BUSINESS THINKER ビジネスシンカー
暮らしを変える、明日を変える
株式会社エル・ローズ
January 1 2026

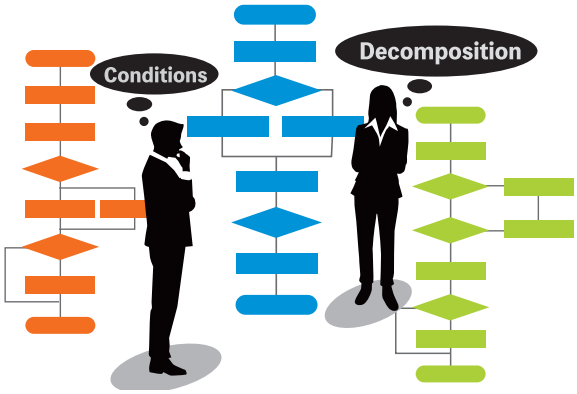
コードを書かずに、思考を実装せよ。
AI時代にビジネスパーソンが手に入れるべき
「構造化」という武器

「プログラミング的思考」の正体

Contents

AIが進化する時代において、ビジネスパーソンの役割は根本的に変わりつつある。 これまでは、自ら汗をかいてタスクを処理する「プレイヤー」「オペレーター」の能力が評価されてきた。これからは、仕事というシステム全体を設計し、AIや部下というリソースを組み合わせて動かす「プログラマー（設計者）」としての能力が問われる時代になる。

そこで求められるのが「プログラミング的思考」である。ただでさえ、IT化、デジタル化、さらにはハラスメント対策でコミュニケーションに齟齬が生まれつつある企業社会において、プログラミング思考が大切などというと、「もううんざり」という声も聞こえて来そうだ。だが逆だ。プログラミング的思考は、不幸なコミュニケーションを減らし、誰もが迷わずに成果を出せるようにするための、「究極の優しさ（ユーザビリティ）」の設計技術なのだ。



昭和世代の指示は若手に通じず、AIにも無視されるのか？

「いい感じにまとめておいて」。 昭和世代の上司が放つ何気ない一言。これに対し、令和世代の部下は内心で頭を抱えている。「『いい感じ』の定義は何ですか？」「フォーマットはありますか？」「何時までですか？」。

上司は嘆く。「最近の若いのは、行間も読めないのか。俺たちの頃は、上司の背中を見て悟ったものだが……」。

一方、部下は陰で呟く。「指示が曖昧すぎるんだ。これじゃAIにも投げられないし、手戻りが怖くて動けない。これだから昭和のオジサンは……」。

このすれ違いは、笑い話では済まされない。ビジネス現場で起きている混乱の正体は個人の能力不足ではなく、使っている「思考のOS（オペレーティング・システム）」の不一致にあるからだ。さらに深刻なのは、AI活用への

影響だ。AIは令和世代以上に「空気」を読まない。「よしなに」と入力しても、AIは幻覚（ハルシネーション）を見るか、沈黙するだけだ。指示が論理的でなければ、AIという最強のツールもただの箱に過ぎない。

「昭和OS」と「令和OS」の思考グセの正体

こうしたギャップを解き明かす前に、まず、私たちが無意識に使っている「思考のクセ」を客観視してみよう。どちらが良い悪いではない。「昭和OS」と「令和OS」は、どちらも名作だが、ファイル形式が違うのだ。どんな違いがあるのか。少し抽出してみよう。

まず昭和世代、とくにバブル期や就職氷河期を生き抜いてきたビジネスパーソンの思考は、極めて「属人的」で「統合的」だ。その特徴は、次のようにまとめることができそうだ。

特徴①変数が定義されていない（Undefined Variables）：「誠意を見せろ」「常識で考えろ」「よしなに頼む」。これらの言葉は、受け手の解釈に依存する「未定義の変数」だ。昭和世代同士なら「共通の辞書」があるため通じるが、他者にはエラーとなる。

特徴②例外処理が現場任せ（Ad-hoc Exception Handling）：「何かあったら臨機応変に」。これはシステム設計で言えば「エラーが起きたら、その場の担当者がなんとかコードを書き換えて対処せよ」という指示に等しい。現場力はつくが、再現性がない。

特徴③ブラックボックス化したアルゴリズム：「長年の勘」や「コツ」という言葉で、プロセスを隠蔽する。入力（Input）に対してなぜその出力（Output）が出たのか、本人にも説明できないことが多い。

一方、デジタルネイティブである令和世代。その思考は、極めて「機能的」で「分割的」だ。

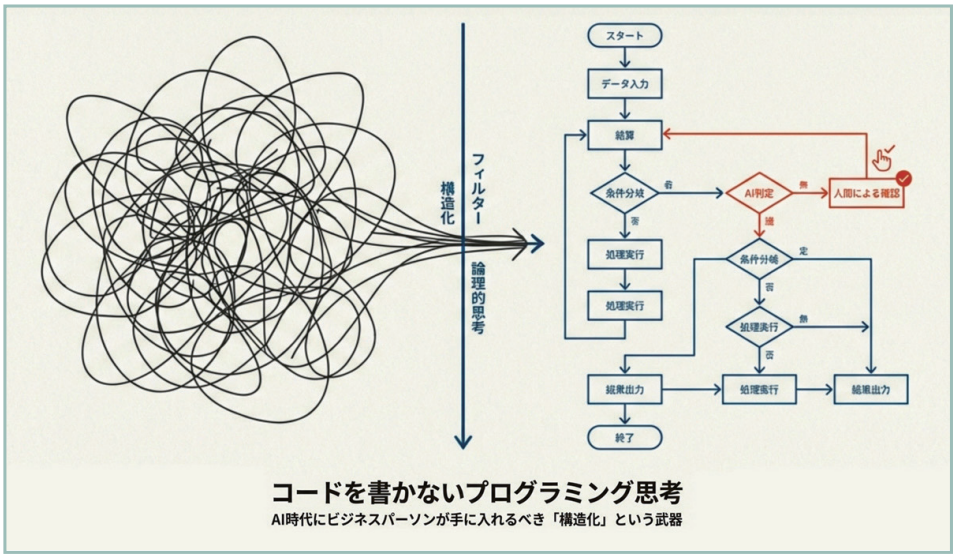
特徴①　構文エラーに厳しい（Syntax Error Strict）：指示に論理的な矛盾や欠落があると、そこで処理を停止する。「Aと言ったのにBとも言っている。どちらが正解ですか?」と、バグを許容しない。

特徴②　ライブラリ依存（Library Dependency）：「マニュアル（ライブラリ）はありますか?」。ゼロから構築するよりも、既存の正解や型（フレームワーク）を呼び出して使うことを好む。効率的だが、型がない事態に弱い。

特徴③　局所最適化（Local Optimization）：与えられたタスク（関数）は完璧にこなすが、それがシステム全体（会社全体）でどう機能するかへの関心が薄い傾向がある。「言われたことはやりました（結果はどうあれ）」というスタンスになりがちだ。

共通言語としての「プログラミング的思考」

この2つの違うOSを接続させるには、「翻訳プロトコル」が必要になる。そしてこの翻訳行為が「構造化」である。すなわち昭和世代の持つ「豊かな文脈と経験」を、令和世代が理解できる「明確なロジックと手順」に変換するのである。この作業ができれば、組織の生産性は劇的に向上し、AIへの指示（プロンプト）も驚くほど高精度になる。



コードを書かないプログラミング思考
AI時代にビジネスパーソンが手に入れるべき「構造化」という武器

これは昭和世代に限ったことではない。上の図の左側にある絡まり合った糸のような状態が、多くのビジネスパーソンの頭の中だ。どんな企業人も新人時代は新しいタスクの連続に慌てふためき、ときに失敗を繰り返してきた。そしていつしか一見複雑そうなプロセスをいとも軽々しくこなせるようになっていく。彼ら彼女らはいつしかベテランや中堅と呼ばれ、複数の部下を持つリーダーやマネージャーとなっている。

武道や茶道の用語に「守破離」という言葉がある。経験の浅いうちはその基本型をひたすら忠実に守ることに徹し、型を身につけた後はその型を破って、応用・発展させ、いずれ基本から離れて独自の境地を切り開くという発展のフェーズを表した言葉だ。守破離はこうした習いごとに限らず、仕事の基本的な習得プロセスでも見られる。最初は頭をフル回転させながら、動作を繰り返すが、型が習得できればいずれ「無意識」に体が動くようになる。仕事や作業というものはすべからく「無意識」に動かせる、あるいは動くようになって「ベテラン」や「達人」と呼ばれ

るようになる。だからベテランが「なんとなく」と指示を出すことは、決して仕事を理解していないということではなく、言語化するまでもなく、体得している、ということなのである。

だが今の時代、これではAIに指示が出せないどころか、業務の自動化も、他人への引き継ぎもままならない。

右側にあるのが、プログラミング的思考によって、その「なんとなく」の仕事が整理された状態だ。入力があり、判断があり、処理があり、出力がある。一直線に整理されたフローは、美しく、明確だ。

プログラミングの世界では、曖昧さはバグ（不具合）を生む。ビジネスも同様だ。指示が曖昧であれば、組織にバグが生じ、ミスや手戻りというコストが発生する。AI時代におけるビジネスパーソンの第一の責務は、自らの頭の中にある「経験」や「勘」というブラックボックスを開き、それを万人が理解できる「アルゴリズム（手順）」に書き換えることにある。

「分解の粒度」を揃える ―― 業務を「原子レベル」まで砕く

こうした組織にプログラミング的思考を定着させるための第一歩が、「タスク分解（Decomposition）」である。ここで問題になるのが「解像度（かいぞうど）」あるいは「粒度（りゅうど）」である。ときに上司にとっての「1つのタスク」が、部下にとっては「100のタスク」に見えることがある。この粒度のズレが、不幸なすれ違いを生む。

1）「チャンク（塊）」を「プロセス（処理）」に割る

とかく昭和世代は業務を大きな「塊」で捉えがちだ。だから下の世代からは“雑”な指示に見えてしまう。例を挙げてみよう。

昭和の上司がこう言ったとする。「A社向けの提案書を作って」。

これは典型的な「巨大チャンク」だ。令和世代の脳内パソコンには、まるっと「A社向け提案書を作る」というアプリはインストールされていないのだ。だから「処理しますか?」ではなく「そのアプリはお使いのPCには入っていません」と表示されて固まってしまうのである。

しかし構造化された指示だとこうなる。

①「A社の過去の取引データを抽出する（Input）」
②「競合B社の動向をWebでリサーチする（Process）」
③「当社の新商品のメリットを3点箇条書きにする（Process）」
④「パワーポイントの社内テンプレート『タイプC』に流し込む（Output）」

このように、チャンクを「動詞」と「目的語」のセットに分解してみる。よりプログラミング的に言えば、巨大な関数を、小さな関数（サブモジュール）に分割する作業である。

2）粒度を揃える「15分ルール」

では、どこまで細かくすればいいのだろう。目安となるのが「15分ルール」だ。つまり「1つのタスクが15分～30分で完了するサイズ」まで分解するのだ。

たとえば「企画を考えて」は、ラーメン屋で「なんかおいしいの出して」と言うのと同じくらい粒度が粗い注文だ。店側としては、「しょうゆなのか、味噌なのか、それともチャーハンなのか」くらいは知りたいところだ。

一方、企画のプロセスを分解し「類似事例を5つ検索する」（15分）とすれば、適切な粒度と言える。

このサイズまで分解されていれば、令和世代の部下は迷わず着手でき、小さな達成感（Small Wins）を積み重ねることができる。そして、この粒度であれば「この15分の作業はAIに任せられるな」という判断も容易になるのである。

3）ゴールの解像度を上げる「定義（Definition）」

「完了」とはどういう状態か、を定義する。

- ×「なる早で」→ ○「明日の14時までに」
- ×「いい感じに」→ ○「PDF形式で、A4版1枚に収まるように」
- ×「徹底的に」→ ○「Google検索の検索結果3ページ目まで確認して」

昭和世代の「形容詞（早い、すごい、丁寧な）」を、令和世代の「数値・形式（時間、フォーマット、数量）」に変換する。これが、粒度を揃えるということだ。

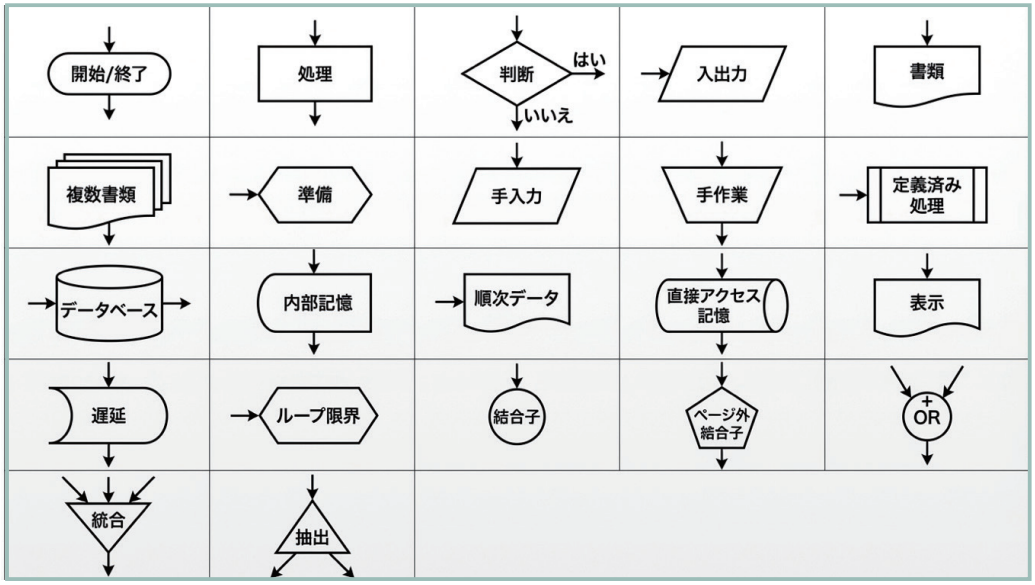
思考の解像度を上げる「7つの論理パターン」

では、具体的にどうすれば思考を構造化できるのか。実は、どんなに複雑そうに見える巨大システムも、職人の「門外不出のコツ」も、分解してみると、だいたい同じパターンの寄せ集めだ。カレーもラーメンも、ベースは「切る・炒める・煮る」の繰り返しなのと、そんなに変わらない。

ここでは、ビジネスパーソンが武器として持つべきプログラミングの「基本パターン」と、「7つの論理パターン」を紹介する。これらはプログラミングの基礎文法であるが、同時にビジネスを動かすための共通言語でもある。

プログラミングはそもそも大量の同じパターンの作業を効率よく処理するために生まれた、計算機械のための手法。

図A

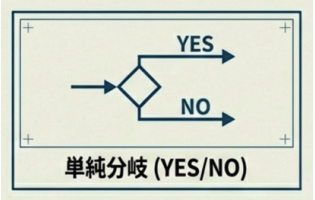


何かを処理するのだから、処理の開始と終わりがある。その間の処理の方法を書き表す言語がプログラミング言語である。いま巷にはさまざまなプログラミング言語がひしめいているが、それぞれ得手不得手がある。厳密に言えば、プログラミング言語に不得手はない。ただ同じ作業をさせるのに時間がかかる言語とそうでもない言語が出てくるだけ。どの言語に何をさせるかは、指示者の判断となるが、いずれも基本的な処理手順は同じだ。その手順を図式で描いたのがフローチャートである。

フローチャートには、処理作業に応じたレゴブロックのような記号が決められており、この“レゴブロック”の組み合わせで、処理手順が可視化される。この“レゴブロック”にはざっと上記図のような記号がある。（図A）

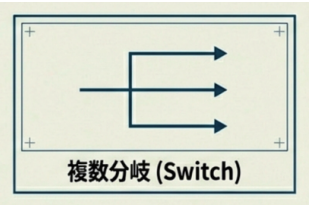
それなりの数だが、基本は次の3つだ。開始と処理、そして終了である。つまりビジネスにおいては、「どの記号で処理を行うか」が重要になる。そしてその処理は次の「判断・分岐」と「繰り返し（ループ）」の組み合わせで実行できる。

①単純分岐（IF / ELSE）
── 判断の最小単位



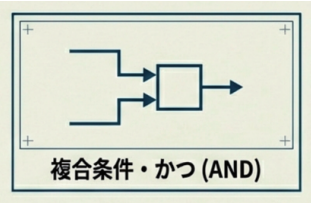
すべての判断の基本形が「単純分岐」だ。「もし条件Aを満たすならXをする。そうでなければYをする」。例えば、「納期まで3日以上あるか?」という問いに対し、YESなら「通常配送」、NOなら「速達配送」を選ぶ。昭和のビジネスパーソンはこれを無意識に行ってきたが、AIに渡すにはこの「分岐点（閾値）」を言語化しなければならない。「急ぎなら」という言葉は通用しない。「何時間以内が急ぎなのか」を定義することから、構造化は始まる。

②複数分岐（Switch）
──「ケースバイケース」の正体



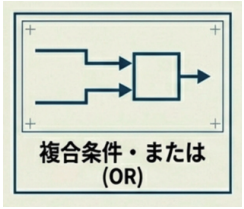
「状況による」という言葉で思考停止していないだろうか。それを分解するのがこのパターンだ。たとえば顧客のランクが「VIP」か「一般」か「新規」かによって、対応を変えるとしよう。このとき重要なのは、想定されるすべてのケースを洗い出し、そのどれにも当てはまらない「その他（Default）」の処理を決めておくことだ。ここが抜けていると、想定外の事態が起きたときにAIはフリーズし、現場はパニックに陥る。

③複合条件（AND / OR）
── リスク管理の要



「金額が100万円以上」かつ（AND）「初回の取引」ならば、部長決裁が必要。「クレーム履歴がある」または（OR）「担当者が不在」ならば、マネージャーに転送。このように複数の条件を組み合わせる際、AND（かつ）なのかOR（または）なのかを厳密に区別することは、リスク管理そのものだ。特にOR条件は、どちらか1つでも当てはまれば発動するため、安全策やアラート機能として使われることが多い。この論理が曖昧だと、重大なコンプライアンス違反や事故につながる。

④全件処理（For Loop＜ループ＞）
――「忙しさ」の正体を暴く



リストにある100件の顧客すべてにメールを送る。月末に100枚の請求書を処理する――このように「決まった回数だけ同じ処理を繰り返す」のがForループだ。多くのビジネスパーソンが「忙しい」と感じている業務の正体は、実はこのループ処理を人間が手作業で行っていることにある。「毎日同じことを繰り返している」と気づいたら、それはあなたのやるべき仕事ではない。ループ処理こそ、コンピュータが最も得意とし、疲れを知らずに正確にこなせる領域だ。ここを自動化できるかどうかが、生産性の分水嶺となる。

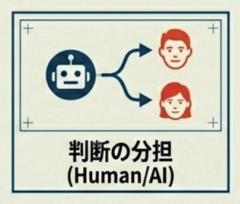
⑤条件付き反復（While Loop＜ループ＞）
――終わりのない仕事に終止符を打つ

「顧客が納得するまで説明する」「品質基準を満たすまで修正する」。Forループと違い、回数が決まっておらず「条件が満たされるまで繰り返す」のがWhileループだ。ここはブラック労働の温床になりやすい。「納得するまで」とは具体的にどの状態か？ 無限ループに陥らないための「脱出条件（Exit）」は設定されているか？ 管理職は、部下の業務が無限ループに入らないよう、この構造を設計する責任がある。

⑥例外処理（Try / Catch）
――プロの仕事はここに宿る

システムエラー、配送の遅延、顧客の急なキャンセル……。「通常ルート」から外れたときにどう動くか。これをあらかじめ設計しておくのが例外処理だ。未熟な設計者は「うまくいった場合」のルートしか考えない。しかし、現実のビジネスは想定外の連続だ。「もしエラーが起きたら（Catch）、サポートセンターに通知し、顧客にはお詫びメールを自動送信する」。このように失敗時の挙動を定義しておくことで、システム（および組織）は止まることなく動き続けることができる。AIは「想定外」に弱い。だからこそ、ここを人間が設計する必要があるのだ。

⑦人間とAIの分担判定



そして最後にこれからの時代に不可欠なのが「この判断はAIに任せるか、人間がやるか」というメタな分岐。現代の最重要分岐である。

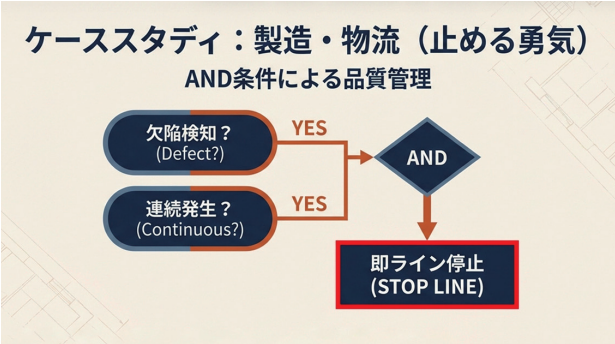
ここのポイントは「ルール化でき、過去のデータがあり、責任の所在が明確」ならばAI。「倫理的判断が必要で、共感が求められ、最終責任を負う必要がある」ならば人間。この仕分けができる人こそが、AI時代のリテラシーを持ったリーダーと言える。

7つの論理の基本は「判断・分岐」だ。そしてどうなったら作業を終えられるのかを設定することだ。

業界別・「条件分岐」のツボ

ではこれらの論理パターンは、実際の現場でどのように使われているのだろうか。業界別のケーススタディで見てみよう。そこには、各業界のプロフェッショナルたちが無意識に行ってきた「思考の型」が見えてくる。

①製造業：品質を守る「止める勇気」の論理化として

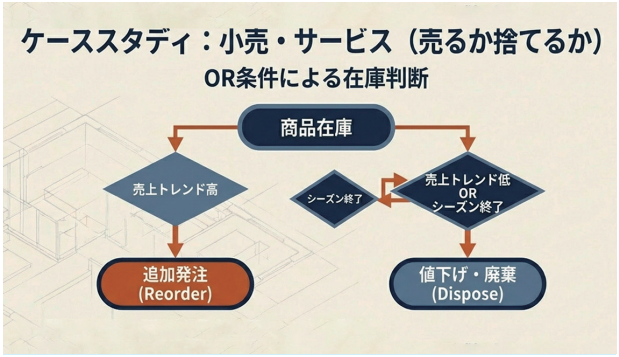


製造ラインにおいて最も重要な判断は「いつラインを止めるか」だ。

ここでは「単純分岐」と「AND条件」が組み合わさっている。

「不良品が出たか（YES/NO）」だけでなく、「連続して発生したか（AND）」という条件を加えることで、偶発的なエラーとシステム的な欠陥を区別している。「なんとなくおかしい」で止めていては生産性が落ちる。しかし止めなければ大量の不良品が出る。このギリギリの判断基準（閾値）を「3回連続したら停止」と数値化・構造化することで、はじめて自動監視システムや画像認識AIに判断を委譲できる。トヨタの「アンドン」システムは、まさにこの例外処理を物理的な仕組みに落とし込んだ好例と言える。

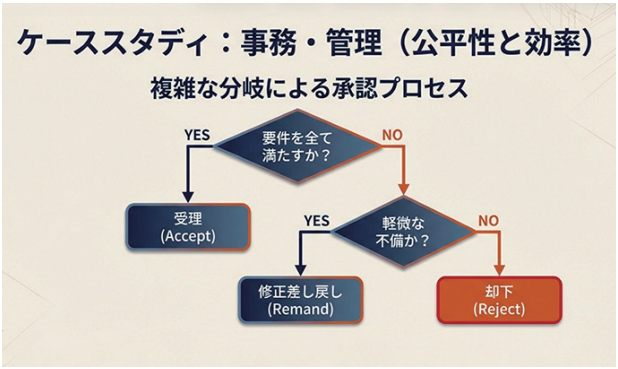
②小売・サービス業：機会損失を防ぐ「在庫」の論理



「売れているのに在庫がない」「売れないのに在庫がある」――小売の悲劇は、判断の遅れから生まれる。

ここでは「複数分岐」が鍵となる。売上が「好調」なら追加発注、「不調」かつ「シーズン終了間近」なら値下げ、「不調」だが「シーズン序盤」なら陳列変更。熟練の店長はこれを肌感覚で行うが、それでは多店舗展開ができない。ユニクロやコンビニエンスストアの強さは、この「発注・値下げのアルゴリズム」がシステム化され、全店で高速に回っている点にある。1週間の判断遅れが利益を消し飛ぶ世界では、思考の構造化は生存戦略そのものだ。

③管理・事務：組織の信頼を作る「ブレない」論理



経費精算や稟議承認。ここでは「公平性」という名の「一貫性」が求められる。「あの人の申請は通るのに、私のは通らない――」。こうした不満は、判断ロジックがブラックボックス化していることから生まれる。人間が判断すると、どうしても「まあ今回は特別に」というノイズが入ってしまう。そこで「予算内なら自動承認」「超過なら差し戻し」というシンプルな「IF/ELSE」を徹底し、システムに実装することで、組織の透明性は高まる。公平であるべき事務処理こそ、感情を持たないアルゴリズムに任せるべき領域である。

プログラミング的思考こそが「最強のプロンプト」である

ここまで解説した「構造化」のスキルは、生成AIへの指示出し（プロンプト）において、そのまま使える。なぜなら、AIが使用する大規模言語モデル（LLM）は、確率的に次の言葉を予測するマシンであり、論理的な制約（ガイドレール）を与えない限り、もっともらしい嘘（ハルシネーション）をついたり、平凡な回答に終始したりするからだ。よくAIが嘘を吐くと言って騙された気分になっているのは、前提の条件と、このガイドレールの設定が曖昧だからなのだ。たとえば単純に「日本の総理大臣は?」とだけ尋ねた場合、質問者の意図や前提条件を共有していないAIは、「いま現在の総理」以外の名前を返すこともある。

これは、たとえば昭和世代のビジネスパーソンが、「電話をかけて」と若手に指示するときに、指でダイヤルを回すジェスチャーを見せたりするのと同じで、スマホ世代には意味の通じない行為なのである。

今、日本は、人口減少や移民問題、国防、防災、インフラ強化など、国家の基盤が揺らぐような大きな岐路に立っている。なかでも静かにかつ大きく広がっているのが、デジタルデバイドならぬ「AIデバイド」である。

日本の社会における生成AI利用は世界の後塵を拝している。総務省の情報通信白書（令和7年版：2025年公表）では、日本の個人の生成AI利用経験は 26.7%で、中国 81.2%、米国 68.8%、ドイツ 59.2%などと比べて圧倒的に低い。しかも、利用者は学生のほうが多く、社会人の利用は学生が約45%に対して、社会人が25%（デル・テクノロジーズ関連調査［2024年］など）と、社会人が少ない。つまり現役のビジネスパーソンが生成AIの利用法を身につけることは、あらゆる組織の喫緊の課題なのである。

おそらく、多くのビジネスパーソンは理解しているだろう。しかし積極的になれないのは、間接的な言い回しを特徴とする日本語に支えられてきた昭和的な曖昧さと、日本人の分厚い暗黙知がそのニーズを阻んでいるからと思われる。

プログラミング思考を身につけることは、昭和的な「あいまい指示」から、抜け出すきっかけを与え、自ずと構造化された「プログラミング的指示」に変化していく。たとえば次のように。

✕ 昭和OS的プロンプト（Bad Case）

「来月の新商品の販促キャンペーンの企画書を、いい感じに書いておいて。ターゲットは若者で、エモい感じで頼むよ」

この指示は、変数（Variables）が定義されておらず、解像度も低い。では、どこを構造化すべきか。

- 「いい感じ」とは？ → 評価関数が未定義。
- 「若者」とは？ → 変数の範囲（Scope）が広すぎる。Z世代か？ ミレニアル世代か？
- 「エモい」とは？ → 出力パラメータが主観的すぎる。

構造化プロンプト（Good Case）

対して、プログラミング的思考を持つリーダーは、こう指示する。

＃命令：あなたは「熟練のマーケティングプランナー」として振る舞ってください（役割の定義）。当社の新商品「完全栄養食クッキー」の販促企画案を作成してください（ゴールの設定）。

＃変数定義（Variables）

- **ターゲット**：都内に住む入社1～3年目の社会人女性。忙しくて朝食を抜きがち。
- **課題（Pain）**：健康は気になるが、自炊する時間も気力もない。
- **ゴール（KPI）**：コンビニでのトライアル購入数 1万個。

＃制約条件（Constraints）

- 予算は500万円以内。
- タレントは起用しない。
- 「エモい」の定義：「頑張りすぎない丁寧な暮らし」への憧れを刺激するトーン&マナー。

＃出力形式（Output Format）以下のフォーマットで出力すること。

1. キャッチコピー（3案）
2. 施策概要（SNSとリアル店舗の連動）
3. スケジュール（フェーズ分けすること）

ここの指示には、ここまで述べてきた論理パターンがすべて詰まっている。「役割」を定義し、「変数」に具体的な値を代入し、「制約条件」で例外を排除し、「出力形式」でフォーマットを指定している。

これは、一般言語で行うプログラミング（Natural Language Programming）である。

「AIを使いこなす力」とは、ITツールの操作スキルではなく、「自分の要望を論理的な要件定義書に落とし込む国語力」にほかならないのだ。

「解像度」が低いリーダーはAI時代に淘汰される

ここまで見てきたように、プログラミング的思考とは、業務をいかに「解像度高く」捉えることができるかにかかっている。「解像度」とは、画像のピクセル数のようなもの。解像度が低いリーダーは、業務を「営業活動」「事務処理」といった大きな塊（モジュール）でしか捉えられない。そのため、AIに指示を出す際も「売上を上げて」「効率化して」といった、実行不可能な命令しか出せない。

一方、解像度が高いリーダーは、業務を「顧客リストの抽出」「アポイントメールの送信」「商談」「見積書作成」といったプロセスに分解し、さらにそれぞれの工程における「判断基準（条件分岐）」や「例外対応」まで見えている。

AIは「具体的」で「論理的」な指示しか受け付けないから、業務の解像度が低い人間は、AIという強力なエンジンのアクセルを踏むことができなくなる。かつては「大雑把だが器の大きい親分肌」のリーダーがガンガンアクセルを吹かしていたかもしれないが、これからは違う。緻密に業務を設計し、メンバーとAIに最適なタスクを配分できる、言語化がうまい「設計者型」のリーダーが組織を成長させる。

「オブジェクト指向」で組織のマイクロマネジメントをなくす

このプログラミング的思考を身につけると、課題の抽出力が高まり、その解決手段を見出して、組織で動かしやすいような構造化がしやすくなる。そしてこのプログラミング思考を進めていくと、その先にある概念にたどり着く。それが「オブジェクト指向」である。オブジェクト指向を経営に応用すると、煩わしいマイクロマネジメントから解放されるようになる。

つまり、組織の誰もが、目的に対して自律的に動けるようになるのだ。

オブジェクト指向を身に着けた上司は、たとえばこんな指示を与える。

- **カプセル化（Encapsulation）**：「営業チーム」というオブジェクトに対し、「今月の売上目標1億円」というメッセージ（入力）だけを送る。その内部で「どうやってテレアポするか」「誰が訪問するか」という詳細な処理（メソッド）は、チーム内部にカプセル化され、外部からは干渉されない。
- **インターフェース（Interface）**：上司が気にするべきは、内部のプロセスではなく、「正しく成果物（Output）が出てくるか」というインターフェース（接続口）の設計だけだ。だから上司がすべきことは「月末にこのフォーマットで報告書が出てくれば、途中経過は問わない」と決めることだ。

つまり部下に「任せるのが怖い」上司が、「任せるのがうまい」上司に変わる。任せるのが怖い上司は、このインターフェース設計ができていない。

「何を（Input）」「いつまでに（期限）」「どんな形で（Output）」返してくれればOKか。ここさえ握れていれば、上司はプロジェクトの“鍋のフタ”を、いちいち開けに行かなくて済む。途中経過のたびに味見されるシチューほど、作っている側のテンションが下がるものはないのだから。

部下を「手足」ではなく、独立した機能を持つ「モジュール」として信頼し、接続する。これこそが、変化に強い「疎結合（Loose Coupling）」な組織をつくる鍵となる。

明日から始める「思考のプログラミング」

プログラミング思考を身にまとうためには別にコードを書く必要はない。ただすべきトレーニングは、自分の仕事を「フローチャート」にしてみることだ。

PowerPointや手書きのノートで構わない。うまく形が描けないという人も大丈夫だ。今なら1000円程度でフローチャートとテンプレートが手に入る。

手始めに朝起きてから会社に行くまでのルーティン、あるいは日々のメールチェックの手順を、四角（処理）とひし形（判断）で描いてみる。すると、驚くほど多くの「曖昧な判断」や「無駄なループ」が見つかるはずだ。

たとえば「メールを確認する」という処理の中に、「重要なら返信、不要なら削除」という分岐があるだろう。そこで解像度を上げるのだ。「重要」の定義は何か？ 送信者か？ タイトルか？と。

あるいは「資料を作る」という処理は、実は「情報を集める」「構成を考える」「執筆する」「推敲する」という複数の処理の連続ではないか、と。

このように思考を可視化・構造化していくプロセスこそが、プログラミング思考の獲得プロセスである。そして、一度構造化されてしまえば、その一部を「ChatGPTに任せる」「RPAで自動化する」という判断が容易になる。

AIという「黒船」は、私たちに「人間はどう思考すべきか」という根源的な問いを突きつけている。曖昧な「空気」の世界から抜け出し、鮮明な「論理」の世界へ。思考のOSをアップデートした者だけが、この進化の時代を楽しみ、新たな価値を創造できるのである。

【あなたの思考OSチェック】

- □ 指示を出すとき、「いつまでに」を明確にしている
- □「いい感じに」「よしなに」を使わない
- □ タスクを15-30分サイズに分解できる
- □ 例外処理（もし失敗したら）を事前に考えている
- □ AIに指示を出すとき、具体的な条件を書いている

【診断結果】

5個	プログラミング的思考の達人
3～4個	良好、さらなる向上の余地あり
1～2個	今日から意識して改善しましょう
0個	むしろチャンス。この記事が、あなたのOSアップデート版リリースノートになる。

参考

〈書籍〉●『アルゴリズム図鑑』石田保輝／宮崎修一［翔泳社］●『具体と抽象』細谷功［dZERO］●『イシューからはじめよ』安宅和人［英治出版］●『解像度を上げる』馬田隆明［英治出版］●『プログラミング的思考の授業』中島聡［SBクリエイティブ］●『トヨタの自工程完結』佐々木眞一［ダイヤモンド社］

〈WEB〉●経済産業省「リスキリングを通じたキャリアアップ支援事業」●IPA 独立行政法人 情報処理推進機構「DX白書」●Udemy Business「非エンジニアのためのプログラミング思考」講座●Harvard Business Review「AI時代のマネジメント」関連論文 ほか

POINT

- 昭和OS（文脈依存）と令和OS（マニュアル依存）の互換性エラーを解消するのが「プログラミング的思考（構造化）」である。
- 「タスク分解」の粒度は「15分でできること」「動詞＋目的語」まで細分化し、AIや若手が即実行可能な状態にする。
- 「IF/ELSE（条件分岐）」でベテランの勘をルール化し、「Switch（複数分岐）」でケースバイケースを撲滅する。
- 「Loop（繰り返し）」業務は人間の仕事ではない。根性論を捨てて自動化・AI化を断行せよ。
- 「Try/Catch（例外処理）」を設計することで、若手の心理的安全性を確保し、組織のリスク耐性を高める。
- 小売・営業・物流・医療など、あらゆる業界の「属人化」した業務は、論理パターンに当てはめることで「資産化」できる。
- AI時代の人間の役割は、業務を遂行することではなく、業務の構造を「設計（デザイン）」することにシフトする。