28. ソフトウェアのリエンジニアリング

目的

本章の目的は、ソフトウェア・システムの保守性を向上させるソフトウェア・リ・エンジニ アリングのプロセスを説明することである。この章を読めば、次のことがわかる:

- リエンジニアリングが、ソフトウェア・システムの進化にとって、時として費用対効果の高い選択肢となる理由を理解する、
- リバース・エンジニアリングやプログラム・リエンジニアリングなどの活動を理解する。ソフトウェアのリエンジニアリングプロセスに関与する可能性のある構造化、
- ソフトウェアとデータのリエンジニアリングの違いを理解する。そして、データのリエンジニアリングが高価で時間のかかるプロセスである理由を理解する。

内容

第26章と第27章では、レガシー・システムとソフトウェア進化のためのさまざまな戦略について紹介した。レガシー・システムとは、ビジネス・プロセスのサポートに不可欠な古いソフトウェア・システムのことである。企業はこれらのシステムに依存しているため、運用を続けなければならない。ソフトウェアの進化戦略には、メンテナンス、リプレース、アーキテクチャの進化、そして本章のテーマであるソフトウェアのリエンジニアリングがある。

ソフトウェア・リエンジニアリングは、レガシーシステムを再実装し、より保守しやすくすることを目的としている。リエンジニアリングには、システムの再ドキュメント化、システムの整理と再構築、システムの最新プログラミング言語への翻訳、システムのデータ構造と値の変更と更新が含まれる。ソフトウェアの機能は変更されず、通常、システム・アーキテクチャも変更されない。

技術的な観点から見ると、ソフトウェアのリエンジニアリングは、システム進化の問題に対する二流の解決策に見えるかもしれない。ソフトウェア・アーキテクチャは更新されないので、集中管理されたシステムを分散させることは難しい。システムのプログラミング言語を根本的に変更することは通常不可能なので、古いシステムをJavaやC++などのオブジェクト指向プログラミング言語に変換することはできない。ソフトウェアの機能が変更されないため、システムに内在する制限が維持される。

しかし、ビジネスの観点からは、ソフトウェアのリエンジニアリングが、レガシーシステムのサービス継続を保証する唯一の実行可能な方法かもしれない。システムの進化に他のアプローチを採用するのは、あまりにも高価でリスクが高すぎるかもしれない。この理由を理解するためには、レガシーシステムの問題を大まかに評価する必要がある。

レガシーシステムに含まれるコードの量は膨大である。1990年には、*1200億*行のソースコードが存在すると推定された（Ulrich, 1990）。これらのシステムの大部分は、ビジネスデータ処理に最適なプログラミング言語であるCOBOLか、FORTRANで書かれてきた。FORTRANは科学的あるいは数学的プログラミングのための言語である。これらの言語はプログラム構造化機能が限られており、FORTRANの場合はデータ構造化のサポートが非常に限られている。

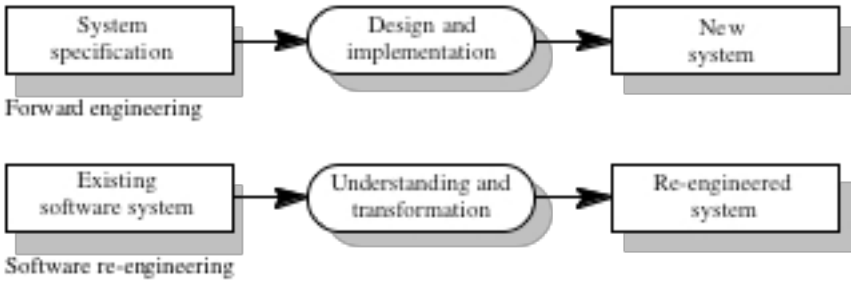
現在、これらのプログラムの多くはリプレースされたが、そのほとんどはまだ現役だろう。一方、1990年以降、ビジネス・プロセスをサポートするためのコンピューター利用が大幅に増加した。そのため、現在ではお

よそ2500億行のソースコードが存在し、それらはメンテナンスされなければならないものと推測される。そのほとんどはオブジェクト指向言語で書かれておらず、その多くはいまだにメインフレームコンピューター上で動いている。

現存するシステムは非常に多く、完全な置き換えや抜本的な再構築は、ほとんどの組織にとって財政的に考えられない。古いシステムのメンテナンスにはますますコストがかかるため、リエンジニアリングによってシステムの耐用年数を延ばすことができる。第26章で論じたように、システムのリエンジニアリングは、ビジネス上の価値は高いが、その維持に費用がかかる場合に、費用対効果が高い。リエンジニアリングは、システム構造を改善し、新しいシステム文書を作成し、理解しやすくする。

ソフトウェア・システムのリエンジニアリングには、システム進化に対するより急進的なアプローチと比較して、2つの重要な利点がある:

1. *リスクの低減* 組織にとって必要不可欠なソフトウェアの再開発には高いリスクが伴う。システムの仕様に誤りがあったり、開発に問題があったりする可能性がある。



2. *コスト*の削減 リエンジニアリングのコストは、新しいソフトウェアを開発するコストに比べて大幅に低い。Ulrich (Ulrich, 1990)は、リインプリメンテーションのコストが5,000万ドルと見積もられていた商用システムの例を引用している。このシステムは1200万ポンドでリエンジニアリングに成功した。この数字が典型的なものであるとすれば、リエンジニアリングは書き直すよりも約4倍安いことになる。

リエンジニアリングという用語は、ビジネスプロセス・リエンジニアリングとも関連している（Hammer, 1990）。ビジネスプロセス・リエンジニアリングは、冗長な活動の数を減らし、プロセスの効率を改善するために、ビジネスプロセスを再設計することに関係する。それは通常、プロセスのためのコンピュータベースのサポートの導入または強化に依存している。レガシーシステムには、既存のプロセスに対する暗黙の依存関係が含まれている可能性があるため、プロセス・リエンジニアリングは、しばしばソフトウェアの進化の原動力となる。プロセスのリエンジニアリングが可能になる前に、これらを発見し、取り除く必要がある。したがって、ビジネスプロセス・リエンジニアリングで必要とされる変更の規模が、通常のプログラムメンテナンスでは対応できないことが明らかになったとき、ソフトウェア・リエンジニアリングの必要性が企業で生じることがある。

リエンジニアリングと新規ソフトウェア開発の決定的な違いは、開発の出発点にある。仕様書から始めるのではなく、旧システムが新システムの仕様書として機能する。Chikofsky and Cross (Chikofsky and Cross, 1990)は、従来の開発を*フォワードエンジニアリング*と呼び、ソフトウェアのリエンジニアリングと区別している。この違いを図 28.1 に示す。フォワードエンジニアリングは、システム仕様から始まり、新しいシステムの設計と実装を含む。リエンジニアリングは、既存のシステムから開始し、元のシステムの理解と変換に基づいて、置き換えのための開発プロセスを行う。

図28.2は、リエンジニアリングの可能なプロセスを示している。このプロセスの入力はレガシープログラムであり、出力は同じプログラムの

構造化、モジュール化されたバージョンである。プログラムのリエンジニアリングと同時に、システムのデータもリエンジニアリングされる。このリエンジニアリングプロセスのアクティビティは以下の通りである：

図28.1 フォワード・エンジニアリングとリエンジニアリング

1. *ソースコード翻訳* プログラムを古いプログラミング言語から、同じ言語の最新版、または別の言語に変換*する*。
2. リバースエンジニアリング プログラムを分析し、その組織と機能を文書化するのに役立つ情報を抽出する。
3. *プログラム構造の改善* プログラムの制御構造を分析し、読みやすく理解しやすいように修正する。

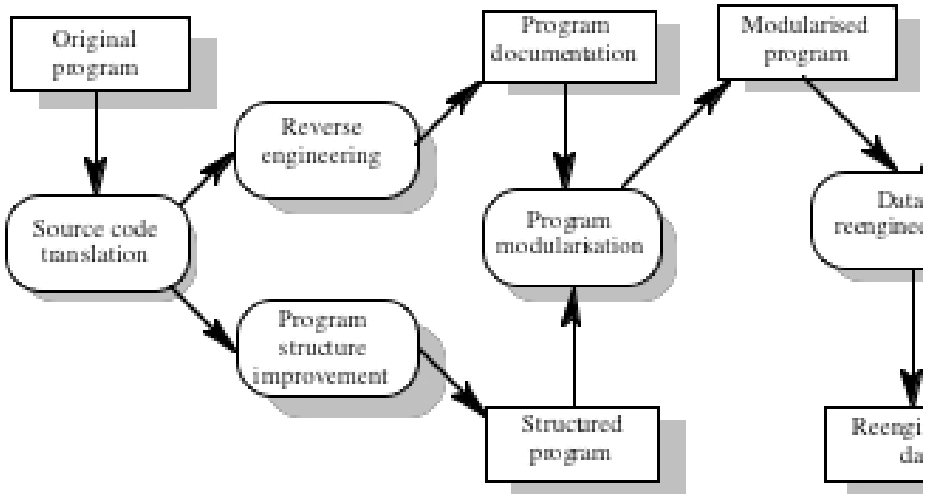


図28.2 リエンジニアリングのプロセス

4. *プログラムのモジュール化* プログラムの関連部分をグループ化し、必要に応じて冗長な部分を削除する。場合によっては、この段階で
第27章で説明するように、アーキテクチャーを変換する必要があるかもしれない。

5. *データのリエンジニアリング* プログラムが処理するデータを、プログラムの変更に合わせて変更する。

プログラム・リエンジニアリングは、必ずしも図のすべてのステップを必要とするわけではない。

28.2.システム開発に使用されたプログラミング言語が、コンパイラ・サプライヤーによってまだサポートされている場合、ソースコードの翻訳は必要ないかもしれない。リエンジニアリングが完全に自動化ツールに依存している場合は、リバース・エンジニアリングによるドキュメンテーションの回復は不要かもしれない。データのリエンジニアリングが必要になるのは、システムのリエンジニアリング中にプログラムのデータ構造が変更された場合だけである。しかし、ソフトウェアのリエンジニアリングは、常にプログラムの再構築を伴う。

リエンジニアリングのコストは、当然ながら、実施する作業の程度に依存する。図28.3に示すように、リエンジニアリングにはさまざまなアプローチがあります。コストは左から右へと増加し、ソースコードの翻訳が最も安価なオプションであり、アーキテクチャの移行の一環としてのリエンジニアリングが最も高価になります。

リエンジニアリングの程度は別として、リエンジニアリング費用に影響を与える主な要因は以下の通りである：

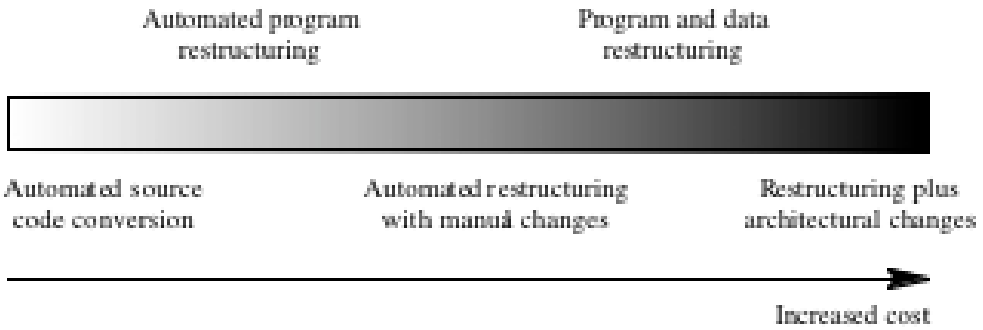
1. *リエンジニアリングするソフトウェアの品質*。ソフトウェアとその関

連文書（もしあれば）の品質が低ければ低いほど、リエンジニアリング費用は高くな

る。

2. *リエンジニアリングに利用可能なツールサポート*。CASEツールを使用してプログラム変更のほとんどを自動化できない限り、ソフトウェアシステムのリエンジニアリングは通常、費用対効果が高いとは言えません。

3. *必要なデータ変換の範囲*。リエンジニアリングに大量のデータ変換が必要な場合、プロセスコストが大幅に増加する。



4. *専門スタッフの有無*。システムの保守を担当するスタッフがリエンジニアリングプロセスに関与できない場合、コストが増大する。システムのリエンジニアリング担当者は、システムを理解するために多くの時間を費やさなければならない。

ソフトウェアのリエンジニアリングの主な欠点は、リエンジニアリングによってシステムを改善できる範囲に現実的な限界があることである。たとえば、関数型アプローチで書かれたシステムをオブジェクト指向システムに変換することは不可能である。大規模なアーキテクチャの変更や、システム・データ管理の抜本的な再編成は、自動的に行うことができないため、高い追加コストがかかる。リエンジニアリングは保守性を向上させるが、リエンジニアリングされたシステムは、最新のソフトウェア工学的手法で開発された新しいシステムほど保守性は高くないだろう。

28.1 ソースコード翻訳

ソフトウェアのリエンジニアリングの最も単純な形態は、あるプログラミング言語のソースコードを他の言語のソースコードに自動的に翻訳するプログラム翻訳である。プログラムの構造や構成自体は変更されない。ターゲット言語は、元の言語の更新版（COBOL-74からCOBOL-85など）である場合もあれば、まったく異なる言語への翻訳（FORTRANからC言語など）である場合もある。

ソースレベルの翻訳が必要になるのは、次のような理由がある：

1. *ハードウェア・プラットフォームの更新* 組織が標準ハードウェア・プラットフォームの変更を希望する場合がある。元の言語のコンパイラは、新しいハードウェアでは利用できない可能性があります。

2. *スタッフのスキル不足* オリジナル言語の訓練を受けたメンテナンススタッフが不足している可能性がある。これは、現在では一般的に使

である。

3. *組織の方針変更* 組織は、サポート・ソフトウェアのコストを最小限に抑えるために、特定の言語を標準化することを決定するかもしれない。古いコンパイラを何バージョンも維持するのは、非常に高くつくことがある。

図28.3 リ・エンジニアリング・アプローチ

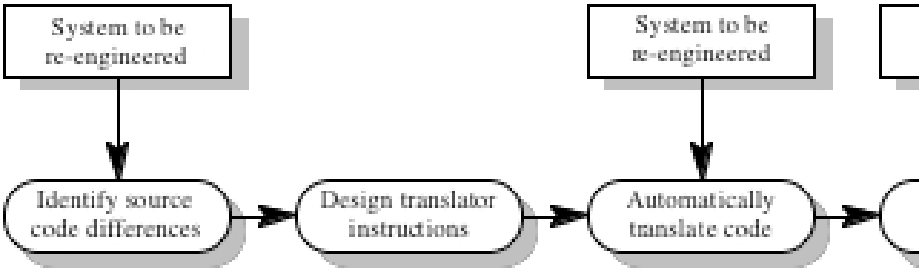


図28.4 プログラムの翻訳プロセス

4. *ソフトウェア・サポートの欠如* 言語コンパイラのサプライヤーが廃業したり、製品のサポートを打ち切ったりする可能性がある。

図28.4は、ソースコード翻訳のプロセスを示している。ソフトウェアの動作を詳細に理解したり、システム・アーキテクチャを修正したりする必要はないかもしれない。プログラム制御構文の等価性など、プログラミング言語の考慮事項に重点を置いた分析が可能である。

ソースコードの翻訳は、翻訳の大部分を自動化された翻訳者が行うことができる場合にのみ、経済的に現実的です。これは、特別に書かれたプログラムであったり、ある言語から別の言語に変換するツールを購入したり、パターンマッチングシステムであったりする。後者の場合、ある表現から別の表現への翻訳をどのように行うかという命令一式を書かなければならない。ソース言語のパラメータ化されたパターンが定義され、ターゲット言語の同等のパターンと関連付けられる。

多くの場合、完全な自動翻訳は不可能である。ソース言語の構文は、ターゲット言語に直接相当するものがない場合がある。ソースコードに、ターゲット言語ではサポートされていない条件付きコンパイル命令が埋め込まれている場合があります。このような状況では、生成されたシステムを調整し改善するために、手動で変更を加える必要があります。

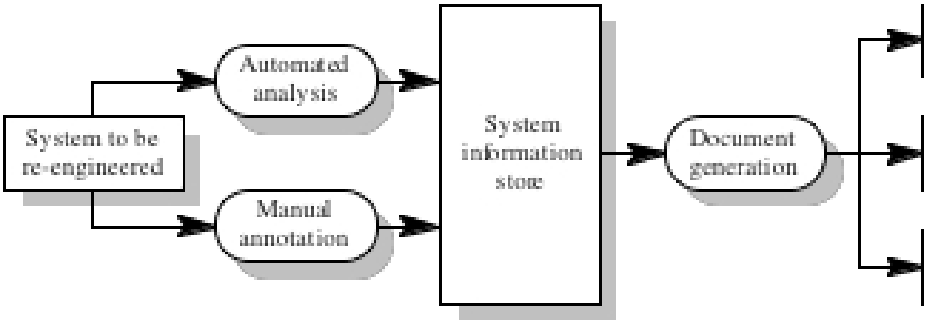
28.2 リバース・エンジニアリング

リバース・エンジニアリングとは、ソフトウェアの設計や仕様を復元する目的でソフトウェアを分析するプロセスである。リバース・エンジニアリングのプロセスでは、プログラム自体は変更されない。リバースエンジニアリングの入力として、ソフトウェアのソースコードは通常入手可能である。しかし、時にはそれさえも失われてしまい、リバース・エンジニアリングは実行コードから始めなければならないこともあります。

リバースエンジニアリングは、リエンジニアリングとは違う。リバースエンジニアリングの目的は、ソースコードからシステムの設計や仕様を

導き出すことである。リエンジニアリングの目的は、より保守性の高い新しいシステムを作ることです。もちろん、図28.2からわかるように、システムの理解を深めるためのリバースエンジニアリングは、しばしばリエンジニアリングプロセスの一部です。

リバースエンジニアリングは、ソフトウェアのリエンジニアリングプロセスにおいて、プログラム設計を復元するために使用される。しかし、リバースエンジニアリングが常にリエンジニアリングに続く必要はない：



1. 既存システムの設計と仕様をリバースエンジニアリングすることで、そのプログラムをリプレースする際の要求仕様のインプットとすることができる。

2. あるいは、設計と仕様をリバースエンジニアリングして、プログラムの保守に役立てることもできる。この追加情報があれば、システムのソースコードをリエンジニアリングする必要はないかもしれない。

リバースエンジニアリングのプロセスを図28.5に示す。プロセスは分析段階から始まる。このフェーズでは、自動化ツールを使ってシステムを分析し、その構造を発見する。これだけでは、システム設計を再作成するには不十分である。次にエンジニアは、システムのソースコードとその構造モデルを使って作業を行う。エンジニアは、システムを理解することによって収集した情報をこれに追加する。この情報は、プログラムソースコードにリンクされた有向グラフとして管理される。

情報ストアブラウザは、グラフ構造とコードを比較したり、余分な情報でグラフに注釈を付けたりするために使用される。プログラム図やデータ構造図、トレーサビリティ・マトリクスなど、さまざまなタイプのドキュメントを有向グラフから生成することができる。トレーサビリティ・マトリクスは、システム内のエンティティがどこで定義され、参照されているかを示す。ドキュメント生成のプロセスは、設計情報を使用してシステム・リポジトリに保持されている情報をさらに洗練させるため、繰り返し行われます。

リバース・エンジニアリングのプロセスをサポートするために、プログラムを理解するためのツールが使用されることがある。これらのツールは通常、さまざまなシステムビューを表示し、ソースコードを通して簡単にナビゲートできるようになっている。例えば、ユーザーがデータ定義を選択し、そのデータ項目が使用されている場所までコード内を移動することができる。このようなプログラムブラウザの例は、Clev

eland (Cleveland, 1989)、Oman and Cook (Oman and Cook, 1990)、Ning *et al.* (Ning, Engberts et al., 1994)によって議論されている。

図28.5 リバース・エンジニアリングのプロセス

システム設計文書が生成された後、システム仕様の再作成を支援するために、さらなる情報が情報ストアに追加されることがある。これには通常、システム構造へのさらなる手作業による注釈が含まれる。システム・モデルから仕様を自動的に推測することはできない。

28.3 プログラム構造の改善

メモリ使用量を最適化する必要性と、多くのプログラマーがソフトウェア工学を理解していないため、多くのレガシーシステムはうまく構造化されていない。その制御構造は、多くの無条件分岐や直感的でない制御ロジックで絡み合っている。この構造は、定期的なメンテナンスによって劣化している可能性もある。プログラムの変更によって、一部のコードが到達不能になっている可能性があるが、これは広範な解析の後にしか発見できない。メンテナンス・プログラマーは、間接的にアクセスされる可能性があるコードをあえて削除しないことが多い。

図28.6は、複雑な制御ロジックが比較的単純なプログラムをいかに難解にするかを示している。このプログラムは、この種のプログラムを書くのによく使われたFORTRANに似た記法で書かれている。しかし、暗号のような変数名を使うことで、プログラムをさらにわかりにくくしているわけではない。図28.6の例は、暖房システムのコントローラである。パネルスイッチは、オン、オフ、制御のいずれかに設定される。システムが制御されている場合、タイマー設定とサーモスタットによってオンとオフが切り替わります。暖房がオンの場合、スイッチ・ヒーティングはそれをオフにし、逆の場合はオフにする。

通常、プログラムはメンテナンス中に変更されることで、このような複雑な論理構造を持つようになる。新しい条件と関連するアクションは、既存の制御構造を変更することなく追加される。短期的には、この方法はシステムに障害をもたらす可能性を減らすため、より迅速でリスクの少ない解決策である。しかし長期的には、理解しがたいコードになってしまう。プログラマーがコードの重複を避けようとした場合にも、複雑なコード構造が生じることがある。これは、プログラムが限られたメモリーに制約されている場合に必要になることがある。

図28.7は、同じ制御システムを構造化制御文を使って書き直したものである。このプログラムは、上から下へと順番に読むことができ、理解しやすくなっている。オン、オフ、制御の3つのスイッチ・ポジションが明確に識別され、関連するコードにリンクされている。元のプログラムはオブジェクト指向ではないので、ここではJavaを使っていない。

構造化されていないコントロールと同様に、複雑な条件も単純化することができる。

```
スタートGet (Time-on, Time-off, Time, Setting, Temp,
        Switch) if Switch = off goto off.
        if Switch = on goto on
        goto Cntrld
off:    if 加熱状態 = on goto Sw-off goto
        ループ
on:     if 加熱状態 = off goto スイッチオ
```

図28.6 スパゲッティ・ロジックによる制御プログラム

```
ループ
   -- Get文は、与えられた変数の値をシステムの
   -- 環境。
   取得（タイムオン、タイムオフ、時間、設定、温度、スイッチ）；
   の場合
      when On => if ヒーティング・ステータス = off then
                        暖房状態 := オン；
                  end if；
      When Off => if ヒーティング・ステータス = on then
                        スイッチ・ヒーティング ; 加熱状態 := オフ；
                  end if;
      when 制御される =>
          もし時間 >= タイムオンかつ時間 < = タイムオフなら
              もし気温＞設定値で、かつヒーティング・ステータスがオンなら
                  スイッチ・ヒーティング; ヒーティング・ステータス＝オフ
                  ;
              温度＜設定値で、ヒーティング・ステータスがオフの場合
                  加熱スイッチ; 加熱状態 := オン；
              end if;
          end if；
   エンドケース；
終了ループ；
```

図 28.7 構造化された制御プログラム

プログラム再構築の一環として。図28.8は、'not'ロジックを含む条件文がどのように理解しやすくなるかを示している。

ボームとヤコピーニ(Bohm and Jacopini, 1966)は、どんなプログラムも単純なif-then-else条件式とwhileループで書き換えられ、無条件のgoto文は必要ないことを証明した。この定理は、自動的なプログラム再構築の基礎となっている。図28.9は、プログラムの自動的な再構築の段階を示している。まず有向グラフに変換され、次にgoto文のない構造化された等価プログラムが生成される。

生成される有向グラフはプログラム・フロー・グラフであり、制御がプログラム中をどのように移動するかを示す。このグラフには、そのセマンティクスを変更することなく、簡略化や変換のテクニックを適用することができる。これらはコードの到達不可能な部分を検出し、削除する。単純化が完了すると、新しいプログラムが生成される。Whileループと単純な条件文が、gotoベースの制御に置き換えられる。このプログラムは、元の言語でもよいし、別の言語でもよい（例えば、FORTRANをCに変換することもできる）。

自動的なプログラム再構築の問題点には以下のようなものがある:

1. *コメントの消失* プログラムにインライン・コメントがある場合、再構築の過程で必ず失われる。

2. *ドキュメントの紛失* 同様に、外部のプログラム・ドキュメントとプログラムの対応関 係 も失われる。しかし、多くの場合、プログラムのコメントとドキュメンテーションの両方が失われている。

```
-- 複雑な条件
でない場合（A＞Bかつ（C＜Dまたは（E＞F）でない場合）
)...

-- 簡易条件
A <= B かつ (C>= D または E > F)...
```

図28.8 コンディションの単純化

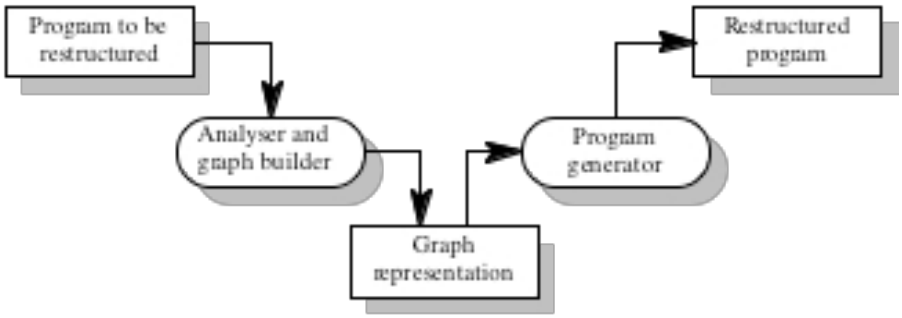


図28.9 自動化されたプログラム再構築

そのため、これは重要な要素ではない。

3. *計算負荷が大きい* 再構造化ツールに組み込まれているアルゴリズムは複雑である。高速で最新のハードウェアを使用しても、大規模なプログラムのリストラクチャリング処理を完了するには長い時間がかかる。

プログラムがデータ駆動型で、コンポーネントが共有データ構造によって緊密に結合されている場合、コードを再構築しても理解しやすさの大幅な改善にはつながらないかもしれない。次のセクションで説明するように、プログラムのモジュール化も必要かもしれない。プログラムが非標準的な言語方言で書かれている場合、標準的なリストラクチャリング・ツールは適切に動作せず、大幅な手作業が必要になることがある。

場合によっては、システム内のすべてのプログラムを再構築することが費用対効果に合わないこともある。あるものは他のものよりも質が高いかもしれないし、あるものは頻繁には変更されないかもしれない。Arthur (Arthur, 1988)は、再編成から最も恩恵を受ける可能性のあるプログラムを特定するために、データを収集することを提案している。たとえば、次のような指標を使用して、再構築の候補を特定することができる:

- 故障率。
- 1年間に変更されたソースコードの割合。
- コンポーネントの複雑さ。

再編成を決定する際には、プログラムや構成要素が現行の基準にどの程度合致しているかなど、その他の要因も考慮されるかもしれない。

28.4 プログラムのモジュール化

プログラムのモジュール化とは、関連するプログラム部分をまとめて1つのモジュールとみなし、プログラムを再編成するプロセスのことである。これが行われると、関連するコンポーネントの冗長性を取り除き、それらの

が容易になる。例えば、地震データを処理するプログラムでは、データのグラフィカルな表示に関連するすべての操作を1つのモジュールにまとめることができる。システムが分散される場合、作成されたモジュールはオブジェクトとしてカプセル化され、共通のインターフェースを介してアクセスすることができる。

相互作用を最適化し、プログラムの残りの部分とのインタフェースを簡素化すること

プログラムのモジュール化プロセスでは、いくつかの異なるタイプのモジュールを作成することができる。以下のようなものがある：

1. *データの抽象化* データと処理コンポーネントを関連付けることによって作成される抽象データ型である。これについては28.4.1節で説明する。

2. *ハードウェア・モジュール* これらはデータ抽象化と密接に関連しており、特定のハードウェア・デバイスを制御するために使用されるすべての機能を集めたものである。

3. *機能モジュール* 類似または密接に関連したタスクを実行する機能を集めたモジュールである。例えば、入力と入力検証に関係するすべての関数を1つのモジュールに組み込むことができる。この種のモジュール化は、プログラム・データの抽象化を回復することが現実的でない場合に考慮されるべきである。

4. *プロセスサポートモジュール* 特定のビジネスプロセスをサポートするために必要なすべての機能と特定のデータ項目がグループ化されたモジュールです。例えば、図書館システムの場合、プロセスサポートモジュールには、図書の発行と返却をサポートするために必要なすべての機能が含まれます。

プログラムのモジュール化は通常、コードを検査したり編集したりして手作業で行われる。プログラムをモジュール化するには、コンポーネント間の関係を特定し、これらのコンポーネントが何をするのかを調べなければならない。ブラウジングや視覚化ツールは助けになるが、このプロセスを完全に自動化することは不可能だ。

28.4.1 データ抽象化の回復

メモリ空間を節約するために、多くのレガシーシステムは共有テーブルや共通データ領域の使用に依存している。これらの領域の情報はグローバルにアクセス可能であり、システムの異なる部分で異なる方法で使用される可能性がある。このようなグローバルなデータ領域に変更を加えることは、データのすべての用途にわたって変更の影響を分析するコストがかかるため、コストが高くつく。

このような共有データ領域の変更コストを削減するために、プログラムのモジュール化プロセスでは、データ抽象化の特定に重点を置くことが

ある。データ抽象化または抽象データ型は、データと関連処理をまとめ、変更に強い。データ抽象化はデータ表現を隠蔽し、データを修正・検査するためのコンストラクタやアクセス関数を提供する。インターフェースが維持されている限り、データ型の変更がプログラムの他の部分に影響を与えることはない。

共有グローバルデータ領域をオブジェクトや抽象データ型に変換する手順は以下の通り：

1. 共通のデータ領域を分析し、論理的なデータ抽象化を特定する。1つの共有データ領域に複数の抽象化が組み合わされていることがよくある。これらを特定し、論理的に再構築する。

2. それぞれの抽象データ型やオブジェクトを作成する。プログラミング言語にデータ隠蔽機能がない場合は、データのすべてのフィールドを更新したりアクセスしたりする関数を用意して、抽象データ型をシミュレートする。

3. プログラム閲覧システムや相互参照ジェネレーターを使って、データへの参照をすべて見つける。これらを適切な関数の呼び出しに置き換える。

このプロセスは時間がかかるように見えるが、比較的簡単である。しかし実際には、共有データ領域の使用方法のため、非常に難しい場合がある。データ構造化機能が制限されているFORTRANなどの古いバージョンの言語では、プログラマーは共有配列を使って実装した複雑なデータ管理戦略を設計しているかもしれない。そのため、配列は実際には別の種類のデータ構造として使用されることがある。共有構造体の間接アドレス指定や、他の構造体からのオフセットによるアドレス指定は、さらなる問題を引き起こす。

元のプログラムのターゲット・マシンのメモリが限られていた場合、これは別の問題を引き起こす。プログラマーはデータの寿命に関する知識を利用し、これをプログラムに組み込んだかもしれない。余分な領域の割り当てを避けるために、同じデータ領域を使ってプログラム中の異なる時点で異なる抽象化を保存しているのだ。これらは、プログラムの詳細な静的解析と動的解析の後にしか発見できない。

28.5 データのリエンジニアリング

これまでのところ、ソフトウェアの進化に関する議論のほとんどは、プログラムやソフトウェア・システムを変更する際の問題に焦点を当ててきた。しかし、多くの場合、データの進化にも関連した問題がある。レガシープログラムで処理されるデータの保存、整理、形式は、ソフトウェアの変更を反映して進化しなければならないかもしれない。より理解しやすくするために、システム内のデータ構造、場合によってはデータ値を分析し、再編成するプロセスは、*データ・リエンジニアリング*と呼ばれる。

原則として、システムの機能に変更がなければ、データのリエンジニアリングは必要ありません。しかし実際には、レガシーシステムのプログラムだけでなく、データも変更しなければならない理由はいくつもあります：

1. *データの劣化* 時間の経過とともに、データの質は低下する傾向にある。データの変更によってエラーが発生したり、重複した値が作成されたり、外部環境の変化がデータに反映されなかったりする。データの寿命は非常に長いことが多いため、これは避けられない。例えば

、個人の銀行データは口座開設時に存在するようになり、少なくとも顧客の生涯にわたって存続しなければならない場合がある。顧客の状況が変化すると、その変化が銀行のデータに適切に含まれない可能性がある。プログラムのリエンジニアリングは、データ品質の問題を浮き彫りにし、関連するデータのリエンジニアリングの必要性を浮き彫りにすることができる。

2. *プログラムに組み込まれた固有の制限* 当初設計されたとき、多くのプログラムの開発者は、処理できるデータ量に制約を組み込んでいた。しかし現在では、開発者が当初想定していたよりもはるかに多くのデータを処理することが求められることが多い。制約を取り除くためには、データのリエンジニアリングが必要になるかもしれない。例えば、ロチェスターとダグラス(Rochester and Douglass, 1993)は、当初99のファンドまで扱えるように設計された資金管理システムについて述べている。このシステムを運用していた会社は、2000以上のファンドを管理していた。

アプローチ	説明
データクリーンアップ	データレコードとその品質を向上させるために分析さ重複は削除され、冗長な情報は削除され、すべてのレコードに一貫したフォーマットが適用される。通常、プログラムの変更は必要ない。
データ拡張	この場合、データと関連プログラムは、データ処理の制限を取り除くために再設計される。この場合、フィールドの長さを長くしたり、テーブルの上限を変更したりするなど、プログラムの変更が必要になることがある。この場合、プログラムの変更を反映させるために、データ自体の書き換えやクリーンアップが必要となる。
データ移行	この場合、データは最新のデータベース管理システムの管理下に移される。データは別々のファイルに保存されていたり、古いタイプのDBMSで管理されていたりする。この状況を図28.11に示す。

そのため、彼らはシステムと関連データを再設計することにした。そこで彼らは、システムと関連データを再構築することにした。

3. *アーキテクチャの進化* 集中型システムを分散型アーキテクチャに移行する場合、そのアーキテクチャの中核は、リモートクライアントからアクセスできるデータ管理システムであることが不可欠である。そのためには、データを個別のファイルからサーバーのデータベース管理システムに移行するための、大規模なデータ再構築作業が必要になるかもしれない。分散プログラムアーキテクチャへの移行は、組織がファイルベースのデータ管理からデータベース管理システムへの移行を決定したときに開始される。

プログラム・リエンジニアリングと同様に、データ・リエンジニアリングが必要とされる理由を反映した、データ・リエンジニアリングへの様々なアプローチがある。これらを図28.10に示す。

Rickets ら(Rickets, DelMonaco et al., 1993)は、複数の協調プログラムによって構成されるレガシーシステムで発生しうるデータに関する問題のいくつかを説明している:

1. *データの名前付けの問題* 名前が暗号化されていて理解しにくい場合がある。システム内の異なるプログラムで、同じ論理エンティティに

異なる名前（同義語）が付けられることが

ある。同じ名前が異なるプログラムで異なる意味に使われることがある。

2. *フィールド長の問題* レコードのフィールド長がプログラムで明示的に割り当てられている場合の問題である。同じ項目でもプログラムによって異なる長さが割り当てられたり、フィールドの長さが現在のデータを表すには短すぎる場合がある。この問題を解決するために、場合によっては他のフィールドが再利用され、システム内のプログラム間で名前付きデータ・フィールドの使い方に一貫性がなくなることがある。

3. *レコード編成の問題* 同じ実体を表すレコードでも、プログラムによって編成が異なることがある。これは、COBOLのようにレコードの物理的な構成がプログラマによって設定され、ファイルに反映される言語では問題となる。C++やJavaのように、レコードの物理的整理がコンパイラの責任である言語では問題にならない。

図28.10 データ・リエンジニアリングのアプローチ

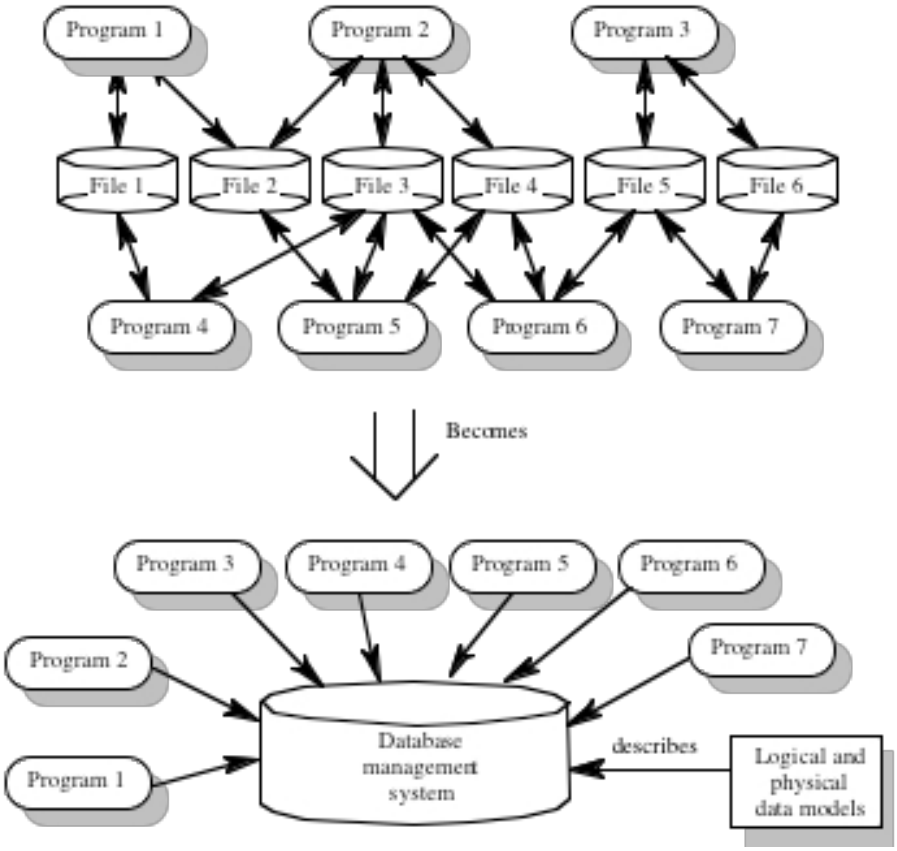


図28.11 データ移行

4. *ハードコードされたリテラル* 税率のようなリテラル値（絶対値）は、シンボリック名を用いて参照するのではなく、プログラムに直接含まれる。

5. *データ辞書が*ない 使用されている名称、その表現、使用方法を定義したデータ辞書がない可能性がある。

一貫性のないデータ定義と同様に、データ値もまた一貫性のない方法で保存されている可能性がある。データ定義が再設計された後、データ値も新しい構造に適合するように変換されなければならない。Ricketsらも、データ値の不整合の可能性について述べている。これらを図28.12に示す。

データのリエンジニアリングの前に、データを使用するプログラムの詳細な分析が不可欠である。この分析は、プログラム中の識別子の機能を発見し、名前付き定数に置き換えるべきリテラル値を発見し、埋め込まれたデータ検証ルールとデータ表現変換を発見することを目的とすべきである。相互参照アナライザーやパターンマッチャーなどのツールが、この分析に役立つ。データ項目が参照されている場所と、それぞれの参照に加え

の命名、データフォーマットの再整理、およびデータ・リエンジニアリングのプロセスを示している。

られる変更を示す一連の表を作成する。

図28.13は、データ定義の変更、リテラル値

データの不整合	説明
一貫性のないデフォルト値	プログラムによって、同じ論理データ項目に異なるデフォルト値が割り当てられる。このため、データを作成したプログラム以外では問題が発生する。欠損値に有効なデフォルト値が割り当てられると、問題はさらに深刻になる。その場合、欠落したデータを発見することはできません。
単位に一貫性が	同じ情報でもプログラムによって単位が異なる例えば、米国や英国では、体重データは古いプログラムではポンドで表示されるが、最近のシステムではキログラムで表示されることがある。この種の大きな問題は、欧州単一通貨の導入に伴ってヨーロッパで発生した。
一貫性のない検証規則	レガシー・システムは各国の通貨単位に対応するように書かれてため、データをユーロに変換しなければならない。 プログラムによって適用されるデータ検証ルールは異なる。によって書き込まれたデータは あるプログラムでは拒否されるかもしれない。これは、データ検証
一貫性のない表現セマンティクス	ルールの変更に合わせて更新されていない可能性のあるアーカイブデータにとって、特に問題となる。 プログラムは、項目の表現方法に何らかの意味を想定している。例
負の値の一貫性のない処理	えば、プログラムによっては、大文字のテキストはアドレスを意味すると仮定する場合がある。プログラムは異なる慣例を使用する可能性があり、そのため意味的に有効なデータを拒否する可能性がある。 プログラムによっては、常に正でなければならないエンティティの負の値を拒否するものもある。しかし、負の値として受け入れたり、負の値として認識できずに正の値に変換したりするプログラムもある。

変換されたデータ値。変更サマリー表は、行われるすべての変更の詳細を保持する。そのため、データ・リエンジニアリングプロセスのすべての段階で使用される。

このプロセスのステージ1では、理解しやすくするためにプログラム内のデータ定義を修正する。データ自体はこれらの修正によって影響を受けることはない。awk (Aho, Kernighan et al., 1988)のようなパターンマッチングシステムを使って定義を検索して置き換えたり、データのXML記述(St Laurent and Cerami, 1999)を作成し、これをデータ変換ツールの駆動に使ったりすることで、このプロセスをある程度自動化することは可能である。

しかし、このプロセ

スを完了させるためには、ほとんどの場合手作業が必要である。単にプログラム内のデータ構造定義の理解しやすさを向上させることが目的であれば、データのリエンジニアリングプロセスはこの段階で止まってしまうかもしれない。しかし、上述したようにデータ値に問題がある場合は、プロセスのステージ2に入ることができる。

図28.12 データ値の不整合

組織がプロセスのステージ2に進むことを決定した場合、ステージ3のデータ変換に取り組むことになる。これは通常、非常にコストのかかるプロセスである。旧組織と新組織の知識を組み込んだプログラムを書かなければならない。これらは古いデータを処理し、変換された情報を出力する。この変換を実行するために、ここでもパターンマッチングシステムが使われる。

キーポイント

- システムのリエンジニアリングの目的は、システムの構造を改善し、理解しやすくすることである。そのため、将来のシステム・メンテナンスのコストを削減する必要がある。

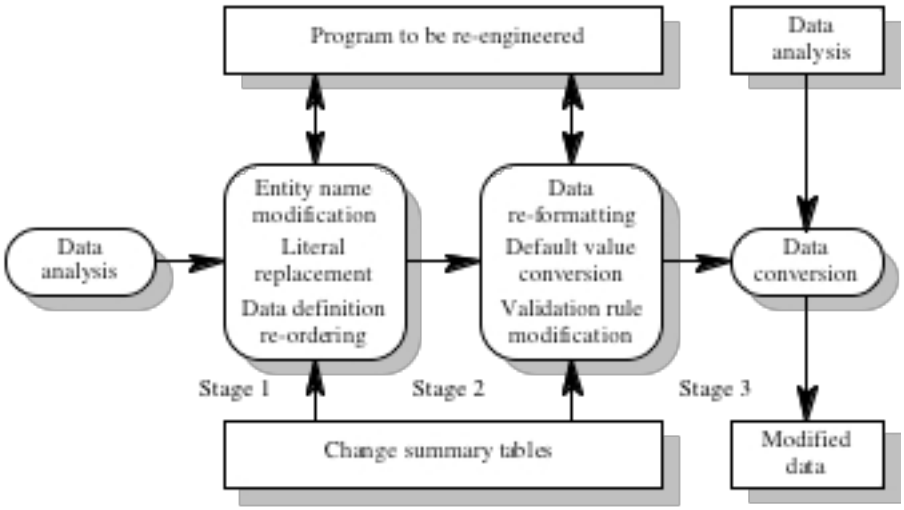


図28.13 データのリエンジニアリング・プロセス

- リエンジニアリングプロセスには、ソースコードの翻訳、リバースエンジニアリング、プログラム構造の改善、プログラムのモジュール化、データのリエンジニアリングなどの活動が含まれる。
- ソースコード翻訳とは、あるプログラミング言語で書かれたプログラムを別の言語に自動的に変換することである。元のプログラミング言語が古くなった場合に必要になることがある。
- リバースエンジニアリングとは、ソースコードからシステムの設計や仕様を導き出すプロセスのことである。このプロセスを支援するために、プログラムブラウザなどのツールが使用されることがある。
- プログラム構造の改善には、ゴトのような構造化されていない制御構文をwhileループや単純な条件文に置き換えることが含まれる。これは自動的に行うことができる。
- プログラムのモジュール化とは、プログラムのソースコードを再編成し、関連する項目をグループ化することである。これにより、理解や変更が容易になる。
- レガシーシステムのプログラムによるデータ管理に一貫性がないため、データ・リエンジニアリングが必要になることがある。データ・リエンジニアリングの目的は、すべてのプログラムが共通のデータベースを使用するようにリエンジニアリングすることである。
- 既存のデータを何らかの新しいフォーマットに変換しなければならない場合、データ・リエンジニアリングのコストは大幅に増加する。

さらに読む

低さが与える影響について論じた論文が多数含まれている。(G. K. TayiとD. P.Ballou, *Comm.ACM*, 41 (2), Feb. 1998）。

Software Re-engineering これはIEEEのチュートリアルで、1992年以前に発表されたリエンジニアリングに関する重要な論文のほとんどを含んでい ま す 。 論文の多くは

データの質の検証」。この特別セクションには、データ品質の問題や、データ品質の

本章で参照した論文は、その中に再掲されている。(R. S. Arnold, IEEE Press, 1994）。

国防総省のレガシーシステム：これは、レガシーシステムで発生する実際的な問題についての良い説明である。この論文は、類似しているが互換性のないデータを管理するシステムが組み合わされた場合の、データのリエンジニアリングに焦点を当てている。リバースエンジニアリングに関するこの特集号の他の論文も関連している。(P. Aiken, A. Muntz, R. Richards, *Comm.ACM*, 37 (5), May 1994）。

エクササイズ

28.1 どのような状況であれば、ソフトウエアはリエンジニアリングではなく、破棄して書き直すべきだと思いますか?

28.2 あなたが知っている任意の2つのプログラミング言語の制御構文（ループと条件分岐）を比較しなさい。つの言語の制御構文を、もう1つの言語の同等の構文に変換す る 方法を簡単に説明しなさい。

28.3 図28.14の構造化されていないルーチンを構造化された同等のルーチンに変換し、そのルーチンが何をす ることになっているかを調べる。

28.4 構造化されていないプログラムでモジュールを見つけるのに役立つガイドラインを書きなさい。

28.5 図28.14のプログラムで使用される変数に意味のある名前を提案し、その名前に対応するデータ辞書項目を作成する。

28.6 あるタイプのデータベース管理システムから別のタイプ（階層型からリレーショナル型、リレーショナル型からオブジェクト指向型など）にデータを変換する場合、どのような問題が生じる可能性があるか?

28.7 システムのソースコードを自動的に解析してシステム仕様を復元することが不可能な理由を説明しなさい。

28.8 データの劣化がもたらす問題について、例を挙げて説明しなさい。

```
ルーチン BS (K, T, S, L)
B:= 1
NXT:　S >= B なら CON
L = -1
STPへ
con: l := 整数 (b / s) l := 整数
((b+s) / 2)
もしT (L) = Kなら、return
if T(L) > K then goto GRT
B := L+1
NXTへ
GRT: S := L-1
NXT
STP: 終
了
```

図28.14 構造化されていないプログラム

データのクリーンアップの過程で取り組むべきこと。

28.9 日付が2桁で表される2000年問題は、多くの組織にとってプログラム保守の大きな問題となった。この問題はデータ・リエンジニアリングにどのような影響を与えたのだろうか?

28.10 ある企業では、アプリケーションのリエンジニアリングに携わるフリーランスのプログラマーに対して、同様の企業との契約ができないような契約条件を課すことが日常的に行われている。その理由は、リエンジニアリングは必然的にビジネス情報を明らかにするからである。契約終了後は請負業者に対して何の義務も負わないことを考えると、これは企業として妥当な立場でしょうか?