5

レイヤー型アプリケーションのガイドライン

概要

この章では、アプリケーションの全体的な構造を、相互に通信したり、他のクライアントやアプリケーションと通信したりする個別のレイヤーにコンポーネントを論理的にグループ化することによって説明します。レイヤーは、コンポーネントや機能の論理的な区分に関心があり、コンポーネントの物理的な場所については考慮しません。レイヤーは異なる層に配置できますが、同じ層に配置される場合もあります。この章では、アプリケーションを別個の論理的な区分に分割する方法、アプリケーションに適した機能レイアウトを選択する方法、およびアプリケーションで複数の種類のクライアントをサポートする方法について紹介します。また、レイヤーでロジックを公開するために使用できるサービスについても概説します。

> レイヤーと層 (“ティア”) の違いを理解するのは重要です。"レイヤー" はアプリケーションの機能とコンポーネントを論理的にグループ化したものです。一方、“層” は個別のサーバー、コンピューター、ネットワーク、または遠隔地に機能とコンポーネントを物理的に分散したものです。レイヤーと層には同じ一連の名前 (プレゼンテーション、ビジネス、サービス、およびデータ) が使用されますが、物理的な分離を意味するのは層 (“ティア”) だけであることを覚えておいてください。一般的には、1 台の物理コンピューター (同じ層) に複数のレイヤーを配置します。層という用語は、2 層、3 層、n 層など物理的な分散パターンを示すと考えることができます。物理層と配置に関する詳細については、第 19 章「物理層と配置」を参照してください。

論理レイヤー型の設計

設計は、論理的にグループ化したソフトウェア コンポーネントに分割できます。これは設計しているアプリケーションの種類にかかわらず、ユーザー インターフェイスあっても、サービスを公開することのみを目的とし

たサービス アプリケーション (アプリケーションのサービス レイヤーとは異なります) であっても関係ありません。このように論理的にグループ化したものをレイヤーと呼びます。レイヤーは、コンポーネントで実行されるさまざまな種類のタスクを区別したり、コンポーネントの再利用性をサポートする設計を簡単に作成したりするのに役立ちます。各論理レイヤーは、サブレイヤーでグループ化され、独立した多数の種類のコンポーネントで構成されており、各サブレイヤーでは固有のタスクを実行しています。

多くのソリューションに存在する汎用的な種類のコンポーネントを特定することで、アプリケーションやサービスのマップを作成して、このマップを設計の青写真として使用できます。アプリケーションを別個の役割や機能を持つレイヤーに分割することで、コードの保守容易性を最大限に高め、さまざまな方法で配置した場合にアプリケーションの動作を最適化したり、テクノロジや設計を決定する必要がある場所を明確に線引きします。

プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤー

最も大まかで抽象的なレベルで見ると、システムの論理アーキテクチャ ビューは、レイヤーでグループ化した一連の連携するコンポーネントだと考えられます。図 1 は、これらのレイヤーや、ユーザー、他のアプリケーション (アプリケーションのビジネス レイヤーに実装されているサービスを呼び出します)、データ ソース (データへのアクセスを提供するリレーショナル データベース、Web サービスなど)、および外部サービスやリモート サービス (アプリケーションで使用されます) との関係を簡略化して大まかに表現したものです。

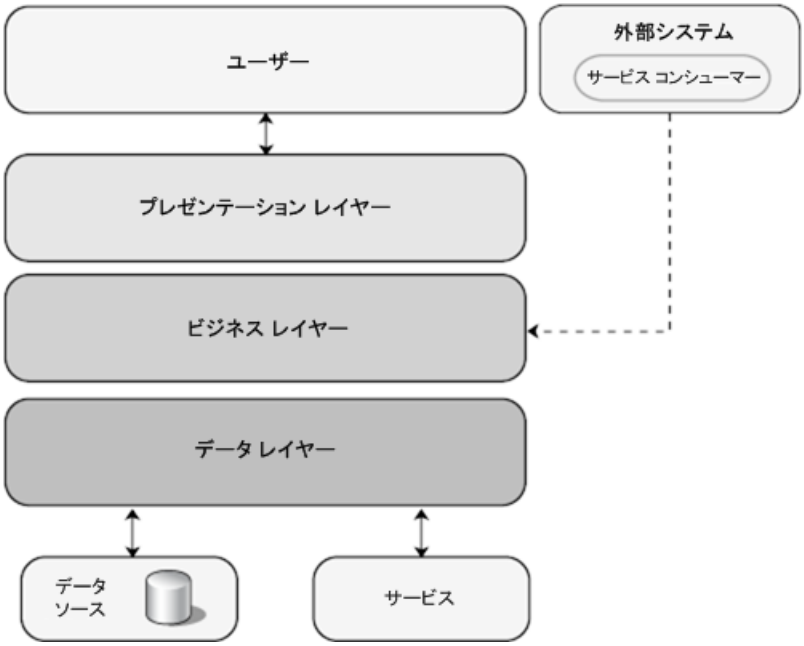

図 1 レイヤー型システムの論理アーキテクチャ ビュー

これらのレイヤーは同じ物理層に配置される場合と別個の物理層に配置される場合があります。レイヤーが別個の層に配置されたり、物理的な境界で分離されている場合、設計では、このビューを考慮する必要があります。詳細については、この章の後半の「レイヤー型構造の設計手順」を参照してください。

図 1 のように、アプリケーションは多数の基本的なレイヤーで構成されています。この一般的な 3 つのレイヤーで構成される設計には、次のレイヤーが含まれます。

- **プレゼンテーション レイヤー**: このレイヤーには、ユーザーとシステムのやりとりを管理するユーザー指向の機能が含まれています。また、通常、ビジネス レイヤーにカプセル化される主要なビジネス ロジックへの共通の橋渡しの役割をするコンポーネントで構成されています。プレゼンテーション レイヤーの設計に関する詳細については、第 6 章「プレゼンテーション レイヤーのガイドライン」を参照してください。プレゼンテーション レイヤーのコンポーネントの設計に関する詳細については、第 11 章「プレゼンテーション レイヤーのコンポーネントの設計」を参照してください。
- **ビジネス レイヤー**: このレイヤーでは、システムの主要機能を実装し、関連するビジネス ロジックをカプセル化します。通常、複数のコンポーネントで構成されており、その一部では他の呼び出し元で使用できるサービス インターフェイスを公開している場合があります。ビジネス レイヤーの設計に関する詳細については、第 7 章「ビジネス レイヤーのガイドライン」を参照してください。ビジネス レイヤーのコンポーネントの設計に関する詳細については、第 12 章「ビジネス レイヤーのコンポーネントの設計」を参照してください。
- **データ レイヤー**: このレイヤーでは、システムの境界内でホストされているデータ、サービスを通じてアクセスできるネットワークに接続している他のシステムで公開されているデータへのアクセスを提供します。データ レイヤーでは、ビジネス レイヤーのコンポーネントで使用できるジェネリック インターフェイスを公開しています。データ レイヤーの設計に関する詳細については、第 8 章「データ レイヤーのガイドライン」を参照してください。データ レイヤー コンポーネントの設計に関する詳細については、第 15 章「データ レイヤー コンポーネントの設計」を参照してください。

サービスとレイヤー

大まかな観点から見ると、サービス ベースのソリューションは複数のサービスで構成され、各サービスはメッセージを渡すことで他のサービスと通信していると見なすことができます。概念的な観点では、サービスはソリューションのコンポーネントと見なすことができます。ただし、各サービスは、内部的には他のアプリケーションと同様にソフトウェア コンポーネントで構成されています。また、このコンポーネントは、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ レイヤーに論理的にグループ化できます。他のアプリケーシ

ョンでは、サービスがどのように実装されているかを知らなくても、サービスを使用することができます。前のセクション「ソフトウェアのアーキテクチャと設計」で説明したレイヤー型の設計の原理は、サービス ベースのソリューションにも同様に適応されます。

サービス レイヤー

アプリケーションで、他のアプリケーションにサービスを提供したり、クライアントを直接サポートする機能を実装する必要がある場合、図 2 のようにアプリケーションのビジネス機能を公開するサービス レイヤーを使用するのが一般的な手法です。サービス レイヤーでは、クライアントが異なるチャネルを使用してアプリケーションにアクセスできるよう、効率的に別のビューを提供します。

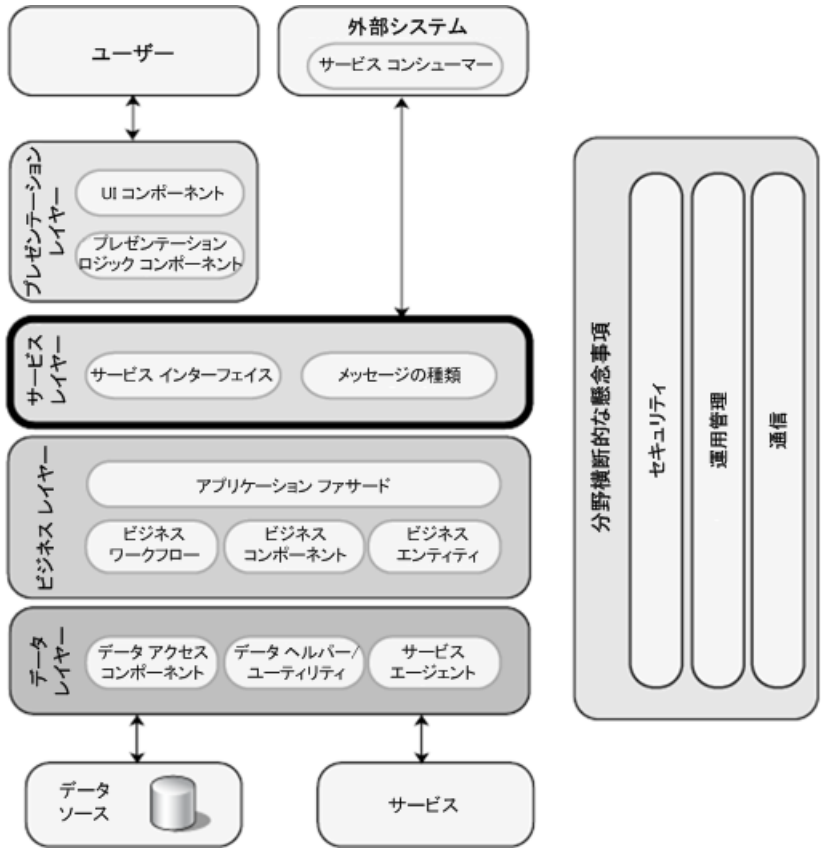

図 2　アプリケーションに含まれるサービス レイヤー

このシナリオでは、ユーザーはプレゼンテーション レイヤー経由でアプリケーションにアクセスできます。プレゼンテーション レイヤーでは、ビジネス レイヤーのコンポーネントと直接通信したり、通信方式で機能を構成する必要がある場合はビジネス レイヤーのアプリケーション ファサード経由で通信します。一方、外部クライアントと他のシステムでは、サービス インターフェイス経由でビジネス レイヤーと通信して、アプリケーションにアクセスして、その機能を使用できます。この構成により、アプリケーションでは、複数の種類のクライ

アントに対してより優れたサポートを提供することができ、アプリケーション間で機能の再利用と高度な構成が促進されます。

プレゼンテーション レイヤーでは、サービス レイヤーを経由してビジネス レイヤーと通信することがあります。ただし、これは絶対要件ではありません。アプリケーションを物理的に配置したときに、プレゼンテーション レイヤーとビジネス レイヤーが同じ層に配置されていると、2 つのレイヤーは直接通信できる場合があります。サービス レイヤーの設計に関する詳細については、第 9 章「サービス レイヤーのガイドライン」を参照してください。レイヤー間の通信に関する詳細については、第 18 章「通信とメッセージ」を参照してください。

レイヤー型構造の設計手順

アプリケーションの設計に着手する際には、まず、ごく大まかなレベルの抽象化に重点を置いて、機能をレイヤーでグループ化します。次に、設計するアプリケーションの種類に基づいて、各レイヤーにパブリック インターフェイスを定義する必要があります。レイヤーとインターフェイスを定義したら、アプリケーションの配置方法を決定する必要があります。最後に、アプリケーションのレイヤーと層の間の通信で使用する通信プロトコルを選択します。構造とインターフェイスは時間の経過と共に変化する可能性がありますが、特にアジャイル開発を使用している場合は、この手順に従うことで、プロセスの開始段階で重要な側面を考慮できます。一般的な設計手順は次のとおりです。

- 手順 1 - レイヤー型に関する方針を選択する
- 手順 2 - 必要なレイヤーを決定する
- 手順 3 - レイヤーとコンポーネントの分散方法を決定する
- 手順 4 - レイヤーをまとめる必要があるかを判断する
- 手順 5 - レイヤー間の通信規則を決定する
- 手順 6 - 横断的関心事を特定する
- 手順 7 - レイヤー間のインターフェイスを定義する
- 手順 8 - 配置に関する方針を選択する
- 手順 9 - 通信プロトコルを選択する

手順 1 - レイヤー型に関する方針を選択する

レイヤー型とは、アプリケーションのコンポーネントを別個の役割や機能を持つグループに論理的に分離したものです。レイヤー型の手法を使用すると、アプリケーションの保守容易性を向上したり、必要に応じてアプリケーションを簡単に拡張してパフォーマンスを向上することができます。関連のある機能をレイヤーでグループ化する方法は数多くあります。ただし、アプリケーションを極端に少ないレイヤーまたは多いレイヤーに分割すると不要に複雑さが増し、全体的なパフォーマンス、保守容易性、および柔軟性が低下する可能性があります。アプリケーションに適したレイヤーの粒度を決定することは、レイヤー型に関する方針を決定するうえで重要な最初の手順です。

また、レイヤー型を実装する目的が、単に機能を論理的に分離するためなのか、または可能であれば物理的に分離するためなのかを検討する必要があります。レイヤーの境界を越えると、ローカルのパフォーマンスでオーバーヘッドが生じます。物理的に離れているコンポーネントの境界を越える場合は特にそうです。ただし、アプリケーションのスケーラビリティと柔軟性を全体的に向上することで、このパフォーマンスのオーバーヘッドを大幅に軽減できます。また、レイヤー型を使用することで、隣接したレイヤーに影響を及ぼすことなく、各レイヤーのパフォーマンスを容易に最適化できます。

論理レイヤー型では、通信するアプリケーション レイヤーは、同じ層に配置され同じプロセス内で動作します。これにより、コンポーネントのインターフェイス経由の直接的な呼び出しなど、より高度なパフォーマンスの通信メカニズムを使用することができます。ただし、今後も論理レイヤー型のメリットを維持して柔軟性を確保するためには、カプセル化とレイヤー間の疎結合を保つ必要があることに注意する必要があります。

別個の層 (異なる物理コンピューター) に配置されたレイヤーでは、接続しているネットワーク経由で隣接したレイヤーと通信します。そのため、選択した設計では、通信の待ち時間を考慮し、レイヤー間の疎結合を保つ適切な通信メカニズムをサポートする必要があります。

また、別個の層に配置する可能性が高いアプリケーション レイヤーと、同じ層に配置する可能性が高いアプリケーション レイヤーを決定するのも、レイヤー型に関する方針において重要な部分です。柔軟性を維持するには、レイヤー間の通信は、必ず疎結合にします。これにより、レイヤーが同じ層に配置されている場合はより高度なパフォーマンスを実現して、必要に応じてレイヤーを複数の層に配置することができます。

レイヤー型の手法を使用すると、複雑さが増し、初期開発時間が長くなる可能性がありますが、適切に実装すると、アプリケーションの保守容易性、拡張性、および柔軟性が大幅に向上します。ただし、再利用性と疎結合のトレードオフを考慮する必要があります。これは、パフォーマンスや複雑さに変化があった場合に生じるものです。アプリケーションで採用するレイヤー型とレイヤー間の通信方法を慎重に検討すると、パフォーマンスと柔軟性の間で適切なバランスをとることができます。通常、レイヤーを使用しない緊密に結合している設計から得られる可能性があるわずかなパフォーマンスの向上は、レイヤー型の設計で得られる柔軟性と保守容易性の向上と比較すると微々たるものです。

一般的なレイヤーの種類の詳細と必要なレイヤーの決定に関するガイダンスについては、この章の前半のセクション「論理レイヤー型の設計」を参照してください。

手順 2 - 必要なレイヤーを決定する

関連のある機能をレイヤーでグループ化する方法は数多くあります。ビジネス アプリケーションにおける最も一般的な手法は、プレゼンテーション、サービス、ビジネス、およびデータ アクセスの機能を別個のレイヤーに分離することです。また、アプリケーションによっては、レポート レイヤー、管理レイヤー、またはインフラストラクチャ レイヤーを導入しているものもあります。

レイヤーの追加は慎重に行います。レイヤーでアプリケーションの保守容易性、スケーラビリティ、または柔軟性を明らかに向上できるような関連コンポーネントを論理的にグループ化しない場合、そのレイヤーは追加しないようにします。たとえば、アプリケーションでサービスを公開しない場合は、別個のサービス レイヤーは必要なく、プレゼンテーション レイヤー、ビジネス レイヤー、およびデータ アクセス レイヤーのみを配置できます。

手順 3 - レイヤーおよびコンポーネントの分散方法を決定する

異なる複数の物理層にレイヤーとコンポーネントを配置するのは、必要な場合にとどめる必要があります。分散配置を実装する一般的な理由は、セキュリティ ポリシー、物理的な制約、共有のビジネス ロジック、およびスケーラビリティです。

- Web アプリケーションでは、プレゼンテーション レイヤーの複数のコンポーネントが同時にビジネス レイヤーのコンポーネントにアクセスする場合、ビジネス レイヤーとプレゼンテーション レイヤーのコンポーネントを同じ物理層に配置することを検討します。セキュリティ上の制限で、この 2 つのコンポーネント間に信頼境界が必要でない限り、この配置により、パフォーマンスが最大限に高まり、運用管理が容易になります。
- デスクトップ上で UI プロセスを使用するリッチ クライアント アプリケーションでは、セキュリティ上の理由と運用管理を容易にするため、ビジネス レイヤーのコンポーネントを物理的に個別のビジネス層に配置する場合があります。
- ビジネス エンティティは、それを使用するコードと同じの物理層に配置します。つまり、ビジネス エンティティは複数の場所に配置する場合があります。たとえば、ロジックでビジネス エンティティを使用したり参照する物理的に別のプレゼンテーション層やデータ層にコピーを配置することがあります。サービス エージェント コンポーネントは、セキュリティ制限でコンポーネント間に信頼境界が必要でない限り、コンポーネントを呼び出すコードと同じ層に配置します。

- 類似した読み込みや I/O の特性を持つ非同期ビジネス コンポーネント、ワークフロー コンポーネント、およびサービスは、個別の物理層に配置することを検討します。このように配置すると、インフラストラクチャを微調整してパフォーマンスとスケーラビリティを最大限に高められます。

手順 4 - レイヤーをまとめる必要があるかを判断する

場合によっては、レイヤーをまとめる (緩和する) 方が理にかなっています。たとえば、ごく少数のビジネス ルールが適用されるアプリケーションや主に検証のためにルールを使用するアプリケーションでは、単一のレイヤーにビジネス ロジックとプレゼンテーション ロジックの両方が実装される場合があります。Web サービスからデータを取得して、そのデータを表示するアプリケーションでは、単に Web サービスの情報を直接プレゼンテーション レイヤーに追加して、直接 Web サービスのデータを使用する方が理にかなっています。この場合、データ アクセス レイヤーとプレゼンテーション レイヤーを論理的に組み合わせています。

レイヤーをまとめる方が理にかなっている場合があります。ただし、基本的に機能はレイヤーでグループ化する必要があります。場合によっては、プロキシの機能を提供したり、多くの機能を提供しなくてもカプセル化や疎結合を実現するパススルー レイヤーとして機能するレイヤーもあります。ただし、機能を分離すると、後で設計に含まれる他のレイヤーにほとんど、またはまったく影響を及ぼすことなく機能を拡張できるというメリットがあります。

手順 5 - レイヤー間の通信規則を決定する

レイヤー型に関する方針では、レイヤー間の通信方法についての規則を定義する必要があります。通信規則を定義する主な理由は、依存関係を最小限に抑えて循環参照を削除するためです。たとえば、2 つのレイヤーがもう一方のレイヤーのコンポーネントへの依存関係を持っている場合、循環する依存関係が発生します。このため、次の手法のいずれかを使用して、レイヤー間では一方向の通信のみを許可するという一般的な規則に従うことをお勧めします。

- **トップダウン方式の通信**: 上位レイヤーは下位レイヤーと通信できますが、下位レイヤーは上位レイヤーと通信しないようにします。この規則により、レイヤー間の循環する依存関係を避けられます。イベントを使用すると依存関係なしに、下位レイヤーで発生した変化が上位レイヤーのコンポーネントで認識されるようになります。
- **厳密な通信**: 各レイヤーが通信できるのは、直下のレイヤーのみです。この規則によって、各レイヤーでは直下のレイヤーのみを把握するという懸念事項の分離が強化されます。この規則には、レイヤーのインターフェイスを変更した場合、直上のレイヤーしか影響を受けないというメリットがあります。時間の経過と共に新しい機能を導入して変化するが、その変化による影響を最小限に抑

える必要があるアプリケーションを設計したり、複数の物理層に分散される可能性があるアプリケーションを設計する場合は、この手法の使用を検討します。

- **厳密でない通信**: 上位レイヤーでは、間にあるレイヤーをバイパスして、下位レイヤーと直接通信できます。これによりパフォーマンスが向上しますが、依存関係が拡大します。つまり、下位レイヤーを変更すると複数の上位レイヤーに影響が及びます。複数の物理層に分散しないアプリケーション (スタンドアロンのリッチ クライアント アプリケーションなど) を設計する場合や、複数のレイヤーに影響を及ぼす変更を簡単に管理できる小さなアプリケーションを設計する場合は、この手法の使用を検討します。

手順 6 - 横断的関心事を特定する

レイヤーを定義したら、レイヤーをまたぐ機能を特定する必要があります。このような機能は “横断的関心事” と表されることが多く、ログ記録、キャッシュ、検証、認証、および例外管理が含まれます。アプリケーションにおいて横断的関心事を特定するのは重要なことで、可能な場合は、この懸念事項を管理するコンポーネントは別個に設計します。この手法を使用すると、再利用性と保守容易性が向上します。

横断的関心事のコードは、各レイヤーのコンポーネントのコードと混在させないようにします。コードを分離することで、レイヤーとそのコンポーネントでは、ログ記録、キャッシュ、認証などの操作を実行する必要がある場合に、横断的関心事のコンポーネントを呼び出すだけでよくなります。レイヤーをまたいで機能を使用できる必要があるのと同じように、レイヤーが別個の物理層に配置されている場合でも、横断的関心事のコンポーネントは、すべてのレイヤーからアクセスできるように配置する必要があります。

横断的関心事の機能はさまざまな手法を使用して処理できます。たとえば、Patterns & Practices Enterprise Library などの一般的なライブラリや、メタデータを使用して横断的関心事のコードをコンパイル済みの出力に直接挿入するアスペクト指向プログラミング (AOP) のメソッドがあります。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。

手順 7 - レイヤー間のインターフェイスを定義する

レイヤーのインターフェイスを定義する際の主な目的は、レイヤー間の疎結合を強化することです。つまり、レイヤーでは、他のレイヤーが依存する可能性がある内部の詳細を公開しないようにします。そのため、レイヤーのインターフェイスは、レイヤーに含まれるコンポーネントの詳細を非公開にするパブリック インターフェイスを提供することで、依存関係を最小限に抑えるように設計する必要があります。このように詳細を公開しないことは “抽象化” と呼ばれており、さまざまな方法で実装できます。レイヤーのインターフェイスは、次の設計手法を使用して定義できます。

- **抽象インターフェイス**: これは抽象型基本クラスを定義するか、具象クラスの型定義として機能するコード インターフェイス クラスを定義することで実現できます。この型では、レイヤーのすべてのコンシューマーがレイヤーとの通信に使用する一般的なインターフェイスを定義します。また、抽象インターフェイスを実装したテスト オブジェクト (モック オブジェクトと呼ばれる場合もあります) を使用できるので、この手法を使用するとテスト容易性が向上します。
- **一般的な設計の型**: 多くの設計パターンでは、インターフェイスをさまざまなレイヤーに定義する具体的なオブジェクト型を定義します。このようなオブジェクト型では、レイヤーに関する詳細を公開しない抽象化の機能を提供します。たとえば、Table Data Gateway パターンでは、データベースのテーブルを表し、データと通信するために必要な SQL クエリを実装するオブジェクト型を定義しています。オブジェクトのコンシューマーは、SQL クエリに関する知識はありません。また、オブジェクトがデータベースに接続してコマンドを実行する方法の詳細も知りません。多くの設計パターンは抽象インターフェイスに基づいていますが、具象クラスに基づいているものもあります。Table Data Gateway パターンなど、適切なパターンの大半は、この点について十分な裏付けが行われています。インターフェイスをレイヤーに迅速かつ簡単に実装する場合またはレイヤーのインターフェイスに設計パターンを実装する場合は、一般的な設計の型の使用を検討します。
- **依存関係の逆転**: これは抽象インターフェイスが、レイヤー以外の場所で定義されているか、レイヤーに依存していないプログラミング スタイルです。レイヤーは相互に依存するのではなく、一般的なインターフェイスに依存します。Dependency Injection パターンは、依存関係の逆転の一般的な実装です。コンテナーでは、依存関係の挿入を使用して、他のコンポーネントが依存している可能性があるコンポーネントの配置方法を指定するマッピングを定義します。また、そのコンテナーでは、このような依存関係のあるコンポーネントを自動的に作成して挿入できます。依存関係の逆転の手法では、コードではなく構成によって依存関係を構築するので、柔軟性があり、プラグ可能な設計の実装が可能です。また、具体的なテスト クラスを設計のさまざまなレイヤーに簡単に挿入できるので、テスト容易性が最大限まで高まります。
- **メッセージ ベース**: メソッドを呼び出したり、オブジェクトのプロパティにアクセスしたりすることで、他のレイヤーのコンポーネントと直接通信するのではなく、メッセージ ベースの通信を使用して、インターフェイスを実装しレイヤー間の通信を提供できます。物理境界とプロセス境界をまたぐ通信をサポートするメッセージング ソリューションには、Windows Communication Foundation、Web サービス、Microsoft Message Queuing などがあります。ただし、抽象インターフェイスを、通信のデータ構造の定義に使用する一般的な種類のメッセージと組み合わせることもできます。メッセージ ベースのインターフェイスとの主な違いは、通信のすべての詳細をカプセル化する共通の構造をレイヤー間の通信で使用することです。この構造では、操作、データ スキーマ、エラー コントラクト、セキュリティ情報など、レイヤー間の通信に関連する多くの構造を定義できます。Web アプリケーションを実装して、プレゼンテーション レイヤーとビジネス レイヤ

ー間のインターフェイスを定義している場合、複数の種類のクライアントをサポートしなければならないアプリケーション レイヤーがある場合、または物理境界とプロセス境界をまたぐ通信をサポートする場合は、メッセージ ベースの手法の使用を検討します。また、共通の構造を使用して通信を形式化する場合や、ステータス情報をメッセージで伝達するステートレスなインターフェイスと通信する場合も、メッセージ ベースの手法の使用を検討します。

Web アプリケーションのプレゼンテーション レイヤーとビジネス ロジック レイヤーの間の通信を実装するには、メッセージ ベースのインターフェイスの使用をお勧めします。ビジネス レイヤーで呼び出し間の状態を保持しない場合 (つまり、プレゼンテーション レイヤーとビジネス レイヤーの間の呼び出しがそれぞれ新しいコンテキストを表す場合)、要求と併せてコンテキスト情報を渡し、プレゼンテーション レイヤーで例外とエラー ハンドルに関する一般的なモデルを提供できます。

手順 8 - 配置に関する方針を選択する

多くのソリューションで採用されているアプリケーションの配置構造には、いくつかの一般的なパターンがあります。アプリケーションに最適な配置ソリューションを決定する際には、まず、一般的なパターンを確認します。さまざまなパターンを十分に理解したら、次にシナリオ、要件、およびセキュリティ上の制約を検討して最も適切なパターンを 1 つまたは複数選択します。配置パターンに関する詳細については、第 19 章「物理層と配置」を参照してください。

手順 9 - 通信プロトコルを選択する

設計のレイヤーまたは層の間で行われる通信に使用する物理的なプロトコルは、アプリケーションのパフォーマンス、セキュリティ、および信頼性において重要な役割を果たします。通信プロトコルの選択は、分散配置の検討よりも重要です。コンポーネントが同じ物理層に配置されると、多くの場合、コンポーネント間の直接的な通信をあてにできます。ただし、多くのシナリオで行われるように、コンポーネントとレイヤーを物理的に別のサーバーとクライアント コンピューターに配置する場合、このようなレイヤーのコンポーネント間で、効率的かつ正確に通信する方法を検討する必要があります。通信プロトコルと通信テクノロジの詳細については、第 18 章「通信とメッセージ」を参照してください。