2

ソフトウェア アーキテクチャの基本原理

概要

この章では、ソフトウェア アーキテクチャの主要な設計原理とガイドラインを紹介します。ソフトウェア アーキテクチャは、システムの構成や構造だとされています。この “システム” とは、特定の機能または一連の機能を実行するコンポーネントのコレクションのことを表します。つまり、アーキテクチャでは、特定の機能をサポートするようにコンポーネントを構成することに重点を置いています。多くの場合、この機能の構成は、コンポーネントを "関連領域" にグループ化することで行います。図 1 に、コンポーネントを関連領域でグループ化した、一般的なアプリケーションのアーキテクチャの例を示します。

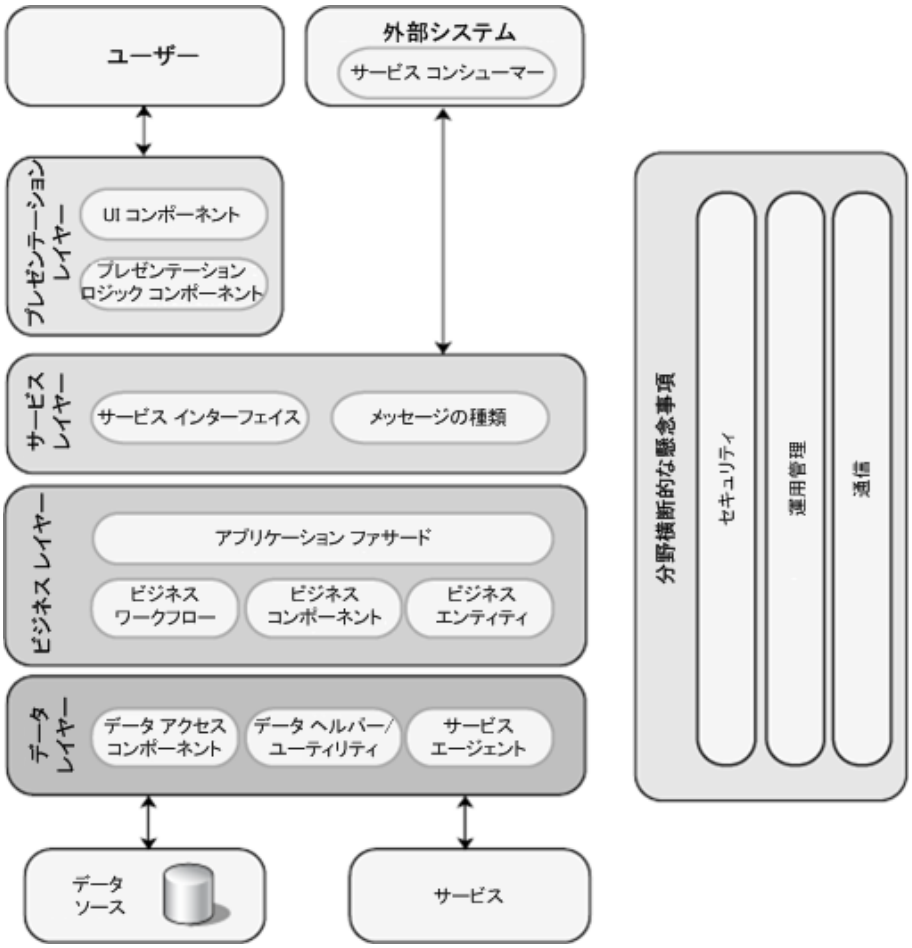

図 1

一般的なアプリケーションのアーキテクチャ

コンポーネントのグループ化に加えて、関連領域では、コンポーネント間の相互作用や異なるコンポーネントがどのように連携するのかについても重点を置いています。この章のガイドラインでは、アプリケーションのアーキテクチャを設計する際に考慮すべき、さまざまな関連領域を取り上げます。

設計の基本原理

設計を開始する際には、アーキテクチャの作成に役立つ基本原理に留意してください。これにより、実績のある原理に準じ、コストと保守の必要性を最小限に抑え、ユーザビリティと拡張性を実現するアーキテクチャを作成できます。基本原理は次のとおりです。

- **関心の分離**: アプリケーションを機能で分割し、機能の重複を可能な限り排除します。相互作用する部分を最小限に抑え、統合性を高めて結合性を抑えるのが重要です。ただし、不適切な境界で機能を分離すると、機能の内容に大きな重複がない場合でも、機能間で高い結合性と複雑さが生じます。

- **単一の役割の原理**: 各コンポーネントとモジュールで管理するのは、特定の機能や機能性、または統合された機能の集合体のみにとどめる必要があります。
- **重複排除の原則 (DRY)**: 機能を提供する場所は 1 か所に絞る必要があります。たとえば、アプリケーションの設計では、特定の機能は単一のコンポーネントに実装する必要があり、他のコンポーネントに実装する機能と重複しないようにします。
- **事前設計の最小化**: 必要な設計のみを行います。開発コストや設計段階のミスによるコストが非常に高ければ、事前に包括的な設計とテストが必要になる場合があります。それ以外の場合は (特にアジャイル開発では)、事前の大規模設計 (BDUF) を回避できます。アプリケーションの要件が明確でなかったり、設計が時間の経過と共に発展する可能性がある場合、早い段階で大規模設計に取り組まないようにします。この原理は、YAGNI (“今必要なことだけを行う”) と呼ばれることもあります。

アプリケーションやシステムを設計する際、ソフトウェア アーキテクトの使命は、設計を関連領域に分離して、アプリケーションやシステムが、なるべく複雑にならないようにすることです。たとえば、ユーザー インターフェイス (UI)、業務プロセス、およびデータ アクセスはすべて、異なる関連領域に存在します。関連領域に分離して設計したコンポーネントでは、その領域に的を絞って、他の関連領域のコードと混在しないようにする必要があります。たとえば、UI 処理コンポーネントには、データ ソースに直接アクセスするコードは含めず、データを取得するためには、ビジネス コンポーネントやデータ アクセス コンポーネントのいずれかを使用する必要があります。

ただし、アプリケーションへの投資については、コストや価値の面から判断する必要もあります。たとえば、UI データを結果セットにバインドできるように、構造を簡略化しなければならない場合があります。通常、機能の境界はビジネスの面からも考慮します。次のガイドラインの概要は、アプリケーションの設計、実装、配置、テスト、および管理の簡略化に影響を及ぼす可能性がある、さまざまな要因を検討する際に役立ちます。

設計に関する慣例

- **各レイヤーで設計パターンの一貫性を維持する**: 可能であれば、論理レイヤーでは、特定の操作に関するコンポーネントの設計で一貫性を維持する必要があります。たとえば、データベースのテーブルやビューへのゲートウェイとして機能するオブジェクトを作成するために、Table Data Gateway パターンの使用する場合は、Repository などの別のパターンを含めないようにします。というのも、Repository パターンでは、データにアクセスしたり、ビジネス エンティティを初期化したりするために異なるパラダイムを使用するからです。ただし、ビジネス トランザクションと

レポートの作成機能を実行するアプリケーションなど、要件に大きなばらつきがあるレイヤーのタスクには、複数の異なるパターンを使用しなければならない場合があります。

- **アプリケーションの機能は重複しないようにする**: 特定の機能は単一のコンポーネントで提供する必要があります。同じ機能は他のコンポーネントで重複して提供しないようにします。このような構成により、コンポーネントを統合し、特定の機能や機能性が変更されても、簡単にコンポーネントを最適化できます。アプリケーションの機能に重複があると、変更を実装するのが困難になり、明瞭さが失われ、不整合が生じる可能性があります。
- **継承ではなく複合 (コンポジション) を使用する**: 機能を再利用する際には、可能な限り、継承ではなく複合を使用します。これは、継承を使用すると親クラスと子クラスの間の依存関係が強くなり、子クラスの再利用が制限されるためです。また、クラスを再利用すると継承階層も崩れるので、対応が非常に困難になります。
- **開発のコーディング スタイルと名前付け規則を確立する**: 組織でコーディング スタイルと名前付けの標準を確立しているかどうかを確認します。確立していない場合は、一般的な標準を確立する必要があります。標準を確立すると、一貫性のあるモデルが提供され、チームのメンバーは他のメンバーが記述したコードのレビューも容易に行えるようになるので、保守容易性の向上につながります。
- **開発時には自動化された QA (品質分析) の技法を使用して、システムの品質を維持する**: 開発中には、単体テストやその他の自動化された品質分析技法 (依存関係分析、スタティック コード分析など) を使用します。コンポーネントとサブシステムについては、明確な動作とパフォーマンス メトリックを定義し、開発時には自動 QA ツールを使用して、ローカルの設計や実装に関する判断がシステムの品質全体に悪影響を及ぼさないことを確認します。
- **アプリケーションの運用を考慮する**: IT インフラストラクチャで必要なメトリックと運用データを特定し、アプリケーションの配置と運用を確認します。アプリケーションのコンポーネントとサブシステムの個々の運用上の要件を明確に理解したうえで、これらを設計すると、全体的な配置と運用が大幅に簡略化されます。開発時には自動 QA ツールを使用して、アプリケーションのコンポーネントとサブシステムで適切な運用データが提供されていることを確認します。

アプリケーション レイヤー

- **関心事で分離する**: アプリケーションを機能で分割し、機能の重複を可能な限り排除します。この手法を使用する一番のメリットは、機能または機能性を、その他の機能や機能性から独立した状態で最適化できることです。また、ある機能でエラーが発生しても、他の機能でエラーが発生する原因にはならず、相互に独立した状態で実行できます。さらに、この手法を使用すると、より簡単に

アプリケーションを理解して設計することが可能で、相互に依存関係がある複雑なシステムの管理が容易になります。

- **レイヤー間の通信方法を明確にする**: アプリケーションのすべてのレイヤーが、相互に通信したり、他のすべてのレイヤーと依存関係を持てるようにすると、そのソリューションを理解して管理するのが困難になります。レイヤー間の依存関係とデータ フローについては明確な判断を下す必要があります。
- **抽象化を使用してレイヤー間の疎結合を実装する**: これは、インターフェイス コンポーネントを定義することで実現できます。たとえば、レイヤーのコンポーネントで認識できる形式に要求を変換する一般的な入出力の機能を持つファサードを定義できます。また、インターフェイス型や抽象型の基本クラスを使用して、インターフェイス コンポーネントで実装する必要がある共通のインターフェイスや抽象化 (依存関係の反転) を定義することもできます。
- **同じ論理レイヤーに種類の異なるコンポーネントを混在させない**: 関連領域を特定してから、各関連領域と関係のあるコンポーネントを論理レイヤーにグループ化します。たとえば、UI レイヤーには、ビジネス プロセスのコンポーネントを含めない代わりに、ユーザー入力やユーザー要求の処理に使用するコンポーネントを必ず含めるようにします。
- **レイヤーやコンポーネント内でデータ形式の一貫性を維持する**: データ形式が混在していると、アプリケーションの実装、拡張、および管理が困難になります。ただし、データの形式を変換する際には、変換コードを実装して、処理のオーバーヘッドを負う必要があります。

コンポーネント、モジュール、および機能

- **コンポーネントやオブジェクトは、他のコンポーネントやオブジェクト内部の詳細に依存しないようにする**: 各コンポーネントやオブジェクトでは、他のオブジェクトやコンポーネントのメソッドを呼び出す必要があります。また、このメソッドでは、要求の処理方法 (状況によっては、要求を適切なサブコンポーネントや他のコンポーネントにルーティングする方法) に関する情報を保持している必要があります。これにより、保守容易性と柔軟性の高いアプリケーションを開発できます。
- **コンポーネントに過剰な機能を持たせないようにする**: たとえば、UI 処理コンポーネントには、データ アクセス コードを含めたり、追加の機能を提供したりしないようにします。多くの場合、過剰な機能を提供するコンポーネントには、ログ記録、例外処理など、横断的な機能性が混在したビジネス機能を提供する多数の機能やプロパティが含まれています。このような設計では、エラーが発生しやすく、管理が難しくなります。単一の役割と関心の分離の原理を適用することで、このような問題を回避できます。

- **コンポーネント間の通信方法を理解する**: これには、アプリケーションでサポートしなければならない配置シナリオを理解する必要があります。すべてのコンポーネントが同じプロセスで実行されるかどうかと、(メッセージ ベースのインターフェイスを実装することによって) 物理境界やプロセス境界で通信がサポートされる必要があるかどうかを確認する必要があります。
- **アプリケーションのビジネス ロジックから抽象化された横断的なコードを可能な限り分離する**: 横断的なコードとは、セキュリティ、通信、または運用管理 (ログ記録、インストルメンテーションなど) に関連するコードのことです。これらの機能を実装したコードをビジネス ロジックと混在させると、拡張や管理が困難な設計となる可能性があります。分野横断的なコードを変更するには、分野横断的なコードに混在するすべてのビジネス ロジック コードに対応する必要があります。分野横断的な懸念事項の管理を可能にするフレームワークと技法 (アスペクト指向プログラミングなど) の使用を検討してください。
- **コンポーネントの明確なコントラクトを定義する**: コンポーネント、モジュール、および機能では、その使用方法と動作を明確に示すコントラクトやインターフェイスの仕様を定義する必要があります。コントラクトでは、コンポーネント、モジュール、または機能に内在する機能性に他のコンポーネントがどのようにアクセスできるのか、前提条件、後条件、副作用、例外、パフォーマンス特性などの要因から見た、この機能性の動作を規定する必要があります。

設計に関する主要な考慮事項

このガイドでは、決定すべき重要事項について説明します。これを理解することにより、アーキテクチャの設計に着手して、繰り返し開発しながら、すべての重要な要因を検討できます。これ以降のセクションで概要を説明する主要な決定事項は次のとおりです。

- アプリケーションの種類の決定
- 配置に関する方針の決定
- 適切なテクノロジの決定
- 品質特性の決定
- 横断的関心事の決定

設計プロセスの詳細については、第 4 章「アーキテクチャと設計の手法」を参照してください。

アプリケーションの種類の決定

適切なアプリケーションの種類を選択することは、アプリケーションを設計するプロセスにおいて重要です。この選択は、固有の要件とインフラストラクチャの制限事項に従って行います。多くのアプリケーションでは、複数の種類のクライアントをサポートする必要があり、複数の基本的な規範を使用する場合があります。このガイドでは、次の基本的なアプリケーションの種類について説明します。

- モバイル デバイス用に設計されたアプリケーション
- 主にクライアント PC で実行されるように設計されたリッチ クライアント アプリケーション (RIA)
- リッチ UI とメディア シナリオをサポートするインターネットから配置されるように設計されたリッチ インターネット アプリケーション (RIA)
- 疎結合コンポーネント間の通信をサポートするように設計されたサービス アプリケーション
- 常時接続シナリオにおいて、主にサーバーで実行されるように設計された Web アプリケーション

より専門的な種類のアプリケーションについての情報とガイドラインも提供します。これには、次のようなものが含まれます。

- ホストされているクラウド ベースのアプリケーションとサービス
- Microsoft Office とマイクロソフト サーバー テクノロジを統合する Office Business Application (OBA)
- ポータル形式でビジネスに必要な情報と機能を提供する SharePoint 基幹業務 (LOB) アプリケーション

アプリケーションの規範の詳細については、第 20 章「アプリケーションの種類の選択」を参照してください。

配置に関する方針の決定

アプリケーションは、さまざまな環境に配置されます。配置先の環境では、コンポーネントが物理的に異なるサーバーに分散していたり、ネットワーク プロトコル、ファイアウォール、ルーターの構成などに制限事項があったりします。一般的な配置パターンがいくつか存在しますが、各パターンでは、さまざまな分散シナリオと非分散シナリオに関するメリットおよび考慮事項を示しています。アプリケーションの要件と、ハードウェアでサポートできる適切なパターンや選択した配置オプションによって環境から課される制約とのバランスを取る必要があります。これらの要素は、アーキテクチャの設計に影響を及ぼします。

配置に関する問題の詳細については、第 19 章「物理層と配置」を参照してください。

適切なテクノロジの決定

アプリケーションで使用するテクノロジを選択する際に検討すべき重要な要素は、開発するアプリケーションの種類と、アプリケーションの配置トポロジおよびアーキテクチャのスタイルに採用するのが望ましいオプションです。また、テクノロジは、組織のポリシー、インフラストラクチャの制限事項、リソースのスキルなどに従って選択します。選択するテクノロジを決定する前には、このすべての要素を考慮して、選択したテクノロジの機能をアプリケーションの要件と比較する必要があります。

マイクロソフト プラットフォームで使用できるテクノロジの詳細については、付録 A「マイクロソフト アプリケーション プラットフォーム」を参照してください。

品質特性の決定

品質特性 (セキュリティ、パフォーマンス、ユーザビリティなど) を使用して、設計に関して解決すべき重要な問題を検討することができます。要件によって、このガイドで取り上げるすべての品質特性について考慮しなければならない場合もあれば、一部だけを考慮すればよい場合もあります。たとえば、アプリケーションの設計では、セキュリティとパフォーマンスを必ず考慮する必要がありますが、相互運用性やスケーラビリティに関してはすべての設計で考慮しなくてもかまいません。まず、要件と配置シナリオを理解すると、設計において重要な品質特性がわかります。品質特性では競合が発生する場合があることに注意してください。たとえば、セキュリティを重視すると、多くの場合、パフォーマンスやユーザビリティが低下します。

品質特性を考慮した設計を行う場合、次のガイドラインを検討してください。

- 品質特性は、システムの機能から切り離したシステムの特性です。
- 技術的な観点では、品質特性を実装するかどうかが、良いシステムと悪いシステムの分岐点となります。
- 品質特性には、実行時に測定されるものと検査でのみ推定可能なものの 2 種類があります。
- 品質特性間のトレードオフを分析します。

品質特性を検討する際には、次の事項を検討する必要があります。

- アプリケーションに必要な重要な品質特性は何ですか。これを設計プロセスの一環として特定します。
- 特定した品質特性に対応する際の重要な要件は何ですか。また、それは定量化できますか。
- 要件が満たされたことを確認する基準は何ですか。

品質特性の詳細については、第 16 章「品質特性」を参照してください。

横断的関心事の決定

横断的関心事は、設計の主要な領域で、アプリケーションの特定のレイヤーとは関連がありません。たとえば、次の事項に関しては、一元管理されるソリューションや共通のソリューションの実装を検討する必要があります。

- 各レイヤーで共通のストアにログを記録したり、後で結果を関連付けることができるように個別のストアにログを記録できるようにするログ記録のメカニズム
- 複数のレイヤー間で ID を渡して、リソースへのアクセスを許可する認証と承認のメカニズム
- 各レイヤー内で機能したり、システムの境界に例外が伝達されたときにはレイヤーをまたいで機能したりする例外管理のフレームワーク
- レイヤー間の通信で使用できる通信の手法
- プレゼンテーション レイヤー、ビジネス レイヤー、データ アクセス レイヤーでデータをキャッシュできる共通のキャッシュ インフラストラクチャ

次の一覧に、アプリケーションを設計する際に検討する必要がある主要な横断的関心事の一部を示します。

- **インストルメンテーションとログ記録**: すべてのビジネス クリティカルなイベントとシステム クリティカルなイベントをインストルメント化します。また、詳細情報をログに記録して、機密情報を除いた情報を使用して、システムでイベントを再作成します。
- **認証**: ユーザーを認証したり、レイヤー間で認証済み ID を渡す方法を決定します。
- **承認**: 各レイヤーと信頼境界内において、適切な粒度で承認が正常に行われるようにします。
- **例外管理**: 機能上、論理上、物理上の境界で例外をキャッチして、機密情報がエンド ユーザーに開示されないようにします。
- **通信**: 適切なプロトコルを選択し、ネットワーク経由の呼び出しを最小限に抑えて、ネットワーク上で渡される機密データを保護します。
- **キャッシュ**: 何をどこにキャッシュする必要があるのかを特定し、アプリケーションのパフォーマンスと応答性を向上します。キャッシュの設計時には Web ファームとアプリケーション ファームの問題を検討します。

横断的関心事の詳細については、第 17 章「横断的関心事」を参照してください。