

原著論文

**生成 AI とソフトウェア開発：
中規模プロジェクトにおける人間と ChatGPT のコミュニケーション分析**

米澤 直記

帝京平成大学人文社会学部経営学科

**Generative AI and Software Development :
An Analysis of Human-ChatGPT Communication in Medium-Scale Projects**

YONEZAWA Naoki

Department of Business, Faculty of Humanities and Social Sciences, Teikyo Heisei University

Abstract

This study critically examines the utility and limitations of the generative AI (Artificial Intelligence) model, ChatGPT, in the development of medium-scale software, specifically focusing on a numerical puzzle game called "FunctorX". The research aims to shed light on how conversational AI can contribute to enhancing the efficiency and productivity of software development. While ChatGPT has shown promising results in small-scale programming tasks, it struggles to maintain consistency in larger projects from design to code generation, necessitating human intervention. In our methodology, we engaged in 643 dialogues with ChatGPT throughout the development of FunctorX, capturing conversation logs and categorizing them using 16 labels corresponding to various development stages, such as "Requirement Analysis", "Design", "Implementation", and "Deployment". The labels helped us determine the extent to which ChatGPT was useful or problematic at each stage. Notably, conversations around the "Implementation: Number Input" and "Implementation: Event Listener" were the most frequent, indicating a concentration of discussion during the implementation phase. Our lexical analysis, done via MeCab, revealed that the dialogues were primarily focused on technical aspects, as words like "button", "cell", "input" appeared frequently from the human side, whereas ChatGPT often used terms like "element" and "case". FunctorX's development also exposed difficulties in coordinating the event handler and the number input mechanism, revealing a nuanced understanding of where human expertise remains critical. Moreover, our research brings forth several recommendations for improving conversational code-generating AI. These include ensuring contextual continuity, clarifying the rationale behind any strategy shifts, reducing excessive apologies, and improving semantic accuracy. We posit that these enhancements are crucial for more effective code generation. Although AI in code generation offers exciting prospects, it is not without challenges that need addressing. By implementing these improvements, we expect to witness a significant leap in the AI's effectiveness and precision, making it

more valuable in software development environments.

Keywords: Conversation Analysis, Code Generation AI, Medium-scale Software Development, ChatGPT, Contextual Continuity

1 はじめに

近年、ChatGPT¹⁾をはじめとする生成 AI の技術が進化し、その応用例も文章^{1, 2)}、画像^{3, 5)}、音楽^{6, 7)}、製品デザイン^{8, 9)} など多岐にわたるようになっている。さらに、生成 AI はソフトウェア開発においてプログラミング言語のコードを生成することも可能である。自然言語を用いることで、アルゴリズムやプログラミング言語の知識が乏しい初心者であってもソフトウェア開発に取り組むことが可能となり、大きな期待が寄せられている。現状、GitHub Copilot¹⁰⁾ や Amazon CodeWhisperer¹¹⁾ といったコードを生成する AI も存在するが、これらはプログラミングの中級者以上をターゲットとしたものである。しかしながら、長期的な視点でみると、自然言語は人間の日常的なコミュニケーション手段であるため、「WHAT」から「HOW」を作り出す手法としては、ChatGPT のような会話ベースでのコード生成が主流になる可能性が高いと考えられる。ここで、「WHAT」は「何をするプログラムを作成するか」すなわち「要件定義」を、「HOW」は「どのような処理をするか」すなわち「プログラミング言語で記述された具体的なコード」を指す。

大学のプログラミング演習などで開発される小規模のプログラムにおいては、ChatGPT を用いることでほぼ正確な目的のコードを得ることができており、「WHAT」から「HOW」を作り出す手法としての完成度はすでに非常に高いと言える。しかし、数ヶ月にわたるプロジェクトベースドラッシングや大学の卒業研究における中規模ソフトウェアの開発に ChatGPT を用いる場合、全体の要件定義から最終的な目的のコードを生成するタスクは、現在の技術水準では難しく、人間の介入が必要である。

本研究では、中規模ソフトウェアの開発過程における、著者自身と ChatGPT との間での会話を取り上げ、分析する。この分析を通じて、ChatGPT を中規模ソフトウェア開発に利用する際の問題点や課題を明確にし、それらに対する改善策を提言することを本研究の目的とする。この研究により、生成 AI を活用したソフトウェア開発の効率や生産性の向上に資することが期

待される。

なお、本研究は人を対象とした研究・動物を扱う研究・遺伝子組換え実験を含む研究のいずれにも該当しない。また、開示すべき利益相反関連事項はない。

2 研究方法

本研究では、次のような研究方法を採用した。

まず、著者自身が ChatGPT (言語モデル: GPT-3.5) を利用し、中規模のソフトウェアを開発する。その過程で著者と ChatGPT の間で交わされた会話のログを取得し、その内容を詳しく分析する。具体的には、ソフトウェア開発の各段階を示すラベルを設定し、各会話にこのラベルを付与する。なお、1 つの会話に複数のラベルが付与される場合もある。次に、付与されたラベルの出現割合について分析し、会話に出現した単語の頻度を調査する。なお、本研究の開発では ChatGPT との会話言語として日本語を用いたため、ここでの頻度分析は、日本語の単語のみを対象とする。さらに、著者が感じた不自然な会話や戸惑いを覚えるような会話を特定し、それらの会話に対する改善案を検討する。

この研究で取り上げる中規模のソフトウェアとは、大学におけるプログラミングの授業での課題として 90 分で作成可能なプログラムよりは規模が大きく、かつ 1 人での開発が可能なものを指す。さらに、具体的な開発期間として 7 日から 30 日程度を要する規模を想定している。

以下、本節では本研究の対象となった FunctorX という数字パズルについて述べるとともに、上述の分析で使用するラベルの詳細についても説明する。

2.1 数字パズル FunctorX

FunctorX¹²⁾ は、著者が開発し Web で公開している数字パズルゲームである。ユーザが Web ページにアクセスすると、問題のサイズや難易度を選択するメニューが提示される。

図 1 は、2 × 2 の問題サイズを選択した場合の問題例を示している。図 1 (a) は問題を選択した直後の

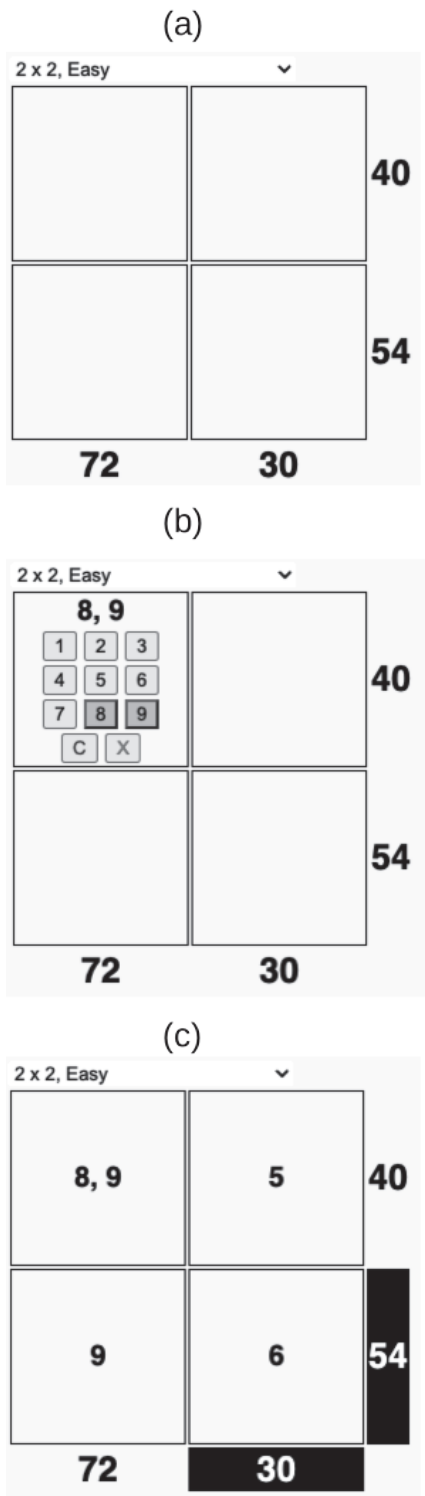


図1 FunctorX のユーザインタフェース

画面であり、 2×2 の表とそれに隣接する欄外の数が表示されている。このパズルの目的は、表の空欄に 1 から 9 までのいずれかの数字を 1 つずつ入力し、行方向の数字の積が欄外の数と一致するようにすることである。列方向についても同様のルールが適用される。

図 1 (b) では、数字入力部の詳細が示されている。1 から 9 までの数字のボタンをクリックするとその数字が上部に表示される。ユーザは複数の数字を候補として入力することも可能である。例えば、積が 72 となる 1 桁の整数は 8 と 9 のみであるため、これら 2 つの数字が解の候補となる。図では「8」と「9」のボタンをクリックした結果として「8, 9」が表示されている。

一方、図 1 (c) は進行中のゲーム画面で、すでにいくつかのセルに数字が入力されている様子がわかる。行や列の積が欄外の数と一致した場合、該当する欄外の数が反転し強調表示される。この問題においては、左上の「8, 9」を「8」に変更すれば、全ての欄外の数が反転し、パズルが正しく解かれたことが確認できる。

なお、他の開発者による作品として Sumplete¹³⁾ という数字パズルが存在する。Sumplete も ChatGPT を活用して開発されたものである。Sumplete では、ゲーム開始の時点で表の中にすでに数字が配置されており、行や列の数字の合計が欄外の数と一致するように、数字を消去していく。Sumplete が和を求めるゲームであるのに対して FunctorX は数字の積を求めるゲームであること、FunctorX では空のセルへの数値入力が必要である点から、FunctorX と Sumplete はゲームの難易度やゲーム性が異なると言える。

2.2 FunctorX の開発プロセス

FunctorX の開発プロセスは、主にアジャイル開発の原則に基づいて進行した。このアプローチは、反復的かつ柔軟な手法であり、プロジェクトの進捗に応じて要件や設計を調整した。具体的には、開発プロセスは時系列として以下のように進行した。なお、以下の「設計」に列挙した各部品の機能については 2.3 節で述べる。

要件定義 開発したいソフトウェアの曖昧なイメージを ChatGPT に投入し、明確化するための対話を繰り返した。

設計 ソフトウェアを構成する部品として、「数字入

力部」「問題選択」「表生成」「問題生成」「正答チェック」「ローカルストレージ」を定義した。

実装 上記の部品を個別に実装した。その過程で、数字入力部での入力処理を支援するために新たな部品である「イベントハンドラー」の設計が必要となった。

統合 「ローカルストレージ」を除く各部品を統合し、1つのソフトウェアとして機能させた。

デプロイ後の改良 1 規模の大きな問題（ 6×6 ）を生成する場合の性能が低いことが判明し、「問題生成」を再実装した。さらに、この過程で新たに「ソルバー」を追加実装した。

デプロイ後の改良 2 「ローカルストレージ」をソフトウェアに統合し、全体の機能を向上させた。

2.3 会話に付与するラベル

本研究における「会話」とは、著者が ChatGPT に質問や要求を入力し、ChatGPT がそれに応答する 1 回のやりとりを指す。これを 1 ターンの会話、あるいは単純に 1 会話と呼ぶことにする。会話ごとに内容を精査し、該当するラベルを付与する。会話の内容によっては、1 会話に対して複数のラベルが付与される場合もある。なお、本研究では、著者の主観に基づいてラベル付けした。

以下に、各ラベルとその定義を示す：

要件定義 開発しようとしている数字パズル FunctorX の内容を明確にするための会話。

設計 どのような技術やツールを活用して FunctorX を実現するか（例：プログラミング言語、実行環境、開発環境の選択）に関する会話。また、ソフトウェアを部品に切り分け、部品間の関係を定義する会話。以下に示すそれぞれの「実装」ラベルは、この「設計」に関する会話で定義された部品に対応する。

実装：数字入力部 数字を FunctorX に入力する部分の実装に関する会話。

実装：ローカルストレージ FunctorX の状態をブラウザのローカルストレージに保存する方法に関する会話（この機能により、ブラウザを閉じた場合でも、プレイを再開することができる）。

実装：問題選択 プルダウンメニューから問題を選択する部分の実装に関する会話。

実装：表生成 選択した問題に基づく表を生成する方

法に関する会話。

実装：問題生成 問題（実質的には、表の欄外の数）を自動生成する方法に関する会話。

実装：正答チェック ユーザの入力に基づき、解答が正しいかどうかを判定する部分の実装に関する会話。

実装：イベントハンドラー セルの選択や数字入力部のボタンクリックなどのイベントを取り扱う部分の実装に関する会話。

実装：ソルバー 自動生成された問題を解く部分の実装に関する会話（ユーザが問題を選択した際に、プログラムは問題を乱数を用いて生成する。ソルバーは、その問題が解けるかどうかプログラム側で事前に判定するために用いられる）。

実装：統合 個別の部品や機能を統合し、単一のアプリケーションとして動作するようにする工程に関する会話。

デプロイメント 完成した FunctorX を Web サーバー上で公開する工程に関する会話。

メンテナンス 公開後のアプリケーションの不具合の修正や改善に関する会話。

ブランディング・マーケティング アプリケーションの名前を決めたり、アプリケーションの知名度向上やユーザ獲得を目指したりする活動に関する会話。

開発支援 プログラミング言語やその他の技術に関する一般的な会話。専門書や Web 検索の代替としての利用を目的とした会話。

挨拶 ChatGPT への感謝や、動作の成功報告などを示す会話。一連の会話の区切りとなることが多いと考えられる。

本研究では、「テスト」に関する独立したラベルは設定していない。この理由は、テストプロセスが「実装」フェーズと密接に結びついているためである。具体的には、開発過程で ChatGPT がコードを生成し、その後で著者がそのコードを実行して機能が正確に動作するかを確認するという会話が繰り返された。このような経緯から、本研究においては「実装」ラベルがテストプロセスも含んでいるものとした。

3 結果

3.1 FunctorX のコード特性と規模評価

著者と ChatGPT の会話について分析した結果を示す前に、開発した FunctorX のコード特性について述べる。FunctorX は、Web アプリケーションであり、3 つの標準的な技術、すなわち、HTML、CSS、JavaScript を組み合わせて作成された。HTML (HyperText Markup Language) はページの構造を定義する言語であり、CSS (Cascading Style Sheets) はスタイルやデザインを指定するための言語である。JavaScript は、クライアントサイドのスクリプティング言語であり、Web ページに対して動的なインタラクションを追加するために使用される。なお、著者は C や Python などの他のプログラミング言語でのプログラミング経験はある^{注1)}ものの、FunctorX の実装を開始した時点では、JavaScript の文法に関する知識はほとんどない状態であった。しかしながら、Web クライアントで作動するアプリケーションのための開発言語としては、実質的に JavaScript と TypeScript に限定されるため、本研究では、JavaScript を採用することとした。さらに、JavaScript の開発フレームワークやライブラリは利用せず、JavaScript の標準機能のみを用いて実装した。

なお、ChatGPT を使用したプログラミング支援については、本研究を通じて初めて取り組んだ。

コードの規模について、各技術に関連する行数は以下の通りである。

HTML 58行 (そのうち、プログラム実行に必要な部分は17行)

CSS 49 行

JavaScript 472 行

本研究では、ここで示したコードの規模を以下のよう評価する。プロフェッショナルなプログラマであれば 1 日で作成できる程度の規模である一方、大学生が 1 回の 90 分授業で完成させることは難しいと考えられる。よって、FunctorX は「中規模」と評価できる。

コードの特性に更に焦点を当てると、関数の個数は無名関数を含めて 18 個であった。このことから、適切にモジュール化されているといえる。具体的に各機能での行数を調査すると、

数字入力部 75 行

ローカルストレージ 8 行

問題選択 17 行

イベントハンドラー 35 行

表生成 89 行

問題生成 64 行

正答チェック 52 行

ソルバー 136 行

となった。このうち、「問題選択」の 17 行は HTML ファイルに含まれており、残りは JavaScript のソースファイルに含まれている。関数の間に空行が存在するため、機能ごとの行数の合計と JavaScript ファイルの行数は一致しない。

3.2 会話のラベル付け

本節では、FunctorX の開発過程で、著者が ChatGPT と交わした会話に対して付与したラベルについて述べる。先述のように、1 会話に対して複数のラベルが付与される場合があったため、ラベルの総数と会話数は一致しない。

会話の総数は 643 回であり、表 1 に各ラベルに付与された回数とその割合を示す。

表 1 会話に付与されたラベルの統計

ラベル	個数	割合 (%)
要件定義	4	0.5
設計	4	0.5
実装: 数字入力部	208	27.5
実装: ローカルストレージ	23	3.0
実装: 問題選択	8	1.1
実装: 表生成	51	6.7
実装: 問題生成	35	4.6
実装: 正答チェック	12	1.6
実装: イベントリスナー	200	26.4
実装: ソルバー	23	3.0
実装: 統合	0	0.0
デプロイメント	40	5.3
メンテナンス	0	0.0
ブランディング・マーケティング	11	1.5
開発支援	118	15.6
挨拶	20	2.6
合計	757	100.0

このうち、「実装：数字入力部」と「実装：イベントリスナー」のラベルが同時に付与された会話は150あった。

3.3 会話に出現する単語の分析

著者と ChatGPT が交わした会話で頻出した単語をそれぞれ表2と表3に示す。会話から単語を抽出し品詞分類するために、形態素エンジン MeCab¹⁴⁾ を利用した。

4 考察

前節で述べた結果を考察する。

4.1 FunctorX のコード特性と規模評価

ソルバーに必要な行数は多かったものの、主観的にはそれほど難しい実装ではなかった点を指摘することができる。逆に、イベントハンドラーと数字入力部を結びつける部分が FunctorX の実装において最も困難な部分であった。一般に、コードの複雑性や難点は、行数だけでは判断できないと言える。これらの点については、次節で詳しく述べる。

4.2 会話のラベル付け

ラベルによる分析から、開発フェーズによって会話の焦点が大きく異なることが明らかになった。特に、「実装：数字入力部」（全ラベルの27.5%）と「実装：イベントリスナー」（全ラベルの26.4%）が非常に高い割合を占めており、これは開発の主要な課題がこれらの部分に集中していたことを示している。さらに、この2つのラベルが同時に付与された会話は先述の通り150（全会話の23.3%）あり、この2つの機能が密接に関連していることが分かる。一方、「開発支援」が全ラベルの15.6%と高く、ChatGPT が開発プロセス全体で多くのサポートを提供していたことがわかる。

興味深い点として、「要件定義」や「設計」のラベルが非常に少ない（全ラベルの0.5%）ことが挙げられる。これは、実装に入る前の初期フェーズで ChatGPT との会話が少なく、それらのフェーズが非常にスムーズだったことを示唆している。具体的には、以下の点が挙げられる：

- 著者が意図するソフトウェアの自然言語による表現を ChatGPT に提供し、その返答を解釈することで、要求定義が明確であることを少数の会話で確

表2 人間の発話における単語の頻度（上位20個）

名詞	頻度	動詞	頻度	形容詞	頻度
ボタン	345	する	1879	ない	87
セル	277	いる	266	よい	64
入力	248	ある	224	ほしい	31
場合	209	なる	215	うまい	26
数字	194	くださる	207	正しい	22
表示	183	押す	116	大きい	22
要素	158	いう	95	新しい	13
以下	151	できる	91	難しい	11
クリック	144	消える	58	おかしい	11
こと	142	しまう	54	いい	11
コード	138	出る	40	長い	8
生成	126	用いる	37	広い	8
領域	122	作る	34	高い	8
数	115	書く	33	少ない	7
関数	110	入れる	33	等しい	4
状態	95	示す	31	惜しい	3
時	93	応ずる	31	欲しい	3
イベント	86	求める	29	汚ない	3
反転	81	呼び出す	26	素晴らしい	2
処理	77	続ける	25	古い	2

表3 ChatGPTの発話における単語の頻度（上位20個）

名詞	頻度	動詞	頻度	形容詞	頻度
こと	862	する	5245	ない	101
要素	707	ある	812	新しい	99
場合	645	いる	752	正しい	72
ボタン	640	できる	558	高い	25
セル	561	なる	358	良い	15
以下	528	くださる	171	大きい	14
関数	526	よる	149	古い	14
ため	472	呼び出す	132	よい	13
コード	452	いう	131	少ない	11
表示	413	使う	114	うまい	8
入力	413	返す	93	広い	6
クリック	345	行う	91	詳しい	6
数字	340	対する	88	多い	5
使用	337	含む	74	楽しい	4
追加	336	持つ	74	等しい	4
イベント	312	みる	68	長い	4
必要	265	示す	60	短い	4
値	231	異なる	58	難しい	4
修正	220	読み込む	56	望ましい	3
配列	211	書く	56	嬉しい	3

認することができた。

- 開発の初期段階で Web ブラウザで作動するソフトウェアを開発することを想定したため、使用する言語、開発環境の選択肢が自ずと限定された。

■ **数字入力部とイベントリスナーの実装** 先述の通り、数字入力部とイベントリスナーに関する会話がそれぞれ200以上と特に多かったという結果が得られた。この事象にはいくつかの理由が考えられる。

- 実装の初期段階で JavaScript に対する著者の知識が不足していたため、基本的な実装にも多くの会話が必要であった。
- 数字入力部には多数のサブコンポーネントが存在するため、これらの要素を ChatGPT に正確に説明する過程が複雑であった。
- 数字入力のイベントをイベントリスナーで処理する機能が初めて実装された際、これらのコンポーネントが複雑に絡み合っていた。その結果、コードの見通しが悪くなった。
- 著者が JavaScript に次第に慣れ、コードの構造を改善するための具体的な指示を ChatGPT に与えられるようになった。具体的には、イベントリスナーに渡される無名関数を独立した関数として切り分けたことで、コードの可読性と管理性が向上した。
- 開発が進むにつれて、著者が ChatGPT の特性と効果的な使い方を理解した結果、より洗練された指示^{注2)}を ChatGPT に与えられるようになった。これにより、単純なコードから始めて段階的にコードを複雑化させるプロセスが確立された。

以上のように、手探りの状態から手法を確立するまでの過程で実装されたため、数字入力部とイベントリスナーの実装に関する会話が多くなったと考えられる。

■ **ソルバーの実装** ソルバーの実装において、行数が136行と最も多い一方で、会話におけるラベル数はわずか23と少なかった点について考察する。この要因として以下が考えられる。

第1に、ソルバーは開発プロセスの最終段階で作成されたことが挙げられる。この段階では、ChatGPT の特性と機能に対する著者自身の理解が進んでいたた

め、より効率的にコードを生成できるようになっていた。

第2に、開発の方法を変更したことが挙げられる。具体的には、ソルバーの開発を開始する時点で、ソルバーのアルゴリズムは ChatGPT を用いずに検討済みであったため、「WHAT」から「HOW」を生成する方法は採らなかった。日本語で記述した「HOW」を ChatGPT に入力することで、適切な JavaScript コードを素早く生成することができた。

第3に、ソルバーの実装で冗長な部分がある点が挙げられる。ソルバーのコードで、ほとんど同じ内容の17行のコードが4個連結している部分がある。この部分は、適切にリファクタリングすることによって約1/4に短縮されるはずである。

4.3 会話に出現する単語の分析

一般に、人間による分析は主観性が介入する可能性が高く、その結果バイアスが生じることもある。この問題を軽減するために、自然言語処理の専門性を持つ生成 AI を利用して、会話に出現する単語を分析した。ここでは、ChatGPT（言語モデル：GPT-4）に単語頻度（表2および表3に示さなかった、すべての単語を含む）を分析させた結果を援用する。

以下が ChatGPT による単語の分析結果である。

技術的な内容 両者の発話に「ボタン」「セル」「入力」「数字」「表示」などの単語が多く登場しているので、おそらくこの会話はプログラミングやコンピュータのインターフェースに関する話題であると推測される。「関数」「コード」「生成」「イベント」「配列」などのプログラミングに関連する用語も多い。

問題解決 「場合」「修正」「必要」「問題」などの単語が多く使われていることから、特定の問題を解決するための手法や方針について話されている可能性がある。

互いの理解と教え合い 人間側の「くださる」や ChatGPT 側の「おる」などの敬語や丁寧な表現が見られるので、両者は教え合い、理解し合う形のコミュニケーションを取っているようである。

アクション指向 人間と ChatGPT 両方で「する」が圧倒的に多く使われている。これは会話が行動指向であることを示している。

感情や評価 形容詞には「よい」「ない」「ほしい」「うまい」「正しい」などがあるため、ただ事実を述べるだけでなく、評価や希望、期待も話題になっている可能性がある。

ChatGPT の発言の多様性 ChatGPT の方が使用する動詞や名詞の種類が多いようだ。これは ChatGPT が多くの観点や解説を提供している可能性を示している。

著者は、会話の当事者として上記の分析結果が妥当であると判断する。

4.4 コード生成 AI の改善に向けての提言

この節では、プログラミングコードの生成能力を以下のように段階化する。

- **レベル 1:** 文法ミスやバグを含むコードから、正確なコードを生成できる能力。
- **レベル 2:** 自然言語で表現された「HOW」に基づいて、特定のプログラミング言語でのコード（すなわち「HOW」）を生成できる能力。
- **レベル 3:** 小規模なアプリケーション開発において、「WHAT」から「HOW」を生成できる能力。
- **レベル 4:** 中規模から大規模なアプリケーション開発において、「WHAT」から「HOW」を生成できる能力。

2023 年 8 月現在の ChatGPT（GPT-3.5 および GPT-4）は、レベル 3 までの能力は有しているが、レベル 4 には到達していないと判断する。さらに、人間とのコミュニケーションが不自然であるため、効率的なコード生成に支障をきたしている。

この問題を解決し、さらにレベル 4 に到達するための具体的な提言を以下にまとめる。

1. **文脈の継続性の確保と適切な応答:** 2022 年 11 月に ChatGPT が登場した段階で文脈の保持はある程度達成されていたが、高度なコード生成には不十分である。具体的には、「問題があるコードに修正 A を施して動くコードが得られる。その後、別の不具合に対して修正 B を施すと、修正 A を施す前のコードに対して修正 B が施された状態になる（すなわち、修正 A に対応するコードが消えてしまう）」ことが頻繁に生じる。これは、漸進的な

コード開発を阻み、高度なコード生成をする上で大きな障害となる。文脈を適切に保持し応答能力を向上させることによって、より効率的なコード生成が実現されるだろう。

2. **方針変更の理由の明示:** FunctorX の開発において、ChatGPT がそれまでの議論を無視して突如として新しいアプローチでコードを生成したことがあった。はじめは戸惑い、不快な思いをしたが、その新しいアプローチには一考の価値があり、最終的には成功につながった。このような方針変更について明示的な説明があれば、人間側がよりスムーズに受け入れられるだろう。
3. **過度な謝罪の制限:** ChatGPT が繰り返し過度に謝罪する傾向にある。この行動は、たとえば軽微な修正依頼や質問に対しても先に謝罪が行われることから明らかである。このような過度な謝罪は、卑屈な印象を与えるだけでなく、人間に罪悪感を引き起こし、それが逆にコミュニケーションの効率を低下させている。この点においては、AI の対人スキルの向上が必要であると言える。
4. **意味の正確な把握:** 著者はある状況でコードの一部分について技術的な質問をしたが、ChatGPT はそれを修正が必要な要求であると誤解し、修正版を生成したことがあった。しかし、その生成されたコードは修正前と全く同一であった。コード生成 AI は、要求されていることを正しく理解するとともに、過去に生成した内容と生成しようとする内容の整合性を理解する能力を向上させる必要がある。

以上の提言は、FunctorX の開発を通じて得られた知見に基づいており、レベル 4 に到達するために必要な機能を示すことを意図している。本論文は、これらの提言だけでレベル 4 に到達可能であると主張するものではなく、さらに、レベル 4 に到達するために他の実現すべき機能を排除するものでもない。2023 年 7 月に ChatGPT の新しい機能として登場した Code Interpreter^{注3)}においては、ユーザからの自然言語による要求からコードを生成し、対話的にデータ分析を進めていく。この方向性が示唆しているように、プログラミング開発においても、自然言語による要件定義のみで目的のソフトウェアが生成されることが期待される。この長期的な目標を達成するために、著者はユーザ側への対応を要求することは考えておらず、技

術的な進歩によるソリューション、すなわち生成 AI 側の改良にのみ焦点を当てている。

5 関連研究

本節では、本研究の文脈と関連する文献について述べる。

コード生成 AI と人間との関係性について、Sarkar ら¹⁵⁾ は、コード生成 AI を利用したプログラミングが従来のプログラマ支援手法と異なる点と共通している点を調査し、「コード生成 AI を利用したプログラミングは、独自の特性と課題を持つ新しいプログラミング手法である」と結論付けている。Liu ら¹⁶⁾ はプログラマによる自然言語の指示と生成されるコードとの間のギャップを埋める手法に焦点を当てている。Barke ら¹⁷⁾ は、GitHub Copilot¹⁰⁾ とプログラマとの会話を解析し、使用モードには加速モードと探索モードの 2 種類が存在することを明らかにしている。Wu ら¹⁸⁾ は、人間と AI が協力するペアプログラミングと、従来の人間同士のペアプログラミングを比較している。Cheng ら¹⁹⁾ はソフトウェア開発者がコード生成 AI をどのように受け入れるかについて調査している。Murillo ら²⁰⁾ はコード生成 AI がソフトウェア開発者の作業方法をどのように変えているかに焦点を当てている。Wang ら²¹⁾ はコード生成 AI に対するソフトウェア開発者の信頼を調査し、それらの間のインターフェースをどのように設計すべきかを議論している。McNutt ら²²⁾ は、Jupyter などの計算ノートブックで用いるためのコード生成 AI をどう設計すべきかについて議論している。

生成されたコードの検査とソフトウェアのテストに関する研究として、Ferdowsi ら²³⁾ は、生成されたコードを検査するためのエンドユーザ向けのスプレッドシートツールを提案している。Kang ら²⁴⁾ は、バグ報告からテストケースを自動生成するフレームワーク、LIBRO を紹介している。Jesse ら²⁵⁾ は、GitHub Copilot¹⁰⁾ が生成する可能性のある単純で愚かなバグ (SStuB) について調査し、その回避策を提案している。

ソフトウェアの最適化への大規模言語モデルの応用として、Kang ら²⁶⁾ は遺伝的改良 (Genetic Improvement) と大規模言語モデルを組み合わせ、ソフトウェアを最適化する方法を探っている。

高性能コンピューティングへの応用として、Nichols

ら²⁷⁾ は並列プログラムに特化した大規模言語モデルを作成し、その効果を評価している。

各研究はそれぞれ異なる問題に対処しており、多様なアプローチが取られていることが分かる。また、多くの未解決の問題がこの分野には存在していると推測される。GitHub Copilot などのコード生成 AI は ChatGPT が登場する以前からソフトウェア開発者に利用されているため、その関連研究も多く存在している。しかし、会話型の AI である ChatGPT が市場に登場して以降、この分野の研究がさらに広がりを見せており、関連研究も今後さらに増えていくと思われる。

6 おわりに

本研究では、生成 AI である ChatGPT が、中規模のソフトウェア開発においてどの程度有用であるか、またその限界は何かを検討した。その結果、ChatGPT は小規模なプログラミングタスクにおいては有望な成果を示しているが、より大規模なプロジェクトでは設計からコード生成に至るまでの一貫性を維持するのが困難であり、人間の介入が必要であることが明らかとなった。また、本研究での数字パズル「FunctorX」の開発において、特に「実装: 数字入力部」と「実装: イベントリスナー」に関する会話が多く行われたことから、実装フェーズでの議論が集中していることが示唆された。

研究手法として、643 回に及んだ会話のログを取得し、それに「要件定義」、「設計」、各部の「実装」、「デプロイメント」など、16 種のラベルを付与した。これにより、ChatGPT が各開発段階でどの程度有用または問題をもたらすのかを評価した。

さらに、コード生成 AI を改善するためのいくつかの提言を行った。具体的には、文脈の継続性の確保、方針変更の理由の明示、過度な謝罪の制限、意味の正確な把握である。これらの改善が実現すれば、AI の効果性と精度が大幅に向上し、ソフトウェア開発環境でのコード生成 AI の価値がさらに高まると考えられる。

今後の課題としては、本研究において会話へのラベル付けは著者の主観に基づいて行われたが、将来的には機械学習や統計的手法を用いて、より客観的な評価を可能とすることが挙げられる。さらに、本研究では著者自身のソフトウェア開発のみに焦点を当てたが、単数あるいは複数の開発者による、より多くのソフトウェア開発を取り上げることで、生成 AI による支援が

ユーザ側にもたらした変化についても注目し、プログラミング言語の習熟度、開発の各段階におけるスキルの成長、生成 AI との会話の洗練化といった側面について評価したいと考えている。

注

注1) 著者は、過去の研究において分散環境のための並列プログラミングの実行環境に関する研究に取り組んだことがある。具体的には、C で書かれた OpenMP プログラムを処理するコンパイラや、分散環境における実行時ライブラリを開発した。この過程において、C やネットワークプログラミングに関する知識を得た。また、指導する学生の卒業研究の準備として、Objective-C を用いた iPhone 用ゲームを開発した経験も有している。近年では、日常業務を自動化するための自作スクリプトを記述するために、Python を用いることが多い。これらの経験は、本研究で取り組んだ JavaScript に関するスキルとは異なるものの、プログラミング全般に対する理解を深めるのに貢献したと考えられる。

注2) ここでの「より洗練された指示」とは、解決方法のアルゴリズム (HOW) を自然言語で与えたという意味ではない。具体的には、ChatGPT が示したコードに不具合がある場合に、ただ「動きません」と伝えるのではなく、不具合のある箇所を具体的に指摘することで、ChatGPT の次のアクションを容易にすることができたことを意味している。さらに、「ChatGPT が扱いやすい問題の大きさ」を感覚的に修得することで、有用な返答が得られやすい質問をするためのユーザ側の技能が向上したことを意味している。

注3) Code Interpreter は、ユーザの要求に応じて、ChatGPT 内で Python のコードを生成・実行する機能である。その後、2023年8月に Advanced Data Analysis と名称を変更し、さらに2023年12月時点の ChatGPT では、その名称を前面に押し出さずに、GPT-4のデフォルトの機能として埋め込まれた状態となっている。

参考文献

- 1) OpenAI, ChatGPT,
<https://chat.openai.com/>, 2023. 12. 12.
- 2) Google AI, Bard,
<https://bard.google.com/>, 2023. 12. 12.
- 3) Adobe, Adobe Firefly,
<https://www.adobe.com/sensei/generative-ai/firefly.html>, 2023. 12.12.
- 4) OpenAI, DALL-E 3,
<https://openai.com/dall-e-3>, 2023. 12. 12.
- 5) A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, 2022, Hierarchical text-conditional image generation with clip latents. arXiv preprint, arXiv:2204.06125.
- 6) A. Agostinelli, T. I. Denk, Z. Borsos, J. Engel, M. Verzetti, A. Caillon, Q. Huang, A. Jansen, A. Roberts, M. Tagliasacchi, and others, 2023, MusicLM: Generating music from text. arXiv preprint, arXiv:2301.11325.
- 7) RIFFIT, SongR,
<https://www.songr.ai/>, 2023. 12. 12.
- 8) J. I. Saadi and M. C. Yang, 2023, Generative Design: Reframing the Role of the Designer in Early-Stage Design Process. Journal of Mechanical Design, 145, 4, 041411.
- 9) N. Yüksel, H. R. Börklü, H. K. Sezer, and O. E. Canyurt, 2023, Review of artificial intelligence applications in engineering design perspective. Engineering Applications of Artificial Intelligence, 118, 105697.
- 10) GitHub and OpenAI, GitHub Copilot,
<https://github.com/features/copilot/>, 2023. 12. 12.
- 11) Amazon, Amazon CodeWhisperer,
<https://aws.amazon.com/codewhisperer/>, 2023. 12. 12.
- 12) N. Yonezawa, FunctorX,
<https://functorx.com/>, 2023. 12. 12.
- 13) D. Tait, Sumplete,
<https://sumplete.com/>, 2023. 12. 12.
- 14) T. Kudo, K. Yamamoto, and Y. Matsumoto, 2004, Applying conditional random fields to Japanese morphological analysis. Proceedings of the 2004

- conference on empirical methods in natural language processing, 230-237.
- 15) A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, 2022, What is it like to program with artificial intelligence?. arXiv preprint, arXiv:2208.06213.
 - 16) M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon, 2023, "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, 1-31.
 - 17) S. Barke, M. B. James, and N. Polikarpova, 2023, Grounded copilot: How programmers interact with code-generating models. Proceedings of the ACM on Programming Languages, 7, OOPSLA1, 85-111.
 - 18) T. Wu, K. Koedinger, and others, 2023, Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. arXiv preprint, arXiv:2306.05153.
 - 19) R. Cheng, R. Wang, T. Zimmermann, and D. Ford, 2022, "It would work for me too": How Online Communities Shape Software Developers' Trust in AI-Powered Code Generation Tools. arXiv preprint, arXiv:2212.03491.
 - 20) A. Murillo and S. D'Angelo, 2023, An Engineering Perspective on Writing Assistants for Productivity and Creative Code. Google Research, 1-2.
 - 21) R. Wang, R. Cheng, D. Ford, and T. Zimmermann, 2023, Investigating and Designing for Trust in AI-powered Code Generation Tools. arXiv preprint, arXiv:2305.11248.
 - 22) A. M. McNutt, C. Wang, R. A. Deline, and S. M. Drucker, 2023, On the design of ai-powered code assistants for notebooks. Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, 1-16.
 - 23) K. Ferdowsi, J. Williams, I. Drosos, A. Gordon, C. Negreanu, N. Polikarpova, A. Sarkar, and B. Zorn, 2023, ColDeco: An End User Spreadsheet Inspection Tool for AI-Generated Code. Microsoft Research, Tech. Rep., <https://www.microsoft.com/en-us/research/publication/coldeco-an-end-user-spreadsheet-inspection-tool-for-ai-generated-code/>.
 - 24) S. Kang, J. Yoon, and S. Yoo, 2023, Large language models are few-shot testers: Exploring llm-based general bug reproduction. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2312-2323.
 - 25) K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, 2023, Large Language Models and Simple, Stupid Bugs. arXiv preprint, arXiv:2303.11455.
 - 26) S. Kang and S. Yoo, 2023, Towards Objective-Tailored Genetic Improvement Through Large Language Models. arXiv preprint, arXiv:2304.09386.
 - 27) D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, 2023, Modeling Parallel Programs using Large Language Models. arXiv preprint, arXiv:2306.17281.

