再設計戦略サマリー

1. 全体アーキテクチャ方針

1-1. 設計思想

- 疎結合 **(Loose Coupling)**: 各機能は独立性を高め、交換やテストを容易にする。
- 関心の分離 **(Separation of Concerns)**: 1クラス1役割を徹底し、コードの見通しを良くする。

1-2. ネットワーク戦略 (Netcode for GameObjects)

- サーバー権威モデル: 全てのゲームロジックと状態決定はサーバー（専用サーバー or ホスト）が行う。クライアントは入力を送信し、サーバーから同期された状態を描画することに専念する。
- シングル/マルチのコード共通化: シングルプレイは「参加者が一人だけのHostモード」として実装する。これにより、NetworkBehaviourを用いたコードを完全に共通化し、開発効率とメンテナンス性を最大化する。（シングルプレイのためのポート開放は不要）

1-3. 依存関係の管理

- **DI (Dependency Injection)** パターンの導入: GameManager.Instanceのようなシングルトンへの直接依存を避け、必要な機能をインターフェースとして外部から注入する方式へ段階的に移行する。これにより、疎結合を促進する。

2. 機能別 設計方針

2-1. プレイヤー機能

- 巨大なPlayerControllerを、以下の4つのクラスに責務分割する。
  - **PlayerFacade**: NetworkBehaviourを継承する司令塔。ネットワーク通信と各コンポーネントへの指示出しに専念。
  - **PlayerInput**: MonoBehaviour。IsOwnerの時だけFacadeによって有効化され、入力検知とイベント発行に専念。ネットワークを意識しない。
  - **PlayerStateMachine**: プレイヤーの状態（通常、移動中、タイピング中など）をステートパターンで管理。
  - **PlayerView**: アニメーションやエフェクトなど、見た目の表現に専念。

2-2. 入力システム

- **Unity新Input System**への移行: キーコンフィグ機能の実装を視野に入れ、抽象化された「アクション」ベースの入力管理を行う。
- **Action Maps**の活用: ゲームの状況（Gameplay, Typingなど）に応じて有効なアクションマップを切り替え、入力のコンフリクトを防ぐ。
- タイピング入力: キーボードの文字入力はKeyboard.current.onTextInputイベントで取得し、アクション入力と共存させる。

2-3. タイピング機能

- データ駆動設計への移行:
  - ローマ字変換ロジック: 巨大なswitch文を廃止し、「かな」と「ローマ字パターン」をマッピングする外部データ（ScriptableObjectやCSV/JSON）を参照する方式に変更する。
  - 問題文データ: level（難易度）の概念を追加し、より柔軟な問題選出ロジックを可能にする。

2-4. アイテム機能

- データと振る舞いの完全分離:
  - **ItemData (ScriptableObject)**: アイテムの名前やアイコンといった「データ」を保持。
  - **IItemEffect (**ストラテジーパターン**)**: アイテムの「効果」そのものをScriptableObjectとして実装し、ItemDataから参照する。これにより、プログラマー以外でもアイテムの効果を組み合わせたり、調整したりすることが可能になる。

3. フォルダ構造

- 機能別（**Feature-based**）アプローチを全面的に採用する。プレイヤー関連のアセットは全てFeatures/Player/に、アイテム関連はFeatures/Items/に集約する。
- 特にアイテムは、Features/Items/Bomb/のようにアイテムごとにフォルダを作成し、関連アセット（データ、アイコン、スクリプト等）をまとめることで、高い凝集度と拡張性を実現する。