

ソフトウェアアーキテクチャ

SOFTWARE ARCHITECTURE



ソフトウェア開発の流れ（復習）



顧客の要求



要求仕様書



設計書



プログラム



製品

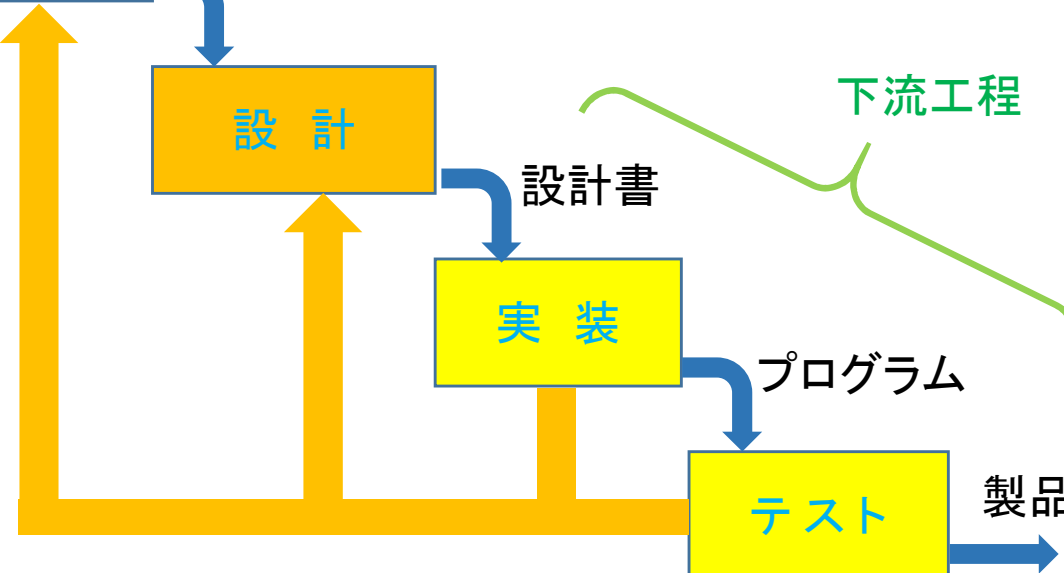


上流工程

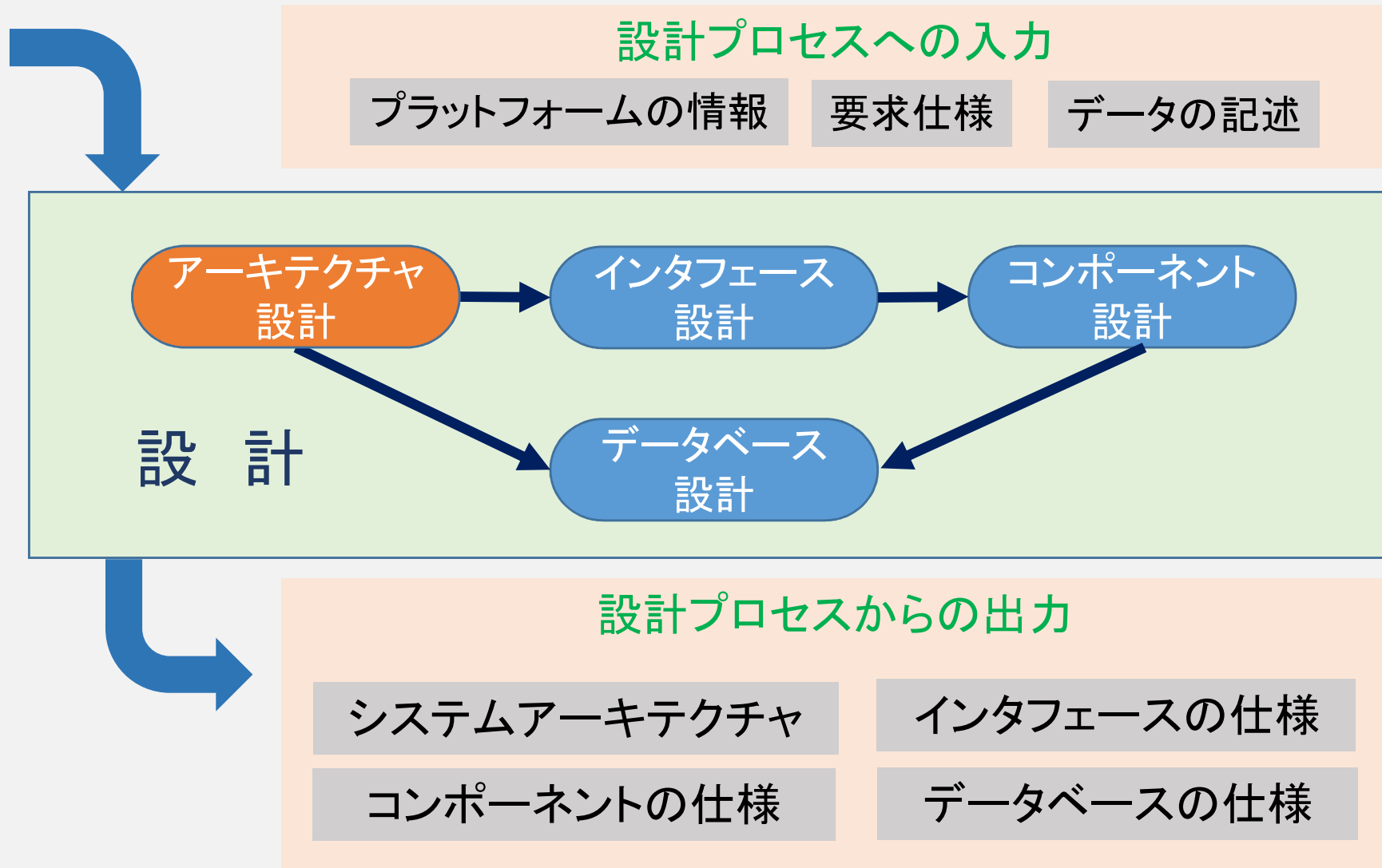
下流工程



プロジェクト管理



ソフトウェア設計のプロセス



アーキテクチャとは

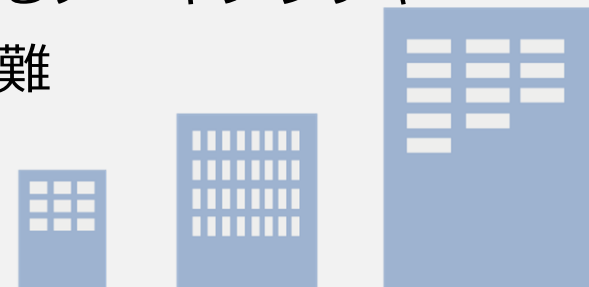
■ アーキテクチャ

Architecture の本来の意味は「建築」

- システムの全体的な構造のこと
- システムの構成要素（コンポーネント=プログラム部品）は何か
- それらが互いにどう関係しているか
- 多くの場合、アーキテクチャは図（ブロック図）で表現される

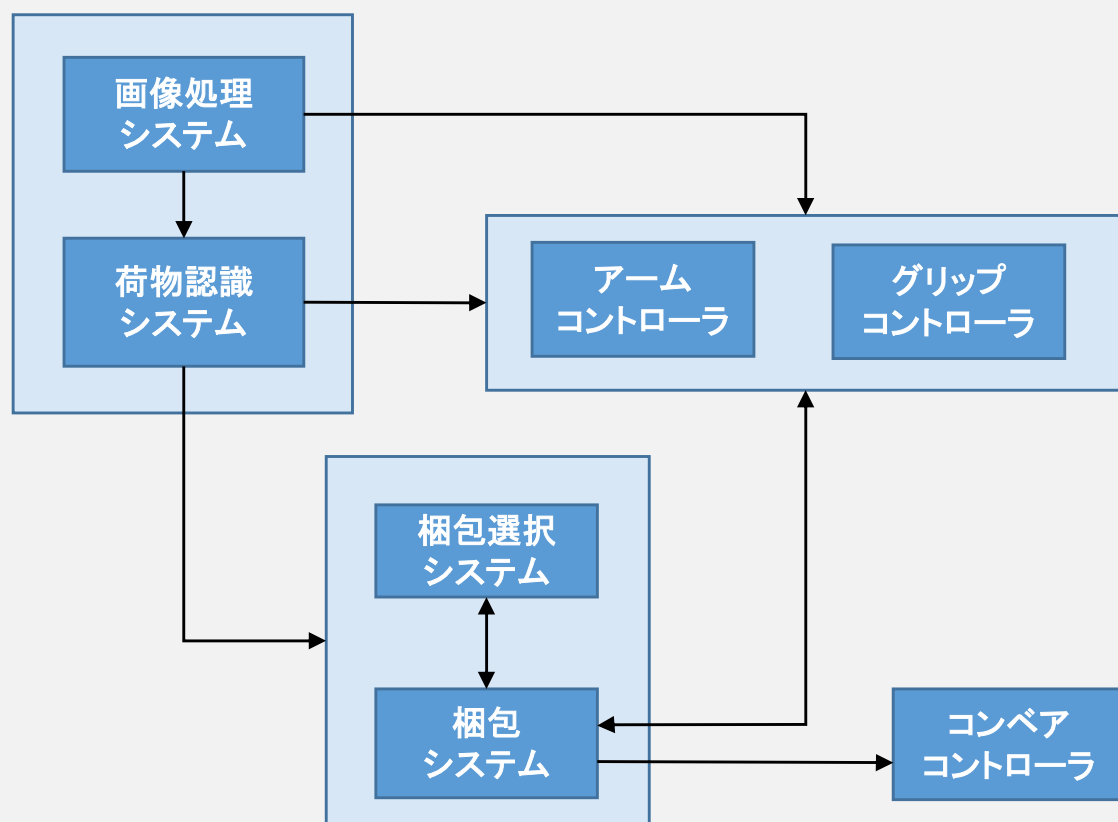
■ アーキテクチャ設計

- ソフトウェア設計のもっとも最初の段階でおこなう
- または要求定義の最終段階でおこなう
- アウトプット=互いに関連するコンポーネントからなるアーキテクチャ
- アーキテクチャをいったん決めるとその後の修正は困難



アーキテクチャの例

【例】荷物梱包ロボット制御システムのアーキテクチャ



1. このシステムは、異なる種類の荷物をロボットにより自動で梱包するためのもの
2. システムは、コンベアに乗って流れてくる荷物の画像からその種類を認識し、適切な梱包方法を選択する
3. システムはコンベアから荷物を取り上げ、梱包し、別のコンベア上に置く

ブロック図の長所と短所

ブロック図は**非形式的**(informal)(非数理的)で、表す内容にあいまい性がある

【長所】 システムの全体像がわかりやすい

- 詳細にまでわされずに、システムの全体像を俯瞰して把握
- 要求や設計について、**ステークホルダー**(stake holder) (関係者) との議論・調整に有用
- 管理者は、開発すべきコンポーネントを認識でき、**作業分担**と**人員計画**の策定に着手

【短所】 システムを厳密には理解できない

- コンポーネントの詳細や相互の関連やインタフェースの詳細が不明
- 今後の詳細設計や実装の基礎として利用



アーキテクチャ設計で考えるべきこと

1. テンプレート(template)となる一般的な(generic)アーキテクチャがあるか？
2. 多数のコアやプロセッサにわたり, どうシステムを分散(distribute)させるか？
3. システムをどのように分解(decompose)し, 構造化(structure)するか？
4. コンポーネントの操作(operation)をどのように制御(control)するか？
5. 非機能的な要求(non-functional requirements)をどのように考慮するか？
6. アーキテクチャ設計をどのように評価(evaluate)するか？
7. アーキテクチャを説明するドキュメント(document)をどのように作成するか？



非機能的要求の考慮

【機能的要求】 (functional requirements) 入力と出力の関係を規定する要求

【非機能的要求】 それ以外の要求

1. 性能(performance) : 処理スピードが速い
→性能上重要な機能は少数の要素内に局所化し, 分散させない
2. セキュリティ(security) : 悪意ある人やシステムから守る
→システムを階層化し, 重要な情報は外部から遠い最内層で保護
3. 安全性(safety) : 誤動作で人の健康や財産を損なわない
→重要機能は少数個の要素に配備し, 異常時にはシャットダウン
4. 可用性(availability) : システムダウンしにくい
→同一機能の要素を複数個配備した冗長システムにより信頼性向上
5. 保守性(maintainability) : システムを改訂しやすい
→独立性の高い小さな要素から構成し, 保守時には要素ごと交換



アーキテクチャパターン

代表的なアーキテクチャを抽象化し、再利用できるようにカタログ化したもの

【例】

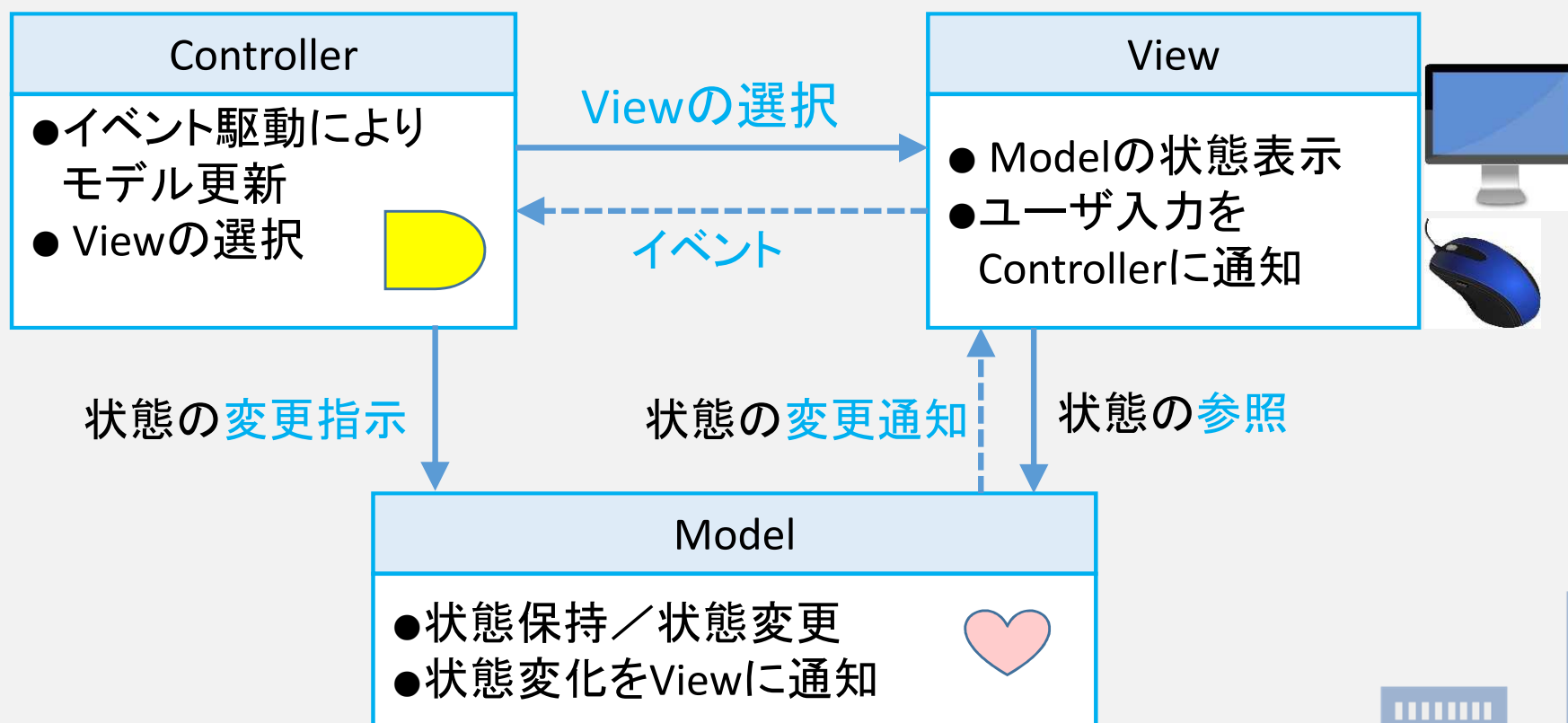
1. MVCアーキテクチャ
2. 階層アーキテクチャ
3. リポジトリアーキテクチャ
4. クライアントサーバアーキテクチャ
5. データフローアーキテクチャ

今回すべてを紹介



MVC アーキテクチャ(1/3)

【一般的なMVCアーキテクチャ】

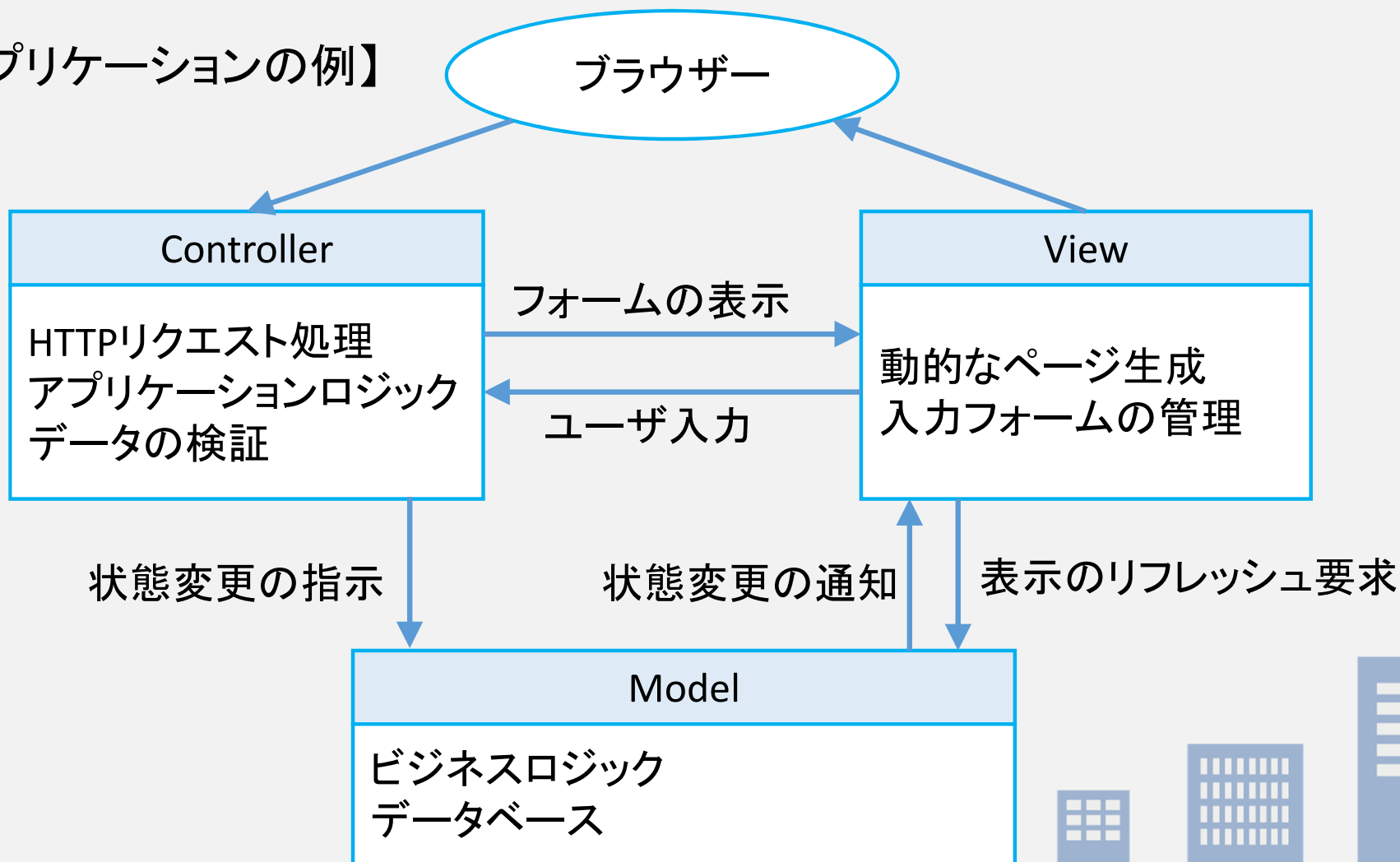


MVC アーキテクチャ(2/3)

名前	概要
概要	<p>MVC (Model-View-Controller)</p> <ul style="list-style-type: none">- システムの状態を表すデータの管理部とユーザとの相互作用(表示／入力などのインタラクション)の管理部を分離- システムは互いに関連するつぎの3つのコンポーネントからなる<ol style="list-style-type: none">(1) Model: データの保持・変更の管理(2) View: データの表示／入力の方法の管理(3) Controller: ユーザとの相互作用の管理, ModelやViewの制御
システム例	<ul style="list-style-type: none">- Webアプリケーション
使用する時	<ul style="list-style-type: none">- データの表示／入力の方法が複数あり, 適宜切り替えたいとき- データの表示／入力の方法の将来の変更を容易にしたいとき
長所	<ul style="list-style-type: none">- データの表示／入力の具体的方法と独立にデータを管理可能- データと表示の一貫性を保証: データを変更すると, 自動的に表示が変更
短所	<ul style="list-style-type: none">- モデルや相互作用が単純なとき, 余分なコードとなる

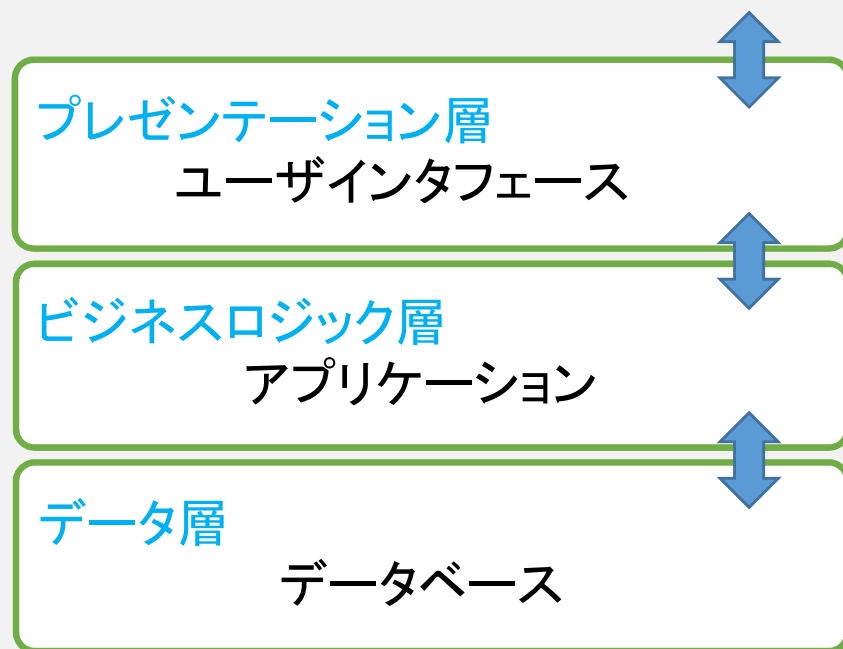
MVC アーキテクチャ (3/3)

【Webアプリケーションの例】

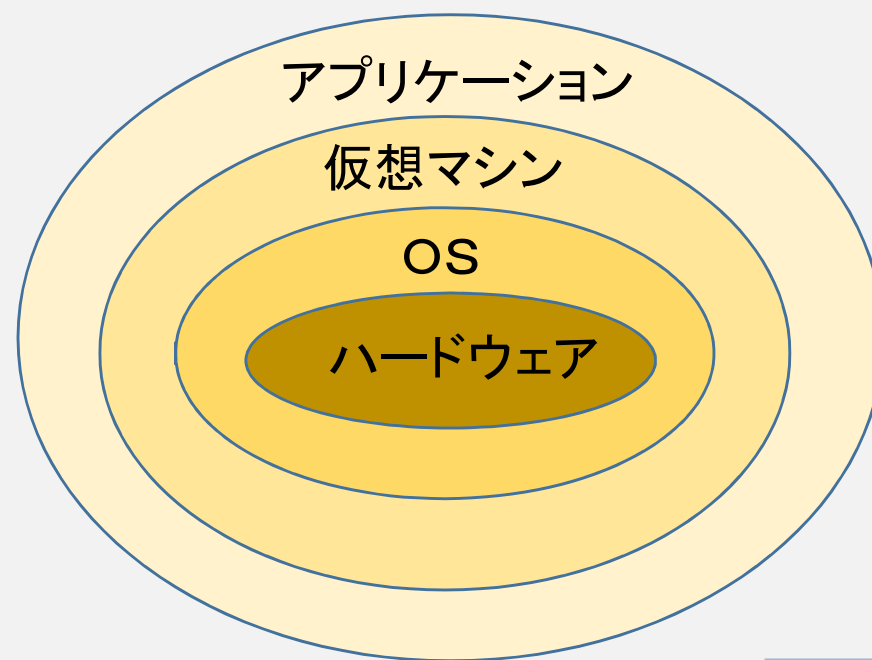


階層アーキテクチャ(1/3)

【三層モデル】



【仮想マシンモデル】



階層アーキテクチャ(2/3)

名前	Layered architecture
概要	<ul style="list-style-type: none">- システムを互いに関連する機能をもつ複数の層(layer)に分離- 層には上下関係があり, 各層はそのすぐ上の層にサービスを提供- 最下層はシステム全体で使われる核(core)となるサービスを担当
システム例	<ul style="list-style-type: none">- 図書館ネットワークシステム
使用する時	<ul style="list-style-type: none">- 既存システムの上に新機能を構築するとき- 開発を複数のチームでおこない, 層ごとに開発分担をおこなうとき- 複数の層からなるセキュリティの要求があるとき
長所	<ul style="list-style-type: none">- 層の間のインタフェースを維持する限り, 層を置換可能- 層のインタフェースを変えても, 影響を与える範囲は隣接した層のみ- マルチプラットフォームへの実装が容易(最下層のみ再実装)- 各層に認証のような冗長な機能の追加が容易
短所	<ul style="list-style-type: none">- 複数の層にきれいに分離することは困難なことも- サービス要求が各層を通過して最下層まで届くので, 性能が問題

階層アーキテクチャ(3/3)

【図書館ネットワークシステムの例】

Webブラウザインタフェース

ログイン フォーム・クエリ管理 印刷管理

分散検索 文書検索 権利管理 課金処理

図書索引

DB1

DB2

DB3

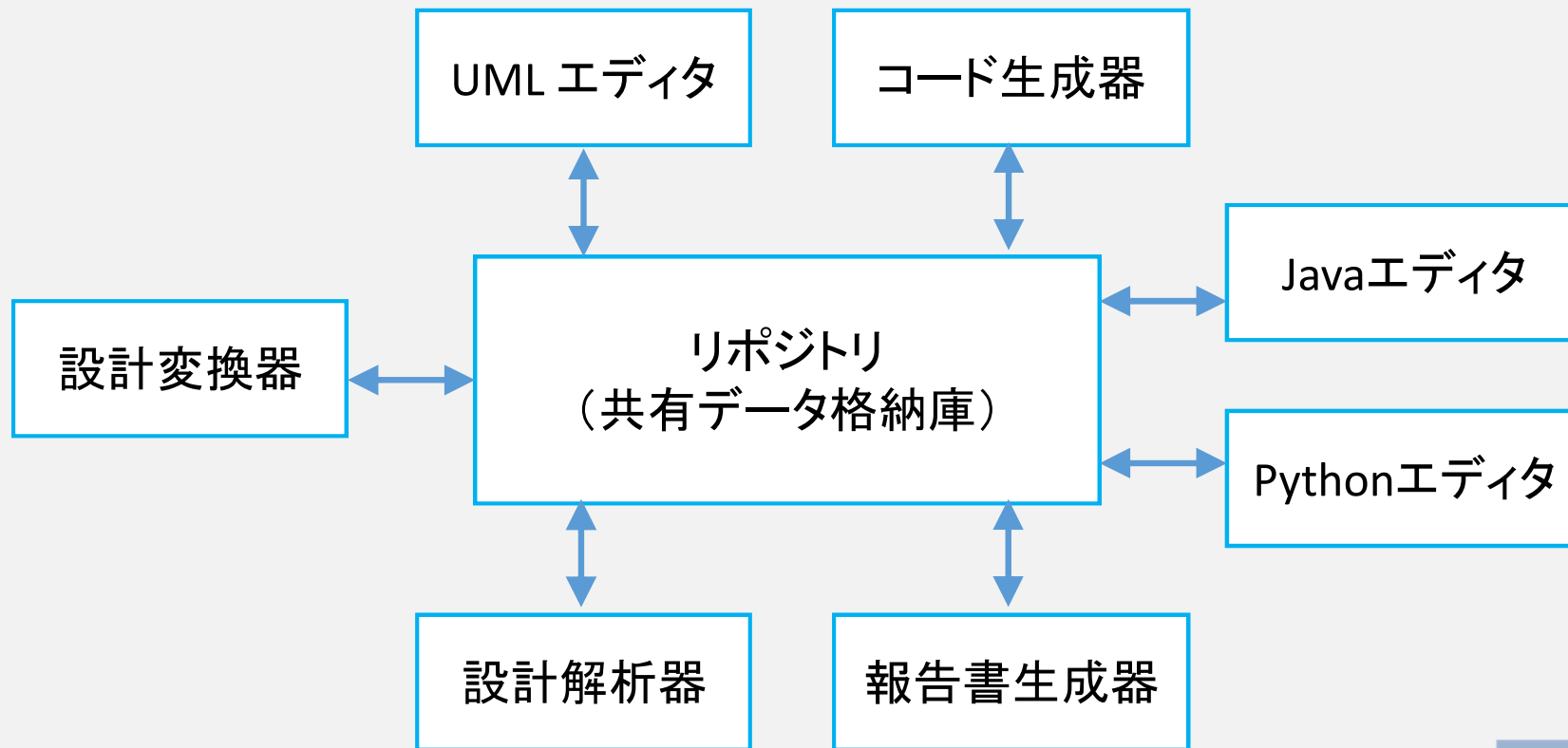
DB4

DBn



リポジトリ アーキテクチャ(1/3)

【ソフトウェア統合開発環境】

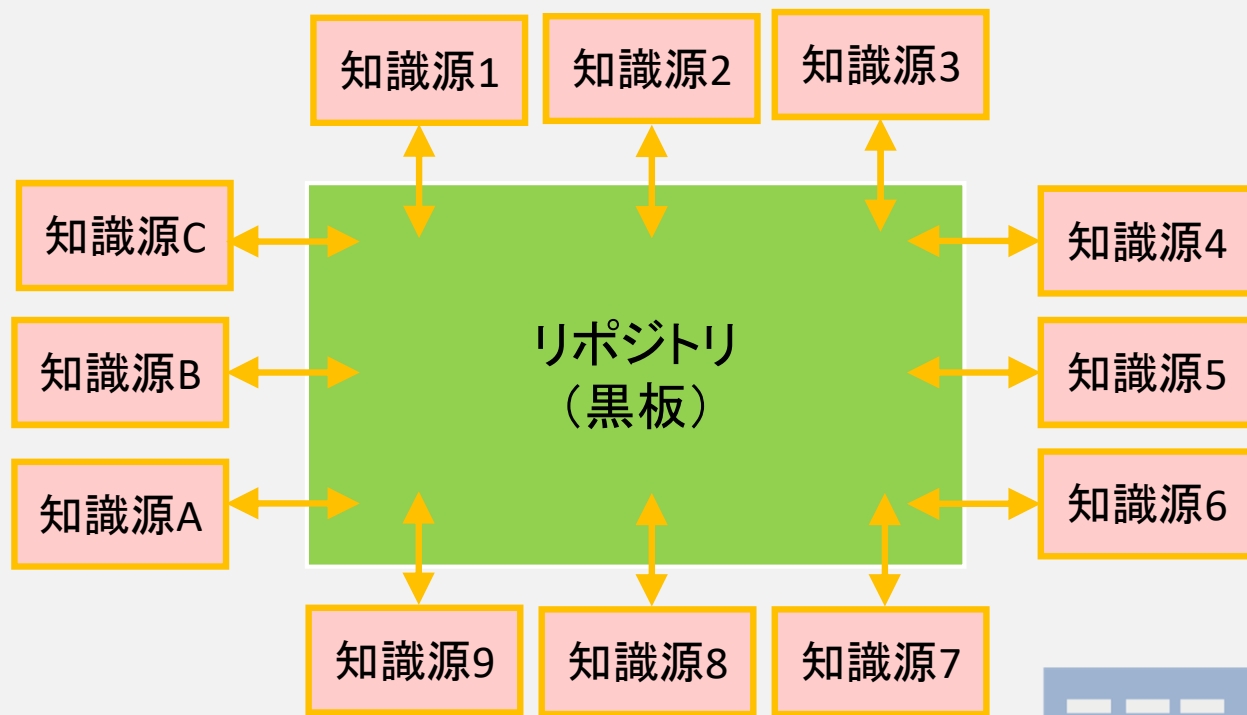
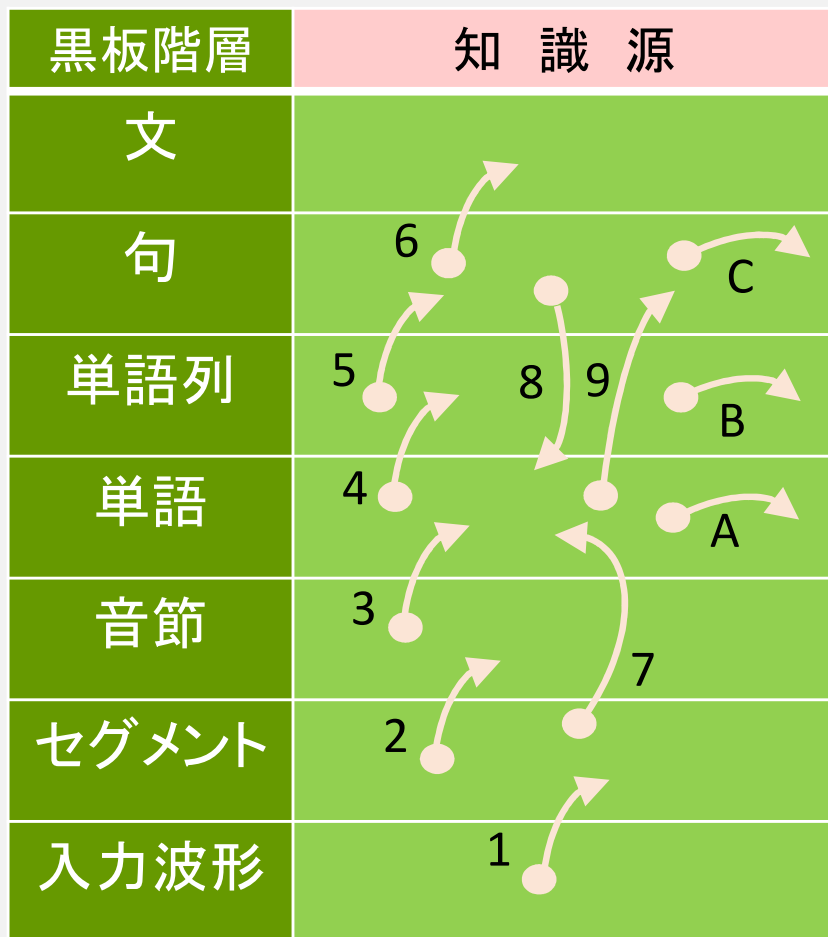


リポジトリ アーキテクチャ(2/3)

名前	Repository
概要	<ul style="list-style-type: none">- システムの全データをリポジトリ(共有データ格納庫)が管理- コンポーネントはリポジトリに直接アクセスできるが、コンポーネントどうしは直接の相互作用はない
システム例	<ul style="list-style-type: none">- ソフトウェア統合開発環境(IDE: Integrated Development Environment)
使用する時	<ul style="list-style-type: none">- 大量のデータが生成され、それを長期間保持する必要があるとき- リポジトリへのデータの書き込みを引き金(trigger)としてアクションやツールが起動するデータ駆動(data-driven)のシステムにしたいとき
長所	<ul style="list-style-type: none">- コンポーネントは互いに独立(互いの存在を知らなくてよい)- あるコンポーネントによるデータの変更を全コンポーネントに伝播可能- 全データが1カ所に集中するのでデータの一貫性があり、バックアップも容易
短所	<ul style="list-style-type: none">- レポジトリは単一障害点(Single Point of Failure): この単一箇所が働かないと、システム全体が障害となる- コンポーネント間通信のすべてをリポジトリ経由にするのは非効率- リポジトリを複数のコンピュータに分散配置するのは困難

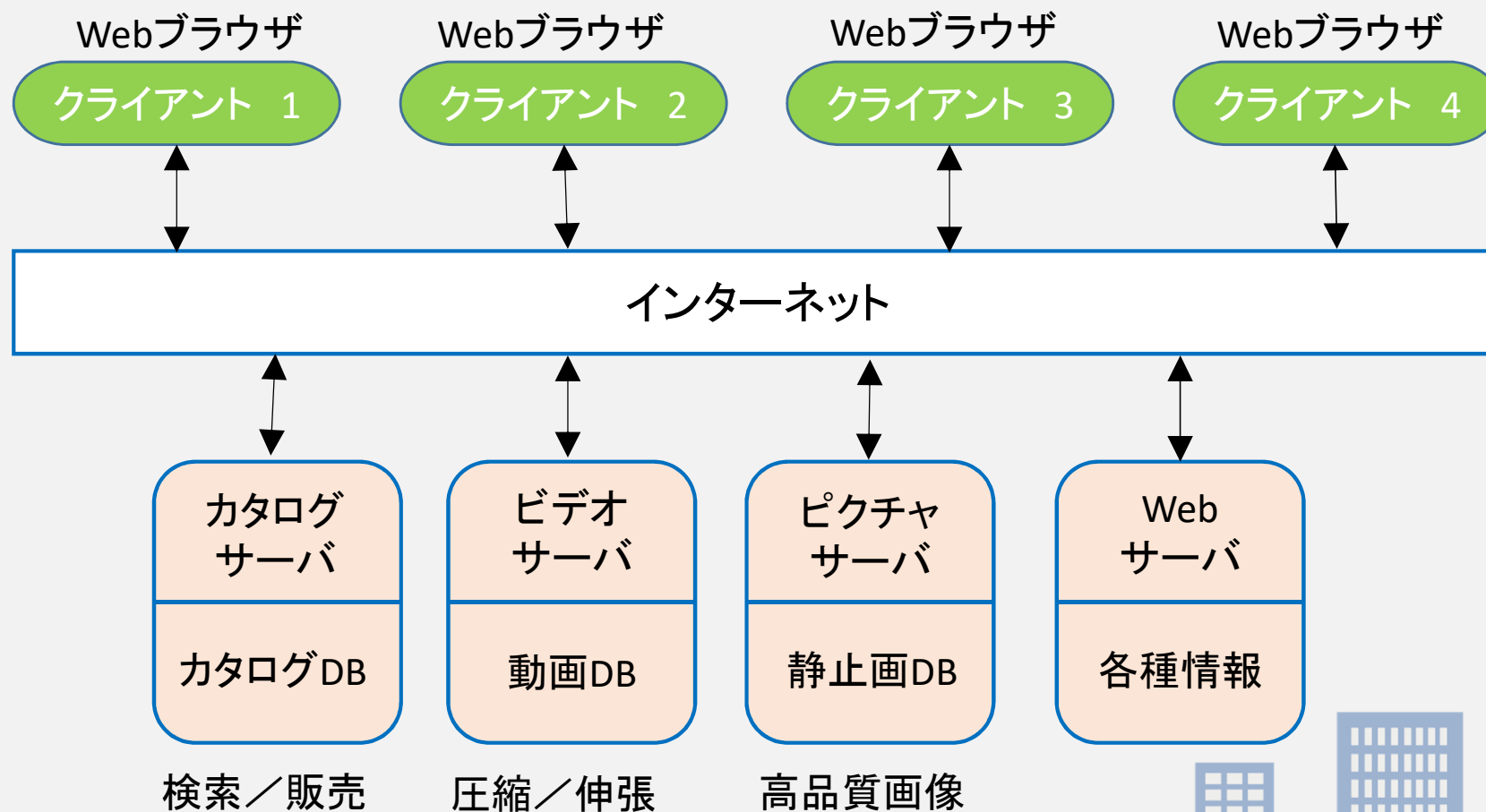
リポジトリ アーキテクチャ(3/3)

【黒板モデル】 音声認識システム



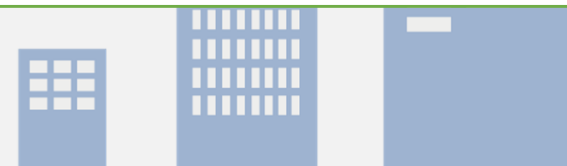
クライアントサーバアーキテクチャ(1/2)

【動画ライブラリ】



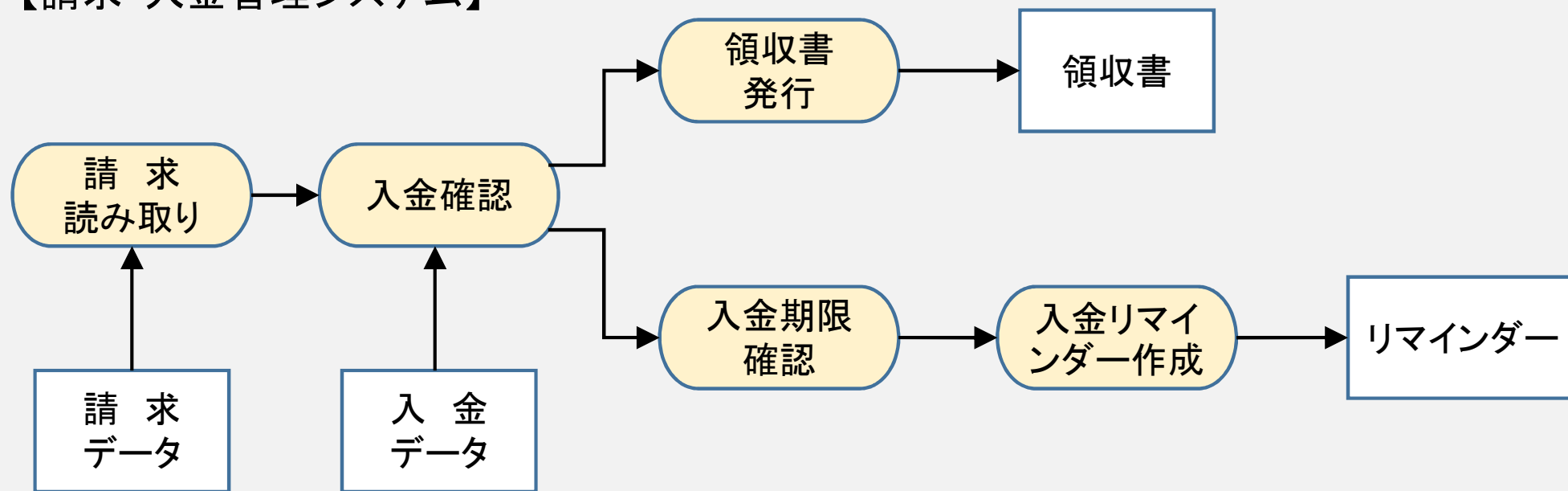
クライアントサーバアーキテクチャ(2/2)

名前	Client-server
概要	<ul style="list-style-type: none">- 複数のサーバが、システムの機能をサービスとして提供- 複数のクライアントが、サーバにアクセスしてこれらのサービスを利用
システム例	<ul style="list-style-type: none">- 動画ライブラリ
使用する時	<ul style="list-style-type: none">- 共有データベース内のデータが、多数の場所からアクセスされるとき- システムの負荷が可変で、必要に応じてサーバを複製したいとき
長所	<ul style="list-style-type: none">- クライアントとサーバをネットワークにわたって分散可能- 各サービスを1サーバのみで提供すれば、全クライアントが利用可- コンポーネントは互いに独立(他のコンポーネントに影響を与えずにサーバを置換/修正可能)
短所	<ul style="list-style-type: none">- サーバは単一障害点なので、DoS (Denial of Service)攻撃や故障に弱い- 性能は、ネットワークにも依存するので予測しにくい- サーバが異なる複数の組織に所有されるときは管理上の問題がある



データフロー アーキテクチャ(1/2)

【請求・入金管理システム】



データフロー アーキテクチャ(2/2)

名前	Dataflow (Pipe and filter)
概要	<ul style="list-style-type: none">- 各コンポーネント(フィルタ)は, 独立してある種のデータ変換を実行- 1つのコンポーネントから別のコンポーネントに(パイプのように)データが流れる
システム例	<ul style="list-style-type: none">- 請求・入金管理システム
使用する時	<ul style="list-style-type: none">- ビジネスのデータ処理(バッチ処理やトランザクション処理)において, 入力を幾つかの段階を経て処理し, 出力を生成するとき
長所	<ul style="list-style-type: none">- 理解が容易で, データ変換コンポーネントは再利用可能- ワークフローの形式が, 多くのビジネスプロセスの構造に良く適合- 新規のデータ変換コンポーネントを追加したソフトウェア進化が容易- 逐次システムとしても並行システムとしても実装可能
短所	<ul style="list-style-type: none">- データ変換間のデータの書式の統一／変換が必要- そのためシステムオーバヘッドが増加したり, 互換性のない書式をもつデータ変換機能の再利用が困難になり得る- GUIなどをもつインタラクティブシステムには向かない

演習問題 12

あなたが将来設計したいと思うソフトウェアシステムを1つ想定し、そのアーキテクチャの概要を設計しなさい。
(全体として2ページ以内で記述すること。)

つぎのことについて記述すること。

- 1) そのソフトウェアシステムの機能の要点
- 2) アーキテクチャのブロック図
- 3) 各コンポーネントの機能の要点

