface を生成する．これは G に対応する flat-chaining と sub-chaining の両方を表現する fluent interface である．

上のアルゴリズムは Yamazaki のアルゴリズムと同じであるが，Yamazaki アルゴリズムは任意の LL(1) から flat-chaining の fluent interface を生成するツールの存在を仮定している．本論文のアルゴリズムは，GNF の LL(1) から flat-chaining の fluent interface を生成するツールが存在することを仮定するので，その正しさを示すためには別途証明が必要である．

G が空列を生成する非終端記号を含まない GNF の LL(1) なら，G’ も GNF の LL(1) になる．G = (T, N, δ, S) が GNF の LL(1) 文法の時，以下の条件が満たされる．

1. $\forall A \in N, \forall X, Y \in$ δ (A).
   X $\neq$ Y $\rightarrow$ $first_G(X) \cap first_G(Y) = \phi$
2. $\forall$ A $\in$ N, δ (A) の規則が以下のいずれかになる．
   A $\rightarrow$ ε (A = S のときのみ)
   A $\rightarrow$ a
   A $\rightarrow$ $aY_1...Y_n(n >= 1, a \in T, Y_i \in N(1 <= i <= n))$
3. 2. の規則の右辺に開始記号が出現しない．
   $Y_i \neq S$

$first_G(A)$ は文法 G における A を展開して得られる終端記号列の最初になり得る記号の集合である．

(i) G が 1. を満たすとき，G が空列を生成する非終端記号を含む場合を除いて G’ も 1. を満たすことが示されている [3].

(ii) G が 2. を満たすとき，
$A_{subchain}$ が終端記号であることから規則 A $\rightarrow$ $A_{subchain}$ は 2. を満たす．前提から δ (A) の規則も 2. を満たすから，δ’(A) = $\{A_{subchain}\}$ $\cup$ δ (A) の規則も 2. を満たす．
すなわち G’ も 2 を満たす．

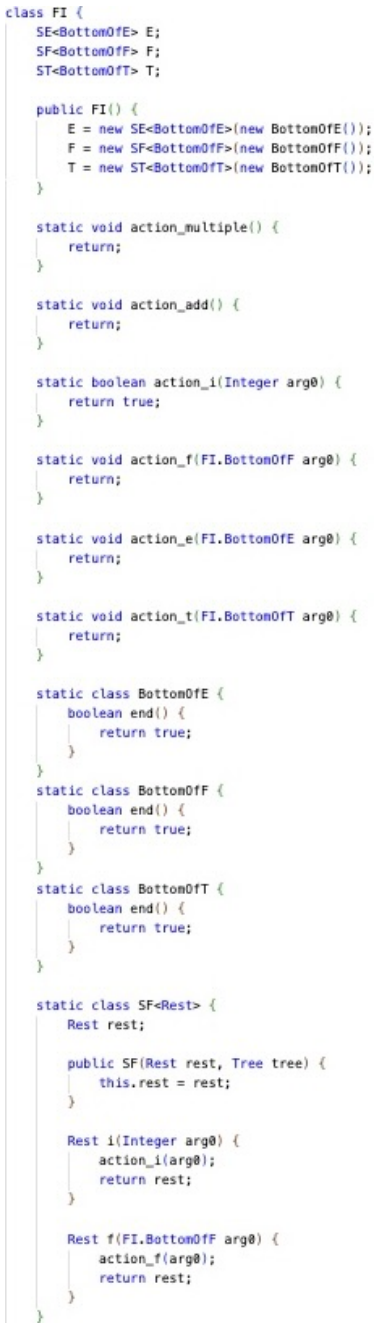
(iii) G が 3. を満たすとき，
$A_{subchain}$ は開始記号ではない．前提から δ (A) の規則も右辺に開始記号を含まないから，δ’(A) = $\{A_{subchain}\}$ $\cup$ δ (A) の右辺に開始記号が出現しない．
すなわち G’ も 3. を満たす．

以上のことから入力文法 G が空列を生成する非終端記号を含まない GNF の LL(1) なら，変換後の文法 G’ も GNF の LL(1) になる．

4 利用例

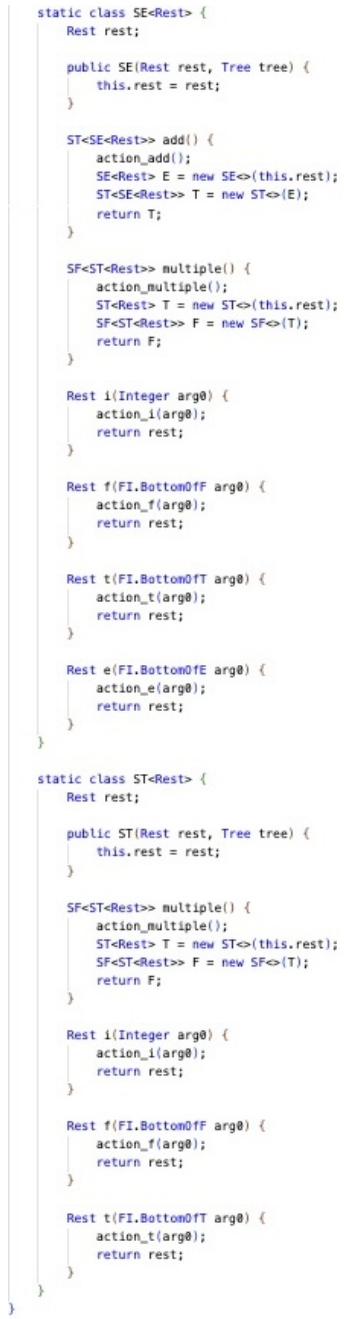
3.1 節で示した加算と乗算の前置記法を表現する文法から生成される fluent interface のプログラムを図 3, 4 に示す．Fluent interface のクラスは，各スタックを表すクラス，初期状態を表すインスタンス，各メソッドが呼ばれた時の動作を利用者が書き込むためのメソッドを持つ．各スタックを表すクラス (`SE` 等) には，スタックの状態を表す変数 rest と，そのスタックがトップにある時に呼び出し可能なメソッドが含まれている．これにより図 5 のように，method chaining の入力途中では，次に選択できるメソッドが表示される．

また図 6 のような SQL を表す入力文法から，図 7 のような sub-chaining の表現を含む fluent inter-



```
class FI {
    SE<BottomOfE> E;
    SF<BottomOfF> F;
    ST<BottomOfT> T;

    public FI() {
        E = new SE<BottomOfE>(new BottomOfE());
        F = new SF<BottomOfF>(new BottomOfF());
        T = new ST<BottomOfT>(new BottomOfT());
    }

    static void action_multiple() {
        return;
    }

    static void action_add() {
        return;
    }

    static boolean action_i(Integer arg0) {
        return true;
    }

    static void action_f(FI.BottomOfF arg0) {
        return;
    }

    static void action_e(FI.BottomOfE arg0) {
        return;
    }

    static void action_t(FI.BottomOfT arg0) {
        return;
    }

    static class BottomOfE {
        boolean end() {
            return true;
        }
    }
    static class BottomOfF {
        boolean end() {
            return true;
        }
    }
    static class BottomOfT {
        boolean end() {
            return true;
        }
    }

    static class SF<Rest> {
        Rest rest;

        public SF(Rest rest, Tree tree) {
            this.rest = rest;
        }

        Rest i(Integer arg0) {
            action_i(arg0);
            return rest;
        }

        Rest f(FI.BottomOfF arg0) {
            action_f(arg0);
            return rest;
        }
    }
```

図 3　生成された fluent interface



```
    static class SE<Rest> {
        Rest rest;

        public SE(Rest rest, Tree tree) {
            this.rest = rest;
        }

        ST<SE<Rest>> add() {
            action_add();
            SE<Rest> E = new SE<>(this.rest);
            ST<SE<Rest>> T = new ST<>(E);
            return T;
        }

        SF<ST<Rest>> multiple() {
            action_multiple();
            ST<Rest> T = new ST<>(this.rest);
            SF<ST<Rest>> F = new SF<>(T);
            return F;
        }

        Rest i(Integer arg0) {
            action_i(arg0);
            return rest;
        }

        Rest f(FI.BottomOfF arg0) {
            action_f(arg0);
            return rest;
        }

        Rest t(FI.BottomOfT arg0) {
            action_t(arg0);
            return rest;
        }

        Rest e(FI.BottomOfE arg0) {
            action_e(arg0);
            return rest;
        }
    }

    static class ST<Rest> {
        Rest rest;

        public ST(Rest rest, Tree tree) {
            this.rest = rest;
        }

        SF<ST<Rest>> multiple() {
            action_multiple();
            ST<Rest> T = new ST<>(this.rest);
            SF<ST<Rest>> F = new SF<>(T);
            return F;
        }

        Rest i(Integer arg0) {
            action_i(arg0);
            return rest;
        }

        Rest f(FI.BottomOfF arg0) {
            action_f(arg0);
            return rest;
        }

        Rest t(FI.BottomOfT arg0) {
            action_t(arg0);
            return rest;
        }
    }
}
```

図 4　生成された fluent interface(続)

face が生成される．最後の Main クラスは生成された fluent interface の利用例である．whereClause メソッドの引数が subchain になっている．この fluent interface のクラス `FI` のインスタンスは外側の chain と subchain の初期状態に対応する 2 つのインスタンスをフィールドの値に持つ．Subchain を受け取るメソッドの引数の型は，引数となる subchain におけるスタックの底に対応する型である．非終端記号 `SelectQuery`，`WhereClause` に対応する sub-chaining 表現のための終端記号に対応するメ

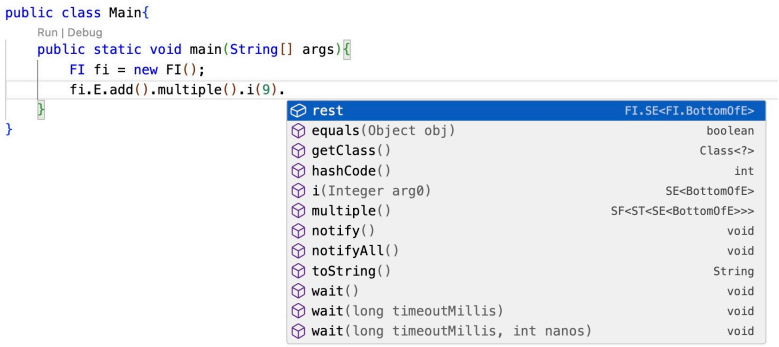


図 5　補完候補のメソッドの表示

```
fi.Lang
.rule().nt('SelectQuery').arrow().t('selectFrom').nt('WhereClause')
.rule().nt('WhereClause').arrow().t('where')
.terminal('selectFrom').arg_type('String').arg_type('String').action_type('void')
.terminal('where').arg_type('String').action_type('void')
.start_from('SelectQuery').end()
```

図 6　SQL を表現する入力文法の記述

ソッド `selectQuery` と `whereClause` が追加される．`selectQuery` が $SelectQuery_{subchain}$，`whereClause` が $WhereClause_{subchain}$ に対応する．

5　関連研究

Sub-chaining を表現できる fluent interface 生成器 Silverchain [7] が提案されている．Silverchain は LL(1) の部分集合から sub-chaining と flat-chaining を生成できるが，GreenChain とは異なり，部分集合が不明である．また LR(1) など大きい文法クラスに対応する fluent interface 生成器が提案されている [8] が，sub-chaining には対応していない．

LL(1) 文法の flat-chaining の fluent interface 生成器が存在すると仮定して，それを利用して sub-chaining の表現も可能にする手法が提案されている [3]．しかし該当する flat-chaining の fluent interface は我々の知るかぎり存在しない．本論文は GNF で書ける LL(1) の flat-chaining の fluent interface 生成器が存在する，というより現実的な仮定を置いて提案をおこなっている．

6　まとめ

本論文は，Python 上の EDSL で BNF 風の文法を記述すると，flat-chaining と sub-chaining の両方を表現する Java の fluent interface を生成するツールを提案した．sub-chaining と flat-chaining を表現する元の入力文法が空列を生成する非終端記号を含まない GNF の LL(1) なら，GNF の LL(1) に sub-chaining の表現を埋め込めることを示した．

参 考 文 献

[1] Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. An empirical study of method chaining in java. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, p. 93–102, New York, NY, USA, 2020. Association for Computing Machinery.

[2] A. M. Keshk and R. Dyer. Method chaining redux: An empirical study of method chaining in java, kotlin, and python. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 546–557, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[3] Tetsuro Yamazaki, Tomoki Nakamaru, and

Shigeru Chiba. Yet another generating method of fluent interfaces supporting flat- and sub-chaining styles. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, pp. 249–259, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, Vol. 12, No. 1, pp. 42–52, jan 1965.

[5] Hao Xu. Erilex: An embedded domain specific language generator. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, pp. 192–212, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, Vol. 17, No. 3, pp. 226–256, 1970.

[7] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent api generator. *SIGPLAN Not.*, Vol. 52, No. 12, pp. 199–211, oct 2017.

[8] Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

```
class FI {
    SSelectQuery<BottomOfSelectQuery> SelectQuery;
    SWhereClause<BottomOfWhereClause> WhereClause;

    public FI() {
        SelectQuery = new SSelectQuery<BottomOfSelectQuery>(new BottomOfSelectQuery());
        WhereClause = new SWhereClause<BottomOfWhereClause>(new BottomOfWhereClause());
    }

    static void action_select(String arg0) {
        return;
    }

    static void action_where(String arg0) {
        return;
    }

    static void action_table(String arg0) {
        return;
    }

    static void action_selectQuery(FI.BottomOfSelectQuery arg0) {
        return;
    }

    static void action_tables(FI.BottomOfTables arg0) {
        return;
    }

    static void action_whereClause(FI.BottomOfWhereClause arg0) {
        return;
    }

    static class BottomOfSelectQuery {
        boolean end() {
            return true;
        }
    }
    static class BottomOfTables {
        boolean end() {
            return true;
        }
    }
    static class BottomOfWhereClause {
        boolean end() {
            return true;
        }
    }

    static class SSelectQuery<Rest> {
        Rest rest;

        public SSelectQuery(Rest rest, Tree tree) {
            this.rest = rest;
        }

        SWhereClause<Rest> select(String arg0) {
            action_select(arg0);
            SWhereClause<Rest> WhereClause = new SWhereClause<>(this.rest);
            return WhereClause;
        }

        Rest selectQuery(FI.BottomOfSelectQuery arg0) {
            action_selectQuery(arg0);
            return rest;
        }
    }

    static class STable<Rest> {
        Rest rest;

        public STable(Rest rest, Tree tree) {
            this.rest = rest;
        }

        Rest table(String arg0) {
            action_table(arg0);
            return rest;
        }
    }

    static class SWhereClause<Rest> {
        Rest rest;

        public SWhereClause(Rest rest, Tree tree) {
            this.rest = rest;
        }

        STable<Rest> where(String arg0) {
            action_where(arg0);
            STable<Rest> Table = new STable<>(this.rest);
            return Table;
        }

        Rest whereClause(FI.BottomOfWhereClause arg0) {
            action_whereClause(arg0);
            return rest;
        }
    }
}

public class Main{
    Run | Debug
    public static void main(String[] args){
        FI fi = new FI();
        System.out.println(fi.SelectQuery.select("*").whereClause(new FI().WhereClause.where("~").table('test_table').end()));
    }
}
```

図 7　Sub-chaining の表現を含む文法の fluent interface