Rational Unified Process を使用した大規模システム開発

Maria Ericsson

Rational Software ホワイト ペーパー

TP 156

Rational™
the software development company

目次

歴史

このホワイト ペーパーは、"Systems of Interconnected Systems" (Ivar Jacobson、Karin Palmkvist、Susanne Dyrhage 著、発行: ROAD、1995 年 5 ～ 6 月) で執筆したものです [1]。 このホワイト ペーパーでは、いくつかの大規模システム開発プロジェクトの情報を利用しており、また Rational Unified Process バージョン 5.1 [2] と UML (統一モデリング言語) [3] と関連付けることも意図されています。

相互接続されたシステムで構成されるシステム

大規模なシステム開発は、非常に複雑な状況になります。 理解しなければならない成果物群が複雑であるだけでなく、大量のリソースを管理することが必要なためにオーバーヘッドも発生します。 このホワイト ペーパーでは、複雑さのために加わるオーバーヘッドの管理に役立つアーキテクチャ パターンについて解説します。 [4] で説明している要素の中では、アーキテクチャ パターンは、**相互接続されたシステムで構成されるシステム (system of interconnected systems)** と呼ばれています。

この構造は、指揮システムや制御システム、または高度に統合された IT ソリューションのような、非常に規模が大きくて複雑なシステムの構築に役に立ちます。 この種の「スーパー システム」は、普通、複数の独立した部分に分かれており、それぞれ独立したシステムとして個別に開発されます。 スーパー システムは、一群の相互接続されたシステムによって実装され、これらのシステム同士が通信することでスーパー システムの機能が実現されます。これらのシステムの 1 つは全体的な機能を表しており、ここではそれを**上位システム (superordinate system)** と呼びます。 その他のシステムは全体の一部を表しており、ここでは**下位システム (subordinate systems)** と呼びます。 上位システムは、それを実装する下位システムとは明らかに異なっています。 異なるタイプのシステム間の関係は区別されます。上位システムから見ると、下位システムはサブシステムです。図 1 を参照してください。

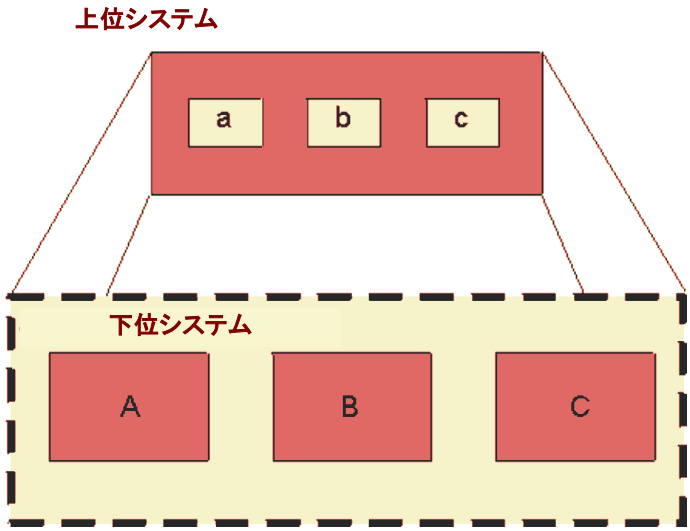

図 1: 上位システムの仕様は、相互接続されたシステムで構成されるシステムによって実装されます。その場合、システム A、B、C は、それぞれ、上位システムのサブシステム a、b、c の実装です。

上位システムを下位システムから切り離すことには、いくつかの利点があります。

- 販売や納品などのすべてのライフサイクル アクティビティを通じて、下位システムを個別に管理できます。
- 下位システムをほかの相互接続されたシステムで構成されるシステムに組み込むことで、下位システムを使用して別の上位システムを簡単に実装できるようになります。
- 相互接続されたシステムで構成されるシステムかどうかにかかわらず、常に意識してシステムの構築を開始するわけではありません。 「簡単な」システム ビューで作業を始めて、ライフサイクルのかなり後になってから、相互接続されたシステムで構成されるシステムというパターンを適用する必要があるかどうかを決定することもできます。

- 上位システムの新バージョンを開発しなくても、下位システムを内部的に変更することができます。上位システムの新バージョンが必要になるのは、主要な機能を変更するときだけです。

個々の下位システムには一連の成果物が関連付けられており、相互間に明確な追跡可能性が確立されています。 下位システムの成果物群から上位システムの対応する成果物群への追跡可能性も維持されています。 各下位システムは、独自のライフサイクル フェーズ (方向づけ、推敲、作成、移行) を持つ独立した開発プロジェクトとして管理できます。

構築している「スーパー システム」が非常に大きい場合は、下位システムをさらに分割し、相互接続されたシステムで構成されるシステムとして扱うことが必要になる場合があります。

ソフトウェア開発ライフサイクル

Rational Unified Process では、開発ライフサイクルは、管理面と開発面という 2 つの観点から提示され検討されます。図 2 を参照してください。

管理の観点からは、4 つのライフサイクル フェーズを経て、システム、つまりシステムの新しい世代を開発します。開発の観点からは、システムの複数のバージョンを反復的に開発しながら、少しずつ完成度を高めていきます。 反復の間に実行する作業は、Rational Unified Process では、基本ワークフロー (作業分野) の集合としてグループ化されています。 各基本ワークフローではシステムのある側面を記述することが重視され、結果として、システムのモデルまたは一群の文書が作成されます。

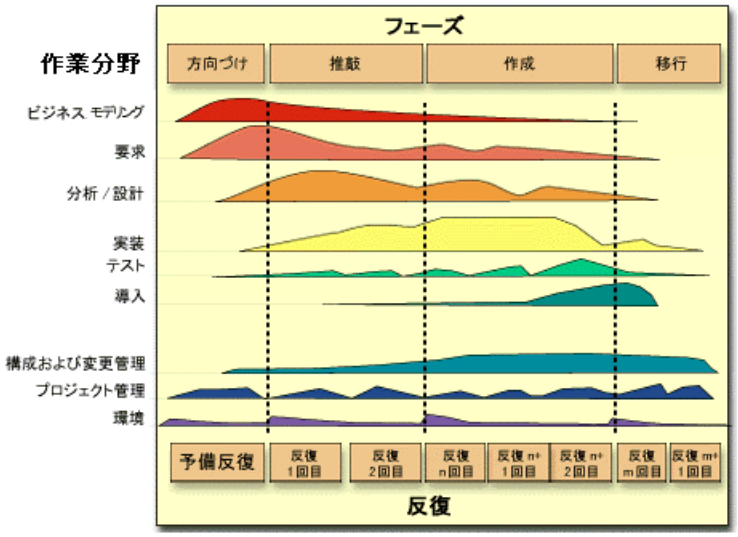

図 2: 反復モデル

相互接続されたシステムで構成されるシステムにこのモデルを適用すると、上位システムとその下位システムはそれぞれ、すべて固有のライフサイクルを経て開発され、それぞれが独立したプロジェクトとして扱われることもよくあります。

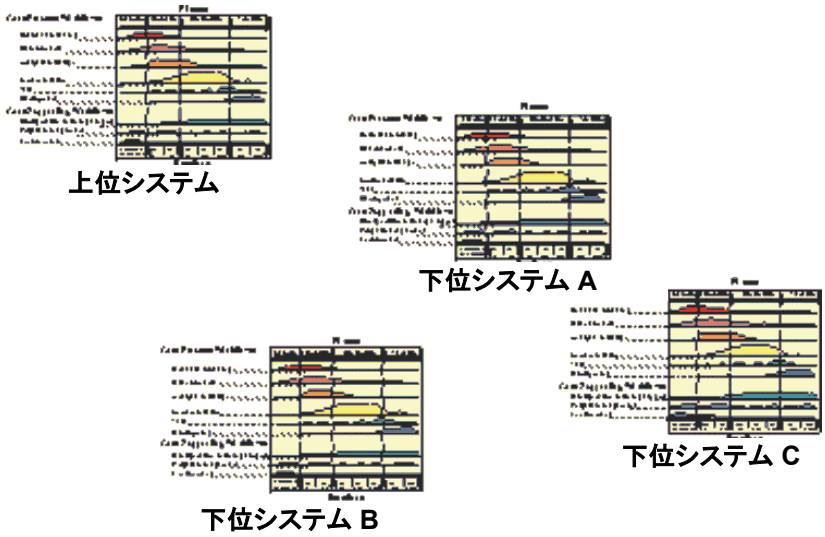

図 3: 上位システムも下位システムも、それぞれ独自のライフサイクルを経て開発されます。

当然のことながら、ライフサイクル間には依存関係があります。 依存関係を正しく管理することは、相互接続されたシステムで構成されるシステムを開発する際の課題の 1 つです。 依存関係には以下の種類があります。

- ライフサイクルは時間的に依存しています。 上位システムのライフサイクルが最初に開始します。 上位システムに対して少なくとも 1 回の反復が実行され、下位システムのインターフェイスがある程度安定したら、下位システムのライフサイクルを開始できます。 実際、上位システムの反復を少なくとも 1 回実行するまでは、どれが下位システムなのかさえわからない場合があります。
- 下位システムのインターフェイスが安定したら、上位システムのライフサイクルは保守に移ることができます。 つまり、下位システムのインターフェイスの変更を必要とする問題が発生した場合を除き、積極的な開発は行われません。
- 下位システムのインターフェイスは、上位システムの開発者が所有します。 インターフェイスの詳細については、[3] と [5] を参照してください。
- 下位システムのインターフェイスを実装するクラスは、下位システムの開発者が所有します。

システム開発のワークフローと成果物

当然の前提として、上位システムだけでなく下位システムも、複合的でないシステムに対する標準的な成果物群とワークフローを使用して開発できます。 開発方法について説明する前に、成果物とワークフローについて触れておく必要があります。 図 4 に示すように、Rational Unified Process には 5 つの基本プロセス ワークフローがあります。 次に、これらについて説明します。

- ビジネス工学 - 目的は、システムを使用する組織を評価し、システムによって解決すべきニーズと問題に対する理解を深めます。 結果はビジネス ユース ケース モデルとビジネス オブジェクト モデルです。 このワークフローは、場合によっては省略してもかまいません。 システムが導入される組織が非常に単純化されていると、システム導入によるメリットがない場合があります。
- 要求 - 目的は、要求を獲得して評価し、使いやすさに重点を置きます。 結果として、システムと関係を持つ外部ユニットを表すアクターを含むユース ケース モデル、アクターに対して測定可能な価値を生み出すトランザクション シーケンスを表すユース ケースが作成されます。

- 分析/設計 - 目的は、想定される実装環境と、それがシステムの構築に及ぼす影響について調査します。このワークフローの結果は、オブジェクト モデル (設計モデル) です。これには、ユース ケースのフローを実行するためにオブジェクトが通信する方法を示すユースケースの実現が含まれます。また、提供される操作に関するクラスとサブシステムの責務を指定する、クラスやサブシステムのインターフェイス定義が含まれる場合もあります。 このオブジェクト モデルは、実装言語や分散などに関しては、実装環境にも適合しています。 分析の結果を独立したモデルとして考えると扱いやすい場合があります。その場合のモデルは、分析モデルと呼ばれます。

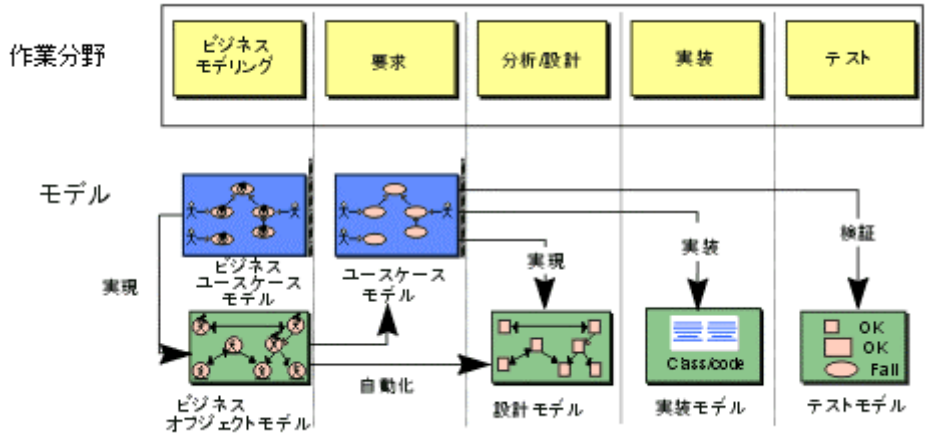

図 4: 個々の基本プロセス ワークフローは、特定のモデル セットと関連付けられています。

- 実装 - 目的は、規定されている実装環境にシステムを実装します。 結果として、ソース コード、実行可能なプログラム、ファイルが作成されます。
- テスト - 目的は、システムが意図したとおりのものであり、実装にエラーがないことを確認します。結果として、システムは動作が保証されて、納品できる状態になります。

相互接続されたシステムで構成されるシステムの開発

実際に行わなければならないことは、まず、システムの責務を複数のシステムに分散させる方法を定義する必要があります。これらのシステムは、明確に定義された責務のサブセットを受け持ちます。 つまり、これらの下位システム間のインターフェイスを定義することが主な目標です。 この目標が達成されれば、後は「分割して攻略する」原則に従って、個々の下位システムに対する作業を個別に進めることができます。 したがって、実装後のテストを除けば、システム全体に対して行わなければならないことはこれだけです。

分解の基準

それでは、システムを分解して、相互接続されたシステムで構成されるシステムにするかどうかは、どのようにして決定すればよいのでしょう。 考慮する必要のある特性は次のとおりです。

- 規模が大きく相当複雑なシステムの場合は、問題を小さな要素に分割することで、容易に理解できるようになります。
- 扱っているシステムは物理的に分離しているでしょうか。 長年にわたって使用されてきたシステムやアーキテクチャを扱う際には、このような場合がよくあります。
- 分解することで、システムの部分間のインターフェイスを自然かつ限定的に定義できるようになります。
- システムの一部を主要な COTS 製品 (民生品) を使用して実装する場合があります。 分解によって、COTS 製品の使用方針を明確にすることができます。

- 分割を行うことで、分散した開発組織を最大限に活用し、地理的に離れた場所にいる複数のチーム間の作業を明確に分けることができます。

一方で、次のようなリスクを考慮する必要があります。

- 分解を進めすぎると、問題の細部のために全体的な問題が隠されてしまう場合があります。
- 物理的に離れたシステムまたはチームを使用すると、どのような形式の再利用もできず、融通の利かないバラバラのシステム (ストーブ煙突型システム) ができあがるリスクがあります。

組織

上記のようなリスクを緩和するには、開発作業全体を監督する役割を持つグループを設けることが重要です。 このグループはアーキテクチャ チームと呼ばれることが多く、以下の重要な問題を中心に扱います。

- 全体的なアーキテクチャが定義されていて、下位システムがそれに従っていること。
- 下位システム間で経験が再利用され共有されるように、適切な注意が払われていること。
- どのような成果物を作成する必要があり、下位システムの成果物と上位システムの成果物の間にどのような関係があるのかが、明確に理解されていること。
- 効果的な変更管理戦略が定義されていて、すべてのチームがそれに従っていること。

アーキテクチャ チームは、場合によって、上位システムの開発を受け持ちます。 組織についての詳細な説明は、[6] を参照してください。

上位システムのライフサイクル

まず、場合によっては**ビジネス工学**を実施して、システムの状況に対する理解を深めます。 ビジネス工学は次のような場合に有効です。

- 開発者が組織についての理解を深める必要がある。
- 組織自体にビジネスの実施方法と語彙に関する不一致があり、プロセスを調整する必要がある。
- ビジネス リエンジニアリング作業と併せて、ソフトウェア工学作業が行われる。

[6] を参照してください。

作業の結果として、ビジネス ユース ケース モデルとビジネス オブジェクト モデルが作成されます。 代わりに、ビジネス工学の一部だけを実施し、ビジネス領域の中心となる概念だけに注目して、ビジネス オブジェクト モデルにそれを記述してもかまいません。 これは普通、ドメイン モデリングと呼ばれます。

ビジネス モデルによって「下準備」ができたなら、システム全体に対する**要求**の入手を始める必要があります。 ほかのシステムと同じように、相互接続されたシステムで構成されるシステムについても要求モデリングを行う必要があります。 結果を示すには、ユース ケース モデルが非常に適した方法です。[7] を参照してください。 この上位ユース ケース モデルの最も直接的な見方は、システムの動作に関する要求がモデルに完全に取り入れられていると考えるものです。 しかし、このような見方ができることはほとんどありません。 ほかのシステムでシステムを実装する必要があるということは、全体的なシステムはおそらく非常に複雑なものです。 したがって、このレベルですべてを網羅しようとするのは賢明ではありません。 つまり、上位ユース ケース モデルは通常、システムの機能要求に関して、完全ではあっても単純化された様子を表しています。 詳細なモデリングは下位システムの個々の実装で行うので、このレベルではあまり詳細に行う必要はありません。 また、一部のサブシステムが含まれていない上位ユース ケースでは、多くの要求が示されていないこともよくあります。 このような要求は、サブシステムに対して「ローカル」なものであると言えます。

分析/設計の目的は、システムの堅牢なアーキテクチャを実現することであり、当然、相互接続されたシステムで構成されるシステムにもこれは非常に重要です。 上位システムの開発者は、下位システムの堅牢な構造を実現する必要がありますが、下位システムの内部構造を考える必要はまったくありません。 したがって、サブシステムを使用してシステムを小さな部分に分割する方法をモデリングします。 適切なサブシステムの集合を作成し、上位システムの責務をこれらのサブシステムに分散させる方法について最初の案を得るには、分析モデルを作成します。 分析クラスは、上位レベルのユース ケースが実行されるときにシステム内でになう役割を表していなければなりません。 したがって、分析モデルでは、上位レベルのユース ケース モデルと対比して、完全なオブジェクト構造が簡略化して表されます。

機能的に関連のある分析クラスは、サブシステムにグループ化されます。 このように、機能条件だけに基づき、たとえば分散のような要求はまったく考慮されていないという意味において、完全なサブシステム構造が得られます。しばしば大きな影響力を持つ要素となるのが、既存のシステムの存在です。 既存のシステムは、分析モデルで定義されている責務の一部または多くを満たしている場合があります。 このようなシステムが存在していると、既存の機能を最大限に再利用できるよう、分析で発見された責務の再分割が必要になる場合もあります。

設計の結果におけるサブシステム構造は、分析の過程で機能条件に基づいて定義されたものとは大きく異なる場合があります。 したがって、最終的には設計サブシステムの構造を作成し、それぞれ下位システムによって実装されることになります。図 5 を参照してください。 このような個々のシステムに対する開発作業を個別に独立して続けられるようにするには、各サブシステムに対してインターフェイスを定義します。 実際、インターフェイスは下位システムの開発に関するルールとなるものなので、インターフェイスの定義は上位レベルにおいて行われる最も重要な作業です。 設計クラスは定義されていないので、設計サブシステムのインターフェイスの定義だけを行います。

システムの特定の技術的側面を調査するために多少のプロトタイピング作業を行う場合があることを除けば、上位システムのライフサイクルにおいては**実装**は行われません。

最後のワークフローは**テスト**です。この場合のテストとは、異なる下位システムを組み合わせたときの統合テスト、相互接続されたシステムが協調して、すべての上位ユース ケースを仕様に従って実行することを確認するテストのことです。

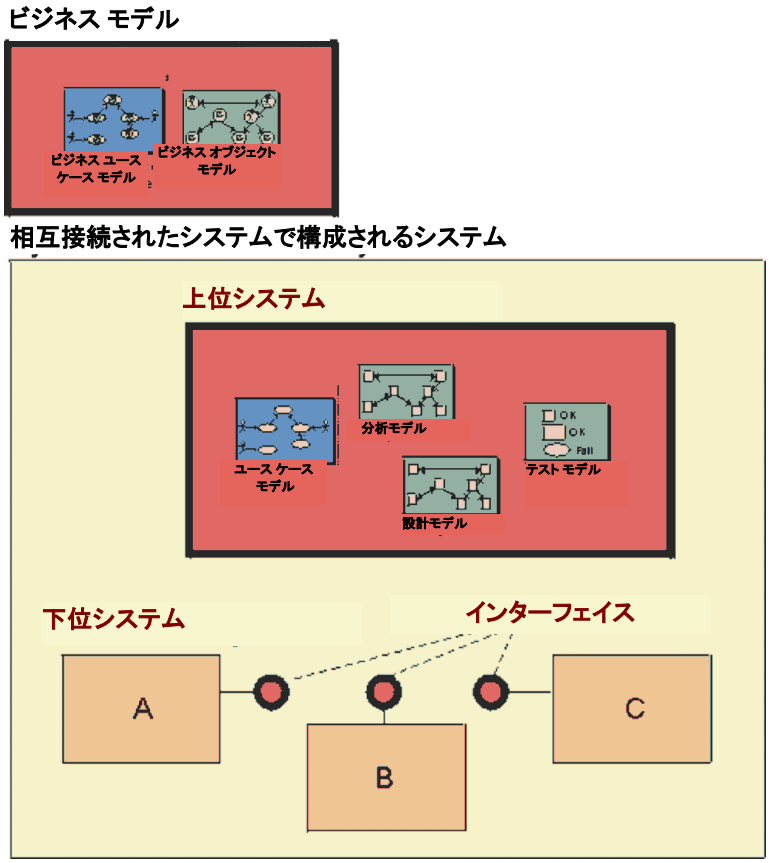

図 5: 上位システムはモデルの集合によって記述され、上位レベルの設計モデルで定義されているサブシステムは下位システムによって実装されます。 下位システムに対するインターフェイスは、上位システムによって所有されます。

上位システムについての作業を進める方法を示すため、反復計画のサンプルとして、上位システムのライフサイクルの方向づけフェーズにおける反復に関する計画と、推敲フェーズにおける反復に関する計画を示します。 反復計画書の記述には、アクティビティ図を使用します。 これらの図のアクション状態は、Rational Unified Process で定義されているワークフローの詳細に対応しています。

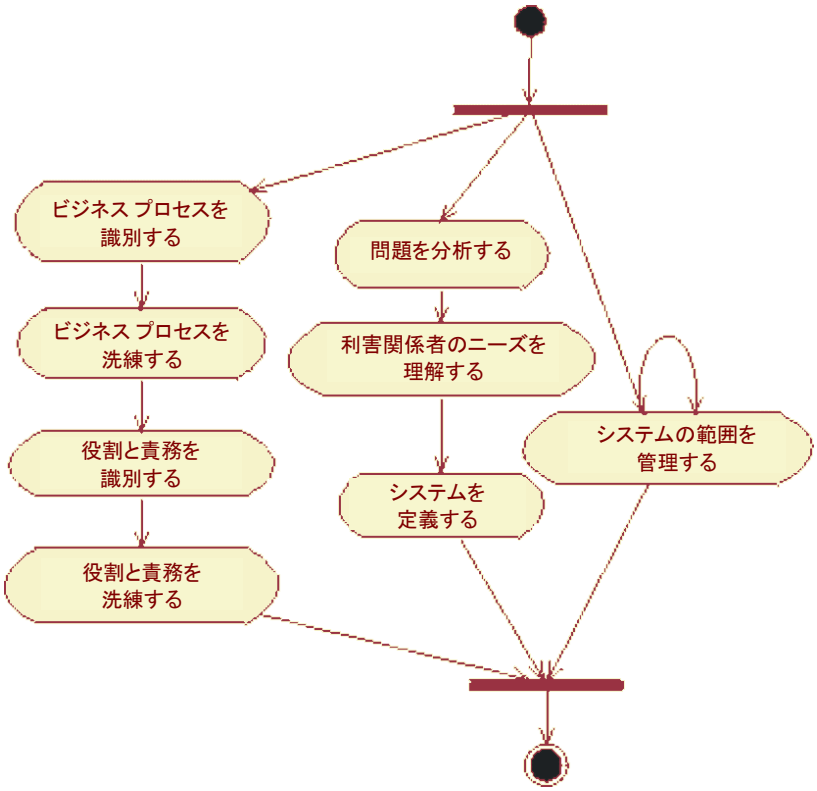

図 6: 上位システムの方向づけ反復計画の例を記述したアクティビティ図

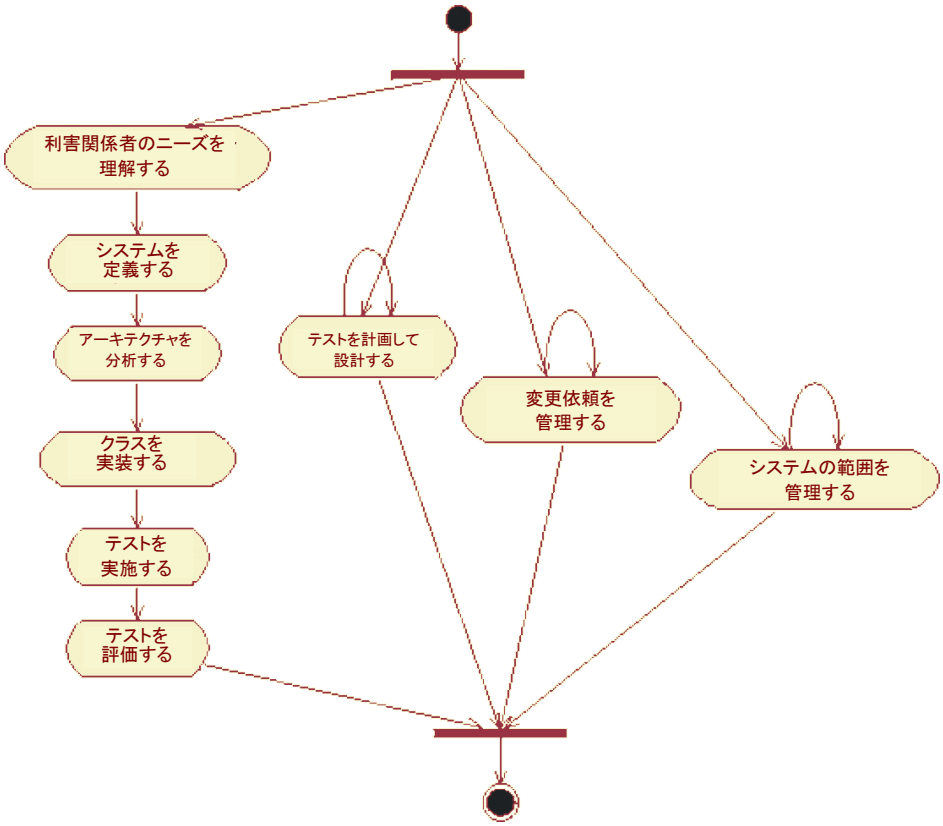

図 7: 上位システムの推敲反復計画の例を記述したアクティビティ図。システムの技術的側面を調査するためにプロトタイプの限定的な実装を行う場合があるので、アクション状態「クラスを実装する」をここに含めます。

下位システムのライフサイクル

個々の下位システムは通常の方法で開発されます。それぞれのシステムがアクターとして関係を持つほかのシステムは、ブラック ボックスと見なします。 前に示したように、このような各システムに対しては、通常の一連の作業を実施し、通常のモデル群を作成します。 上位のレベルにおいてあらゆる細部までモデルが定義されている場合は、異なるレベルのモデル間で完全な反復が得られますが、先に触れたように、このようなケースは実際にはほとんどありません。

下位システムに対しては、**要求**ワークフローを実行します。 上位システムのインターフェイスとユース ケースが、下位システムの境界とアクターを理解するための主要な入力情報になります。

下位システムの**分析/設計**を行うときは、上位システムで定義されているインターフェイスが、上位レベルのユース ケースと共に「境界条件」になります。

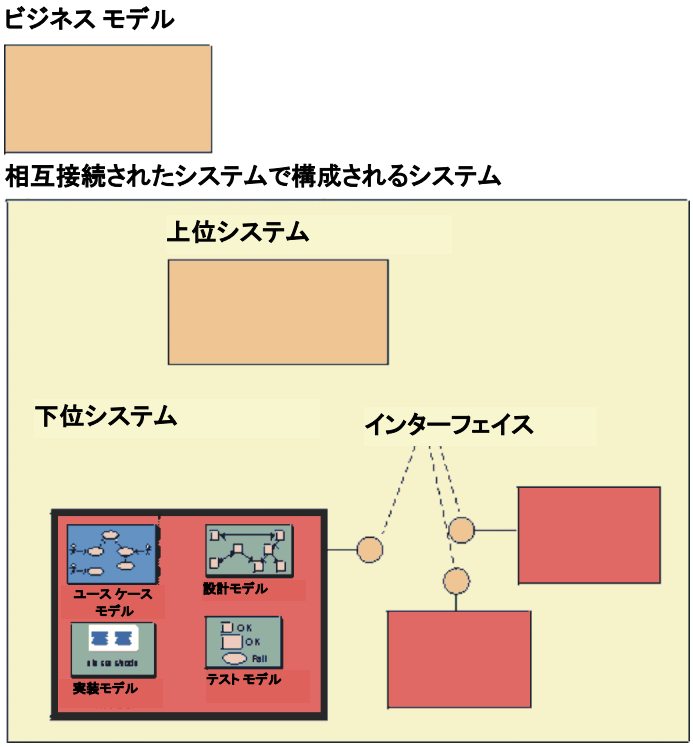

図 8: 下位システムは固有のモデル群で記述されます。

下位システムでの作業方法を示すため、ここでは下位システムのライフサイクルから反復計画の 2 つのサンプルを紹介します。

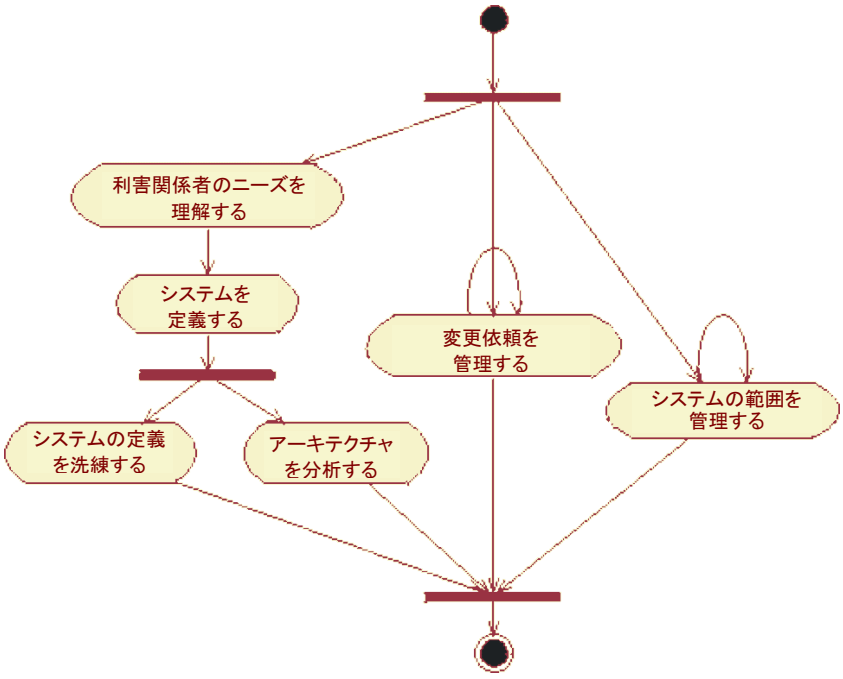

図 9: 下位システムの方向づけ反復計画のサンプル。 実行可能ファイルが作成されないので、これは不完全な反復です。

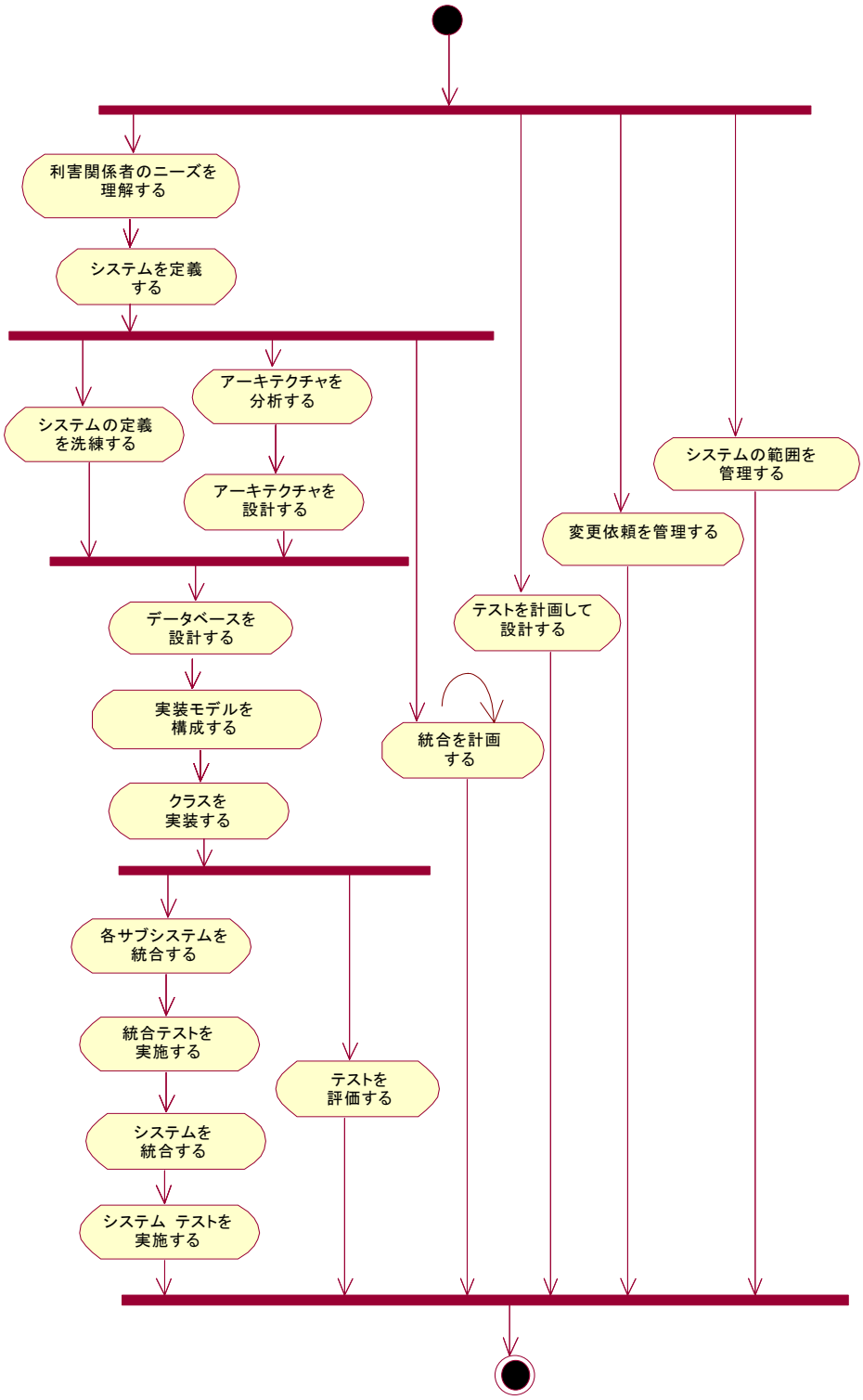

図 10: 下位システムの推敲反復計画のサンプル。 推敲の目的は、システムの定義とアーキテクチャを洗練して完成させることです。

相互接続されたシステムで構成されるシステムのユース ケース

相互接続されたシステムで構成されるシステムでは、上位と下位のシステムごとに 1 つずつユース ケース モデルを作成する必要があります。 これらのモデルは、以下のようにして相互に依存しています。

- 上位システムにおける上位レベルのユース ケースは、必ずではありませんが、通常はサブシステムに分割されます。 分割された各サブシステムは、下位システムに関するモデルのユース ケースになります。図 11 を参照してください。
- 1 つの下位システムから見ると、ほかの下位システムはユース ケース モデルにおけるアクターになります。図 12 を参照してください。

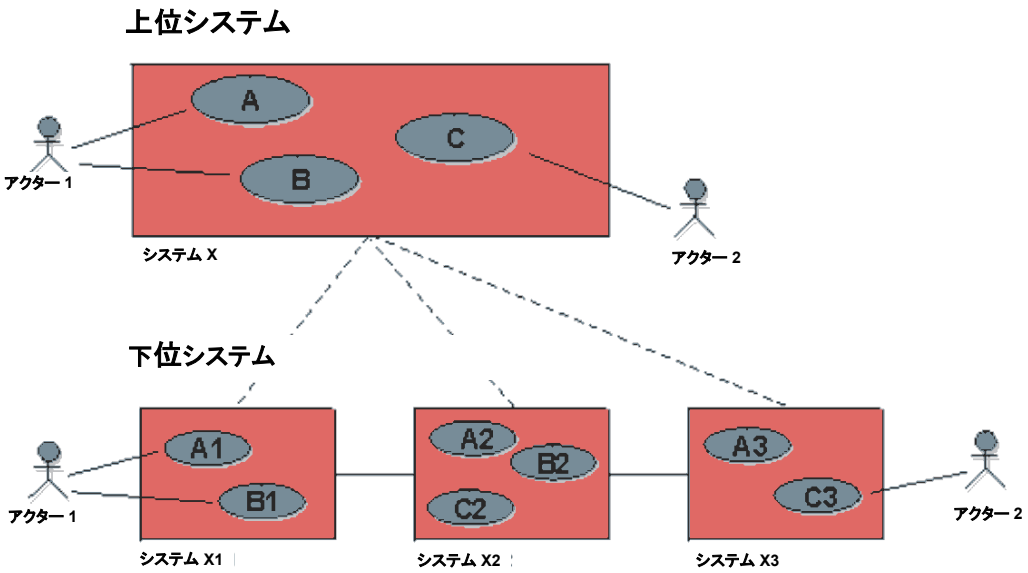

図 11: 上位システムにおける上位レベルのユース ケースと、下位システムにおける詳細なユース ケースとの関係

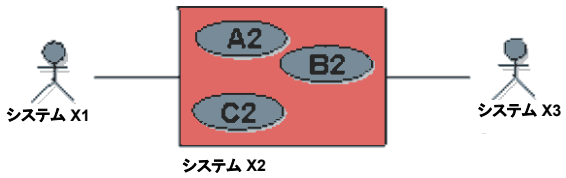

図 12: 下位システム X2 のユース ケース モデルでは、ほかの下位システム X1 と X3 はアクターと見なされます。

上位システムを記述するユース ケースについては、特別な考慮事項がいくつかあります。 各下位システムではすべての要求をある程度記述し直すことになるので、上位のユース ケースをあまり詳細に記述しても意味がありません。 通常は、上位レベル ユース ケースのイベントのフローに対してステップごとに概要を記述すれば十分であり、文章で詳細に説明する必要はありません。

上位のユース ケース モデルでは、ユース ケースの関係 (汎化、拡張、包含) は使用しません。 一般に、以下のような理由から関係を使用してもメリットはありません。

- 上位レベルのユース ケースを詳細に記述することはないので、複数の場所に出現するテキストに対する配慮は必要ありません。
- 上位レベルのユース ケースを下位システムに「分割する」ときには、いずれにしても情報を構成することになります。 これをほかの構成メカニズムと組み合わせると、混乱する可能性があります。

ただし、これには重要な例外があります。それは、相互接続されたシステムで構成されるシステムにおいて、再利用可能なコンポーネントを発見しようとしている場合です。 上位ユース ケース モデルを構成して汎用的なユース ケースを発見することは、再利用可能なコンポーネントを発見するための強力な方法です。 このトピックの詳細については、[6] を参照してください。

相互接続されたシステムで構成されるシステムにおける設計モデル

相互接続されたシステムで構成されるシステムの各システムは、上位と下位のどちらについても、独自の設計モデルを持つ必要があります。 設計モデル同士は、次のようにして関連付けられます。

- 上位システムに関する設計モデルのサブシステムは、下位システムの境界を定義しています。
- 上位システムのサブシステムについて定義されている操作は、下位システムとのインターフェイスの定義に対する入力です。

上位システムの設計モデルは、下位の設計モデルほど詳細に記述されません。 以下のものを作成します。

- サブシステム (簡単に記述したもの)。
- ユース ケースの実現 (サブシステムが連携動作する方法に関して)。上位レベルでのユース ケースの実現を文書化する一般的な方法は、シーケンス図を作成することです。 シーケンス図を作成することで、上位レベルのユース ケースの下位システムへの「分割」を定義します。図 13 を参照してください。
- サブシステムの操作。
- サブシステムのインターフェイス定義。

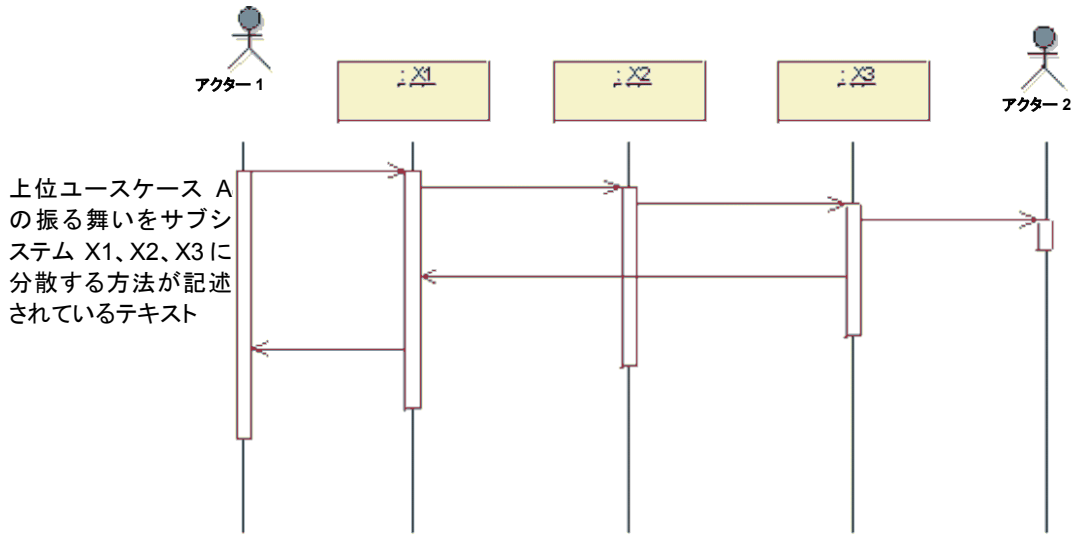

図 13: 上位ユース ケース A の実現に関するシーケンス図

相互接続されたシステムで構成されるシステムにおける情報セット

成果物の管理方法を理解し、その依存関係を正しく理解することに、ほとんどの組織が多くの労力を費やします。前の項では特に、ユース ケース モデルと設計モデルの上位モデルと下位モデルの間にある依存関係について説明しました。 依存関係については、考慮する必要のある一般的な問題もあります。

システムがライフサイクルを進んで行くにつれて、情報セット [8] として編成できる成果物が作成されます。図 14 を参照してください。 これらのセットは、「一緒に」発展していく成果物に基づいて編成されます。

- 構築しているアプリケーションの種類に基づいて各セットの厳密な内容はカスタマイズできますが、セット自体は変化しません。
- 効果的に成果物間の追跡可能性を維持できるよう、セット間の依存関係を理解する必要があります。

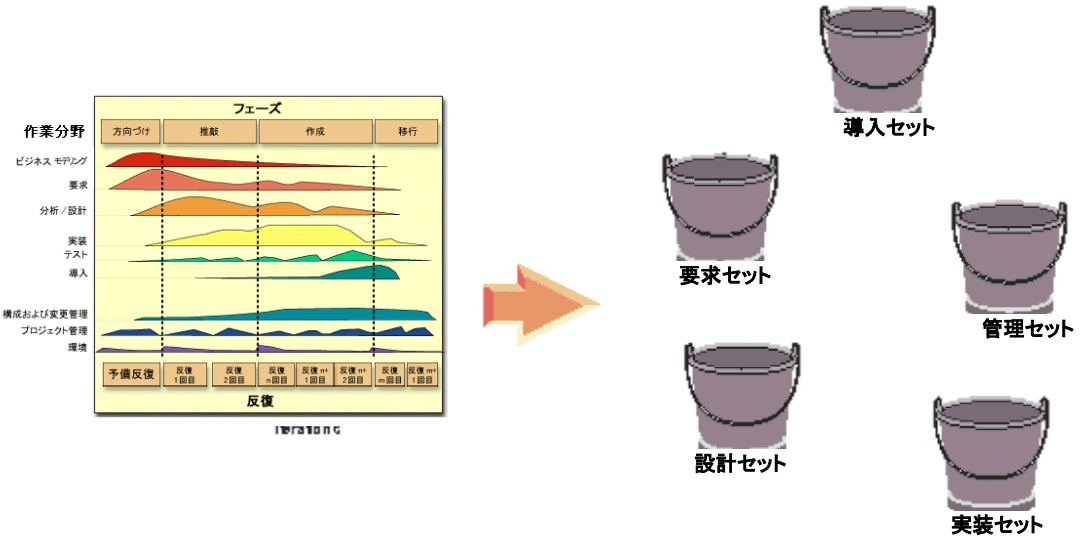

図 14: システムのライフサイクルは情報セットを生成します。

相互接続されたシステムで構成されるシステムでは、上位システムと各下位システムは、独自の情報セットの集合を作成します。図 15 を参照してください。

- 下位の情報セットは、対応する上位情報セットに対して依存関係があります。
- アプリケーションのタイプが異なる場合があるので、コンテンツ タイプの対応する情報セットが下位システム間で異なる場合があります。
- 対応する下位情報セットは独立している必要があります。ただし、上位システムで定義されている同じサブシステム インターフェイスを持つ点は除きます。

少なくとも、上位システムと下位システムの成果物間の追跡可能性を維持する作業を、継続して行わなければなりません。 システム内の追跡可能性の維持を優先させる必要があります。

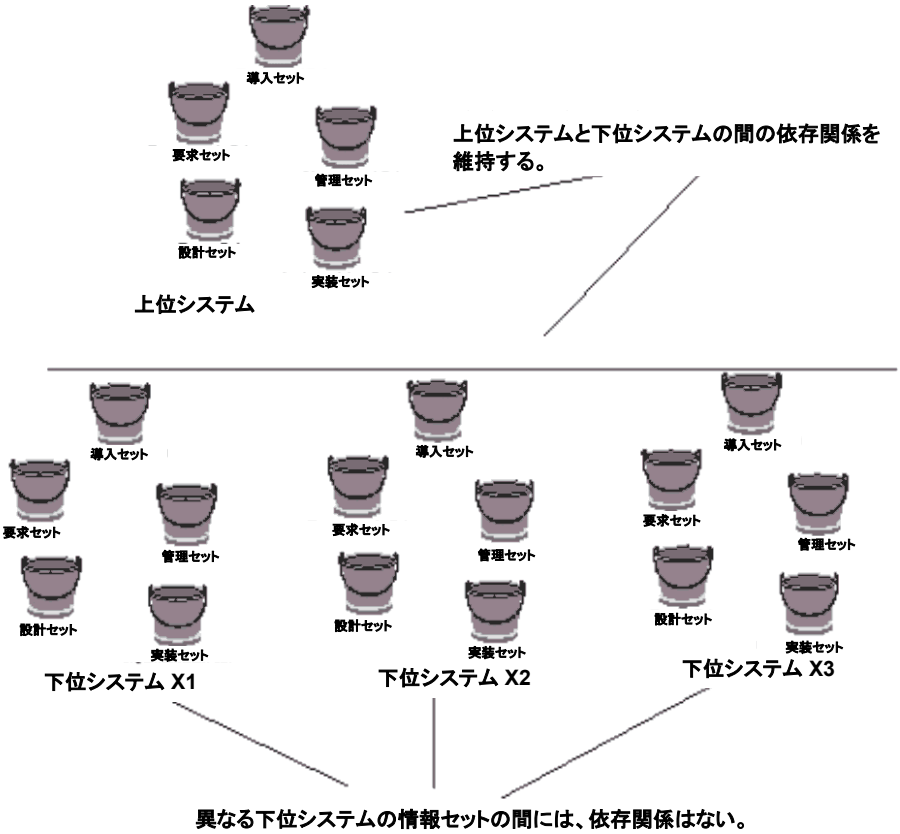

図 15: 相互接続されたシステムで構成されるシステムにおける各システムは、独自の情報セット群を生成します。

相互接続されたシステムで構成されるシステムにおけるアーキテクチャ

相互接続されたシステムで構成されるシステムにおける各システムは、上位と下位のどちらについてもアーキテクチャが定義されている必要があります。

上位システムの場合、アーキテクチャ文書では以下のことが記述されていなければなりません。

- 上位システムに関する主要なユース ケースまたはシナリオ。
- 相互接続されたシステムで構成されるシステムのレイヤリング。
- 下位システム間で再利用を行う方法と再利用の対象。
- 主要なメカニズムとその実装。すべての下位システムで使用するのに十分一般的なもの。 たとえば、すべての下位システムは、通信、エラー通知、障害管理について共通のメカニズムを使用する必要があります。そうでないと、上位システムは同種のシステムとして動作しません。

下位システムの場合、アーキテクチャ文書では以下のことが明記されていなければなりません。

- 相互接続されたシステムで構成されるシステムにおける下位システムの役割。
- 下位システムに関する主要なユース ケースまたはシナリオ。
- 相互接続されたシステムで構成されるシステムに定義されているレイヤ構造を、下位システムが使用する方法。 つまり、上位システムのレイヤ化されたアーキテクチャで、下位システムに定義されている役割を下位システムが果たす方法を定義する必要があります。
- 使用される一般的な主要メカニズムとその使用方法。追加されているアプリケーション固有の主要メカニズム。

- 再利用の適用方法。 特に、複数の下位システム間で共通のサブシステムと、下位システム間の通信を可能にするために構築されるメカニズム。

システム間の関係

相互接続されたシステムで構成されるシステムによって実装されるシステムに対しても、通常のシステム開発作業が適用できることを見てきました。 この利点は、ほかのシステムで使用される場合とまったく異なる方法でこのようなシステムを扱わなくてもよいことです。 また、ほかの下位システムの形式で、上位システムを実装からきれいに分離することもできます。 相互接続されたシステムで構成されるシステムでは、各システムは独自のライフサイクルを持っています。 各システムの特性は異なるので、それぞれに適した異なる開発プロセスを使用してシステムを開発できます。 Rational Unified Process [2] の観点から見ると、システムごとに異なる開発個別定義書を使用します。

以下は、相互接続されたシステムで構成されるシステムに含まれる、システム間の独立性に関する最後の注意です。

まず、下位システムを見てみます。 各下位システムは、上位システムの設計モデルの 1 つのサブシステムを実装します。
サブシステムは相互に明示的に依存するのではなく、相互のインターフェイスに依存します。図 12 を参照してください。
したがって、あるサブシステムを新しいバージョンに交換する場合、インターフェイスが変わらなければ、ほかのサブシステムに影響を与えずに行うことができます。 下位システム間の関係はまったく同じです。 各下位システムは、周囲をインターフェイスの集合として見ます。 そのため、新しいシステムがほかのシステムに対して同じ役割をになう限り、つまり同じインターフェイスの集合を提示できる限り、システムを別のものに置き換えることができます。 上位モデルで対応するサブシステムとインターフェイス間の関係で指定されているように、システムは相互のインターフェイスを参照します。
下位システムのユース ケース モデルでは、相互作用するほかの下位システムのインターフェイスは、アクターとして表されます。 下位システムは、別のシステムのインターフェイスを対応するアクターが提供するものと見なし、したがって、ほかのシステムを直接参照することはありません。図 16 を参照してください。 図 16 ではインターフェイス B が複数の場所に出現していることに注目してください。これは、インターフェイス B が、実際には、上位システムのサブシステムとそれに対応する下位システムによって参照される同じインターフェイスであることを示しています。

相互接続されたシステムで構成されるシステム

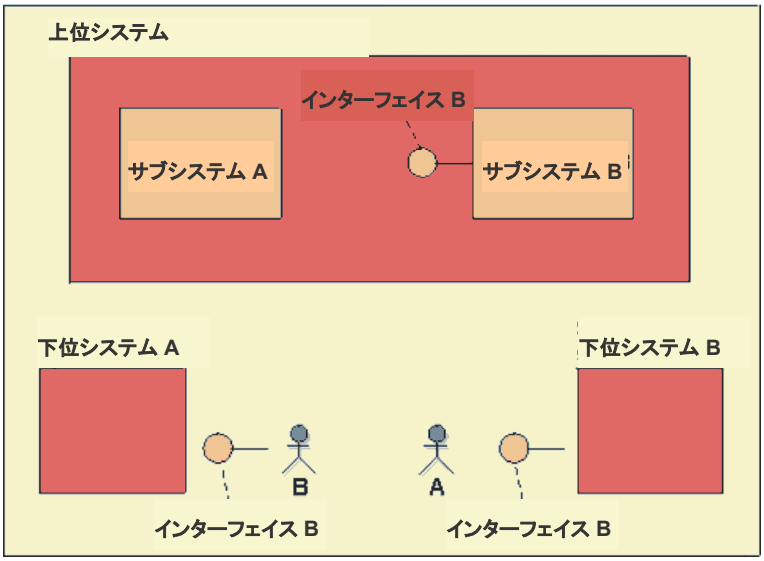

図 16: 上位システムのサブシステムは、インターフェイスを介してのみ相互に依存します。したがって、サブシステムを実装する下位システムにも、同様の独立性があります。上位システムのモデルでは、サブシステム B はほかのサブシステムに対してインターフェイス B を提供します。したがって、対応する下位システム B は、ほかの下位システムに対して同じインターフェイス B を提供する必要があります。

上位システムについてはどうでしょう。上位システムの下位システムに対する関係はどのようなものでしょう。 上位システムは、以下のような意味で実装システムから独立しています。各実装システムは、上位システムのモデルで指定されていることの実装でしかなく、仕様の一部ではありません。 実際的な理由から、要求を追跡するために、異なるレベルでシステム間の追跡可能性リンクを定義する必要があり、そのための最も「簡潔な」方法は、インターフェイス間だけにこのようなリンクを定義することです。図 11 を参照してください。 実際、下位システムは上位モデルで定義されているインターフェイスを提供する実装でしかありません。

しかし、単純な例以上のものであるシステムでは、これでは不十分です。 インターフェイスは、特定の相互作用ポイントで行われること以外は指定しません。 下位システムには何百ものインターフェイスがあり、各インターフェイスには何十もの操作があるかもしれません。 インターフェイスの記述において、あるインターフェイスの入力を別のインターフェイスの 1 つ以上の出力に関連付けるのは、現実的ではありません。 このような理由から、下位システムのセマンティクスを説明するためにユース ケースが必要になります。

結論として、相互接続されたシステムで構成されるシステムによってあるシステムが実装されたときに関係する各システムは、ほかのシステムとは独立していますが、相互のインターフェイスには強く依存しています。 このことは、下位システムを並行開発するために非常にすぐれたプラットフォームになります。

適用分野

相互接続されたシステムで構成されるシステムのアーキテクチャとモデリング技法は、次のような異なるタイプのシステムにも使用できます。

- 分散システム
- 非常に大規模または複雑なシステム
- 複数のビジネス領域を結合するシステム
- ほかのシステムを再利用するシステム
- システムの分散開発

逆の状況も考えられます。つまり、既存のシステム群を組み合わせて、相互接続されるシステムで構成されたシステムを定義することもできます。 実際、大規模なシステムは、発展の初期フェーズではこのような方法で発展する場合があります。 相互接続できるシステムがあり、そのようにして「大規模なシステム」を構築すると、独立したシステムを 2 つ作るよりメリットが大きいことがわかります。

実際、システムの異なる部分を独自のシステムとして見ることのできるシステムは、相互接続されたシステムで構成されるシステムとして定義することをお勧めします。 現在は単一のシステムであっても、分散開発、再利用、一部だけ購入したいという顧客のニーズなどの理由から、後で複数の独立した製品にシステムを分割する必要性が出てくるかもしれません。

最後に、相互接続されたシステムで構成されるシステムのアーキテクチャを使用できる事例をいくつか詳しく見ていくことにします。 それぞれの事例では、問題となっているシステムを単独のシステムと独立したシステムの集合の両方として検討する必要があることを示した上で、相互接続されたシステムで構成されるシステムで実装される上位システムとして扱う必要があることを示します。

大規模なシステム

電話ネットワークは、おそらく、相互接続されたシステムで構成されるシステムとしては世界最大です。 複雑さを管理するために 3 つ以上のシステム レベルが必要になるよい例です。 また、最上位のシステムを標準化団体が所有し、この標準に準拠しなければならない下位システムを、競争関係にある複数の企業が開発している例でもあります。 ここでは、モバイル電話ネットワークの GSM (Global System of Mobile Telephony) について説明して、相互接続されたシステムで構成されるシステムとして大規模システムを実装するメリットを示します。

非常に大規模なシステムの機能では、通常、複数のビジネス領域が結合されています。 たとえば、GSM の標準は、呼び出し元の加入者から呼び出し先の加入者までのシステム全体をカバーしています。 つまり、モバイル電話とネットワーク ノードの両方の動作が含まれています。 システムの異なる部分は、個別に、また場合によっては異なる顧客によって購入される独立した製品であるため、それぞれを独自のシステムとして扱う必要があります。 たとえば、完全な GSM システムを開発する企業は、モバイル電話を加入者に販売し、ネットワーク ノードを電話事業者に販売します。 これは、GSM システムの異なる部分を異なる下位システムとして扱う 1 つの理由になります。

もう 1 つの理由は、GSM のように大規模で複雑なシステムを単一のシステムとして開発しようとすると、あまりにも時間がかかりすぎることです。異なる部分は、複数の開発チームで並行して開発する必要があります。

一方で、GSM 標準はシステム全体を対象とするので、システムを全体として、つまり上位システムとして考えることも必要です。 そうすることで、開発者は、問題領域や、異なる部分が相互に関係する仕組みを理解しやすくなります。

分散システム

複数のコンピュータ システムに分散されるシステムには、相互接続されたシステムで構成されるシステムのアーキテクチャが非常に適しています。 定義上、分散システムは必ず最低 2 つの部分で構成されています。 分散システムには明確に定義されたインターフェイスが必要なので、この種のシステムは、分散方式での**開発**、つまり複数の自立的な開発チームによる並行作業に、非常によく適しています。 分散システムの下位システムは、独立した製品として販売できます。 つまり、分散システムを独立したシステムの集合として考えるのは当然のことです。

分散システムに対する要求は、通常、システム全体の機能を対象にしており、異なる部分間のインターフェイスが事前に定義されていないこともあります。 さらに、開発者にとって経験のない問題領域の場合、開発者はまず、システムが将来どのように分散されるのかには関係なく、システム全体の機能を検討する必要があります。 以上は、分散システムを単一のシステムとして見る、非常に重要な 2 つの理由です。

既存システムの再利用

たいていの場合、大規模なシステムでは既存のシステムを再利用します。 既存システムは下位システムとして記述できます。 そして、既存システムのユース ケース モデルと分析モデルを「リエンジニアリング」して、上位システムの大規模な状況の中で既存システムがどのように動作できるのかを理解します。 リエンジニアリングされたモデルは、完全なものでなくてもかまいません。ただし、最低限、相互接続されたシステムで構成されるシステムのほかの部分の機能に影響を与えたり、変更が必要になる可能性のある既存システムの機能は、カバーしている必要があります。

既製パッケージの使用

複数の既製パッケージを統合してカスタマイズすることで、システムを構築できる場合があります。 ERP (Enterprise Resource Planning) システムはそのよい例です。 多くの ERP システムは、MRP (資材所用量計画)、在庫管理、サプライ チェーン管理などの下位システムを組み合わせて作られています。 人事や給与計算のアプリケーションなどのほかの分野でも、同様の方法を利用できます。 このようなアプリケーションは既製のシステムと同じように、完全なシステムを構築するためには、カスタマイズし、ほかの標準パッケージと相互接続する必要があります。 パッケージ群が連携して何を行うのかを理解するには、上位システムが必要です。 これは、金融業界の多くの顧客が今日直面しているケースです。

まとめ

このホワイト ペーパーでは、相互接続されたシステムで構成されるシステムのアーキテクチャ パターンを紹介しました。 この構造を使用すると、1 つのモデル内だけではない反復が可能になります。この構造では、各サブシステムは独立したシステムとして考えられ、各システムのあらゆる成果物セット間に反復があります。 ここで紹介したアーキテクチャは、相互に通信する複数のシステムによって実装されるシステムで使用されます。 含まれる各システムは、ほかのシステムのモデルとは関係なく、そのシステム固有のモデル群によって記述されます。

この技法を使用する利点は明らかです。つまり、かなり複雑な問題でも扱うことができ、「分割して攻略する」技法を使用して問題を理解できます。 ただし、欠点もあり、オーバーヘッドが増加し、スケジュールの同期が取れなくなるというリスクがあります。 また、上位システムに対して反復的ライフサイクルを採用するのが非常に困難で、そのためにリスクが上位システムのライフサイクルの最後に押しやられてしまう危険が発生する例も見てきました。適切で効果的な再利用戦略を守って、再利用ができないような「ストーブの煙突型」システムが開発されないよう注意することも必要です。

紹介した事例は、相互接続されたシステムで構成されるシステムをモデリングするためのアーキテクチャが、さまざまなアプリケーション分野で役に立つことを示しています。 実際、異なる部分を独立したシステムとして見ることのできるシステムには、紹介したアーキテクチャを適用できます。

参考資料

[1] Jacobson, I.; Palmkvist, K.; and Dyrhage, S., "*Systems of Interconnected Systems*", ROAD, 2(1), 1995.

[2] Rational Unified Process バージョン 5.1

[3] Rumbaugh, J.; Booch, G.; Jacobson,I., "*UML Reference Manual*", Addison Wesley Longman, 1999 (邦訳: 『UML リファレンス マニュアル』日本ラショナルソフトウェア社訳、石塚 圭樹 監訳)

[4] Herbert A. Simon, "*The Sciences of the Artificial*", MIT Press, 1981.

[5] Jacobson, I.; Bylund, S.; Jonsson, P., "*Using Contracts and Use Cases to Build Plugable Architectures*", Journal of Object-Oriented Programming, May/June, 1995.

[6] Jacobson, J.; Griss, M.; Jonsson, P., "*Software Reuse – Architecture, Process and Organization for Business Success*", Addison Wesley Longman, 1997 (邦訳: 『ソフトウェア再利用ガイドブック - アーキテクチャー、プロセス、組織の変革による再利用ビジネス成功への道』杉本宣男／吉田幸彦／落合 修／田中正仁 監訳)

[7] Jacobson, I., "*Use Cases in Large-Scale Systems*", ROAD, 1(6), 1995.

Rational™
the software development company

日本ラショナル ソフトウェア株式会社

103-0007
東京都中央区日本橋浜町 3-42-3
住友不動産浜町ビル 5F
電話: 03-5642-9100
Fax: 03-5642-9120
E-mail: support@japan.rational.com
Web: http://www.rational.co.jp
各国の情報: http://www.rational.com/worldwide/