# 卒 業 研 究 報 告 書

題　目　　異なるプログラミング言語間での
コードクローン規模の調査

指導教員　　水野　修　教授

崔　恩瀞　助教

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　19122011

氏　　名　　大満　尚人

令和5年2月13日提出

# 異なるプログラミング言語間でのコードクローン規模の調査

令和 5 年 2 月 13 日　　19122011　大満　尚人

## 概　要

ソースコード中に存在する構文的，意味的に一致，または類似したコード片はコードクローンと呼ばれ，一般にソフトウェアの保守を難しくする原因の一つと言われている．その保守を容易にするために，コードクローンの検出を自動的に行うツールの開発が行われてきた．現在ソフトウェア開発の需要と共にプログラミング言語の多様化は進んできた．プログラミング言語の特徴や使用範囲等は様々であるため，検出されるコードクローンの特徴についても異なる可能性がある．その差異について調べることで，各プログラミング言語に必要なコードクローン検出による支援の違いを明らかにできる．本論文ではその前段階として，9 個の人気なプログラミング言語に対してコードクローン規模の調査を行い，プログラミング言語とコードクローン規模の間にはどのような関係性があるかを明らかにした．具体的には，PYPL から選択した 9 個の人気なプログラミング言語について GitHub 上からリポジトリをそれぞれ 30 個選択し，コードクローン検出を行ってその結果を分析した．

調査の結果，全てのプログラミング言語についてコードクローン検出数の分布は小さい値への偏りを示したが，統計的検定を行ったところ，コードクローン検出数はプログラミング言語間で有意な差があった．また，C，C++，Objective-C 等の派生関係にあるような構文的に類似するプログラミング言語間では，コードクローン検出数の分布や，有意な差があるプログラミング言語が類似していた．さらに，プログラミング言語間の有意差の有無と特徴の違いの有無についてカテゴリカルデータ分析を行ったところ，プログラミングパラダイム，コンパイルクラス，タイプクラスといった構文に与える影響が小さい言語クラスは，プログラミング言語間の有意差に与える影響が小さく，実行形式，メモリ管理といった構文に与える影響が大きい言語クラスは，与える影響が大きいという結果が得られた．これらの結果は，異なるプログラミング言語間でのコードクローン規模は異なり，また，プログラミング言語の構文的類似性や特徴等がコードクローン規模に影響を与えている可能性があることを示唆している．

# 目　次

# 1. 緒言

コードクローンとはソースコード中に存在する構文的，意味的に一致，または類似したコード片を指し [1]，主にプログラマが既存のコード片に対してコピーアンドペーストを用いた複製を行うことで発生する．コードクローンはソフトウェアの保守を難しくする原因の一つである．例えば，コードクローンを成す片方のコード片に欠陥が混入していると，もう片方のコード片にも欠陥が混入している可能性があるため，同様な欠陥のチェックを繰り返す必要がある．また，コードクローンの位置が把握できていない場合，その位置を特定する作業も増えるため，さらに保守が難しくなる．そのため，コードクローンの検出を自動的に行うツールの開発が行われてきた [2, 3, 4, 5, 6, 7, 8]．

GitHub,Inc. が毎年発表しているレポート Octoverse（注 1）によると，GitHub（注 2）上において現在 150 個以上のプログラミング言語が使用されている．プログラミング言語の構文的特徴や型の扱い方，使用頻度や使用範囲等は様々である．そのため，検出されるコードクローンについても，その出現頻度や種類，有意性等が異なる可能性があり，その差異について調べることで，各言語に必要なコードクローン検出による支援の違いを明らかにできる．様々な言語に対してコードクローンを検出するために，Zhu らはコードクローン検出ツールである MSCCD[8] を開発した．しかし，Zhu らの研究では主にコードクローンの粒度と各言語の構文についての考察をしており，プログラミング言語間のコードクローンの違い等の考察は行われていない．そのため，本論文ではプログラミング言語間のコードクローンの違いの調査の一部として，9 個の人気なプログラミング言語に対してそれぞれコードクローンの規模について調査を行い，その差異についての考察を行った．

本論文の調査では，まず，PYPL（PopularitY of Programming Language）（注 3）から人気なプログラミング言語を 9 個選択した．次に，各言語について GitHub から人気なリポジトリを取得し，コードクローン検出ツールである MSCCD を用いてコードクローンの出現数を計測した．

以降は本論文の章構成を示す．まず，第 2 章では，本論文の調査で必要となる背景

（注 1）: https://octoverse.github.com/
（注 2）: https://github.co.jp/
（注 3）: https://pypl.github.io/PYPL.html

知識についての説明を行う．第 3 章では，本論文の調査目的についての説明を行う．第 4 章では，本論文の調査手法や利用するデータセット，コードクローン検出ツールについての説明を行う．第 5 章では，調査結果について示し，第 6 章では，その調査結果について各プログラミング言語の特徴を踏まえて考察を行う．第 7 章では，妥当性への脅威について説明を行い，最後の第 8 章では，結言を示す．

# 2. 背景

## 2.1 コードクローン

コードクローンとはソースコード中に存在する構文的，意味的に一致，または類似したコード片を指す [1]．互いにコードクローンとなる 2 つのコード片はクローンペア，コードクローンとなるコード片の集合はクローンセットと呼ばれる [5]．

### 2.1.1 コードクローンのタイプ

コードクローンは以下の Type1〜4 に分類することができる [2]．

Type1 **コードクローン**

空白やタブ文字，コーディングスタイル，コメントの違いを除いて一致するコード片同士によるコードクローン

Type2 **コードクローン**

Type1 の違いに加え，識別子，リテラル，型の違いを除いて，構文的に一致するコード片同士によるコードクローン

Type3 **コードクローン**

Type2 の違いに加え，文の挿入，削除，変更等が行われているコード片同士によるコードクローン

Type4 **コードクローン**

類似した処理を行うが，構文上の実装が異なるコード片同士によるコードクローン

テキスト的，構文的な類似性に基づいている Type1〜3 コードクローンは主にコード片の複製による再利用が原因である．プログラムの構文に依存しない，機能的，意味的な類似性に基づいている Type4 コードクローンはコード片の複製による再利用以外が原因となる．

### 2.1.2 コードクローンの生成要因

コードクローンが生成される原因は様々である [1]．それらの例を以下に示す．

### (1) コード片の複製による再利用

コードクローンが生成される主な原因である．既存のコード片の処理と同様または少しだけ異なった処理が必要となった場合に，そのコード片がコピーアンドペーストによって複製され，必要に応じて定数や変数名等が書き換えられる事でコードクローンが生じる．

### (2) コードの自動生成ツール

統合開発環境等に含まれるコードの自動生成ツールによって生成されるコードは，同じようなパターンが多く含まれることがある．多くの場合，自動生成されたコードはソースコード解析をする場合に除外される [9]．また，人間が生成したコードクローンではないため，コードクローンとして扱われない傾向もあるが，本論文ではコードクローンとして扱い，解析対象としている．

### (3) プログラミング言語の機能不足

プログラミング言語の進化により，複数の場所で利用される機能を共通の機能として定義し，利用することができるプログラミング言語が増えている．それ以外のプログラミング言語ではその共通の機能を複数回書くことになるため，これがコードクローンの生成要因となる．

### (4) 定型処理

特定の機能を呼び出す際に，ほぼ定型のコードを書くことを要求される場合がある．例として，データベースのライブラリを利用する際には，データベースのオープン，クエリの生成，問い合わせ，結果の受け取り，データベースのクローズといった一連の定型処理を書く必要がある．一般に，定型処理はライブラリの利用例で示されており，プログラマは利用例を参考にコードを記述している．このような定型処理は，誰が書いても類似したコードとなることから，コードクローンであるが，コード片のコピーアンドペーストとは異なって問題となることは少ない．

#### (5) 設計の失敗や不足

設計段階における共通機能の括りだしが不十分なまま，その設計を基に開発者がコーディングを行う場合，共通の機能を複数回に書くことになる．そのため，その共通機能がコードクローンとなってしまう．

### 2.1.3 コードクローンにより発生する問題

一般的にコードクローンが含まれることは，ソフトウェアの保守を難しくすると言われている．

Baker の研究 [10] では，典型的なソフトウェアシステムにおけるコードの 19%から 24%がコードクローンであることがわかっている．このようなコードクローンの多くは意図的であり，ソフトウェアアーキテクチャを理解しやすい状態に保ったり，信頼できるコード片を再利用することでエラーを減らす等に利用されている [11]．これらのコードクローンは目的を持って意図的に生成されたため有用であるが，ソフトウェアの保守や進化の妨げとなるコードクローンも存在する．例として，ソースコード中に存在するクローンペアの片方のコード片に欠陥が含まれていた場合，そのコード片と類似するクローンペアやクローンセットにも欠陥が含まれている可能性が高いため，それらコード片のすべてに対して同様の欠陥を確認する必要がある．

他の問題点として，コピーアンドペーストによる複製によってコードクローンが生成される場合，コードの拡張や，定数，変数名等の変更作業が必要となる．また，この作業が十分でない場合，そのコードクローンとなるコード片に欠陥が含まれる可能性がある．実際に欠陥が含まれていた場合，類似するクローンペアやクローンセットに対して欠陥の確認を行うことになり，よりソフトウェアの保守を困難にしてしまう．

一般に，これらの問題を引き起こすコードクローンは，リファクタリング (注 4)することで除去できることが知られており，これまでに様々なリファクタリングパターンが考案されている [13].

---

(注 4): ソースコードに対して外部的振る舞いを保ちつつ内部構造を改善することで，ソフトウェアシステムを変更するプロセス [12].

### 2.1.4 自動検出ツール

ソースコード中に含まれるコードクローンの位置を特定できれば，クローンペアの片方のコード片に欠陥が含まれていた場合に，容易にもう片方のコード片の位置を特定できるため，2.1.3 節で示したコードクローンの問題点であるソフトウェアの保守が難しくなることを防ぐことができる．しかし，2 つのコード片が与えられたときに，それらがクローンペアであるかを目視で判断することは容易であるが，ソースコード中に含まれるすべてのコードクローンを発見することは人手では難しい．特に大規模なプログラムを対象にする場合，より困難になる．そのため，大規模なプログラムに対しても現実的な時間計算量と空間計算量ですべてのコードクローンを検出できるコードクローン検出ツールが多く開発された [2, 3, 4, 5, 6, 7, 8]．また，コードクローン検出ツールはコードクローンを検出する以外にも,「検出したコードクローンを種類ごとに分類できる」,「検出したコードクローンが問題を引き起こすかの観点から有意性を判定できる」,「検出対象のプログラミング言語に依存しない拡張性をもつ」等の特徴が求められている [14]．しかし，コードクローン検出ツールによっては，一部の種類のコードクローンしか検出できないものや，一部のプログラミング言語で書かれたソースコードに対してしか検出できないものがある．コードクローン検出ツールが検出できるコードクローンの種類や対象にできるプログラミング言語は，そのツールが利用している検出手法に依存している．

コードクローンの検出手法はこれまでに多く提案されてきた．これらの手法を用いて作成されるコードクローン検出ツールは主に以下の手順で検出を行う [2].

**手順 1. 前処理**

検出に必要とされない部分を取り除く．同時にソースコードを分割して比較の対象領域を決定する．

**手順 2. 変換**

比較に適した中間表現に変換する．

**手順 3. コードクローンの検出**

変換されたものに対してコードクローンの検出を行い，クローンペアやクローンセットのリストを作成する．

**手順 4. 出力**

得られた検出結果を元のソースコードと照らし合わせて出力する．

これまでに提案されてきたコードクローンの検出手法における検出アプローチは，主に，テキスト，字句，構文，意味の 4 つの解析単位に基づいて分類される [2]．しかし，これら以外にバイナリコードを解析するもの，実行時のメモリを解析するもの，複数のアプローチを取り入れたもの等も存在する．

### (1)　テキストによるアプローチ

テキストによるアプローチでは，コードクローンの検出における比較以前にソースコードに対して変換や正規化をほとんど行わない，すなわちソースコードを文字や行の並びのまま扱ってコードクローンを検出する．例として，一定の行数のコード片をハッシュ値に変換して比較する方法や，ソースコードに対して自然言語処理と同様なヒューリスティックを用いる方法等の検出方法がある．このアプローチは，中間表現への変換が容易，もしくはほぼ無いため，他のアプローチよりも計算量が少なく済む利点がある．

このアプローチを用いるコードクローン検出ツールに NICAD[3] がある．NICAD は構文解析を行える TXL という言語を用いてコードをより多くの行に分割した後に，テキストの行に基づいたコードクローンの検出を行っている．コードクローンの検出時はテキストによるアプローチがとられているが，前処理段階で構文解析を行っているため，テキストによるアプローチと構文によるアプローチのハイブリッドなコードクローン検出器である．

### (2)　字句によるアプローチ

字句によるアプローチでは，コンパイラ形式の字句解析を用いてソースコードを字句の並び（トークン列）に変換し，類似する部分列を検出することでコードクローンの検出を行う．このアプローチは識別子やリテラルを抽象化できるため，Type2 コードクローンを検出できる．また，意味を成さない空白やタブ文字，コーディングスタイル，コメントの違いを容易に吸収できるといった利点がある．

このアプローチを用いるコードクローン検出ツールに CCFinder[4] がある．CCFinder は字句解析によって生成される接尾辞木を用いたアルゴリズムにより

コードクローンの検出を行っている．Type3 コードクローンは Type2 コードクローンに対して，文の挿入，削除，変更等を許容したものであるため，Type1 または Type2 の 2 つのコードクローン間の行が一定行数以下である場合に，それらを合わせると Type3 コードクローンと見なせる．CCFinder ではこの手法を拡張し，より有意性のあるコードクローンが検出できるようにソースコードの正規化を行っている．また，他の基礎技術として多く利用されており，CCFinder を再設計した CCFinderX や，その CCFinderX を容易に言語拡張できる CCFinderSW[5] 等も存在する．

### (3) 構文によるアプローチ

構文によるアプローチでは，構文解析器を用いてソースコードを解析木や抽象構文木に変換し，類似する部分木を検出したり，構造メトリクスを比較したりすることでコードクローンの検出を行う．構文解析によって生成される解析木や抽象構文木はコンパイラでよく利用されるデータ構造であるため，正規化が行いやすく，また，ソースコードにおける制御ブロックを考慮したコードクローンの検出が容易となる．

このアプローチを用いるコードクローン検出ツールに Deckard[6] がある．Deckard では，抽象構文木から類似する部分木を効率的に検出するために，部分木に対してユークリッド空間における特定の特徴ベクトルを計算し，ユークリッド距離に基づいて類似する特徴ベクトルをクラスタリングすることでコードクローンを検出している．また，MSCCD[8] もこのアプローチを用いている．MSCCD では構文解析に基づいてトークンバッグ (注 5)を抽出し，その類似度を比較することでコードクローンを検出している．最初に行われる構文解析では，構文解析器の生成器である ANTLR を利用しており，ANTLR は様々なプログラミング言語に対応しているため，MSCCD は言語拡張性が高い．

### (4) 意味によるアプローチ

意味によるアプローチでは，静的プログラム解析を用いて，構文的な類似性ではなく意味的な類似性を考慮してコードクローンの検出を行う．意味的なアプローチ

(注 5): コードブロック中に含まれる順序を考慮しない字句（トークン）とその出現頻度を示す集合.

であるため，機能的，意味的な類似性に基づいたType4コードクローンの検出に適している．このアプローチにおいては様々な手法があり，ソースコードからプログラム依存グラフを抽出して制御やデータの依存関係を基にコードクローンの検出を行うものや，近年では機械学習を用いたもの等がある．

このアプローチを用いるコードクローン検出ツールにSCDetector[7]がある．SCDetectorでは，まず，静的解析に基づいて検出単位であるメソッドの制御フローグラフを抽出する．次に，得られた制御フローグラフに対して効率的な類似度計算を行うために，各ノードに対して中心性の分析を行い，グラフの中心性を持つ意味的トークンを得る．最後に，意味的トークンに対して深層学習を用いて類似度計算を行い，コードクローンを検出している．

## 2.2 プログラミング言語

### 2.2.1 特徴と分類

現在，利用されているプログラミング言語は数多く存在する．すべての言語には異なる特徴があるため利用される範囲や対象が異なる．本論文で対象とした人気な9言語に対して特徴別に分類したものを以下の表2.1に示す.

#### (1) プログラミングパラダイム

プログラミングパラダイムとはプログラム自体の見方や捉え方を指す．本論文では言語間の分類を明確にすることで比較を容易にするために，プログラミングパラダイムを手続き型とオブジェクト指向の2つに大別する．なお，このどちらにも含まれないプログラミング言語も存在する．手続き型は手続きと呼ばれる部分プログラムの定義と呼び出しでプログラム全体を構成するものである．オブジェクト指向はプログラムでオブジェクトと呼ばれるものを表現し，それらを相互に作用させることでプログラム全体を構成するプログラミングパラダイムである．

**表 2.1　特徴に基づいたプログラミング言語の分類**

| 言語クラス | 分類 | プログラミング言語 |
|---|---|---|
| プログラミング | 手続き型 | C |
| パラダイム | オブジェクト指向 | Python，JavaScript，TypeScript，Java，<br>C++，C#，Objective-C，PHP |
| 実行方式 | コンパイル方式 | C，C++，Objective-C |
| | インタプリタ方式 | Python，JavaScript，PHP |
| | 中間方式 | Java，TypeScript，C# |
| コンパイルクラス | 静的型付け | Java，C，C++，C#，Objective-C，TypeScript |
| | 動的型付け | Python，JavaScript，PHP |
| タイプクラス | 弱い型付け | JavaScript，C，C++，Objective-C，PHP |
| | 強い型付け | Python，TypeScript，Java，C# |
| メモリ管理 | 必要 | C，C++，Objective-C |
| | 不必要 | Python，Java，JavaScript，TypeScript，<br>C#，PHP |

### (2) 実行方式

ソースコードから機械語に変換するタイミングの違いによって，プログラミング言語はコンパイル方式，インタプリタ方式，コンパイル方式とインタプリタ方式の中間方式に分けられる．コンパイル方式ではソースコードを機械語に変換し，その機械語となったプログラムを実行する．インタプリタ方式ではソースコードを機械語に変換しながらプログラムを実行する．コンパイル方式とインタプリタ方式の中間方式ではソースコードがソースコードと機械語の中間言語に変換された後，その中間言語を機械語に変換しながらプログラムを実行する．

### (3) コンパイルクラス

変数等のデータの扱い方である「型」の定め方の性質として，静的型付けと動的型付けがある．ソースコードの時点ですでに型を定められる場合を静的型付け，ソースコード時点で型は決まっておらず，コンパイラによるコンパイル時やインタプリタによる実行時に型を解釈，決定する場合を動的型付けと呼ぶ．

### (4) タイプクラス

型付けにおいて型を解釈する際の性質として，弱い型付けと強い型付けがある．正確な定義は存在しないが，本論文では次のように定める．弱い型付けは型変換や異なる型同士の演算が許されたり等，型を多様に解釈できる方法を許容したもので，強い型付けはこれが許されていない．

### (5) メモリ管理

プログラミング言語によっては，プログラマにメモリ管理を要求するものとメモリについて考えなくていいものがある．メモリ管理が必要なプログラミング言語では，プログラマが確保したメモリ領域を解放しなければメモリを確保したままとなってしまい，メモリリークという不具合を起こす可能性がある．メモリ管理が必要でない言語では，不要となったデータが格納されたメモリ領域は自動的に解放されるため，プログラマがメモリについて意識する必要がない．

### 2.2.2 プログラミング言語の人気

オープンソースソフトウェアのプラットフォームともなっている GitHub を運営している GitHub,Inc. が毎年発表しているレポート Octoverse によると，2022 年には GitHub 上で約 500 個の言語がソフトウェア開発に利用されている．しかし，GitHub 上で主に利用されているプログラミング言語はその一部である．同レポートによると，開発や教育，機械学習，データサイエンス等の幅広い分野に利用できる JavaScript は 2014 年から変わらず利用され，次いで，Python，Java，TypeScript，C#，C++，PHP，Shell，C，Ruby が主に利用されている．GitHub 上の人気なリポジトリに対する調査を行った論文 [15] では，2016 年における人気なリポジトリの 30%以上が主に JavaScript で構成され，次に多い Python でも 10%未満となっていることから，JavaScript が圧倒的に利用されているということがわかる．しかし，GitHub 上で主に利用されてきたプログラミング言語を示す図 2.1 によると，毎年度言語の利用頻度の順位は変化していて，中でも TypeScript のような急上昇してきた言語や，PHP のような時代と共に順位を落としている言語も存在する．また，同レポートによると IaC（Infrastructure as Code）のツールである Terraform の人気に伴って HCL と呼ばれるプログラミング言語の利用率が急上昇してきている．

プログラミング言語のチュートリアルが Google で検索される頻度を分析することでプログラミング言語の人気度を示している PYPL では，プログラミング言語の人気度の変遷を示す図 2.2 が得られる．この図 2.2 によると，最近では Python が常に一番人気なプログラミング言語であり，次いで Java，JavaScript，C#，C/C++等，GitHub で多く利用されている言語と同じ言語が挙げられている．また，PYPL においても TypeScript，Rust，Go のような人気が急上昇している言語も確認できる．

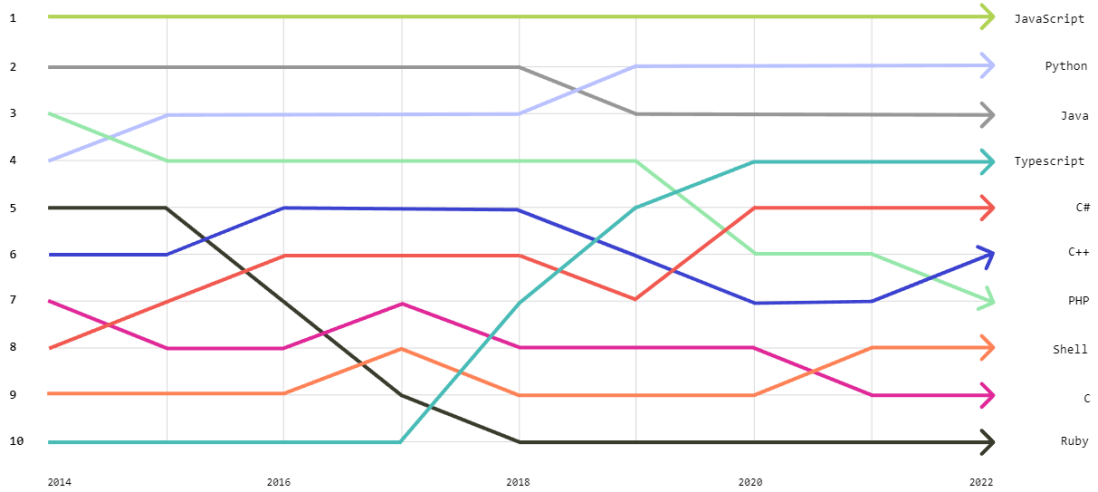


図 2.1　GitHub 上で主に利用されたプログラミング言語の変遷 [16]

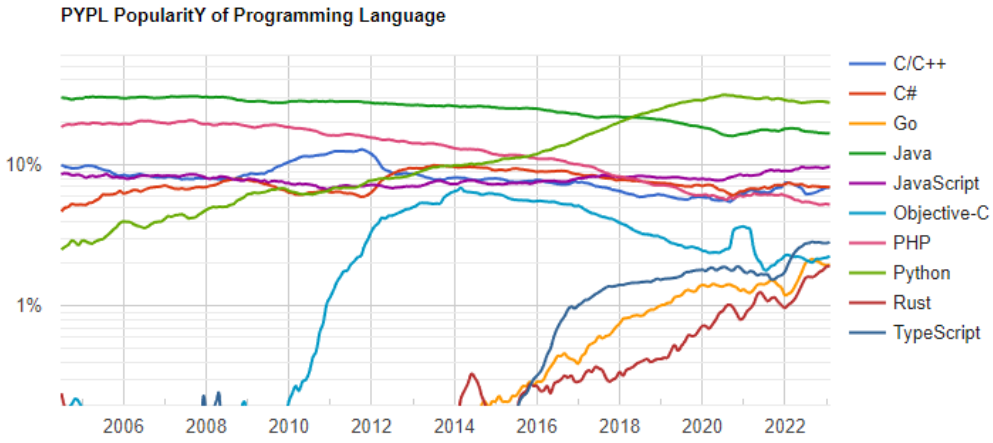


図 2.2　プログラミング言語の人気度の変遷 [17]

# 3. 調査目的

2.2.2 節で説明したように，現在 GitHub 上で約 500 のプログラミング言語がソフトウェア開発に利用されている．GitHub 上で公開されているソフトウェア以外でも，様々なプログラミング言語が利用されているのは明白である．そして，各言語には 2.2.1 節で説明したように，プログラミングパラダイム，実行方式，コンパイルクラス，タイプクラス，メモリクラス等の様々な特徴の違いがあり，また各言語が利用される場面や得意な場面は異なる．これらから，各言語ごとにコードクローンの出現頻度や種類，有意性等が異なる可能性がある．そのため，各言語間におけるコードクローンの出現頻度や種類，有意性等の差異を調べることで，各言語に必要なコードクローン検出による支援の違いを明らかにできる．本論文ではその前段階として，いくつかの人気なプログラミング言語に対してそれぞれコードクローン規模の調査を行い，プログラミング言語とコードクローン規模の間にはどのような関係性があるかを明らかにすることを目標とする．

# 4. 調査

## 4.1 概要

本論文では，次の手順でいくつかの人気なプログラミング言語におけるコードクローンの規模の調査を行う．また，その概要図を図 4.1 に示す.

**手順 1. 調査対象のプログラミング言語の選定**

プログラミング言語の人気順を示している PYPL から，調査に利用するコードクローン検出ツールである MSCCD が対応している上位 9 言語を調査対象のプログラミング言語として選択する．調査対象のプログラミング言語を 9 言語とした理由は主に 2 つある．1 つ目は，できるだけ多くの特徴が異なる言語を調査対象とすることで，偏ったドメイン (注 6)ではなく，プログラミング言語間での比較といったより一般的な調査をするためである．2 つ目は，研究期間も考慮して全体の調査対象を制限し，かつ，各言語について調査対象のリポジトリ数をある程度設けるためである.

**手順 2. 調査対象のデータセットの作成**

選択したそれぞれのプログラミング言語について，GitHub 上で Star 数の上位からリポジトリのリスト化を行い，調査目的に沿わないリポジトリの除外を行う．最終的に，各言語について 30 個のリポジトリを選択し，計 270 個のリポジトリが含まれているデータセットを作成した．これらのデータセットの概要については 4.2 節に記載する.

**手順 3. MSCCD を用いたコードクローンの検出**

データセットとしてリスト化した各言語の各リポジトリに対して，2.1.4 節で説明した MSCCD を用いてコードクローンの検出を行う．なお，研究期間を考慮して，検出時に検出時間が 5 日を超えたリポジトリについては分析対象から除外した．MSCCD の概要等については 4.3 節に記載する.

(注 6): ソフトウェアが利用される領域．アプリケーションやシステム，Web ライブラリ・フレームワーク，ソフトウェアツール等のドメインがある [15].

**手順 4. 検出されたコードクローンの分析**

各プログラミング言語の各リポジトリに対して，コードクローンの検出結果からコードクローン検出数を抽出し，統計的分析を行う．

## 4.2 調査対象のデータセット

本論文では調査対象のデータセットを作成するために，ソフトウェア開発のプラットフォームである GitHub を利用している．GitHub 上には多くリポジトリが存在し，一般的な利用や開発が行われているソフトウェアが公開されている．これらはコードクローンの検出による支援の対象となるソフトウェアそのものであることから，本論文での調査目的に沿ったデータセットの作成元として GitHub を利用した．GitHub 上では，Star と呼ばれる気に入ったリポジトリを覚えておくための機能があり，リポジトリにおける Star 数はそのリポジトリの人気度の指標ともなる．人気度の高いリポジトリはそのプログラミング言語の代表的な，ある程度の規模を持ったリポジトリであるため，本論文では調査対象として GitHub 上で Star 数の多いリポジトリを利用する．

GitHub には REST API と GraphQL API と呼ばれる API が用意されており，これらを用いることでリポジトリの一覧やそれらの Star 数を含んだメタデータ，そのメタデータを用いたソート等を自動的に行うことができる．本論文ではその GitHub API を利用して各プログラミング言語についてリポジトリを Star 数でソートし，リポジトリの情報を取得した．

GitHub における公開リポジトリの中にはプログラミング言語のチュートリアル文書や，プログラムを集めたコレクション等のソフトウェアではないリポジトリも含まれている．これらのリポジトリは本論文での調査目的であるコードクローン規模の調査に沿わないため，調査対象から除外する必要がある．本論文の調査では，各リポジトリの概要や README ファイル等を参考にして目視で選別を行った．

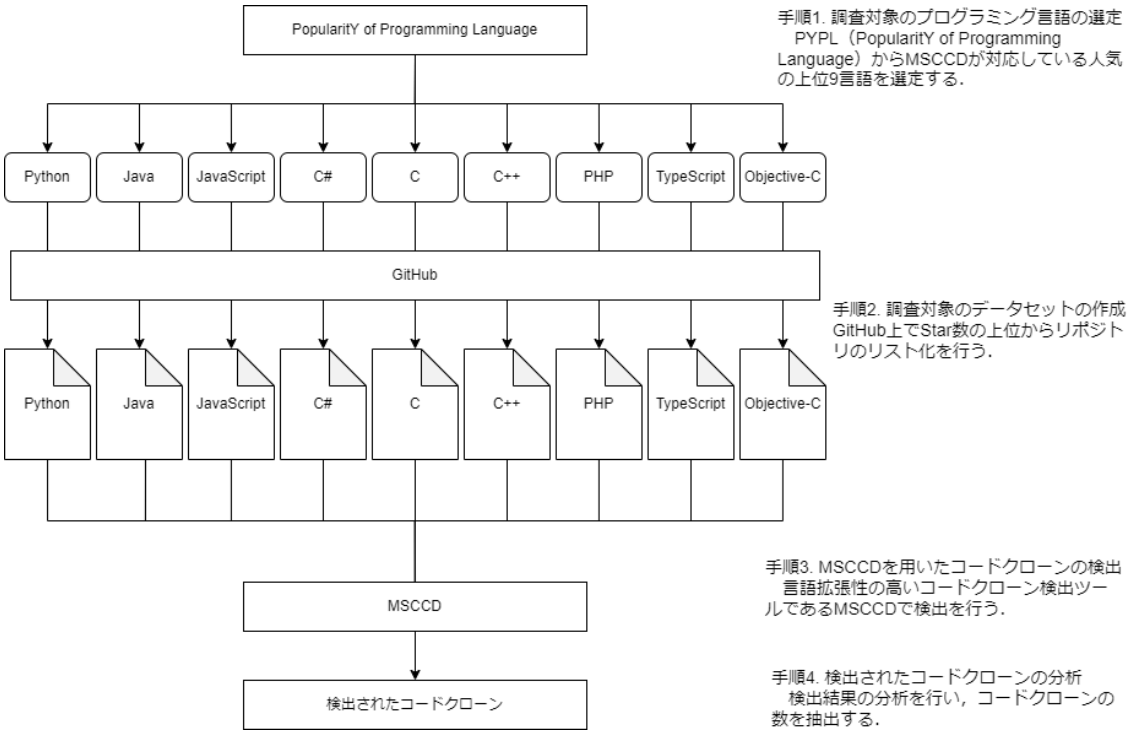
PopularitY of Programming Language
手順1. 調査対象のプログラミング言語の選定
PYPL（PopularitY of Programming Language）からMSCCDが対応している人気の上位9言語を選定する．
Python
Java
JavaScript
C#
C
C++
PHP
TypeScript
Objective-C
GitHub
手順2. 調査対象のデータセットの作成
GitHub上でStar数の上位からリポジトリのリスト化を行う．
Python
Java
JavaScript
C#
C
C++
PHP
TypeScript
Objective-C
MSCCD
手順3. MSCCDを用いたコードクローンの検出
言語拡張性の高いコードクローン検出ツールであるMSCCDで検出を行う．
検出されたコードクローン
手順4. 検出されたコードクローンの分析
検出結果の分析を行い，コードクローンの数を抽出する．

**図 4.1　調査手順の概要**

## 4.3 調査に利用するコードクローン検出ツール MSCCD

### 4.3.1 採用理由

本論文ではコードクローンの自動検出ツールとしてMSCCD[8]を利用する．MSCCDを利用する理由は2つある．

1つ目は，MSCCDは言語拡張性が高いためである．MSCCDではまず，構文解析器の生成器であるANTLRに基づいて，構文解析とコードブロック分割を行うトークナイザを生成する．次に，そのトークナイザを用いて対象のソースコードをトークンバッグに分割し，そのトークンバッグ同士の類似度をもとにコードクローンを検出する．すなわち，ANTLRにおける文法定義ファイルが存在する言語であればトークナイザを生成でき，コードクローンを検出することができる．また，ANTLRにおける文法定義ファイルのリポジトリとしてgrammars-v4 (注 7)があり，ここには150言語以上の文法定義ファイルが置かれている．これらを用いることにより，MSCCDは容易に複数のプログラミング言語に対応することが可能であるため，言語拡張性が高い．本論文の調査では異なるプログラミング言語で書かれた複数のリポジトリに対して同一条件でコードクローンの検出を行うことが望ましいため，言語拡張性の高いコードクローン検出ツールを選択した．

2つ目は，十分な検出精度があるためである．MSCCDの提案論文によると，再現率は，Type1，Type2コードクローンについては100%，98%，Type3の中でも類似度が高いVery Strongly Type3コードクローンについては93%と非常に高い．また，精度についても，92%と非常に高い．これらの再現率と精度の値は，CCFinderXやNICAD等の既存のコードクローン検出ツールと比較しても高く，また，最新のコードクローン検出ツールと同等であることが示されている．

### 4.3.2 検出手順概要

MSCCDは，コードブロック抽出器の生成，トークンバッグ生成，コードクローンの検出の3つの手順で処理が行われる．

**手順 1. コードブロック抽出器の生成**

トークナイザの生成器に，コードクローンの検出対象とするプログ

(注 7): https://github.com/antlr/grammars-v4

ラミング言語についての ANTLR の文法定義ファイルを入力として与えることで，トークナイザを生成する．

**手順 2. トークンバッグ生成**

生成されたトークナイザに対して，コードクローンの検出対象とするソースコードとキーワードフィルタに用いるキーワードリストを入力として与えることで，トークンバッグの集合が生成され，出力される．

**手順 3. コードクローンの検出**

生成されたトークンバッグの集合から類似するペアを見つけることにより，コードクローンを検出する．出力として，クローンペアとなるトークンバッグの情報のリストが得られる．

### 4.3.3 検出するコードクローンの種類

MSCCD は構文解析を用いてコードブロックに分割し，コードクローンを検出しているため，構文的なアプローチに分類される．そのため，構文的な類似性に基づいたコードクローンは検出することが可能であるが，意味的な類似性に基づいたコードクローンの検出には不向きである．そのため，本論文では構文的な類似性に基づいたコードクローンである Type1，Type2，Type3 コードクローンを対象に調査を行った．

### 4.3.4 検出データ

MSCCD では，コードクローンの検出における中間生成物であるトークンバッグの集合が記された「tokenBags」ファイルや，どのトークンバッグがコードクローンに対応しているかを示す「pairs.file」ファイル等が出力される．「pairs.file」では一行ごとにどのトークンバッグの対がコードクローンに対応しているかが記されているため，「pairs.file」の行数がコードクローン検出数となる．本論文の調査では各リポジトリに対してこのコードクローン検出数を集計し分析を行った．

### 4.3.5 ハイパーパラメータ

MSCCD では，検出時にいくつかのハイパーパラメータを設定しなければならない．ハイパーパラメータの説明と設定した数値，その数値に設定した理由を以下に示す．

minTokens: 20

コードクローンの検出時における中間生成物であるトークンバッグの最小トークン数．初期値であり，また，MSCCD の提案論文 [8] でも使用されている値に設定した．

minTokensForBagGeneration: 20

トークンバッグ生成時におけるトークンバッグの最小トークン数．このハイパーパラメータを小さくすると，minTokens の設定できる範囲が広くなるが，今回の調査では minTokens の値を変更しないため，minTokens と同様の値に設定した．

detectionThreshold: 0.7

コードクローンの検出時における類似度の閾値．初期値に設定した．

maxRound: 10

検出対象とする最大粒度値．コードブロックを検出対象とする値とした．

# 5. 結果

## 5.1 コードクローン検出数の統計量

PYPL より，2022 年 10 月における MSCCD が対応し，かつ，人気な 9 個のプログラミング言語は上位から，Python，Java，JavaScript，C#，C，C++，PHP，TypeScript，Objective-C であった．データセットに対して行ったコードクローン検出の結果を分析して得られた，それぞれの人気なプログラミング言語におけるコードクローン検出数についての統計量を表 5.1 に示す．また，それぞれのプログラミング言語におけるコードクローン検出数の箱ひげ図を図 5.1 に示す．なお，図 5.1 中のバツマーカーは平均値，丸マーカーは外れ値を表している．また，図中に収まらない外れ値は省略している．

## 5.2 プログラミング言語間に対する統計的検定

全てのプログラミング言語間のコードクローン検出数についてクラスカル・ウォリス検定を行ったところ，p 値は $4 \times 10^{-12} < 0.05$ となり，有意水準 5%で分布に差があることが認められた．どのプログラミング言語間に差があるのかを確認するために，すべての言語の組み合わせに対してマン・ホイットニーの U 検定を行った結果の p 値と有意水準 5%で有意差があるかをそれぞれ表 5.2，5.3 に示す．

## 5.3 プログラミング言語間の有意差の判定結果とプログラミング言語の特徴に対するカテゴリカルデータ分析

「対象のプログラミング言語間のコードクローンの検出数は，マン・ホイットニーの U 検定により有意水準 5%で有意差が認められるか（以降，項目 1)」と「対象のプログラミング言語間に言語クラスの分類の違いはあるか（以降，項目 2)」に対するカテゴリカルデータ分析を行う．なお，項目 2 については図 2.1 に基づく．項目 2 における言語クラスをプログラミングパラダイムとした場合の分割表を表 5.4，実行方式とした場合の分割表を表 5.5，コンパイルクラスとした場合の分割表を表 5.6，タイプクラスとした場合の分割表を表 5.7，メモリ管理とした場合の分割表を表 5.8

に示す．また，これらに対してカイ二乗検定を行ったときの p 値，有意水準 5%での有意差の有無，クラメールの連関係数 (注 8)を表 5.9 に示す．

(注 8)：2 つの項目間における関連性を測る指標．0 以上 1 以下の値をとり，関連性が強いほど 1 に近い値となる．

**表 5.1　それぞれのプログラミング言語におけるコードクローン検出数の統計量**

| プログラミング言語 | 平均 | 第 1 四分位数 | 中央値 | 第 3 四分位数 |
|---|---|---|---|---|
| Python | 14894.67 | 171.25 | 638.00 | 6873.00 |
| Java | 39383.20 | 1389.25 | 6264.00 | 53732.25 |
| JavaScript | 55278.57 | 178.25 | 1601.00 | 8147.25 |
| C# | 26036.23 | 509.00 | 3705.50 | 12953.00 |
| C | 809.93 | 46.25 | 120.00 | 408.25 |
| C++ | 415.00 | 28.25 | 51.50 | 388.25 |
| PHP | 5287.17 | 52.75 | 697.00 | 3006.25 |
| TypeScript | 1726.27 | 106.25 | 326.00 | 1717.00 |
| Objective-C | 744.20 | 38.25 | 82.00 | 546.75 |

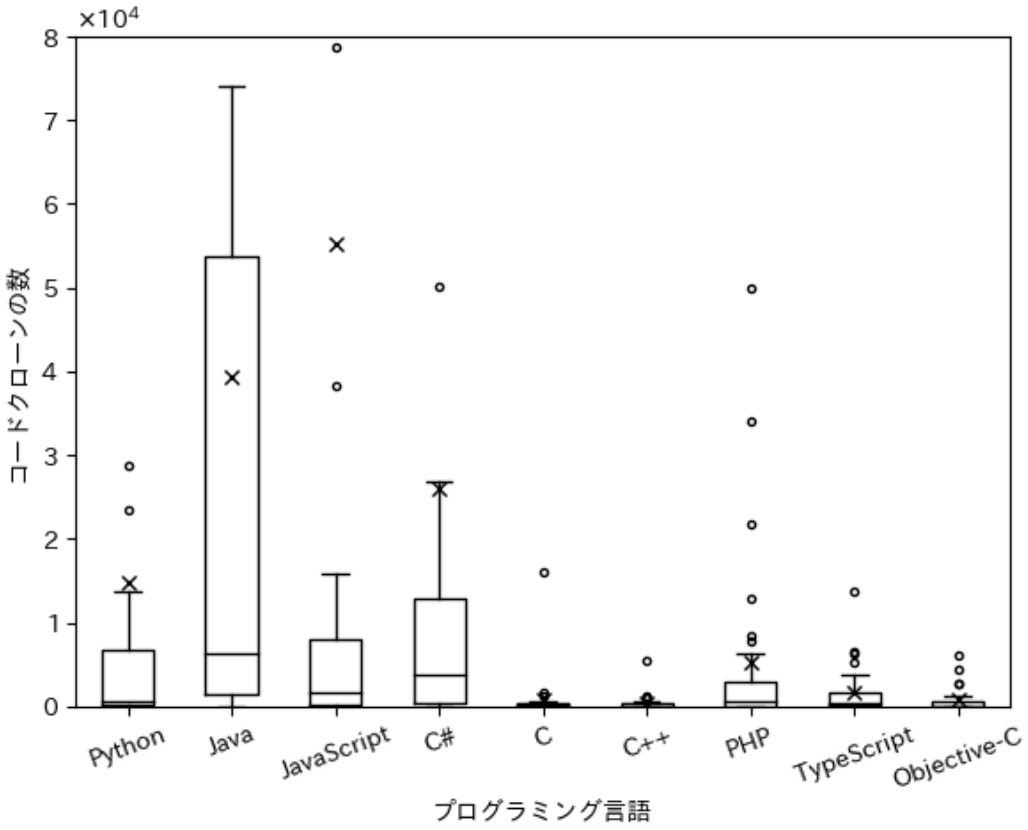


**図 5.1　それぞれのプログラミング言語におけるコードクローン検出数の箱ひげ図**

**表 5.2　プログラミング言語間のコードクローン検出数に対するマン・ホイットニーの U 検定の結果（p 値）**

| | Py | Java | JS | C# | C | C++ | PHP | TS | ObjC |
|---|---|---|---|---|---|---|---|---|---|
| Python | - | 0.005 | 0.871 | 0.147 | 0.001 | 0.000 | 0.455 | 0.139 | 0.002 |
| Java | 0.005 | - | 0.023 | 0.141 | 0.000 | 0.000 | 0.001 | 0.000 | 0.000 |
| JavaScript | 0.871 | 0.023 | - | 0.297 | 0.004 | 0.001 | 0.363 | 0.181 | 0.013 |
| C# | 0.147 | 0.141 | 0.297 | - | 0.000 | 0.000 | 0.042 | 0.005 | 0.000 |
| C | 0.001 | 0.000 | 0.004 | 0.000 | - | 0.315 | 0.023 | 0.026 | 0.756 |
| C++ | 0.000 | 0.000 | 0.001 | 0.000 | 0.315 | - | 0.007 | 0.002 | 0.217 |
| PHP | 0.455 | 0.001 | 0.363 | 0.042 | 0.023 | 0.007 | - | 0.663 | 0.064 |
| TypeScript | 0.139 | 0.000 | 0.181 | 0.005 | 0.026 | 0.002 | 0.663 | - | 0.043 |
| Objective-C | 0.002 | 0.000 | 0.013 | 0.000 | 0.756 | 0.217 | 0.064 | 0.043 | - |

**表 5.3　プログラミング言語間のコードクローン検出数に対するマン・ホイットニーの U 検定の結果（有意差）**

| | Py | Java | JS | C# | C | C++ | PHP | TS | ObjC |
|---|---|---|---|---|---|---|---|---|---|
| Python | - | ○ | | | ○ | ○ | | | ○ |
| Java | ○ | - | ○ | | ○ | ○ | ○ | ○ | ○ |
| JavaScript | | ○ | - | | ○ | ○ | | | ○ |
| C# | | | | - | ○ | ○ | ○ | ○ | ○ |
| C | ○ | ○ | ○ | ○ | - | | ○ | ○ | |
| C++ | ○ | ○ | ○ | ○ | | - | ○ | ○ | |
| PHP | | ○ | | ○ | ○ | ○ | - | | |
| TypeScript | | ○ | | ○ | ○ | ○ | | - | ○ |
| Objective-C | ○ | ○ | ○ | ○ | | | | ○ | - |

**表 5.4　言語クラスをプログラミングパラダイムとしたときの項目 1 と項目 2 の分割表**

| 項目 1 ＼ 項目 2 | 一致する | 一致しない | 合計 |
|---|---|---|---|
| 有意差あり | 17 | 6 | 23 |
| 有意差なし | 11 | 2 | 13 |
| 合計 | 28 | 8 | 36 |

**表 5.5　言語クラスを実行方式としたときの項目 1 と項目 2 の分割表**

| 項目 1 ＼ 項目 2 | 一致する | 一致しない | 合計 |
|---|---|---|---|
| 有意差あり | 2 | 21 | 23 |
| 有意差なし | 7 | 6 | 13 |
| 合計 | 9 | 27 | 36 |

**表 5.6　言語クラスをコンパイルクラスとしたときの項目 1 と項目 2 の分割表**

| 項目 1 ＼ 項目 2 | 一致する | 一致しない | 合計 |
|---|---|---|---|
| 有意差あり | 11 | 12 | 23 |
| 有意差なし | 7 | 6 | 13 |
| 合計 | 18 | 18 | 36 |

**表 5.7　言語クラスをタイプクラスとしたときの項目 1 と項目 2 の分割表**

| 項目 1 ＼ 項目 2 | 一致する | 一致しない | 合計 |
|---|---|---|---|
| 有意差あり | 8 | 15 | 23 |
| 有意差なし | 8 | 5 | 13 |
| 合計 | 16 | 20 | 36 |

**表 5.8　言語クラスをメモリ管理としたときの項目 1 と項目 2 の分割表**

| 項目 1 ＼ 項目 2 | 一致する | 一致しない | 合計 |
|---|---|---|---|
| 有意差あり | 6 | 17 | 23 |
| 有意差なし | 12 | 1 | 13 |
| 合計 | 18 | 18 | 36 |

**表 5.9　項目 1 と項目 2 に対するカイ二乗検定の結果とクラメールの連関係数**

| 項目 2 の言語クラス | p 値 | 有意差（5%） | クラメールの連関係数 |
|---|---|---|---|
| プログラミングパラダイム | 0.4582 | | 0.1236 |
| 実行方式 | 0.0027 | ○ | 0.5008 |
| コンパイルクラス | 0.7286 | | 0.0578 |
| タイプクラス | 0.1207 | | 0.2586 |
| メモリ管理 | 0.0001 | ○ | 0.6361 |

# 6. 考察

## 6.1 コードクローン検出数の統計量

表 5.1 のコードクローン検出数の統計量と，図 5.1 のコードクローン検出数の分布について考察する．

表 5.1 より，全てのプログラミング言語で平均値が中央値より大きいことが分かる．これは，図 5.1 より，大きい値の外れ値が多いことから，これらが平均値を吊り上げているためであると考えられる．また，中央値と第 3 四分位数の差が第 1 四分位数と中央値の差より大きいことが分かる．これらのことから，GitHub 上のリポジトリにおいてはコードクローン検出数は比較的小さい値に分布が偏っており，また，どのプログラミング言語においてもコードクローン検出数が比較的大きいリポジトリが外れ値として存在することが読み取れる．これは，コードクローン検出数と相関関係にあると考えられるリポジトリの規模の分布が小さい側に偏っているため，または，リポジトリの規模が大きくなるほど，コードクローン検出数が加速度的に大きくなるため等の原因が考えられる．

## 6.2 プログラミング言語間に対する統計的検定

プログラミング言語間のコードクローン検出数に対して行った統計的検定の結果について考察する．

全てのプログラミング言語間のコードクローン検出数に対して行ったクラスカル・ウォリス検定では，有意水準 5%で有意差が認められた．そのため，どのプログラミング言語間にコードクローン検出数の分布の有意差があるかを特定するために，全ての言語対に対してマン・ホイットニーの U 検定を行った結果が表 5.2，5.3 である．表 5.3 より，有意差が認められるプログラミング言語対は，認められないプログラミング言語対よりも多い．このことから，GitHub 上のリポジトリにおいて，プログラミング言語がコードクローン検出数に与える影響は大きいことが推測される．また，C，C++，Objective-C の 3 言語や，JavaScript，TypeScript の 2 言語について，これらの派生元と派生先の関係にあるプログラミング言語については，図 5.1 より，コードクローン検出数の分布が比較的類似しており，表 5.3 より，有意差が認められる

言語が類似していることが分かる．さらに，表5.2より，それらの言語間のp値は比較的高い．このことから，プログラミング言語の派生関係にあるような，非常に構文的な類似度が高い言語同士については，コードクローン検出数の分布についても類似度が高くなることが推測される．C#はC言語の派生言語ではあるが，構文的な類似度が他の派生言語よりも低く，また，C#は主に.NET Framework (注9)やUnity (注10)等といった，C言語とは異なる利用範囲を持つため，コードクローン検出数の類似度が高くなかったと考えられる．

## 6.3 プログラミング言語間の有意差の判定結果とプログラミング言語の特徴に対するカテゴリカルデータ分析

プログラミング言語間の有意差の判定結果と，プログラミング言語の特徴による分類について，表5.9のカテゴリカルデータ分析を基に考察する．

プログラミングパラダイムはプログラム自体の見方や捉え方を指すため，手続き型とオブジェクト指向は構文的な違いが大きい．しかし，表5.9より，プログラミングパラダイムの違いによる有意差は認められず，クラメールの連関係数も小さい値を取った．これは，オブジェクト指向に対応したプログラミング言語を利用していても，オブジェクト指向的記述をしていないソースコードを含むリポジトリも存在することが原因であると考えられる．

実行方式におけるコンパイル方式とインタプリタ方式と中間方式の違いは，定義によると直接的に構文上の違いとしては現れない．しかし，表5.9より，実行方式の違いによる有意差は認められ，クラメールの連関係数は大きい値を取った．そのため，実行方式の違いは間接的に構文上の違いに影響を与えていると考えられる．本論文で参照した言語クラスだけで考慮しても，実行方式のコンパイル方式とメモリ管理が必要なプログラミング言語が，表2.1より，C，C++，Objective-Cと一致しているため，実行形式がメモリ管理の分類に関係することで，間接的に構文上の違いに影響を与えていることが推測される．

コンパイルクラスにおける静的型付けと動的型付けの構文的な違いは，主に型定

(注9)：MicroSoftが開発しているWindows上で実行されるアプリケーションの開発，実行環境．
(注10)：Unity Technologiesが開発しているゲームエンジンを搭載したリアルタイム開発プラットフォーム．

義時のデータ型の記述の有無であり，コードクローン検出数に与える影響は小さいと考えられる．そのため，表 5.9 より，コンパイルクラスの違いによる有意差は認められず，クラメールの連関係数が小さい値を取ったと考えられる．

タイプクラスにおける弱い型付けと強い型付けの構文的な違いは，コンパイルクラスとは異なり，型定義時以外の演算部分等にも現れるが，全体的な構文としての違いは大きくない．そのため，表 5.9 より，タイプクラスの違いによる有意差は認められないが，コンパイルクラスと比較してクラメールの連関係数は大きな値を取ったと考えられる．

メモリ管理における必要であるかないかの構文的な違いは，他の言語クラスとは異なり，メモリ管理に関する命令行や関数の有無等の構文全体に関わるため，コードクローン検出数に与える影響は大きいと考えられる．そのため，表 5.9 より，メモリ管理の違いによる有意差が認められ，クラメールの連関係数が大きな値を取ったと考えられる．

# 7. 妥当性への脅威

## 7.1 外的妥当性

本論文では，ソフトウェア開発のプラットフォームである GitHub 上で公開されているリポジトリから，9 個のプログラミング言語についてそれぞれ 30 個，計 270 個のリポジトリをデータセットとして調査を行った．まず，GitHub 上で公開されているリポジトリを利用していることから，それ以外の企業内等で開発されているソフトウェア等では，本論文で利用したデータセットと特徴が異なる可能性がある．次に，対象としたプログラミング言語については，PYPL の人気言語から MSCCD が対応した言語を上位から 9 個選択しているため，人気度の高いプログラミング言語については調査を行えたが，比較的人気度の低い他のプログラミング言語について調査を行うと異なる結果が得られる可能性がある．最後に，対象としたリポジトリについては，9 言語のそれぞれについて GitHub 上における Star 数の多い 30 個，計 270 個のリポジトリに対して調査を行ったが，全てのリポジトリの特徴を網羅できていない可能性がある．

## 7.2 内的妥当性

本論文では，統計量の算出や，クラスカル・ウォリス検定，マン・ホイットニーの U 検定，カイ二乗検定といった統計的検定に基づいて比較検証を行った上で議論している．

## 7.3 構成概念妥当性

本論文での調査は，2022 年 10 月から 2023 年 2 月までの期間を要したため，データセットのリスト作成時とコードクローンの検出期間が異なり，また，全てのリポジトリの参照時期が一時点ではないため，リポジトリの人気順やコードクローン検出数といった情報が，データセットのリスト作成時と一致していない可能性がある．また，このデータセットのリストは GitHub API を用いて作成したため，GitHub API の処理結果に依存する．コードクローンの検出においては，2022 年 12 月時点で最

新の MSCCD を用いているため，その時点における MSCCD の処理に依存しており，欠陥が存在していた場合，妥当性の脅威となり得る．

# 8. 結言

本論文では，人気なプログラミング言語である Python，Java，JavaScript，C#，C，C++，PHP，TypeScript，Objective-C の 9 言語を対象に，各言語 30 個，計 270 個の GitHub 上で公開されているリポジトリに対してコードクローンの検出を行い，各言語のコードクローン検出数について分析を行った．

調査の結果，コードクローン検出数の統計量からは，全てのプログラミング言語においてコードクローン検出数は分布が小さい値に偏る傾向にあり，また，コードクローン検出数が大きい値を取る外れ値が多く存在したことから，平均値が中央値よりも大きく，また，分布が小さい側に偏る傾向にあるといった結果が得られた．プログラミング言語間に対する統計的検定からは，多くのプログラミング言語間においてコードクローン検出数に有意な差があり，また，プログラミング言語の派生関係にあるような非常に構文的な類似度が高い言語間では，コードクローン検出数の分布や有意差があるプログラミング言語も類似するという結果が得られた．マン・ホイットニーの U 検定の結果とプログラミング言語の特徴に対するカテゴリカルデータ分析からは，プログラミングパラダイム，コンパイルクラス，タイプクラスといった言語クラスが，プログラミング言語間におけるコードクローン検出数の検定結果に影響を与えている可能性は低く，実行形式，メモリ管理といった言語クラスが影響を与えている可能性は高いという結果が得られた．これについて，構文に与える影響が大きい言語クラスほど，コードクローン検出数に与える影響も大きいといった原因が考えられる．

# 謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授ならびに崔恩瀞助教に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻渡邉紘矢先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

# 参考文献


[1] 神谷年洋，肥後芳樹，吉田則裕，“コードクローン検出技術の展開,” コンピュータ ソフトウェア，vol.28，no.3，pp.29–42，2011.

[2] C.K. Roy, J.R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming, vol.74, no.7, pp.470–495, 2009.

[3] C.K. Roy and J.R. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” Proceeding of the 16th IEEE International Conference on Program Comprehension, pp.172–181, 2008.

[4] K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” Annual report of Osaka University: academic achievement, vol.28, no.7, pp.654–670, 2002.

[5] 瀬村雄一，吉田則裕，崔　恩瀞，井上克郎，“多様なプログラミング言語に対応可能なコードクローン検出ツール CCFinderSW,” 電子情報通信学会論文誌 D，vol.J103-D，no.4，pp.215–227，2020.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” Proceeding of the 29th International Conference on Software Engineering, pp.96–105, 2007.

[7] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, “SCDetector: software functional clone detection based on semantic tokens analysis,” Proceedings of the 35th International Conference on Automated Software Engineering, pp.821–833, 2020.

[8] W. Zhu, N. Yoshida, T. Kamiya, E. Choi, and H. Takada, “MSCCD: Grammar pluggable clone detection based on ANTLR parser generation,” Proceedings of the 30th International Conference on Program Comprehension, pp.460–470, 2022.

[9] 下仲健斗，鷲見創一，肥後芳樹，楠本真二，“機械学習を用いた自動生成コードの特定,” 電子情報通信学会技術研究報告; 信学技報，vol.115，no.420，pp.165–170，2016.

[10] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proceedings of 2nd Working Conference on Reverse Engineering, pp.86–95, 1995.

[11] C.J. Kapser and M.W. Godfrey, “‘‘cloning considered harmful’’ considered harmful: patterns of cloning in software,” Empirical Software Engineering，vol.13，pp.645–692，2008.

[12] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.

[13] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング,” コンピュータ ソフトウェア，vol.28，no.4，pp.43–56，2011.

[14] 井上克郎，神谷年洋，楠本真二，“コードクローン検出法,” コンピュータ ソフトウェア，vol.18，no.5，pp.529–536，2001.

[15] H. Borges, A. Hora, and M.T. Valente, “Understanding the factors that impact the popularity of GitHub repositories,” Proceeding of the 32nd International Conference on Software Maintenance and Evolution, pp.334–344, 2016.

[16] Inc. GitHub, “The top programming languages — The State of the Octoverse,” `https://octoverse.github.com/2022/top-programming-languages`, 2022. （最終閲覧日: 2023 年 2 月 3 日）.

[17] P. Carbonnelle, “PYPL PopularitY of Programming Language index,” `https://pypl.github.io/PYPL.html`, 2022. （最終閲覧日: 2023 年 2 月 3 日）.