Rocket® software

基幹システムアプリケーション資産の可視化・最適化ソリューション

Version.2
Last updated: 2026 年 4 月

Rocket software

目次

各機能の詳細に関しては、製品マニュアルページからご利用になるバージョンを選択後、内容をご確認ください。
https://www.amc.rocketsoftware.co.jp/support/manuals.asp

1. はじめに

プログラミング言語である COBOL は、メインフレーム時代からの長い歴史を持つ基幹システムの中で企業のビジネスの根幹を支え続けてきました。現在も新たなビジネスを創出するアプリケーションの新規開発でも幅広く活用されています。

一方、長年のビジネス変化に対応し続けてきた COBOL アプリケーションは、度重なる変更・保守の結果、複雑さを増し、それに伴う弊害が発生することもあります。もっとも深刻な弊害は、仕様書が存在しない、もしくは存在していても現状とはマッチせず、これらを把握するために工数をかける必要があるという問題です。この状態では些細な変更ですら重大なデグレードを引き起こす危険性があります。そこまで深刻ではなくとも、複数の観点の異なる改修を施された結果のプログラムには、様々な品質上・性能上の問題点が潜在化することがあります。

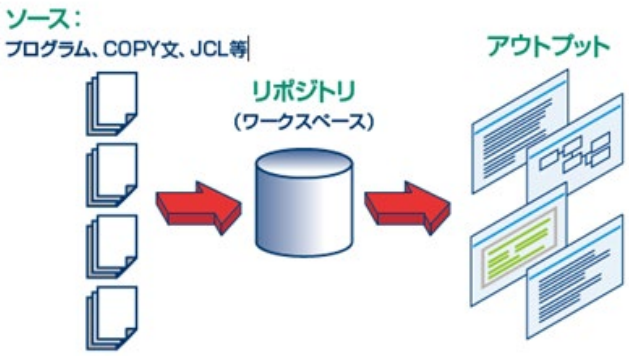
2. Enterprise Analyzer を活用した COBOL アプリケーションのブラッシュアップ

Enterprise Analyzer は、COBOL コードのリバースエンジニアリング・ツールとして、特にメインフレーム・アプリケーションに対して長い実績がある製品です。潜在的な品質・性能上の問題点を自動検出する機能など、多種多様な機能を提供しています。これらを活用することによって、現役で活躍しているアプリケーションのコードをブラッシュアップし、将来に渡りさらに活用して行くための準備を行うことができます。

本書ではこのような点に着目した Enterprise Analyzer の活用シナリオをご紹介します。

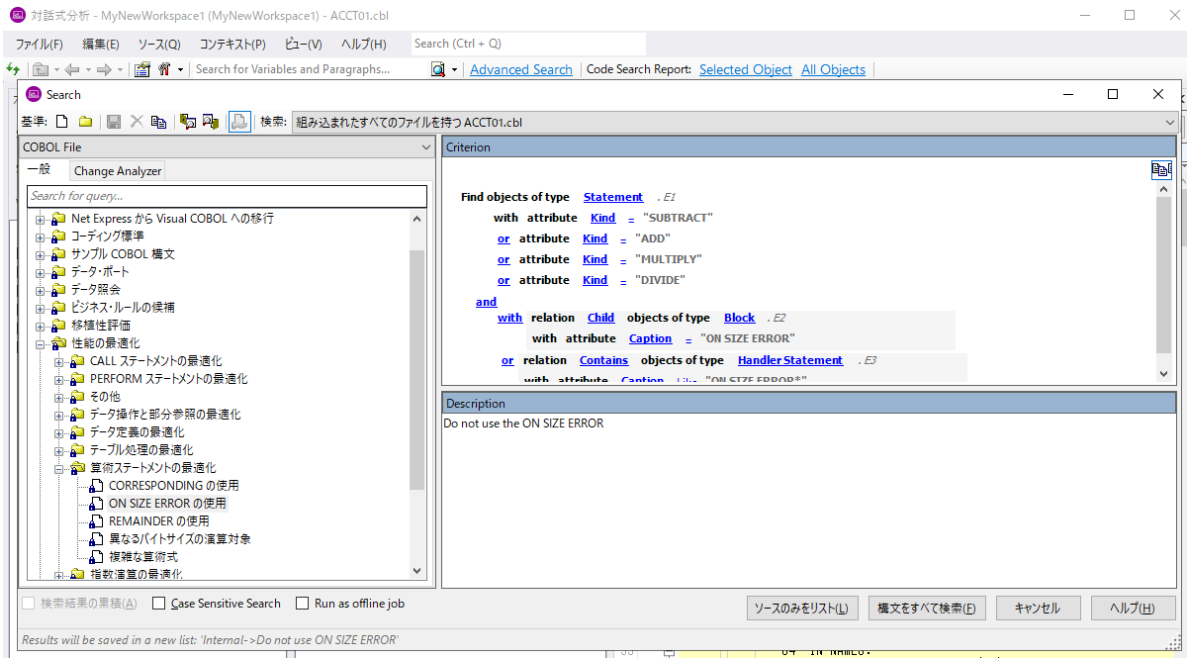
3. Enterprise Analyzer のアーキテクチャ

Enterprise Analyzer は、COBOL、PL/I などのプログラムソースのみならず、JCL、CICIS リソース定義、IMS システム生成マクロなどを総合的に構文解析し、その関連を正確にリポジトリ化します。このリポジトリを利用して、一覧表、グラフ、チャートなど様々な形式のアウトプットを生成します。



アウトプットの形式は、様々なカスタマイズ手段が用意されており、目的と用途に応じて最適な形に整理された情報を得ることができます。特に Clipper と呼ばれるクエリー機能は、COBOL 構文を解析した結果のデータベースに対して直接照会することができるため、膨大な量のソースコードの中からある固有のパターンを自動検出するのに役立ちます。

また、出荷時設定で定義済みの豊富なクエリーのライブラリが用意されていますが、以下のように固有の構文でスキーマに対する複雑な照会条件を柔軟に記述することができます。作成されたクエリーは、エクスポート・インポート機能を使用して他の Enterprise Analyzer ユーザーに展開することもできます。



以下に基本的なクエリー機能を示します：

- 項目名、プログラム名、ファイル名のパターンマッチ
- 項目、プログラム、ファイルの使用箇所と使用モード（照会・更新）
- 項目の長さ、十進桁数（整数部・小数部）、USAGE 種別
- 節、段落名のパターンマッチと、GO TO、PERFORM での使用箇所

Clipper が自動検出した問題箇所は、HTML や Excel シートの形式でレポートとして保存することができ、Enterprise Analyzer を使用しない方に配布可能な文書として再利用することができます。以下の章でこのクエリー機能を活用して抽出することができるプログラムの問題箇所の例を解説します。

4. 品質向上

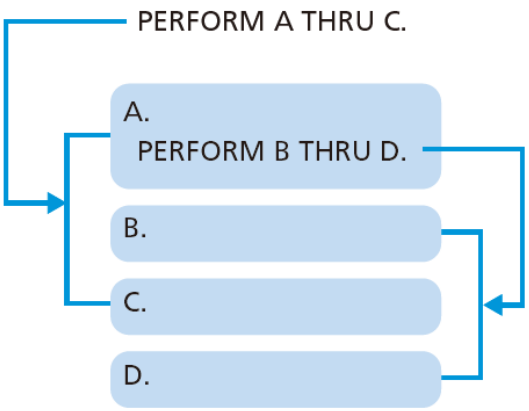
プログラムの品質問題には、いくつかのカテゴリーがあります。不正なプログラミングによってアプリケーションの出力結果に誤りがあるようなケースは、問題が顕在化するため、発見されやすいものです。次に、COBOL 言語仕様上正しくないプログラミングであるにもかかわらず、出力結果には問題が無いというケースもあります。そのような箇所は、現時点で結果に問題がなくても、プログラムの改修・再コンパイル・移植などのタイミングで問題が顕在化する可能性があります。以下のようなケースがこれに該当します。

●変数を初期化せずに使用する

COBOL では、VALUE 句を書かないデータ項目の初期値は言語として規定されていません。COBOL コンパイラの実装によって規定されることがあり、Rocket COBOL ではデフォルトでは空白(X' 20')で初期化されます。このようなコンパイラの種類に依存する仕様を仮定したプログラミングは、現在は問題なく稼働していても将来の再コンパイルや移植で問題が顕在化する可能性があり、品質上好ましくありません。

●複数の PERFORM 文でその実行対象範囲に重なりがある

COBOL の PERFORM A THRU B 文法は、引き続く複数の節・段落をまとめて PERFORM するものです。この実行対象範囲が重複するような複数の PERFORM 文の動作は、言語仕様上規定されておらず、コンパイラの実装により異なります。



●GO TO などによって明示的に終了しない PERFORM 文

COBOL では PERFORM 文による手続き実行は、実行範囲の最後の文で明示的に終了しなければならない規則となっています。PERFORM 文による実行範囲の中から GO TO 文で外部へ分岐することは許容されていますが、必ず再び実行範囲に戻って終了しなければなりません。終了しないまま処理を続行していると、COBOL ランタイムはまだ PERFORM 文を実行中の状態で処理を継続するため、プログラマの意図した挙動と異なることがあります。

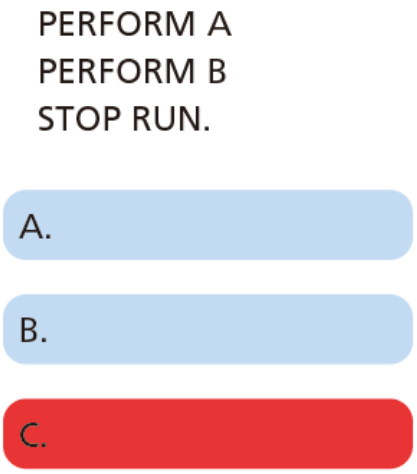
●FILE SECTION 中に書かれた VALUE 句

COBOLでは、VALUE句の記述はWORKING-STORAGE SECTIONでのみ意味があり、FILE SECTIONでは書かれても無視される規則となっています。しかしコンパイラによってはこれを有効とするものがあり、これを仮定して記述されたプログラムはたまたま動作しているに過ぎません。

また、言語仕様上不正ではないがプログラミング作法上問題があり、将来にわたる保守作業の品質に悪影響を与えうるものとして以下のようなケースがあります。

●決して到達することの無い実行文(デッドコード)

STOP RUN 文の直後に書かれた実行文が代表的なデッドコードですが、その他にも例えば以下のような書き方でデッドコードは発生します。



上の例では、段落 C は、その直前の段落 B からのフォールスルーで実行される可能性があるように見えますが、段落 B が PERFORM 文でしか実行されていないため、フォールスルーはあり得ません。
デッドコードは、出口ラベルのような作法として記述する場合に必然性をもって発生することがありますが、そうではなく単にプログラム修正を重ねた結果として発生するケースが多く、これはテストカバレッジの達成や保守性の観点から好ましくありません。

●使用しないデータ(デッドデータ)

デッドデータは、プログラム修正を重ねた結果、既に使用されなくなったデータ項目が削除されずに残っている状態で発生します。保守性の観点のみならず、実行時に消費するメモリも発生するため、好ましくありません。ただし、汎用的な COPY メンバーの中で宣言された項目のうち、一部だけを使用するといったケースは問題とはなりません。

●深すぎる入れ子の制御構造

IF 文などの条件文の入れ子が深すぎるプログラムは可読性を低下させ、バグ混入の原因となります。一般的なコーディング規約では5階層以上の場合は、PERFORM 文などで外出しにすべきであると言われています。また、複雑なディシジョンテーブルは、IF 文で実現せずに EVALUATE 文を利用すると整理された形で記述することができます。

5. 性能向上

どのようなプログラミング言語でも、同じ結果を得るためのコーディング方法は複数存在します。このようなケースでは、COBOL 言語の特性、または弊社のコンパイラの特性として実行性能の良い方法と悪い方法があります。性能の悪い方法でプログラミングされた部分は、実行結果に問題がないため多くは顕在化しません。しかし、このような箇所を改善することで、アプリケーションの処理速度を高速化できる可能性があります。元々時間がかかっているバッチプログラムにループ処理がある場合には、目に見えて効果的な改善になることも少なくありません。

以下のようなケースが性能の悪い方法に該当します。

●表参照の添字付けにゾーン十進項目を使用

ゾーン十進形式は、算術演算や表参照の添字付けで使用するにはもっとも効率が悪い形式です。添字付けは INDEXED BY 句で宣言した指標を使用するのがもっとも高速です。特に意味が無ければ指標以外の項目を使用するべきではありません。

●内部使用の作業用数字項目にゾーン十進項目を使用

カウンタ変数や複合的な計算処理の一時結果を保持するだけの目的で宣言された作業用数字項目は、画面や帳票に送られることがないため、ゾーン十進で宣言する意味はありません。最も効率的な COMP-5 項目などを使用するべきです。

●基本項目の初期化に INITIALIZE 文を使用

INITIALIZE 文は、集団項目の初期化を1つの命令文で実行するときの可読性、保守性を高めるものですが、実行性能は、必ずしも良くありません。特に対象が基本項目である場合は、INITIALIZE 文を使用するメリットはないため MOVE 文による初期化を使用するべきです。

●集団項目中のバイナリデータ項目がアライメント(整列)されていない

ほとんどの CPU では、バイナリデータのレジスタへのロード、ストアは、4バイトまたは8バイト境界にアライメントされている時が最も効率的です。また、RISC マシンではアライメントされていないデータのロード、ストアは許容されていません。このため C 言語などでは、コンパイラによる自動的なアライメントが行われますが、COBOL はデフォルトではそれを行いません。アライメントされていないバイナリ項目に対する操作ではコンパイラが余分なコードを生成しており、効率が低下します。

●Intel プラットフォームで COMP(ビッグエンディアン)を使用している

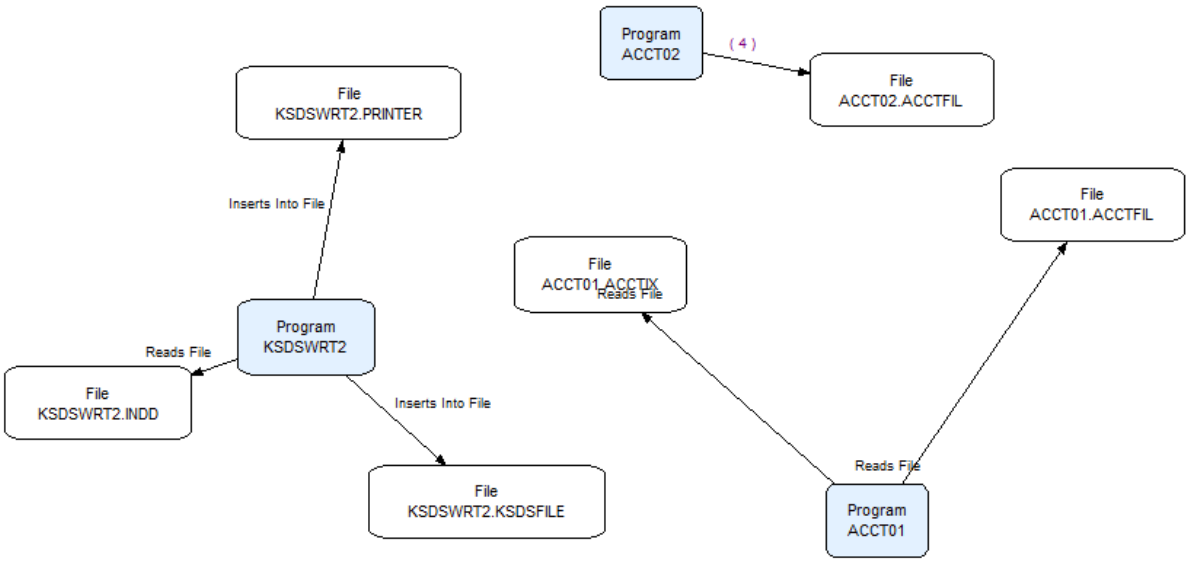
COBOL では COMP 項目はプラットフォームに依存せずにビッグエンディアン表現となっており、リトルエンディアンがネイティブである Intel CPU では効率が低下します。特に理由が無い限り COMP-5 を使用するべきです。

6. 可視化

ここまでは、「潜在的な」問題箇所を自動抽出するという点に着目して事例を挙げてきました。この他にも Enterprise Analyzer は、広範囲な機能を提供しており、リバースエンジニアリング・ツールの主な目的である可視化でも活用することができます。可視化に対するニーズは、仕様書が整備されていないアプリケーションの保守作業時に必要とされ、Enterprise Analyzer が以下のような場面で役に立ちます。

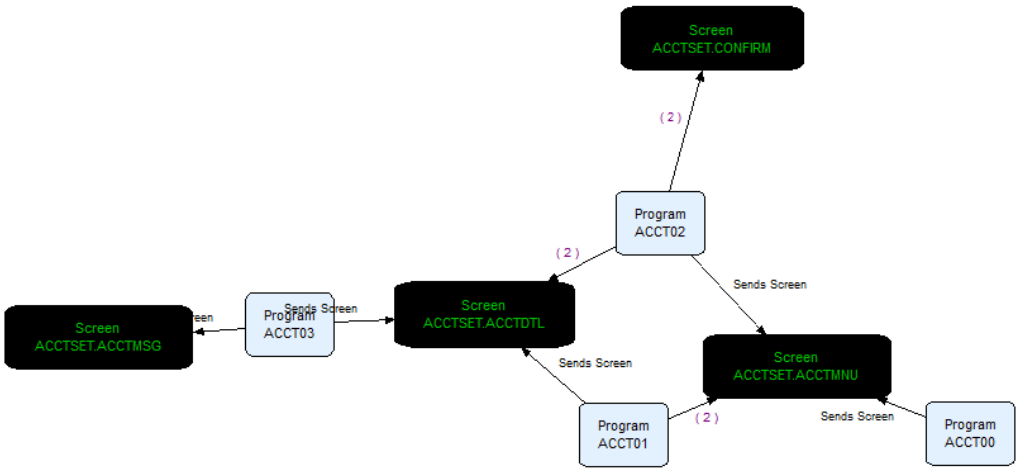
6.1 データフロー

プログラムで使用されているファイルが入出力別にグラフィカルに表示されます。



6.2 画面フロー

プログラムと CICS のスクリーン画面などの画面とどのように関連しているのか図式化して表示します。



6.3 CRUD レポート

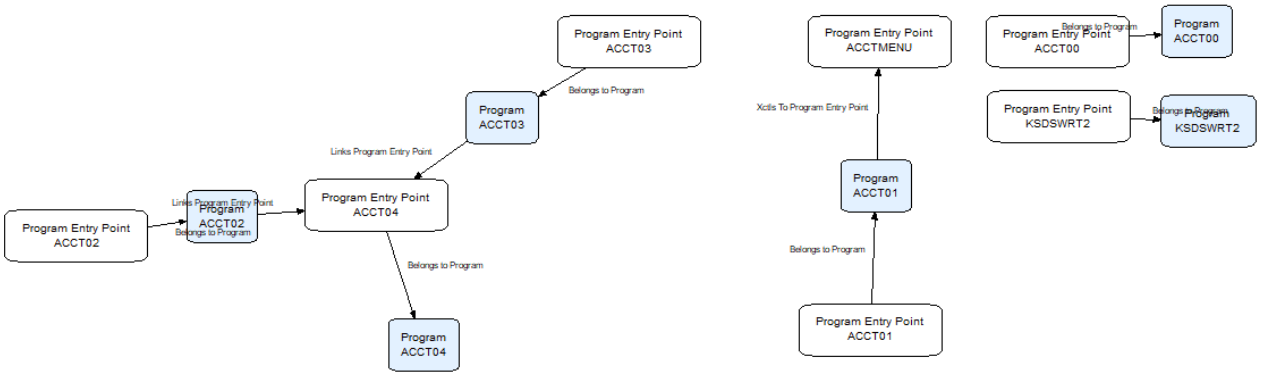
プログラムとファイル・データベーステーブルの間のクロス参照関係を一覧表示します。特にデータのライフサイクルを示すため、Create（作成）／Read（読み取り）/Update（更新）／Delete（削除）のオペレーションを区別して表示しますので、単にアクセスしているだけの関係ではなく、関係の種別に応じた検索が可能となります。

CRUD レポート - MyNewWorkspace1 (MyNewWorkspace1)

プログラム	ファイル名	データ・ストア	データ	タイプ	作成	読み取り	更新	削除
ACCT01	ACCT01.cbl		ACCTFIL	File		+		
ACCT01	ACCT01.cbl		ACCTIX	File		+		
ACCT02	ACCT02.cbl		ACCTFIL	File	+	+	+	+
KSDSWRT2	KSDSWRT2.cbl		INDD	File		+		
KSDSWRT2	KSDSWRT2.cbl		KSDSFILE	File	+			
KSDSWRT2	KSDSWRT2.cbl		PRINTER	File	+			

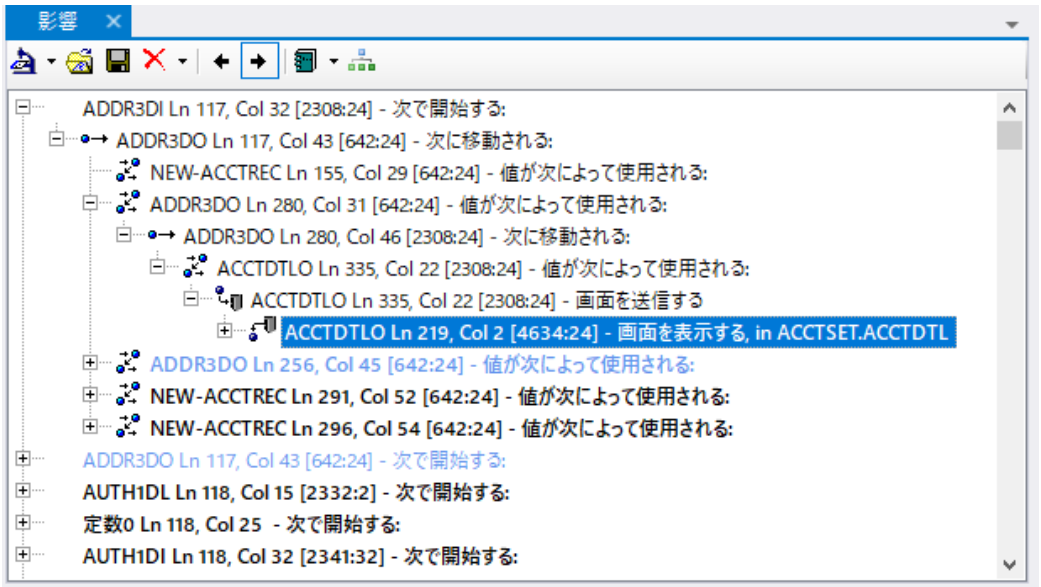
6.4 マップの呼び出しフローダイアグラム

プログラムとプログラムの間の呼び出し関係をグラフィカルに関連づけて表示します。このダイアグラムはこのほかにも JCL からプログラム起動なども表示することができます。



6.5 影響分析

現在、着目している項目がどのような経路を経て、画面・ファイル・データベースに出力されるか（順方向分析）、または画面・ファイル・データベースから入力されるか（逆方向分析）を順次検索して表示します。出力結果に不正が確認された際に、その結果がどのような処理を経由して出力されたのかを効果的にトレースすることができ、目視による作業と比較して著しい生産性と作業品質の向上を得ることができます。



7. AI との連携

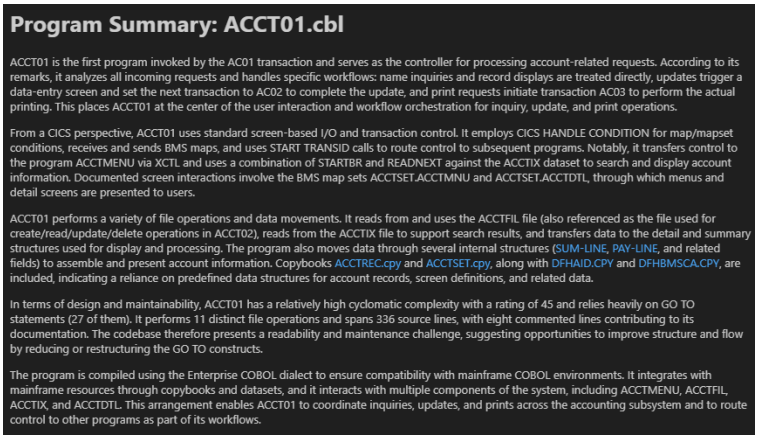
Enterprise Analyzer は、自然言語によるチャット、AI による用語集の生成、ビジネスルールの生成など、AI を活用したさまざまな機能を提供しています。これらの機能を使用するには、あらかじめ AI を使用するように Enterprise Analyzer を構成する必要があります。また、IDE からの利用方法として、Visual Studio Code 用の Analyzer アドインも提供されています。

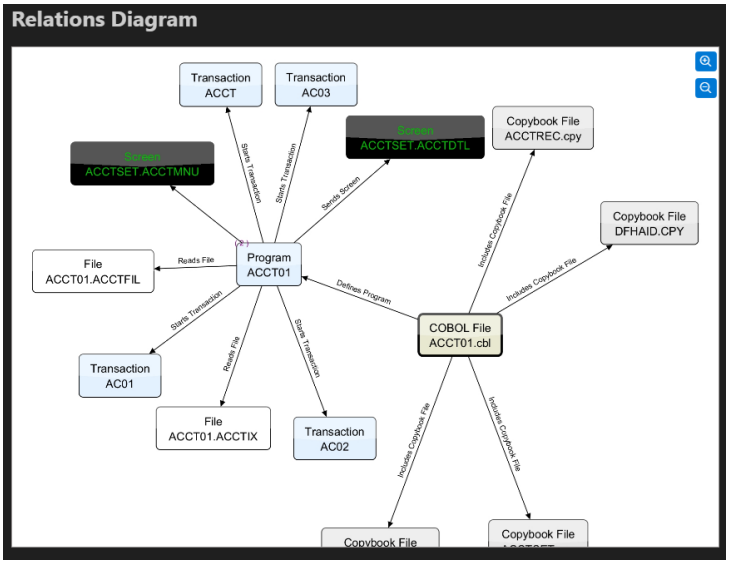
7.1 自然言語によるチャットウィンドウの問い合わせ

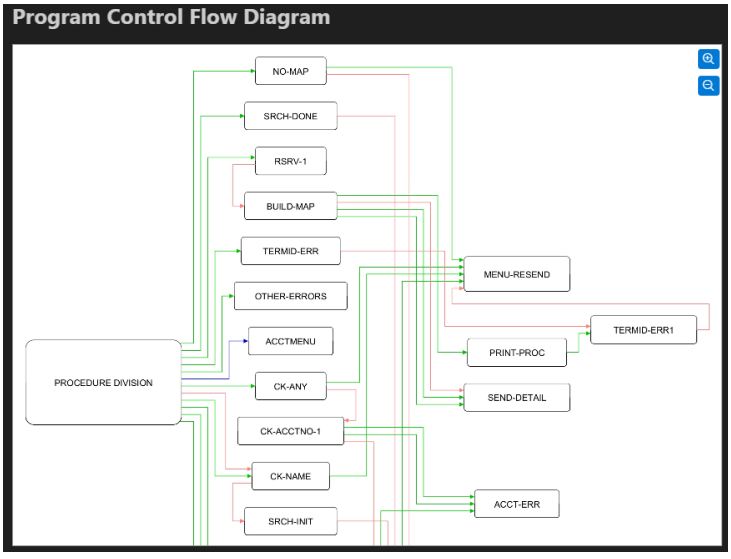
この機能は、生成 AI との連携を行い、チャットウィンドウから自然言語で問い合わせを行い、結果の詳細を確認することができます。下記の例では、COBOL ソースの品質に問題があるか、と日本語で抽象的な問い合わせを行っても具体的な回答を得られ、さらにプログラム内の該当箇所の詳細を確認することができます。

7.2 Visual Studio Code との連携

Enterprise Analyzer の Visual Studio Code 専用プラグインを設定することで無償の統合開発環境である Visual Studio Code 内で Enterprise Analyzer や生成 AI と連携させてプログラムの概要やレーションダイアグラム、プログラムコントロールフローダイアグラムを生成することができます。







8. まとめ

ここまで、COBOL ソースコードのブラッシュアップに Enterprise Analyzer を活用する方法について例示してきました。

現在は、現在重大な問題が発生せずに稼動しているアプリケーションであっても、潜在的な問題を含んでいる可能性があります。

安定稼働しているものにあえて手を入れるリスクを負う必要はありません。しかし、プラットフォーム移行、サーバーリプレースなどのタイミングでこのような作業を計画されてみてはいかがでしょうか。長いライフサイクルを持つ COBOL プログラムであればこそ、このような施策によって将来に渡って安定的にビジネスを支え続けていくことができるようになります。